Universität des Saarlandes
Fachrichtung 6.2 – Informatik

Abbildung von StoCharts nach MoDeST

Christophe Boutter
boutter@ps.uni-sb.de
Dependable Systems and Software

Fortgeschrittenen-Praktikum

Angefertigt unter Leitung von
Prof. Dr.-Ing. Holger Hermanns

Betreut von
David Jansen und Holger Hermanns

7.4.06

Inhaltsverzeichnis

1 Einleitung

Motivation

Beim Design von komplexen Systemen möchte man sich das Zusammenspiel der Komponenten und den Ablauf vorab anschauen können. Dies erreicht man, indem man das System modelliert und mit dem Modell das Verhalten dann simuliert. Solche Simulationen können anhand der Sprache MoDeST [1] mit dem Tool MoTor [7] durchgeführt werden.

Um solche Modelle besser nutzen und verstehen zu können, findet man es als Benutzer einfacher, solche Modelle anhand einer grafischen Repräsentation zu benutzen. Eine weit verbreitete Repräsentation sind Statecharts [2], wir werden hier die Erweiterung StoCharts [4] betrachten. Diese ermöglicht es, Zustandsübergänge anhand von Wahrscheinlichkeiten zu steuern, so lassen sich z.B. menschlische Handlungen modellieren. In StoCharts kann man auch stochastische Systeme darstellen, welche große Vorteile bei der Modellierung von Zeitabhängigen Systemen bieten, so dass hier ein sehr breites Spektrum an Modellen zum Einsatz kommen kann.

MoDeST

MoDeST ist eine Modellierungssprache für stochastische zeitabhängige Systeme, die an der Universität Twente entwickelt wurde [1]. Sie dient dazu, Modelle für probabilistische, nicht-deterministische Systeme mit Realzeit-Bedingungen zu modellieren. Die dadurch entstandenen Modelle eignen sich gut dazu, mit einem Model-Checker verifiziert zu werden, oder mittels Diskrete-Ereignis-Simulation analysiert zu werden. Die MoDeST zugrunde liegenden semantischen Modelle sind sogenannte "Stochastische Zeitautomaten".

Das Tool MoTor [7] (*MOdest TOol enviRonment*) gibt uns die Möglichkeit MoDeST-Spezifikationen zu simulieren und die Ergebnisse zu analysieren. Die MoDeST-Syntax ist eng an die C-Syntax gehalten um dem Einstieg in die Sprache zu erleichtern. So finden wir z.B. ein `do`-Konstrukt für allg. Schleifen, Konstrukte für branching (`alt`), prob. branching (`palt`), sowie guards (`when`). Wichtig für diese Arbeit sind Ausnahmen und deren Behandlung anhand von `try`- und `catch`-Konstrukten ähnlich wie bei *Java*.

StoCharts

StoCharts [5] sind eine Erweiterung von UML-Statecharts [4], die hierarchische, zeitabhängige, probabilistische, sequentielle und/oder parallele Systeme auf grafische Weise zu modellieren erlauben. StoCharts erweitern UML-Statecharts unter anderem um stochastische Konstrukte, insbesondere um Timer, die durch Wahrscheinlichkeitsdistributionen gesteuert werden. Die zugrunde liegende semantischen Modelle sind Variationen der obigen "Stochastischen Zeitautomaten", bei denen Zustandsübergänge mit Ein- oder Ausgaben dekoriert sind.

Um StoCharts darzustellen verwenden wir das *Toolkit for Conceptual Modelling* (TCM) [9], genauer gesagt TSCD. TSCD ist einer von vielen Editoren, die in TCM zusammengefasst sind, und wurde speziell für StateCharts [2] und deren Erweiterungen konzipiert.

Übersicht über die Arbeit

Ziel dieser Arbeit war es, einen Übersetzer zu konzipieren, der eine StoCharts-Spezifikation nimmt (in unserem Fall eine TCM-Datei) und daraus eine MoDeST-Datei generiert. Diese Ausarbeitung stellt nun die Überlegungen und Arbeiten vor, die zu diesem Ziel geführt haben. Wir werden zunächst den grundlegenden Aufbau der Übersetzung sehen, in dem auch die Erweiterungen bzw. Überarbeitungen der Machbarkeitsstudie [3] erläutert werden. StoCharts bieten durch ihre flexible Semantik ein Unmenge an Möglichkeiten, Modelle darzustellen, wir mussten hier einige Einschränkungen vornehmen, um die Übersetzung realisierbar zu machen. Nach der Vorstellung dieser Einschränkungen wird eine formale Semantik präsentiert, die zeigt, wie die StoCharts-Konstrukte in MoDeST übersetzt werden. Im Anschluss werden Teile der Implementierung erklärt. Als Anwendungsbeispiel für den Übersetzer werden wir eine Studie über ETCS (European Train Controlling System) sehen und das Ergebnis mit dem Modell aus [3] vergleichen. Ein kleines Fazit rundet die Arbeit ab.

2 Grundlegendes

2.1 Designentscheidungen

Im Rahmen einer Machbarkeitsstudie [3] wurden schon einige Gedanken durchgespielt. Darauf basierend sollte diese Übersetzung die guten Ideen beibehalten und manche Ansätze verfeinern. So wurde die natürliche Entsprechung von States zu Prozessen beibehalten, aber die Darstellung von P-Transitionen ist nun eine ganz andere geworden.

Es wurde versucht, so nah wie möglich an der natürlichen Entsprechung der einzelnen Konstrukte zu bleiben. So werden Trigger und Events als MoDeST-Actions modelliert, obwohl sich daraus einiges Schwierigekeiten ergeben, wie z. B. ungewollte Synchronisation mit anderen Prozessen wegen des gemeinsamen Alphabets. Aus diesem Grund mussten wir *enabledness* garantieren und zu jeder P-Transition auch einen tau-Übergang hinzufügen.

P-Transitionen werden in MoDeST-Exceptions übersetzt, weniger weil P-Transitionen aussergewöhnliche Vorkommnisse sind, sondern weil Exceptions wichtige Eigenschaften haben, die für die Übersetzung sehr nützlich sind. Exceptions haben als Nebeneffekt, dass sie die Hierarchie der laufenden Prozesse beenden, bis sie durch ein `catch`-Statement gefangen werden. Dafür musste man eine komplizierte Platzierung des `try-catch`-Blocks anhand des Scopes der P-Transition in Kauf nehmen, damit nur die relevanten Prozesse beendet werden.

Es gibt jedoch kein Konstrukt in MoDeST, welches die Übersetzung von eingehenden Bordercrossing-P-Edges intuitiv machen würde. Aus diesem Grund wird den And- und Or-States repräsentierenden Prozessen ein `decision` genanntes Argument übergeben. Anhand dieser Variablen können die Alternativen eindeutig identifiziert werden. Im Standardverhalten hat `decision` den Wert 0.

Es musste noch ein besondere Familie an Variablen eingeführt werden um sog. *isIn*-Guards zu ermöglichen. *isIn*-Guards sind Guards, die überprüfen, ob ein bestimmter State gerade Teil der Zustandsmenge ist. Um dies möglich zu machen, wird jedem State eine *isIn*-Variable zugeordnet, die anzeigt, ob sich der State gerade in der Zustandsmenge befindet.

2.2 Einschränkungen

Da wir einige Einschränkungen vornehmen mussten, hauptsächlich aus technischen Gründen, möchte ich diese hier nochmal genauer vorstellen.

Hyperedges

Hyperedges sind bei unserer Übersetzung nicht erlaubt. Neben kleineren Problemen bei unserer Darstellung, ist das Hauptargument dass TCM [9] (noch) keine Hyperedges unterstützt und somit die Möglichkeit der graphischen Repräsentation gar nicht besteht. Dies bedeutet auch, dass Quell- und Zielzustandsmengen, wie sie in [5] definiert sind, immer nur ein Element enthalten können.

Wahrscheinlichkeitsdistributionen

Die Wahrscheinlichkeitsdistributionen die bei einem `after`-Konstrukt benutzt werden dürfen, müssen auf folgende drei eingeschränkt werden. Wir unterstützen *normale*, *uniforme* und *exponentielle* Distributionen, da MoTor [7] auch nur diese drei unterstützt.

"On entry" und "On exit" Trigger

Die "On entry" und "On exit" Trigger, welche beim Eintritt, bzw. beim Verlassen, eines States sind in der Implementierung nicht berücksichtigt, da es für TCM [9] keine festgelegte, einheitliche Darstellung gibt.

Aufbau der Transitionsbeschriftungen

Untenstehend sind Regeln für die Beschriftung der Transitionen angegeben. Ausdrücke, die sich in geschweiften Klammern befinden, sind als optional zu verstehen.

- Trigger t: `(a-zA-Z0-9)*` | `after(`dist`[`value`])`
- Verteilung dist: `EXP` | `UNIF` | `NORM`
- Wert value: numerischer Wert
- Event e: `(a-zA-Z0-9)*`
- Guard g: `isIn(`$Statename$`)` | `in(`$Statename$`)` | boolescher Ausdruck
- Gewicht w: numerischer Wert
- Triviale P-Transition: `{t}{[g]}/{e}`
- Eingehende P-Edge: `{t}{[g]}`
- Ausgehende P-Edge: w`/{e}`

2.3 Translationale Semantik

2.3.1 Einleitung

Es soll hier eine formale Semantik für die Übersetzung von StoCharts nach MoDeST eingeführt werden. An dieses Problem kann man von zwei Arten herangehen: entweder stateorientiert oder transitionsorientiert. Hier wird der stateorientierte Ansatz gewählt. In diesem Ansatz wird die Menge der States sequentiell abgearbeitet, und so werden die repräsentierenden MoDeST-Prozesse nacheinander aufgebaut.

Diese Arbeit übernimmt die Prozedur ϕ, die als Argument die Liste der States eines StoCharts bekommt und diese der Reihe nach betrachtet. Um den Prozessen ein Verhalten zu geben, werden die vom betrachteten State ausgehenden P-Transitionen übersetzt.

Diese Übersetzung wird von drei verschiedenen Prozeduren erarbeitet, je nach Statetyp. Für die Basic-States ist das die Prozedur δ, für die And-States ist das α und für Or-States ist das θ. Ein ganz wichtiger Punkt in dieser Arbeit ist, die Übergänge von einem Prozess zum nächsten an der richtigen Stelle zu machen. In den drei o. g. Prozeduren, welche die P-Transitionen übersetzen, sorgt die Prozedur σ dafür, dass die Aufrufe der Nachfolgeprozesse an der richtigen Stelle passieren.

Die Transitionsrepräsentation umfasst eine gewichtete Menge von Nachfolgestates. Dies ermöglicht es, eine P-Transition, wie sie in [5] definiert wird, als ein einziges Objekt darzustellen, so wie es von der Semantik auch vorgegeben ist. Obwohl man mit Hilfe von TCM leicht P-Transitionen zeichnen könnte, deren Transitionen verschiedene Scopes haben, wird dies ausdrücklich in [5] ausgeschlossen. So wird auch in dieser Arbeit pro P-Transition nur ein Scope betrachtet.

Es seien hier noch die beiden Mengen BC und $After - Trans$ erwähnt. BC ist die Menge der Bordercrossing-P-Transitionen. Diese Menge wird zu Beginn der Übersetzung mit allen vorkommenden Bordercrossing-P-Transitionen gefüllt. $After - Trans$ ist die statische Menge aller P-Transitionen, die ein `after`-Statement als Trigger haben. Diese P-Transitionen müssen gesondert betrachtet werden, da sie ein anderes Verhalten darstellen als die "normalen" P-Transitionen.

2.3.2 Definitionen

Ich will hier erläutern, wie im folgenden die States und P-Transitionen dargestellt werden sollen:

- ein Basic-State: $\bigcirc_{name}$
- ein AND-State: $\bigoplus_{name}$
- ein OR-State: $\bigotimes_{name}$
- eine P-Transition:
  $\rightarrow (src, \{(weight, dest, event)_1 \ldots (weight, dest, event)_n\}, trigger, guard, scope)$
  src und $dest$ müssen States aus einem der drei o.g. Typen sein. Eine Guard muss ein boolscher Ausdruck sein, und $scope$ ist der Scope der P-Transition, wie er in [5] definiert wird.
- eine P-Transition, die das stochastische After-Konstrukt als Trigger hat:
  $\rightarrow_{after} (src, \{(weight, dest, event)_1 \ldots (weight, dest, event)_n\}, dist, guard, scope)$
  Es gelten nahezu die gleichen Bedingungen wie für einfache P-Transitionen, nur dass hier anstelle eines Triggers eine Wahrscheinlichkeitsverteilung steht.

Es folgt der formalisierte Übergang der StoCharts Konstrukte, in der o. g. Darstellung. Dazu führen wir zunächst eine Funktion ϕ ein, die als Definitionsbereich die Menge aller möglichen StoCharts nimmt und als Bildbereich den dazugehörigen MoDeST-Code hat. Innerhalb dieser Prozedur werden noch weitere Prozeduren definiert, so dass die Übersetzung eine bessere Struktur bekommt.

Im Folgenden kürzen wir Bordercrossing-P-Transitionen mit Bordercrossing-Transitionen ab.

Die Notation innerhalb der Definitionen ist wie folgt zu interpretieren:

- `In dieser Schrift gedruckte Passagen sind endgültiger MoDeST-Code`
- Alles andere sind Zuweisungen oder Anwendungen von Prozeduren

2.3.3 Definition der Übergangsprozedur ϕ

Die Prozedur ϕ hat genau vier Fälle, nämlich je einen für jeden Statetyp und den Fall, in dem der betrachtete State ein Basic-State ist, der keine ausgehenden P-Transitionen hat. Diese Fälle werden wir der Reihe nach betrachten.

1. ϕ wird angewandt auf einen Basic-State mit Namen $Name$ ohne ausgehende P-Transition. Dieser wird übersetzt in:

   process $\Gamma(\bigcirc_{Name})$`() {`
       `{= isIn`$Name$`=1 =}; stop`
   `};`

   Γ nimmt Objekte aus einem StoChart (States, P-Transitionen) und gibt den Namen des entsprechenden MoDeST-Objekts (Prozess, Exception) zurück. Da der Knoten keine ausgehenden P-Transition hat, ist er ein Endzustand, so dass er mit dem Befehl `stop` blockiert.

2. Im zweiten Fall ist die Menge der ausgehenden P-Transitionen nicht leer, wir betrachten immer noch einen Basic-State mit Namen $Name$. Wenn wir die Menge der vom State $Name$ ausgehenden P-Transitionen mit $Trans$ bezeichnen, wird ϕ den State in folgenden Code umsetzen:

   `process` $\Gamma(\bigcirc_{Name})$ `{`
     `{= isIn`$Name$`=1` $afterInit(Trans)$ `=};`
   $\delta_{\bigcirc_{Name}}(Trans)$
   `};`

   Die Funktion $afterInit$ initialisiert die (Clock-) Variablen der P-Transitionen, die ein `after`-Statement als Trigger haben. Die Prozedur δ übersetzt die P-Transitionen, die im Scope von Name liegen in den dazugehörigen MoDeST-Code.

3. Der dritte Fall behandelt OR-States. Zusätzlich zu der Menge der von ihm ausgehenden P-Transitionen $Trans$ müssen wir noch die Menge aller möglichen Startkonfigurationen $StartConfig$, die durch eingehende Bordercrossing-Transitionen enstehen, betrachten. Daraus folgt folgender MoDeST-Code:

   `process` $\Gamma(\bigotimes_{Name})$ `{`
     `{= isIn`$Name$`=1` $afterInit(Trans)$ `=};`
   $\theta_{\bigotimes_{Name}}(Trans \cup BC, StartConfig)$
   `}`

   θ formuliert das Verhalten des OR-States aus, dazu braucht sie als Übergabeparameter die Menge der von Name ausgehenden P-Transitionen, sowie die noch nicht abgehandelten Bordercrossing-Transitionen, um überprüfen zu können, ob diese nun abgehandelt werden müssen. Ebenso braucht sie die Menge der Startkonfigurationen, um die Alternativen die durch eingehende Bordercrossing-Transitionen hervorgerufen werden, abhandeln zu können. Um diese Alternativen auswählen zu können, hat der Prozess ein Argument mit dem Namen `decision`. BC ist eine Referenz auf die Menge der Bordercrossing-Transitionen. Sie muss auch an θ übergeben werden, da manche dieser P-Transitionen als letzten State $Name$ verlassen könnten und daher der Prozess Name den Nachfolgeprozess aufrufen muss.

4. Den letzten Fall stellen die noch nicht behandelten AND-States dar. Wie schon bei den OR-States brauchen wir auch hier wieder zusätzlich zu der Menge $Trans$ der von ihm ausgehenden P-Transitionen, die Menge mit den möglichen Startkonfigurationen $StartConfig$.

   $\texttt{process } \Gamma(\bigoplus_{Name}) \texttt{ \{}$
   $\quad\texttt{\{= isIn}Name\texttt{=1} \quad afterInit(Trans) \texttt{ =\};}$
   $\alpha_{\bigoplus_{Name}}(Trans \cup BC, StartConfig)$
   $\texttt{\}}$

   α übersetzt das Verhalten des AND-States, auch hier werden wieder die von $Name$ ausgehenden P-Transitionen sowie die Bordercrossing-Transitionen übergeben, um zu überprüfen, ob diese im Scope von $Name$ liegen und der Nachfolgeprozess aufgerufen werden muss. Weiter müssen auch die Startkonfigurationen übergeben werden, damit die Prozedur die Alternativen, die durch eingehende Bordercrossing-Transitionen bedingt sind, erzeugen kann. Um diese Alternativen auswählen zu können, hat der Prozess ein Argument mit dem Namen `decision`, wie schon bei den OR-States gesehen.

2.3.4 Definition der Prozedur δ – Transitionseinleitung

Die δ-Prozedur ist bei allen Basic-States für die Transitionsbehandlung und deren Umsetzung in MoDeST-Code zuständig.
δ ist wie folgt definiert:

$$
\begin{aligned}
\delta_{Sta}(Trans) \quad = \quad & if\ \exists t \in Trans.scope(t) = parent(Sta) \\
& then\ \texttt{try\{do\{:: alt\{}\ \gamma_{Sta}(Trans)\texttt{\}\}\}}\ \sigma_{Sta}(Trans) \\
& else\ \texttt{do\{::alt\{}\ \gamma_{Sta}(Trans)\ \texttt{\}\},}
\end{aligned}
$$

Zunächst wird im if-Statement überprüft, ob eine der betrachteten P-Transitionen im gleichen Scope liegt wie der betrachtete State, das bedeutet, dass diese P-Transition als letztes diesen State verlässt. Danach wird σ aufgerufen. σ erzeugt aus diesen P-Transitionen die dazugehörigen `catch`-Statements und die resultierenden Prozessaufrufe.

Existieren keine solchen P-Transitionen, fällt der Aufruf von σ weg, dementsprechend braucht man hier auch kein `try`-Statement mehr, da sich der `try-catch`-Block in einem anderen Prozess befindet. Es wird dann lediglich γ verwendet um aus den P-Transitionen das eigentliche Verhalten zu erhalten. γ ist u. a. dafür zuständig die `throw`-Statements auszuformulieren, sowie die enabledness Konditionen zu garantieren.

2.3.5 Definition der Prozedur σ – Transitionsübergang

Die σ-Prozedur ist eine zentrale Prozedur, die bei allen Statetypen angewendet wird. Sie organisiert die `catch`-Statements in der Prozessdefinition, so dass die Exceptions die durch die P-Transitionen verursacht werden im richtigen Scope gefangen werden.
σ ist wie folgt definiert:

$$
\begin{aligned}
\sigma_{Sta}(Trans) \quad = \quad & forall\ t \in Trans. \\
& if\ scope(t) = parent(Sta)\ \&\&\ isChild_r(source(t), Sta) \\
& then\ \texttt{catch}\ \ \Gamma(t)\texttt{\{} \\
& \texttt{\{=}left(t)\texttt{=\};\ tau\ palt\ \{} \\
& forall\ (weight, dest, event) \in getDestSet(t)\ do \\
& \quad \texttt{:}weight\texttt{:}\ \ event\texttt{;}\ \ \Gamma(dest) \\
& \}
\end{aligned}
$$

Zuerst wird für jede P-Transition überprüft, ob sie sich im Scope des betrachteten States befindet und ihren Ursprung innerhalb des betrachteten States hat. Wenn dies der Fall ist, wird ein `catch`-Statement mit Hilfe von Γ generiert. In diesem Statement werden sämtliche `isIn` Variablen von der Funktion *left* zurückgesetzt. Die Funktion *getDestSet* gibt die Menge der gewichteten Nachfolgestates zurück. Diese werden dann in ein `palt`-Statement geschrieben. Bei nicht-trivialen P-Transitionen, werden die Gewichtungen entsprechend gesetzt. Bei trivialen P-Transitionen, ist das Gewicht des einzigen Nachfolgestates immer 1.

2.3.6 Definition der Prozedur γ – Enabledness garantieren

Die γ-Prozedur übernimmt die Guard-Konstruktion, so dass Ambiguitäten in der Wahl des Nachfolgeprozesses gleich ausgeräumt werden und durch die Form der Guards nicht zugelassen werden. Es wird ferner auch Enabledness garantiert, indem eine Flower-Konstruktion für den Prozess angefertigt wird. So wird garantiert, dass das Modell nicht "hängen" bleibt, da in MoDeST die Actions von parallelen Prozessen ja blockierend sind.
γ ist wie folgt definiert:

$$
\begin{aligned}
\gamma_{Sta}(Trans) \quad = \quad & forall\ t \in Trans. \\
& scope(t) = parent(Sta). \\
& if\ t \in After - Trans \\
& then\ guard := guard(t)\&\&clock(t) >= timer(t) \\
& else\ guard := guard(t)\&\&(\neg \bigvee \{g | (a, g, scope') \in \tau(trigger(t)) \\
& \wedge isChild(scope(t), scope')\}) \\
& \texttt{::\ when}\ \ (guard)\ \ trigger(t)\texttt{;}\ \ \texttt{throw}(\Gamma(t)) \\
& \\
& if\ t \notin After - Trans\ then
\end{aligned}
$$

$$
\begin{aligned}
&forall\ tr \in \{tr | \exists t.tr = trigger(t) \wedge t \in Trans \wedge scope(t) = parent(Sta)\}\\
&guard := (\neg \bigvee \{guard(t) | t \in Trans \wedge tr = trigger(t)\\
&\wedge scope(t) = parent(Sta)\}) \vee\\
&(\bigvee \{g | (a, g, scope') \in \tau(tr) \wedge isChild(Sta, scope') \wedge \nexists t.t \in Trans\\
&\wedge guard(t) = g \wedge scope(t) = parent(Sta)\})\\
&\texttt{::\ when}\texttt{(}guard\texttt{)}\,tr
\end{aligned}
$$

Es wird für jede Transition überprüft, ob es sich um eine Tansition mit einem `after`-Trigger handelt. Ist dies zutreffend, wird die Guard um die Überprüfung der dazugehörigen Clock-Variablen erweitert.

Im anderen Fall wird die Guard der P-Transition um die negierte Veroderung der Guards der P-Transitionen, die den gleichen Trigger, aber eine höhere Priorität haben, erweitert. So ist gewährleistet, dass die P-Transition nur dann genommen wird, wenn keine P-Transition mit höherer Priorität enabled ist und genommen werden müsste. Die Menge der P-Transitionen, die den gleichen Trigger haben wie die betrachtete P-Transition, wird durch die Funktion τ geliefert.

Diese so zusammengestellte Guard wird dann vor den Übergang gestellt. Um die Blockierung durch Synchronisation über das gemeinsame Alphabet zu umgehen, wird noch eine weitere "dummy"-Transition hinzugefügt, und zwar für jeden Trigger der von Sta ausgehenden P-Transitionen eine. Die "dummy"-Transition umfasst die negierte Veroderung der Guards der P-Transitionen mit dem gleichen Trigger, sowie die Veroderung der Guards der P-Transitionen mit dem gleichen Trigger aber einer höheren Priorität.

2.3.7 Definition der Prozedur θ – Or-States

Die θ-Prozedur generiert den Code, der das Verhalten von OR-States simulieren soll.
θ ist wie folgt definiert:

$$
\begin{aligned}
\theta_{Sta}(Trans, StartConfig) \quad = \quad & \texttt{alt \{}\\
& for(i = 0; i < \#StartConfig; i++)\\
& \texttt{::when (decision == } i\texttt{)}\\
& if\ \exists t \in Trans.\ scope(t) = parent(Sta)\\
& then\ \texttt{try \{ par \{::}\ \Gamma(\#i(StartConfig))\\
& \quad \texttt{:: do \{:: alt \{}\\
& \gamma_{Sta}(Trans)\ \texttt{\} \} \} \}}\\
& \sigma_{Sta}(Trans)\\
& else\ \texttt{par \{::}\ \Gamma(\#i(StartConfig))\\
& \quad \texttt{:: do \{:: alt \{}\\
& \gamma_{Sta}(Trans)\ \texttt{\} \} \}}\\
& \texttt{\},}
\end{aligned}
$$

Für jede der möglichen Startkonfigurationen wird eine eigene Alternative erstellt, was hier durch eine `for`-Schleife dargestellt wird. Diese Alternativen werden durch den Wert der Variable `decision` unterschieden, wobei `decision` der einzige Parameter des Sta darstellenden Prozesses ist. `decision` wird in der Prozessdeklaration durch Γ eingeführt. Dann wird die übliche Unterscheidung getroffen, ob es nur Bordercrossing-Transitionen gibt, und dementsprechend Exceptionhandling eingebaut oder nicht.

Was nun etwas überraschend erscheinen mag, ist, dass ein `par`-Statement benutzt wird. Dieses brauchen wir, um den Prozess des Kind-States aufzurufen, aber dennoch die enabledness der von Name ausgehenden P-Transitionen zu erhalten. $\#i(StartConfig)$ ist die Projektion auf das i-te Element der Liste $StartConfig$, und mit dem Aufruf von Γ wird der entsprechende Prozess gestartet. Parallel zu diesem Prozess werden dann noch in bekannter Weise die P-Transitionen abgehandelt.

2.3.8 Definition der Prozedur α – And-States

Die α-Prozedur behandelt das Verhalten der AND-States und nutzt dabei das parallel Konstrukt von MoDeST aus, um die Parallelität innerhalb des States zu simulieren.
α ist wie folgt definiert:

$$
\begin{array}{rcl}
\alpha_{Sta}(Trans, StartConfig) & = & \texttt{alt \{} \\
& & for(i = 0; i < \#StartConfig; i++) \\
& & \texttt{:: when (decision==}i\texttt{)} \\
& & if\ \exists t \in Trans.scope(t) = parent(Sta) \\
& & \quad \texttt{try \{ par \{:: do \{:: alt \{} \\
& & \gamma_{Sta}(Trans)\ \texttt{\} \}} \\
& & for(j = 0; j < \#(\#i(StartConfig)); j++) \\
& & \quad \texttt{::}\ \Gamma(\#j(\#i(StartConfig))) \\
& & \quad \texttt{\} \}}\ \sigma_{Sta}(Trans) \\
& & else\ \texttt{par \{:: do \{:: alt \{} \\
& & \gamma_{Sta}(Trans)\ \texttt{\} \}} \\
& & for(j = 0; j < \#(\#i(StartConfig)); j++) \\
& & \quad \texttt{::}\ \Gamma(\#j(\#i(StartConfig))) \\
& & \}
\end{array}
$$

Die Prozedur α ist θ sehr ähnlich, nur dass hier die Liste $StartConfig$ eine Liste von Listen ist. Jedes Element von $StartConfig$ enthält ja mehrere States die parallel aktiv sind. Deshalb ähneln sich die beiden Prozeduren auch, bis auf den Aufruf einer zweiten `for`-Schleife, welche die einzelnen parallelen Prozesse startet. Dies geschieht mit einer zweifachen Projektion auf $StartConfig$.

2.3.9 Definition der Funktion $afterInit$ – Variablen Initialisierung

Die $afterInit$-Funktion hat zur Aufgabe, die für die vorkommenden After-P-Transitionen erforderlichen Variablen zu initialisieren. Dies sind insbesondere ein Timer, dessen Wert aus einer Wahrscheinlichkeitsverteilung herausgezogen wird, und eine Clock-Variable.

$afterInit$ ist wie folgt definiert:

$$afterInit(Trans) \;=\; \forall t \in Trans \wedge t \in After - Trans.$$
$$,\; timer(t)=dist(t),\;\; clock(t)=0$$

2.3.10 Hilfsfunktionen

An dieser Stelle werden noch die Hilfsfunktionen definiert, die in der Semantik vorkamen, jedoch nicht genauer erklärt worden sind. Meist sind ihre Namen auch schon selbsterklärend.

- Die Funktion Γ steht für den Kontext, sie gibt zu einem Objekt den passenden Namen bzw. den dazugehörigen Prozess zurück.
- Die Funktion τ liefert zu einem Trigger die Menge der P-Transitionen mit dem gleichen Trigger.
- Die Funktion $parent$ wird auf einen Knoten angewendet und gibt den Vaterknoten des betrachteten Knotens zurück.
- Die Funktion $scope$ wird auf Transitionen angewendet und gibt den scope der Transition zurück. Dies wird hauptsächlich dazu genutzt, um Bordercrossing-Transitionen zu erkennen.
- Die Funktionen $left$ wird auf Transitionen angewendet und setzt alle “isIn” Variablen zurück, die auf Knoten verweisen, die von der betrachteten Transition verlassen werden.
- Die Funktion $guard$ gibt zu einer Transition die dazugehörige Guard zurück.
- Die Funktion $trigger$ liefert den Trigger einer Transition zurück.
- Die Funktion $dist$ hat als Rückgabewert die Wahrscheinlichkeitsverteilung einer After-Transition.
- Die Funktion $timer$ gibt zu einer After-Transition die entsprechende Timervariable zurück.
- Die Funktion $clock$ gibt zu einer After-Transition die entsprechende Clockvariable zurück.
- Die Funktion $getDestSet$ gibt eine Menge aus Triplen der Gewichte von Nachfolgestates, der Nachfolgestates selbst und der Events zurück.
- Die Funktion $isChild$ testet, ob ein State in der Kinderhierarchie des betrachteten States liegt. $isChild_r$ ist dabei die reflexive Version von $isChild$.

3 Realisierung

3.1 Grundlegendes

MosML [6] wurde als Implementierungssprache für den Übersetzer gewählt. Der Grund ist, dass Standard-ML die Möglichkeit bietet, relativ einfach die mathematischen Definitionen, die in Abschnitt 2.3 eingeführt wurden, in Code zu verwandeln und die mathematischen Strukturen weiterzuverwenden. Der Programmcode ist größtenteils portabel und ermöglicht so auch neben einem Linux-Binary in kurzer Zeit ein Windows-Binary zu erstellen, so dass der Übersetzer plattformunabhängig ist.

MosML bietet zudem hervorragende Lexer- (mosmllex) und Parsergeneratoren (mosmlyac). Anhand dieser wurde der umständliche Prozess, einen eigenen Parser für die von TCM [9] erzeugten Dateien zu erstellen, erheblich vereinfacht.

Das Programm wurde in einem Drei-Stufen-Modell realisiert. Die erste Stufe ist das Einlesen der TCM-Datei mit dem Parser. Die zweite Stufe verarbeitet die Ausgabe des Parsers zu Datentypen die der formalen Darstellung von StoCharts [5] sehr gleichen. Diese Datentypen werden in Listen (State- und P-Transitionslisten) zusammegefasst und in zwei Maps (State- und P-Transitionsmaps) eingetragen, um die Suche zu vereinfachen. In der dritten Stufe wird die MoDeST-Ausgabedatei generiert.

Im folgenden werden wir die Kernideen und die Schwierigkeiten der letzten beiden Stufen kennenlernen.

3.2 Stufe 2 – Vorbereitung

Diese Programmstufe soll aus einzelnen Zustandsbeschreibungen, Transitionen und Linien eine kohärente State- und P-Transitionsdarstellung erstellen. Der Großteil der Arbeit besteht darin, Daten aus mehreren Objekten zusammenzuführen. Die Hauptschwierigkeit ist, aus den And-Linien entstandene Or-States korrekt zu generieren und den Scope der Kind-States neu festzulegen. Dies wird anhand einer Überprüfung der Koordinaten der States realisiert. In diesem Teil werden auch die P-Edges mit dem gleichen Ursprung zu einer P-Transition zusammengefasst.

3.3 Stufe 3 – Übersetzung

In dieser Stufe halten wir uns sehr eng an die Vorgaben aus Abschnitt 2.3. Die Prozedurnamen wurden übernommen und die mathematischen Definitionen umgesetzt. Da die Definitionen schon erklärt wurden, sollen hier nur die besonderen Schwierigkeiten nochmals erläutert werden.

Bevor wir zu den eigentlichen Problemen kommen, sollen hier noch einige Erläuterungen gemacht werden. Es existieren zwei `Intmap`-Datenstrukturen, die zum einen alle

States (genannt `All_s`) und zum anderen alle P-Transitionen (genannt `All_t`) beinhalten und anhand ihrer jew. Id indizieren. Zum besseren Verständnis folgen die Typdeklarationen.

```
(* Basic types for our StoChart representation *)
type name = string

(* The integer represents the id of the state
 * which is the scope *)
type scope = int

(* States w/ their name and TCM id *)
datatype Sta = Bas of name * int
         | And of name * int
         | Or of name * int

type trigger = string
type guard = string
type event = string
type weight = string

(* Transitions w/ their source, destination, trigger,
 * guard, event and scope *)
datatype Transition =
      T of int * Sta * (weight * Sta
          * event) list * trigger * guard * scope
    | AT of int * Sta * (weight * Sta
          * event) list * trigger * guard * scope

(* Start configuration alternative for an and or or state
 * the int is the id of the transition which triggers this
 * config *)
type StartCfg = int * Sta list

(* States w/ their name, TCM id, transition set and maybe
 * child set *)
datatype States = B of name * int * Transition list * scope
          | A of name * int * Transition list * Sta list
            * StartCfg list * scope
          | O of name * int * Transition list * Sta list
            * StartCfg list * scope
```

3.3.1 Zielfindung

Eine der kompliziertesten Funktionen der Übersetzung ist `get_dest`. Die Schwierigkeit liegt, wie so häufig in dieser Arbeit, bei dem Scope-Problem. Die Funktion `get_dest` wird an zwei Stellen eingesetzt, einmal in der Funktion `sigma` zur Bestimmung des

Nachfolgestates und einmal zur Bestimmung der Alternativen in einem And- oder Or-State. Die beiden Aufrufe unterscheiden sich deshalb, weil `sigma` nur den State mit dem niedrigsten Scope aufrufen muss und in der Alternativenbestimmung der State mit dem nächst höheren Scope (womöglich mit der korrekten Alternative) betrachtet werden muss.

Die Suche nach dem richtigen State wird bottom-up geführt, von einem hohen Scope (Zielstate der P-Transition) zu einem niedrigen Scope entlang der Vaterhierarchie des Zielstates. Die Rekursion wird beendet, wenn der betrachtete State den richtigen Scope hat, also mit der Variablen `scope` übereinstimmt, oder sich im Root-Scope befindet. Als Ausgabe wird der korrekte Prozessaufruf gegeben.

Es folgt der Code für einen Basic-State. Im `let`-Teil wird die Rekursion vorbereitet. Im letzten `if`-Statement wird dann der Scope überprüft.

```
fun get_dest (sta as (Bas (n, i))) scope trans_id=
    let
        val node = Intmap.retrieve (All_s, i)
        val sc = parent node
        val par = if sc = 0
                  then sta
                  else
                      case Intmap.retrieve (All_s, sc) of
                          (B (n, i, _, _)) => Bas (n, i)
                        | (A (n, i, _, _, _, _)) => And (n, i)
                        | (O (n, i, _, _, _, _)) => Or (n, i)
    in
        if sc = scope
        then if n = ""
             then "p_" ^ Int.toString(i) ^ "()"
             else n ^ "()"
        else get_dest par scope trans_id
    end
  ...
```

Es folgt der Fall eines And-States. Hier muss im Fall, dass der betrachtete State auch der gesuchte ist, noch die richtige Alternative gefunden werden. Deshalb werden die Startkonfigurationen des States nach der Id der P-Transition durchsucht und die Variable `decision` entsprechend angepasst.

```
  ...
| get_dest (sta as (And (n, i))) scope trans_id =
    (case Intmap.retrieve (All_s, i) of
         (A (name, id, trans, children, configs, sc)) =>
         if scope = sc
         then
             let
                 fun find_config (nil) counter = 0
                   | find_config (y::ys) counter =
                     if (#1(y)) = trans_id then counter
                     else find_config ys (counter+1)
                 val decision = find_config configs 0
```

```
        in
            if name = ""
            then "p_" ^ Int.toString(id)
                 ^ "(" ^ Int.toString(decision) ^ ")"
            else name ^ "(" ^ Int.toString(decision)
                 ^ ")"
        end
    else let
            val par = if sc = 0
                      then sta
                      else
                          case Intmap.retrieve
                               (All_s, sc)
                          of
                              (B (n, i, _, _))
                              => Bas (n, i)
                            | (A (n, i, _, _, _, _))
                              => And (n, i)
                            | (O (n, i, _, _, _, _))
                              => Or (n, i)
        in
            get_dest par scope trans_id
        end)
...
```

Der Fall für einen Or-State ist ähnlich und braucht daher nicht näher erläutert zu werden.

3.3.2 Enabledness garantieren

Da die Prozesse bei einem Transitionsübergang nicht blockieren sollen, müssen wir für jeden Prozess enabledness garantieren. Somit umgehen wir die blockierende Synchronisation über das gemeinsame Alphabet.

Um dies zu erreichen, ist eine komplizierte Guardkonstruktion nötig, so wie sie in 2.3.6 erläutert wird. Zunächst wird dafür eine Liste (`blossom_guard_parts`) aufgebaut welche zu jedem der im betrachteten State vorkommenden Trigger die entsprechende Disjunktion der Guards der höher priorisierten P-Transitionen enthält.

In einer eigenen Funktion (`blossoming`) wird dann die Negierung der eigentlichen Guard(s) vorgenommen, und die vorher bestimmten Teile werden zusammengesetzt.

```
...
fun blossoming nil str = str
  | blossoming (x::xs) str =
    let
        val trig = trigger x
        val (pos, neg) = List.partition
                         (fn y => trig = trigger y)
                         (x::xs)
```

```
        val transition_guards =
        foldl (fn (y,a) => if guard y = "" then a
                           else guard y ^ " || " ^ a)
              (guard (hd pos)) (tl pos)

        val guard =
        case List.find
             (fn y => #1(y) = trig)
             (!blossom_guard_parts)
        of
           SOME z => if #2(z) = ""
                     then "(!(" ^ transition_guards ^ "))"
                     else "((!(" ^ transition_guards
                          ^ ")) && " ^ #2(z) ^ ")"
         | NONE => raise bigErrorInBlossoming
        val result = ":: when " ^ guard ^ " " ^ trig ^ "\n"
    in
        blossoming neg (result ^ str)
    end
val blossom_list = List.filter
                   (fn x => case x of
                            AT (_, _, _, _, _, _) => false
                          | T (_, _, _, _, _, _) => true)
                   (x::xs)
val blossoms = blossoming blossom_list ""
   ...
```

Die übergebene Liste ist die Liste der P-Transitionen des betrachteten States. Es werden zuerst die weiteren P-Transitionen gesucht, die denselben Trigger haben wie die gerade betrachtete. Aus dem Ergebnis wird dann der Guardteil der P-Transitionen des States zusammengesetzt und im Anschluss die Teile der höher priorisierten P-Transitionen.

3.3.3 Zurücksetzen der isIn-Variablen

Um die `isIn`-Variablen konsistent mit den tatsächlich aktiven States zu halten, müssen diese nicht nur bei Eintritt in einen Prozess, sondern auch bei Terminierung durch eine Exception zurückgesetzt werden. Dies übernimmt die Funktion `unset_left`.

```
fun unset_left tr =
    let
        fun find_last_parent sta scope=
            let
                val state = Intmap.retrieve (All_s,id_s(sta))
                val par = parent (state)
            in
                if  par = scope
                then sta
```

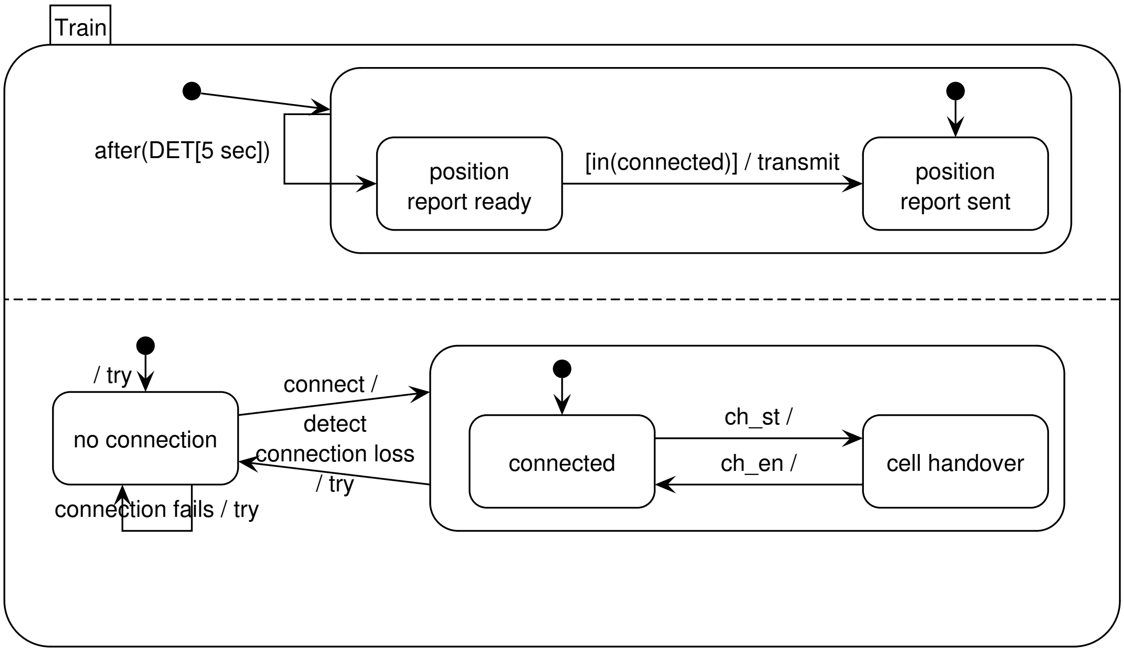
```
                    else case Intmap.retrieve (All_s,par) of
                              A (name,id,_,_,_,_) =>
                               find_last_parent (And(name,id)) scope
                            | O (name,id,_,_,_,_) =>
                               find_last_parent (Or(name,id)) scope
                            | B (name,id,_,_) =>
                               find_last_parent (Bas(name,id)) scope
                end
            val last_parent = find_last_parent (source tr) (scope tr)
        in
            unset_left' last_parent (scope tr)
        end
```

In `unset_left` wird der State mit dem kleinsten Scope gesucht, der von der betrachteten P-Transition verlassen wird. Die Hilfsfunktion `unset_left'` schreibt dann den String, in dem die `isIn`-Variablen von `last_parent` und dessen Kindstates zurückgesetzt werden.

4 Anwendungsbeispiel

In dem hier betrachteten Anwendungsbeispiel sehen wir die Modelle aus dem AVACS Technical Report [3] wieder, welche schon für die Machbarkeitsstudie benutzt wurden. Wir werden dann die Ergebnisse vergleichen und die Unterschiede sowie deren Gründe erläutern.

Wir betrachten den Sender aus dem AVACS Technical Report. Der entsprechende Sto-Chart ist im Folgenden zu sehen.



Wir betrachten einen Teil des aus dem obigen Modell resultierenden Codes, welcher das untere Kind des And-States `Train` darstellt.

```
process p_49(int decision) {
    {= isIn49=1 =};
    alt { :: when (decision==0)par { :: no_connection()}
    }
}

process cell_handover() {
    {= isIncell_handover=1 =};
    try {do {::alt {
        :: when (true) ch_en; throw (trans_17)
        :: when (!(true)) ch_en
            }}
        } catch trans_17 { {= isIncell_handover=0 =};
          tau palt {:1: tau; connected()
```

```
        } }
}

process p_8(int decision) {
    {= isIn8=1 =};
    alt { :: when (decision==0) action_try { par {
            :: connected()
            :: do { :: alt {
                :: when (true) detect_connection_loss;
                   throw (trans_20)
                :: when (!(true)) detect_connection_loss
               }}
            }
    } catch trans_20 {
      {= isIn8=0 , isInconnected=0 , isIncell_handover=0 =};
      tau palt {:1: action_try; no_connection()
      } }
    }
}

process connected() {
    {= isInconnected=1 =};
    try {do {::alt {
        :: when (true) ch_st; throw (trans_16)
        :: when (!(true)) ch_st
            }}
        } catch trans_16 { {= isInconnected=0 =};
          tau palt {:1: tau; cell_handover()
          } }
}

process no_connection() {
    {= isInno_connection=1 =};
    try {do {::alt {
        :: when (true) connect; throw (trans_19)
        :: when (true) connection_fails; throw (trans_21)
        :: when (!(true)) connect
        :: when (!(true)) connection_fails
            }}
        } catch trans_21 { {= isInno_connection=0 =};
          tau palt {:1: action_try; no_connection()
          } }
          catch trans_19 { {= isInno_connection=0 =};
          tau palt {:1: tau; p_8(0)
          } }
}
```

Zum Vergleich hier der Code wie er während der Machbarkeitsstudie [3] entwickelt wurde, und zwar für den gleichen Teil wie schon oben betrachtet.

```
process Connection_status() {
  do{::
    do{:: action_try;
      alt{:: connection_fails
          :: connect {= connected=1 =}; break
      }
    };

    // Connected
    try{
      par{:: do{:: h_start {= connected=0 =};
                   h_end {= connected=1 =}
             }
          :: detect_connection_loss {= connected=0 =};
             throw retry
      }
    }
    catch (retry){tau}
  }
}
```

Auf den ersten Blick ist direkt zu erkennen, dass der vom Übersetzer generierte Code viel länger ist als der von Hand überlegte. Dies ist auf mehrere Faktoren zurückzuführen. Der erste ist, dass der Code in [3] nur das Verhalten modelliert und der Code dementsprechend auch optimiert ist. der Übersetzer erzeugt auch pro State einen eigenen MoDeST-Prozess, was ebenfalls den Code in die Länge zieht. Desweiteren werden die Transitionen auch immer in MoDeST-Exceptions übersetzt, was für jede P-Transition einen `try-catch`-Block zur Folge hat. In [3] musste auch keine enabledness beachtet werden, welche im Übersetzer generisch erzeugt wird, da Blockieren im Modell nicht vorkommen kann. Wenn man diese Faktoren beachtet, erkennt man leicht warum der automatisch vom Übersetzer erzeugte Code empfindlich länger ist. Die Vorteile die dadurch entstehen sind bei Beachtung der generischen Anwendung des Übersetzers auch direkt zu erkennen. Die Begründung der Entscheidungen wurde ja schon in 2.1 geliefert.

5 Fazit

Während dieser Arbeit wurde ein Übersetzer [8] entwickelt, mit dem man StoCharts in MoDeST-Code übersetzen kann. Einige Aspekte waren schon von vornherein klar, andere mussten erst in langen Diskussionen entwickelt werden. So wurde Enabledness erst relativ spät in die Arbeit eingeführt, hat sich allerdings als absolut notwendig erwiesen.

Das Ziel der Arbeit war es, einen generischen Übersetzer zu schaffen, mit dem die bis dahin statischen StoChart-Modelle durch Simulation zum Leben erweckt werden können. Dies ist nicht uneingeschränkt gelungen, aber der vom Übersetzer gelieferte Code ist eine gute Grundlage für eine Simulation. Die Änderungen dieser Grundlage sind meist nur syntaktischer Natur, so dass sie auch relativ einfach von Hand zu bewältigen sind.

In einer weiterführenden Arbeit kann sich vorstellen den Übersetzer in die MoTor Umgebung [7] einzubetten, so dass man das Laden der Quelldateien umgehen kann. Ein neuer Release von TCM [9] steht an, hier sollen z. B. auch Hyperedges möglich sein. Man könnte dann auch diese in den Übersetzer einbauen.

Literaturverzeichnis

[1] Pedro R. D'Argenio, Holger Hermanns, Joost-Pieter Katoen, and Ric Klaren. MoDeST – a modelling and description language for stochastic timed systems. In Luca de Alfaro and Stephen Gilmore, editors, *PAPM-PROBMIV*, volume 2165 of *LNCS*, pages 87–104. Springer, 2001.

[2] David Harel. Statecharts: A visual formulation for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[3] Holger Hermanns, David N. Jansen, and Yaroslav S. Usenko. A comparative reliability analysis of ETCS train radio communications. Reports of SFB/TR 14 AVACS 2, SFB/TR 14 AVACS, February 2005. ISSN: 1860-9821, http://www.avacs.org.

[4] David N. Jansen, Holger Hermanns, and Joost-Pieter Katoen. A QoS-oriented extension of UML statecharts. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *«UML» 2003 : the unified modeling language*, volume 2863 of *LNCS*, pages 76–91. Springer, 2003.

[5] David Nicolaas Jansen. *Extensions of statecharts with probability, time, and stochastic timing*. Proefschrift, Universiteit Twente, Inmarks, Bern, October 2003. ISBN 3–9522850–0–5.

[6] MosML homepage: `http://www.dina.kvl.dk/~sestoft/mosml.html`.

[7] MoToR homepage: `http://fmt.cs.utwente.nl/tools/motor/`.

[8] StoChart converter homepage:
`http://www.ps.uni-sb.de/~boutter/fopra.php`.

[9] TCM homepage: `http://www.cs.utwente.nl/~tcm/`.