Model-Checking zur Analyse von Message Sequence Charts über Statecharts[1]

E. Canver

Zusammenfassung

Die *Unified Modeling Language* (UML) enthält sowohl *Statecharts* als auch mit *Sequence Diagrams* eine Variante von *Message Sequence Charts* (MSCs). Da beide eingesetzt werden können, um verschiedene Aspekte eines Systems zu beschreiben, ist es sinnvoll, die Konsistenz zwischen beiden Beschreibungstechniken zu prüfen. Im vorliegenden Bericht werden beide Ansätze miteinander notationell und semantisch integriert und eine formale Konsistenzbeziehung formuliert. Zu diesem Zweck werden hier die MSC-Notation zu MSC_{CTL} erweitert, die Semantik von Statecharts in Termini von Transitionssystemen beschrieben und in MSC_{CTL} formulierte Anforderungen nach CTL* abgebildet. Die eigentliche Analyse erfolgt unter Einsatz eines Model-Checkers. Dazu sind hier Abbildungsvorschriften für einen speziellen Model-Checker angegeben. Aufgrund der mit der Prüfung von CTL*-Formeln verbundenen Komplexität ist hier zudem ein Verfahren angegeben, um die Beweisverpflichtungen in eine einfacher zu handhabende Form zu umzusetzen.

[1] Die Arbeiten wurden zum Teil von der Deutschen Forschungsgemeinschaft im Rahmen des Schwerpunktprogramms (1064) "Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen" gefördert

Inhaltsverzeichnis

Kapitel 1

Einleitung

Statecharts [Har87] werden häufig eingesetzt um Syteme und ihr Verhalten zu beschreiben. Systeme werden zustandsbasiert modelliert und das Verhalten wird mittels Zustandsübergängen dargestellt. Um komplexe Systeme zu modellieren kann man Zustände hierarchisch zerlegen und gelangt somit zu feineren Strukturen; ein wesentlicher Vorzug von Statecharts ist ihre graphische Notation. STATEMATE [HLN+90] ist eine Implementierung von Statecharts und kann als Referenzwerkzeug betrachtet werden, das den Standard für die Semantik definiert.

Message Sequence Charts (MSC) [RGG96] stellen eine Beschreibungstechnik zur graphischen Darstellung von Szenarien aus der Interprozess-Kommunikation dar. Dabei interessiert man sich insbesondere für die Anordnung bestimmter Ereignisse während der Interaktion. Neben der graphischen Notation gibt es auch eine textuelle Sprache zu der eine formale Semantik im Stil einer Prozess-Algebra definiert wurde [MR94].

Im Softwareentwicklungsprozess werden häufig beide Beschreibungsmittel gleichzeitig eingesetzt. Beispielsweise werden Statecharts und *Sequence Diagrams*, deren Notation und Bedeutung direkt von MSCs abgeleitet ist, von der *"Unified Modeling Language" (UML)* [UML97] unterstützt. Um frühzeitig Fehler in der Softwareentwicklung aufdecken zu können, sollte die Konsistenz zwischen beiden Beschreibungsarten analysiert werden. Dazu ist es notwendig, beide Beschreibungstechniken auf eine gemeinsame semantische Basis abzubilden. Dabei kann man feststellen, dass nicht festgelegt ist, was ein MSC in Relation zu einer Systemspezifikation bedeuten soll: beschreiben MSCs sämtliche Ausführungspfade eines Systems oder nur einen Teil daraus? Die Verwendung von MSCs zur Beschreibung von Szenarien legt nahe, dass es sich dabei um nur einen Teil der Ausführungspfade handelt. Allerdings werden solche Szenarien im Verlauf einer Software-Entwicklung oftmals ergänzt mit dem Ziel, das Verhalten des Systems vollständig zu beschreiben. Diese Lücke in der Definition von MSCs wurde von Damm und Harel erkannt und als Antwort wurden "Live Sequence Charts" (LSC) [DH98] entwickelt. Sie definieren unter welchen Bedingungen ein Ausführungspfad eines Systems ein LSC erfüllt. Wenn Szenarien betrachtet werden, untersucht man, ob die von einem LSC definierte Menge von Ausführungspfaden eine Teilmenge der von der Systemspezifikation definierten Menge von Ausführungspfaden ist. Will man die Vollständigkeit nachweisen, so muss die umgekehrte Inklusion gezeigt werden.

In unserem Ansatz, MSC_{CTL}, werden MSCs in einer erweiterten Syntax eingebettet. Man kann die zu betrachtetenden Pfade durch Quantifikation mit den temporallogischen Operatoren von CTL* [Eme90] festlegen. Im Gegensatz zu LSCs können in unserem Ansatz MSCs miteinander logisch ($\neg$, $\wedge$, $\vee$) verknüpft werden; dadurch kann man z.B. Alternativen und unerwünschte Ausführungspfade charakteisieren. Wenn man eine Systembeschreibung in Statcharts geeignet als

Zustandsübergangssystem darstellt, kann man darüber hinaus einen Model-Checker verwenden, um die Konsistenz zwischen der Systembeschreibung in Statecharts und in MSC_{CTL} zu analysieren.

Wir geben zunächst kurze Überblicke zu Statecharts und MSCs und beschreiben informell unseren Ansatz, beide Beschreibungstechniken miteinander zu integrieren; dazu definieren wir MSC_{CTL} als syntaktische Erweiterung von MSCs. Im folgenden wird dieser Ansatz formalisiert indem wir eine Umsetzung von Statecharts in ein geeignetes Transitionssystem beschrieben und die Verknüpfung die Konsistenzbedingung zwischen diesem Transitionssystem und MSC_{CTL} formal definieren. Auf dieser Basis wird eine Umsetzung von Statecharts und MSC_{CTL} in die Sprache eines geeigneten Model-Checkers entwickelt; damit steht eine Methodik bereit, um die Konsistenz zwischen beiden Darstellungen werkzeugunterstützt zu analysieren. Den Ansatz führen wir exemplarisch an einem Beispiel aus dem Eisenbahnwesen durch.

Kapitel 2

Statecharts und MSCs

Statecharts und MSCs werden eingesetzt um Systeme und ihr Verhalten zu beschreiben. Liegt eine Systembeschreibung in beiden Notationen vor, ist es wünschenswert, die Konsistenz zwischen beiden Beschreibungen prüfen zu können. Allerdings ist eine Konsistenzbedingung zwischen diesen Beschreibungsarten noch nicht festgelegt. In diesem Kapitel geben wir zunächst einen Überblick zu Statecharts und zu MSCs. Anschliessend beschreiben wir unseren Ansatz um eine Beziehung zwischen beiden Notationen herzustellen.

2.1 Statecharts

Statecharts dienen der Beschreibung von Zustandsübergangssystemen. In UML werden sie eingesetzt um das Verhalten von Teilsystemen bzw. von Objekten zu beschreiben. Statecharts können hierarchisch strukturiert werden, d.h. sie können aus weiteren Statecharts aufgebaut sein (AND-Statechart, OR-Statechart) oder auch einfach (nicht weiter strukturiert) auftreten. Ist ein Statechart P in einem anderen Statechart Q enthalten, sagen wir P ist ein Subchart von Q. AND-Strukturen werden zur Beschreibung von zueinander nebenläufig auftretenden Aktivitäten eingesetzt. OR-Strukturen beschreiben Aktivitäten, die nicht gemeinsam auftreten können; Transitionen zwischen den Subcharts eines OR-Statechart beschreiben Zustandsübergänge, d.h. der Statechart von dem die Transition ausgeht wird dabei inaktiv und der Statechart zu dem die Transition führt wird aktiviert.

Die grafische Notation von Statecharts ist gut geeignet um die hierarchische Struktur zu visualisieren. Auf der anderen Seite erleichtert eine textuelle Darstellung den Zugang zu einer formalen Behandlung, die wir benötigen, um den Zusammenhang zwischen Statecharts und MSCs genau festzulegen. Daher haben wir zunächst eine einfache Syntax definiert um Statecharts zu beschreiben; siehe Abbildung 2.1.

Eine Statechart Beschreibung muss bestimmte strukturelle Anforderungen erfüllen, damit sie gültig ist. Diese Anforderungen hängen mit dem Begriff der Aktivität eines Statechart zusammen: Ein Statechart kann aktiv oder inaktiv sein. Ein einfacher Statechart kann sich somit in einem aktiven oder inaktiven Zustand befinden. Komplexer ist die Lage, wenn man strukturierte Statecharts betrachtet. Wenn ein AND-Statechart aktiv ist, sind alle seine direkten Subcharts aktiv. Ansonsten sind alle seine Subcharts inaktiv. Andere Konfigurationen sind nicht zulässig. Entsprechend ist in einem aktiven OR-Statechart genau einer seiner direkten Subcharts aktiv. Wenn ein OR-Statechart inaktiv ist, ist keiner seiner Subcharts aktiv. Auch hierbei sind andere Konfigurationen nicht zulässig. Zulässige Konfigurationen für einen aktiven Statechart beschreiben somit, welche seiner Subcharts (auch über

```
<sc spec>   ::=  <sc def>
<sc def>    ::=  <basic sc> | <and sc> | <or sc>
<basic sc>  ::=  sc <id> :  basic
<and sc>    ::=  sc <id> :  and <sc list> endsc <id>
<or sc>     ::=  sc <id> :  or <sc list> <init> <trans> endsc <id>
<sc list>   ::=  <sc def> | <sc def> <sc list>
<init>      ::=  init :  <stmts> [ <ids> ]
<trans>     ::=  <> | <tr> <trans>
<tr>        ::=  tr <label> [ <sources> -> <targets> ]
<label>     ::=  <id> :  <stmts>
<sources>   ::=  <ids>
<targets>   ::=  <target> | <target> , <targets>
<target>    ::=  <id> | h(<id>) | h*(<id>)
<ids>       ::=  <id> | <id> , <ids>
```

Abbildung 2.1: Syntax für Statecharts

mehrere Strukturebenen hinweg) gemeinsam aktiv sein können bzw. müssen.

Eine Transition kann mehrere Ausgangs- und mehrere Zielpunkte haben. Die Menge der Ausgangspunkte und die Menge der Zielpunkte müssen jeweils zulässigen Konfigurationen des Statechart bilden, zu dem die Transition gehört; nur wenn die Statecharts aus der Menge der Ausgangspunkte aktiv sind, kann die Transition eintreten und wenn sie eintritt werden die Statecharts aus der Menge der Zielpunkte aktiviert. Eine Transition kann optional mit Anweisungen versehen sein, in der Ereignisse und Zustandsvariablen referenziert werden können. Wenn eine Transition eintritt, werden die zugehörigen Anweisungen ausgeführt. Eine Anweisung besteht aus einer optionalen Bedingung und einer optionalen Aktion. Damit eine Transition eintreten kann, müssen auch die Bedingungen an allen ihren Anweisungen erfüllt sind. Die Aktionen modifizieren Variablen indem sie ihnen Werte zuweisen oder erzeugen Ereignisse. Die Syntax für die Deklaration von Ereignissen und Variablen sowie für Anweisungen ist in Anhang A beschrieben.

Für einen OR-Statechart wird festgelegt, welche der in ihm enthaltenen Statecharts initial aktiv sein sollen (Initialkonfiguration); diese müssen eine zulässige Konfiguration bilden. Wenn ein OR-Statechart aktiviert wird, werden seine Subcharts entsprechend dieser Initialkonfiguration aktiviert (initialisiert). Zusammen mit der Initialkonfiguration kann man auch Anweisungen festlegen, die bei der Iniatialisierung ausgeführt werden sollen; sie werden analog zu den Anweisungen an den Transitionen behandelt.

Allerdings kann man Transitionen zu einem Statechart auch so definieren, dass bei Eintritt der Transition andere Subcharts als die aus der Initialkonfiguration aktiviert werden, indem man diese Subcharts explizit ausweist. Daneben können auch spezielle Konnektoren auftreten, die dazu dienen in dem Zustand wieder aufzusetzen, in dem ein Statechart verlassen wurde. Ein einfacher *history connector*, der einem Statechart Q zugeordnet ist (`h(`Q`)`), dient dazu, sich zu merken, welcher der Subcharts von Q zuletzt aktiv war. Wird Q über diesen Konnektor wieder aktiviert, wird dabei auch der zuletzt aktive Subchart mit seiner Initialkonfiguration aktiviert. Zudem kann man einem Statechart Q einen sogenannten *deep history connector* zuordnen (`h*(`Q`)`). Mit diesem merkt man sich nicht nur auf der Ebene von Q, welcher direkte Subchart aktiv war, sondern auch welche der in diesem Subchart enthaltenen Statecharts aktiv waren, d.h. man merkt sich die komplette Konfiguration. Wird ein Statechart über diesen Konnektor aktiviert, so befindet er sich in genau der Konfiguration, in der er zuletzt aktiv war.

```
sc Z: or
  sc Y: and
    sc A: or
      sc B: basic
      sc C: basic
      init: [B]
      tr e: [B -> C]
      tr f: [C -> B]
    endsc A
    sc D: or
      sc E: basic
      sc F: basic
      sc G: basic
      init: [F]
      tr k: [E -> F]
      tr g: [E -> G]
      tr i: [F -> G]
    endsc D
  endsc Y
  sc H: basic
  sc I: basic
  init: [Y]
  tr l: [I -> A,h(D)]
  tr m: [H -> C,D]
  tr n: [C,E -> H]
  tr o: [I -> C,F]
  tr p: [Y -> I]
endsc Z
```

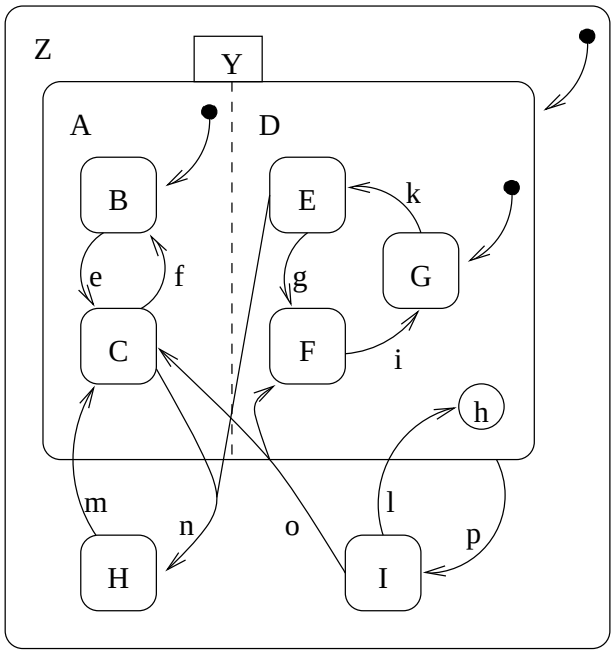

Abbildung 2.2: Ein Statechart Beispiel: textuelle und graphische Darstellung

Beispiel: In Abbildung 2.2 ist der gleiche Statechart textuell und graphisch dargestellt: `Z` ist ein OR-Statechart, der aus den drei Statecharts `Y`, `H` und `I` aufgebaut ist. `Y` selber ist ein AND-Statechart aus den OR-Statecharts `A` und `D`. `B` und `C` sind Subcharts von `A`; wenn `A` aktiv wird ist initial `B` aktiv. Mit den Transitionen `e` und `f` kann die Aktivität zwischen `B` und `C` wechseln. Wenn `Z` aktiviert wird, wird zunächst sein Subchart `Y` und damit auch `A`, `B`, `D` und `G` aktiviert. Transition `n` kann eintreten, wenn `C` und `E` aktiv sind und danach ist `H` aktiv. `o` kann eintreten wenn `I` aktiv ist und danach sind `C` und `F` aktiv. Transition `m` führt von `H` nach `C`. Allerdings ist `C` allein keine zulässige Konfiguration für `Y`; dazu müsste die Menge der Zielpunkte von `m` um eine zulässige Konfiguration von `D` ergänzt werden. In solchen Fällen wird implizit eine Ergänzung zu einer Konfiguration auf einer möglichst hohen Ebene vorgenommen – in diesem Fall zu {`C`,`D`}. Analog wird in Fällen verfahren, in denen die Menge der Ausgangspunkte einer Transition zu einer zulässigen Konfiguration ergänzt werden müssen. Transition `l` verbindet mit einem Konnektor `h(D)`. Wenn `l` eintritt, wird `D` mit dem Subchart aktiviert der zuletzt in `D` aktiv war. Da dies wieder keine zulässige Konfiguration für `Y` ergibt, erfolgt implizit eine Ergänzung zu der Konfiguration {`A`,`h(D)`}. Dies alles haben wir in der textuellen Darstellung explizit aufgeführt.

Da die strukturellen Anforderungen an eine Statechart Beschreibung mit den zulässigen Konfigurationen verbunden sind, definieren wir nachfolgend die Operation *Cfg*, die für einen Statechart Q die Menge seiner zulässigen Konfigurationen für aktive Zustände liefert; jede zulässige Konfiguration ist hierbei eine Menge von Statecharts, die gemeinsam aktiv sein können. Da Zielpunkte auch Konnektoren

sein können, definieren wir die Operation Cfg^h, die neben den Konfigurationen aus Cfg auch solche Mengen enthalten kann, die für Subcharts Q' von Q Zielpunkte der Gestalt $h(Q')$ und $h^*(Q')$ enthalten. Als Hilfskonstrukt definieren wir dazu als weitere Operationen

$$Cov(Q) \triangleq \{\{Q\}\} \cup Cfg(Q)$$

$$Cov^h \triangleq \{\{Q\}, \{h(Q)\}, \{h^*(Q)\}\} \cup Cfg^h(Q)$$

Die Formen Cfg^h und Cov^h enthalten zusätzlich die Konnektoren für *history* und *deep-history*. Daneben definieren wir im folgenden auch eine kompakte Darstellungform für Beschreibungen in Statecharts, auf die wir auch später für die formale Behandlung zurückgreifen werden.

1. Ist Q ein einfacher Statechart, so stellen wir ihn kompakt durch seinen Namen Q dar und defininiereen $Cfg(Q) \triangleq \emptyset$ und $Cfg^h(Q) \triangleq \emptyset$.

2. Ist Q ein AND-Statechart, der aus den Statecharts $Q_1, \ldots, Q_n$ aufgebaut ist, so stellen wir ihn kompakt dar durch $\bigwedge_{i=1}^n Q_i$ und definieren

$$Cfg(Q) \triangleq \{\cup_{i=1}^n c_i \mid c_i \in Cov(Q_i)\}$$
$$Cfg^h(Q) \triangleq \{\cup_{i=1}^n c_i \mid c_i \in Cov^h(Q_i)\}$$

3. Ist Q ein OR-Statechart, der aus den Statecharts $Q_1, \ldots, Q_n$ aufgebaut ist, so stellen wir ihn kompakt dar durch $(\bigvee_{i=1}^n Q_i, Tr)$. Tr ist die Menge der in Q auftretenden Transitionen. Jede Transition stellen wir durch ein Tupel $(sources, label, targets)$ dar, wobei *sources* die Ausgangspunkte der Transition, *targets* die Zielpunkte der Transition und *label* die Anweisungen an der Transition beschreiben. Die Initialkonfiguration wird durch eine Transition der Form $(\emptyset, init_{SC}, targets)$ dargestellt. Zulässige Konfigurationen definieren wir durch

$$Cfg(Q) \triangleq \cup_{i=1}^n Cov(Q_i)$$
$$Cfg^h(Q) \triangleq \cup_{i=1}^n Cov^h(Q_i)$$

Damit eine gültige Beschreibung in Statecharts vorliegt, muss sie syntaktisch korrekt sein. Zudem müssen die Transitionen folgende Anforderungen erfüllen: Ist Q ein OR-Statechart der Form $(\bigvee_{i=1}^n Q_i, Tr)$ und ist $(sources, label, targets) \in Tr$

- die Transition für die initiale Konfiguration ($sources = \emptyset$), so ist *targets* eine zulässige Konfiguration von Q : $targets \in Cfg(Q)$.
- nicht die Transition für die initiale Konfiguration ($sources \neq \emptyset$), so gilt
  1. *sources* ist eine zulässige Ausgangskonfiguration von Q: $sources \in Cfg(Q)$
  2. *targets* ist eine zulässige Zielkonfiguration von Q: $targets \in Cfg^h(Q)$

2.2 Message Sequence Charts

Mit MSCs wird die Kommunikation zwischen verschiedenen Objekten bzw. (Teil-) Systemen beschrieben. MSCs können, wie auch Statecharts, mit einer graphischen Notation visuell gut erfassbar dargestellt werden. Daneben gibt es auch eine textuelle Syntax für MSCs. Im Rahmen dieses Papiers gehen wir i.W. von einfachen MSCs aus, wie sie in [RGG96, MR94] beschrieben sind; wir erweitern lediglich die Syntax um eine Möglichkeit, Bedingungen zu spezifizieren.

```
<msc>        ::=  msc <id> <msc body> endmsc ;
<msc body>   ::=  <inst def> | <inst def> <msc body>
<inst def>   ::=  instance <id> ; <inst body> endinstance ;
<inst body>  ::=  <event> | <event> <inst body>
<event>      ::=  in <id> from <inst> ;
                |  out <id> to <inst> ;
                |  action <id> ;
                |  cond <cond> ;
<inst>       ::=  <id> | env
```

Abbildung 2.3: Syntax für MSCs

```
msc example
  instance I1
    in m1 from env;
    action a;
    out m2 to I2;
    cond c;
    in m4 from I2;
  endinstance;
  instance I2
    action b;
    in m2 from I1;
    out m3 to env;
    out m4 to I2;
  endinstance;
endmsc;
```

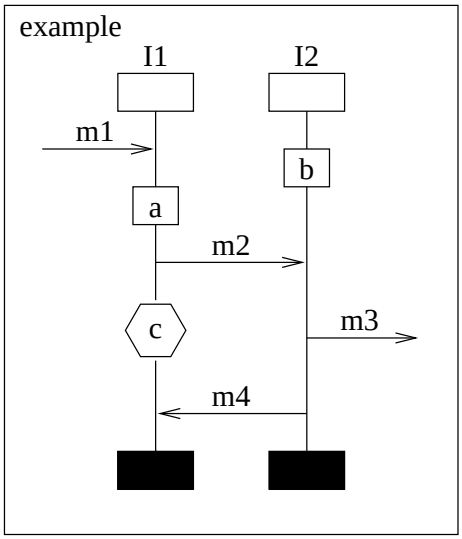

Abbildung 2.4: Ein Beispiel MSC: textuelle und graphische Darstellung

Die Syntax ist in Abbildung 2.3 angegeben. Ein MSC besteht aus sogenannten Instanzen (`instance`); eine Instanz beschreibt eine Abfolge von Ereignissen für ein Objekt bzw. (Teil-)System[1] und Bedingungen, die zwischen dem Auftreten dieser Ereignissen gelten sollen. Dabei werden die Ereignisse und Bedingungen in der Reihenfolge aufgezählt, in der sie auftreten sollen. Ereignisse umfassen den Eingang und Ausgang von Nachrichten, die Ausführung von Aktionen.

Ein Beispiel für ein MSC ist in Abbildung 2.4 präsentiert. Es beschreibt Abfolgen von Ereignissen und Bedingungen für zwei Objekte, die miteinander kommunizieren. `I1` beschreibt, dass zunächst Nachricht `m1` aus der Umgebung empfangen wird, danach Aktion `a` auftritt, dann Nachricht `m2` an `I2` geschickt wird, danach die Bedingung `c` gilt und schliesslich Nachricht `m4` von `I2` empfangen wird. Entsprechend beschreibt `I2`, dass zunächst Aktion `b` auftritt, danach Nachricht `m2` von `I1` empfangen wird, dann Nachricht `m3` an die Umgebung geschickt wird und schliesslich Nachricht `m4` an `I2` geschickt wird.

Mit einem MSC wird somit eine partielle Ordnung auf den Ereignissen und den Bedingungen festgelegt. Die Ereignisse entlang einer Instanz können nur in der Reihenfolge auftreten, in der sie aufgezählt sind. Ausserdem kann eine Nachricht nicht empfangen werden, bevor sie abgeschickt wurde. Ereignisse, die in unterschiedlichen Instanzen aufgezählt werden, sind unabhängig voneinander. So ist im Beispiel

[1] Nachfolgend setzen wir Objekte und (Teil-)Systeme gleich und sprechen nur noch von Objekten

(Abbildung 2.4) nicht festgelegt, welche der Aktionen `a` oder `b` zuerst auftritt, d.h. sie können relativ zueinander in beliebiger Reihenfolge auftreten. Somit ergibt sich eine Fülle von Möglichkeiten, um die Ereignisse anzuordnen.

2.3 Verküpfung von MSCs mit Statecharts

MSCs werden zur Beschreibung von Szenarien aus der Interaktion zwischen Objekten eingesetzt. Hingegen beschreiben Statecharts das Verhalten von Objekten in Termini von Zuständen und Zustandsübergängen — in den Zuständen werden die Aktivität von Statecharts, die Werte von Zustandsvariablen und die vorliegenden Ereignisse repräsentiert; Nachrichten, die zur Kommunikation verschickt werden, treten nicht explizit auf. Um eine Relation zwischen MSCs und Statecharts zu etablieren, muss man zunächst die Begriffe der beiden Beschreibungstechniken miteinander integrieren. Sowohl Statecharts als auch die in MSCs auftretenden Instanzen sind mit Objekten assoziiert; wenn ein Statechart und eine Instanz in einem MSC mit dem gleichen Objekt assoziiert sein sollen, verwenden wir für beide den Namen des Objektes, d.h. eine Assoziation mit dem gleichen Objekt wird durch Namensgleichheit ausgedrückt. Als weitere begriffliche Integration legen wir fest, wie Nachrichten repräsentiert werden sollen; wir verbinden das Versenden einer Nachricht mit einer Variablen und mit zwei Ereignissen: Die Variable repräsentiert das Vorliegen einer bestimmten Nachricht von einem bestimmten Absender zu einem bestimmten Empfänger; die Ereignisse stehen für den Ausgang und den Eingang der Nachricht.

```
<data decl>        ::=  ...
                      |  message <id> :  <id> , <id>
                      |  action <id> :  <id>
<action>           ::=  ...
                      |  <send action>
                      |  <receive action>
<send action>      ::=  out ( <id> )
<receive action>   ::=  in ( <id> )
```

Abbildung 2.5: Erweiterte Statechart-Syntax für Nachrichten

Wir erweitern die Syntax für Statecharts entsprechend um Nachrichten deklarieren, senden und empfangen zu können; siehe Abbildung 2.5. Wenn eine Nachricht abgeschickt wird, liegt sie vor, bis sie empfangen wird. In unserer Modellierung machen wir die Einschränkung, dass eine bestimmte Nachricht nicht noch einmal abgeschickt werden kann, solange sie bereits vorliegt. Eine Deklaration der Form `message msg: O1,O2` entspricht den drei Deklarationen

```
event msg_out;
event msg_in;
var msg: bool;
```

Die Aktion `out(msg)` kann nur innerhalb eines Statechart (für) `O1` auftreten und kann ausgeführt werden, wenn (`msg=false`) gilt; wenn sie ausgeführt wird, wird der Variablen `msg` der Wert `true` zugewiesen und das Ereignis `msg_out` erzeugt. Analog dazu kann `in(msg)` nur innerhalb des Statechart für `O2` auftreten und kann nur ausgeführt werden, wenn `msg=true` gilt; wenn sie ausgeführt wird, wird der Variablen `msg` der Wert `false` zugewiesen und das Ereignis `msg_in` erzeugt.

Für MSC-Aktionen deklarieren wir spezielle Ereignisse; siehe Abbildung 2.5. Ein Ereignis, das in der Form `action a:O;` deklariert ist, kann nur in der MSC-Instanz

```
<msc ctl>   ::=  E <msc ltl> | A <msc ltl>
               |  <cond>
               |  ( <msc ctl> )
               |  not <msc ctl>
               |  <msc ctl> and <msc ctl>
               |  <msc ctl> or <msc ctl>
<msc ltl>   ::=  <msc form>
               |  X <msc ctl>
               |  F <msc ctl>
               |  G <msc ctl>
               |  <msc ctl> U <msc ctl>
<msc form>  ::=  <id>
               |  ( <msc form> )
               |  not <msc form>
               |  <msc form> and <msc form>
               |  <msc form> or <msc form>
```

Abbildung 2.6: Syntax für MSC-CTL

und in dem Statechart für Objekt `O` auftreten. Bedingungen in MSCs interpretieren wir wie die Bedingungen in den Transitionen eines Statechart.

Nachfolgend definieren wir, wie in unserem Ansatz MSCs als Anforderungen über Statecharts interpretiert werden sollen: MSCs sollen Abfolgen von Ereignissen und Bedingungen beschreiben, die bei der Ausführung eines System auftreten, das durch einen entsprechende Statechart definiert ist. Zwischen den Ereignissen, die für eine Instanzen definiert sind, sollen Ereignisse, die nicht für diese Instanz definiert sind, auftreten dürfen. Die für eine Instanz definierten Ereignisse sollen aber nur in der Reihenfolge auftreten, in der sie in der Instanz aufgezählt werden. Diese Interpretation ist entspricht i.W. der in Termini von Prozessalgebren angegebenen Semantik [MR94] von MSCs: Da mit dem prozessalgebraischen Ansatz eine "Interleaving-Semantik" vorgegeben ist, können voneinander unabhängige Ereignisse (aus unterschiedlichen Instanzen) zwar in beliebiger Reihenfolge, aber nicht gemeinsam auftreten. Diese Einschränkung gibt es in unserem auf Temporallogik basierten Ansatz nicht.

Die Semantik aus [MR94] hat allerdings noch weitere Mängel: Es ist nicht festgelegt, ob ein MSC für jede Ausführung eines Systems gelten soll oder nur für manche. Je nach Sichtweise kann einmal die eine und ein anderes mal die andere Anforderung gewünscht sein. Ein Ausweg aus dieser Situation besteht darin, den Anwender dieser Beschreibungstechniken explizit wählen zu lassen, welche der Anforderungen er wünscht. Mit MSC_{CTL} liegt ein zu LSCs [DH98] ähnlicher Ansatz zur Lösung dieses Problems vor: Das Grundprinzip bei LSCs ist, den Anwender explizit festlegen zu lassen, ob etwas (z.B. eine Bedingung, ein Ereignis, ein ganzes Chart) auftreten muss (hot) oder auftreten kann (cold). Dafür wird die Syntax von MSCs und den syntaktischen Konstrukten, aus denen MSCs aufgebaut sind, entsprechend verändert und erweitert. Hiervon unterscheidet sich unser Ansatz, MSC_{CTL}, zunächst darin, dass wir MSCs unverändert übernehmen und sie in eine erweiterte Notation einbetten: Dabei werden MSCs über die Namen, mit denen sie definiert wurden, referenziert. Die Auswahl über die zu betrachtenden Ausführungen wird ausgedrückt, indem explizit ein Pfadquantor A (jede Ausführung) bzw. E (manche Ausführung) verwendet wird (Syntax in Abbildung 2.6). Darüber hinaus können MSCs in unserem Ansatz negiert werden, wenn entsprechende Abfolgen nicht eintreten sollen, sie können konjugiert werden, wenn beide Abfolgen auftre-

```
<consistency>  ::=  <sc spec> <msc def> satisfies <msc specs>
<msc def>      ::=  <msc> | <msc> <msc def>
<msc-specs>    ::=  <msc spec> | <msc spec> <msc specs>
<msc spec>     ::=  spec :  <msc ctl>
```

Abbildung 2.7: Syntax für Konsistenzrelation zwischen Statecharts und MSCs

ten sollen und sie können mit einer Disjunktion verknüpft sein, wenn Alternativen ermöglicht werden sollen; für eine solche Verknüpfung von MSCs kann man mit den Pfadquantoren festlegen, ob sie für alle oder nur manche der Ausführungen eines Systems gelten sollen. Diese können dann als Teilformeln innerhalb von Ausdrücken im Stil von CTL stehen, um auch komplexere Sachverhalte modellieren zu können: Beispielsweise können während der Ausführung eines Statecharts bestimmte Abläufe (in Schleifen) iteriert auftreten; hierbei kann es sinnvoll sein, festzulegen, ab welchen Positionen in einer Ausführung ein MSC erfüllt sein soll. Solche Positionen kann man beispielsweise mit Bedingungen charakterisieren; bei LSCs werden diese als Aktivierungsbedingung bezeichnet. In unserem Ansatz kann man entsprechende Aussagen formulieren, indem man geeignete Formeln (entsprechend der syntaktischen Regel für `<msc ctl>` in Abbildung 2.6) bildet: Wenn beispielsweise jedesmal, wenn eine Bedingung c gilt, in allen nachfolgenden Ausführungen des Systems der mit dem Namen *msc* definierte MSC gelten soll, kann das durch die Formel

`A G ( (not` c`) or (A` msc`) )`

ausgedrückt werden; der MSC_{CTL}-Ausdruck `(E F E` msc`)` besagt, dass eine Ausführung existiert, so dass irgendwann der MSC *msc* erfüllt wird. Zusätzlich bietet uns dieser Ansatz auch die Möglichkeit, Anforderungen auch unabhängig von MSCs direkt in CTL zu formulieren, solche Anforderungen mit den Anforderungen aus den MSCs zu verbinden oder auch Abhängigkeiten zwischen den MSCs auszudrücken.

Die Syntax ist rudimentär gehalten und kann bei Bedarf mit den üblichen notationellen Abkürzungen (z.B. Implikation) erweitert werden. Einige Abkürzungen sind bereits enthalten und es gelten die üblichen Regeln für Pfadquantoren und temporallogische Operatoren: $\mathsf{A}\Phi \triangleq \neg\mathsf{E}\neg\Phi$, $\mathsf{G}\Phi \triangleq \neg\mathsf{F}\neg\Phi$, $\mathsf{F}\Phi \triangleq \mathit{true}\mathsf{U}\Phi$.

Wir definieren, dass eine Statechart-Spezifikation konsistent mit einer MSC-Spezifikation ist, wenn der Statechart die MSC_{CTL} Formel erfüllt. Die zwischen einem Statechart und den MSCs gewünschte Konsistenzrelation wird syntaktisch durch die in Abbildung 2.7 angegebene Form spezifiziert.

Kapitel 3

Formalisierung der Integration von Statecharts und MSCs

Wir integrieren Statecharts und MSCs miteinander, indem wir Statecharts als Spezifikationen von Transitionssystemen auffassen und MSCs als Spezifikationen von Eigenschaften über den von dem Transitionssystem generierten Ausführungspfaden interpretieren; dazu haben wir MSCs zu MSC_{CTL} erweitert und interpretieren sie als temporallogische Formeln über dem Transitionssystem. Die Konsistenz zwischen einer Statecharts- und einer MSC-Spezifikation ergibt sich dann ganz regulär als Erfülltheit der temporallogischen Formeln über dem Transitionssystem.

Die nachfolgend präsentierte Definition eines abstrakten Transitionssystems für Statecharts ist abgleitet aus der Semantik für Statecharts [HN96, HPSS87]. Die Umsetzung eines Statechart in ein abstraktes Transitionssystem erfolgt in zwei Schritten: In einem ersten Schritt wird aus der Struktur eines Statechart ein markiertes Transitionssystem (*labeled transition system*) abgeleitet, in dem Anweisungen nur als Marken betrachtet werden. Aus diesem wird in einem zweiten Schritt ein Transitionssystem abgeleitet, in dem die Anweisungen entsprechend ihrer Bedingungen und Aktionen berücksichtigt sind. Danach beschreiben wir, wie MSCs und unsere notationelle Erweiterungen in CTL* dargestellt werden und wir legen formal die Konsistenzbedingung zwischen Statecharts und MSCs fest.

3.1 Markiertes Transitionssystem

Statecharts sind hierarchisch strukturiert aufgebaut. Das kann man für die Definition eines markierten Transitionssystems verwenden: das markierte Transitionssystem für ein Statechart kann in Termini der markierten Transitionssysteme seiner Subcharts definiert werden. Das nachfolgend definierte markierte Transitionssystem ist reichhaltiger als der zugrundeliegende Statechart; es enthält Zustände und Zustandsübergänge, die im Statechart nicht vorkommen. Allerdings vereinfacht diese Vorgehensweise die Modellierung; bei der Definition des vollständigen Transitionssystems werden wir die Zustandsübergänge, die in dem Statechart nicht auftreten, wieder weglassen und den Initialzustand so wählen, dass er der Initialkonfiguration entspricht.

Ein markiertes Transitionssystem für einen Statechart SC ist ein Tupel

$$LTS_{SC} \triangleq (B_{SC}, A_{SC}, L_{SC}, I_{SC}, \Delta_{SC})$$

B_{SC} ist die Menge der Grundzustände von SC.

$A_{SC} \subseteq B_{SC}$ ist die Menge der Grundzustände in denen SC aktiv ist.

L_{SC} ist die Menge der Markierungen; Markierungen entsprechen den Anweisungen an einer Transition des Statechart.

I_{SC} ist der anfängliche Grundzustand des markierten Transitionssystems. Zu beachten ist, dass er nicht der Initialkonfiguration des Statechart entspricht. In I_{SC} ist der Statechart nicht aktiviert; der Grundzustand, der der Initialkonfiguration entspricht wird durch einen Zustandsübergang erreicht, der von I_{SC} aus in einen aktiven Zustand führt.

$\Delta_{SC} \subseteq B_{SC} \times 2^{L_{SC}} \times B_{SC}$ definiert die (markierte) Zustandsübergangsrelation. An einem Zustandsübergang können mehrere Anweisungen (Markierungen) beteiligt sein.

Das markierte Transitionssystem für einen Statechart SC definieren wir über seinen strukturellen Aufbau. Für die Definitionen von LTS_{SC} verwenden wir noch einige Begriffe, Notationen und Hilfskonstrukte, die wir hier zunächst einführen.

$active_{SC}(b)$ ist eine Alternative für $b \in A_{SC}$.

$activate_{SC} : Cov^h(SC) \rightarrow 2^{B_{SC} \times 2^{L_{SC}} \times B_{SC}}$ ist eine Funktion, die zu aktivierende Zielkonfigurationen abbildet nach Mengen von Zustandsübergängen, die einen nicht-aktiven Grundzustand von SC in einen Grundzustand von SC überführen, der entsprechend der angegebenen Zielkonfiguration aktiviert ist. Wir definieren die Funktion weiter unten gemeinsam mit LTS_{SC} über der Struktur von SC.

$deactivate_{SC} : Cov(SC) \rightarrow 2^{B_{SC} \times 2^{L_{SC}} \times B_{SC}}$ ist eine Funktion, die (Ausgangs-)Konfigurationen abbildet nach Mengen von Zustandsübergängen, die einen entsprechend der Konfiguration aktiven Grundzustand von SC in einen nicht-aktiven Grundzustand von SC überführen. Wir definieren die Funktion weiter unten gemeinsam mit LTS_{SC} über der Struktur von SC.

Ist SC ein einfacher Statechart, legen wir LTS_{SC}, $activate_{SC}$ und $deactivate_{SC}$ durch folgende Definitionen fest:

- Der Grundzustandraum besteht aus den boole'schen Werten: $B_{SC} \triangleq bool$
- Der aktive Statechart entspricht dem Grundzustand *true*: $A_{SC} \triangleq \{true\}$
- Der anfängliche Grundzustand ist *false*: $I_{SC} \triangleq false$
- Da keine internen Zustandsübergänge definiert werden können, gibt es auch keine Anweisungen/Markierungen: $L_{SC} \triangleq \emptyset$
- Ein einfacher Statecharts wird ohne interne Zustandsübergänge definiert. Allerdings ist es möglich, ihn mit anderen Statecharts parallel zu kombinieren; dabei können Zustandsübergänge mit Auswirkungen in den anderen Statecharts auftreten, ohne dass der einfache Statechart seinen Zustand ändert. Zu diesem Zweck definieren wir einen Zustandsübergang, der keine Auswirkungen hat: $\Delta_{SC} \triangleq \{(true, \emptyset, true)\}$.
- Die Aktivierung von SC führt vom Grundzustand *false* in den Grundzustand *true*: $activate_{SC}(c) \triangleq \{(false, \emptyset, true)\}$ für alle $c \in Cov^h(SC)$.

- Die Deaktivierung des Statechart führt vom Grundzustand *true* zum Grundszustand *false*: $deactivate_{SC}(SC) \triangleq \{(true, \emptyset, false)\}$

Ist SC ein AND-Statechart der Form $\bigwedge_{i=1}^{n} Q_i$ und sind LTS_{Q_i} die markierten Transitionssysteme der Q_i so legen wir LTS_{SC}, $activate_{SC}$ und $deactivate_{SC}$ durch folgende Definitionen fest[1]:

- Der Grundzustandsraum besteht aus dem Kreuzprodukt der Grundzustandsräume der Q_i: $B_{SC} \triangleq \times_{i=1}^{n} B_{Q_i}$
- SC ist genau dann aktiv, wenn alle Q_i aktiv sind:
  $A_{SC} \triangleq \{(q_i)_1^n \in B_{SC} \mid \forall\, i : active_{Q_i}(q_i)\}$
- Der anfängliche Grundzustand setzt sich zusammen aus den I_{Q_i}:
  $I_{SC} \triangleq (I_{Q_i})_1^n$
- Die Menge der Markierungen ist die Vereinigung der Markierungen der Q_i:
  $L_{SC} \triangleq \bigcup_{i=1}^{n} L_{Q_i}$
- Die Zustandsübergänge sind parallele Kompositionen der Zustandsübergänge der Q_i: $\Delta_{SC} \triangleq \{((q_i)_1^n, \bigcup_{i=1}^{n} L_i, (r_i)_1^n) \mid \forall\, i : (q_i, L_i, r_i) \in \Delta_{Q_i}\}$
- $activate_{SC}(tgt)$ beschreiben wir mit einer Fallunterscheidung nach der Zielkonfiguration tgt:

  1. Ist $tgt \in \{\{SC\}, \{h(SC)\}\}$, wird SC aktiviert, indem alle Q_i aktiviert werden:
     $activate_{SC}(tgt) \triangleq \{\ ((q_i)_1^n, \bigcup_{i=1}^{n} L_i, (r_i)_1^n) \mid \forall\, i : (q_i, L_i, r_i) \in activate_{Q_i}(\{Q_i\})\ \}$
  2. Eine Aktivierung mit $tgt = \{h^*(SC)\}$ wird erreicht, indem die Q_i mit *deep-history* aktiviert werden:
     $activate_{SC}(tgt) \triangleq \{\ ((q_i)_1^n, \bigcup_{i=1}^{n} L_i, (r_i)_1^n) \mid \forall\, i : (q_i, L_i, r_i) \in activate_{Q_i}(\{h^*(Q_i)\})\ \}$
  3. SC wird mit einer Konfiguration $tgt \in Cfg^h(SC)$ aktiviert, indem die Q_i mit den zugehörigen Teilkonfigurationen aus tgt aktiviert werden:
     $activate_{SC}(tgt) \triangleq \{\ ((q_i)_1^n, \bigcup_{i=1}^{n} L_i, (r_i)_1^n) \mid \exists\, tgt_1, \ldots, tgt_n : tgt = \bigcup_{i=1}^{n} tgt_i \wedge \forall\, i : tgt_i \in Cov^h(Q_i) \wedge (q_i, L_i, r_i) \in activate_{Q_i}(tgt_i)\ \}$

- $deactivate_{SC}(src)$ definieren wir mit einer Fallunterscheidung nach der Ausgangskonfiguration src:

  1. Die Deaktivierung von SC in Konfiguration $src = \{SC\}$ entspricht der Deaktivierung aller Q_i:
     $deactivate_{SC}(src) \triangleq \{\ ((q_i)_1^n, \bigcup_{i=1}^{n} L_i, (r_i)_1^n) \mid \forall\, i : (q_i, L_i, r_i) \in deactivate_{Q_i}(\{Q_i\})\ \}$
  2. SC wird in einer Konfiguration $src \in Cfg(SC)$ deaktiviert, indem die Q_i in den zugehörigen Teilkonfigurationen aus src deaktiviert werden:
     $deactivate_{SC}(src) \triangleq \{\ ((q_i)_1^n, \bigcup_{i=1}^{n} L_i, (r_i)_1^n) \mid \exists\, src_1, \ldots, src_n : src = \bigcup_{i=1}^{n} src_i \wedge \forall\, i : src_i \in Cov(Q_i) (q_i, L_i, r_i) \in deactivate_{Q_i}(src_i)\ \}$

[1] $(q_i)_1^n$ bzw. $(q_i)_{i=1}^n$ steht für das Tupel $(q_1, \ldots, q_n)$

Ist SC ein OR-Statechart der Form $(\bigvee_{i=1}^{n} Q_i,\ Tr_{SC})$ und sind LTS_{Q_i} die markierten Transitionssysteme der Q_i so legen wir LTS_{SC}, $activate_{SC}$ und $deactivate_{SC}$ durch folgende Definitionen fest:

- Der Grundzustandsraum besteht aus dem Kreuzprodukt der Grundzustandsräume der Q_i sowie einer Komponente um festzuhalten, welcher der Q_i zuletzt aktiv war: $B_{SC} \triangleq (\times_{i=1}^{n} B_{Q_i}) \times \{\text{“void”}, \text{“}\mathsf{Q}_1\text{”}, \ldots, \text{“}\mathsf{Q}_n\text{”}\}$

- SC ist aktiv, wenn einer der Q_i aktiv ist:
  $A_{SC} \triangleq \{((q_i)_1^n, h) \in B_{SC} \mid \exists\, i : active_{Q_i}(q_i)\}$

- Der anfängliche Grundzustand setzt sich zusammen aus den I_{Q_i}; die zusätzliche Komponente wird mit “void” initialisiert:
  $I_{SC} \triangleq ((I_{Q_i})_1^n, \text{“void”})$

- Die Menge der Markierungen besteht aus den Markierungen für die Q_i und den Anweisungen aus den für SC definierten Transitionen:
  $L_{SC} \triangleq (\bigcup_{i=1}^{n} L_{Q_i}) \cup (\{l \mid \exists\, s, t : (s, l, t) \in Tr_{SC}\})$

- Die Zustandsübergangsrelation setzt sich zusammen aus den internen Zustandsübergängen der Subcharts Q_i sowie der Intialtransition und den Transitionen zwischen den Q_i. Wenn ein Q_i aktiv ist und ein interner Zustandsübergang für ihn eintritt, so bleiben die anderen Subcharts unverändert. Die Initialtransition entspricht einer Aktivierung von SC ausgehend vom anfänglichen Grundzustand I_{SC}. Den Zustandsübergang für eine Transition von einer Ausgangskonfiguration src zu einer Zielkonfiguraion tgt definieren wir als Deaktivierung von SC mit src und anschliessender Aktivierung mit tgt:
  $$\begin{aligned}\Delta_{SC} \triangleq\ & \{(((q_i)_1^n, h), L, ((r_i)_1^n, h)) \mid \exists\, j : active_{Q_j}(q_j) \wedge (q_j, L, r_j) \in \Delta_{Q_j} \\ & \qquad \wedge \forall\, i \neq j : q_i = r_i\} \\ \cup\ & \{(I_{SC}, L, a) \mid (I_{SC}, L, a) \in activate_{SC}(\{SC\})\} \\ \cup\ & \{(a_1, L_1 \cup \{l\} \cup L_2, a_2) \mid \exists\, (s, l, t) \in Tr_{SC}, \exists\, b \in B_{SC} : \\ & \qquad s \in Cfg(SC) \wedge t \in Cfg^h(SC) \wedge \\ & \qquad (a_1, L_1, b) \in deactivate_{SC}(s) \wedge \\ & \qquad (b, L_2, a_2) \in activate_{SC}(t)\}\end{aligned}$$

- $activate_{SC}(tgt)$ definieren wir mit einer Fallunterscheidung nach der Zielkonfiguration tgt:

  1. Ist $tgt = \{SC\}$, wird SC aktiviert, indem die Zielpunkte aus der Transition für die Initialkonfiguration aktiviert werden:
     $$\begin{aligned}activate_{SC}(tgt) \triangleq\ & \{\ (b, \{init_{SC}\} \cup L, a) \mid \exists\, (\emptyset, init_{SC}, t) \in Tr_{SC} : \\ & \qquad (b, L, a) \in activate_{SC}(t)\ \}\end{aligned}$$

  2. Ist $tgt = \{h(SC)\}$, wird der zuletzt in SC aktive Subchart inital aktiviert; war SC noch nicht aktiv, so wird seine Initialkonfiguration aktiviert:
     $$\begin{aligned}activate_{SC}(tgt) \triangleq\ & \{\ (((q_i)_1^n, h_1), L, ((r_i)_1^n, h_2)) \mid \\ & \qquad (h_1 = \text{“void”} \wedge \\ & \qquad\ (((q_i)_1^n, h_1), L, ((r_i)_1^n, h_2)) \in activate_{SC}(\{SC\})) \\ & \quad \vee\ (\exists\, j : h_1 = \text{“}\mathsf{Q}_j\text{”} \wedge h_1 = h_2 \wedge \\ & \qquad (q_j, L, r_j) \in activate_{Q_j}(\{Q_j\}) \wedge \\ & \qquad \forall\, i \neq j : q_i = r_i)\ \}\end{aligned}$$

  3. Ist $tgt = \{h^*(SC)\}$, erfolgt die Aktivierung mit der zuletzt aktiven Konfiguration; war SC noch nicht aktiv, so wird seine Initialkonfiguration

aktiviert:

$$activate_{SC}(tgt) \triangleq \{ (((q_i)_1^n, h_1), L, ((r_i)_1^n, h_2)) \mid (h_1 = \text{“void”} \wedge (((q_i)_1^n, h_1), L, ((r_i)_1^n, h_2)) \in activate_{SC}(\{SC\})) \vee (\exists j : h_1 = \text{“}\mathsf{Q}_j\text{”} \wedge h_1 = h_2 \wedge (q_j, L, r_j) \in activate_{Q_j}(\{h^*(Q_j)\}) \wedge \forall i \neq j : q_i = r_i) \}$$

4. Ist $tgt \in Cfg^h(SC)$, so wird der Subchart von SC, in dem tgt liegt, mit dieser Konfiguration aktiviert:

$$activate_{SC}(tgt) \triangleq \{ (((q_i)_1^n, h_1), L, ((r_i)_1^n, h_2)) \mid \exists j : tgt \in Cov^h(Q_j) \wedge h_2 = \text{“}\mathsf{Q}_j\text{”} \wedge (q_j, L, r_j) \in activate_{Q_j}(tgt) \wedge \forall i \neq j : q_i = r_i) \}$$

- $deactivate_{SC}(src)$ definieren wir mit einer Fallunterscheidung nach der Ausgangskonfiguration src:

1. Ist $src = \{SC\}$, so wird der gerade aktive Subchart von SC deaktiviert:

$$deactivate_{SC}(src) \triangleq \{ (((q_i)_1^n, h), L, ((r_i)_1^n, h)) \mid \exists j : active_{Q_j}(q_j) \wedge (q_j, L, r_j) \in deactivate_{Q_j}(\{Q_j\}) \wedge \forall i \neq j : q_i = r_i) \}$$

2. Ist $src \in Cfg(SC)$, so gibt es ein Q_j mit $src \in Cov(Q_j)$ und die Deaktivierung von SC mit src entspricht der Deaktivierung von Q_i mit der Konfiguration src:

$$deactivate_{SC}(src) \triangleq \{ (((q_i)_1^n, h), L, ((r_i)_1^n, h)) \mid \exists j : src \in Cov(Q_j) \wedge active_{Q_j}(q_j) \wedge (q_j, L, r_j) \in deactivate_{Q_j}(src) \wedge \forall i \neq j : q_i = r_i) \}$$

3.2 Vollständige Umsetzung in ein Transitionssystem

Im vorangegangenen Abschnitt haben wir die Umsetzung von Statecharts in markierte Transitionssysteme beschrieben. Dabei haben wir die Bedeutung der Anweisungen an den Zustandsübergängen (Markierungen) ignoriert. In diesem Abschnitt beschreiben wir die vollständige Umsetzung von Statecharts in die zugehörigen abstrakten Transitionssysteme, in denen die Semantik der Anweisungen ebenfalls berücksichtigt ist.

Eine Anweisung (bzw. Markierung) besteht aus einer Bedingung und einer Aktion, die jeweils optional sind. In der Bedingung können das Vorliegen von Ereignissen und Eigenschaften über den Zustandsvariablen geprüft werden. Ein Zustandsübergang kann nur dann ausgeführt werden, wenn ihre Bedingung erfüllt ist; eine leere Bedingung ist immer erfüllt. Die Aktionen können Ereignisse erzeugen oder Zustandsvariablen verändern. Wenn man die Bedeutung von Anweisungen hinzunimmt, führt das somit im Wesentlichen zu einer Reduzierung der möglichen Ausführungssequenzen: Zustandsübergänge deren Bedingung nicht erfüllt sind, können nicht auftreten und wenn verschiedene Aktionen in Konflikt zueinander stehen, können die zugehörigen Zustandsübergänge nicht gleichzeitig auftreten.

Wir setzen Statecharts in abstrakte Transitionssysteme um, indem wir von den zugehörigen markierten Transitionssystemen ausgehen. Auf der einen Seite schränken wir dazu die Zustandsübergangsrelation entsprechend der Bedeutung der

Anweisungen ein. Auf der anderen Seite erweitern wir den Zustandsraum so, dass Variablen und Ereignisse repräsentiert werden können.

Wie oben sei wiederum SC ein Statechart und LTS_{SC} das zugehörige markierte Transitionssystem. X sei die Menge der in SC auftretenden Variablen. X besteht aus Datenelementen mit Deklarationen der Form "`var x:`W_x" und "`message m:...`" (boole'sche Variable `m`). Für jede Variable $x \in X$ bezeichnen wir mit ihrem Typnamen W_x auch ihren Wertebereich. E sei die Menge der in SC auftretenden Ereignisse. E besteht aus Datenelementen mit Deklarationen der Form "`event e`", "`action a:...`" und "`message m:...`" (Ereignisse `m_in` und `m_out`). Der Zustandsraum um Variablen und Ereignisse zu repräsentieren ist definiert durch

$$S_{XE} \triangleq (\times_{x \in X} W_x) \times (\times_{e \in E} bool)$$

Für ein $s \in S_{XE}$ sei $x(s)$ der in s repräsentierte Wert von $x \in X$; $e(s)$ stellt dar, ob Ereignis $e \in E$ in s vorliegt ($e(s) = true$) oder nicht ($e(s) = false$).

Wir gehen davon aus, dass kein Ereignis vorliegt, solange der zugehörige Statechart SC sich im anfänglichen Grundzustand I_{SC} befindet. Die Menge der anfänglichen Variablen- und Ereigniszustände bezeichnen wir mit S_0; für alle $s_0 \in S_0$ und alle $e \in E$ soll gelten[2] $e(s_0) = false$.

Eine Anweisung l besteht aus Bedingungen und Aktionen. Die Erfülltheit einer Bedingung in einem Zustand s ist in der üblichen Weise definiert, d.h. sie wird über $x(s)$ und $e(s)$ ausgewertet. Die Aktionen können Ereignisse erzeugen und Variablen verändern. Die Semantik einer Anweisung l definieren wir in Form eines Tupels:

$$[\![l]\!] \triangleq (event_l, var_l, op_l)$$

- $event_l$ ist die Menge der von l erzeugten Ereignisse.
- var_l ist die Menge der von l modifizierten Variablen.
- $op_l \subseteq S_{XE} \times S_{XE}$ ist die zugehörige Zustandsübergangsrelation.

Die konkrete Semantik von Anweisungen ist über den syntaktischen Aufbau definiert. Die technischen Details sind in Anhang B beschrieben.

In den (markierten) Zustandsübergängen aus Δ_{SC} können mehrere Anweisungen gemeinsam auftreten. Wenn ein Zustandsübergang eintritt, werden alle darin auftretenden Anweisungen gleichzeitig ausgeführt. Daher erweitern wir die Semantik auf Mengen ($L \subseteq 2^{L_{SC}}$) von Anweisungen:

$$[\![L]\!] \triangleq (event_L, var_L, op_L)$$

- $event_L \triangleq \bigcup_{l \in L} event_l$, d.h. bei gleichzeitiger Ausführung aller Anweisungen ist die Menge der erzeugten Ereignisse gerade die Vereinigung über die Mengen der von den einzelnen Anweisungen erzeugten Ereignisse.
- $var_L \triangleq \bigcup_{l \in L} var_l$, d.h. bei gleichzeitiger Ausführung aller Anweisungen ist die Menge der modifizierten Variablen gerade die Vereinigung über die Mengen der von den einzelnen Anweisungen modifizierten Variablen.
- $op_L \triangleq \bigcap_{l \in L} op_l$, d.h. Zustandsübergänge für die gleichzeitige Ausführung aller Anweisungen sind gerade die, die für jede der Anweisungen in der Menge ihrer Zustandsübergänge vorhanden sind.

Ein abstraktes Transitionssystem für ein Statechart SC ist definiert als ein Tripel

$$TS_{SC} \triangleq (S_{SC}, Init_{SC}, \delta_{SC})$$

[2] Man kann bei Bedarf S_0 auch für Initialwerte von Variablen einschränken.

- S_{SC} ist der Gesamtzustandsraum von SC; er ergibt sich als Kreuzprodukt aus dem Grundzustandsraum und dem Variablen- und Ereigniszustandsraum:

$$S_{SC} \triangleq B_{SC} \times S_{XE}$$

- $Init_{SC} \subseteq S_{SC}$ sind die initialen Zustände; sie ergeben sich durch Ausführung der Initialtransition für SC in einem Ausgangszustand aus S_0. Ereignisse, die in der Initialtransition nicht explizit generiert werden, sollen in den Initialzuständen nicht vorliegen. Zusätzlich fordern wir, dass Variablen, die nicht explizit modifiziert werden, ihre Werte aus dem Ausgangszustand beibehalten.

$$\begin{aligned} Init_{SC} \triangleq \{\ & (b,s) \mid \exists\, L \subseteq L_{SC} \,:\, (I_{SC}, L, b) \in \Delta_{SC} \ \wedge\ L \neq \emptyset \ \wedge \\ & \exists\, s_0 \in S_0 \,:\, (s_0, s) \in op_L \ \wedge \\ & \qquad \forall\, (x \in (X \setminus var_L)) \,:\, x(s) = x(s_0) \ \wedge \\ & \qquad \forall\, (e \in (E \setminus events_L)) \,:\, e(s) = \mathit{false}\ \} \end{aligned}$$

- $\delta_{SC} \subseteq S_{SC} \times S_{SC}$ ist die Zustandsübergangsrelation. Die Zustandsübergänge wirken sich auf dem Grundzustandsraum entsprechend der markierten Zustandsübergänge (Δ_{SC}) aus. Der Zustandsübergang auf dem Variablen- und Ereignis-Zustandsraum ergibt sich aus den Anweisungen des zugehörigen markierten Zustandsübergangs. Dabei muss berücksichtigt werden, dass Variablen, die in den Anweisungen nicht explizit auftreten, ihre Werte beibehalten; Ereignisse, die in den Anweisungen nicht explizit generiert werden, sollen nach dem Zustandsübergang nicht vorliegen. Mit $L \neq \emptyset$ erzwingen wir, dass jeder Übergang zumindest einer Transition aus dem zugrundegelegten Statechart entspricht.

$$\begin{aligned} \delta_{SC} \triangleq \{\ & ((b_1,s_1),(b_2,s_2)) \mid \exists\, L \subseteq L_{SC} \,: \\ & (b_1, L, b_2) \in \Delta_{SC} \ \wedge\ (s_1, s_2) \in op_L \ \wedge \\ & \forall\, (x \in (X \setminus var_L)) \,:\, x(s_2) = x(s_1) \ \wedge \\ & \forall\, (e \in (E \setminus events_L)) \,:\, e(s_2) = \mathit{false} \ \wedge \\ & L \neq \emptyset\ \} \end{aligned}$$

Das hier entwickelte Transitionssystem unterscheidet sich (zumindest) in zwei Punkten von der in [HN96] beschriebenen Semantik: Die *greedy* Eigenschaft wurde für die Definition der Zustandsübergangsrelation nicht übernommen, da sie die Modularität des strukturellen Aufbaus stört: Mit der *greedy*-Semantik kann ein Statechart in einer Umgebung ein Verhalten aufweisen, das nicht auftritt, wenn man den Statechart für sich betrachtet[3]. Die höhere Priorität von Transitionen eines Statecharts gegenüber den Transitionen seiner Subcharts haben wir nicht berücksichtigt[4].

3.3 Darstellung von MSCs in CTL*

Ausdrücke in MSC_{CTL} entsprechen temporallogischen Formeln. Wir stellen dies dar, indem wir diese Ausdrücke in CTL* [Eme90] wiedergeben: Die Ereignisse in einer Instanz innerhalb eines MSC sollen in der Reihenfolge ihrer Aufzählung eintreten; zwischen zwei solchen Ereignissen können nur Ereignisse vorkommen, die nicht explizit in der Instanz auftreten.

Für eine Instanz I sei vis_I die Menge der in I sichtbaren Ereignisse. vis_I enthält somit Ereignisse, die als Aktionen in der Form "`action a:I`" oder als Ereignisse für

[3] Lt. [HN96, pg. 321] ist es legitim, die Semantik ohne *greedy*-Eigenschaft zu definieren.
[4] Lt. [HN96, pg. 328] soll die aktuelle Prioritätsregel geändert werden

das Empfangen und Versenden von Nachrichten in der Form "`message m:I,...`" bzw. "`message m:...,I`" deklariert wurden.

Zum Zweck der Umsetzung nach CTL* modellieren wir Ereignisse durch boole'sche Variablen, die zustandsabhängig die Werte wahr oder falsch annehmen können. Zur besseren Lesbarkeit definieren wir noch weitere Prädikate: Mit dem Prädikat $none_I \triangleq \bigwedge_{e \in vis_I} \neg e$ beschreiben wir die Bedingung, dass keines der Ereignisse aus vis_I vorliegt und für ein $e \in vis_I$ beschreiben wir mit dem Prädikat $just_I(e) \triangleq e \wedge \bigwedge_{f \in (vis_I \setminus \{e\})} \neg f$ die Bedingung, dass aus vis_I genau Ereignis e vorliegt.

Wird in I spezifiziert, dass e_1 von e_2 gefolgt sein soll, entspricht das der Formel $just_I(e_1) \wedge \mathsf{X}\,(none_I\ \mathsf{U}\ just_I(e_2))$. Die Formel besagt, dass zunächst genau e_1 und ab dem fogenden Zustand irgendwann genau e_2 vorliegen soll, ohne dass dazwischen ein Ereignis aus vis_I auftritt. Bedingungen behandeln wir in einer etwas anderen Weise: Wird in I spezifiziert, dass Bedingung c auf Ereignis e folgt, so entspricht das der Formel $just_I(e) \wedge (c \vee \mathsf{X}\,((none_I \wedge \neg c)\ \mathsf{U}\ (none_I \wedge c)))$. Die Formel besagt, dass zunächst genau e vorliegen und im gleichen oder einem späteren Zustand Bedingung c erfüllt sein soll, ohne dass dazwischen ein Ereigniss aus vis_I auftritt.

Wir Verallgemeinern nun dieses Prinzip und geben ein allgemeines Schema für die Umsetzung von Instanzen an. Wir definieren zunächst Schemata ($ins2ctl_I$) zur Umsetzung von Sequenzen die der syntaktischen Regel `<inst body>` entsprechen in die entsprechenden temporallogischen Formeln; das einzusetzende Schema hängt ab von der Instanz I in der die Sequenz auftritt:

$$
\begin{array}{lcl}
seq2ctl_I(\epsilon) & \triangleq & true \\
seq2ctl_I(\texttt{in}\ m\ \texttt{from}\ i;\sigma) & \triangleq & \mathsf{X}\,(none_I\ \mathsf{U}\ (just_I(m_{in})\ \wedge\ seq2ctl_I(\sigma))) \\
seq2ctl_I(\texttt{out}\ m\ \texttt{to}\ i;\sigma) & \triangleq & \mathsf{X}\,(none_I\ \mathsf{U}\ (just_I(m_{out})\ \wedge\ seq2ctl_I(\sigma))) \\
seq2ctl_I(\texttt{action}\ a;\sigma) & \triangleq & \mathsf{X}\,(none_I\ \mathsf{U}\ (just_I(a)\ \wedge\ seq2ctl_I(\sigma))) \\
seq2ctl_I(\texttt{cond}\ c;\sigma) & \triangleq & (c\ \wedge\ seq2ctl(\sigma)) \vee \\
 & & \quad \mathsf{X}\,((none_I \wedge \neg c)\ \mathsf{U}\ (none_I\ \wedge\ c\ \wedge\ seq2ctl_I(\sigma)))
\end{array}
$$

Hierbei bezeichnet ϵ die leere Sequenz und σ den Rest einer Sequenz, der seinerseits wieder der syntaktischen Regel `<inst body>` entspricht. Das Schema ($ins2ctl$) zur Umsetzung einer Instanz I baut auf $seq2ctl_I$ auf:

$$ins2ctl(\texttt{instance}\ I;\sigma\ \texttt{endinstance}) \triangleq seq2ctl_I(\sigma)$$

Ein MSC wird umgesetzt, indem die Formeln für die Instanzen miteinander konjugiert werden. Für den Inhalt mb einer MSC-Definition (`msc` M mb `endmsc;`) legen wir als Schema für die Umsetzung fest

$$
mb2ctl(mb) \triangleq \begin{cases}
ins2ctl(mb) & : mb \text{ entsprechend Regel } \texttt{<inst def>} \\
ins2ctl(ins)\ \wedge\ mb2ctl(mb2) & : mb = ins;mb2 \text{ mit } ins \text{ entsprechend Regel } \texttt{<inst def>} \text{ und } mb2 \text{ entsprechend Regel } \texttt{<msc body>}
\end{cases}
$$

Weiterhin definieren wir $msc2ctl_M \triangleq mb2ctl(mb)$ als die temporallogische Formel, die dem MSC M entspricht. Die einfachen MSC-Formeln, die der Syntax `<msc form>` entsprechen werden durch ein einfaches Schema umgesetzt: Negationen werden nach Negationen, Konjunktionen nach Konjunktionen und Disjunktionen

nach Disjunktionen übersetzt:

$$
\begin{array}{lcl}
mf2ctl(M) & \triangleq & msc2ctl_M \\
mf2ctl(\texttt{not } mf) & \triangleq & \neg\, mf2ctl(mf) \\
mf2ctl(mf_1 \texttt{ and } mf_2) & \triangleq & mf2ctl(mf_1) \ \wedge \ mf2ctl(mf_2) \\
mf2ctl(mf_1 \texttt{ or } mf_2) & \triangleq & mf2ctl(mf_1) \ \vee \ mf2ctl(mf_2)
\end{array}
$$

Die in MSC_{CTL}-Formeln auftretenden Pfadquantoren (A, E) und temporallogischen Operatoren (X, F, G, U) sowie die logischen Verknüpfungen bilden wir identisch nach CTL* ab. Der MSC_{CTL} Ausdruck `(E F E example)` für den in Abbildung 2.4 gegebenen MSC `example` entspricht somit der CTL*-Formel

$$
\begin{array}{l}
\mathsf{EFE}(\ \mathsf{X}(none_{I1}\,\mathsf{U}(just_{I1}(m1_{in})\wedge \\
\quad \mathsf{X}(none_{I1}\,\mathsf{U}(just_{I1}(a)\wedge \\
\quad\quad \mathsf{X}(none_{I1}\,\mathsf{U}(just_{I1}(m2_{out})\wedge \\
\quad\quad\quad ((c\wedge \mathsf{X}(none_{I1}\,\mathsf{U}just_{I1}(m4_{in})))\vee \\
\quad\quad\quad\quad \mathsf{X}((none_{I1}\wedge\neg c)\,\mathsf{U}(none_{I1}\wedge c\wedge \mathsf{X}(none_{I1}\,\mathsf{U}just_{I1}(m4_{in})))))))))) \\
\quad \wedge \\
\quad \mathsf{X}(none_{I2}\,\mathsf{U}(just_{I2}(b)\wedge \\
\quad\quad \mathsf{X}(none_{I2}\,\mathsf{U}(just_{I2}(m2_{in})\wedge \\
\quad\quad\quad \mathsf{X}(none_{I2}\,\mathsf{U}(just_{I2}(m3_{out})\wedge \\
\quad\quad\quad\quad \mathsf{X}(none_{I2}\,\mathsf{U}just_{I2}(m4_{out})))))))))
\end{array}
$$

3.4 Konsistenz zwischen Statechart- und MSC- Spezifikationen

Statecharts und MSCs werden syntaktisch durch die Form

$$SC \ \texttt{satisfies} \ MSC$$

in Relation zueinander gesetzt. Die intendierte Bedeutung dahinter ist, dass das durch SC definierte System die Spezifikation MSC erfüllen soll. SC erfüllt die Spezifikation MSC genau dann, wenn das zugehörige Transitionssystem TS_{SC} die aus MSC abgeleitete CTL*-Formel msc_{CTL} erfüllt; dabei ist die Erfülltheit von msc_{CTL} über TS_{SC} in der herkömmlich für CTL* definierten Weise erklärt; vgl. [Eme90].

Diese Definition ermöglicht es, bereits existierende Verfahren und Werkzeuge zur Prüfung von CTL*-Formeln über Transitionssystemen für die Analyse von MSC Spezifikationen über Statecharts einzusetzen. Insbesondere können auf diese Weise auch Model-Checker verwendet werden.

Kapitel 4

Einsatz eines Model-Checkers

In diesem Kapitel beschreiben wir einen Ansatz zur Analyse der Konsistenz zwischen Statechart- und MSC-Spezifikationen durch Einsatz eines Model-Checkers. Dazu muss die Erfülltheit von CTL*-Formeln, die der MSC-Spezifikation entsprechen über einem von der Statechart-Spezifikation abgeleiteten Transitionssystem geprüft werden. Allerdings gibt es nur eine verhältnismässig kleine Anzahl von Werkzeugen, die das ermöglichen. Da wir uns noch in einem experimentellen Stadium unserer Arbeiten befinden, wollen wir zunächst nur die prinzipielle Durchführbarkeit unserer Arbeiten untersuchen. Daher haben wir uns entschieden, einen Model-Checker für den μ-Kalkül einzusetzen und einen Konverter zu verwenden, der CTL*-Formeln in den μ-Kalkül umsetzt[1]. Sowohl ein Model-Checker für den μ-Kalkül als auch ein Konverter für CTL*-Formeln steht mit dem *mucke*-System zur Verfügung [Bie97, Ref96]; wir Beschreiben nachfolgend die Abbildung unserer Formalisierungen in die Eingabesprache von *mucke*.

Die Möglichkeiten zur Steigerung der Effizienz wollen wir erst zu einem späteren Zeitpunkt untersuchen. Potentiell kommen dafür die Verfeinerung unserer Kodierung bzw. der Einsatz anderer Werkzeuge in Betracht.

4.1 Statecharts in mucke

Transitionssysteme werden in mucke durch die Festlegung von Zustandsraum und Zustandsübergangsrelation beschrieben. Um Zustandsräume zu modellieren, die aus vielen Komponenten bestehen, bietet mucke die Möglichkeit Verbundtypen (Records) hierarchisch strukturiert zu definieren. Daneben können (endliche) Aufzählungs-, Bereichs- sowie Vektortypen definiert werden. Der Typ bool ist vordefiniert. Die Zustandsübergangsrelation wird als zweistellige boolesche Funktion über dem Zustandsraum modelliert. Diese Funktion kann schichtenweise aus Funktionen für Zustandsübergänge in den Zunstandskomponenten aufgebaut werden; dies kann man ausnutzen, um die Zustandsübergangsrelation für einen Statechart in Termini der Zustandsübergangsrelationen für seine Subcharts auszudrücken.

In den Abschnitten 3.1 und 3.2 beschreiben wir die Umsetzung von Statecharts in ein Transitionssystem. Darauf aufbauend definieren wir hier die Darstellung von Statecharts als Transitionssysteme in mucke. Für einen Statechart SC seien LTS_{SC} bzw. TS_{SC} das zugehörige markierte bzw. vollständige Transitionssystem.

[1] Da der μ-Kalkül ausdrucksstärker als CTL* ist, ist diese Umsetzung immer möglich; jedoch kann sie bisweilen so komplex sein, dass dieser Ansatz unpraktikabel wird

Weiterhin gehen wir davon aus, dass das Gesamtsystem mit dem Statechart Q und je einer endliche Menge von Variablen $X = \{x_1, \ldots, x_k\}$ und Ereignissen $E = \{e_1, \ldots, e_p\}$ spezifiziert ist. Der Wertebereich einer Variablen x sei so gegeben, dass W_x in mucke als Typname für eine Deklaration dienen kann.

Der Zustandsraum eines Statecharts setzt sich aus den Grundzuständen der in ihm enthaltenen Statecharts und dem Variablen- und Ereigniszustandsraum zusammen. Die Zustandsräume werden durch entsprechende Typdefinitionen in mucke modelliert. Für die Grundzustandsräume der Statecharts definieren wir folgende Typen:

- Ist SC ein einfacher Statechart und somit $B_{SC} = bool$, definieren wir den Typ

  ```
  class B_SC { bool activity; }
  ```

- Ist SC ein AND-Statechart und somit $B_{SC} = \times_{i=1}^{n} B_{Q_i}$, definieren wir den Typ

  ```
  class B_SC { B_Q1 q1; ...; B_Qn qn; }
  ```

- Ist $SC = (\bigvee_{i=1}^{n} Q_i, Tr_{SC})$ ein OR-Statechart, womit dann auch gilt $B_{SC} = (\times_{i=1}^{n} B_{Q_i}) \times \{\text{“void”}, \text{“}\mathsf{Q}_1\text{”}, \ldots, \text{“}\mathsf{Q}_n\text{”}\}$, und ist die Menge der Transitionen $Tr_{SC} = \{(\emptyset, init_{SC}, tgt_0), (src_1, l_1, tgt_1), \ldots, (src_m, l_m, tgt_m)\}$, definieren wir für die Historien den Typ

  ```
  enum history_SC { void_SC, Q1, ..., Qn }
  ```

  für die Markierungen den Typ

  ```
  enum label_SC { nil_SC, init_SC, l1, ..., lm }
  ```

  und für den Grundzustandsraum den Typ

  ```
  class B_SC { B_Q1 q1; ...; B_Qn qn;
               history_SC history; label_SC label; }
  ```

Für das Gesamtsystem werden noch folgende Typen definiert:

- Der Variablen- und Ereigniszustandsraum S_{XE} wird mit einem Verbundtyp modelliert, der als Komponenten sämtliche Variablen $x \in X$ und Ereignisse $e \in E$ enthält. Variablen x werden mit den ihnen zugeordneten Typen W_x und Ereignisse mit boole'schem Typ deklariert.

  ```
  class S_XE { W_x1 x1; ...; W_xk xk; bool e1; ... bool ep; }
  ```

- Für den Gesamtzustandsraum S_Q eines Systems, das mit dem Statechart Q, Variablen X und Ereignissen E spezifiziert ist ergibt sich der Typ

  ```
  class State { B_Q b; S_XE xe; }
  ```

Um die Zustandsübergangsrelation festzulegen definieren wir in mucke für jeden Statechart SC folgende Gruppe von Prädikaten:

`active_`SC`(s)` beschreibt den Aktivierungszustand des Statechart im Grundzustand `s`. Es charakterisiert die Menge A_{SC}.

`I_`SC`(s)` charakterisiert den Grundzustand I_{SC}.

`Delta_`SC`(`s_1`,`s_2`)` charakterisiert die Menge Δ_{SC}.

Insbesondere Δ_{SC} haben wir in Abschnitt 3.1 mit Hilfe von Funktionen $activate_{SC}$ und $deactivate_{SC}$ definiert, die ganz allgemein auf Konfigurationen angewendet werden können. Die Konfigurationen stammen aus den Ausgangs- und Zielpunkten der Transitionen. Anstatt diese Funktionen direkt in mucke nachzubilden, definieren wir Übersetzungsregeln $a2m$ und $d2m$ mit denen wir an den benötigten Stellen mucke-Quelltexte für $activate_{SC}(tgt)$ und $deactivate_{SC}(src)$ generieren können. Die genauen Definitionen von $a2m$ und $d2m$ befinden sich in Anhang C. `active_`SC, `I_`SC und `Delta_`SC definieren wir analog zu Abschnitt 3.1:

- Ist SC ein einfacher Statechart, so definieren wir in mucke

```
bool active_SC(B_SC s)
     (s.activity);

bool I_SC(B_SC s)
     (!s.activity);

bool Delta_SC(B_SC s1, B_SC s2)
     (s1.activity & s2.activity);
```

- Ist $SC = \bigwedge_{i=1}^{n} Q_i$ ein AND-Statechart, so definieren wir in mucke

```
bool active_SC(B_SC s)
     (active_Q1(s.q1) & ... & active_Qn(s.qn));

bool I_SC(B_SC s)
     (I_Q1(s.q1) & ... & I_Qn(s.qn));

bool Delta_SC(B_SC s1, B_SC s2)
     (Delta_Q1(s1.q1,s2.q1) & ... & Delta_Qn(s1.qn,s2.qn));
```

- Ist $SC = (\bigvee_{i=1}^{n} Q_i, Tr_{SC})$ ein OR-Statechart mit den Transitionen $Tr_{SC} = \{(\emptyset, init_{SC}, t_0)(src_1, l_1, tgt_1), \ldots, (src_m, l_m, tgt_m)\}$, so definieren wir in mucke

```
bool active_SC(B_SC s)
     (active_Q1(s.q1) | ... | active_Qn(s.qn));

bool I_SC(B_SC s)
     (I_Q1(s.q1) & ... & I_Qn(s.qn) &
      s.history=void_SC & s.label=nil_SC);

bool Delta_SC(B_SC s1, B_SC s2)
     ((s1.history=s2.history & s2.label=nil_SC &
       ((active_Q1(s1.q1) & Delta_(s1.q1,s2.q1) &
         s1.q2=s2.q2 & ... & s1.qn=s2.qn) |
        ... |
        (active_Qn(s1.qn) & Delta_Qn(s1.qn,s2.qn) &
         s1.q1=s2.q1 & ... & s1.qn-1=s2.qn-1)))
```

```
|
(I_SC(s1) & a2m(SC,{SC},s1,s2))
|
(exists B_SC b1 . exists B_SC b2 .
 d2m(SC,{src_1},s1,b1) &
 b1.q_1=b2.q_1 & ... & b1.q_n=b2.q_n &
 b1.history=b2.history & b2.label=l_1 &
 a2m(SC,{tgt_1},b2,s2)) |
 ... |
(exists B_SC b1 . exists B_SC b2 .
 d2m(SC,{src_n},s1,b1) &
 b1.q_1=b2.q_1 & ... & b1.q_n=b2.q_n &
 b1.history=b2.history & b2.label=l_n &
 a2m(SC,{tgt_n},b2,s2)));
```

Für jede Markierung $l \in L_Q$ definieren wir ein Prädikat `op_`l, das für die zugehörige Anweisung die Menge der Zustandsübergänge op_l aus $[\![l]\!]$ charakterisiert; in Anhang C definieren wir eine Regel $l2m$ zur Generierung von mucke-Quelltext für Anweisungen l.

```
bool op_l(S_XE xe1, S_XE xe2)
     (l2m(l,xe1,xe2));
```

Ein Zustandsübergang aus op_l tritt nur gemeinsam mit einer Transition ein, die die Markierung l trägt. Aus einem Grundzustand kann man ermitteln, welche Transitionsitionen zu diesem Grundzustand geführt haben. In Anhang C definieren wir uns eine weitere Übersetzungsregel $testl2m$, die mucke-Quelltext generiert, um zu prüfen, ob eine Transition mit Markierung l vorliegt. Wenn keine Transition eintritt, so dass aufgrund der mit ihr assoziierten Anweisungen eine bestimmte Variable x modifiziert bzw. ein bestimmtes Ereignis e erzeugt wird, behält x seinen Wert bei bzw. liegt e nicht vor. Die Menge der Markierungen $mark(x)$ bzw. $mark(e)$, mit denen die Modifikation der Variablen x bzw. die Erzeugung des Ereignisses e einhergeht, werden definiert durch:

- $mark(x) \triangleq \{l \in L_Q \mid x \in var_l\}$
- $mark(e) \triangleq \{l \in L_Q \mid e \in event_l\}$

Nun können wir in mucke ein Prädikat definieren, mit dem die Zustandsübergänge für Anweisungen charakterisiert sind:

```
bool OP(S_XE xe1, B_Q b, S_XE xe2)
     ( // folgenden Implikation für jede Markierung l
       (testl2m(Q,l,b) -> op_l(xe1,xe2)) &
       ... &
       // folgenden Implikation für jede Variable x
       ( // folgende Bedingung für jede Markierung l ∈ mark(x)
         !(testl2m(Q,l,b)) &
         ...
         -> xe1.x=xe2.x ) &
       ... &
       // folgenden Implikation für jedes Ereignis e
       ( // folgende Bedingung für jede Markierung l ∈ mark(e)
         !(testl2m(Q,l,b)) &
         ...
```

```
        -> xe2.e=false ) &
      ...
    );
```

In Ausgangszuständen S_0 sollen keine Ereignisse vorliegen. Die Menge S_0 charakterisieren wir in mucke mit dem Prädikat

```
bool S_0(S_XE xe)
     (!xe.e_1 & ... & !xe.e_p);
```

Für die Zustandsübergänge hatten wir in der Semantik gefordert das zumindest eine der Markierungen auftritt. Ist $L_Q = \{l_1, \ldots, l_r\}$ die Gesamtmenge der Markierungen, so definieren wir das Prädikat

```
bool Progress(B_Q b)
     ( testl2m(Q, l_1,b) | ... | testl2m(Q, l_1,b) );
```

Nach diesen Vorbereitungen können wir das vollständige Transtitionssystem in mucke angeben. Die Menge der Initialzustände $Init_Q$ charakterisieren wir in mucke mit dem Prädikat `Init_Q`.

```
bool Init_Q(State s)
     ((exists B_Q b . I_Q(b) & Delta_Q(b,s.b)) &
      (exists S_XE xe0 . S_0(xe0) & OP(xe0,s.b,s.xe)) &
      Progress(s.b));
```

Die Zustandsübergangsrelation δ_Q für das Gesamtsystem definieren wir in mucke mit dem Prädikat `Trans`.

```
bool Trans(State s1, State s2)
     (Delta_Q(s1.b,s2.b) &
      OP(s1.xe,s2.b,s2.xe) &
      Progress(s2.b));
```

4.2 CTL*-Formeln für MSCs in mucke

Mit einem Tool, das man in Verbindung mit mucke einsetzen kann, lassen sich CTL*-Formeln in den μ-Kalkül umwandeln [Ref96]. Das Verfahren der Umsetzung erfolgt nach einer verbesserten Version des Algorithmus von Dam [Dam94, Dam90].

Ausgangspunkt für die Umsetzung sind CTL*-Formeln für MSC_{CTL} die entsprechend der Beschreibung in Abschnitt 3.3 gewonnen werden. Für die in den Formeln auftretenden Ausdrücke $none_I$ und $just_I(e)$ definieren wir mucke-Prädikate; ist $vis_I = \{v_1, \ldots, v_r\}$ die Menge der in I sichtbaren Ereignisse, definieren wir

```
bool none_I(State s)
     (!s.xe.v_1 & ... & !s.xe.v_r);
```

Für jedes v_i definieren wir ein Prädikat

```
bool just_Iv_i(State s)
     (!s.xe.v_1 & ... & !s.xe.v_{i-1} &
      s.xe.v_i &
      !s.xe.v_{i+1} & ... & !s.xe.v_r);
```

Das Werkzeug geht bei der Umsetzung davon aus, dass der Zustandsraum des zugrundeliegenden Transitionssystems in mucke mit dem Typen `State` gegeben ist und dass die Zustandsübergänge mit einem Prädikat charakterisiert werden, dessen Name `Trans` lautet (vgl. mit vorangegangenem Abschnitt). Das Ergebnis der Umsetzung ist ein Prädikat mit Namen `spec`. Um den Model-Checker darauf anzusetzen, muss noch ein Beweisziel formuliert werden, das im Wesentlichen die Anfangszustände festlegt:

```
forall State s . Init_Q(s) -> spec(s)
```

Kapitel 5

Beispiel

Zur Verdeutlichung unseres Ansatzes modellieren wir hier einen kleinen Ausschnitt aus einem Bahnübergang. Der Bahnübergang besteht aus einigen Komponenten, die parallel zueinender ablaufen und miteinander kommunizieren:

```
main sc uebergang: and
  sc licht
  sc schranke
  sc funkmodul
  sc steuerung
endsc uebergang
```

Bei der Interaktion dieser Komponenten werden Nachrichten von der Steuerung an die Licht- und Schrankenkomponente verschickt; das Funkmodul leitet Anforderungen und Freigaben vom Zug an die Steuerung weiter:

```
message gelb_an: steuerung,licht
message rot_aus: steuerung,licht
message oeffnen: steuerung,schranke
message schliessen: steuerung,schranke
message anforderung: funkmodul,steuerung
message zugfreigabe: funkmodul,steuerung
```

Das Verhalten der einzelnen Komponenten ist jeweils als Zustandsübergangssystem mit Statecharts beschrieben. Die Komponente `licht` ist zunächst ausgeschaltet; bei Empfang der Nachricht `gelb_an` schaltet sie gelb ein und schaltet danach selbstätig um nach `rot`. Mit Empfang der Nachricht `rot_aus` wird das licht wieder ausgeschalten.

```
sc licht: or
  sc licht_aus: basic
  sc gelb: basic
  sc rot: basic
  init: [licht_aus]
  tr ein: in(gelb_an) [licht_aus -> gelb]
  tr um: [gelb -> rot]
  tr aus: in(rot_aus) [rot -> licht_aus]
endsc licht
```

Die Komponente `schranke` ist zunächst oben; mit Nachricht `schliessen` bewegt sich die Schranke nach unten und mit Nachricht `oeffnen` bewegt sich die Schranke nach oben.

```
  sc schranke: or
    sc oben: basic
    sc unten: basic
    init: [oben]
    tr zu: in(schliessen) [oben -> unten]
    tr auf: in(oeffnen) [unten -> oben]
  endsc schranke
```

Die Komponente `funkmodul` ist zunächst im Bereitschaftsmodus; aus diesem Modus kann sie die Anforderung eines Zuges an die Steuerung weiterleiten. Für das Funkmodul gilt dann der Bahnübergang als besetzt, bis vom Zug eine Freigabe erfolgt. Diese Nachricht wird auch wieder an die Steuerung weitergeleitet.

```
  sc funkmodul: or
    sc bereit: basic
    sc besetzt: basic
    init: [bereit]
    tr start: out(anforderung) [bereit -> besetzt]
    tr stop: out(zugfreigabe) [besetzt -> bereit]
  endsc funkmodul
```

Die Komponente `steuerung` Koordiniert die Aktionen der Komponenten `licht` und `schranke`. Auf eine Anforderung hin wird gelbes Licht eingeschalten, danach die Schranke gesperrt. Wenn vom Zug eine Freigabe erfolgt, wird die Schranke wieder geöffnet und das Licht ausgeschalten.

```
  sc steuerung: or
    sc frei: basic
    sc reserviert: basic
    sc gesperrt: basic
    sc geschlossen: basic
    sc belegt: basic
    sc ueberquert: basic
    sc geoeffnet: basic
    init: [frei]
    tr empfangen: in(anforderung) [frei -> reserviert]
    tr licht_ein: out(gelb_an) [reserviert -> gesperrt]
    tr schranke_zu: out(schliessen) [gesperrt -> geschlossen]
    tr fahrt_frei: [geschlossen -> belegt]
    tr befahrung: in(zugfreigabe) [belegt -> ueberquert]
    tr schranke_auf: out(oeffnen) [ueberquert -> geoeffnet]
    tr licht_aus: out(rot_aus); [geoeffnet -> frei]
  endsc steuerung
```

Die Anforderungen an den Bahnübergang formulieren wir in Termini von MSC_{CTL}. Ein Beispiel für ein erwünschtes Szenario ist

```
msc szenario1
  instance licht;
    in gelb_an from steuerung;
    in rot_aus from steuerung;
  endinstance;
  instance schranke;
    in schliessen from steuerung;
    in oeffnen from steuerung;
  endinstance;
```

```
  instance steuerung;
    in anforderung from env;
    out gelb_an to licht;
    out schliessen to schranke;
    in zugfreigabe from env;
    out oeffnen to schranke;
    out rot_aus to licht;
  endinstance;
endmsc;
```

Als unerwünschtes Szenario definieren wir einen Ablauf, in dem die Schranke geschlossen und wieder geöffnet wird, ohne dass dazwischen vom Zug eine Freigabe eingeht:

```
  msc szenario2
    instance schranke;
      in schliessen from steuerung;
      in oeffnen from steuerung;
    endinstance;
    instance steuerung;
      out schliessen to schranke;
      out oeffnen to schranke;
    endinstance;
  endmsc;
```

Konkret wollen wir fordern, dass zum einen jedesmal, wenn vom Zug eine Anforderung vorliegt, danach eine Ausführung existiert so dass `szenario1` wahr wird; dazu wollen wir auch prüfen ob denn auch immer wieder eine Anforderung vorliegen kann. Zum anderen fordern wir, dass kein Ausführungspfad jemals `szenario2` efüllt. In unserem Ansatz formulieren wir diese Anforderungen in der Form

```
  sc steuerung...endsc steuerung
  msc szenario1...endmsc
  msc szenario2...endmsc
  satisfies
  spec: A G (not anforderung or E F E szenario1)
  spec: A G E F anforderung
  spec: not E F E szenario2
```

Dieses Beispiel haben wir nach mucke umgesetz und zur Plausibilitätsprüfung weitere Anforderungen definiert und mit dem Model-Checker mucke analysiert. Da die Umsetzung der Statecharts in die Eingabesprache von mucke manuell erfolgt ist, war es wichtig, solche Plausibilitätsprüfungen (z.B. die Nicht-Existenz von Deadlocks) durchzuführen. Dabei konnten auch manche Fehler erkannt und beseitigt werden.

Allerdings stösst schon bei diesen in MSC_{CTL} formulierten Anforderungen der Übersetzer von CTL* nach μ-Kalkül [Ref96] an seine Grenzen (lediglich `szenario2` konnte umgesetzt werden). Dieses Ergebnis belegt, dass es notwendig ist, die Umsetzung der Formeln zu optimieren. Ansätze hierfür bieten sich aus der eingeschränkten syntaktischen Gestalt der CTL*-Formeln für die MSCs. In Anhang D beschreiben wir ein Verfahren, wie aus den von MSC_{CTL} abgeleiteten CTL*-Formeln äquivalente CTL-Formeln gewonnen werden können; dafür haben wir ein prototypisches Werkzeug entwickelt, das zunächst entsprechen der im Anhang beschriebenen Regeln CTL-Formeln erzeugt hat. Da die so erzeugten Formeln zu gross[1] wurden, um

[1] Als ASCII-Text in der Grössenordnung von mehreren MB

sie mit mucke bearbeiten zu können, haben wir unser Werkzeug so erweitert, dass es direkt MSC_{CTL} einliesst, und die in den MSCs vorgegebene partielle Ordnung auf den Ereignissen dazu verwendet, kürzere Beweisziele zu erzeugen. Damit konnten wir auch die Beweisverpflichtung für `szenario1` prüfen. Wir haben noch weitere Anforderungen, die wir in MSC_{CTL} formuliert haben, geprüft und dabei festgestellt, dass das oben präsentierte Statechart-Modell des Bahnübergangs noch einige Mängel aufweist. Da uns bisher noch keine Werkzeugunterstützung zur Verfügung steht, um Statechart in ein Transitionssystem abzubilden, und uns andererseits die Erfahrungen mit diesem Beispiel die prinzipielle Durchführbarkeit demonstriert haben, haben wir zum jetzigen Zeitpunkt auf weitere Untersuchungen an diesem Beispiel verzichtet.

Kapitel 6

Zusammenfassung und Ausblick

Wir haben in diesem Papier einen Ansatz zur Prüfung von MSCs über Statecharts beschrieben. Dabei verfolgen wir das Ziel, Zustandsübergangssysteme mit Statecharts zu beschreiben und die MSCs lediglich zur Beschreibung von Ausführungssequenzen heranzuziehen, die aus den zugehörigen Zustandsautomaten ableitbar sein sollen. Über den Ausführungspfaden des zugrundegelegten Systems kann in unserem Ansatz, ähnlich zu LSCs, universell und existentiell quantifiziert werden. Jedoch im Gegensatz zu LSCs betrachen wir für komplexere Eigenschaften Kombinationen über MSCs, die sich in natürlicher Weise als Konjunktion, Disjunktion und Negation ausdrücken lassen. Diese Ausdrücke können wir dann im Stil von CTL-Formeln zu komplexeren Aussagen kombinieren. Mit einer Umsetzung der Beweisverpflichtungen in einen Model-Checker bietet sich die Möglichkeit, die Prüfung zu automatisieren.

Diesen Ansatz haben wir exemplarisch mit obigem Beispiel durchgespielt. Allerdings haben wir bisher aufgrund noch fehlender Werkzeugunterstützung wenig Erfahrungen zur Praktikabilität sammeln können; die Umsetzung von CTL*-Formeln in den μ-Kalkül hat sich als problematisch erwiesen. Daher haben wir ein Verfahren entwickelt (Anhang D) um die aus MSC_{CTL} erzeugten Beweisverpflichtungen nach CTL umzusetzen. Die Gültigkeit der in Anhang D beschriebenen Umsetzungsregeln haben wir mit dem PVS-System[COR+95] nachgewiesen. Unter Ausnutzung der partiellen Ordnung über den Ereignissen ist es uns gelungen, die Beweisverpflichtungen in ausreichend kompakter Form nach CTL abzubilden. Dieses Verfahren haben wir in einem prototypischen Werkzeug implementiert, um die in MSC_{CTL} formulierten Anforderungen nach CTL und in den μ-Kalkül umzusetzen. Der Model-Checker mucke kommt mit den (bisher analysierten) Beweiszielen gut zurecht. Mit der Umsetzung nach CTL eröffnet sich allerdings auch ein Weg zum Einsatz von hoch-performanten CTL-Model-Checkern, wie zum Beispiel dem SMV-System [McM92, McM93].

Da Statecharts hierarchisch strukturiert sind, könnte man versuchen, diese Struktur für eine Modularisierung der Analyse durch Model-Checking auszunutzen und einen für Model-Checking geeigneten Abstraktionsbegriff speziell für Statecharts zu entwickeln. An dieser Stelle würde es sich auch Anbieten, diesen Ansatz in dem PVS-System[ORRS96, Sha96] zu modellieren, insbesondere da hier bereits ein Model-Checker für den μ-Kalkül und Ansätze zur Abstraktion vorhanden sind. Die Untersuchung dieser Möglichkeiten wollen wir zu einem späteren Zeitpunkt angehen.

Anhang A

Syntax

Wir fassen in diesem Abschnitt die Gesamtsyntax zur Beschreibung von Statecharts, MSC_{CTL}-Formeln und der hier definierten Konsistenzrelation zusammen.

Eine Spezifikation beschreibt Statecharts und MSCs deren Konsistenz miteinander überprüft werden soll. Sie kann optional einen Vorspann enthalten; wir nutzen dessen Inhalt um die Semantik von Zuweisungsperationen und Prädiken als auch Typen für Variablendeklarationen in der Sprache des einzusetzenden Modelprüfers zu definieren.

```
<spec>          ::= <opt prelude> <consistency>
<opt prelude>   ::= <> | <prelude>
<prelude>       ::= beginprelude <any text> endprelude
<consistency>   ::= <data dict> <sc spec> <msc def> satisfies <msc specs>
```

Als Datenelemente werden Variablen, Ereignisse, Aktionen und Nachrichten deklariert. Aktionen sind Ereignisse für die festgelegt ist, in welcher MSC-Instanz sie sichtbar sind. In Deklarationen von Nachrichten wird festgelegt, von welcher zu welcher MSC-Instanz eine Nachricht verschickt werden kann. Eine Nachricht entspricht einer boole'schen Variable sowie zwei Ereignissen – einer für das Absenden der Nachricht und einer für das Empfangen der Nachricht.

```
<data dict>     ::= <> | <data decl> <data dict>
<data decl>     ::= event <id>
                  | var <id> :  <type>
                  | message <id> :  <id> , <id>
                  | action <id> :  <id>
```

Mit der im Text definierten Syntax für Statecharts müssen diese als monolithische Blöcke spezifiziert werden. Daher definieren wir hier eine erweiterte Syntax für Statecharts, die es erlaubt, ihre Beschreibung textuell besser zu strukturieren, indem sie über ihren Namen referenziert werden können.

Allerdings betrachten wir Definitionen von Statecharts als Instanzen und erheben daher eine weitere Einschränkung: ein Statechart darf nur an einer Stelle als Subchart eines anderen Statechart referenziert werden[1].

```
<sc spec>       ::= main <sc> | <sc def> <activity>
<sc>            ::= <sc ref> | <sc def>
<sc ref>        ::= sc <id>
```

[1] Dies könnte man aufweichen, indem man Definitionen von Statecharts als Definitionen von Klassen von Statecharts und Referenzen auf diese als Instanzenbildung interpretiert.

```
<sc def>          ::= <basic sc> | <and sc> | <or sc>
<basic sc>        ::= sc <id> :  basic
<and sc>          ::= sc <id> :  and <sc list> endsc <id>
<or sc>           ::= sc <id> :  or <sc list> <init> <trans> endsc <id>
<sc list>         ::= <sc> | <sc> <sc list>
<init>            ::= init :  <stmts> [ <ids> ]
<trans>           ::= <> | <tr> <trans>
<tr>              ::= tr <label> [ <sources> -> <targets> ]
<label>           ::= <id> :  <stmts>
<sources>         ::= <ids>
<targets>         ::= <target> | <target> , <targets>
<target>          ::= <id> | h(<id>) | h*(<id>)
<ids>             ::= <id> | <id> , <ids>
```

Anweisungen bestehen aus optionalen Bedingungen und Aktionen. Die Syntax für Bedingungen und Aktionen haben wir hier vereinfacht; zur Vereinfachung gehen wir auch davon aus, dass Prädikate und Operationen vordefiniert sind (z.B. in `<prelude>`), die für `<pred>` und `<op>` eingesetzt werden können. Um eine notationelle Verbindung zwischen Statechart und MSCs herzustellen, gibt es spezielle Aktionen für das Versenden und Empfangen von Nachrichten. Die in MSCs aufgeführten Ereignisse beziehen sich insbesondere auf die Effekte dieser Aktionen.

```
<stmts>           ::= <stmt> | <stmt> ; <stmts>
<stmt>            ::= <opt cond> <opt action>
<opt cond>        ::= <> | [ <cond> ]
<cond>            ::= <id> | <pred> ( <opt ids> )
                      | ( <cond> ) | not <cond>
                      | <cond> and <cond> | <cond> or <cond>
<opt action>      ::= <> | <action>
<action>          ::= <gen event> | <assign>
                      | <send action> | <receive action>
<gen event>       ::= <id>
<assign>          ::= <op> ( <id> ; <opt ids> )
<send action>     ::= out ( <id> )
<receive action>  ::= in ( <id> )
<opt ids>         ::= <> | <ids>
```

MSCs werden in der herkömmlichen textuellen Notation beschrieben. Die Nachrichten und Aktionen können nur in den MSC-Instanzen auftreten, für die sie deklariert wurden.

```
<msc def>         ::= <msc> | <msc> <msc def>
<msc>             ::= msc <id> <msc body> endmsc ;
<msc body>        ::= <inst def> | <inst def> <msc body>
<inst def>        ::= instance <id> ; <inst body> endinstance ;
<inst body>       ::= <event> | <event> <inst body>
<event>           ::= in <id> from <inst> ;
                      | out <id> to <inst> ;
                      | action <id> ;
                      | cond <cond> ;
<inst>            ::= <id> | env
```

Die von einem Statechart zu erfüllenden Eigenschaften werden mit MSC$_{CTL}$-Formeln definiert.

```
<msc-specs>      ::= <msc spec> | <msc spec> <msc specs>
<msc spec>       ::= spec :  <msc ctl>
```

MSC_{CTL}-Formeln werden über MSCs definiert.

```
<msc ctl>        ::= A <msc ltl> | E <msc ltl>
                   | <cond>
                   | ( <msc ctl> )
                   | not <msc ctl>
                   | <msc ctl> and <msc ctl>
                   | <msc ctl> or <msc ctl>
<msc ltl>        ::= <msc form>
                   | X <msc ctl>
                   | F <msc ctl>
                   | G <msc ctl>
                   | <msc ctl> U <msc ctl>
<msc form>       ::= <id>
                   | ( <msc form> )
                   | not <msc form>
                   | <msc form> and <msc form>
                   | <msc form> or <msc form>
```

Der Statechart aus Abbildung 2.2 kann mit dieser Syntax äquivalent in beispielsweise folgender Form aufschreiben werden:

```
sc A: or
  sc B: basic
  sc C: basic
  init: [B]
  tr e: [B -> C]
  tr f: [C -> B]
endsc A

sc D: or
  sc E: basic
  sc F: basic
  sc G: basic
  init: [F]
  tr k: [G -> E]
  tr g: [E -> F]
  tr i: [F -> G]
endsc D

sc H: basic

main sc Z: or
  sc Y: and
    sc A
    sc D
  endsc Y
  sc H
  sc I: basic
  init: [Y]
```

```
  tr l: [I -> A,h(D)]
  tr m: [H -> C,D]
  tr n: [C,E -> H]
  tr o: [I -> C,F]
  tr p: [Y -> I]
endsc Z
```

Anhang B

Semantik von Anweisungen

Die Semantik von Anweisungen $[\![l]\!]$ ist über den syntaktischen Aufbau von Markierungen definiert. Das Resultat von von $[\![l]\!]$ ist ein Tupel $(event_l, var_l, op_l)$ wobei $event_l$ die Menge der von l erzeugten Ereignisse, var_l die von l modifizierten Variablen und op_l die Zustandsübergangsrelation über S_{XE} darstellen.

Als Operation ($||$) zur Verknüpfung der Semantik zweier Anweisungen, l_1 und l_2, die gemeinsam ausgeführt werden, definieren wir

$$[\![l_1]\!] \,||\, [\![l_2]\!] \;\triangleq\; (event_{l_1} \cup event_{l_2}, var_{l_1} \cup var_{l_2}, op_{l_1} \cap var_{l_2})$$

Die Semantik einer Anweisung l ist in folgender Weise erklärt:

- Ist $l =$ "id: stmts", so ist $[\![l]\!] \triangleq [\![$"stmts"$]\!]$.
- Ist $l =$ "stmt; stmts", so ist $[\![l]\!] \triangleq [\![$"stmt"$]\!] \,||\, [\![$"stmts"$]\!]$.
- Ist $l =$ "[cond] action", so ist $[\![l]\!] \triangleq [\![$"[cond]"$]\!] \,||\, [\![$"action"$]\!]$.
- Ist $l =$ "", eine leere Aktion, so ist $[\![l]\!] \triangleq (\emptyset, \emptyset, S_{XE} \times S_{XE})$.
- Ist $l =$ "e" eine Aktion die ein Ereignis generiert, so ist $[\![l]\!] \triangleq (\{e\}, \emptyset, S_{XE} \times \{xe \in S_{XE} \mid xe.e = true\})$.
- Ist $l =$ "op(x;id$_1$,...,id$_k$)") eine Zuweisung, wobei für die Zuweisungsoperation "op" ihre Semantik in Form einer Menge $op \subseteq S_{XE} \times S_{XE}$ vorliegt, so ist $[\![l]\!] = (\emptyset, \{x\}, op)$.
- Ist $l =$ "out(msg)" eine Aktion um Nachricht msg zu versenden, so kann die Aktion eintreten, wenn noch keine Nachricht msg vorliegt und dann wird Ereignis msg_{out} erzeugt und Variable msg auf den Wert $true$ gesetzt, d.h. $[\![l]\!] = [\![$"[not msg] msg$_{out}$"$]\!] \,||\, (\emptyset, \{msg\}, S_{XE} \times \{xe2 \in S_{XE} | xe2.msg = true\})$.
- Ist $l =$ "in(msg)" eine Aktion um Nachricht msg zu empfangen, so kann die Aktion eintreten, wenn Nachricht msg vorliegt und dann wird Ereignis msg_{in} erzeugt und Variable msg auf den Wert $false$ gesetzt, d.h. $[\![l]\!] \triangleq [\![$"[msg] msg$_{in}$"$]\!] \,||\, (\emptyset, \{msg\}, S_{XE} \times \{xe2 \in S_{XE} \mid xe2.msg = false\})$.
- Ist $l =$ "[c$_1$ and c$_2$]" die Konjunktion zweier Bedingungen, so ist $[\![l]\!] \triangleq [\![$"[c$_1$]"$]\!] \,||\, [\![$"[c$_2$]"$]\!]$.
- Ist $l =$ "[c$_1$ or c$_2$]" eine Disjunktion mit $[\![$"[c$_i$]"$]\!] = (event_{c_i}, var_{c_i}, op_{c_i})$, so ist $[\![l]\!] \triangleq (event_{c_1} \cup event_{c_2}, var_{c_1} \cup var_{c_2}, op_{c_1} \cup op_{c_2})$.

- Ist $c =$ "[not c]" eine Negation mit $[\![$"[c]"$]\!] = (event_c, var_c, op_c)$, so ist $[\![l]\!] \triangleq (event_c, var_c, (S_{XE} \times S_{XE}) \backslash op_c)$.
- Ist $c =$ "[id]" eine Bedingung wobei id eine boole'sche Variable bzw. ein Ereignis referenziert, so ist $[\![l]\!] \triangleq (\emptyset, \emptyset, \{xe \in S_{XE} \mid xe.id = true\} \times S_{XE})$.
- Ist $c =$ "[pred(id$_1$,|dots,id$_k$)]" ein Prädikat dessen Semantik mit einer Menge $pred \subseteq S_{XE}$ vorliegt, so ist $[\![l]\!] \triangleq (\emptyset, \emptyset, pred \times S_{XE})$.

Anhang C

Übersetzungsregeln

C.1 Regel *a2m*

Die Übersetzungsregel $a2m$ ist über den strukturellen Aufbau der Statecharts definiert. Sie wird in der Form $a2m(SC, tgt, s1, s2)$ angewendet, wobei SC der zu aktivierende Statechart ist, tgt die zu aktivierende Zielkonfiguration bezeichnet und *s1* sowie *s2* die an dem Zugangsübergang beteiligten Zustände in mucke-Notation darstellen.

- Ist SC ein einfacher Statechart ist $a2m(SC, tgt, s1, s2)$ definiert als

  `(!`*s1*`.activity & `*s2*`.activity)`

- Ist $SC = \bigwedge_{i=1}^{n} Q_i$, so ist $a2m(SC, tgt, s1, s2)$ für

  1. $tgt = \{SC\}$ und $tgt = \{h(SC)\}$ definiert als

     $(a2m(Q_1, \{Q_1\}, s1.q_1, s2.q_1)$ & ... &
     $a2m(Q_n, \{Q_n\}, s1.q_n, s2.q_n))$

  2. $tgt = \{h^*(SC)\}$ definiert als

     $(a2m(Q_1, \{h*(Q_1)\}, s1.q_1, s2.q_1)$ & ... &
     $a2m(Q_n, \{h*(Q_n)\}, s1.q_n, s2.q_n))$

  3. $tgt = \bigcup_{i=1}^{n} tgt_i$ mit $tgt_i \in Cov^h(Q_i)$ definiert als

     $(a2m(Q_1, tgt_1, s1.q_1, s2.q_1)$ & ... &
     $a2m(Q_n, tgt_n, s1.q_n, s2.q_n))$

- Ist $SC = (\bigvee_{i=1}^{n} Q_i, Tr_{SC})$ mit $Tr_{SC} = \{(\emptyset, init_{SC}, tgt_0), (src_1, l_1, tgt_1), \ldots, (src_n, l_n, tgt_n)\}$, so ist $a2m(SC, tgt, s1, s2)$ für

  1. $tgt = \{SC\}$ mit $tgt_0 \in Cov^h(Q_j)$ definiert als

     (*s2*.history=Q_j & *s2*.label=*init_SC* &
     $a2m(Q_j, tgt_0, s1.q_j, s2.q_j)$ &
     $s1.q_1$=$s2.q_1$ & ... & $s1.q_{j-1}$=$s2.q_{j-1}$ &
     $s1.q_{j+1}$=$s2.q_{j+1}$ & ... & $s1.q_n$=$s2.q_n$)

2. $tgt = \{h(SC)\}$ definiert als

```
((s1.history=void_SC & a2m(SC,{SC},s1,s2)) |
 (s1.history=s2.history & s2.label=nil_SC &
  ((s1.history=Q_1 & a2m(Q_1,{Q_1},s1.q_1,s2.q_1) &
    s1.q_2=s2.q_2 & ... & s1.q_n=s2.q_n) |
   ... |
   (s1.history=Q_n & a2m(Q_n,{Q_n},s1.q_n,s2.q_n) &
    s1.q_1=s2.q_1 & ... & s1.q_n-1=s2.q_n-1))))
```

3. $tgt = \{h^*(SC)\}$ definiert als

```
((s1.history=void_SC & a2m(SC,{SC},s1,s2)) |
 (s1.history=s2.history & s2.label=nil_SC &
  ((s1.history=Q_1 & a2m(Q_1,{h*(Q_1)},s1.q_1,s2.q_1) &
    s1.q_2=s2.q_2 & ... & s1.q_n=s2.q_n) |
   ... |
   (s1.history=Q_n & a2m(Q_n,{h*(Q_n)},s1.q_n,s2.q_n) &
    s1.q_1=s2.q_1 & ... & s1.q_n-1=s2.q_n-1))))
```

4. $tgt \in Cov^h(Q_j)$ definiert als

```
(s2.history=Q_j & s1.label=s2.label &
 a2m(Q_j,tgt,s1,s2) &
 s1.q_1=s2.q_1 & ... & s1.q_j-1=s2.q_j-1 &
 s1.q_j+1=s2.q_j+1 & ... & s1.q_n=s2.q_n)
```

C.2 Regel *d2m*

Die Übersetzungsregel $d2m$ ist über den strukturellen Aufbau der Statecharts definiert. Sie wird in der Form $d2m(SC, src, s1, s2)$ angewendet, wobei SC der zu deaktivierende Statechart ist, src die zu deaktivierende Zielkonfiguration bezeichnet und *s1* sowie *s2* die an dem Zugangsübergang beteiligten Zustände in mucke-Notation darstellen.

- Ist SC ein einfacher Statechart ist $d2m(SC, src, s1, s2)$ definiert als

```
(s1.activity & !s2.activity)
```

- Ist $SC = \bigwedge_{i=1}^{n} Q_i$, so ist $d2m(SC, src, s1, s2)$ für

  1. $src = \{SC\}$ definiert als

```
(d2m(Q_1,{Q_1},s1.q_1,s2.q_1) & ... &
 d2m(Q_n,{Q_n},s1.q_n,s2.q_n))
```

  2. $src = \bigcup_{i=1}^{n} src_i$ mit $src_i \in Cov(Q_i)$ definiert als

```
(d2m(Q_1,src_1,s1.q_1,s2.q_1) & ... &
 d2m(Q_n,src_n,s1.q_n,s2.q_n))
```

- Ist $SC = (\bigvee_{i=1}^{n} Q_i, Tr_{SC})$, so ist $d2m(SC, src, s1, s2)$ für

1. $src = \{SC\}$ definiert als

```
(s1.history=s2.history & s2.label=nil_SC &
 (active_Q1(s1.q1) & d2m(Q1,{Q1},s1.q1,s2.q1) &
  s1.q2=s2.q2 & ... & s1.qn=s2.qn) |
 ... |
 (active_Qn(s1.qn) & d2m(Qn,{Qn},s1.qn,s2.qn) &
   s1.q1=s2.q1 & ... & s1.qn-1=s2.qn-1))
```

2. $src \in Cov(Q_j)$ definiert als

```
(s1.history=s2.history & s2.label=nil_SC &
 active_Qj(s1.qj) & d2m(Qj,src,s1.qj,s2.qj) &
 s1.q1=s2.q1 & ... & s1.qj-1=s2.qj-1 &
 s1.qj+1=s2.qj+1 & ... & s1.qn=s2.qn)
```

C.3 Regel *l2m*

Die Übersetzungsregel $l2m$ ist analog zu $[\![l]\!]$ über den syntaktischen Aufbau von Markierungen definiert. Sie wird angewendet in der Form $l2m(l,$*xe1*$,$*xe2*$)$. l ist die Umzusetzende Markierung; *xe1* und *xe2* sind die an dem Zustandsübergang beteiligten Zustände.

- Ist $l =$ "id: stmts", so ist $l2m(l,$*xe1*$,$*xe2*$)$ definiert als

  $l2m($"stmts",*xe1*$,$*xe2*$)$

- Ist $l =$ "stmt; stmts", so ist $l2m(l,$*xe1*$,$*xe2*$)$ definiert als

  $l2m($"stmt",*xe1*$,$*xe2*$)$ & $l2m($"stmts",*xe1*$,$*xe2*$)$

- Ist $l =$ "[cond] action", so ist $l2m(l,$*xe1*$,$*xe2*$)$ definiert als

  $l2m($"[cond]",*xe1*$,$*xe2*$)$ & $l2m($"action",*xe1*$,$*xe2*$)$

- Ist $l =$ "" eine leere Aktion, so ist $l2m(l,$*xe1*$,$*xe2*$)$ definiert als

  `true`

- Ist $l =$ "e" die Generierung eines Ereignisses, so ist $l2m(l,$*xe1*$,$*xe2*$)$ definiert als

  xe2`.e=true`

- Ist $l =$ "op(x;id$_1$,...,id$_k$)" eine Zuweisung so ist $l2m(l,$*xe1*$,$*xe2*$)$ definiert als

  $op($*xe2*`.x,`*xe1*`.`id_1`,...,`*xe1*`.`$id_k$$)$

Wir setzen hierbei voraus, dass für eine Zuweisungsoperation "op" ihre Semantik in mucke in Form einer boole'schen Funktion *op* vorliegt.

- Ist $l =$ "out(msg)" eine Aktion um Nachricht *msg* zu versenden, so ist $l2m(l, xe1, xe2)$ definiert als

  $l2m($"[not msg] msg_{out}"$, xe1, xe2)$ `&` *xe2.msg*

- Ist $l =$ "in(msg)" eine Aktion um Nachricht *msg* zu empfangen, so ist $l2m(l, xe1, xe2)$ definiert als

  $l2m($"[msg] msg_{in}"$, xe1, xe2)$ `& !` *xe2.msg*

- Ist $l =$ "[c_1 and c_2]" die Konjunktion zweier Bedingungen, so ist $l2m(l, xe1, xe2)$ definiert als

  $l2m($"[c_1]"$, xe1, xe2)$ `&` $l2m($"[c_2]"$, xe1, xe2)$

- Ist $l =$ "[c_1 or c_2]" die Disjunktion zweier Bedingungen, so ist $l2m(l, xe1, xe2)$ definiert als

  $(l2m($"[c_1]"$, xe1, xe2)$ `|` $l2m($"[c_2]"$, xe1, xe2))$

- Ist $l =$ "[not c_1]" die Negation einer Bedingungen, so ist $l2m(l, xe1, xe2)$ definiert als

  `!(`$l2m($"[c_1]"$, xe1, xe2)$`)`

- Ist $l =$ "[id]" eine Bedingung wobei "id" eine boole'sche Variable bzw. ein Ereignis referenziert, so ist $l2m(l, xe1, xe2)$ definiert als

  xe1.id`=true`

- Ist $l =$ "[pred(id_1,ldots,id_k)]" ein Prädikat dessen Semantik in mucke mit der boole'schen Funktion *pred* vorliegt, so ist $l2m(l, xe1, xe2)$ definiert als

  pred`(`*xe1.*id_1`,...,`*xe1.*id_k`)`

C.4 Regel *testl2m*

Die Übersetzungsregel $testl2m$ ist über den strukturellen Aufbau der Statecharts unter Einbeziehung der jeweiligen markierten Transitionssysteme definiert. Sie wird in der Form $testl2m(SC, l, s)$ angewendet; dabei ist s ein Grundzustand von SC (in mucke-Notation) der auf das Vorliegen von l geprüft werden soll. Wir setzen in unserer Definition $l \in L_{SC}$ voraus.

- Ist $SC = \bigwedge_{i=1}^{n} Q_i$ mit $l \in L_{Q_j}$, so ist $testl2m(SC, l, s)$ definiert als

$s.testl2m(Q_j, l, q_j)$

- Ist $SC = (\bigvee_{i=1}^{n} Q_i, Tr_{SC})$, so ist $testl2m(SC, l, s)$ für
  1. $l \in L_{Q_j}$ definiert als

     $s.testl2m(Q_j, l, q_j)$

  2. $(s, l, t) \in Tr_{SC}$ definiert als

     `s.label=l`

Anhang D

MSC_{CTL}-Formeln in CTL

Im Beispiel aus Kapitel 5 wurde deutlich, dass eine direkte Umsetzung der Formeln in den μ-Kalkül aufgrund des im allgemeinen doppelt exponentiellen Aufwands nicht praktikabel ist. Probleme bereiten insbesondere die Subformeln der Gestalt `A <msc form>` und `E <msc form>`. Dies sind i.W. LTL-Formeln, die aussen mit einem Pfadquantor versehen sind. Man kann allerdings zeigen, dass MSC_{CTL}-Formeln bereits in CTL ausgedrückt werden können. Dazu gibt man Regeln zur Umformung der aus MSC_{CTL}-Ausdrücken gewonnenen CTL*-Formeln an; dabei werden Pfadquantoren so in die LTL-Formel für `<msc form>` hineingezogen, dass wie in CTL Pfadquantoren und LTL-Konnektoren zusammenstehen.

Unter Ausnutzung der Äquivalenz zwischen $\mathsf{A}\Phi$ und $\neg\mathsf{E}\neg\Phi$ beschränken wir uns nachfolgend auf Formeln der Gestalt `E <msc form>` bzw. deren Umsetzung nach CTL*. Ausdrücke der Gestalt `<msc form>` und deren Umsetzungen nach CTL* sind mit Disjunktionen, Konjunktionen und Negationen über (Namen von) MSCs bzw. deren Übersetzungen aufgebaut. Die MSCs selber entsprechen wiederum Konjunktionen über den in ihnen enthaltenen Instanzen. Entspricht mf der Form `<msc form>` so kann man die CTL*-Formel $\mathsf{E}(mf\,2\,ctl(mf))$ in die disjunktive Normalform überführen, wobei die Instanzen (bzw. deren Umsetzungen) als Atome betrachtet werden. Hier setzen wir mit Regeln an, die letztendlich zu einer Umformung der aus MSC_{CTL} gewonnenen CTL*-Ausdrücke zu CTL-Formeln führen.

Wir unterscheiden zwischen Zustands- und den echten[1] Pfadformeln: Zustandsformeln sind zustandsabhängige Prädikate oder tragen als äussersten Operator einen Pfadquantor oder sind über Negationen, Konjunktionen und Disjunktionen von Zustandsformeln aufgebaut; die übrigen Formeln sind echte Pfadformeln. Nachfolgend ist eine Liste mit den relevanten Umformungsregeln angegeben. Aufgrund der unten aufgeführten Regel 1 kann man sich hauptsächlich auf (Teil-)Formeln für positive und negierte Instanzen und Konjunktionen darüber konzentrieren. Auch wenn im Lauf der Umformung Disjunktionen auftreten kann man unter Anwendung der Distributivgesetze und der Regel (1) wieder zu dieser Klasse von Formeln zurückkehren.

Eine wichtige Rolle für die Umformung spielt der Aufbau der Formeln für Instanzen: In der Regel steht unter einem binären LTL-Konnektor als linkes Argument eine Zustandsformel. Wichtig im Zusammenhang mit der Umformung negierter Instanzen ist, dass Formeln der Gestalt "$\neg\mathsf{X}(\alpha\mathsf{U}\beta \wedge \Phi)$" auftreten wobei α und β Zustandsformeln sind, die nicht gleichzeitig wahr werden: α enthält i.W. $none_I$ (bzw. im Fall von Bedingungen $\neg cond$) und β enthält i.W. $just_I(e)$ (bzw. im Fall von Bedingungen $cond$) womit sich beide gegenseitig ausschliessen.

[1] In der Literatur zu CTL* werden üblicherweise alle Formeln als Pfadformeln bezeichnet; die Zustandsformeln bilden eine Teilklasse davon. Wir bezeichnen hier Formeln die nicht Zustandsformeln sind als echte Pfadformeln.

Die Regeln zur Umformung von CTL*-Formeln für MSC_{CTL} nach CTL sind Tautologien für CTL*-Formeln; ihre Anwendungen liefern somit Formeln, die zur Ausgangsformel äquivalent sind:

1. E distributiert über Disjunktionen von Pfadformeln:
   $\mathsf{E}(\Phi \vee \Psi) = (\mathsf{E}\Phi) \vee (\mathsf{E}\Psi)$

2. E kann in die Formel für eine Instanz gezogen werden; sind α und β Zustandsformeln und ist Φ eine Pfadformel, so gelten jeweils:
   $\mathsf{EX}(\alpha \mathsf{U} \Phi) \;=\; \mathsf{EXE}(\alpha \mathsf{U} \mathsf{E}\Phi)$
   $\mathsf{EX}(\alpha \mathsf{U} \beta) \;=\; \mathsf{EXE}(\alpha \mathsf{U} \beta)$

3. E kann sowohl unter eine Disjunktion als auch eine Konjunktion aus Zustands- und Pfadformel gezogen werden:
   $\mathsf{E}(\alpha \vee \Phi) \;=\; \mathsf{E}\alpha \vee \mathsf{E}\Phi$
   $\mathsf{E}(\alpha \wedge \Phi) \;=\; \alpha \wedge \mathsf{E}\Phi$

4. Steht E vor einer Zustandsformel α, so erzeugt man eine äquivalente CTL-Formel mit:
   $\mathsf{E}\alpha = \alpha \wedge \mathsf{EF}\mathit{true}$

5. Die Konjunktion zweier Instanzen kann man umformen, indem man die möglichen Anordnungen der Ereignisse aus beiden Instanzen in Form einer Disjunktion darstellt; sind α und β Zustandsformeln sowie ϕ und ψ beliebig Pfad- oder Zustandsformeln, so gilt:
   $$\begin{aligned}(\mathsf{X}(\alpha \mathsf{U} \phi)) \wedge (\mathsf{X}(\beta \mathsf{U} \psi)) \;=\; & (\mathsf{X}((\alpha \wedge \beta) \mathsf{U} (\phi \wedge \psi))) \vee \\ & (\mathsf{X}((\alpha \wedge \beta) \mathsf{U} (\phi \wedge \beta \wedge (\mathsf{X}(\beta \mathsf{U} \psi))))) \vee \\ & (\mathsf{X}((\alpha \wedge \beta) \mathsf{U} (\alpha \wedge (\mathsf{X}(\alpha \mathsf{U} \phi)) \wedge \psi)))\end{aligned}$$

6. Negierte Instanzen werden in Teilformeln zerlegt, die leichter zu behandeln sind; ist Φ eine Pfadformel und sind α und β Zustandsformeln für die in jedem Zustand gilt $\neg(\alpha \wedge \beta)$, so ist:
   $\neg\mathsf{X}(\alpha \mathsf{U} (\beta \wedge \Phi)) = \neg\mathsf{X}(\alpha \mathsf{U} \beta) \;\vee\; \mathsf{X}(\alpha \mathsf{U} (\beta \wedge \neg\Phi))$

7. Werden Instanzen negiert, in denen Bedingungen auftreten, wenden wir zunächst die *de Morgan*'schen Regeln und das Distributivgesetz an:
   $\neg((\mathit{cond} \wedge \Phi) \vee \Psi) = (\neg \mathit{cond} \wedge \neg\Psi) \vee (\neg\Phi \wedge \neg\Psi)$

8. Das erste Disjunkt aus Regel (6) kann weiter umgeformt werden; sind α und β Zustandsformeln für die in jedem Zustand gilt $\neg(\alpha \wedge \beta)$, so ist:
   $\neg\mathsf{X}(\alpha \mathsf{U} \beta) = \mathsf{XG}(\alpha) \vee \mathsf{X}(\alpha \mathsf{U} (\neg\alpha \wedge \neg\beta))$

9. Konjunktionen über XG... werden mit folgender Regel umgeformt:
   $(\mathsf{XG}\alpha) \wedge (\mathsf{XG}\beta) = \mathsf{XG}(\alpha \wedge \beta)$

10. Bei Konjunktionen von $\mathsf{XG}\alpha$ mit Instanzen kann $\mathsf{XG}\alpha$ in die Instanz hineingezogen werden; ist ϕ eine Zustands- oder Pfadformel und sind α und β Zustandsformeln, so gilt:
    $(\mathsf{XG}\alpha) \wedge (\mathsf{X}(\beta \mathsf{U} \phi)) = \mathsf{X}((\alpha \wedge \beta) \mathsf{U} (\alpha \wedge (\mathsf{XG}\alpha) \wedge \phi))$

11. Zur Umformung von Formeln der Gestalt $\mathsf{XG}\alpha$ mit vorangestelltem existentiellem Pfadquantor dient die Regel:
    $\mathsf{EXG}\alpha = \mathsf{EXEG}\alpha$

Die Gültigkeit dieser Regeln haben wir mit dem PVS-System[COR+95] nachgewiesen, indem wir alle Regeln als Theoreme über einem abstrakten Transitionssystem formalisiert und als wahr bewiesen haben.

Die Regeln für den Pfadquantor E (1,2,3,4,11) führen zu einer Verkürzung der Teilformeln, die noch nicht in CTL-Form sind. Die übrigen Regeln dienen zusammen mit dem Distributivgesetz für $\wedge$, $\vee$ und mit dem Kommutativgesetz für $\wedge$ dazu, die Formeln so umzuwandeln, dass die Regeln für E anwendbar werden. Aus diesen Regeln kann man für die aus MSC_{CTL} gewonnenen CTL*-Formeln ein Termersetzungssystem erzeugen, das terminiert und als Ergebnis eine zur Ausgangsformel äquivalente CTL-Formel liefert.

Zusätzlich können Optimierungen zur Verkleinerung der generierten Formel vorgenommen werden. Dabei nutzt man die Tatsache aus, dass Nachrichten nur empfangen werden können, nachdem sie ausgesandt wurden; auch die Ereignisse entlang einer Instanz können nur nacheinander auftreten. Daraus kann man eine partielle Ordnung über den Ereignissen aufstellen. In der Umsetzungsregel (5) fallen dann die Disjunkte weg, die in Konflikt mit der partiellen Ordnung auf den Ereignissen stehen. Wenn viele Paare von Ereignissen in der partiellen Ordnung stehen, kann das zu einer erheblichen Verkleinerung der generierten Formel führen; der Gewinn ist hierbei natürlich am grössten, wenn eine vollständige Ordnung vorliegt.

Literaturverzeichnis

[Bie97] Armin Biere. *Effiziente Modelprüfung des μ-Kalküls mit binären Entscheidungsdiagrammen.* Dissertation, Universität Karlsruhe, 1997.

[CFG99] Paolo Ciancarini, Alessandro Fantechi, and Roberto Gorrieri, editors. *Formal Methods for Open Object-Based Distributed Systems*, Florence, February 1999. IFIP, Kluwer Academic Publishers.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995.

[Dam90] Mads Dam. Translating CTL* into the Modal μ-Calculus. Technical report, University of Edinbourgh, 1990.

[Dam94] Mads Dam. CTL* and ECTL* as Fragments of the Modal μ-Calculus. *Theoretical Computer Science*, 126:77–96, 1994.

[DH98] Werner Damm and David Harel. LSCs: Breathing Live into Message Sequence Charts. Technical Report CS98-09, Weizmann Institute, Israel, April 1998. Also published in [CFG99].

[Eme90] E. Allen Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier Science Publishers B.V., 1990. Formal Models and Semantics.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[HN96] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[HPSS87] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the Formal Semantics of Statecharts. In *2nd IEEE Symposium on Logic in Computer Science*, pages 54–64, New York, 1987. IEEE Press.

[McM92] K.L. McMillan. *The SMV system.* Carnegie Mellon University, draft edition, February 1992.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Carnegie Mellon University, 1993. Revised version of PhD thesis.

[MR94] S. Mauw and A. Reniers. An Algebraic Semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.

[ORRS96] S. Owre, S. Rajan, J.M. Rushby, and N. Shankar. PVS: Combining Specification, Proof Checking, and Model Checking. volume 1102 of *LNCS*, pages 411–414, New Brunswick, NJ, July 1996. Springer.

[Ref96] Frank Reffel. Übersetzung von CTL*-Formeln in den μ-Kalkül. Diplomarbeit, Universität Karlsruhe, September 1996.

[RGG96] Ekkart Rudolph, Peter Graubmann, and Jens Grabowski. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.

[Sha96] Natarajan Shankar. Unifying Verification Paradigms. In Bengt Jonsson, editor, *FTRTFT'96*, volume 1135 of *LNCS*, pages 22–39, Uppsala, Sweden, September 1996. Springer.

[UML97] Rational Software Corporation, Santa Clara, CA, USA. *Unified Modeling Language*, 1.1 edition, September 1997. `http://www.rational.com/uml`.