Seminar: Formale Software-Entwicklung

Dynamische Modellprüfung im Bereich von UML durch Übersetzung von UML-Modellen nach PROMELA, basierend auf einem Ansatz von Toni Jusilla

Rolf Haynberg
Fakultät für Informatik
Universität Karlsruhe
E-Mail: rolf@haynberg.de

Institut für Theoretische Informatik
Betreuer: Mattias Ulbrich

1 Einleitung

UML-Modelle sind weit verbreitet. Dadurch werden Techniken, diese maschinell prüfen zu können (mit Model Checking) immer mehr gefordert. In dieser Arbeit wird ein Verfahren vorgestellt, mit dem sich dynamische UML-Modelle automatisch in PROMELA, die Eingabesprache des populären Model Checkers SPIN, übersetzen lassen. So lassen sich dynamische UML-Modelle hocheffizient auf beliebige Temporallogische Aussagen hin prüfen (wie z.B. Verklemmungen). Das hier vorgestellte Verfahren zur Übersetzung ist nicht neu. Es wurde von Toni Jussila[1] 2006 entwickelt. Sein Ansatz eignet sich besonders für Protokoll-Modelle.
Die Übersetzung nach PROMELA orientiert sich am Aufbau des späteren Quelltextes. Variablen und Begriffe die zu Beginn gebraucht werden, tauchen später im Programm wieder auf. Daher müssen die Kapitel dieser Arbeit als aufeinander aufbauend verstanden werden.

2 Die Eingabe: Das UML-Modell

Der Begriff Modellprüfung ist nur im Kontext von dynamischen Modellen sinnvoll. Rein statische Modelle wie etwa ein Klassendiagramm lassen sich nicht auf temporallogische Aussagen hin prüfen. Deswegen kommen für die Übersetzung nur dynamische Modelle, wie sie etwa in der Verhaltensmodellierung vorkommen, in Frage. Allerdings müssen auch dort Einschränkungen getroffen werden, denn UML-Diagramme können sehr kompliziert sein. Das wird dann ersichtlich, wenn man die Sprachkonzepte von UML-Diagrammen, also das UML-Metamodell, betrachtet[2]. Es ist derart komplex, dass oftmals nur ein Ausschnitt gezeigt wird. Nicht einmal alle Modellierungswerkzeuge([3],[4]), unterstützen die Vielzahl der unterschiedlichen Modellierungsmöglichkeiten, obwohl diese im Gegensatz zu der Übersetzung die Semantik nicht berücksichtigen. Damit kommen

wir zu dem zweiten Grund für die zu treffenden Einschräkungen: Die Semantik ist in UML oft nicht vollständig und eindeutig beschrieben, was es für Modellprüfungen unmöglich macht feste Aussagen über das Modell zu treffen.
Aufgrunddessen ist es nach dem heutigen Stand der Forschung praktisch unmöglich, ein beliebiges, dynamisches UML-Modell zu übersetzen. Es ist die Vielfalt der Modellierungselemente und die teilweise fehlende Semantik, die ein Problem für derartige Übersetzungungen darstellen; nicht die Größe der Eingabe. Ein großes Diagramm stellt für die spätere Übersetzung kein Problem dar, solange wenige verschiedene Modellierungselemente verwendet werden.
Die Eingabe muss also semantisch beschränkt werden. Dieses Kapitel beschreibt welche Einschränkungen in der Arbeit von Toni Jussila gemacht bzw. welche UML-Modelle als Eingabedaten unterstützt werden. Die Einschränkungen entscheiden maßgeblich über die Einsatzmöglichkeiten und damit auch über den Einsatzbereich der Übersetzung.
Die hier vorgestellte Übersetzung zielt auf Protokoll-Modelle ab: Dynamische Modelle also, die beschreiben, wie Nachrichten zwischen Instanzen (Objekten) ausgetauscht werden.
Andere Arbeiten zu diesem Thema unterscheiden sich in den Einschränkungen. Das OMEGA Projekt[5] hat sich beispielsweise auf Modelle von Echtzeitsystemen und eingebetteten Systemen spezialisiert.
Die größte Einschränkung findet man bei den Diagrammtypen. Das hier vorgestellte Eingabemodell darf nämlich nur drei Diagrammentypen enthalten. Diese sind:

- Ein Klassendiagramm zur Beschreibung der Struktur des Systems bzw. den beteiligten Instanzen (Objekten). Außerdem können hier bereits Assoziationen zwischen Klassen enthalten sein. Sie helfen dabei, die Kommunikationswege zu strukturieren.
- Für jede Klasse ein Zustandsdiagramm zur Beschreibung des Klassenverhaltens. Diese Diagramme stellen den wichtigsten Teil des Modells dar weil sie das Verhalten beschreiben welches nach der Übersetzung geprüft werden soll. Sie modellieren außerdem das Sende- und Empfangsverhalten der Instanzen.
- Ein Verteilungsdiagramm zur Beschreibung der Verteilung von Instanzen und somit des Aufbaus des Systems. Es ist notwendig um die Zustandsdiagramme ausführbar zu machen.

Diese Diagrammtypen reichen bereits aus, um die Kommunikation zwischen Objekten zu modellieren.
Die Einschränkungen setzen sich aber auch innerhalb der Diagramme fort. Wie diese Einschränkungen aussehen, bzw. welche Elemente in den Diagrammen unterstützt werden, ist in den folgenden Abschnitten (2.1, 2.2 und 2.3) beschrieben.

2.1 Das Klassendiagramm

Das Klassendiagramm unterliegt bei Weitem den stärkesten Einschränkungen. Sie wurden so gewählt, dass das Klassendiagramm einen einzigen Zweck erfüllt:

Objekte in einem Protokoll-Modell zu beschreiben.
Das Klassenverhalten sowie die Kommunikation zwischen Objekten wird durch Zustandsdiagramme beschrieben. Alle verwendeten Klassen sind also aktive Klassen. Das macht Operationen in Klassen überflüssig. Sie werden daher nicht unterstützt.
Möglich ist hingegen die Angabe von Instanzattributen jeglichen Typs. Weil aber Klassenattribute nicht zur Beschreibung von Objekten in Protokoll-Modellen dienen werden diese ebenfalls nicht unterstützt.
Das Klassendiagramm darf auch Assoziationen enthalten. Die Assoziationen stellen später Kommunikationsbeziehungen im Verteilungsdiagramm dar (2.3). Sie werden also genutzt, um indirekt den Signalfluss zwischen Objekten zu modellieren.
Generalisierungen werden allerdings nicht unterstützt. Genauso alle anderen, hier nicht erwähnten Elemente.

2.2 Das Zustandsdiagramm

Das Zustandsdiagramm dient in unserem Fall zur Beschreibung von Objektaktivitäten bzw. Klassenverhalten. Es ist folglich immer an eine Klasse gebunden. Es ist ein zentraler Bestandteil des Eingabemodells. Erstens, weil sich die Beschreibung von Protokollen im Objektverhalten wiederspiegelt und zweitens, weil es die Elemente zum Empfangen und Versenden von Nachrichten zwischen den Objekten, enthält.
Schauen wir uns an, welche Einschränkungen hier gemacht wurden:
Die Verwendung von Exit- oder Entry-Aktivitäten in Zuständen wird nicht unterstützt. Sie lassen sich aber mit den Aktivitäten der ein- und ausgehenden Transitionen ersetzen, d.h. die Ausdrucksfähigkeit wird dadurch nicht vermindert. Auch do-Aktivitäten werden nicht unterstützt, denn sie sind in UML kontinuierlich ablaufende Operationen. PROMELA hingegen modelliert Prozesse nur in diskreten Schritten. Eine Abbildung ist somit nicht möglich.
An Pseudozuständen werden neben den obligatorischen Initial(Start-)zuständen auch Entscheidungs(choice-)- und Endzustände unterstützt. *Join* und *Fork* werden nicht unterstützt. Anderfalls könnten zwei Transitionen bei einem Ereignis schalten. Weshalb es gilt solche Fälle zu vermeiden, ist in Abschnitt 3.2 beschrieben bzw. in Abb. 2 zu erkennen.
Die Übersetzung erlaubt außerdem die Verwendung von komplexen (auch: Super---, zusammengesetzten, geschachtelten) Zuständen. Sowohl UND- als auch ODER-Verfeinerungen sind zugelassen. Zur Erinnerung:
Bei der ODER-Verfeinerung ist immer genau ein Unterzustand aktiv, wenn der Superzustand aktiv ist.[6]
Bei der UND-Verfeinerung enthält ein Zustand mehrere orthogonale Regionen. Diese sind nebenläufig, d.h. aus jeder Region ist immer genau ein Zustand aktiv, wenn der zusammengesetzte Zustand aktiv ist.[6]
Aber auch hier muss eine Einschränkung berücksichtigt werden: Es dürfen sich in zueinander orthogonalen Regionen keine zwei Transitionen befinden, die vom

gleichen Ereignis ausgelöst werden. Dadurch wird garantiert, dass in dem Modell mit jedem Ereignis höchstens eine Transition ausgelöst wird und ermöglicht somit eine praktikable Implementierung der Übersetzung(siehe auch Abschnitt 3.2). Diese Einschränkung lässt sich darüber hinaus beheben, indem stattdessen mehrere Objekte ohne orthogonale Regionen verwendet werden.
Eine weitere unterstützte Eigenschaft ist das Aufschieben von Ereignissen (deferring). Zu jedem Zustand kann angegeben werden, welche Ereignisse aufgeschoben werden dürfen.
Die Assoziationen zwischen Zuständen (Transitionen) können zu einem hohen Grad parametrisiert werden:

- Es kann ein Ereignis angegeben werden, bei dem die Transition schaltet.
  Diese Angabe ist optional. Fehlt sie, schaltet die Transition sobald der Vorzustand seine Aktivität beendet hat (sog. *Komplettierungs-Transition*).
- In den Ereignissen können Argumente verwendet werden.
  Durch die Argumente wird die zuvor angesprochene asynchrone Kommunikation zwischen Objekten nutzbar und die Verwendung von Objektoperationen überflüssig.
- Desweiteren kann eine Bedingung (guard) angegeben werden. Nur wenn sie efüllt ist, kann die Transition schalten.
- Schließlich kann eine Aktivität angegeben werden.
  Sie beschreibt zum einen welche Vorgänge beim Schalten ausgeführt werden und zum anderen welche Signale gesendet werden. Da keine Entry-, Exit-, und Do-Aktivitäten in den Zuständen unterstützt werden, stellt die Aktivität in den Transitionen die einzige Möglichkeit dar, das Modell während der Ausführung zu verändern und eine Kommunikation zwischen Objekten zu ermöglichen.

Da UML die Wahl der Sprache für Aktivitäten offen lässt, ist die Modellierungsstärke also besonders von der Wahl bzw. der Mächtigkeit der gewählten Sprache abhängig.
In der Arbeit von Toni Jussila wurde Jumbala gewählt. Jumbala wurde speziell für Aktivitätsangaben in UML-Verhaltensdiagrammen entwickelt. Ihre Syntax ähnelt der von Java.
Der Umfang der Sprache muss aber an die Einschränkungen des UML-Modells angepasst werden. Die unterstützte Teilmenge von Jumbala enthält:

- Alle geläufigen boolschen Operatoren, konditionale Ausdrücke (if/else), while-Schleifen und Assertions.
- Konstruktoren, d.h., dass neue Objekte während der Ausführung erstellt werden können.
- Zugriff auf Instanzattribute. Dadurch wird die bereits angesprochene, mögliche Angabe von Instanzattributen erst nutzbar.
- Abweichend von Java gibt es eine Funktion um parametrisierte Signale (also Nachrichten) zu senden. Diese Eigenschaft ist notwendig für die Kommunikation unter den Objekten.

2.3 Das Verteilungsdiagramm

In den vorangegangenen Kapiteln wurde zum einen auf das Klassendiagramm eingegangen und zum anderen auf die Zustandsdiagramme, die das Verhalten und Kommunizieren von Instanzen beschreiben. Jedoch kann noch kein wirkliches Szenario modelliert werden, weil nicht definiert wurde welche Objekte existieren.
Dies ist die Aufgabe des Verteilungsdiagramms. Es beschreibt eine konkrete Konfiguration. Es handelt sich hierbei also um ein Verteilungsdiagramm auf Instanzebene.
Die (Instanz-)Knoten sind durch ungerichtete Kommunikationsbeziehungen miteinander verbunden. Sie erlauben es, Signale zwischen den Instanzknoten auszutauschen. Welche Verbindungen vorhanden sind, wird durch das Klassendiagramm vorgegeben (vgl. 2.1).

Wir haben in diesem Kapitel gesehen, dass sich das Eingabemodell für die Übersetzung aus drei Diagrammtypen zusammensetzt. Mit den Einschränkungen, die getroffen wurden, ist das Einsatzgebiet der Übersetzung weitgehend auf Protokoll-Modelle festgelegt. Im nächsten Kapitel wird gezeigt, wie das Modell in PROMELA-Code übersetzt wird.

3 Die Übersetzung in Promela

Es wurde im vorangegangenen Kapitel gezeigt, welche Modelle für die Übersetzung verwendet werden können. In diesem Kaptiel wird die eigentliche Übersetzung beschrieben. Damit gibt es die Antwort auf die zentrale Fragestellung dieser Arbeit; nämlich wie UML Modelle transformiert werden, um sie später Modellprüfungen unterziehen zu können.
Ein wichtiger Punkt zum Verständnis der Übersetzung ist, dass Klassen in PROMELA zu Prozesstypen (*Proctypes*) werden. Entsprechend werden Objekte, also Instanzen von Klassen, in PROMELA zu Prozessen, also Instanzen von Proctypes. Analog dazu wird das Verhalten der Objekte, welches in den Zustandsdiagrammen modelliert ist, zum Inhalt des Proctypes.
Die Beschreibung der Übersetzung des Zustandsdiagramms wird einen Großteil dieses Kapitels einnehmen. Die Klassen- und Verteilungsdiagramme tauchen dagegen nur im ersten Unterabschnitt *Globale Variablen und Initialisierung* auf.

3.1 Globale Variablen und Initialisierung

Dieser Abschnitt behandelt den ersten Teil des PROMELA-Programms. Er besteht aus wichtigen Variablen-Deklarationen sowie dem Initialisierungsblock, der bei der Ausführung des Programms als erster ausgeführt wird.

Globale Variablen Im gesamten Programm existieren strenggenommen nur zwei 'echte' globale Variablen: `MAXIDS` und `QSIZE`.
`MAXIDS` gibt an, wieviele Instanzen pro Klasse während des gesamten Ablaufs des

Modells höchstens existieren. Da Objekte während der Laufzeit instanziiert und auch terminiert werden können, ist es schwierig bis unmöglich diese Zahl a priori zu bestimmen. Sie muss daher vom Anwender beim Programmaufruf angegeben oder geschätzt werden. Da aber unterschiedliche Klassen unterschiedlich viele Objekte haben können, müsste für jede Klasse ein Wert angegeben werden. Um dem Benutzer diesen Aufwand zu ersparen, wird in der Arbeit von Toni Jussila nur der größte von allen Werten verlangt und dieser für alle Klassen verwendet. Sollte während des Ablaufs der Übersetzung mehr als `MAXIDS` Instanzen einer Klasse gleichzeitig exisiteren, wird ein Fehler ausgegeben.
`QSIZE` muss ebenfalls vom Anwender angegeben werden. Sie gibt die maximale Größe des Ereignisstapels eines Objekts an. Hier besteht das gleiche Problem: Da unterschiedliche Klassen unterschiedlich große Ereignisstapel haben können, müsste eigentlich für jede Klasse eine Zahl angegeben werden. Aus demselben, oben erwähnten Grund, wird hier ebenfalls nur der größte von allen Werten verlangt. Auch hier wird ein Fehler ausgegeben, wenn eine zu niedrige Zahl angegeben wurde, d.h. der Stapel überläuft.
Die anderen globalen Variablen im Prorgramm gehören eigentlich zu Klassen und würden in einem objektorientierten Umfeld, öffentlichen, statischen Variablen entsprechen. Da es dazu aber keine Entsprechung in PROMELA gibt, werden sie global definiert. Ihnen wird ein `klassenname_` voran gestellt, um die Klassenzugehörigkeit anzudeuten.
Die Variablen sind folgende:

- Die Anzahl der bereits instanziierten Objekte dieser Klasse (`procid`).
- Ein Array `inputqueues` mit `MAXIDS` Einträgen. Jeder Eintrag ist ein Ereignisstapel für ein Objekt dieser Klasse. Da höchstens `MAXIDS` Objekte während der Laufzeit gleichzeitig existieren, ist das Array groß genug. Jede `inputqueue` hat die Größe `QSIZE`. In ihr werden die Ereignisse für ein Objekt gesammelt. Sowohl externe Ereignisse als auch solche, die durch Signale übermittelt wurden.
- Ein weiteres Array `deferredqueues` mit den gleichen Eigenschaften wie `inputqueues` nimmt aufgeschobene Ereignisse auf.
- Eine Liste mit den Ereignissen, die Objekte dieser Klasse aufschieben dürfen.
- Mehrere Konstanten für Zustandskonfigurationen. Weil diese später eine wichtige Rolle spielen werden, werden diese von nun an **Zustandskonstanten** genannt. An dieser Stelle wird nur ihre Syntax definiert; die Semantik wird später im Abschnitt über Proctypes (3.2) erläutert. Für jeden Zustand wird eine Zustandskonstante definiert. Sie sind vom Typ `byte` und bekommen eine laufende Nummer als Wert zugewiesen. Als Namenskonvention verwendet Toni Jusilla in seiner Arbeit `s_Superzustand_..._Unterzustand`.
  Dazu kommen noch Zustandskonstanten mit einer speziellen Endung (`_busy, _final und _None`). Für welche Zustandstypen sie definiert sind und warum sie gebraucht werden, wird in Abschnitt 3.2 beschrieben.
- Eine Liste pro ausgehender Assoziation dieser Klasse. Die enthaltenen Werte werden die Prozess-Ids der Objekte aus der Zielklasse der Assoziation sein. Jede Liste hat `MAXIDS` Felder, denn es können per Definition höchstens `MAXIDS` Objekte der Zielklasse exisitieren.

– Alle möglichen, von außen auftretenden Signale. Sie sind ebenfalls vom Typ `byte` und haben eine laufende Nummer. Sie bekommen den Namen `signal_signalname`.

Ein Beispiel dazu ist unter in Abschnitt 4, Algorithmus 2 zu sehen.

Init-Block Betrachten wir nun die Initialisierung der Objekte. Dies geschieht in einem `init`-Block, der beim Programmstart als erster aufgerufen wird. Der Code ist außerdem von einem `atomic`-Block umschlossen, damit er nicht von anderen Prozessen unterbrochen werden kann.
Dem Verteilungsdiagramm entsprechend, werden nun die Proctypes mittels `run` instanziiert. Dabei wird die `procid` der entsprechenden Klasse gemäß ihrer Definition um 1 erhöht (siehe oben).
Außerdem werden währenddessen die Listen mit den Assoziationen aktualisiert. Da diese ungerichtet sind, müssen sowohl bei dem neuen Objekt selbst, als auch bei den verbundenen Objekten die Assoziations-Arrays angepasst werden. In die Liste mit den Assoziationen des neuen Objekts werden all die Prozess-IDs der verbundenen Objekte (sofern sie schon instanziiert wurden) eingetragen. Diese lassen sich leicht aus dem Verteilungsdiagramm ablesen. Außerdem wird in jedes Assoziations-Array der verbundenen Objekte die Prozess-ID des gerade neu instanziierten Objekts (Prozesses) eingetragen.
Ein Beispiel dazu ist in dem folgenden Alg. 1 dargestellt.

```
init{
  atomic{
    run M(proc_id);
    proc_id = proc_id + 1;

    run M(proc_id);
    loopback[proc_id] = proc_id-1;
    loopback[proc_id-1] = proc_id;
    proc_id = proc_id + 1;

    run M(proc_id);
    loopback[proc_id] = proc_id-1;
    loopback[proc_id] = proc_id-2;
    loopback[proc_id-1] = proc_id;
    loopback[proc_id-2] = proc_id;
  }
}
```

Algorithm 1: Beispiel eines Init-Blocks für ein Szenario in dem es gibt nur eine Klasse `M` gibt, die über eine Assoziation `loopback` mit sich selbst verknüpft ist. Im Verteilungsdiagramm sind drei Objekte dieser Klasse angegeben worden. Während der Erzeugung der Objekte wird das Array der Assoziation aktualisiert.

In den nächsten Kapiteln wird auf die Zustandsdiagramme bzw. den Inhalt der Proctypes eingegangen.

3.2 Proctypes

Deklarationen und Initialisierungen Es sei daran erinnert, dass jede Klasse aus dem Modell in PROMELA zu einem Proctype wird. Demnach ist es verständlich, dass die Instanzattribute zu Variablen-Initialisierungen in der Proctype-Definition werden. Dies geschieht als erstes.
Anschließend wird mit dem Schlüsselwort `xr` sichergestellt, dass andere Objekte nicht auf den Ereignis- und Aufschubstapel zugreifen können.
Danach werden alle Zustände die im Zustandsdiagramm auftreten, als Variablen vom Typ `byte` initialisiert. Weil diese Variablen später noch eine wichtige Rolle spielen, werden sie im Folgenden **Zustandsvariablen** genannt. In der Arbeit von Toni Jusilla wurde als Variablenname der Zustandsname mit vorgeschobenem `state_` gewählt. Tieferliegende Zustände werden durch einen Unterstrich getrennt. Zum Beispiel: `state_Superzustand_Unterzustand`.
Der übrige Teil der Proctype-Definition steuert die Ausführung der Transitionen.

Ausführung der Transitionen In UML gibt es bezüglich der Abarbeitung der Transitionen folgende Prioritäten:

1. Transtitionen ausgehend von Pseudozuständen
   Da sich Zustandsautomaten nicht in Pseudozuständen aufhalten können, schalten Transitionen, die von ihnen ausgehen, konzeptionell unmittelbar nach erreichen des Zustands. Damit werden Pseudozustände transparent für den Informationsfluss des Zustandsautomaten. Dies wird erreicht, indem man diesen Transitionen höchste Priorität einräumt.
2. Komplettierungs-Transitionen
   Sobald die Aktivität eines Zustands beendet ist, werden die ausgehenden Transitionen geschaltet.
3. Signalgesteuerte Transitionen
   Niedrigste Priorität haben Transitionen, die durch Signale ausgelöst werden.

Diese Prioritäten finden sich auch in der Übersetzung wieder. Für jede der drei Transitionstypen gibt es einen zusammenhängenden Code-Block in der Proctype-Definition. Die drei Blöcke werden ihren Prioritäten entsprechend nacheinander angeordnet. Dadurch wird der Block mit der höheren Priorität früher ausgeführt, d.h., dass die dazugehörigen Transitionen zuerst schalten. Nach der Schaltung einer Transition, wird die Ausführung wieder oben begonnen. Sollte bei einem Typ keine Tranisition schalten, geht die Ausführung zum nächsten Block über. Durch dieses Ablaufprinzip wird sichergestellt, dass die höherprioren Transitionen zuerst schalten.
Bei den Blöcken handelt es sich um `if`-Blöcke und jede Option darin steht für eine Transition entsprechenden Typs. Damit eine Transition schalten darf, muss

ihr Quellzustand aktiv und ihre Bedingung erfüllt sein. Beides wird in den Optionen geprüft. Jetzt wird auch verständlich, warum es vermieden wurde2.2, dass ein Ereigniss zwei Transitionen auslösen kann: Da in dem `if`-Block jeweils nur eine Option pro Schritt schaltet kann auch nur eine Transition ausgelöst werden. Andernfalls müssten an dieser Stelle komplizierte Abhängigkeiten beachtet werden.
Allen Blöcken gemein, ist außerdem die Implementierung des Schaltens von Transitionen. Wenn eine Transition schaltet, bedeutet dies zum einen, dass die Aktivität der Transition ausgeführt wird und zum anderen, dass der Zielzustand aktiv und der Quellzustand inaktiv wird.
Bisher wurde noch nichts dazu gesagt, wie geprüft wird, ob ein Zustand aktiv ist oder wie ein Zustand als aktiv gekennzeichnet wird. Dies ist ein wichtiger Punkt zum Verständnis der Übersetzung. Um das Konzept verstehen zu können, müssen wir uns an die *Zustandskonstanten* aus dem Abschnitt *Globale Variablen und Initialisierung* und an die *Zustandsvariablen* zu Beginn des Kapitels erinnern. Den Zustandsvariablen wird nämlich der Wert der Zustandskonstanten zugewiesen, um anzuzeigen, welche Zustände aktiv sind.
Um anzugeben, dass ein Zustand aktiv ist, wird dessen Zustandskonstante der Zustandsvariable seines Superzustands zugewiesen. Um zu prüfen, ob ein Zustand aktiv ist, wird folglich verglichen, ob die Zustandsvariable des nächst höheren Zustands den Wert der Zustandskonstante des zu prüfenden Zustands hat.
Dieses Konzept wurde gewählt, da es der Vorgabe in UML entspricht, dass zu jeder Zeit höchstens ein Unterzustand eines ODER-Verfeinerten Zustands aktiv sein darf. Allerdings birgt das Konzept auch Probleme in sich. Diese lassen sich aber lösen. Die Probleme und die dazugehörigen Lösungsstrategien sind im Folgenden tabellarisch zusammengefasst:

1. Gewöhnlich haben Zustände in einem Diagramm keinen Superzustand. Deswegen wird ein imaginärer, komplexer Zustand `Top` angenommen, der alle Zustände enthält. So lassen sich die Zustände des Diagramms als Unterzustände beschreiben.
2. Ist kein Unterzustand in einem komplexen Zustand aktiv, ist nicht klar, welchen Wert die Zustandsvariable des Superzustands anzunehmen hat. Für diesen Fall wird ein imaginärer Zustand `None` als Unterzustand für jeden komplexen Zustand eingeführt. Wenn kein echter Unterzustand aktiv ist, wird der Zustandsvariablen der Wert der Zustandskonstanten von `None` zugewiesen.
3. UND-verfeinerte Zustände haben nicht nur einen aktiven Unterzustand. Zu diesem Zweck gibt es für die UND-verfeinerten Zustände noch eine zusätzliche Zustandskonstante mit der Endung `_busy`. Sie wird verwendet, wenn der UND-verfeinerte Zustand aktiv ist.
4. UND-verfeinerte Zustände sind abgeschlossen, wenn sich alle Unterzustände in einem Endzustand befinden. Wie lässt sich das feststellen? Das Problem rührt eigentlich daher, dass die Endzustände nicht immer gleiche Namen haben. Um dies zu beheben, gibt es zu jedem komplexen Zustand noch eine

Zustandskonstante mit Endung `_final`. Sie wird dem Superzustand zugewiesen, wenn ein Unterzustand in einen Endzustand übergeht. Durch den einheitlichen Namen lässt sich nun bei UND-verfeinerten Zuständen prüfen, ob alle Unterzustände den Wert der Konstante mit Endung `_final` enthalten.

5. Wie lässt sich feststellen, ob eine Komplettierungs-Transition bereits ausgelöst wurde, wenn sie dabei nicht geschaltet hat (beispielsweise weil ihre Bedingung nicht erfüllt war)? Dies stellt ein Problem dar, weil laut UML-Definition Komplettierungs-Transitionen nur einmal ausgelöst werden. Lösen lässt es sich ebenfalls mit Zustandskonstanten mit der Endung `_busy`. Wenn für einen Zustand noch keine Komplettierungs-Transition geschaltet wurde, wird für jedes seiner Vorkommen die Zustandskonstante mit `_busy` verwendet. Wurde hingegen eine solche Transition geschaltet, wird die Zustandskonstante ohne Endung verwendet. Es wird also bei Erreichen eines Zustands immer zuerst die Konstante mit Endung `_busy` verwendet (außer bei Pseudozuständen).

Es wird im Folgenden noch auf ein paar Besonderheiten der `if`-Blöcke der Komplettierungs- und ereignisgesteuerten Transitionen hingewiesen.

Komplettierungs-Transitionen von echten Zuständen In diesem Block erscheinen nur Transitionen, deren Startzustand ein echter Zustand, d.h. ein einfacher oder komplexer Zustand, ist. Handelt es sich bei dem Quellzustand um einen UND-verfeinerten Zustand, muss außerdem geprüft werden, ob sich alle Regionen in Endzuständen befinden.
In Abschnitt 3.2(Problem 5), wurde bereits beschrieben, warum und wie kenntlich gemacht wird, dass eine Komplettierungs-Transition bereits ausgeführt wurde, diese aber nicht geschaltet hat, da ihre Bedinung nicht erfüllt war. Um die an gleicher Stelle beschriebene Lösung umzusetzen, wird die Bedinung der Transition nicht wie bei den anderen Transitionstypen in die Bedingung der Option aufgenommen, sondern in den Ausführungsbereich der Option. Sie ist dort als `if`-Block implementiert, dessen erste Option prüft, ob die Bedingung der Transition erfüllt ist. Ist dies der Fall, schaltet sie. Andernfalls ist die Bedinung nicht erfüllt und der Zustandsvariablen des Superzustands wird die Zustandskonstante ohne Endung zugewiesen (dies ist das Zeichen dafür, dass keine Komplettierungs-Transitionen mehr ausgeführt werden dürfen).

Ereignisgesteuerte-Transitionen Dieser Typ von Transitionen ist etwas komplizierter in der Implementierung als die vorherigen. Denn zum einen werden Ereignisse zur Auswertung benötigt und zum anderen muss berücksichtigt werden, dass die Transitionen tieferer Zustände höhere Priorität haben; d.h. sie konsumieren das Signal bevor es höhrere Transitionen erreicht.
Das Holen der Ereignisse: Dazu dient ein `if`-Block, welcher nichtdeterministisch ein Signal auswählt. Dieses Signal wird entweder aus der `inputqueue` dieses Prozesses oder von externen Signalen geholt. Zu diesem Zweck enthält der `if`-Block eine Option, bei der das Signal aus der `inputqueue` gelesen wird und weitere

Optionen für jedes externe Signal (zur Erinnerung: die Signale wurden als globale Variable definiert). Damit ein externes Signal gelesen wird, muss aber ein Zustand aktiv sein, dessen ausgehende Transition das Signal konsumieren kann.
Die Prioritäten unter Transitionen: Dieses Problem lässt sich durch Verschachtelung beheben. Dazu wird der Aufführungsteil bei Transitionen, deren Quellzustand komplex ist, geändert. Er enthält nun zwei `if`-Blöcke. Der erste `if`-Block prüft alle Transitionen der Unterzustände nach dem gleichen Schema, welches wir bereits kennen. Natürlich kann einer der Unterzustände auch wieder komplex sein etc. Erst wenn dort keine Transition geschaltet wurde, werden die eigenen Transitionen in dem zweiten `if`-Block geprüft.
Durch diese Verschachtelung wird die Priorität der tieferen Transitionen gewahrt, weil die tiefsten Zustände zuerst im Code behandelt werden.
Wenn auch in diesem letzten Block keine Transition ausgelöst wurde, gilt das Ereignis als verschluckt. Die Ausführung beginnt wieder oben, läuft bis zum letzten Block und holt ein neues Ereignis.
Bei diesem Typ von Transitionen muss bei der Prüfung nicht nur geprüft werden, ob der Quellzustand aktiv und die Bedingung erfüllt ist, sondern auch ob das Ereignis auf die Transition passt.
Im zweiten Block (den Komplettierungs-Transitionen) konnten den Zustandsvariablen Zustandskonstanten ohne Endung zugewiesen werden. Deswegen muss hier bei der Prüfung, ob der Quellzustand aktiv ist, auf beiden Zustandskonstanten (mit und ohne Endung) geprüft werden, da die ereignisgesteuerten Transitionen nicht berücksichtigen, ob für einen Zustand bereits ein Komplettierungs-Ereignis ausgelöst wurde.

Eine abschließende Bemerkung zu dem Kapitel über das Ausführen der Transitionen: Es wurde bereits kurz erwähnt, dass zwischen den Optionen eines `if`-Blocks nichtdeterministisch gewählt wird[7], [8]. Diese Eigenschaft ist wichtig, denn nach UML erfolgt die Auswahl der zu schaltenden Transition bereits nichtdeterministisch. An der nichtdeterministischen Wahl der Signale würden deterministische Ansätze sogar scheitern, da dadurch unendliche Squenzen von externen Signalen auftreten können. Solche Squenzen ließen sich natürlich nicht ‚durch ausprobieren', d.h. deterministisch prüfen.

4 Ein Beispiel

Das folgende Beispiel wurde der Arbeit von Toni Jusilla[1] entnommen. Es veranschaulicht die Übersetzung der Zustandsdiagramme in Proctypes. Zu sehen ist ein Beispiel eines verschachtelten UML-Zustandsdiagramms (Abb. 1) und dem dazugehörigen PROMELA-Code nach der Übersetzung (Abb. 2) inkluse der globalen Variablen Definitionen (Alg. 2).

```
byte MAXIDS = 2;
byte QSIZE = 2;
byte M_proc_id = 1;
channel M_inputqueues[MAXIDS] = [QSIZE] of {byte};
channel M_deferredqueues[MAXIDS] = [QSIZE] of {byte};
byte M_deferrable[QSIZE];
byte M_transitions[MAXIDS];
byte M_loopback[MAXIDS];
s_Top_Init = 1;
s_Top_Failure = 2;
s_Top_Main = 3;
s_Top_Main_R1 = 4;
s_Top_Main_R1_Init = 5;
s_Top_Main_R1_Final = 6;
s_Top_Main_R1_A = 7;
s_Top_Main_R1_A_busy = 8;
s_Top_Main_R1_B = 9;
s_Top_Main_R1_B_busy = 10;
s_Top_Main_R1_None = 11;
s_Top_Main_R1_busy = 12;
s_Top_Main_R1_final = 13;
s_Top_Main_R2 = 14;
s_Top_Main_R2_Init = 15;
s_Top_Main_R2_Final = 16;
s_Top_Main_R2_Loop = 17;
s_Top_Main_R2_Loop_busy = 18;
s_Top_Main_R2_None = 19;
s_Top_Main_R2_busy = 20;
s_Top_Main_R2_final = 21;
s_Top_Main_busy = 22;
byte signal_$tick = 1;
byte signal_e = 2;
```

Algorithm 2: Dieses Beispiel zeigt die Definitionen der globalen Variablen. Es bezieht sich auf das in Abb. 1 dargestellte Zustandsdiagramm.

Die Bedeutung des Codes aus Abb. 2 ist hier tabellarisch beschrieben:

5-7 Instanziierung der Zustandsvariablen mit den Werten von Zustandskonstanten (vlg. 3.2)

8 Deklaration der Instanzattribute (vlg. 3.2)

9 Deklaration der Signale und ihren Parametern

10 Mit dem Schlüsselwort `xr` wird sichergestellt, dass ein Prozess dieses Proctypes den alleinigen Zugriff auf seinen Nachrichtenkanal `inputqueues` besitzt.

11 Label für Rücksprung

12-21 Dieser `if`-Block enthält Einträge von Transitionen deren Quellzustände Pseudozustände sind. Jede Option enthält also die Bedingung der Transition sowie ihre Aktivitäten. Anschließend erfolgt ein Rücksprung zum Anfang des Blocks. Siehe dazu auch Abschnitt 3.2.

22-34 Dieser `if`-Block behandelt Transitionen, die von echten, d.h. einfachen oder komplexen, Zuständen ausgehen. (Beschrieben in Abschnitt 3.2 und 3.2)

27-31 Dieser `if`-Block gehört zu der Transition, die von **Main** zum Endzustand **Final** führt. Er prüft, ob die Bedingung der Transition erfüllt ist. Wenn dies nicht der Fall ist, wird dem Superzustand von **Main** eine Konstante ohne Endung zugewiesen, um anzudeuten, dass bereits eine Komplettierungs-Transition ausgelöst wurde. Der Hintergrund dieser Vorgehensweise ist in Abschnitt 3.2(Problem 5) erklärt.

35-39 Dieser `if`-Block dient dazu ein Ereignis zu laden. Dabei wird nichtdeterministisch zwischen der Ereignisliste oder einem externen Ereignis gewählt.

40-57 In diesem `if`-Block werden die ereignisgesteuerten Transitionen behandelt (vgl. 3.2,3.2)

41 Hier wird geprüft, ob der Zustand **Main** aktiv ist. Dazu sind zwei Vergleiche notwendig, weil er bereits eine Komplettierungs-Transition ausgelöst haben könnte. Je nachdem enthält seine Zustandsvariable einen anderen Wert (vgl. 3.2).

42-50 Dieser Block gehört zum Zustand **Main**. Hier wird als erstes geprüft, ob enthaltene, tiefere Transitionen (also Transitionen zwischen Unterzuständen) das Ereignis verwenden können, da diesen höhere Priorität beim Konsumieren von Ereignissen eingeräumt wird (vgl. 3.2).

51-55 Wenn das Ereignis nicht von tieferen Transitionen (siehe Zeilen 42-50) konsumiert wurde, wird dieser Block ausgeführt. Er prüft, ob das Ereignis auf die Transition passt und ob die Bedingung erfüllt ist. Wenn beides der Fall ist, wird geschaltet.

57 Rücksprung zu den Ereignissen, wenn das Signal nicht konsumiert wurde. Theoretisch könnte auch wieder an den Anfang des Prozesses gesprungen werden. Da sich aber an den Konfigurationen der Zustände nichts geändert hat, würde das Programm bis zum Holen des Ereignisses durchlaufen. Deswegen wird direkt zu den Ereignissen gesprungen und dort ein neues Ereignis geholt (3.2).

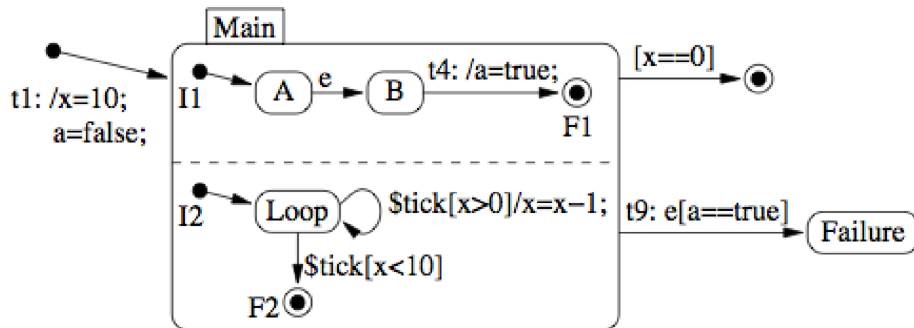

Abbildung 1. Beispiel eines verschachtelten Zustandsdiagramms. In der Mitte ist der komplexe Zustand **Main** zu sehen. Er teilt sich in zwei orthogonale Regionen (implizit **R1** und **R2** genannt). Die ausgehenden Transitionen von **Loop** können durch das externe Ereignis **$tick** ausgelöst werden.

Literatur

1. Jussila, T., Dubrovin, J., Junttila, T., Latvala, T., Porres, I.: Model checking dynamic and hierarchical UML state machines. In: MoDeV^{2}a: Model Development, Validation and Verification; 3rd International Workshop, Genova, Italy, October 2006. (2006) 94–110
2. Ahr, D.: Das UML-Metamodell. http://www.iwr.uni-heidelberg.de/groups/comopt-/teaching/uml/html/kapitel_4_800x600/index.htm
3. Visual Paradigm: Visual Paradigm for UML. http://www.visual-paradigm.com-/product/vpuml/
4. Telelogic: Telelogic Tau. http://www.telelogic.com/products/tau/
5. Information Society Technologies: Correct Development of Real-Time Embedded Systems. http://www-omega.imag.fr
6. Hitz, M., Kappel, G., Kapsammer, E., Retschitzegger, W.: UMLatWork Objektorientierte Modellierung mit UML2. Volume 3. Auflage. dpunkt Verlag (2005)
7. Faragó, D.: Model Checking with SPIN. (2008)
8. Rob Gerth and D.Faragó: Extraction from Concise Promela Reference. http://i12www.ira.uka.de/~mulbrich/teaching/formsys07/concise_promela.html (June 1997)

```
proctype M(int proc_id) {
5:     byte state_Top = s_Top_Init;
       byte state_Top_Main_R1 = s_Top_Main_R1_None;
       byte state_Top_Main_R2 = s_Top_Main_R2_None;
       byte x; bool a; /* Instance attributes of the owning class */
       byte trigger, p1, ...; /* for signal type & parameters */
10:    xr inputqueues[proc_id];

       evalcompletions:
       if /* Try to fire completion transitions from pseudostates */
       :: (state_Top == s_Top_Init) ->
         x = 10; a = false; state_Top = s_Top_Main_busy; state_Top_Main_R1 = s_Top_Main_R1_Init;
15:      state_Top_Main_R2 = s_Top_Main_R2_Init; goto evalcompletions;
       :: (state_Top_Main_R1 == s_Top_Main_R1_Init) ->
         state_Top_Main_R1 = s_Top_Main_R1_A; goto evalcompletions;
       :: (state_Top_Main_R2 == s_Top_Main_R2_Init) ->
         state_Top_Main_R2 = s_Top_Main_R2_Loop; goto evalcompletions;
20:    :: else -> skip;
       fi
       if /* Try to fire completion transitions from real states */
       :: (state_Top_Main_R1 == s_Top_Main_R1_B_busy) ->
         a = true; state_Top_Main_R1 = s_Top_Main_R1_Final; goto evalcompletions;
25:    :: (state_Top = s_Top_Main_busy && state_Top_Main_R1 == s_Top_Main_R1_final &&
         state_Top_Main_R2 == s_Top_Main_R2_final) ->
         if
         :: (x == 0) -> state_Top = s_Top_Final; state_Top_Main_R1 == s_Top_Main_R1_None;
           state_Top_Main_R2 == s_Top_Main_R2_None; goto evalcompletions;
30:      :: else -> state_Top = s_Top_Main; goto evalcompletions; /* Quiescing step */
         fi
       :: else -> skip; /* No completion transition was enabled */
       fi
       evaltriggers:
35:    if
       :: inputqueues[proc_id]?trigger,p1; /* Consume signal event (if any) */
       /* Non-deterministically create an external signal if in a state that can consume it */
       :: (state_Top_Main_R1 == s_Top_Main_R1_Loop) -> trigger = signal_$tick;
       fi
40:    if
       :: (state_Top == s_Top_Main_busy || state_Top == s_Top_Main) ->
         if
         :: (state_Top_Main_R1 == s_Top_Main_R1_A && trigger == signal_e && true) ->
           state_Top_Main_R1 = s_Top_Main_R1_B_busy; goto evalcompletions;
45:      :: (state_Top_Main_R2 == s_Top_Main_R2_Loop && trigger == signal_$tick && x > 0) ->
           x = x - 1; goto evalcompletions;
         :: (state_Top_Main_R2 == s_Top_Main_R2_Loop && trigger == signal_$tick && x < 10) ->
           state_Top_Main_R2 == s_Top_Main_R2_Final -> goto evalcompletions;
         :: else -> skip
50:      fi
         if /* Signal was not consumed by any substate of Main */
         :: (trigger == signal_e && a == true) -> state_Top_Main_R1 = s_Top_Main_R1_None;
           state_Top_Main_R2 = s_Top_Main_R2_None; state_Top = s_Top_Failure; goto evalcompletions;
         :: else -> skip
55:      fi
       :: else -> skip;
       fi
       goto evaltriggers; /* implicit consumption occurred */
}
```

Abbildung 2. Die Proctype-Definition in PROMELA zum Zustandsdiagramm aus Abb. 1. Das Zustandsdiagramm galt der Klasse **M**. Oben finden die Variablen-Deklarationen statt. Darunter die beiden `if`-Blöcke unter dem Label `evalcompletions` für Komplettierungs-Transitionen sowie der dritte `if`-Block unter `evaltriggers` mit vorgeschaltetem Block zum Holen der Ereignisse.