Universität Freiburg
Institut für Informatik
Abteilung für Grundlagen der Künstlichen Intelligenz

SS 2007

Diplomarbeit

Transformation von SPS- in Realzeitautomaten

vorgelegt von Diana Dragojević

18. Mai 2007

Erstgutachter: Prof. Dr. Bernhard Nebel
Zweitgutachter: Dr. habil. Henning Dierks

Danksagung Ein großes Dankeschön an alle, die zum Gelingen dieser Arbeit beigetragen haben. Prof. Dr. Nebel danke ich dafür, dass er diese Arbeit ermöglicht und unterstützt hat. Mein besonderer Dank gilt außerdem Sebastian Kupferschmid für die ausgezeichnete Betreuung, die unzähligen Ratschläge und das Durchsehen der Arbeit. Dr. habil. Henning Dierks danke ich für die einwöchige Einführung in die in dieser Arbeit verwendeten Automaten und die Aufklärung meiner Unklarheiten bezüglich Moby/RT.

Erklärung Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, den 18. Mai 2007 Diana Dragojević

Inhaltsverzeichnis

1. Einleitung

Eingebettete Systeme, wie z. B. speicherprogrammierbare Steuerungen, kommen heutzutage in sehr vielen Bereichen des täglichen Lebens vor. Eine speicherprogrammierbare Steuerung, kurz SPS, kann man sich als einen kleinen Computer mit einem Realzeit-Betriebssystem vorstellen, der von einem Bus lesen und Ausgaben auf diesen schreiben kann. Die Einsatzbereiche solcher Systeme reichen von Fahrzeugtechnik bis hin zur Steuerung in der Verfahrenstechnik oder in Kernkraftwerken. In all diesen Bereichen kann das fehlerhafte Funktionieren dieser Komponenten verheerende Folgen haben. Daher ist es wichtig, dass diese Systeme beweisbar korrekt und zuverlässig funktionieren. Die Modellprüfung (engl. model checking) ist eine Technik, mit der man automatisch überprüfen kann, ob ein System bezüglich einer gegebenen Eigenschaft korrekt funktioniert. Ein SPS-Automat ist ein Automatenmodell, mit dem man das Verhalten einer SPS formal beschreiben kann. SPS-Automaten können mit Moby/RT[1] entwickelt und überprüft werden. Um einen SPS-Automaten zu überprüfen wird dieser in ein Netzwerk von erweiterten Realzeitautomaten transformiert. Dies ist notwendig, da es keinen Modellprüfer gibt, der direkt SPS-Automaten als Eingabesprache akzeptiert. Für Realzeitautomaten gibt es Werkzeuge wie z. B. Uppaal mit denen eine automatische Modellprüfung durchgeführt werden kann.

In der vorliegenden Arbeit wird eine neue Transformation von SPS-Automaten in Realzeitautomaten beschrieben. Bisher wird bei dieser Transformation von Moby/RT für jede, den Guard einer Transition erfüllenden Belegung, eine Transition im Realzeitautomat erstellt. Diese Art der Übersetzung war notwendig, da zu der Zeit, als Moby/RT implementiert wurde, Uppaal nur sehr eingeschränkte Ausdrücke in den Guards akzeptierte. Sie bläht jedoch den übersetzten Automaten unnötig auf, und wesentliche strukturelle Eigenschaften gehen in der Übersetzung verloren. Die hier vorgestellte Übersetzung nutzt aktuelle Neuerungen von Uppaal aus, die es ermöglichen die Guards der SPS-Automaten zu erhalten. Somit ist eine natürlichere Transformation möglich, die meistens mit weniger Transitionen auskommt.

Die geringere Anzahl an Transitionen wirkt sich positiv auf die benötigte Zeit für die Modellprüfung aus, denn je weniger Transitionen, desto weniger Guards muss der Modellprüfer auswerten, um festzustellen welche Transition genommen werden kann.

[1] **MO**delling of distri**B**uted s**Y**stems/**R**eal**T**ime

Die hier vorgestellte Transformation und die von MOBY/RT führen zu äquivalenten Systemen, da sie sich lediglich in der Anzahl der Transitionen unterscheiden die von einem Zustand zu einem anderen führen und die Ereignisse, die zu einem Zustandübergang führen bleiben die gleichen. Bei gleicher Eingabe führen beide Automaten zur gleichen Ausgabe.

Im folgenden Kapitel werden SPS-Geräte beschrieben gefolgt von einer kurzen Einführung in UPPAAL. Anschließend werden die Realzeitautomaten definiert, über welche die Semantik der SPS-Automaten definiert wird und die das Ziel der Transformation darstellen. Im darauffolgenden Kapitel werden SPS-Automaten und Netzwerke von SPS-Automaten eingeführt. Anschließend wird das Werkzeug MOBY/RT und das von MOBY/RT unterstüzte SIM-Format vorgestellt, welches der Ausgangspunkt der Transformation ist. Danach folgt die Beschreibung der Transformation eines Netzwerkes von SPS-Automaten in einen Realzeitautomaten. Schließlich werden dann die Ergebnisse vorgestellt, gefolgt von einer abschließenden Zusammenfassung.

2. Grundlagen

Da es in dieser Arbeit um die Übersetzung von speicherprogrammierbaren Steuerungen in Realzeitautomaten geht, sollen diese beiden Begriffe in diesem Kapitel kurz erklärt werden. Eine SPS ist ein elektronisches Gerät und ein Realzeitautomat ein formales Automatenmodell. Der Zusammenhang zwischen diesen beide Begriffen ergibt sich daraus, dass das Verhalten einer SPS formal durch einen so genannten SPS-Automaten beschrieben werden kann. Die Semantik dieser SPS-Automaten ist dann über die Semantik von Realzeitautomaten definiert.

2.1. Speicherprogrammierbare Steuerungen

Eine speicherprogrammierbare Steuerung [5] (engl. programmable logic controller) ist eine Art kleiner Computer und besteht aus einem Prozessor, einem EPROM[1], einem Bussystem, einer Energieversorgung und Schnittstellen für Peripheriegeräte. Wie der Name schon verrät, ist eine SPS programmierbar und somit sehr flexibel einsetzbar. Seit den 60er Jahren sind sie Standard in der Automatisierungstechnik. Weltmarktführender Hersteller von SPS-Systemen ist heute Siemens.

Eine SPS führt ständig wiederkehrend einen Zyklus, bestehend aus dem Einlesen der aktuell anliegenden Werte, dem Berechnen neuer Werte gemäß des vom Benutzer gespeicherten Programms und dem Ausgeben dieser Werte aus. Auf der SPS befindet sich ein Realzeit-Betriebssystem, welches das zyklische Verhalten der Maschine vorschreibt. Abbildung 2.1 veranschaulicht dieses Verhalten. Wenn die SPS eingeschaltet wird, dann wird sie zuerst initialisiert und dann wird das im Speicher befindliche Programm zyklisch immer wieder ausgeführt. Ein Programmende gibt es also nicht. Jeder Zyklus der SPS besteht aus den folgenden drei Phasen:

- Einlesen (Polling): In dieser Phase werden die Werte, die am Eingabebus anliegen, eingelesen und in ein spezielles Register der SPS gespeichert.

- Berechnen (Computing): Danach wird das Programm genau ein mal ausgeführt. Hierbei können beliebige Berechnungen über die Werte, die im Speicher und den speziellen Registern für die Eingabe liegen, unter Berücksich-

[1] Erasable Programmable Read-Only-Memory, wörtlich: Löschbarer, programmierbarer Nur-Lese-Speicher

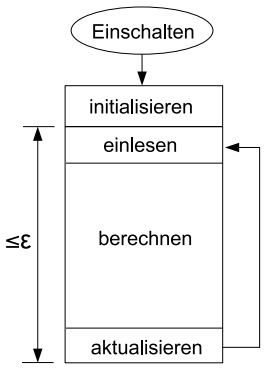

Abbildung 2.1.: Das zyklische Verhalten einer SPS.

tigung von Zeitrestriktionen erfolgen. Außerdem können die Werte für den Ausgabebus, die in speziellen Registern liegen, geändert werden.

- Aktualisieren (Updating): In der letzten Phase werden die Inhalte der Ausgaberegister auf den Bus geschrieben.

Die tatsächlich Dauer eines Zyklus hängt von der Anzahl der Ein- und Ausgaben und der Berechnungen, die während dieses Zyklus stattfinden ab. Da meistens nicht in jedem Zyklus die gleichen Berechnungen stattfinden, ist die tatsächliche Zykluszeit nicht für alle Zyklen gleich, aber immer höchstens so groß wie die maximale Zykluszeit ε. Ist für ein gegebenes Programm eine kleinere Zykluszeit erwünscht, so muss eine schnellere SPS verwendet werden.

Die maximale Reaktionszeit einer SPS, auf der ein bestimmtes Programm abläuft, beträgt zweimal die maximale Zykluszeit dieser SPS. Dies liegt daran, dass sich im schlechtesten Fall die Eingabe direkt nach Beginn eines Zyklus ändert und somit also für diesen Zyklus nicht berücksichtigt wird. Die Reaktion, d. h. die Ausgabe entsprechender Ausgangssignale, wird erst nach Beendigung eines zweiten Zyklus, in der Aktualisierungsphase, wirksam. Der Vorteil dieses Mechanismus ist, dass die Daten während eines Zyklus konsistent sind. Ein Beispiel ist in Abbildung 2.2 zu sehen. Hier erfolgt die Änderung der Eingabe von e zu $\neg e$ direkt nach Beginn des ersten Zyklusses, so das im ersten Zyklus noch mit e gerechnet wird und am Ende dieses Zyklusses (nach ε Zeiteinheiten) die Ausgabe $\delta(e)$ ausgegeben wird. Erst im darauffolgenden Zyklus wird der neue Wert $\neg e$ eingelesen und am Ende dieses Zyklusses, also nach 2ε Zeiteinheiten wird auf dies Eingabe reagiert, indem die Ausgabe $\delta(\neg e)$ ausgegeben wird.

In Abbildung 2.2 ist auch gut zu erkennen, dass Eingaben die nur kurz anliegen, übersehen werden können. Die SPS reagiert auf den Eingabewechsel von e zu $\neg e$ während des dritten Zykluses nicht. Um dies zu vermeiden muss sichergestellt werden, das die maximale Zykluszeit kleiner ist als die minimale Zeit in der eine

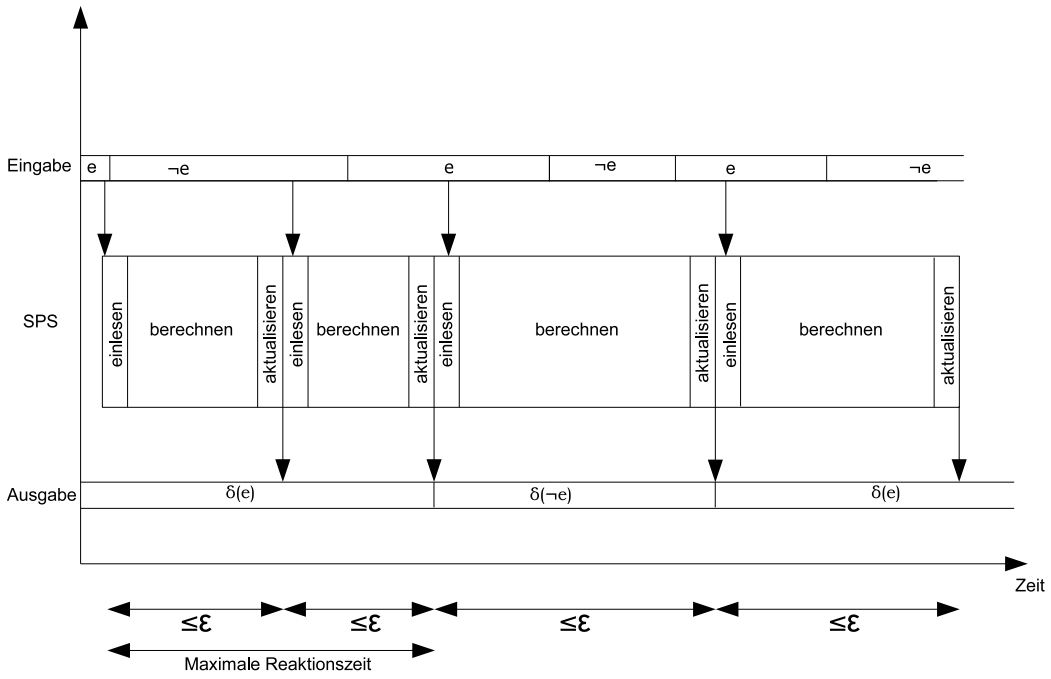

Abbildung 2.2.: Die maximale Reaktionszeit ist 2-mal die maximale Zykluszeit

Eingabe konstant ist.

SPS-Automaten [8] beschreiben eine SPS in der Berechnungsphase und wurden im Rahmen des UniForM[2]-Projekts [13] entwickelt. Sie stellen eine abstrakte Sicht auf eine SPS dar:

- Die Eingabe des Automaten ist die Eingabe, die von der SPS gelesen wird.
- Ein Zustand des Automaten entspricht dem internen Status der SPS.
- Die Ausgabe des Automaten ist die berechnete Ausgabe der SPS.
- Im SPS-Automat wird in jedem Zyklus genau eine Transition genommen.
- Die Timervariablen des Automaten modellieren das zeitliche Verhalten der SPS

Die Semantik eines SPS-Automaten wird über einen Realzeitautomaten definiert, welcher im anschließenden Kapitel vorgestellt werden.

[2]UniForM **Uni**verselle Entwicklungsumgebung für **For**male **M**ethoden.

2.2. Realzeitautomaten

Realzeitautomaten (engl. timed automata) wurden zuerst von Alur und Dill [2, 3] vorgeschlagen. Sie definierten sie als eine Erweiterung von sogenannten *Büchi-automaten*, um eine zeitliche Komponente. Ein Büchiautomat ist ein endlicher Automat, der unendliche Wörter akzeptiert. Ein Wort wird akzeptiert, wenn der Automat beim Lauf über das Wort unendlich oft *akzeptierende* Zustände besucht. Zeit wird bei Realzeitautomaten über eine endlichen Menge von reellwertigen Variablen, sogenannten *Uhren* modelliert. Eine Uhr kann auf null zurückgesetzt werden und gibt die verstrichene Zeit an, seit sie das letzte mal zurückgesetzt wurde. Alle Uhren gehen synchron mit konstanter Geschwindigkeit. Kantenbedingungen sogenannte *Guards*, die über diese Uhren formuliert sind, können benutzt werden, um das Verhalten des Automaten zu beschränken: eine Kante kann nur genommen werden, wenn die Werte der Uhren den Guard erfüllen. Henzinger et al. [12] schlugen eine Vereinfachung, nämlich *Timed Safety Automata* vor. Diese Version wird heue, mit einigen Erweiterungen, von vielen Werkzeugen wie z. B. Uppaal [14] oder Kronos [16] zur Verifikation von Realzeitautomaten benutzt.

2.2.1. Uppaal

Uppaal [4] wurde gemeinsam von der Unitversität **Upp**sala und der Unitversität **Aal**borg entwickelt. Es ist ein Programm zur Verifizierung von Netzwerken von erweiterten Realzeitautomaten und ist unter `http://www.uppaal.com/` frei erhältlich. Die hier relevanten Erweiterungen und Bestandteile der erweiterten Realzeitautomaten werden nun im Folgenden erläutert.

- **Beschränkte Integervariablen**, das sind Integervariablen, dessen Wertebereich innerhalb eines bestimmten Intervalls liegt. Deklariert werden sie mit `int[`*min*`,`*max*`]` *Variablenname*, wobei *min* die untere und *max* die obere Schranke ist.

- **Boolesche Variablen** werden mit `bool` *Variablenname* deklariert und sind Typkompatibel mit Integers. Ein Integerwert von 0 wird zu `false` ausgewertet und alle anderen Integerwerte werden zu `true` ausgewertet. Der boolesche Wert `true` wird zu dem Integerwert 1 und `false` wird zu dem Integerwert 0 ausgewertet. Der Vergleich `5 == true` z. B. wird zu `false` ausgewertet, da `true` zum Intergerwert 1 ausgewertet wird, wie in C++. Intern werden sie zu Bounded Integervariablen mit einem Bound von `[0,1]` dargestellt. D. h. immer wenn hier von Bounded Integervariablen gesprochen wird, dann werden damit auch die booleschen Variablen gemeint.

- **Binäre Synchronisation** ermöglicht es zwei Prozesse mit Hilfe von Kanälen (channels) zu synchronisieren. Hierbei werden die Kanten mit einer Syn-

chronisationsbeschriftung (Eingabeaktion `c?` und Ausgabeaktion `c!`) beschriftetet und der Kanal `c` wird durch `chan c` deklariert. Es können immer nur 2 Kanten miteinander synchronisieren. Wenn eine Kante mit einer Synchronisationsbeschriftung versehen ist, dann kann sie erst dann genommen werden, wenn es eine Kante in einem anderen Prozess gibt, die mit einer entsprechenden Synchronisationsbeschriftung gekennzeichnet ist und auch genommen werden kann. Dann werden beide Übergänge gleichzeitig ausgeführt, d. h.der aktuelle Knoten beider Prozesse wird geändert. Der Update Ausdruck an einer Kante mit dem Synchronisationslabel `c!` wird vor dem Update Ausdruck an einer mit `c?` gekennzeichneten synchronisierenden Kante ausgeführt. D. h. die Updates werden zwar hintereinader aber zum gleichen Zeitpunkt ausgeführt.

- **Knoten vom Typ committed**: Ein Zustand ist committed wenn er mindesten einen Knoten vom Typ committed enthält. In einem committed Zustand darf keine Zeit vergehen und die nächste Transition muss eine ausgehende Kante von mindestens einem der committed Knoten enthalten. (Dies ist strenger als die Invariante x<=0, weil hier immer eine Kante eines committed Knoten genommen werden muss. Gilt nur die Invariante x<=0, dann kann im selben Zeitpunkt beliebig viele andere Kanten genommen werden bis irgendwann immer noch im selben Zeitpunkt eine ausgehende Kante eines Zustandes mit dieser Invariante genommen wird)

- **Initialisierer** werden benutzt, um die Integer- und booleschen Variablen zu initialisieren (z. B. `int[1,10] i = 2;` oder `bool b = true;`).

- Eine **Invariante** ist eine Bedingung, die einem Knoten zugeordnet ist, die erfüllt sein muss, solange sich das System in einem Zustand aufhält, der diesen Knoten enthält. Eine Invariante ist eine Konjunktion von Ausdrücken der Form `(x<e)` *op* `b` oder `(x<=e)` *op* `b` oder Teilausdrücken hiervon, wobei `x` eine Uhr ist, `e` ist ein Ausdruck der zu einem Integer ausgewertet werden kann, *op* $\in$ {`and, or`} und `b` ist ein beliebiger boolescher Ausdruck über Integervariablen oder Integers.

- Ein **Guard** ist eine Bedingung, die einer Kante zugeordnet ist, die erfüllt sein muss, damit sie genommen werden darf. Ein Guard ist eine Konjunktion von booleschen Ausdrücken der Form `(x` *comp* `e)` *op* `b` oder `(x-y` *comp* `e)` *op* `b` oder Teilausdrücke hiervon, wobei *comp* $\in$ {`<, <=, ==, >, >=`}, `x` und `y` Uhren sind und `e`, *op* und `b` die gleiche Bedeutung wie bei der Invariante haben.

- Ein **Assignment** ist eine durch Komma separierte Liste von Zuweisungen der Form `v=e`, wobei `v` eine Variable und `e` ein beliebiger, zu `v` typkonformer Ausdruck ist. Das Assignment ist einer Kante zugeordnet, d. h., wenn

die Kante genommen werden kann, dann werden alle Zuweisungen in der Assignment-Liste ausgeführt.

Die von UPPAAL unterstützte Anfragesprache ist eine Teilmenge von TCTL [1] (Timed Computation Tree Logic). TCTL ist eine Erweiterung von CTL [10] (Computation Tree Logic) mit der Zeit. Im Gegensatz zu TCTL ist in UPPAAL die Verschachtelung von Pfadformeln nicht erlaubt.

2.2.2. Realzeitautomaten in Uppaal

Im Folgenden werden jetzt Realzeitautomaten, wie sie von UPPAAL unterstützt werden eingeführt [4].

Zunächst werden typisierte Variablen und Belegungen definiert, von denen auch später noch Gebrauch gemacht wird.

Definition 2.1 (Variablen, Uhren und Belegung)**.** Sei X eine Menge von Variablen, wobei jedem $x \in X$ eine Menge t_x von Werten (ein Typ) zugeordnet ist. Eine *Uhr* ist eine Variable x mit $t_x = \mathbb{R}_{\geq 0}$. Uhren sind die einzigsten Variablen mit unendlichem Wertebereich.

Eine *Belegung von* X ist eine Abbildung v, die jeder Variable $x \in X$ einen Wert $v(x) \in t_x$ zuordnet. Weiter sei $\mathcal{V}(X)$ die Menge der Belegungen von X. Weiter sei für eine Belegung $v \in \mathcal{V}(X)$, eine Variable $x \in X$ und $w \in t_x$ die Belegung $v[x \mapsto w] \in \mathcal{V}(X)$ wie folgt definiert, für $y \in X$ sei

$$v[x \mapsto w](y) := \begin{cases} v(y) & \text{falls } y \neq x \\ w & \text{falls } y = x \end{cases}$$

Sei $v' \in \mathcal{V}(Y)$ eine Belegung der Menge Y von Variablen. Eine Belegung $v \leftarrow v' \in \mathcal{V}(X \cup Y)$ ist wie folgt definiert, für alle $v \in X \cup Y$ sei

$$(v \leftarrow v')(x) := \begin{cases} v(x) & \text{falls } x \in X \setminus Y \\ v'(x) & \text{falls } x \in Y \end{cases}$$

Für $d \in \mathbb{R}_{\geq 0}$ sei weiter die Belegung $v + d \in \mathcal{V}(X)$ definiert als

$$(v + d)(x) := \begin{cases} v(x) + d & \text{falls } t_x = \mathbb{R}_{\geq 0} \\ v(x) & \text{sonst} \end{cases}$$

■

Die Belegungsfunktion $v[x \mapsto w]$ überschreibt den Werte von x mit w. Die Belegungsfunktion $v \leftarrow v'$ überschreibt die Werte von v, die im Wertebereich von v' enthalten sind, mit den Werten von v'. Schließlich addiert die Belegungsfunktion $v + d$ zu jeder Uhr die Zeit d.

Definition 2.2 (Uhrenbedingung, -ausdruck, Invariante und Assignment)**.** Sei V eine Menge von Variablen und C eine Menge von Uhren. Eine *Uhrenbedingung (über (V, C))* ist ein boolescher Ausdruck der Form $(x \sim e)$ oder $x - y \sim e$, wobei $x, y \in C$ Uhren seien, e ein Integerausdruck über V und $\sim \in \{<, \leq, =, \geq, >\}$ sei. Ein *Uhrenausdruck (über (V, C))* ist ein Ausdruck φ gemäß der Grammatik

$$\varphi ::= \beta \mid \psi \mid \psi \diamond \beta \mid \varphi \wedge \varphi$$

wobei ψ eine Uhrenbedingung, β ein boolescher Ausdruck über Variablen aus V und $\diamond \in \{\wedge, \vee\}$ sei. Eine *Invariante* ist ein Uhrenausdruck dessen Uhrenbedingungen nur aus Vergleichen $\sim \in \{<, \leq\}$ besteht. Weiter ist ein *Assignment (über (V, C))* eine durch Komma separierte Liste von Zuweisungen der Form $x = e$, wobei $x \in V \cup C$ und e ein zu x typkonformer Ausdruck über Variablen aus V sei.

Die Menge der Uhrenausdrücke über Variablen aus V und Uhren aus C wird mit $\mathcal{U}(V, C)$ bezeichnet. Die entsprechende Teilmenge der Invarianten wird als $\mathcal{I}(V, C)$ und die entsprechende Menge der Assignments als $\mathcal{A}(V, C)$ bezeichnet. Das *leere Wort* wird mit λ bezeichnet.

Sei $v \in \mathcal{V}(V \cup C)$. Für $g \in \mathcal{U}(V, C)$ sei $g(v) \in \{\top, \bot\}$ die Auswertung von g bezüglich der Belegung v auf übliche Weise definiert; dabei stehe $\top$ für wahr und $\bot$ für falsch. Für $a \in \mathcal{A}(V, C)$ sei $a(v) \in \mathcal{V}(V \cup C)$ die Auswertung von a bezüglich v wie folgt rekursiv definiert:

- $\lambda(v) := v$
- $(x = e)(v) := v[x \mapsto e(v)]$ für $x \in V \cup C$ und einen zu x typkonformen Ausdruck e
- $(a_1, a_2)(v) := a_2(a_1(v))$ für $a_1, a_2 \in \mathcal{A}(V, C)$

■

Definition 2.3 (Realzeitautomaten)**.** Ein *Realzeitautomat* $\mathcal{T}$ ist ein Tupel

$$\mathcal{T} = (L, V, C, S, E, I, \mathcal{C}, l_0, v_0),$$

hierbei ist

- L eine endliche Menge von *Knoten* (engl. Locations),
- V eine Menge von Variablen,
- C eine Menge von Uhren,
- S eine endliche Menge von *Synchronisationsbeschriftungen*, die aus Paaren von Ausgabe- und Eingabeaktionen $s!$ und $s?$ sowie einer internen Aktion τ besteht.

- $E \subseteq L \times S \times \mathcal{U}(V,C) \times \mathcal{A}(V,C) \times L$ eine endliche Menge von *Kanten*. Man schreibt auch $l \xrightarrow{s,g,a} l'$, mit $l, l' \in L$, $s \in S$, $g \in \mathcal{U}(V,C)$ und $a \in \mathcal{A}(V,C)$,
- $I \colon L \to \mathcal{I}(V,C)$ eine Funktion, die jedem Knoten eine Invariante zuordnet.
- $\mathcal{C} \subseteq L$ eine endliche Menge von *comitted Locations*,
- $l_0 \in L$ ist der *Startknoten* und
- $v_0 \in \mathcal{V}(V \cup C)$ ist die *Startbelegungen*, wobei $v_0|_C = 0$.

Außerdem muss $(I(l_0))(v_0) = \top$ gelten. ■

Die Semantik eines Realzeitautomaten wird über ein Transitionssystem definiert. Ein Zustand eines solchen Systems bestehen aus dem aktuellen Knoten des Automaten und den aktuellen Werten der Uhren und Variablen. Zustände können entweder durch eine *Zeittransition* (engl. timed transition), eine Transition, die einfach Zeit verstreichen lässt, oder durch eine *diskrete Transition*, eine Transition, die der Ausführung einer Kante im Automaten entspricht, miteinander verbunden sein. Formal lässt sich die Semantik folgendermaßen definieren.

Definition 2.4 (Semantik eines Realzeitautomaten)**.** Die Semantik eines Realzeitautomaten $\mathcal{T} = (L, V, C, S, E, I, \mathcal{C}, l_0, v_0)$ wird über das *Transitionssystem* $(Z, z_0, \to)$ definiert. Hierbei ist $Z := L \times \mathcal{V}(V \cup C)$ die Menge der Zustände, $z_0 = (l_0, v_0) \in Z$ ist der Startzustand und $\to \subseteq Z \times (\mathbb{R}_{\geq 0} \cup S) \times Z$ ist die Transtitonsrelation, die durch folgende Regeln definiert ist:

- $(l, v) \xrightarrow{d} (l, v + d)$ für $d \in \mathbb{R}_{>0}$ falls $(I(l))(v + t) = \top$ für alle $0 \leq t \leq d$ und $l \notin \mathcal{C}$
- $(l, v) \xrightarrow{\tau} (l', v')$ falls es eine Kante $(l, \tau, g, a, l') \in E$ gibt mit $g(v) = \top$, $v' = a(v)$ und $(I(l'))(v') = \top$

Ein *Lauf von* $\mathcal{T}$ ist eine Sequenz $(s_i, z_i)_{0<i<n} \in (\mathbb{R}_{>0} \cup S) \times Z$ mit $n \in \mathbb{N} \cup \{\infty\}$ und $z_{i-1} \xrightarrow{s_i} z_i$ für alle $0 < i < n$ und falls $n \neq \infty$ ist, so existiert kein $z \in Z$ und $s \in \mathbb{R}_{>0} \cup S$ mit $z_{n-1} \xrightarrow{s} z$, und falls $n = \infty$, so ist $|\{i \in \mathbb{N} \mid s_i \in S\}| = \infty$. ■

Oft bestehen Realzeitsysteme aus mehreren Komponenten, die auf die eine oder andere Weise miteinander kommunizieren können. Die *Parallelkomposition* ist ein Werkzeug, um solch ein System mit Realzeitautomaten modellieren zu können, ohne einen riesigen Produktautomaten bilden zu müssen. Realzeitautomaten lassen sich durch den CSS-Operator zur Parallelkomposition [15] zu einem Netzwerk von Realzeitautomaten verknüpfen. In so einem Netzwerk können Automaten durch binäre Synchronisation gleichzeitig eine Transition ausführen, oder asynchron über gemeinsame Variablen kommunizieren. Die Semantik eines solchen Netzwerks wird im Folgenden definiert.

Definition 2.5 (Netzwerk von Realzeitautomaten)**.** Ein *Netzwerk von Realzeitautomaten* ist ein Tupel $\mathcal{N} = (\mathcal{T}_1, \ldots, \mathcal{T}_n)$ mit $\mathcal{T}_i = (L_i, V, C, S, E_i, I_i, \mathcal{C}_i, l_{0,i}, v_0)$. Das Transitionssystem für das Netzwerk $\mathcal{N}$ wird definiert als $(Z, z_0, \rightarrow)$, wobei $Z := L_1 \times \cdots \times L_n \times \mathcal{V}(V \cup C)$, $z_0 := (\overline{l_0}, v_0) \in Z$ und $\rightarrow \subseteq Z \times (\mathbb{R}_{>0} \cup S) \times Z$ wobei

- $(\bar{l}, v) \xrightarrow{d} (\bar{l}, v + d)$ für $d \in \mathbb{R}_{>0}$ falls $(I(\bar{l}))(v + t) = \top$ für alle $0 \leq t \leq d$ und $l_i \notin \mathcal{C}_i$ für alle $1 \leq i \leq n$

- $(\bar{l}, v) \xrightarrow{\tau} (\bar{l}[l_i'/l_i], v')$ falls es eine Kante $(l_i, \tau, g, a, l_i') \in E_i$ gibt mit $g(v) = \top$, $v' = a(v)$, $(I(\bar{l}[l_i'/l_i]))(v') = \top$ und entweder $l_i \in \mathcal{C}_i$ oder $l_j \notin \mathcal{C}_j$ für alle $1 \leq j \leq n$

- $(\bar{l}, v) \xrightarrow{s} (\bar{l}[l_i'/l_i][l_j'/l_j], v')$ falls $i \neq j$ ist und es Kanten $(l_i, s!, g_i, a_i, l_i') \in E_i$ und $(l_j, s?, g_j, a_j, l_j') \in E_j$ gibt mit $(g_i \wedge g_j)(v) = \top$, $v' = a_j(a_i(v))$, $(I(\bar{l}[l_i'/l_i][l_j'/l_j]))(v') = \top$ und entweder $l_i \in \mathcal{C}_i$ oder $l_j \in \mathcal{C}_j$ oder $l_k \notin \mathcal{C}_k$ für alle $1 \leq k \leq n$

Der Lauf von $\mathcal{N}$ ist genauso definiert wie für einzelne Realzeitautomaten. ■

Beispiel 2.6 (Lampe)**.** Abbildung 2.3 zeigt die Modellierung einer Lampe und einem Benutzer mittels zweier Realzeitautomaten. Hierbei modelliert der linke Automat die Lampe und der rechte den Benutzer. Die Initialzustände beider Automaten sind jeweils durch die doppelt eingekreisten Knoten gekennzeichnet. Der Benutzer kann entweder Zeit verstreichen lassen, oder er kann auf den Lichtschalter drücken (die Kante mit der Synchronisationsbeschriftung „`drücken!`“). Ist die Lampe aus, so wird diese durch einmaliges Drücken des Lichtschalters eingeschaltet. Wird dann, innerhalb von zwei Zeiteinheiten, ein weiteres mal auf den Lichtschalter gedrückt, so wird das Licht heller. Ansonsten wird durch ein weiteres Drücken auf den Lichtschalter die Lampe wieder ausgeschaltet.

Durch die Synchronisationsbeschriftungen an den Kanten werden beide Automaten (oder Prozesse wie sie in Uppaal heißen) synchronisiert. D. h. der linke Automat kann eine Kante mit der Beschriftung „`drücken?`“ nur dann nehmen wenn der rechte Automat gleichzeitig eine Kante mit der Beschriftung „`drücken!`“ nehmen kann (und umgekehrt). Durch die Synchronisation der beiden Automaten wird auf jede Änderung des simulierten Benutzerverhaltens reagiert. Der linke Automat kann wie folgt formal definiert werden:

$$\mathcal{T}_1 = (L, V, C, S, E, I, \mathcal{C}, l_0, v_0)$$

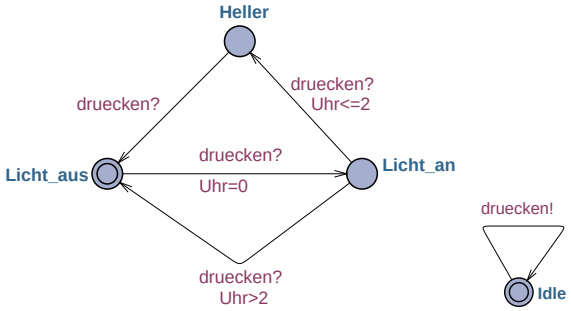

Abbildung 2.3.: Modellierung einer Lampe mittels Realzeitautomaten

hierbei ist:

$$
\begin{aligned}
L &= \{\texttt{Licht\_aus}, \texttt{Licht\_an}, \texttt{Heller}\} \\
V &= \emptyset \\
C &= \{\texttt{Uhr}\} \\
S &= \{\texttt{druecken!}, \texttt{druecken?}\} \\
E &= \{(\texttt{Licht\_aus}, \texttt{druecken?}, \top, \texttt{Uhr=0}, \texttt{Licht\_an}), \\
&\qquad (\texttt{Licht\_an}, \texttt{druecken?}, \texttt{Uhr<=2}, \lambda, \texttt{Heller}), \\
&\qquad (\texttt{Licht\_an}, \texttt{druecken?}, \texttt{Uhr>2}, \lambda, \texttt{Licht\_aus}), \\
&\qquad (\texttt{Heller}, \texttt{druecken?}, \top, \lambda, \texttt{Licht\_aus})\} \\
I &: L \to \{\top\} \\
\mathcal{C} &= \emptyset \\
l_0 &= \texttt{Licht\_aus} \\
v_0 &= [\texttt{Uhr} \mapsto 0]
\end{aligned}
$$

Zusammen mit dem rechten Automat, der schlicht die Form

$$\mathcal{T}_2 = (\{\texttt{Idle}\}, V, C, S, \{(\texttt{Idle}, \texttt{druecken!}, \top, \lambda, \texttt{Idle})\}, \top, \emptyset, \texttt{Idle}, v_0)$$

hat, erhält man das Netzwerk $\mathcal{N} = (\mathcal{T}_1, \mathcal{T}_2)$, das die Lampe samt Benutzer modelliert.

3. SPS-Automaten

In der Dissertationsschrift von Dierks [8] werden drei Typen von SPS-Automaten beschrieben: (einfache) SPS-, HSPS- (hierarchische SPS-) und GSPS-Automaten (generalisierte SPS-Automaten). Die SPS-Automaten bilden eine Unterklasse der HSPS-Automaten, welche wiederum eine Unterklasse der GSPS-Automaten bilden. In dieser Arbeit werden allerdings nur die GSPS-Automaten und eine neue Unterklasse der GSPS-Automaten, die MSPS-Automaten, verwendet. Die MSPS-Automaten sind ähnlich zu den HSPS-Automaten und werden in Abschnitt 3.4 eingeführt.

3.1. Generalisierter SPS-Automat

In diesem Abschnitt werden generalisierte SPS-Automaten (GSPS-Automaten) eingeführt, wie sie im UniForM-Projekt [13] definiert wurden. Diese Definition ermöglicht es, das Verhalten realer speicherprogrammierbarer Steuerungen einfach durch GSPS-Automaten modellieren zu können.

Die Definition entspricht der von Dierks [7] mit dem Unterschied, dass GSPS-Automaten hier deterministisch definiert werden.

Definition 3.1 (GSPS-Automat)**.** Ein *generalisierter SPS-Automat*, kurz *GSPS-Automat*[1], ist ein Tupel $\mathcal{G} = (Q, \Sigma, L, T, \Omega, \delta, g_0, \varepsilon, \Xi, \Theta)$ hierbei ist:

- Q eine nicht leere, endliche Menge von *Zuständen*,
- Σ eine endliche Menge von *Eingabevariablen*,
- L eine endliche Menge von *lokalen Variablen*,
- T eine endliche Menge von booleschen *Timervariablen*,
- Ω eine endliche Menge von *Ausgabevariablen*,
- δ eine *Übergangsfunktion* vom Typ $Q \times \mathcal{V}(L \cup \Omega \cup \Sigma \cup T) \longrightarrow Q \times \mathcal{V}(L \cup \Omega)$,
- $g_0 \in Q \times \mathcal{V}(L \cup \Omega)$ die *Startbelegung*,
- $\varepsilon > 0$ die *obere Zeitschranke* für einen Zyklus,

[1] engl. GPLC-Automaton

- Ξ eine Funktion vom Typ $T \longrightarrow 2^Q$ die jeder Timervariable eine Menge von Zuständen zuordnet, in denen der Timer aktiviert ist (*Aktivierungsregion*) und

- Θ eine Funktion vom Typ $T \longrightarrow \mathbb{R}_{>0}$ die jeder Timervariable ein *Laufzeit* zuweist.

Außerdem sind Σ, L, T und Ω disjunkte Mengen und die folgende Bedingung muss gelten:

$$\forall q \in Q, v \in \mathcal{V}(L \cup \Omega), \varphi \in \mathcal{V}(\Sigma), \tau_1, \tau_2 \in \mathcal{V}(T) : \\ \tau_1|_{\{t \in T | q \in \Xi(t)\}} = \tau_2|_{\{t \in T | q \in \Xi(t)\}} \implies \delta(q, v, \varphi, \tau_1) = \delta(q, v, \varphi, \tau_2)$$

Diese Bedingung sagt aus, das Timer, die nicht aktiviert sind das Verhalten des Systems nicht beeinflussen können. ■

Man schreibt $(q, v) \xrightarrow{\varphi,\tau}_{\mathcal{G}} (q', v')$ mit $q, q' \in Q$, $v, v' \in \mathcal{V}(L \cup \Omega)$, $\varphi \in \mathcal{V}(\Sigma)$ und $\tau \in \mathcal{V}(T)$ falls $(q', v') = \delta_{\mathcal{G}}(q, v, \varphi, \tau)$ gilt.

Die booleschen Timervariablen geben an, ob die Verzögerungszeit für eine Aktivierungsregion noch nicht erreicht ist oder schon abgelaufen ist. Wenn also $t \in T$ eine Timervariable mit der Aktivierungsregion $\Xi(t) \subseteq Q$ ist, dann hat sie den Wert `true`, wenn das System weniger als $\Theta(t)$ Zeiteinheiten in der Aktivierungsregion verblieben ist, und `false`, wenn das System seit mindestens $\Theta(t)$ Zeiteinheiten in der Aktivierungregion verblieben ist.

Das operationale Verhalten eines SPS-Automaten, wie er oben definiert wurde, ist das Folgende: In jedem Zyklus speichert das System die eingelesenen Eingabewerte in den Variablen aus Σ. Dann nimmt das System die Transition, die für den aktuellen Zustand und die aktuelle Belegung der Eingabe-, Timer-, Ausgabevariablen und lokalen Variablen gültig ist. Im Anschluss daran, werden die Werte der Ausgabevariablen nach außen sichtbar gemacht und der Zyklus ist beendet.

Das ε stellt eine Bedingung für die maximale Zykluszeit der SPS dar, auf der der SPS-Automat implementiert werden soll. Diese darf höchstens ε betragen, damit die gewünschte Funktionalität erreicht wird.

3.2. Realzeitautomatensemantik von SPS-Automaten

Mit Moby/RT kann aus einem SPS-Automaten ausführbarer Quellcode generiert werden [8]. Die folgende Semantik beschreibt das Verhalten der SPS, die diesen Quellcode innerhalb einer gegebenen maximalen Zykluszeit ausführt. Zuvor werden jedoch einige Hilfsausdrücke definiert.

Definition 3.2. Für eine Menge V von Variablen, eine Menge C von Uhren und eine Belegung $v \in \mathcal{V}(V)$ sei der Uhrenausdruck $\beta_v \in \mathcal{U}(V, C)$ definiert als

$$\beta_v := \bigwedge_{x \in V} x = v(x)$$

Weiter sei das Assignment $\alpha_v \in \mathcal{A}(V, C)$ definiert als

$$\alpha_v := \prod_{x \in V} x = v(x)$$

dabei stehe $\prod_{i=1}^{n} a_i$ für das Assignment $a_1, \ldots, a_n$, also die durch Komma separierte Liste der Assignments a_i. ■

Man beachte, dass hier „=“ in einem Uhrenausdruck als Vergleichsoperator und in einem Assignment als Zuweisungsoperator zu interpretieren ist.

Der Uhrenausdruck β_v fordert, dass alle Variablen x aus V einen Wert gemäß der Belegung v besitzen. Es gilt also $\beta_v(w) = \top \iff w = v$ für alle Belegungen $w \in \mathcal{V}(W)$. Das Assignment α_v weist allen Variablen x aus V einen Wert entsprechend der Belegung v zu. Es ist also $\alpha_v(w) = v$ für jede Belegung $w \in \mathcal{V}(V)$.

Nun folgt die formale Definition der Semantik eines GSPS-Automaten und im Anschluss daran eine Erläuterung derselben.

Definition 3.3 (Realzeitautomatensemantik eines GSPS-Automaten)**.** Sei

$$\mathcal{G} = (Q, \Sigma, L_{\mathcal{G}}, T, \Omega, \delta, g_0, \varepsilon, \Xi, \Theta)$$

ein GSPS-Automat. Der Realzeitautomat

$$\mathcal{T}(\mathcal{G}) = (L, V, C, \emptyset, E, I, \emptyset, l_0, v_0)$$

ist definiert durch:

- $L := \{0, 1, 2\} \times Q$, die Menge der Knoten,
- $V := \Pi \cup \Sigma \cup L_{\mathcal{G}} \cup \Omega \cup T$ wobei $\Pi := \{\pi_\sigma \mid \sigma \in \Sigma\}$ eine disjunkte Kopie von Σ sei
- $C := \{x, z\} \cup \{y_t \mid t \in T\}$, die Menge der Uhren,
- $S := \{\tau\}$, die Menge der Synchronisationsbeschriftungen,
- $I(l) := z \leq \varepsilon$, die Invariante für $l \in L$,
- $l_0 := (0, q)$ und $v_0 := v$ für ein beliebiges $v \in \mathcal{V}(V)$ mit $(q, v|_{L_{\mathcal{G}} \cup \Omega}) = g_0$ und $v|_C = 0$.

- E, die Menge der Kanten, ist wie folgt definiert: für $i \in \{0,1,2\}$, $q \in Q$ und $p \in \mathcal{V}(\Pi)$

$$(i,q) \xrightarrow{\tau,\top,(\alpha_p,x=0)} (i,q) \tag{1}$$

und für $s' \in \mathcal{V}(\Sigma)$ mit $s'(\sigma) = p(\pi_\sigma)$ für alle $\sigma \in \Sigma$

$$(0,q) \xrightarrow{\tau,\,\beta_p \wedge 0<x \wedge 0<z,\,\alpha_{s'}} (1,q) \tag{2}$$

und für $\vartheta \in \mathcal{V}(T)$

$$(1,q) \xrightarrow{\tau,\gamma_\vartheta^q,A_\vartheta} (2,q) \tag{3}$$

und für $q' \in Q$, $s \in \mathcal{V}(\Sigma)$ und $u, u' \in \mathcal{V}(L \cup \Omega)$ mit $(q,u) \xrightarrow{s,\vartheta}_{\mathcal{G}} (q',u')$

$$(2,q) \xrightarrow{\tau,\,\beta_u \wedge \beta_s \wedge \beta_\vartheta,\,(\alpha_{u'},R_q^{q'})} (0,q') \tag{4}$$

wobei der Uhrenausdruck $\gamma_\vartheta^q \in \mathcal{U}(V,C)$ definiert ist als

$$\gamma_\vartheta^q := \bigwedge_{\substack{q \in \Xi(t) \\ \vartheta(t)=\top}} y_t < \Theta(t) \wedge \bigwedge_{\substack{q \in \Xi(t) \\ \vartheta(t)=\bot}} y_t \geq \Theta(t)$$

und die Assignments $A_\vartheta, R_q^{q'} \in \mathcal{A}(V,C)$ als

$$A_\vartheta := \prod_{q \in \Xi(t)} t = \vartheta(t)$$

$$R_q^{q'} := z = 0,\ \prod_{\substack{q' \in \Xi(t) \\ q \notin \Xi(t)}} y_t = 0$$

■

Jeder Knoten des Realzeitautomaten $\mathcal{T}(\mathcal{G})$, der aus einem gegebenen SPS-Automaten $\mathcal{G}$ hervorgeht, ist ein Tupel (i,q), hierbei ist:

- $i \in \{0,1,2\}$ eine Variable, die den aktuellen Status der SPS beschreibt („Programmzähler“),
- $q \in Q$ der aktuelle Zustand des GSPS-Automaten,

Die Variablenmenge V von $\mathcal{T}(\mathcal{G})$ enthält die folgenden Mengen von Variablen:

- Π für die anliegende Eingabe (externe Eingabevariablen),
- Σ für die zuletzt eingelesene Eingabe (interne Eingabevariablen),
- L für die lokalen Variablen,
- Ω für die Ausgabevariablen und
- T für die Timervariablen.

Desweiteren enthält die Menge der Uhren C drei verschiedene Arten von Uhren:

- x misst die Zeit wie lange die aktuell anliegende Eingabe schon stabil ist.
- y_t misst die Zeit die vergangen ist, seit der Timer t das letzte mal gestartet (d. h. auf 0 zurückgesetzt) wurde
- z misst die Zeit die im aktuellen Zyklus der SPS vergangen ist.

Eine Transition der Form $l \xrightarrow{s,\, g,\, a} l'$ ist dabei so zu verstehen wie in der Definition des Realzeitautomaten. s ist eine Synchronisationsbeschriftung, g ist ein Uhrenausdruck (ein Guard) und a ist ein Assignment.

Dabei modelliert der Programmzähler i den internen Status der SPS. Wenn der Programmzähler den Wert 0 hat, dann hat das Einlesen der anliegenden Eingabe im aktuellen Zyklus noch nicht stattgefunden. Der Übergang von 0 zu 1 repräsentiert das Einlesen der Eingabe (2), deshalb wird hier über das Assignment α'_s den internen Eingabevariablen der Wert der externen Eingabevariablen zugewiesen. Die x-Uhr wird jedesmal auf 0 zurückgesetzt, wenn der Wert, der an der Eingabeschnittstelle anliegt, von außen geändert wird (1). Da nur Werte eingelesen werden sollen, die länger als nur zu einem Zeitpunkt anliegen, muss überprüft werden ob die x-Uhr einen Wert größer als 0 besitzt. Dies ist nötig, weil es sonst vorkommen kann, dass zwei speicherprogrammierbare Steuerungen, welche die gleiche Eingabe einlesen (siehe parallele Komposition, Abschnitt 3.3.1), zum selben Zeitpunkt unterschiedliche Werte einlesen. Es kann dann nämlich sein, dass im selben Zeitpunkt erst die SPS_1 die Eingabewerte einliest, dann ändern sich die Eingabewerte und dann liest die SPS_2 die neuen Eingabewerte ein, immer noch im selben Zeitpunkt. Wenn man nun fordert, dass die x-Uhr größer 0 sein muss, dann darf im obigen Fall die SPS_2 nicht im selben Zeitpunkt die Werte einlesen wie SPS_1, weil sich die Eingabewerte geändert haben. Zwei speicherprogrammierbare Steuerungen dürfen also nur dann im selben Zeitpunkt (gleichzeitig) die Eingabewerte einlesen, wenn diese sich in diesem Zeitpunkt nicht ändern.

Hat der Programmzähler den Wert 1, dann hat das Einlesen der Eingabe bereits stattgefunden und die Berechnungen werden ausgeführt. Weil das Ergebnis der Berechnungen erst am Ende des Zyklus nach außen sichtbar wird, (vergl. Abb. 2.2), reicht es hier nur die aktuellen Werte der Timervariablen zu speichern (3), da mit

ihnen und der eingelesenen Eingabe das Ergebnis der Berechnung fest steht. Für die Berechnungen ist es wichtig, welche Belegung die Timervariablen haben. Die Bedingung γ_ϑ^q ist für einen Zustand q und eine Timer-Belegung ϑ erfüllt, wenn für alle Timer t die in q aktiv sind ($q \in \Xi(t)$) folgendes gilt:

- Entweder hat der Timer den Wert `true` *und* die entsprechende Uhr y_t hat noch nicht die Grenze $\Theta(t)$ erreicht oder
- der Timer hat den Wert `false` und die entsprechende Uhr y_t hat die Grenze $\Theta(t)$ erreicht.

Wenn der Programmzähler den Wert 2 hat, dann sind die Berechnungen beendet und die Aktualisierungsphase beginnt. Jetzt werden die Ergebnisse der Berechnung nach außen sichtbar. Aufgrund der gespeicherten Werte der eingelesenen Eingabe und den Werten der Timer, kann die beobachtbare Ausgabe abgeleitet werden. Wenn diese Phase beendet ist und der Zähler wieder auf 0 gesetzt wird, endet der Zyklus. Hierbei wird der Zustand und die lokalen Variablen als auch die Ausgabevariablen, entsprechend den Berechnungen, geändert (4). In diesem Schritt muss die z-Uhr, welche die Zyklusdauer misst, auf 0 zurückgesetzt werden und ebenso müssen alle y_t Uhren der Timer t auf 0 zurückgesetzt werden, die durch die Transition von q nach q' aktiviert wurden. Dies wird durch das Assignment $R_q^{q'}$ realisiert.

Die Semantik von SPS-Automaten lässt sich auch über den Duration Calculus [6, 11] definieren. Eine Definition findet man in Dierks Dissertationsschrift [8]. Der Duration Calculus ist eine Logik und ein Kalkül für Realzeitsysteme. Der Vorteil dieser Semantik ist, dass hier logisches Schließen einfacher ist als mit Realzeitautomaten.

Aus Gründen der besseren Lesbarkeit wird hier darauf verzichtet für jede mögliche Variablenbelegung eine Transition zu zeichnen. Wenn also an einem Zustand für eine Variablenbelegung keine Tansition dargestellt ist, dann hat dieser Zustand eine implizite Loop-Transition die keine Veränderung der lokalen Variablen und der Ausgabevariablen bewirkt.

Beispiel 3.4 (Lampe Teil 2)**.** Abbildungen 3.1 zeigt eine Modellierung des in Beispiel 2.6 beschriebenen Lichtschalter mittels eines SPS-Automaten. Hier muss zwischen dem Gedrückthalten des Schalters und dem Loslassen des Schalters unterschieden werden, da sonst ein langes Drücken wie zwei oder mehrmals drücken interpretiert werden würde. Zu Beginn befindet sich der Automat im Zustand `aus1` und die Lampe ist aus. Wird dann der Schalter gedrückt (`d`), dann wechselt der Automat über Transition (1) in den Zustand `an1` und die Lampe geht an (`Status := an`). Wird nun der Schalter wieder losgelassen (`not d`), so wechselt der Automat über Transition (2) in den Zustand `an2`. Wird jetzt innerhalb von 2 Zeiteinheiten noch einmal auf den Schalter gedrückt, so wechselt der Automat über

Transition (6) in den Zustand `an_heller` und die Lampe leuchtet heller (`Status := heller`). Wenn nicht, dann wechselt der Automat über die Transition (3) in den Zustand `an3` was er auch macht, wenn er sich im Zustand `an_heller` befindet und der Schalter wieder losgelassen wird, nur diesmal über Transition (7). Wird dann auf den Schalter gedrückt, so wechselt der Automat über Transition (4) in den Zustand `aus2` und die Lampe wird ausgeschaltet (`Status := aus`). Sobald der Schalter wieder losgelassen wird wechselt der Automat über Transition (5) wieder in den Startzustand `aus1`.

Der hier beschriebene SPS-Automat kann folgendermaßen durch einen GSPS-Automaten $\mathcal{G}_L$ beschrieben werden:

$$\mathcal{G}_L = (Q, \Sigma, L, T, \Omega, \delta, g_0, \varepsilon, \Xi, \Theta)$$

hierbei ist

$$\begin{aligned}
Q &:= \{\texttt{aus1},\ \texttt{aus2},\ \texttt{an1},\ \texttt{an2},\ \texttt{an2},\ \texttt{an_heller}\} \\
\Sigma &:= \{\texttt{d}\} \\
L &:= \emptyset \\
T &:= \{t_{\texttt{an2}}\} \\
\Omega &:= \{\texttt{Status}\} \\
g_0 &:= (\texttt{aus1}, [\texttt{Status} \mapsto \texttt{aus}]) \\
\varepsilon &:= 1 \\
\Xi(t_{\texttt{an2}}) &:= \{\texttt{an2}\} \\
\Theta(t_{\texttt{an2}}) &:= 2
\end{aligned}$$

$$\begin{aligned}
\delta(\texttt{aus1}, [\texttt{Status} \mapsto \texttt{aus}][\texttt{d} \mapsto \texttt{true}][\texttt{t}_{\texttt{an2}} \mapsto *]) &:= (\texttt{an1}, [\texttt{Status} \mapsto \texttt{an}]) \\
\delta(\texttt{aus}, [\texttt{Status} \mapsto \texttt{aus}][\texttt{d} \mapsto \texttt{false}][\texttt{t}_{\texttt{an2}} \mapsto *]) &:= (\texttt{aus}, [\texttt{Status} \mapsto \texttt{aus}]) \\
\delta(\texttt{an1}, [\texttt{Status} \mapsto \texttt{an}][\texttt{d} \mapsto \texttt{true}][\texttt{t}_{\texttt{an2}} \mapsto *]) &:= (\texttt{an1}, [\texttt{Status} \mapsto \texttt{an}]) \\
\delta(\texttt{an1}, [\texttt{Status} \mapsto \texttt{an}][\texttt{d} \mapsto \texttt{false}][\texttt{t}_{\texttt{an2}} \mapsto *]) &:= (\texttt{an2}, [\texttt{Status} \mapsto \texttt{an}]) \\
\delta(\texttt{an2}, [\texttt{Status} \mapsto \texttt{an}][\texttt{d} \mapsto \texttt{true}][\texttt{t}_{\texttt{an2}} \mapsto *]) &:= (\texttt{an3}, [\texttt{Status} \mapsto \texttt{an}]) \\
\delta(\texttt{an2}, [\texttt{Status} \mapsto \texttt{an}][\texttt{d} \mapsto \texttt{false}][\texttt{t}_{\texttt{an2}} \mapsto \texttt{true}]) &:= (\texttt{an2}, [\texttt{Status} \mapsto \texttt{an}]) \\
\delta(\texttt{an2}, \{\texttt{Status} \mapsto \texttt{an}][\texttt{d} \mapsto \texttt{false}][\texttt{t}_{\texttt{an2}} \mapsto \texttt{false}]) &:= (\texttt{an_heller}, [\texttt{Status} \mapsto \texttt{heller}]) \\
\delta(\texttt{an_heller}, [\texttt{Status} \mapsto \texttt{heller}][\texttt{d} \mapsto \texttt{true}][\texttt{t}_{\texttt{an2}} \mapsto *]) &:= (\texttt{an_heller}, [\texttt{Status} \mapsto \texttt{heller}]) \\
\delta(\texttt{an_heller}, [\texttt{Status} \mapsto \texttt{heller}][\texttt{d} \mapsto \texttt{false}][\texttt{t}_{\texttt{an2}} \mapsto *]) &:= (\texttt{an3}, [\texttt{Status} \mapsto \texttt{heller}]) \\
\delta(\texttt{an3}, [\texttt{Status} \mapsto \texttt{an}][\texttt{d} \mapsto \texttt{true}][\texttt{t}_{\texttt{an2}} \mapsto *]) &:= (\texttt{aus2}, [\texttt{Status} \mapsto \texttt{aus}]) \\
\delta(\texttt{an3}, [\texttt{Status} \mapsto \texttt{an}][\texttt{d} \mapsto \texttt{false}][\texttt{t}_{\texttt{an2}} \mapsto *]) &:= (\texttt{an3}, [\texttt{Status} \mapsto \texttt{an}]) \\
\delta(\texttt{an3}, [\texttt{Status} \mapsto \texttt{heller}][\texttt{d} \mapsto \texttt{true}][\texttt{t}_{\texttt{an2}} \mapsto *]) &:= (\texttt{aus2}, [\texttt{Status} \mapsto \texttt{aus}])
\end{aligned}$$

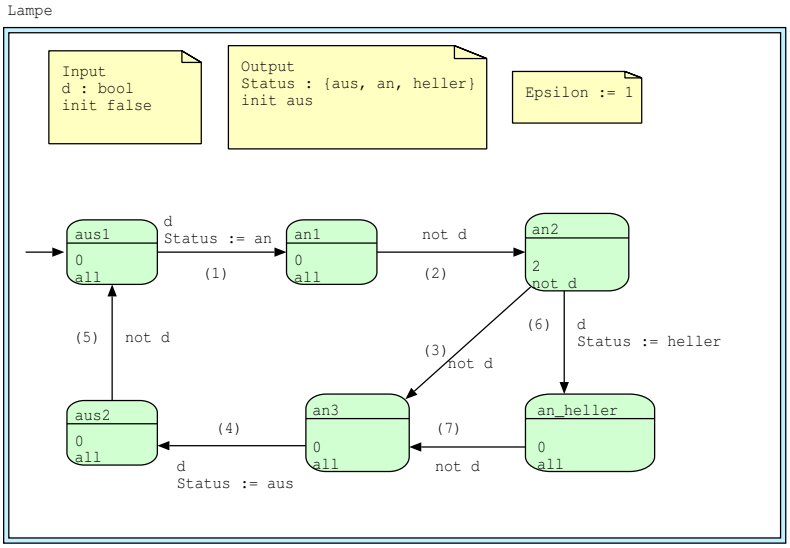

Abbildung 3.1.: Modellierung einer Lampe durch einen SPS-Automaten.

$$\delta(\texttt{an3}, [\texttt{Status} \mapsto \texttt{heller}][\texttt{d} \mapsto \texttt{false}][\texttt{t}_{\texttt{an2}} \mapsto *]) := (\texttt{an3}, [\texttt{Status} \mapsto \texttt{heller}])$$
$$\delta(\texttt{aus2}, [\texttt{Status} \mapsto \texttt{aus}][\texttt{d} \mapsto \texttt{true}][\texttt{t}_{\texttt{an2}} \mapsto *]) := (\texttt{aus2}, [\texttt{Status} \mapsto \texttt{aus}])$$
$$\delta(\texttt{aus2}, [\texttt{Status} \mapsto \texttt{aus}][\texttt{d} \mapsto \texttt{false}][\texttt{t}_{\texttt{an2}} \mapsto *]) := (\texttt{aus1}, [\texttt{Status} \mapsto \texttt{aus}])$$

Das Sternsymbol steht hier für einen beliebigen Wert aus dem Wertebereich der Variable.

3.3. Komposition von SPS-Automaten

In diesem Abschnitt werden nun zwei Möglichkeiten der Komposition von SPS-Automaten vorgestellt. Da SPS-Automaten in der Praxis sehr groß werden können, wäre das Modellieren eines einzigen Automaten sehr unübersichtlich. Daher ist eine Aufteilung in mehrere kleine Automaten, die für Teilaufgaben zuständig sind und miteinander kommunizieren können, äußerst sinnvoll und einfacher zu warten. Hierzu dienen die sequentielle und die parallele Komposition von GSPS-Automaten, die im Folgenden vorgestellt werden.

3.3.1. Parallele Komposition

Bei einer parallelen Komposition von GSPS-Automaten, wird jeder Automat auf eine eigene SPS implementiert. Eine mögliche Kommunikation findet über die

Schnittstellen der SPS statt, d. h. eine SPS kann die Ausgabe einer anderen SPS über ihre Schnittstelle einlesen.

Durch die Verwendung mehrerer speicherprogrammierbarer Steuerungen ist die Reaktionszeit des Systems so groß wie die der langsamsten Komponente, also zweimal die maximale Zykluszeit der langsamsten SPS (siehe Abschnitt 2.1).

3.3.2. Sequentielle Komposition

Bei einer sequentiellen Kompostion werden die GSPS-Automaten auf der selben SPS implementiert. Wenn hierbei ein Automat die Ausgabe eines anderen Automaten als Eingabe hat, dann muss sie nicht erst eingelesen werden (wie bei der parallelen Komposition) sondern es kann jederzeit auf sie zugegriffen werden, da dies eine gemeinsame Variable ist. Außerdem können mehrere Automaten die gleiche Eingabe einlesen. Um dies zu ermöglichen können Eingabevariablen umbenannt werden. Wenn also zum Beispiel zwei Automaten A_1 und A_2 jeweils eine Eingabevariable E_1 bzw. E_2 besitzen die beide die gleiche Eingabe einlesen sollen, dann wird entweder E_1 in E_2 umbenannt oder E_2 in E_1. Wenn die Eingabevariable E_1 des Automaten A_1 die Ausgabvariable O_2 des Automaten A_2 einliest, dann wird die Eingabevariable E_1 in O_2 umbenannt. Desweiteren können Ausgabevariablen zu lokalen Variablen transformiert werden, um die Ausgabe nicht nach außen sichtbar zu machen (engl. hiding). Dies wird hier aber nicht berücksichtigt, da bei der nachfolgenden Übersetzung nicht zwischen sichbaren und nicht sichtbaren Ausgaben unterschieden wird.

Durch eine sequentielle Komposition zweier GSPS-Automaten $\mathcal{G}_1$ und $\mathcal{G}_2$, entsteht also ein neuer GSPS-Automat $\mathcal{G}$. Das zyklische Verhalten der SPS, auf der, der GSPS-Automat $\mathcal{G}$ implementiert ist, ist das folgende:

- Einlesephase: Einlesen der Eingabevariablen beider Automaten, aber nur die, die nicht mit einer Ausgabevariable verbunden sind.

- Berechnungsphase: Feststellen der aktuellen Werte der Timervariablen („Die SPS schaut auf die Uhr“).

  - Ausführen einer Transition in $\mathcal{G}_1$ mit der Belegung der Eingabevariablen, die durch das Einlesen und der Ausgabe in $\Sigma_1 \cap \Omega_2$ des vorhergehenden Zyklus determiniert ist.

  - Ausführen einer Transition in $\mathcal{G}_2$ mit der Belegung der Eingabevariablen, die durch das Einlesen und der Ausgabe in $\Sigma_2 \cap \Omega_1$ des aktuellen Zyklus determiniert ist.

- Aktualisierungphase: Aktualiesiern der Ausgabe $\Omega_1 \cup \Omega_2$

Ein Zustandsübergag in $\mathcal{G}$ verhält sich also so, wie wenn im selben Zeitpunkt erst ein Zustandsübergang z_1 in $\mathcal{G}_1$ stattfindet und dann einer z_2 in $\mathcal{G}_2$. Die Ausführung dieser Transitionen im selben Zeitpunkt wird dadurch realisiert, indem zur Berechnung all dieser Zustandsübergänge die gleiche Timer-Belegung verwendet wird. Dies entspricht der Vorgehensweise aus der Definition 3.3, in der die Belegungen der Timervariablen (3), vor der Berechnung des Zustandübergangs (4) bestimmt wird. Wird durch den Zustandsübergang z_1 die Werte der lokalen Variablen oder die der Ausgabevariablen geändert, so werden diese neuen Wert für die Berechnung des Zustandübergangs z_2 (im selben Zyklus) verwendet.

Wie in Abschnitt 3.1 beschrieben, ist ε eine obere Schranke für die maximale Dauer eines Zyklus. Der zugehörige SPS-Automat darf also nur auf einer SPS mit einer maximalen Zykluszeit von ε implementiert werden, damit die gewünschte Funktionalität auch erreicht wird. Wenn also zwei SPS-Automaten A_1 und A_2 mit den jeweiligen maximalen Zykluszeiten ε_1 und ε_2 auf der gleichen SPS implementiert werden sollen, dann darf die maximale Zykluszeit dieser SPS also höchstens so groß sein wie das Minimum von ε_1 und ε_2.

Die maximale Reaktionszeit ist dann $2 \cdot \min\{\varepsilon_1, \varepsilon_2\}$, wie schon in Abschnitt 2.1 beschrieben.

Definition 3.5 (Sequentielle Komposition von GSPS-Automaten)**.** Sei

$$\mathcal{G}_i = (Q_i, \Sigma_i, L_i, T_i, \Omega_i, \delta_i, g_{0,i}, \varepsilon_i, \Xi_i, \Theta_i)$$

ein GSPS-Automat für $i = 1, 2$ und disjunkten Mengen $L_1, L_2, T_1, T_2, \Omega_1, \Omega_2$ und $\Sigma_i \cap (L_{3-i} \cup T_{3-i}) = \emptyset$ für $i = 1, 2$. Der GSPS-Automat

$$\mathcal{G}_1; \mathcal{G}_2 = (Q, \Sigma, L, T, \Omega, \delta, g_0, \varepsilon, \Xi, \Theta)$$

ist genau dann eine *sequentielle Komposition von* $\mathcal{G}_1$ *und* $\mathcal{G}_2$ wenn

$$\begin{aligned}
Q &= Q_1 \times Q_2 \\
\Sigma &= (\Sigma_1 \cup \Sigma_2) \setminus (\Omega_1 \cup \Omega_2) \\
L &= L_1 \cup L_2 \\
T &= T_1 \cup T_2 \\
\Omega &= \Omega_1 \cup \Omega_2 \\
\delta &= \delta_1; \delta_2 \\
g_0 &= ((q_1, q_2), v) \quad \text{mit } (q_i, v|_{L_i \cup \Omega_i}) = g_{0,i} \text{ für } i = 1, 2 \\
\varepsilon &= \min\{\varepsilon_1, \varepsilon_2\} \\
\Xi(t) &= \begin{cases} \Xi_1(t) \times Q_2 & \text{falls } t \in T_1 \\ Q_1 \times \Xi_2(t) & \text{falls } t \in T_2 \end{cases} \\
\Theta(t) &= \begin{cases} \Theta_1(t) & \text{falls } t \in T_1 \\ \Theta_2(t) & \text{falls } t \in T_2 \end{cases}
\end{aligned}$$

wobei die sequentielle Komposition zweier Transitionen in Definition 3.6 beschrieben ist. ■

Definition 3.6 (Sequentielle Komposition von Transitionen)**.** Seien L_i, T_i, Ω_i für $i = 1, 2$ disjunkte Variablenmengen, $\Sigma_i \cup (L_{3-i} \cup T_{3-i}) = \emptyset$, und Q_1, Q_2 seien Mengen von Zuständen. Wenn δ_i Funktionen vom Typ $Q_i \times \mathcal{V}(L_i \cup \Omega_i \cup \Sigma_i \cup \mathcal{T}_i) \to Q_i \times \mathcal{V}(L_i \cup \Omega_i)$ sind, dann ist die *sequentielle Komposition von* δ_1 *und* δ_2 definiert durch

$$\delta_1;\delta_2 \colon (Q_1 \times Q_2) \times \mathcal{V}(L_1 \cup L_2 \cup \Omega_1 \cup \Omega_2 \cup \Sigma_1 \cup \Sigma_2 \cup T_1 \cup T_2) \longrightarrow (Q_1 \times Q_2) \times \mathcal{V}(L_1 \cup L_2 \cup \Omega_1 \cup \Omega_2)$$

hierbei ist $(\delta_1;\delta_2)((q_1,q_2),v) := ((q_1',q_2'),v')$ mit $\delta_1(q_1, v|_{L_1 \cup \Omega_1 \cup \Sigma_1 \cup T_1}) = (q_1', v'')$, $\delta_2(q_2, (v \leftarrow v'')|_{L_2 \cup \Omega_2 \cup \Sigma_2 \cup T_2}) = (q_2', v''')$ und $v' = (v \leftarrow v'') \leftarrow v'''$. ■

Eine Realzeitautomatensemantik von sequentiellen Netzwerken, in der die Transitionen wirklich nacheinander ausgeführt werden und nicht zu einer einzigen Transition zusammengesetzt werden [8], ist für die folgende Übersetzung zwar passender, aber da sie sehr intuitiv ist, wird sie hier nicht formal definiert.

Beispiel 3.7 (Lampe Teil 3)**.** In Abbildung 3.2 ist eine weitere Modellierung des in Beispiel 2.6 beschrieben Lichtschalters zu sehen. Hier allerdings wird er durch ein sequentielles Netzwerk zweier SPS-Automaten modelliert. Der obere SPS-Automat modelliert den Benutzer so, dass die Eingabe welche einen Status darstellt, in eine Ausgabe umgewandelt wird, die ein Ereignis darstellt (Ereignisproduktion). Dadurch wird für die eigentliche Steuerung des Schalters (unterer Automat), wenigere Zustände benötigt. Der untere Automat reagiert also auf die steigende Flanke des Signals `gedrueckt` und nicht wie in Abbildung 3.2 auf die steigende und die fallende Flanke. Der Pfeil zwischen den zwei Automaten veranschaulicht, dass der untere Automat die Ausgabe des oberen als Eingabe verwendet. Hier ist es wichtig, dass beide Automaten auf der selben SPS implementiert werden, die Automaten also durch die sequentielle Komposition `Benutzer;Lampe` verbunden werden. Denn es ist wichtig, dass beide Automaten gleichzeitig starten und immer erst eine Transition des Automaten `Benutzer` und dann eine Transition des Automaten `Lampe` ausgeführt wird. Ist dies nicht der Fall, so kann es zum Beispiel passieren, dass der Automat `Benutzer` seinen Zyklus schneller beendet als der Automat `Lampe` und so ein langes drücken des Schalters als ein zweimaliges Drücken des Schalters interpretiert wird.

3.4. MSPS-Automaten

Die in MOBY/RT darstellbaren Automaten entsprechen den hier nun vorgestellten MSPS-Automaten. So wie die HSPS-Automaten in der Dissertationsschrift

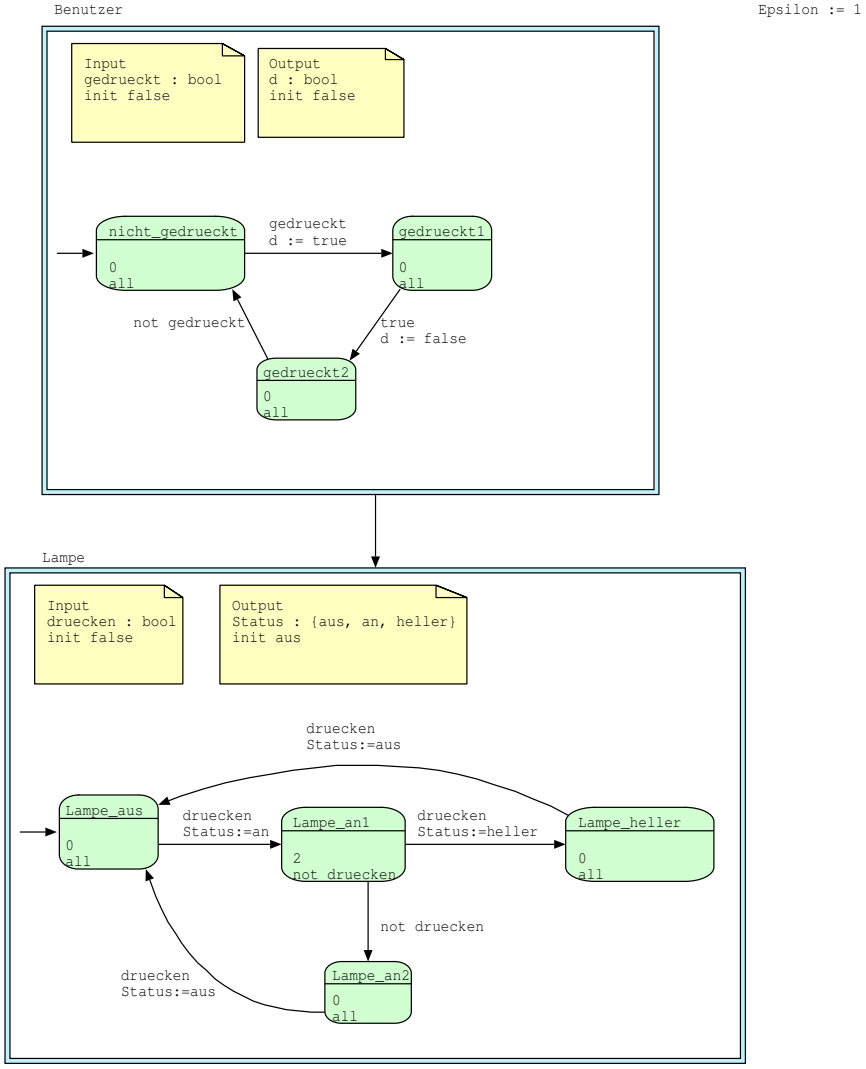

Abbildung 3.2.: Modellierung einer Lampe und eines Benutzers mittels sequentiellem Netzwerk zweier SPS-Automaten.

von Dierks [8], besitzen sie ein Hierarchiekonzept, d. h. hinter einem Zustand kann sich wiederum ein ganzer SPS-Automat verbergen. Dadurch lassen sich auch komplexere Automaten noch übersichtlich darstellen.

Im Folgenden werden nun erstmal die Guard- und Aktionsausdrücke und der Hierarchiebaum eines MSPS-Automaten definiert bevor die eigentliche Definition eines MSPS-Automaten folgt.

Definition 3.8 (Guardausdruck und Aktionsausdruck)**.** Sei X eine Menge von Variablen. Ein Ausdruck der Form *GuardExpr* gemäß der Grammatik in Abbildung 4.3, wobei *ID* für Bezeichner von Variablen aus X stehe, ist ein *Guardausdruck über* X. Weiter ist $G(X)$ die Menge der Guardausdrücke über X. Entsprechend sei $E(X)$ die Menge der Ausdrücke von der Form *Expression*.

Seien $L, R \subseteq X$. Ein Ausdruck der Form *ActionExpr*, wobei Variablen aus R nicht auf der linken Seite einer Zuweisung „ID `:=` *Expression*“ stehen dürfen, ist ein *Aktionsausdruck über* (L, R). Weiter ist $A(L, R)$ die Menge der Aktionsausdrücke über (L, R) zusammen mit dem leeren Wort λ.

Für eine Belegung $v \in \mathcal{V}(X)$ und einen Ausdruck $e \in E(X)$ sei $e(v)$ die Auswertung des Ausdrucks e bezüglich der Belegung v; dabei sei die Auswertung von Ausdrücken der Form *BoolExpr* und *IntExpr* wie für boolsche Ausdrücke bzw. Integerausdrücke üblich definiert und es stehe $\top$ für wahr und $\bot$ für falsch.

Für Aktionsausdrücke $a \in A(L, R)$ sei die Auswertung $a(v) \in \mathcal{V}(L \cup R)$ wie folgt rekursiv definiert, für $\ell \in L$, $e \in E(L \cup R)$ typkonform mit ℓ, $g \in G(L \cup R)$ und $a_1, a_2 \in A(L, R)$ sei

- $(\ell \texttt{ := } e)(v) := v[\ell \mapsto e(v)]$
- $(\mathit{If}\, g\ \mathit{Then}\, a_1\ \mathit{Endif})(v) := \begin{cases} a_1(v) & g(v) = \top \\ v & g(v) = \bot \end{cases}$
- $(\mathit{If}\, g\ \mathit{Then}\, a_1\ \mathit{Else}\, a_2\ \mathit{Endif})(v) := \begin{cases} a_1(v) & g(v) = \top \\ a_2(v) & g(v) = \bot \end{cases}$
- $(a_1 \texttt{ ; } a_2)(v) := a_2(a_1(v))$
- $\lambda(v) := v$

■

Die hierarchischen Anordnung der Zustände und Superzustände eines MSPS-Automaten wird durch einen Hierarchiebaum ausgedrückt, der wie folgt definiert wird.

Definition 3.9 (Hierarchiebaum)**.** Ein *Hierarchiebaum* ist ein gerichteter Baum $T = (M, H)$ mit der Knotenmenge M und der Nachfolgerrelation H. Die *Wurzel*

von T wird mit r bezeichnet. Mit $\prec$ wird die transitive Hülle von H bezeichnet und entsprechend mit $\preceq$ die transitive reflexive Hülle. Für $s, t \in T$ sei $w_s^t \in T$ die größte gemeinsame Wurzel von s und t, also $w_s^t \preceq s \wedge w_s^t \preceq t$ und $u \not\preceq s \vee u \not\preceq t$ für alle $u \succ w_s^t$. Für $s, t \in T$ sei $[s,t] := \{u \in T \mid s \preceq u \preceq t\}$ und entsprechend $(s,t]$, (s,t) und $[s,t)$. Weiter ist $h(t) := |(r,t]| - 1$ die Ebene[2] im Hierarchiebaum in der der Knoten t liegt. Für einen Knoten $t \in T$ und $n \leq |(r,t]|$ wird mit $t - n$ der n-te Vorgänger von t bezeichnet. ■

Zur Vereinfachung gegenüber den Automaten in Moby/RT werden bei der Definition von MSPS-Automaten die Systemports weggelassen und das Ziel der Transitionen wird aufgelöst, so dass dies immer Blätter im Hierarchiebaum sind. Eine Transition, die aus mehreren Kanten zusammengesetzt ist (siehe Kapitel 4), wird so weit verfolgt, bis das eigentlich Ziel, ein Blatt im Hierarchiebaum, erreicht ist. Dieses Blatt ist dann im MSPS-Automat das Ziel der Transition. Desweiteren werden auch die Startzustände aufgelöst, d. h. ist ein Superzustand ein Startzustand, dann ist im MSPS-Automat der Startzustand des Superzustandes ein Startzustand. Falls dieser Startzustand wieder ein Superzustand ist, dann wird nicht dieser sondern sein Startzustand als Startzustand des MSPS-Automaten interpretiert, usw. bis auch hier ein Blatt im Hierarchiebaum erreicht ist.

Die MSPS-Automaten bilden eine Unterklasse der GSPS-Automaten, wie es in der Definition 3.11 der Semantik eines MSPS-Automaten durch einen GSPS-Automaten deutlich wird.

Definition 3.10 (MSPS-Automat)**.** Ein *Moby-SPS-Automat*, kurz *MSPS-Automat*, ist ein Tupel

$$\mathcal{M} = (Q, S, \Sigma, L, \Omega, \delta, g_0, \varepsilon, T, S_t, S_e)$$

hierbei ist

- Q ist eine nicht leere endliche Menge von *Zuständen*,
- S ist eine endliche Menge von *Superzuständen* mit $Q \cap S = \emptyset$,
- Σ ist eine endliche Menge von *Eingabevariablen*,
- L ist eine endliche Menge von *lokalen Variablen*,
- Ω ist eine endlich Menge von *Ausgabevariablen*,
- $\delta \subseteq (Q \cup S) \times G(\Sigma \cup L \cup \Omega) \times A(L \cup \Omega, \Sigma) \times Q$ ist eine deterministische[3] *Übergangsrelation*,

[2] Aus praktischen Gründen wird die Ebene der Wurzel r von T auf $h(r) = -1$ festgelegt.

[3] D. h. für alle $q \in Q$, alle $v \in \mathcal{V}(\Sigma \cup L \cup \Omega)$ und alle $s \preceq q$ gibt es höchstens eine Transition $(s, g, a, q') \in \delta$ mit $g(v) = \top$ oder es gibt ein $t \prec s$ und eine Transition $(t, g, a, q') \in \delta$ mit $g(t) = \top$.

- $g_0 \in Q \times \mathcal{V}(L \cup \Omega)$ ist die *Startbelegung*,
- $\varepsilon > 0$ ist die *obere Zeitschranke* für einen Zyklus,
- $T = (\{r\} \cup Q \cup S, H)$ ist ein Hierarchiebaum, wobei die Wurzel r nicht in $Q \cup S$ enthalten ist und die Elemente aus Q die Blätter des Hierarchiebaumes T sind,
- S_t ist eine Funktion vom Typ $Q \cup S \longrightarrow \mathbb{R}_{\geq 0}$ die jedem Zustand $z \in Q \cup S$ eine *Verzögerungszeit* (delay time) zuweist, die angibt, wie lange die Eingabe, die den Ausdruck $S_e(z)$ erfüllt, ignoriert werden soll,
- S_e ist eine Funktion vom Typ $Q \cup S \longrightarrow G(\Sigma)$ die jedem Zustand z einen Ausdruck, *Verzögerungsbedingung* genannt, zuweist, der keinen Zustandsübergang verursacht für die ersten $S_d(z)$ Zeiteinheiten, die der Automat in diesem Zustand verbleibt.

Die Guardausdrücke in $G(\Sigma \cup L \cup \Omega)$ werden schlicht als Guards bezeichnet und die Aktionsausdrücke in $A(L \cup \Omega, \Sigma)$ als Aktionen. ■

3.4.1. GSPS-Automaten Semantik eines MSPS-Automaten

Die Semantik eines MSPS-Automaten wird über eine GSPS-Automaten definiert. Da die GSPS-Automaten keine hierarchische Struktur besitzen, müssen die MSPS-Automaten „flach“ gemacht werden. Dafür werden die Superzustände aufgelöst, indem ihre Transitionen bis zu den Blättern „runtergezogen“ werden.

Definition 3.11 (Semantik eines MSPS-Automaten)**.** Für einen MSPS-Automaten $\mathcal{M} = (Q, S, \Sigma, L, \Omega, \delta, g_0, \varepsilon, T, S_t, S_e)$ wird sein GSPS-Automat definiert durch

$$\mathcal{G}(\mathcal{M}) := (Q, \Sigma, L, T_{\mathcal{M}}, \Omega, \delta_{\mathcal{M}}, g_0, \varepsilon, \Xi_{\mathcal{M}}, \Theta_{\mathcal{M}})$$

wobei

- $T_{\mathcal{M}} := \{t_s \mid s \in Q \cup S \wedge S_t(s) > 0\}$
- Für $(q, v) \in Q \times \mathcal{V}(\Sigma \cup L \cup T \cup \Omega)$ ist

$$\delta_{\mathcal{M}}(q, v) := \begin{cases} (q', a(v)|_{L \cup \Omega}) & \text{falls } (s, g, a, q') \in D_{\mathcal{M}}(q, v) \text{ existiert} \\ (q, v|_{L \cup \Omega}) & \text{falls } D_{\mathcal{M}}(q, v) = \emptyset \end{cases}$$

  dabei sei $D_{\mathcal{M}}(q, v) \subseteq \delta$ die Menge aller $(s, g, a, q') \in \delta$ mit

  (i) $s \preceq q$

  (ii) $g(v) = \top$ und für alle $(s', g', a', q'') \in \delta$ mit $s' \prec s$ gilt $g'(v) = \bot$

(iii) für alle $s' \preceq q$ mit $s' \not\preceq q'$ gilt $(S_e(s'))(v) = \bot$ oder $v(t_{s'}) = \bot$

- $\Xi_{\mathcal{M}}(t_s) := \{q \in Q \mid s \preceq q\}$
- $\Theta_{\mathcal{M}}(t_s) := S_t(s)$

■

Die Menge $D_{\mathcal{M}}(q, v)$ repräsentiert die Menge aller Transitionen, die im Zustand q bei der Belegung v erlaubt sind. Eine Transition $(s, g, a, q') \in \delta$ ist in q für eine Belegung v erlaubt, wenn v den Guard dieser Transition erfüllt ($g(v) = \top$) und den Guard aller Transitionen von über s liegenden Superzustände nicht erfüllt (für alle $(s', g', a', q'') \in \delta$ mit $s' \prec s$ gilt $g'(v) = \bot$) (ii). Desweiteren müssen alle Verzögerungsaudrücke der über s liegenden Superzustände ($s' \preceq s$) die durch die Transition verlassen werden ($s' \not\preceq q'$), erfüllt sein (iii).

Da δ deterministisch ist, enthält $D_{\mathcal{M}}(q, v)$ also höchstens ein Element und somit ist $\delta_{\mathcal{M}}(q, v)$ wohldefiniert.

Der Fall $D_{\mathcal{M}}(q, v) = \emptyset$ in der Definition von $\delta_{\mathcal{M}}(q, v)$ stellt gerade die Loop-Transitionen dar, die in MOBY/RT nicht explizit angegeben werden müssen. Also die Transition die immer dann genommen werden kann, wenn sonst keine andere Transition des Zustandes genommen werden kann.

Im anschließenden Kapitel wird ein Beispiel für einen SPS-Automaten (*Stanze*) und seine Definition als MSPS-Automat sowie seine Semantik als GSPS-Automat gegeben.

4. Moby/RT

Moby/RT [9] ist ein Werkzeug zum grafischen Design von SPS-Automaten. Hiermit können die Automaten modelliert, simuliert und verifiziert werden. Die Verifikation erfolgt durch Transformation in einen Realzeitautomaten, der dann z. B. mit Uppaal verifiziert werden kann. Desweiteren kann die Spezifikation in strukturierten Text (ST) ausgegeben werden, einem Format, das in ausführbaren Quellcode für eine SPS transformiert werden kann.

Bei der semantischen Übersetzung von Moby/RT von einem SPS-Automaten in einen Realzeitautomaten geht viel Struktur verloren, da eine Transition im SPS-Automat in viele Transitionen im Realzeitautomat übersetzt wird. Dies liegt daran, dass Uppaal früher nur Konjunktionen im Guard zuließ. Inzwischen müssen aber nur noch die Uhrenbedingungen auf oberster Ebene durch eine Konjunktion verbunden sein, wodurch der Guard der Transition direkt übernommen werden kann und somit die Anzahl der Transitionen verringert werden kann. Dies ermöglicht also die hier im Folgenden noch erläuterte syntax-orientierte Übersetzung.

Die in Moby/RT dargestellten SPS-Automaten entsprechen denen in Abschnitt 3.4 vorgestellten MSPS-Automaten und können mit Moby/RT in einem so genannten SIM-Format ausgegeben werden. Diese Darstellung der SPS-Automaten ist der Ausgangspunkt der hier vorgestellten Transformation.

4.1. Das SIM-Format

Die Syntax des SIM Formats ist definiert durch die Grammatik für ein *System* und ist in den Abbildung 4.2 und 4.3 zu finden. NUM bezeichnet eine beliebige Integer-Zahl und NAT eine positive Integer-Zahl. ID ist ein Bezeichner der aus alphanumerischen Zeichen und „_“ zusammengesetzt ist und nicht mit einer Ziffer beginnt. Es ist noch anzumerken, dass mit dieser Grammatik auch unzulässige Ausdrücke erzeugt werden können.

Im Folgenden werden nun einzelne Elemente der Grammatik beschrieben, wobei in runden Klammern der entsprechende Ausdruck der Grammatik angegeben ist. Zuvor sei noch zu bemerken, dass in Moby/RT eine Transition, die mehrere Hierarchieebenen durchquert, in mehrere Kanten über sogenannte Ports aufgespalten wird, damit sie im Editor graphisch darstellbar ist.

- **Interface:** Jeder Automat besitzt ein Interface (*InterfaceDef*), in dem die Eingabe- und Ausgabevariablen sowie die lokalen Variablen mit den Schlüs-

selwörtern `in`, `out` bzw. `local` deklariert und optional auch initialisiert werden. Es gibt drei verschiedene Datentypen (*TypeDef*) die hier angegeben werden können.

- Boolesche Variablen werden mit dem Schlüsselwort `bool` deklariert. Z.B.

  ```
  out Fehler : bool init false
  ```

- Integer Variablen werden durch Angabe ihres Wertebereichs, in Form eines Intervalls {*Untergrenze*..*Obergrenze*} deklariert. Z.B.

  ```
  in Note : {1..6}
  ```

  Innerhalb der geschweiften Klammern können auch mehrere durch Komma getrennte Intervalle und auch einzelne Zahlen angegeben werden. Dies wird dann als ein einziges Intervall interpretiert in dem alle angegebenen Werte enthalten sind. Das Intervall $[1, 5]$ lässt sich also als `{1,5}` oder `{1..5}` als auch `{1..2,4,5}` schreiben.

- Enumeration-Variablen werden durch Aufzählung ihrer Werte deklariert: {$Wert_1$, $Wert_2$, ..., $Wert_n$}. Z.B.

  ```
  out Status : { an, aus, Fehler } init aus
  ```

Wird kein Initialwert durch das Schlüsselwort `init` angegeben, dann wird ein Default-Wert angenommen. Der Default-Wert für boolesche Variablen ist false, für Integer-Variablen ist es die Untergrenze, und für Enumeration-Variablen ist es der erste Wert, also $Wert_1$.

- **Zustände und Superzustände:** Ein Zustand kann durch einen Subautomaten verfeinert werden, man nennt ihn dann Superzustand. Der Name des Superzustands bezeichnet also einen Subautomaten. Die Zustände des Subautomaten können wiederum verfeinert werden. Die Zustände eines MSPS-Automaten bilden somit eine Hierarchie von Verfeinerungen, im Weiteren Hierarchiebaum genannt. Im Moby/RT-Editor erkennt man einen Superzustand daran, dass um ihn zwei Rahmen und nicht wie sonst nur ein Rahmen gezeichnet sind.

- **Ports:** Eine Transition, die durch benachbarte (darüber oder darunter liegenden) Hierarchieebenen führt, wird, wie Eingangs erwähnt, durch Kanten und so genannte Ports graphisch veranschaulicht. Man unterscheidet zwischen Inports (Eingänge), die in eine Ebene hineinführen und Outports (Ausgänge), die aus einer Ebene hinausführen. Es wird also z. B. in der einen Hierarchieebene eine Kante von einem Zustand zu einem Outport gezeichnet und in der benachbarten Hierarchieebene eine Kante von einem Inport zu

einem Zustand. Die Verbindung vom Outport zum Inport wird in den darüber liegenden Hierarchieebenen, durch die die Transition führt, ebenfalls durch eine Kante kenntlich gemacht. Im SIM-Format ist diese Verbindung in den entsprechenden Subautomaten durch einen *Connector* definiert.

- **Systemports:** Systemports nennt man die Ports auf der obersten Ebene, sie entsprechen den Schnittstellen für Peripheriegeräte der SPS. Durch die Definition eines Inports können mehreren Automaten die gleiche Eingabe einlesen.

- **Edges:** In MOBY/RT bestehen Transitionen aus einer Kante oder einer Kette von aneinander gehängten Kanten. Kanten haben einen Start- und einen Zielpunkt, welches Zustände oder Ports sein können. Kanten dürfen nur über Ports benachbarter Hierarchieebenen zusammengesetzt werden. Dabei darf ein Port nur Startpunkt einer einzigen Kante sein, jedoch darf er durchaus Zielpunkt von mehreren Kanten sein. Eine Kante, die in einem Outport eines Subautomaten startet, wird mit dem Schlüsselwort `connect` bei der Definition des zugehörigen Superzustandes definiert (siehe *Connector*). Dabei wird auch der Verbindungstyp mit angegeben. Also z. B.

  `connect` $Outport_1$ `into` $Superzustand_2$`.`*Inport*`;`

  Hierbei darf das Ziel durchaus im selben Subautomaten liegen. Eine Kante, die in einem Inport eines Subautomaten startet, wird gleich bei der Deklaration des Inports (siehe *SubInterface*) definiert. Eine Kante, die in einem Zustand startet (was über *EdgeDef* definiert wird), besitzt einen Guard und eine Aktion.

  - **Guard:** Ein Guard (*GuardExpr*) ist ein boolescher Ausdruck der durch die Belegung der Variablen (Ein-, Ausgabe- und lokalen Variablen) erfüllt sein muss, damit die zugehörige Transition genommen werden darf.
  - **Aktion:** Eine Aktion (*ActionExpr*) ist eine Menge von Zuweisungen die ausgeführt werden, wenn die zugehörige Transition genommen wird.

- **Stubbed Edges:** Eine Kante zu oder von einem Superzustand, die nicht mit einem Port des zugehörigen Subautomaten verbunden ist, nennt man „Stubbed Edge“. Man unterscheidet folgende zwei Fälle:

  - Eine Kante, die zu einem Superzustand führt und nicht mit einem seiner Inports verbunden ist, ist implizit mit dem Startzustand der Verfeinerung des Superzustandes verbunden.

  - Eine Kante, die von einem Superzustand ausgeht und nicht über einen seiner Outport hinaus führt, ist implizit mit allen Zuständen der Verfeinerung verbunden. Außerdem hat die mit dieser Kante gebildete Transition eine höhere Priorität als die Transition in der Verfeinerung.

  Eine „Stubbed Edge“ wird dadurch graphisch kenntlich gemacht, dass sie nur bis zum äußeren Rahmen des Superzustandes gezeichnet ist, wohingegen „normale“ Transitionen durch den äußeren Rahmen hindurch bis zum inneren Rahmen des Superzustandes reicht.

- **Verzögerung:** Jedem Zustand z ist ein Verzögerung (*DelayDef*) zugeordnet. Die Verzögerung besteht aus einer Verzögerungsbedingung $S_e(z)$ (in Moby/RT „delay condition“ oder auch „events to ignore“ genannt) und einer Verzögerungszeit $S_t(z)$ (in Moby/RT „delay time“ genannt). Die Verzögerungszeit $S_t(z)$ gibt an, wie lange die Eingabe die die Verzögerungsbedingung erfüllt, ignoriert wird. Oder anders gesagt, wie lange im aktuellen Zustand z gewartet wird, wenn die Verzögerungsbedingung erfüllt ist. Ist die Verzögerungsbedingung nicht erfüllt, dann muss auch nicht gewartet werden und eine Transition, deren Gurad erfüllt ist, kann genommen werden. Im SIM-Format wird dies durch das Schlüsselwort `delay` angegeben, gefolgt von eine positive Zahl, die die Länge der Verzögerung, die Verzögerungszeit, angibt. Danach folgt das Schlüsselwort `for` gefolgt von der Verzögerungsbedingung, die ein boolescher Ausdruck über die Eingabevariablen ist. Steht an Stelle diese Ausdrucks das Schlüsselwort `all`, dann ist dies gleichbedeutend mit dem booleschen Ausdruck `true`.

- **Verbindungstyp:** Der Verbindugstyp (*ConnectorTyp*) gibt Auskunft über das Ziel einer Kante:

  - `->` gehe zu einem „normalen“ Zustand oder Outport
  - `to` gehe zum Startzustand des Subautomaten des Superzustands (Stubbed Edge)
  - `into` gehe zum Inport des Subautomaten des Superzustands

- **Interface im Subautomten:** Das Interface im Subautomaten (*SubInterface*) steht gleich zu Beginn in der Subautomatendefinition. Hier werden die Outports des Subautomaten durch das Schlüsselwort `outport` und die Inports des Subautomaten durch das Schlüsselwort `inport` deklariert. Dabei wird hier auch das Ziel des Inports angegeben. Z. B.

  `inport` $Inport_1$ `to` $Superzustand_2$`;`

Wobei eine Kante von einem Inport zu einem lokalen (d. h. auf der selben Hierarchieebene im selben Subautomaten) Outport nicht möglich und auch nicht sinnvoll ist.

- **Kanäle:** Mit sogenannten Kanälen (*ChannelDef*) wird die Kommunikation zweier Automaten dargestellt. Dient eine Ausgabevariable eines Automaten einem anderen Automaten als Eingabe, dann wird die Ausgabevariable mit der Eingabevariable mit dem Schlüsselwort `channel` verbunden. Außerdem werden so ebenfalls die Ein- und Ausgabevariablen an die Systemports angeschlossen. Es wird immer eine Quelle mit einer Senke verbunden, wodurch drei verschiedene Verbindungskombinationen entstehen können, die folgende Kardinalitäten besitzen:

$$\begin{aligned} \textit{Inport}\ (1)\ \texttt{to}\ (n)\ \textit{Eingabevariable} \\ \textit{Ausgabevariable}\ (1)\ \texttt{to}\ (n)\ \textit{Outports} \\ \textit{Ausgabevariable}\ (1)\ \texttt{to}\ (n)\ \textit{Eingabevariable} \end{aligned}$$

  In Moby/RT wird dies durch `channel` *Quelle* `to` *Senke*`;` definiert

Eine Transition, die von einem Zustand ausgeht, der nicht in einem Subautomaten liegt, wird genommen, wenn ihr Gurard erfüllt ist und entweder die Verzögerungszeit des Zustandes abgelaufen ist oder die Verzögerungsbedingung nicht erfüllt ist, oder beides. Eine Transition, die von einem Zustand in einem Subautomaten ausgeht, muss desweiteren noch die Guards der Transitionen der ihr übergeordneten Superzustände (dies sind die Stubbed Edges) berücksichtigen, da diese eine höhere Priorität haben. Desweiteren muss überprüft werden, ob der Superzustand der durch die Transition verlassen werden soll auch verlassen werden darf. Es muss also überprüft werden, ob die Verzögerungszeit des Superzustandes abgelaufen ist, oder die Verzögerungsbedingung nicht erfüllt ist. Eine solche Transition kann also nur dann genommen werden wenn zusätzlich kein Guard einer Transition eines übergeordneten Superzustandes erfüllt ist und von allen Superzuständen z, die durch die Transition verlassen werden, muss der Verzögerungsausdruck $y_{t_z} \geq S_t(z) \vee \neg S_e(z)$ erfüllt sein (hierbei ist y_{t_z} eine Uhr gemäß Definition 3.3).

Eine Besonderheit bei den Automaten in Moby/RT ist, dass wenn eine Transition genommen werden kann, aber die Aktion dieser Transition einer Integervariable einen Wert zuweist, der nicht in ihrem Wertebereich liegt, dann wird die Transition zwar genommen aber die Aktion wird nicht ausgeführt. In Uppaal hingegen führt eine solche Transition zu einer Fehlerausgabe und zur Beendigung der Simulation bzw. der Verifikation.

Die Symbole in der Grammatik für Guard- und Aktionsausdrücke haben die übliche Bedeutung. Es sei nur erwähnt, dass '`#`' und '`<>`' Ungleichheit, '`=`' Gleichheit, '`/`' Division und '`%`' Modulo bedeuten.

Die hier angegebene Syntax des SIM Formats unterscheidet sich an zwei Stellen zu den in Moby/RT eingebbaren Automaten.

- In Moby/RT muss jeder Transition ein Guard angegeben werden, wohingegen hier in der *EdgeDef* der ganze Ausdruck '`condition`' *GuardExp* weggelassen werden kann. Ist hier also kein Guard angegeben, so wird dies als der Guard `condition true` interpretiert.
- In Moby/RT kann nur bei den Eingabevariablen der Startwert weggelassen werden, wohingegen hier die Angabe des Startwerts auch bei den Ausgabevariablen als auch bei den lokalen Variablen optional ist.

Beispiel 4.1 (Stanze)**.** Als ein weiteres Beispiel ist in Abbildung 4.1 die Steuerung einer Stanze in Form eines SPS-Automaten, der mit Moby/RT erstellt wurde zu sehen. Die Platten, in die die Stanze ein Loch stanzen soll, werden über ein Fließband zur Stanze hin und auch wieder weg geführt. Wird die Stanze eingeschaltet, so schaltet sich auch das Fließband an. Wenn in den ersten 500 Zeiteinheiten keine Platte unter der Stanze liegt, dann soll dies nicht zum Zustand Fehlfunktion führen, da es nach dem Einschalten bis zu 500 Zeiteinheiten dauern kann, bis die erste Platte die Stanze erreicht hat. Die Eingabevariable `Platte` signalisiert, ob der Strom von Platten auf dem Fließband geschlossen (`Platte=true`) oder unterbrochen (`Platte=false`) ist. Am Anfang wechselt sie von `false` auf `true`, wenn die erste Platte die Stanze erreicht hat und wechselt erst wieder auf `false`, wenn der Strom von Platten auf dem Fließband unterbrochen ist.

Zu Beginn ist die Stanze also aus und der Automat befindet sich im Startzustand `Stanze_aus`. Wird dann die Stanze eingeschaltet, so wechselt der Automat über die Transition (1) mit dem Guard `stanzen = an` in den Superzustand `Stanze_an`. Nach dem Einschalten der Stanze, wartet die Stanze maximal 500 Zeiteinheiten auf eine Platte bevor sie in den Fehlerzustand wechselt. Dies wird zum Einen durch die zum Superzustand gehörende Verzögerungszeit von 500 und der Verzögerungsbedingung `stanzen = an` ausgedrückt, was dafür sorgt, dass der Superzustand in den ersten 500 Zeiteinheiten nicht verlassen werden darf, solange die Stanze an ist. Zum Anderen darf die Transition (2), die die Abwärtsbewegung der Stanze auslöst, nicht genommen werden, solange der Guard der Transition (5) erfüllt ist, da diese eine höhere Priorität besitzt und somit die Transitionen im Subautomaten `Stanze_an` und damit die Stanze blockiert.

Die Transition (4) besteht aus zwei Kanten: eine vom Zustand `oben` zum Outport `ausschalten` und eine von diesem Outport zum Zustand `Stanze_aus`. Über diese Transition kann die Stanze jeder Zeit ausgeschaltet (`stanzen = aus`) werden, auch während der ersten 500 Zeiteinheiten, weil dann die Verzögerungsbedingung `stanzen = an` des Superzustandes `Stanze_an` und der beiden Zustände `Oben` und `Unten` nicht erfüllt ist und somit die genannten Zustände verlassen werden dürfen. Falls die Stanze gerade unten ist, wenn sie ausgeschaltet wird, wird vorher

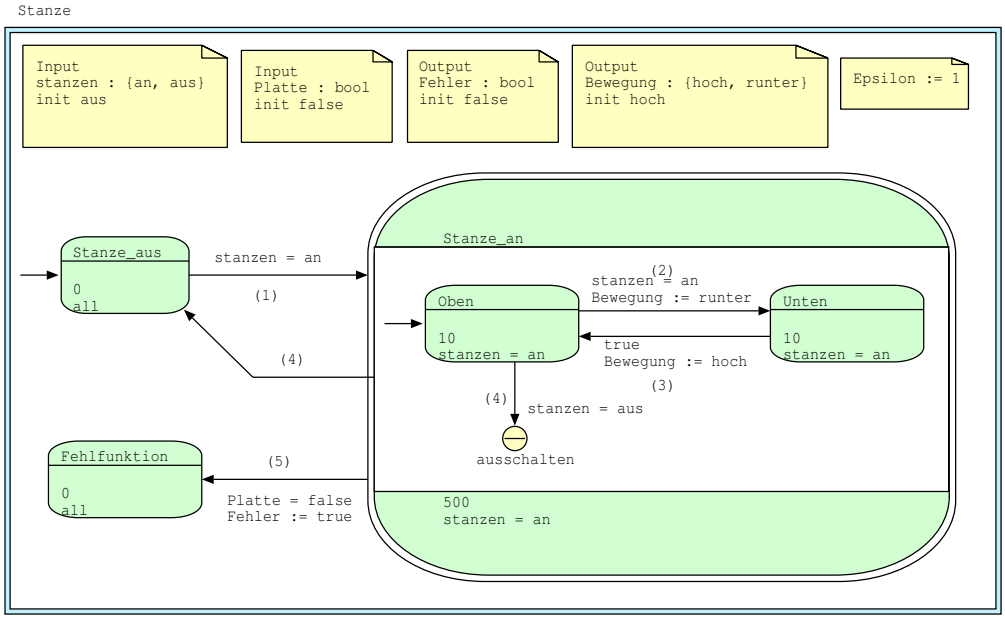

Abbildung 4.1.: Modelierung einer Stanze mittels eines MSPS-Automaten

noch die Stanze ohne Verzögerung über die Transition (3) nach oben gefahren. Sollte jedoch `Platte = false` sein, wenn die Stanze ausgeschaltet wird, so wird in den Fehlerzustand gewechselt, da die Transition (5) eine höhere Priorität als die Transition (4) besitzt. In diesem Fall wird die Stanze aus dem selben Grund auch nichtmehr nach oben gefahren.

Falls sich nach den ersten 500 Zeiteinheiten keine Platte unter der Stanze befindet, dann schaltet die Transition (5) und der Zustand Fehlfunktion wird eingenommen. Ansonsten wechselt der Automat zwischen den Zuständen `Oben` und `Unten` über die Transitionen (2) und (3) mit einer Verzögerung von jeweils 10 Zeiteinheiten.

Die `system` Zeile im SIM Format gibt an, welche Automaten auf der selben SPS modelliert sind. Die durch „`;`“ getrennten Automaten werden auf der selben SPS modelliert (sequentielle Komposition, siehe Abschnitt 3.3). Mit „`||`“ werden die einzelnen speicherprogrammierbaren Steuerungen getrennt (parallele Komposition). Also z. B. `system A ; B || C;` besagt, dass die Automaten A und B auf einer gemeinsamen SPS modelliert sind und der Automat C ist auf einer anderen SPS modelliert.

Beispiel 4.2 (Stanze Teil 2)**.** Der SPS-Automat der in Beispiel 4.1 angegebenen Stanze wird wie folgt im SIM-Format angegeben:

System → (*AutomatonBlock* [*SubautomatonBlock*])*
(*OCI_Def*)* *SystemDef*
AutomatonBlock → 'automaton' '{' *AutomatonDef* '}'
AutomatonDef → *EpsilonDef InterfaceBlock StartStateDef*
(*StateBlock* | *SuperStateBlock*)+
EpsilonDef → 'epsilon' ':=' NAT ';'
InterfaceBlock → 'interface' '{' (*InterfaceDef*)* '}'
StartStateDef → 'start' ID ';'
InterfaceDef → ('in' | 'out' | 'lokal') ID ':' *TypeDef* ['init' [*Value*]] ';'
Value → 'true' | 'false' | NUM | ID
StateBlock → 'state' ID '{' *StateDef* '}'
TypeDef → 'bool'
| '{' ID ['.' ID] (',' ID ['.' ID])* '}'
| '{' *IntInterval* (',' *IntInterval*)* '}'
IntInterval → NUM '..' NUM | NUM
StateDef → *DelayDef* (*EdgeDef*)*
DelayDef → 'delay' NAT 'for' ('all' | *BoolExpr*) ';'
EdgeDef → 'nextstate' *ConnectorTyp* ID ['.' ID] '{'
['condition' *GuardExpr* ';']
['action' *ActionExpr* ';'] '}'
ConnectorTyp → '->' | 'to' | 'into'
SubautomatonBlock → 'subautomaton' ID '{' *SubautomatonDef* '}'
SubautomatonDef → (*SubInterface*)* *StartStateDef* ';'
(*StateBlock* | *SuperStateBlock*)+
SubInterface → 'outport' ID ';' | 'inport' *SubConnectorDef* ';'
SuperStateBlock → 'subautomaton' ID '{' *SuperStateDef* '}'
SuperStateDef → *DelayDef* (*Connector* | *EdgeDef*)*
Connector → 'connect' *SubConnectorDef* ';'
SubConnectorDef → ID ['.' ID] *ConnectorTyp* ID ['.' ID]
OCI_Def → *ChannelDef* | *PortDef*
PortDef → ('outport' | 'inport') (ID ':' ID | ':' ID) ';'
ChannelDef → 'channel' *SysPortDecl* 'to' *SysPortDecl* ';'
SysPortDef → ID (':' ID | '.' ID) | ':' ID
SystemDef → 'system' [ID] (';' ID | '||' ID)* ';'

Abbildung 4.2.: Gramatik des SIM Formats.

GuardExpr → *BoolExpr*
BoolExpr → *AtomicBool*
| *BoolExpr BinaryOpBool BoolExpr*
| *UnaryOpBool BoolExpr*
| *If BoolExpr Then BoolExpr* [*Else BoolExpr*] *Endif*
| '(' *BoolExpr* ')'
| *AtomicBool Equality AtomicBool*
| *IntExpr Comparison IntExpr*
BinaryOpBool → 'and' | 'or' | 'AND' | 'OR' | '&' | '|'
AtomicBool → ID | 'true' | 'TRUE' | 'false' | 'FALSE'
If → 'if' | 'IF'
Then → 'then' | 'THEN'
Else → 'else' | 'ELSE'
Endif → 'endif' | 'ENDIF'
Comparison → *Equality* | *Inequality*
Equality → '=' | '<>' | '#'
Inequality → '>=' | '<=' | '>' | '<'
UnaryOpBool → 'not' | 'NOT' | '!'
IntExpr → ID | NAT
| *IntExpr BinaryOpInt IntExpr*
| '-' *IntExpr*
| '(' *IntExpr* ')'
BinaryOpInt → '+' | '-' | '*' | '/' | '%'
ActionExpr → *Action* (';')* | *Action* (';')+ *ActionExpr*
Action → ID ':=' *Expression*
| *If BoolExpr Then ActionExpr* [*Else ActionExpr*] *Endif*
Expression → *BoolExpr* | *IntExpr*

Abbildung 4.3.: Grammatik für Guard- und Aktionsausdrücke

```
automaton Stanze {
  epsilon := 1;
  interface {
    in stanzen : {an, aus} init aus;
    out Fehler : bool init false;
    in Platte : bool init false;
    out Bewegung : {hoch, runter} init hoch;
  }
  start Stanze_aus;
  state Stanze_aus {
    delay 0 for all;
    nextstate to stanze_an {
      condition stanzen = an;
    }
  }
  state Fehlfunktion {
    delay 0 for all;
  }
  subautomaton stanze_an {
    delay 500 for stanzen = an;
    nextstate -> Fehlfunktion {
      condition Platte = false;
      action Fehler := true;
    }
    connect ausschalten -> Stanze_aus;
  }
}
subautomaton stanze_an {
  outport ausschalten;
  start Oben;
  state Oben {
    delay 10 for stanzen = an;
    nextstate -> Unten {
      condition stanzen = an;
      action Bewegung := runter;
    }
    nextstate -> ausschalten {
      condition stanzen = aus;
    }
  }
  state Unten {
    delay 10 for stanzen = an;
```

```
    nextstate -> Oben {
      condition true;
      action Bewegung := hoch;
    }
  }
}
system Stanze;
```

Dieser Automat wird durch folgenden MSPS-Automaten beschrieben:

$$\mathcal{M}_{Stanze} := (Q, S, \Sigma, L, \Omega, \delta, g_0, \varepsilon, T, S_t, S_e)$$

hierbei ist

$$\begin{aligned}
Q &:= \{\texttt{Stanze_aus, Oben, Unten, Fehlfunktion}\} \\
S &:= \{\texttt{Stanze_an}\} \\
\Sigma &:= \{\texttt{stanzen, Platte}\} \\
L &:= \emptyset \\
\Omega &:= \{\texttt{Fehler, Stanze}\} \\
\delta &:= \{(\texttt{Stanze_aus, stanzen = an, } \lambda\texttt{, Stanze_an}) \\
&\qquad (\texttt{Oben, stanzen=an, Bewegung:=runter, Unten}) \\
&\qquad (\texttt{Unten, true, Bewegung:=hoch, Oben}) \\
&\qquad (\texttt{Oben, stanzen=aus, } \lambda\texttt{, Stanze_aus}) \\
&\qquad (\texttt{Stanze_an, Platte = false, Fehler:=true, Fehlfunktion}) \\
g_0 &:= (\texttt{Stanze_aus}, [\texttt{Fehler} \mapsto \texttt{false}][\texttt{Bewegung} \mapsto \texttt{hoch}]) \\
\varepsilon &:= 1 \\
T &:= (\{r, \texttt{ Stanze_aus, Stanze_an, Oben, Unten, Fehlfunktion}\}, H) \\
H &:= \{(r, \texttt{ Stanze_aus}), (r, \texttt{ Stanze_an}), \\
&\qquad (r, \texttt{ Fehlfunktion}), (\texttt{Stanze_an, Oben}), (\texttt{Stanze_an, Unten})\}
\end{aligned}$$

Eine graphische Darstellung des Hierarchiebaums ist in Abbildung 4.4 zu sehen. Weiter sind die Verzögerungszeiten definiert als

$$\begin{aligned}
S_t(\texttt{Stanze_aus}) &:= 0 \\
S_t(\texttt{Stanze_an}) &:= 500 \\
S_t(\texttt{Oben}) &:= 10 \\
S_t(\texttt{Unten}) &:= 10 \\
S_t(\texttt{Fehlfunktion}) &:= 0
\end{aligned}$$

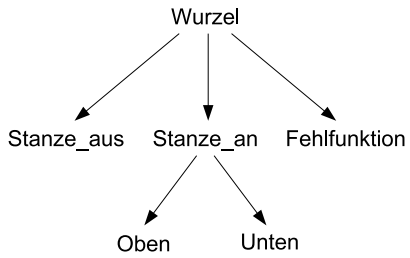

Abbildung 4.4.: Der Hierarchiebaum das MSPS-Automaten aus Beispiel 4.2

und die Verzögerungsbedingungen als

$$\begin{aligned}
S_e(\texttt{Stanze_aus}) &:= \texttt{true} \\
S_e(\texttt{Stanze_an}) &:= \texttt{stanzen=true} \\
S_e(\texttt{Oben}) &:= \texttt{stanzen=true} \\
S_e(\texttt{Unten}) &:= \texttt{stanzen=true} \\
S_e(\texttt{Fehlfunktion}) &:= \texttt{true}
\end{aligned}$$

Die Semantik dieses Automaten wird durch den folgenden GSPS-Automaten definiert:

$$\mathcal{G}(\mathcal{M}_{Stanze}) := (Q, \Sigma, L, T_{\mathcal{M}}, \Omega, \delta_{\mathcal{M}}, g_0, \varepsilon, \Xi_{\mathcal{M}}, \Theta_{\mathcal{M}})$$

hierbei ist

$$\begin{aligned}
T_{\mathcal{M}} &:= \{t_{\texttt{Stanze_an}}, t_{\texttt{Oben}}, t_{\texttt{Unten}}\} \\
\Xi_{\mathcal{M}}(t_{\texttt{Stanze_an}}) &:= \{\texttt{Oben}, \texttt{Unten}\} \\
\Xi_{\mathcal{M}}(t_{\texttt{Oben}}) &:= \{\texttt{Oben}\} \\
\Xi_{\mathcal{M}}(t_{\texttt{Unten}}) &:= \{\texttt{Unten}\} \\
\Theta_{\mathcal{M}}(t_{\texttt{Stanze_an}}) &:= S_t(\texttt{Stanze_an}) = 500 \\
\Theta_{\mathcal{M}}(t_{\texttt{Oben}}) &:= S_t(\texttt{Oben}) = 10 \\
\Theta_{\mathcal{M}}(t_{\texttt{Unten}}) &:= S_t(\texttt{Unten}) = 10
\end{aligned}$$

und $\delta_{\mathcal{M}}$ gemäß Tabelle 4.1.

q	v							$\delta_{\mathcal{M}}(q,v) = (q', v')$		
	Eingabevar.		Ausgabevar.		Timervariablen			q'	v'	
	stanzen	Platte	Fehler	Bew.	$t_{\text{Stanzen\_an}}$	t_{Oben}	t_{Unten}		Fehler	Bew.
Stanze_aus	an	*	*	*	*	*	*	Oben	S	S
Stanze_aus	aus	*	*	*	*	*	*	Stanze_aus	S	S
Oben	an	true	*	*	*	false	*	Unten	S	runter
Oben	an	true	*	*	*	true	*	Oben	S	S
Oben	aus	true	*	*	*	*	*	Stanze_aus	S	S
Oben	an	false	*	*	false	false	*	Fehlfunktion	true	S
Oben	an	false	*	*	true	*	*	Oben	S	S
Oben	an	false	*	*	false	true	*	Oben	S	S
Oben	aus	false	*	*	*	*	*	Fehlfunktion	true	S
Unten	an	true	*	*	*	*	true	Unten	S	S
Unten	an	true	*	*	*	*	false	Oben	S	hoch
Unten	an	false	*	*	true	*	*	Unten	S	S
Unten	an	false	*	*	false	*	true	Unten	S	S
Unten	an	false	*	*	false	*	false	Fehlfunktion	true	S
Unten	aus	true	*	*	*	*	*	Oben	S	hoch
Unten	aus	false	*	*	false	*	*	Fehlfunktion	true	S

Tabelle 4.1.: Übergangsfunktion. Das Symbol „*“ bezeichnet Werte die für das Ergebnis von $\delta_{\mathcal{M}}(q, v)$ irrelevant sind. Das Symbol „S“ in den Ergebnisspalten von $\delta_{\mathcal{M}}(q, v)$ gibt an, dass sich dieser Wert durch den Zustandsübergang nicht geändert hat.

5. Transformation von SIM in die Uppaal-Syntax

In diesem Kapitel wird nun die Transformation eines Systems von MSPS-Automaten im SIM-Format, in die Realzeitautomatensyntax von Uppaal beschrieben. In der Habilitationsschrift von Dierks [7] und in Moby/RT wird für jede, den Guard erfüllende Belegung ein Transition erstellt. Um so mehr Variablen im Guard vorkommen und umso größer deren Wertebereich ist, desto mehr Transitionen werden benötigt. Dies war früher nötig, da in der Uppaal-Syntax nur Konjunktionen im Guard erlaubt waren. Inzwischen müssen aber nur noch die Uhrenbedingungen auf oberster Ebene mit einer Konjunktion verbunden sein, für die restlichen Bedingung gibt es keine Restriktionen mehr, so dass der Guard, so wie er ist, übernommen werden kann. Dadurch ist eine natürlichere Übersetzung möglich, da es mehr direkte Übereinstimmungen gibt. Der Unterschied zu der in der Habilitationsschrift von Dierks vorgestellten Übersetzung liegt darin, dass hier

- der „starting“ Zustand des PLC Prozesses weggelassen wurde, da die Variablen gleich bei der Deklaration initialisiert werden und
- der Guard der Transitionen bleibt erhalten und bekommt noch weitere Bedingungen hinzu. D. h. der Guard besteht nicht mehr aus Konjunktionen von Belegungsabfragen, sonder ist ein komplexer boolescher Ausdruck. Hierdurch können einige Transitionen eingespart werden, da nicht für jede den ursprünglichen Guard erfüllende Belegung eine neue Transition hinzugefügt werden muss.

5.1. Transformation der Uhren, Variablen und Ausdrücke

Entsprechend der Semantik wird eine Uhr `PLC`i`_z` und eine Uhr `PLC`i`_x` für z bzw. x einer jeden SPS erstellt. Desweiteren wird für jeden Automaten A je eine Uhr A`_c`i für jede Hierarchieebene i von A erstellt. Man braucht nicht für jede Timervariable eine eigene Uhr verwenden, da Timervariablen mit disjunkten Aktivierungsregionen eine gemeinsame Uhr verwenden können.

Die 3 Variablentypen werde wie folgt übersetzt. Die booleschen Variablen können übernommen werde, da Uppaal diese auch besitzt. Da die Integervariablen

in MOBY/RT immer mit einem Wertebereichsinterval angegeben werden müssen, können diese ebenfalls übernommen werden, da sie den begrenzten Integervariablen in UPPAAL entsprechen. Variablen vom Typ Enumeration werden in Begrenzten Intergervariablen konvertiert, derart, dass jeder Komponente der Enumeration ein Intergerwert zugeordnet wird entsprechend ihrer Position in der Deklaration der Enumeration.

Die Guard- und Aktionsausdrücke können leicht an den UPPAAL Syntax angepasst werden. Hierfür wird die Abbildung U definiert, die Guard- und Aktionsausdrücke auf ihre entprechenden Ausdrücke in UPPAAL abbildet. Dabei werden die folgenden Symbole ersetzt (vergleiche Abbildung 4.3):

SIM	UPPAAL
'and', 'AND', '&'	'&&'
'or', 'OR', '|'	'||'
'='	'=='
'<>', '#'	'!='
':='	'='
'not', 'NOT'	'!'

Desweiteren wird das If-Then-Else-Konstrukt durch den ?-Operator ersetzt. Für Guardausdrücke ist dies trivial:

- $U(\mathit{If}\ g\ \mathit{Then}\ g_1\ \mathit{Endif}) := U(g)$ `?` $U(g_1)$`: true`
- $U(\mathit{If}\ g\ \mathit{Then}\ g_1\ \mathit{Else}\ g_2\ \mathit{Endif}) := U(g)$ `?` $U(g_1)$ `:` $U(g_2)$

Bei Aktionsausdrücken wird ein kleiner Trick benötigt, da im ?-Operator keine Listen von Zuweisungen stehen dürfen. Ein Aktionsausdruck der Form

$$\mathit{If}\ g\ \mathit{Then}\ a_1\texttt{;}\ldots\texttt{;}a_n\ \mathit{Endif}$$

wobei $a_1, \ldots, a_n$ Ausdrücke der Form *Action* seien, wird zu

$$U(g)\ \texttt{?}\ \texttt{((}U(a_1)\texttt{)==}\cdots\texttt{==(}U(a_n)\texttt{)) : true}$$

übersetzt und entsprechend für Aktionsausdrücke mit einem Else-Teil.

Um die Einlese- und die Aktualisierungsphase der SPS zu simulieren, werden für jede Ein- und Ausgabvariable im SIM-Format zwei Variablen benötigt. Eine „interne“ Variable für den internen Wert und eine „externe“ für den von außen anliegenden Wert bzw. für den nach außen sichtbaren Wert.

Für jede Eingabevariable eines Automaten werden keine, eine oder zwei Variablen erzeugt, je nach dem woher die Eingabevariable ihre Werte bekommt:

- Es wird keine Eingabevariable erzeugt, wenn die Eingabevariable ihre Eingabe über ein `channel` von einem anderen Automaten auf der gleichen SPS

erhält. Dann muss der Wert nämlich nicht eingelesen werden und es kann direkt auf die interne Ausgabevariable des anderen Automaten zugegriffen werden. Es wird also überall da wo die Eingabevariable abgefragt wird, diese mit der internen Ausgabevariable des anderen Automaten ersetzt.

- Es wird eine Eingabevariable erzeugt, wenn die Eingabevariable über ein `channel` mit einer Ausgabevariable eines anderen Automaten auf einer anderen SPS verbunden ist. Dann braucht man nämlich nur eine Variable für den internen Wert der Eingabevariable anlegen (`polled_`*Automat*`$IN$`*Eingabevariable*), die externe Variable ist mit der Ausgabevariable des anderen Automaten schon gegeben. In der Einlesephase erfolgt dann die Zuweisung

  `polled_`$Automat_1$`$IN$`*Eingabevariable* = $Automat_2$`$OUT$`*Ausgabevariable*

- Es werden zwei Eingabevariablen erzeugt, wenn die Eingabevariable entweder über ein `channel` mit einem System-Inport verbunden ist oder wenn sie überhaupt nicht verbunden ist. Für den externen Wert wird im ersten Fall die Variable *Systeminport* und im zweiten Fall die Variable *Automat*`$IN$`*Eingabevariable* erstellt. Für den internen Wert wird in beiden Fällen die Variable `polled_`*Automat*`$IN$`*Eingabevariable* erstellt. In der Einlesephase erfolgt dann die Zuweisung

  `polled_`*Automat*`$IN$`*Eingabevariable* = *Systeminport*

  bzw.

  `polled_`*Automat*`$IN$`*Eingabevariable* = *Automat*`$IN$`*Eingabevariable*

Ebenso werden auch für die Ausgabevariablen zwei Variablen definiert. Eine für die nach außen sichtbare Ausgabe (*Automat*`$OUT$`*Ausgabevariable* bzw. *Systemoutport*, *Systemoutport* falls die Ausgabevariable an ein Systemoutport angeschlossen ist und *Ausgabevariable* sonst) und eine für den aktuellen internen Wert der Ausgabe (deren Wert sich in der Berechnungphase ändern kann). Analog zu den Eingabvariablen wird der letzteren Variable das Präfix „`update_`“ beigefügt, also `update_`*Automat*`$OUT$`*Ausgabevariable*.

In der Aktualisierungsphase erfolgt dann die Zuweisung

Automat`$OUT$`*Ausgabevariable* = `update_`*Automat*`$OUT$`*Ausgabevariable*

bzw.

Systemoutport = `update_`*Automat*`$OUT$`*Ausgabevariable*

5.2. Transformation der Zustände und Transitionen

Für jeden Automat wird eine *interne Zustandsvariable* „*Automat*_IntState“ definiert, deren Wertebereich das Intervall $[0, |Q|]$ ist, wobei $|Q|$ die Anzahl der Zustände des Automaten beschreibt. Ist der Wert dieser Variablen 0, so befindet sich die SPS in der Einlesephase. Die anderen Werte stehen für die Zustände des Automaten.

Jeder Automat wird in einen Prozess[1] umgewandelt. Die Zustände[2] des Prozesses sind die Zustände des Automaten. Ebenso ist der Startzustand der gleiche. Deshalb bleibt, bei der Übersetzung in die Syntax von UPPAAL, der Zustandsraum jeder Komponente unverändert. Dies ist sehr vorteilhaft, wenn Eigenschaften des Systems verifiziert werden sollen, die sich auf den Zustandsraum der Komponenten beziehen.

Die Transitionen des Prozesses entsprechen denen des Automaten. Es wird für jede Transition des Automaten eine Transition erstellt, die den internen Zustandsübergang wiederspiegelt und eine, die den nach außen sichtbaren Zustandsübergang modelliert. Desweiteren muss noch eine Loop-Transition, die in MOBY/RT implizit vorhanden ist, explizit mit aufgeführt werden. Das Synchronisationslabel an den Transitionen dient der Synchronisation mit dem entsprechenden `PLC` Prozess der das zyklische Verhalten der SPS modelliert.

Sei zum Beispiel $\mathcal{M} = (Q, \emptyset, \Sigma, L, \Omega, \delta, g_0, \varepsilon, T, S_t, S_e)$ ein MSPS-Automat mit dem Namen A, der keine Superzustände besitzt. Eine Transition $(q, g, a, q') \in \delta$ mit $d := S_t(q)$ und $e := S_e(q)$ wird vom SIM Format:

```
state q {
  delay d for e;
  nextstate -> q' {
    condition g;
    action a;
  }
}
```

wie folgt in den UPPAAL-Syntax transformiert:

1. Ein interner Zustandsübergang. Diese Transition synchronisiert mit der Transition des Zustandes `executing_`A und wird somit in der Berechnungsphase der SPS ausgeführt:

   ```
   q -> q
     { guard (A_IntState == 0) && U(g) && (!U(e) || x <= d);
   ```

[1] In UPPAAL heißen die Automaten Prozesse, also auf deutsch Prozesse

[2] Die Knoten des Realzeitautomaten werden in der UPPAAL-Syntax States, also Zustände genannt.

```
    sync channel_PLC_A;
    assign U(a), A_IntState = n_q';
  }
```

2. Ein nach außen sichbarer („externer“) Zustandsübergang. Diese Transition synchronisiert mit der Transition des Zustandes `resetting_`A und wird somit in der Aktualisierungsphase der SPS ausgeführt:

```
q -> q'
  { guard A_IntState == n_q';
    sync channel_PLC_A;
    assign A_IntState = 0, x = 0, A_c0 = 0;
  }
```

   Falls $q = q'$ ist, entfällt die Zuweisung `A_c0 = 0`, da bei einem Loop die Uhr nicht zurückgesetzt wird, weil keine neue Aktivierungsregion betreten wird.

Außerdem muss für jeden Zustand eine Loop-Transition explizit angegeben werden, die genommen werden kann, wenn keine anderen Transition des Zustandes möglich ist. Dies ist der interne Zustandsübergang des Loops, der wie alle internen Zustandsübergänge durch die Synchronisation in der Berechnungsphase ausgeführt wird:

```
q -> q
  { guard A_IntState == 0 && (!U(g) || (U(e) && x < d));
    sync channel_PLC_A;
    assign A_IntState = n_q;
  }
```

Und hier ist der nach außen sichtbare („externe“) Zustandsübergang des Loops. Diese Transition synchronisiert mit der Transition des `resetting_`A Zustandes und wird somit in der Aktualisierungsphase der SPS ausgeführt:

```
q -> q
  { guard A_IntState == n_q;
    sync channel_PLC_A;
    assign A_IntState = 0;
  }
```

Sind dem Zustand noch Superzustände übergeordnet (d. h. der Zustand liegt in einem Subautomaten), dann wird der Guard in der UPPAAL-Syntax komplexer und die Loop-Transition muss unter Umständen in mehrere Transitionen aufgespalten werden, da der Guard mehrere Uhrenbedingungen, die mit einer Disjunktion miteinander verbunden sind, enthalten kann und dies aber in UPPAAL nicht zulässig ist. Wie im Abschnitt 2.2 beschrieben, dürfen in der UPPAAL-Syntax die

Uhrenbedingungen nur durch Konjunktionen verbunden sein. Eine genau Beschreibung der Übersetzung der Transitionen erfolgt in Abschnitt 5.5.

5.3. Simulation der SPS-Hardware – Der PLC-Prozess

Um das Verhalten der Hardware zu simulieren wird für jede modellierte SPS i ein sogenannter PLCi Prozess erstellt.

```
process PLCi {
state polling                                  { PLCi_z <= z },
      testing                                  { PLCi_z <= z },
      executing_Automat_1                      { PLCi_z <= z },
        ⋮
      executing_Automat_n                      { PLCi_z <= z },
      updating                                 { PLCi_z <= z },
      resetting_Automat_1                      { PLCi_z <= 0 },
        ⋮
      resetting_Automat_n                      { PLCi_z <= 0 };
commit starting,
       executing_Automat_1,..., executing_Automat_n,
       resetting_Automat_1,..., resetting_Automat_n;
init polling;

trans
polling -> testing
  { guard PLCi_z > 0, PLCi_x > 0;
    assign polled_Automat_i_Eingabevariable
                = { Automat_i_Eingabevariable
                    Inport
                    Automat_j_Ausgabevariable },
                ⋮
  }

testing -> executing_Automat_1 { },
executing_Automat_1 -> executing_Automat_2
  {sync channel_PLCi_Automat_1?; },
  ⋮
executing_Automat_n-1 -> executing_Automat_n
  { sync channel_PLCi_Automat_n-1?; },
```

executing_$Automat_n$ -> updating
  { sync channel_PLCi_$Automat_n$?; },

updating -> resetting_$Automat_1$
  { assign PLCi_z = 0, PLCi_x = 0,
    $\left\{\begin{matrix} Automat_i\_Ausgabevariable \\ Outport \end{matrix}\right\}$ = update_$Automat_i$_$Ausgabevariable$,
    $\vdots$
  },

resetting_$Automat_1$ -> resetting_$Automat_2$
  { sync channel_PLCi_$Automat_1$; },
resetting_$Automat_2$ -> resetting_$Automat_3$
  { sync channel_PLCi_$Automat_2$; },
  $\vdots$
resetting_$Automat_{n-1}$ -> resetting_$Automat_n$
  { sync channel_PLCi_$Automat_{n-1}$; },
resetting_$Automat_n$ -> polling
  { sync channel_PLCi_$Automat_{n-1}$; };
}

In der Habilitationsschrift von Dierks wird im PLC-Prozess ein weiterer Zustand „starting“ hinzugefügt, der auch der Startzustand ist. Dieser Zustand wird nur einmal als aller erstes besucht. Beim Übergang von starting nach polling werden dann, den lokalen Variablen und den Ausgabevariablen ein Startwert zugewiesen. Dieser Zustand kann hier aber weggelassen werden, da alle Variablen bereits bei der Deklarierung initialisiert werden.

Die Bedingung, dass ein Zyklus höchstens z Zeiteinheiten dauern darf, wird durch die Invariante PLCi_z <= z, die in geschweiften Klammern neben einer Zustandsdeklaration steht, realisiert.

Der Zustand polling repräsentiert die Einlesephase. Hier vergeht etwas Zeit, solange die Invariante, nicht verletzt wird. Der Guard der Transition von polling nach testing stellt sicher, dass keine Eingabewerte eingelesen werden, die nur für einen Zeitpunkt anliegen (PLCi_x > 0) und das etwas Zeit im Zustand polling vergangen ist. Ist der Guard erfüllt, dann werden die anliegenden Eingaben eingelesen. Somit implementiert dies die Transition (2) aus Definition 3.3.

Die Zustände von testing bis zum letzten executing_ Zustand repräsentieren die Berechnungsphase und die Transitionen von testing nach updating über die dazwischen liegenden Zustände executing_$Automat_1, \ldots,$ executing_$Automat_n$ realisieren die Transition (3) aus Definition 3.3. Hier werden im Unterschied zu (3) nicht die Werte der Timervariablen τ gemerkt, um später in der Aktualisierungsphase die Ausgabe bestimmen zu können, sondern es wird für jede Ausgabevariable

(wie oben schon beschrieben) zwei Variablen angelegt. Eine (`updating_...`), die den internen Wert speichert und ein, die den nach außen sichbaren Wert speichert. Durch die Synchronisationen mit den Prozessen $Automat_1$ bis $Automat_n$ werden diese gezwungen eine Transition auszuführen. Dadurch das in `testing` Zeit vergehen darf, wird somit die Dauer der Berechnungen in der Berechnungsphase simuliert. Da alle `executing_` Zustände die Eigenschaft `commit` haben; es also keine Zeit in ihnen vergehen darf; werden die Transitionen zwischen den `executing_` Zuständen bis zum `updating` hintereinander aber im gleichen Zeitpunkt ausgeführt. Dadurch wird das Verhalten eines GSPS-Automaten simuliert, indem ein Zustandsübergang stattgefunden hat.

Die Zustände von `updating` bis zum letzten `resetting_` Zustand repräsentieren die Aktualisierungsphase. In `updating` darf wieder Zeit vergehen. Bei der Transition von `updating` zum ersten `resetting_` Zustand wird die z und die x Uhr auf 0 zurückgesetzt und alle Ausgabevariablen werden durch ihre entsprechende internen Ausgabvariablen (`update_...`) aktuallisiert, wodurch die nach außen sichtbare Ausgabe modelliert wird. Da alle `resetting_` Zustände die Eigenschaft `commit` haben, werden die Transitionen zwischen dem ersten `resetting_` Zustand und `polling` hintereinander, im selben Zeitpunkt ausgeführt. Durch die Synchronisationsbeschriftung werden die entsprechenden Prozesse gezwungen ebenfalls eine Transition auszuführen. Hier sind das die Transitionen in den Prozessen die im Assignment der Transition die Zuweisung *Automat*`_IntState = 0` stehen haben, also der interne Zustand wider auf polling gesetzt wird. Also die, die oben mit „nach außen sichbarer („externer“) Zustandsübergang“ beschrieben wurden. Dies entspricht den Transitionen vom Typ (4) aus Definition 3.3.

5.4. Simulation der Eingabe – Der Drive-Prozess

Schließlich wird für jede Eingabevariable eines Automaten, die nicht über `channel` oder `connect` verbunden ist, und für jeden Inport des Systems ein sogenannter `drive` Prozess erstellt. Diese Prozesse implementieren die Transitionen vom Typ (1) aus Definition 3.3 und simulieren die, durch die Umwelt bereitgestellten Eingaben, die von der SPS in der Einlesephase eingelesen werden. Der `drive` Prozess für eine Eingabevariable eines Automaten, der auf der SPS i realisiert ist, hat die Form:

```
process drive_Automat_Eingabevariable {
state loop;
init loop;
trans
loop
  -> loop
    { guard Automat_Eingabevariable != Wert_1;
```

```
        assign Automat_Eingabevariable = Wert_1, PLCi_x = 0; },
      ⋮
    -> loop
      { guard Automat_Eingabevariable != Wert_n;
        assign Automat_Eingabevariable = Wert_n, PLCi_x = 0; };
}
```

hierbei sei n die Anzahl der möglichen Werte, die die Eingabevariable annehmen kann.

Da ein Inport mit mehreren Automaten verbunden sein kann und diese auf mehr als einer SPS realisiert sein können, kann es mehrere speicherprogrammierbare Steuerungen geben, die in der Einlesephase ihren Wert von diesem Inport einlesen. Daher muss für jede zugehörige SPS i_j die x-Uhr zurückgesetzt werden. Der `drive` Prozess für einen Inport mit n möglichen Werten hat daher die Form

```
process drive_Inport {
state loop;
init loop;
trans
loop
  -> loop
    { guard Inport != Wert_1;
      assign Inport = Wert_1, PLCi_1_x = 0,..., PLCi_m_x = 0; },
    ⋮
  -> loop
    { guard Inport != Wert_n;
      assign Inport = Wert_n, PLCi_1_x = 0,..., PLCi_m_x = 0; };
}
```

5.5. Transformation der Guards eines MSPS-Automaten

Sei im weiteren $\mathcal{M} = (Q, S, \Sigma, L, \Omega, \delta, g_0, \varepsilon, T, S_t, S_e)$ ein MSPS-Automat. Zur besseren Lesbarkeit werden hier Guardausdrücke metasprachlich mit $\neg$, $\wedge$ und $\vee$ verknüpft. Dabei sei $\bigwedge_{g \in \emptyset} g := \top$ und $\bigvee_{g \in \emptyset} g := \bot$. Außerdem wird im Folgenden bei einem Guardausdruck g der Einfachheit halber nicht zwischen g und seiner Übersetzung in den UPPAAL-Syntax $U(g)$ unterschieden.

Definition 5.1. Für $t \in T$ sei $E_t := \{(s, g, a, q) \in \delta \mid s \preceq t\}$ die Menge der Transitionen von t. ■

Für die Wurzel r des Hierarchiebaums T ist also $E_r = \emptyset$, da nach Definition 3.10 die Wurzel nicht in $Q \cup S$ liegt und somit keine Transition in δ existiert, die in r startet.

Wie in der Definition 3.11 der Semantik eines MSPS-Automaten beschrieben, ist eine Transition $(s, g, a, q') \in E_q$ in $q \in Q$ erlaubt, falls sie Element von $D_{\mathcal{M}}(q, v)$ ist. Die Belegung v ist dabei die aktuelle Belegung während eines Laufs. Nach Definition 3.11 ist eine Transition $(s, g, a, q') \in E_q$ genau dann in $D_{\mathcal{M}}(q, v)$, wenn für die Belegungen v gilt

(ii) $g(v) = \top \wedge \bigwedge_{(s',g',a',q'') \in \delta, s' \prec s} g'(v) = \bot$

(iii) $\bigwedge_{s' \preceq q, s' \not\preceq q'} \big((S_e(s'))(v) = \bot \vee v(t_{s'}) = \bot\big)$

Dies kann in UPPAAL durch den Guard

$$\theta_q^{s,g,q'} := g \wedge \bigwedge_{(s',g',a',q'') \in \delta, s' \prec s} \neg g' \wedge \bigwedge_{s' \preceq q', s' \not\preceq q} (\neg S_e(s') \vee y_{t_{s'}} \geq S_t(s'))$$

ausgedrückt werden.

Falls v der Art ist, dass keine Transition möglich ist, d. h. $D_{\mathcal{M}}(q, v) = \emptyset$, dann wird ein Loop ausgeführt. Der Guard dieser Loop-Transition kann wie folgt formuliert werden

$$L_q := \bigwedge_{(s,g,a,q') \in E_q} \neg \theta_q^{s,g,q'}$$

Dieser Guard kann aber Disjunktionen von Uhrenbedingungen enthalten und muss deshalb umformuliert werden, damit er an den „obersten“ Disjunktionen in mehrere Transitionen aufgeteilt werden kann. Hierzu werden einige Ausdrücke als Abkürzung definiert.

Definition 5.2. Für $s \in Q \cup S$ sei $d_s := \neg S_e(s) \vee y_{t_s} \geq S_t(s)$ der *Verzögerungsausdruck* von s. Weiter seien für $t \in T$ und $q \in Q$ die Hilfsausdrücke

$$H_t := \bigwedge_{(u,g,a,q) \in E_t} \neg g$$

$$\Delta_t^q := \bigwedge_{w_t^q \prec u \preceq t} d_u$$

definiert. Schließlich sei für $t \in T$ und $(s, g, a, q) \in E_t$ der Guard

$$\theta_t^{s,g,q} := g \wedge H_{s-1} \wedge \Delta_t^q$$

sowie der Loop-Guard von t

$$L_t := \bigwedge_{(u,g,a,q) \in E_t} \neg \theta_t^{u,g,q}$$

definiert. ■

Zu Beginn seien einige Eigenschaften dieser Definition festgehalten, die später noch gebraucht werden.

Bemerkung 5.3. Nach Definition ist $H_r \iff \top$, da $E_r = \emptyset$, und es gilt für alle $s \succ r$

$$H_s \iff H_{s-1} \wedge \bigwedge_{(s,g,a,q)\in\delta} \neg g$$

Weiter gilt daher $\neg H_s \implies \neg H_t$ für alle $s \preceq t$.

Bemerkung 5.4. Man beachte, dass $w_s^t = s \iff s \preceq t$ und $w_s^t \prec s \iff s \not\preceq t$ ist. Daher gilt für $s \in Q \cup S$ und $q \in Q$

$$\Delta_s^q \iff \begin{cases} \Delta_{s-1}^q \wedge d_s & \text{falls } s \not\preceq q \\ \Delta_{s-1}^q & \text{falls } s \preceq q \end{cases}$$

und somit für $t \in Q \cup S$ und $(s, g, a, q) \in \delta$

$$\theta_t^{s,g,q} \iff \begin{cases} \theta_{t-1}^{s,g,q} \wedge d_t & \text{falls } t \not\preceq q \\ \theta_{t-1}^{s,g,q} & \text{falls } t \preceq q \end{cases}$$

5.5.1. Aufteilung der Loop-Transitionen

Da UPPAAL, wie in Abschnitt 2.2 beschrieben, nur Konjunktionen von Uhrenbedingungen zulässt, müssen die Guards auf entsprechende Form gebracht werden. Dies ist für die Guards $\theta_t^{s,g,q}$ bereits der Fall, da nur in Δ_t^q Uhrenbedingungen (in zulässiger Form) enthalten sind. Beim Loop-Guard L_t ist dies allerdings im Allgemeinen nicht der Fall, da L_t eine Konjunktion von negierten $\theta_t^{s,g,q}$ ist, wodurch die Uhrenbedingungen in L_t in einer Konjunktion von Disjunktionen stehen. Daher müssen die Uhrenbedingungen in L_t auf disjunktive Form gebracht werden, um die Disjunktion auf mehrere Transitionen mit zulässigen Guards verteilen zu können.

Definition 5.5. Für $t \succ r$ sei der Knoten $\mu_t := \min(\{w_t^q \mid (s, g, a, q) \in E_t\} \cup \{t\})$ und für $s \succeq r$ die Menge $E_t^s := \{(u, g, a, q) \in E_t \mid w_t^q = s\}$ definiert. ■

μ_t ist die kleinste (oberste) aller größten gemeinsame Wurzel von t und dem Ziel einer in oder über t startenden Transition. E_t^s ist die Menge aller Transitionen die in oder über t starten und dessen Ziel und t die gleiche größte gemeinsame Wurzel s haben.

Bemerkung 5.6. Wegen $w_t^q \in [\mu_t, t]$ für alle $(s, g, a, q) \in E_t$, ist $(E_t^s)_{s\in[\mu_t,t]}$ eine Partition von E_t.

Die Guards der Transitionen, in die der Guard L_t der Loop-Transition aufgespalten wird, haben folgende Form, was im Anschluss gezeigt wird.

Definition 5.7. Für $t \in T$ und $u \in [\mu_t, t]$ ist χ_t^u definiert als

$$\chi_t^u := \bigwedge_{s \preceq t} (\neg H_{s-1} \vee \bigwedge_{v \in [u,t]} \bigwedge_{(s,g,a,q) \in E_t^v} \neg g)$$

und der Guard λ_t^u ist definiert als

$$\lambda_t^u := \begin{cases} H_t & \text{falls } u = \mu_t \\ \chi_t^u \wedge \neg d_u & \text{falls } u \in (\mu_t, t] \end{cases}$$

■

Um zu zeigen, dass die Aufspaltung der Loop-Transition mit dem Guard L_t, in mehrere Transitionen mit den Guards λ_t^u äquivalent ist, muss gezeigt werden, dass L_t äquivalent zur Disjunktion der λ_t^u ist. Hierfür benötigen man das folgende technische Lemma.

Lemma 5.8. *Seien $(\gamma_i)_{i \in \mathbb{N}}$ und $(\delta_i)_{i \in \mathbb{N}}$ Ausdrücke. Für $n \in \mathbb{N}$ gilt*

$$\bigwedge_{i \in [0,n]} (\gamma_i \vee \bigvee_{j \in (i,n]} \delta_j) \iff \bigvee_{i \in [0,n]} \psi_n^i$$

wobei

$$\psi_n^i := \begin{cases} \bigwedge_{j \in [0,n]} \gamma_j & i = 0 \\ \delta_i \wedge \bigwedge_{j \in [i,n]} \gamma_j & i \in (0, n] \end{cases}$$

Beweis. Der Beweis erfolgt per Induktion über n, wobei die Definition $\varphi_n^i := \gamma_i \vee \bigvee_{j \in (i,n]} \delta_j$ für $n \in \mathbb{N}$ und $i \in [0, n]$ als Abkürzung dient. Die Induktionsvoraussetzung ist $\bigwedge_{i \in [0,n-1]} \varphi_{n-1}^i \iff \bigvee_{i \in [0,n-1]} \psi_{n-1}^i$. Für $n = 0$ ist $\varphi_0^0 = \gamma_0 = \psi_0^0$. Sei nun $n > 0$, dann folgt aus der Definition $\varphi_n^i = \varphi_{n-1}^i \vee \delta_n$ und $\psi_n^i = \psi_{n-1}^i \wedge \gamma_n$ für $i \in [0, n-1]$. Weiter ist $\varphi_n^n = \gamma_n$ und $\psi_n^n = \delta_n \wedge \gamma_n$. Es gilt also nach Induktionsvoraussetzung

$$\begin{aligned} \bigwedge_{i \in [0,n]} \varphi_n^i &\iff \gamma_n \wedge \bigwedge_{i \in [0,n-1]} (\varphi_{n-1}^i \vee \delta_n) \iff \gamma_n \wedge (\delta_n \vee \bigwedge_{i \in [0,n-1]} \varphi_{n-1}^i) \\ &\iff \gamma_n \wedge (\delta_n \vee \bigvee_{i \in [0,n-1]} \psi_{n-1}^i) \iff (\delta_n \wedge \gamma_n) \vee \bigvee_{i \in [0,n-1]} (\psi_{n-1}^i \wedge \gamma_n) \\ &\iff \bigwedge_{i \in [0,n]} \psi_n^i \end{aligned}$$

□

Wenn man dieses Lemma auf L_t und die λ_t^u anwendet, erhält man das gewünschte Ergebnis, wie das folgende Lemma zeigt.

Lemma 5.9. *Für $t \in T$ gilt*

$$L_t \iff \bigvee_{u \in [\mu_t, t]} \lambda_t^u$$

Beweis. Nach Definition von L_t ist

$$\begin{aligned} L_t &\iff \bigwedge_{(s,g,a,q) \in E_t} (\neg g \vee \neg H_{s-1} \vee \neg \Delta_t^q) \\ &\overset{(5.6)}{\iff} \bigwedge_{u \in [\mu_t, t]} \bigwedge_{(s,g,a,q) \in E_t^u} (\neg g \vee \neg H_{s-1} \vee \neg \Delta_t^q) \end{aligned}$$

Da $w_t^q = u$ für alle $(s, g, a, q) \in E_t^u$, ist $\neg \Delta_t^q = \bigvee_{v \in (u,t]} \neg d_v$ für $(s, g, a, q) \in E_t^u$. Man kann also Δ_t^q ausklammern und erhält

$$L_t \iff \bigwedge_{u \in [\mu_t, t]} \Big(\bigwedge_{(s,g,a,q) \in E_t^u} (\neg g \vee \neg H_{s-1}) \vee \bigvee_{v \in (u,t]} \neg d_v \Big)$$

Mit $\gamma_u := \bigwedge_{(s,g,a,q) \in E_t^u} (\neg g \vee \neg H_{s-1})$ und $\delta_u := \neg d_u$ für $u \in [\mu_t, t]$ gilt daher nach Lemma 5.8

$$L_t \iff \bigwedge_{u \in [\mu_t, t]} (\gamma_u \vee \bigvee_{v \in (u,t]} \delta_v) \iff \bigvee_{u \in [\mu_t, t]} \psi_t^u$$

wobei

$$\psi_t^u = \begin{cases} \bigwedge_{v \in [\mu_t, t]} \bigwedge_{(s,g,a,q) \in E_t^v} (\neg g \vee \neg H_{s-1}) & u = \mu_t \\ \neg d_u \wedge \bigwedge_{v \in [u, t]} \bigwedge_{(s,g,a,q) \in E_t^v} (\neg g \vee \neg H_{s-1}) & u \in (\mu_t, t] \end{cases}$$

Nun ist

$$\begin{aligned} &\bigwedge_{v \in [u,t]} \bigwedge_{(s,g,a,q) \in E_t^v} (\neg g \vee \neg H_{s-1}) \iff \bigwedge_{v \in [u,t]} \bigwedge_{s \preceq t} \bigwedge_{(s,g,a,q) \in E_t^v} (\neg g \vee \neg H_{s-1}) \\ &\iff \bigwedge_{s \preceq t} \bigwedge_{v \in [u,t]} \bigwedge_{(s,g,a,q) \in E_t^v} (\neg g \vee \neg H_{s-1}) \iff \bigwedge_{s \preceq t} (\neg H_{s-1} \vee \bigwedge_{v \in [u,t]} \bigwedge_{(s,g,a,q) \in E_t^v} \neg g) \\ &\iff \chi_t^u \end{aligned}$$

Es ist also $\psi_t^u \iff \lambda_t^u$ für $u \in (\mu_t, t]$. Weiter ist

$$\begin{aligned} \psi_t^{\mu_t} &\iff \bigwedge_{v \in [\mu_t, t]} \bigwedge_{(s,g,a,q) \in G_t^v} (\neg g \vee \neg H_{s-1}) \overset{(5.6)}{\iff} \bigwedge_{s \preceq t} \bigwedge_{(s,g,a,q) \in \delta} (\neg g \vee \neg H_{s-1}) \\ &\iff \bigwedge_{s \preceq t} (\neg H_{s-1} \vee \bigwedge_{(s,g,a,q) \in \delta} \neg g) \overset{(5.3)}{\iff} \bigwedge_{s \preceq t} (\neg H_{s-1} \vee H_s) \end{aligned}$$

Falls $\psi_t^{\mu_t}$ falsch ist, gibt es also ein $s \preceq t$ für das H_s falsch ist. Wegen Bemerkung 5.3 muss also auch H_t falsch sein.

Falls $\psi_t^{\mu_t}$ wahr ist, gilt $H_{s-1} \implies H_s$ für alle $s \preceq t$. Da $H_r \iff \top$ ist, sind also alle H_s für $s \preceq t$ wahr. Also auch H_t.
Daraus folgt $\psi_t^{\mu_t} \iff H_t \iff \lambda_t^{\mu_t}$ und somit $L_t \iff \bigvee_{u\in[\mu_t,t]} \lambda_t^u$. □

Die Loop-Transition mit dem Guard L_t kann also in mehrere Transitionen mit den Guards λ_t^u aufgeteilt werden. Die Transitionen des MSPS-Automaten, dessen Namen mit A bezeichnet wird, können also wie folgt nach UPPAAL transformiert werden. Für jeden Zustand $q \in Q$ werden für jede seiner Transitionen $(s, g, a, q') \in E_q$ die Guards $\theta_q^{s,g,q'}$ sowie die Loop-Guards λ_q^u für $u \in [\mu_q, q]$ berechnet. Dann wird für jede Transition $(s, g, a, q') \in E_q$ der interne Zustandsübergang

```
q -> q
  { guard (A_IntState == 0) && θ^{s,g,q'};
    sync channel_PLC_A;
    assign U(a), A_IntState = n_{q'};
  }
```

und der externe Zustandsübergang

```
q -> q'
  { guard A_IntState == n_{q'};
    sync channel_PLC_A;
    assign A_IntState = 0, x = 0, A_ci_0 = 0,..., A_ci_{ℓ-1} = 0;
  }
```

erzeugt, dabei sei $i_j := h(q'-j)$ für $0 \le j < \ell := h(q')-h(w_q^{q'})$ die Hierarchieebene des Knoten $q'-j$ in T. Damit werden die Uhren der zugehörigen Timervariablen zurückgesetzt, deren Aktivierungsregion durch die Transition von q nach q' betreten werden.

Desweiteren werden für jedes $u \in [\mu_q, q]$ die internen Loop-Transitionen

```
q -> q
  { guard A_IntState == 0 && λ_q^u;
    sync channel_PLC_A;
    assign A_IntState = n_q;
  }
```

sowie die externe Loop-Transition

```
q -> q
  { guard A_IntState == n_q;
    sync channel_PLC_A;
    assign A_IntState = 0;
  }
```

erzeugt.

Bemerkung 5.10. Falls die Verzögerungszeit $S_t(u) = 0$ ist, so ist die Uhrenbedingung $y_{t_u} \geq S_t(u) \iff \top$ und somit $d_u \iff \top$. Falls die Verzögerungsbedingung $S_e(u) \iff \bot$ ist, so ist ebenfalls $d_u \iff \top$. In beiden Fällen kann also der Verzögerungsausdruck d_u, welcher im Normalfall eine Uhrenbedingung enthält, im Loop-Guard λ_q^u weggelassen werden. Somit enthält λ_q^u keine Uhrenbedingung mehr und muss nicht als eigene Transition realisiert werden, sondern kann disjunktiv zu einem der anderen Loop-Guards hinzugefügt werden. Man muss also nur so viele Loop-Guards realisieren wie es Verzögerungsausdrücke d_u gibt, die eine Uhrenbedingung enthalten. Da $\lambda_q^{\mu_q}$ keine Uhrenbedingung enthält, werden also im schlimmsten Fall $|(\mu_q, q]|$ Loop-Transitionen benötigt und daher höchstens $h(q) + 1$.

6. Ergebnisse

Die hier vorgestellte Transformation wurde in Java implementiert. Das entstandene Programm plc2ta wird im Anhang A beschrieben. Die Übersetzung von plc2ta unterscheidet sich von der von Moby/RT erstellten in der Übersetzung des Guards einer Transition. Mit Moby/RT wird für jede den Guard erfüllende Belegung eine (interne) Transition erstellt, wohingegen hier der Guard, so wie er ist, direkt übernommen wird. So lassen sich einige Transition im Vergleich zur Übersetzung von Moby/RT einsparen (mit Ausnahme des Spezialfalls wo garkeine erfüllende Belegung für den Guard existiert). Da aber in der Uppaal-Syntax keine Disjunktionen von Uhrenbedingungen erlaubt sind, müssen unter Umständen auch hier einzelne Transition in mehrere aufgespalten werden. Dies passiert, falls ein Zustand mehrere in der Hierarchie darüber liegende Superzustände besitzt, deren Verzögerungszeit größer Null ist und die durch eine Transition verlassen werden können. In diesem Fall muss die zugehörige Loop-Transition, wie in Abschnitt 5.5.1 beschrieben, aufgespalten werden. Somit wird für jeden Superzustand mit einer Verzögerungszeit größer Null eine eigene Loop-Transition benötigt. Es werden also höchstens so viele Loop-Transitionen erzeugt, wie die Tiefe des Zustandes im Hierarchiebaum ist (siehe Bemerkung 5.10). Im Folgenden werden nun die wichtigsten Gemeinsamkeiten und Unterschiede der beiden Übersetzungen empirisch untersucht.

6.1. Äquivalente Modelle

Die hier vorgestellte Übersetzung und die von Moby/RT erzeugen äquivalente Automaten. Das sollten sie auch, denn schließlich unterscheiden sie sich nur in der Art der Übersetzung. Um das empirisch zu belegen kann man für einen gegeben SPS-Automaten folgendes Experiment anstellen. Zuerst werden die beiden Übersetzungen des SPS-Automaten durch Parallelkomposition zu einem Uppaal-System zusammengefügt. Dann werden die PLC-Prozesse synchronisiert. Da beide Automaten die gleichen Eingaben lesen sollen, ist es nicht notwendig, dass jede Übersetzung seine eigenen Driver-Prozesse hat. Stattdessen gibt es gemeinsame Drive-Prozesse, die den Drive-Prozessen der einzelnen Übersetzungen entsprechen. Auf diese Weise ist sichergestellt, dass beide Übersetzungen immer die gleichen Werte lesen. Desweiteren besteht das System aus einem zusätzlichem „Überwachungsprozess“. Der Überwachungsprozess überwacht die Ausgaben der jeweili-

gen Prozesse der zwei Übersetzungen und wechselt in einen Fehlerzustand, sobald sich die Ausgaben zweier korrespondierenden Prozesse unterscheidet. Ist nun der Fehlerzustand des Überwachungsprozesses nicht erreichbar, dann sind die beiden Übersetzungen äquivalent. Empirisch wurde die Äquivalenz der beiden Übersetzungen dann an einigen Beispielen aus dem MOBY/RT-Toolkit belegt.

6.2. Moby/RT versus plc2ta

In diesem Abschnitt sollen jetzt die Unterschiede, die sich aus den Übersetzungen ergeben präsentiert werden. Hierzu wurden eine Reihe von SPS-Automaten von MOBY/RT und von plc2ta übersetzt und dann von UPPAAL geprüft. Die unterschiedlichen Übersetzungen wirken sich auch unterschiedlich auf die benötigte Zeit für die Modellprüfung aus. Alle Automaten wurden auch mit einer x-Uhr Abstraktion transformiert. Bei der x-Uhr Abstraktion werden einfach alle x-Uhren weggelassen.

Bevor nun die UPPAAL-Ergebnisse präsentiert werden, sollen die verwendeten Beispiele kurz beschrieben werden.

Mutex. Das Fallbeispiel „Mutex“ ist eine Modellierung eines Echtzeitprotokols, mit dem der gleichzeitige Zugriff auf eine Resource verhindert wird. Das Protokoll ist als verteiltes System implementiert, bei dem die einzelnen Automaten asynchron über gemeinsame Variablen kommunizieren. Von diesem Fallbeispiel gibt es drei verschiedene Versionen. Bei der ersten Variante M_1 sind alle Automaten auf einer SPS untergebracht, bei M_2 wurde eine Verzögerungszeit so abgeändert, dass ein Fehlerzustand erreichbar wurde. Bei der letzten Variante M_3 wurde jeder Automat auf eine eigene SPS gesetzt. UPPAAL soll hier analysieren, ob es möglich ist, dass gleichzeitig von zwei Prozessen auf die Resource zugegriffen wird.

Single-tracked Line Segment. Das Fallbeispiel „Single-tracked Line Segment“ (SLS) modelliert eine verteilte Straßenbahnsteuerung. Ein Teilstück eines Straßenbahnnetzes kann in beide Richtungen befahren werden. Eine Kontrolleinheit, die als verteiltes System implementiert ist, soll sicherstellen, dass nie zwei Straßenbahnen, die in verschiedene Richtungen fahren, zur gleichen Zeit auf dem kritischen Teilstück sind. Hier soll UPPAAL untersuchen, ob es einen erreichbaren Zustand gibt, in dem das kritische Teilstück in beide Richtungen befahren wird.

Komplex. Das letzte Fallbeispiel „Komplex“ wurde eigens dafür konstruiert, um die Vorteile zu verdeutlichen, die dadurch entstehen, dass der Guard direkt übernommen werden kann. Der Automat enthält nämlich Variablen

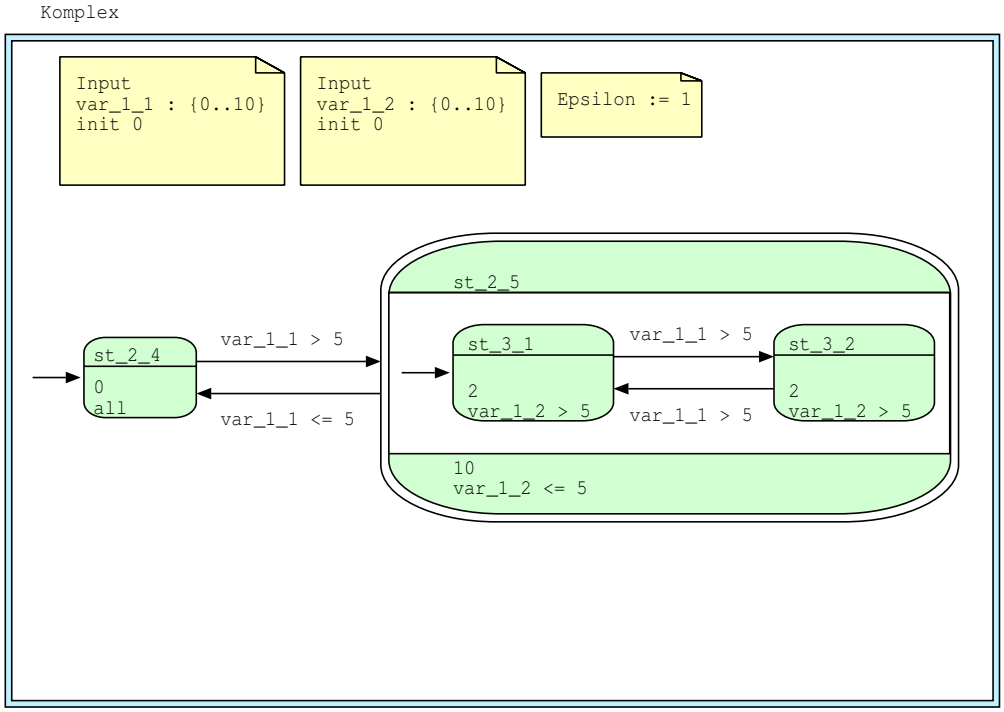

Abbildung 6.1.: Ungünstiger Automat für die Übersetzung von MOBY/RT

mit einem großen Wertebereich. Er ist in Abbildung 6.1 zu sehen. Bei diesem Automat war die x-Uhr Abstraktion mit MOBY/RT nicht möglich, da dieser mit einem Speicherzugriffsfehler abbricht. Deshalb wurden hier die x-Uhren nachträglich „per Hand“ eingefügt. Von diesem Fallbeispiel gibt es drei verschiedene Versionen K_i mit $i \in \{1, 5, 10\}$. Der Index gibt die obere Schranke für den Wertebereich der beiden Eingabevariablen `var_1_1` und `var_1_2` an. An UPPAAL wurden eine erfüllbare und eine unerfüllbare Erreichbarkeitsanfrage gestellt. Die Unerfüllbare ist mit einem hochgestellten f gekennzeichnet. Außerdem enthalten die Automaten Ko zusätzlich noch Ausgabevariablen mit dem gleichen Wertebereich wie die Eingabvariablen.

Alle aufgeführten Experimente wurden auf einem Intel Xeon mit 3 GHz ausgeführt.

Wie in den Tabellen 6.1 und 6.2 zu sehen ist, ist der Speicherbedarf von MOBY/RT und plc2ta fast gleich. Das liegt wahrscheinlich daran, dass die Anzahl der Zustände in beiden Übersetzungen fast gleich bleibt.

Es ist auch deutlich zu sehen, dass bei MOBY/RT die Ausgabevariablen die Verifizierung deutlich verlangsamen, wohingegen die Ausgabevariablen bei plc2ta keinen Einfluss darauf haben.

	Zeit (s)		Speicher (KB)	
	plc2ta	Moby	plc2ta	Moby
M_1	0,66	0,65	37 280	37 392
M_2	0,56	0,56	37 276	37 392
M_3	21,91	20,41	40 320	40 316
SLS	280,32	300,73	524 740	525 152
K_1	0,00	0,00	1 344	1 344
K_5	0,03	0,06	1 344	1 344
K_7	0,11	0,14	37 268	38 224
K_9	0,29	0,35	37 796	39 288
K_{11}	0,68	0,74	38 984	41 128
K_1^f	0,14	0,14	36 860	36 872
K_5^f	24,14	27,78	39 828	40 320
K_7^f	103,60	119,03	46 536	47 472
K_9^f	287,01	377,02	60 640	61 580
K_{11}^f	717,68	978,90	86 448	87 592
Ko_1^f	0,14	0,14	36 864	36 880
Ko_5^f	24,47	31,36	39 960	41 048
Ko_7^f	97,02	140,78	47 160	49 000
Ko_9^f	285,45	450,58	62 216	64 548
Ko_{11}^f	714,00	1211,70	89 856	93 364

Tabelle 6.1.: Modelle ohne x-Uhren

Die Zeitersparnis von plc2ta ist im Vergleich zu Moby/RT bei einfachen Anfragen nur gering, wohingegen eine deutliche Beschleunigung bei schwierigen Anfragen erreicht wird.

	Zeit (s)		Speicher (KB)	
	plc2ta	Moby	plc2ta	Moby
M_1	0,96	0,99	37 280	37 392
M_2	0,78	0,82	37 280	37 400
M_3	41,90	43,51	40 716	40 860
SLS	334,21	345,01	524 768	525 188
K_1	0,00	0,01	1 348	1 344
K_5	0,04	0,06	1 344	1 348
K_7	0,14	0,18	37 272	38 332
K_9	0,38	0,40	37 804	39 292
K_{11}	0,84	0,88	38 992	41 328
K_1^f	0,17	0,17	36 880	36 872
K_5^f	30,31	33,60	39 904	40 324
K_7^f	121,89	142,96	46 628	47 476
K_9^f	367,66	458,28	60 648	61 592
K_{11}^f	911,80	1182,91	86 456	87 600
Ko_1^f	0,17	0,18	36 880	36 884
Ko_5^f	30,63	38,47	40 044	41 056
Ko_7^f	123,39	159,99	47 164	49 012
Ko_9^f	368,51	519,96	62 228	64 560
Ko_{11}^f	901,87	1390,71	89 980	93 372

Tabelle 6.2.: Modelle mit allen x-Uhren

7. Zusammenfassung

Die hier vorgestellte syntaktische Transformation bietet gegenüber der semantischen von MOBY/RT einige Vorteile, aber auch Nachteile. Bei einfachen Automaten, die wenige Variablen mit einem kleinen Wertebereich besitzen und somit wenige Belegungen haben, bietet diese Übersetzung keinen großen Vorteil gegenüber der von MOBY/RT. Es kann sogar dazu kommen, dass zwar weniger Transitionen generiert werden, die Verifikation mit UPPAAL aber dennoch langsamer wird, wie es in Tabelle 6.1 am Beispielautomaten M_3 zu sehen ist.

Weil diese Transformation meistens weniger Transitionen erzeugt als die von MOBY/RT, benötigt die Verifikation mit UPPAAL weniger Zeit, da nicht so viele Guards betrachtet werden müssen, um festzustellen welche Transition genommen werden kann.

Da die hier vorgestellte Transformation auf syntaktischer Ebene arbeitet, können unerfüllbare Guards nicht als solche erkannt werden (außer bei einfachen Fällen wie $\bot$ oder $a \wedge \bot$ etc.). Dies kann im Vergleich zur Transformation von MOBY/RT, welche auf semantischer Ebene arbeitet, zu mehr Transitionen führen, da Transitionen, deren Guard nicht erfüllbar ist, nicht weggelassen werden.

Würden in der UPPAAL-Syntax auch Disjunktionen zwischen den Uhrenbedingungen erlaubt sein, so könnte jede Transition im SPS-Automaten in genau eine (interne und eine externe) Transition im Realzeitautomat übersetzt werden.

Eine weitere Möglichkeit die Modellprüfung zu beschleunigen, wäre nicht erreichbare Zustände bei der Übersetzung erst gar nicht in den Realzeitautomaten mit aufzunehmen. Man könnte auch Anstelle den Guard so zu übernehmen wie er ist, die Menge der ihn erfüllenden Belegungen berechnen und diese Menge in Intervalle aufteilen. Diese Intervalle können dann mit der logischen Verknüpfung `and` verknüpft werden und dienen so als Guard im Realzeitautomat. Wenn die Anzahl dieser Intervalle von Belegungen die den Guard erfüllen größer ist als die Zahl m, dann wird doch der ursprünglich Guard als Guard im Realzeitautomat übernommen. m könnte man abhängig machen von der Anzahl der im Guard vorkommenden Variablen und deren Wertebereich. Somit könnt der Fall, das eine Transition die einen ziemlich komplizierten Guard hat, der aber nur wenige ihn erfüllende Intervalle von Belegungen hat besser gelöst werden und die Anzahl der erzeugten Transitionen wäre dann immer höchstens so groß wie die Anzahl der von MOBY/RT erzeugten Übersetzung.

A. Das Programm plc2ta

Implementiert wurde die Transformation in der Programmiersprache Java unter Zuhilfenahme von JavaCC und JArgs. Javacc ist ein Werkzeug mit dem die SIM-Datei eingelesen wird. Jargs ist eine Bibliothek zum Parsen von Argumenten der Komandozeile, (command-line argument parsing library) hiermit werden also die Optionen beim Aufruf des Übersetzungsprogramms eingelesen. Eingegeben werden kann ein System von SPS-Automaten im SIM-Format und die Ausgabe ist eine Übersetzung in ein System von Realzeitautomaten in der UPPAAL-Syntax. Als Optionen können folgende Paramter angegeben werden

- `c`
  Ist die Option `-c` nicht gesetzt, so wird der Loop-Guard nicht aufgeteilt, er enthält dann also unter Umständen Uhrenbedingungen die in einer Disjunktion stehen. Dies wird von UPPAAL aber nicht unterstützt.

- `x` „`none`“ | „`one`“ | „`all`“
  Hiermit kann eine x-Uhr Abstraktion eingeschaltet werden: `none` heißt, es wird keine x-Uhr generiert, `one` es wird nur eine x-Uhr für alle PLC-Prozesse generiert und `none` heißt, es wird keine x-Uhr erstellt.

- `s`
  Hiermit kann man eine Vereinfachung der Guard-Ausdrücke ein und ausschalten.

Literaturverzeichnis

[1] Alur, Rajeev, Costas Courcoubetis und David L. Dill: *Model-Checking in Dense Real-time*. Information and Computation, 104(1):2–34, 1993.

[2] Alur, Rajeev und David L. Dill: *Automata For Modeling Real-Time Systems*. In: *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP 1990)*, Band 443 der Reihe *Lecture Notes in Computer Science*, Seiten 322–335. Springer-Verlag, 1990.

[3] Alur, Rajeev und David L. Dill: *A theory of timed automata*. Theoretical Computer Science, 126(2):183–235, 1994.

[4] Behrmann, Gerd, Alexandre David und Kim G. Larsen: *A Tutorial on* Uppaal. In: Bernardo, Marco und Flavio Corradini (Herausgeber): *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, Nummer 3185 in *Lecture Notes in Computer Science*, Seiten 200–236. Springer-Verlag, 2004.

[5] Bender, Klaus und Frank Schiller: *Vorlesungsskript Automatisierungstechnik*, 2006. Technischen Universität München.

[6] Chaochen, Zhou, C. A. R. Hoare und Anders P. Ravn: *A Calculus of Durations*. Information Processing Letters, 40(5):269–276, 1991.

[7] Dierks, Henning: *Time, Abstraction and Heuristics – Automatic Verification and Planning of Timed Systems using Abstraction and Heuristics*. Habilitationsschrift, Universität Oldenburg, 2006.

[8] Dierks, Henning: *Specification and Verification of Polling Real-Time Systems*. Doktorarbeit, Universität Oldenburg, 1999.

[9] Dierks, Henning, Hans Fleischhack und Josef Tapken: *The* Moby/plc *Tutorial*, 2001. Universität Oldenburg.

[10] Emerson, E. Allen: *Temporal and Modal Logic*. In: Leeuwen, J. van (Herausgeber): *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, Seiten 995–1072. Elsevier, Amsterdam, 1990.

[11] Hansen, Michael R. und Zhou Chaochen: *Duration Calculus: Logical Foundations*. Formal Aspects of Computing, 9(3):283–330, 1997.

[12] Henzinger, Thomas A., Xavier Nicollin, Joseph Sifakis und Sergio Yovine: *Symbolic model checking for real-time systems*. Information and Computation, 111(2):193–244, 1994.

[13] Krieg-Brückner, Bernd, Jan Peleska, Ernst-Rüdiger Olderog und Alexander Baer: *The UniForM Workbench, a Universal Development Environment for Formal Methods*. In: Wing, Jeannette M., Jim Woodcock und Jim Davies (Herausgeber): *FM'99 – Formal Methods*, Band 1709 der Reihe *Lecture Notes in Computer Science*, Seiten 1186–1205. Springer-Verlag, 1999.

[14] Larsen, Kim G., Paul Pettersson und Wang Yi: *Uppaal in a Nutshell*. Journal on Software Tools for Technology Transfer, 1(1-2):134–152, 1997.

[15] Milner, Robin: *A Calculus of Communicating Systems*, Band 92 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[16] Yovine, Sergio: *Kronos: A Verification Tool for Real-time Systems*. Journal on Software Tools for Technology Transfer, Seiten 123–133, 1997.