Verifikation des dynamischen Verhaltens beim Entwurf von Feldbuskomponenten hinsichtlich ihrer Passfähigkeit

Eine Studien-Arbeit im Rahmen des Großen Beleges

Martin Pitt
martin@piware.de
Matrikel-Nummer 2 69 44 57

Betreuer:
Dipl.-Inf. Gunnar Stein

Verantwortlicher Hochschullehrer:
Prof. Dr.-Ing. habil. Klaus Kabitzsch

Technische Universität Dresden

18. November 2003

Inhaltsverzeichnis

1 Einführung **5**
1.1 Ziel dieser Arbeit . . . 5
1.2 Beschreibungsmodell von CCM . . . 6
1.3 Aufbau dieser Arbeit . . . 7
1.4 Mathematische Notation . . . 8

2 Anwendungsfälle **9**
2.1 Ungeeignete Komponente . . . 9
2.2 Echtzeitanalyse . . . 10

3 Beschreibung der Stubs **11**
3.1 Schnittstelle . . . 11
3.1.1 Kanäle und Parameter . . . 11
3.1.2 Datentypen . . . 11
3.1.3 Notation . . . 12
3.2 Verhalten . . . 12
3.2.1 Modellierung . . . 12
3.2.2 Definition der Transitionen . . . 14
3.2.3 Definition der Prozessalgebra . . . 15
3.2.4 Semantik . . . 16
3.3 Notation einer Stub-Beschreibung . . . 18
3.4 CCMB-Syntax . . . 18
3.5 Beispiele . . . 18

4 Verbindung von Stubs zu Netzwerken **21**
4.1 Modellierung . . . 22
4.2 Definition von Bindungen . . . 23
4.3 Freie und gebundene Kanäle . . . 23
4.4 Semantik . . . 24
4.5 Beschreibung und Kompositionen . . . 27
4.6 Initialisierung . . . 27

5 Verifikation von Anforderungen **28**
5.1 Modaler μ-Kalkül . . . 28
5.2 Semantik . . . 30
5.3 Beschreibung andauernder Eigenschaften durch Fixpunkte . . . 31
5.3.1 Beispiel: Uhren . . . 31
5.3.2 Kleinste und größte Fixpunkte . . . 32

5.4 Entfaltung . . . 33
5.5 Formulierung von Eigenschaften . . . 35
5.5.1 Sicherheits- und Lebendigkeitseigenschaften . . . 35
5.5.2 Until-Eigenschaften . . . 37
5.5.3 Echtzeiteigenschaften . . . 37
5.6 Negation . . . 37
5.6.1 Voraussetzungen . . . 37
5.6.2 Normalform . . . 38
5.6.3 Beispiel . . . 38
5.6.4 Weitere Operatoren . . . 39
5.7 Deadlock . . . 39
5.8 Präzedenzregeln . . . 39
5.9 Verifikationskalkül . . . 40
5.10 Intuition des Tableauverfahrens . . . 42

6 Demonstration an den Anwendungsfällen 43
6.1 Echtzeitanalyse . . . 43
6.1.1 Stubs . . . 43
6.1.2 Netzwerk . . . 45
6.1.3 Anforderungen . . . 45
6.1.4 Verifikation . . . 45
6.2 Ungeeignete Komponente . . . 47
6.2.1 Stubs . . . 47
6.2.2 Netzwerk . . . 48
6.2.3 Anforderungen . . . 49
6.2.4 Verifikation . . . 50

7 Implementation 53
7.1 Prozess- und Netzwerkdefinition . . . 54
7.2 Prozess-Semantik . . . 55
7.3 Tableau-Beweiser . . . 58
7.4 Beispiele . . . 60
7.4.1 Uhren . . . 60
7.4.2 Anwendungsfall Temperaturkontrolle . . . 61
7.4.3 Anwendungsfall Hausbeleuchtung . . . 62
7.5 Laufzeitverhalten . . . 63

8 Zusammenfassung und Ausblick 65

Abbildungsverzeichnis

2.1 Anwendungsfall: Beleuchtung mit mehreren Schaltern/Tastern 10
2.2 Anwendungsfall: Temperaturalarm 10

3.1 Syntax der Schnittstellenbeschreibung von Stubs 12
3.2 Syntax der Transitionen 15
3.3 Syntax der Prozessalgebra zur Verhaltensbeschreibung von Stubs 16
3.4 Präzedenz und Assoziativität der Prozessalgebra-Operatoren 16
3.5 Semantik der Prozessausdrücke für Stubs 17
3.6 Semantik der Prozessausdrücke für atomaren Prädikate 18

5.1 Präzedenz und Assoziativität der Operatoren des μ-Kalküls 39
5.2 Tableau-Regeln für das μ-Kalkül 41

6.1 Prozessbeschreibung und enstprechendes Zustandsdiagramm für `LAMP` . . . 44
6.2 Prozessbeschreibung und entsprechendes Zustandsdiagramm für `SENSOR` . . 44
6.3 Prozessbeschreibung und entsprechendes Zustandsdiagramm für `SLOW_CTR` . 44
6.4 Prozessbeschreibung und entsprechendes Zustandsdiagramm für `FAST_CTR` . 45
6.5 Prozessbeschreibung und entsprechendes Zustandsdiagramm für `SWITCH` . . 47
6.6 Prozessbeschreibung und entsprechendes Zustandsdiagramm für `TOGGLE_CTR` 48
6.7 Prozessbeschreibung und entsprechendes Zustandsdiagramm für `MAIN_CTR` . 48

7.1 Prolog-Syntax der Transitionen und Prozesse 54
7.2 Prolog-Syntax der μ-Formeln 58
7.3 Anwendungsfall „Temperaturkontrolle“ in ProLog 61
7.4 Anwendungsfall „Hausbeleuchtung“ in ProLog 62

1 Einführung

„Die Mathematik befriedigt den Geist durch ihre außerordentliche Gewissheit“

Johannes Kepler (1571 – 1630), dt. Astronom

1.1 Ziel dieser Arbeit

Beim Zusammensetzen von Systemen aus Einzelkomponenten werden genaue Kenntnisse über die innere Arbeitsweise der Komponenten benötigt, um die notwendigen Bindungen und Einstellungen korrekt vornehmen zu können. Diese genauen Kenntnisse hat aber normalerweise nur der Komponenten-Entwickler, nicht jedoch der Systemintegrator.

Üblicherweise stammen solche Komponenten oft von verschiedenen Herstellern, sind nicht notwendigerweise aufeinander abgestimmt und folgen auch keiner offiziell vorgegebenen Dokumentationsform.

Die Idee zu dieser Arbeit entstand bei der Diskussion mit meinem Betreuer, Dipl.-Inf. Gunnar Stein, über seine Promotionsarbeit. In dieser wird eine formale Sprache und ein System „Component Composition Model“ (CCM) definiert, die die formale Beschreibung der Schnittstelle von Feldbuskomponenten erlaubt. Solch eine formale Beschreibung einer Komponente bietet folgende Vorteile:

- Missverständnisse in Prosa-Dokumentation werden durch eine klar definierte Syntax und Semantik der formalen Sprache vermieden.
- Eine formale Beschreibung erzwingt Vollständigkeit und Detailgenauigkeit.
- Der Systemintegrator braucht keine genauen Kenntnisse über die interne Implementation einer Komponente, da dieses Wissen schon dort abstrahiert werden kann, wo es vorhanden ist, nämlich beim Hersteller.
- Eine formale Schnittstellenbeschreibung stellt eine Vereinbarung dar; an ihr kann frei jeglicher zweideutiger Interpretation entschieden werden, ob ein Fehler in der Komponente oder in der Zusammenschaltung liegt.

Die momentane Version von CCM definiert die statischen Aspekte einer Schnittstelle: die Datentypen und Wertebereiche der Ein- und Ausgänge und der inneren Parameter. Bei der Diskussion darüber wurde klar, dass es wichtige Fehlertypen gibt, die sich nicht schon in einer inkompatiblen Schnittstelle, sondern erst zur Laufzeit unter bestimmten Voraussetzungen ergeben.

Das Ziel dieser Arbeit ist, CCM um eine Beschreibung des *Verhaltens* einer Schnittstelle und der Anforderungen an das Gesamtsystem zu erweitern, und eine automatische Verifikation der Anforderungen bezüglich der Spezifikation der Komponenten zu erlauben. Der neue Teil zur Beschreibung von Verhalten und Anforderungen heißt „Component Composition Model for Behavior“ (CCMB).

Dabei kann man nicht erwarten, eine vollständige funktionale Verifikation zu erhalten, d. h., man kann sich nicht darauf verlassen, dass das reale System korrekt funktioniert, wenn die Verifikation erfolgreich ist. Da die formale Beschreibung eine starke Abstraktion des realen Systems darstellt und viele wichtige Aspekte nicht berücksichtigt, können sich Aussagen der Verifikation natürlich nur auf dieser Modellebene bewegen.

Die Nützlichkeit der Verifikation besteht deshalb insbesondere darin, festzustellen, ob ein System **nicht** funktioniert. Durch ein automatisiertes Finden von Gegenbeispielen und Analyse des Beweises kann die Fehlerbehebung wesentlich vereinfacht werden.

Schließlich soll diese Arbeit auch untersuchen und aufzeigen, bis zu welcher Größenordnung eine formale Verifikation überhaupt praktisch durchführbar ist.

1.2 Beschreibungsmodell von CCM

CCM führt einige Begriffe ein, die das Beschreibungsmodell bilden. Diese werden hier kurz erläutert und in dieser Arbeit verwendet.

Device: Ein Device ist eine konkrete Instanz einer Feldbuskomponente. Es hat ein inneres, herstellerspezifisches Verhalten, was hier nicht betrachtet wird, denn das Ziel von CCM ist ja gerade die Abstraktion dieses Verhaltens.

Die Schnittstelle und das Verhalten eines Devices werden über →Stubs exportiert.

Stubs: Ein Stub kapselt einen funktional zusammengehörenden Teil der Schnittstelle und des Verhaltens eines Devices. Es besteht aus inneren (zum Device hin) und äußeren (zum Netzwerk hin) Ein- und Ausgängen und kann außerdem Parameter haben, die sein Verhalten bestimmen. Ein Stub stellt auch eine Beschreibung seines Verhaltens zur Verfügung.

Die von einem Device exportierten Stubs stellen die von außen zugängliche Schnittstelle als formale Abstraktion des Devices dar. Sie sind deshalb die kleinsten zu behandelnden Strukturen. Die Ein- und Ausgänge von Stubs können mit anderen Stubs verbunden werden, um Netzwerke zu bilden.

Stubs sollen den Charakter von Bausteinen haben, indem sie einen funktional zusammengehörenden Schnittstellenteil möglichst allgemein (wieder-)verwendbar kapseln und eine sehr einfache Struktur haben.

Stub Groups: Eine Stub Group ist eine Zusammenschaltung von Stubs, die sich konzeptionell *innerhalb* eines Devices befindet. Sie sind nützlich, um das Verhalten eines komplexeren Schnittstellenteils durch die Aggregation von Standard-Stubs zu bilden.

Network Profiles: Zusammenschaltungen von Stubs über mehrere Devices bilden ein Netzwerk. Ein Network Profile beschreibt dabei *eine mögliche* Zusammenschaltung, d. h. eine konkrete Assoziation von Stub-Eingängen mit Stub-Ausgängen (genannt **Bindung**) zusammen mit der notwendigen Einstellung der Stub-Parameter.

Application: Eine Anwendung ist ein wohldefiniertes Network Profile, d. h., ohne freie Parameter und mit konkret festgelegten Devices.

1.3 Aufbau dieser Arbeit

Um die praktische Relevanz zu demonstrieren und um Beispiele für die Notation und Verifikation zu haben, stellt Kapitel 2 einige Anwendungsfälle dar, die zunächst informal beschrieben werden. Die Anwendungsfälle haben verschiedene Fehlerkategorien und sind klein genug, um die darin enthaltenen Fehler gut nachvollziehbar zu halten.

Kapitel 3 beschreibt die kleinsten betrachteten Strukturen, die Stubs. Es erfolgt die Definition der Syntax der Schnittstellen- und Verhaltensbeschreibung und deren Semantik, d. h., die Interpretation der Sprache auf einem endlichen Automaten.

Da die Beschreibung eines Stubs die erste Formalisierung ist, kann sie natürlich nicht verifiziert werden; sie wird deshalb als axiomatisch für die Verifikation des Gesamtsystems verstanden. Deshalb ist hier auch noch keine Logik zu finden.

Kennt man nun das Verhalten der einzelnen Komponenten, muss noch formal beschrieben werden, wie sie miteinander zu Netzwerken verbunden werden. Dieser Aufgabe widmet sich Kapitel 4, welches die Beschreibung der Bindungen einführt und die Semantik des zusammengeschalteten Gesamtsystems definiert. Letztere dient für die Verifikation des Netzwerkes als Modell, an dem die Eigenschaften zu beweisen sind.

In Kapitel 5 wird das Verifikationssystem diskutiert. Ein solches besteht aus folgenden Teilen:

- **Syntax:** Die Formulierung von Anforderungen, d. h. Eigenschaften, die das Gesamtsystem haben soll, erfolgt in einer Logik.
- **Semantik:** Die Bedeutung der durch die Logik definierten Formeln wird durch deren Semantik bestimmt. Damit die Beschreibung überhaupt Sinn macht, muss die Semantik natürlich auf den selben Bildbereich abbilden wie die Beschreibung der Stubs und Netzwerke, in diesem Falle also auf endliche Automaten.
- **Verifikationskalkül:** Dies ist ein Algorithmus, der die (Un)Gültigkeit von Formeln bezüglich eines Modells direkt auf der syntaktischen Ebene beweisen kann, also ohne den „Umweg“ der expliziten Berechnung der Semantik. Dazu muss natürlich sichergestellt sein, dass das Kalkül korrekt bezüglich der Semantik der Logik ist.
- **Pragmatik:** Es werden häufig benutzte Klassen von Eigenschaften untersucht und erklärt, wie man diese in der verwendeten Logik ausdrücken kann.

Mit der nun eingeführten Beschreibungssprache und dem Verifikationskalkül werden in Kapitel 6 nun die Anwendungsfälle aus Kapitel 2 formalisiert. Dieses Kapitel zeigt auch, wie die Verifikation aussieht und wie sich die Fehler darin äußern.

Schließlich bespricht das letzte Kapitel 7 eine prototypische Implementation des Verifikationskalküles in Prolog. Hier werden auch die Grenzen sowohl der Implementation, aber auch des prinzipiellen Verfahrens sichtbar, so dass man einen Eindruck von der Größe noch verifizierbarer Systeme erhält.

In dieser Arbeit wird nach dem Prinzip vorgegangen, alle neu eingeführten Begriffe zuerst exakt mathematisch zu definieren. Wenn man irgendeinen Teil der Beschreibungen von Modellen, Formeln oder der Verifikation nicht mathematisch formuliert, dann gibt es auch keinen Grund, den Ergebnissen im mathematischen Sinne zu vertrauen, deshalb ist die Verwendung von sehr vielen Formalismen – wenn es auch beim Lesen als hinderlich empfunden werden kann – unabdingbar. Deshalb wird versucht, die Intention der Definitionen in Umgangssprache zu erläutern.

1.4 Mathematische Notation

Die logische Notation folgt der Prädikatenlogik mit den folgenden Symbolen:

$F \wedge G$	F und G sind wahr (Konjunktion)
$F \vee G$	F oder G oder beide sind wahr (Disjunktion)
$F \rightarrow G$	F ist falsch oder G ist wahr (Implikation)
$F \Leftrightarrow G$	F ist wahr genau dann, wenn G wahr ist (Äquivalenz)
$\exists x.\, F(x)$	es existiert (mindestens) ein x, für das $F(x)$ wahr ist (Existenzquantor)
$\forall x.\, F(x)$	für alle x ist $F(x)$ wahr (Allquantor)

Die verwendete mathematische Notation folgt der üblichen Zermelo-Fraenkelschen Mengenlehre, die die Begriffe „Menge“, „Klasse“ und die Relation $\in$ („ist Element von“) axiomatisiert.

Um Missverständnissen vorzubeugenwerden die verwendeten Operationen hier kurz definiert. Dabei seien $A, B, x, y \in \Omega$ Mengen.

$\varnothing$	$=_{def}$	$\{x \mid x \neq x\}$	(leere Menge)
$A \subseteq B$	$\Leftrightarrow$	$x \in A \rightarrow x \in B$	(Teilmenge)
$A \cup B$	$=_{def}$	$\{x \mid x \in A \vee x \in B\}$	(Vereinigung)
$A \cap B$	$=_{def}$	$\{x \mid x \in A \wedge x \in B\}$	(Schnitt)
$A \setminus B$	$=_{def}$	$\{x \mid x \in A \wedge x \notin B\}$	(Differenz)
$A \times B$	$=_{def}$	$\{(x, y) \mid x \in A \wedge y \in B\}$	(kartesisches Produkt)
$\mathcal{P}(A)$	$=_{def}$	$\{x \mid x \subseteq A\}$	(Potenzmenge)
$\mathbb{N}$	$=_{def}$	ω	(Menge der natürlichen Zahlen $\{0, 1, 2, \ldots\}$)
A^n	$=_{def}$	$A \times A \times \ldots \times A$ (n-mal)	(n-faches kartesisches Produkt, Menge der Wörter der Länge n über A)
ε	$=_{def}$	A^0	(0-faches kartesisches Produkt, „leeres Wort“)
A^*	$=_{def}$	$\bigcup_{n \geq 0} A^n$	(Menge aller endlich langen Wörter über A, Listen mit Elementen aus A)

2 Anwendungsfälle

"Ein gutes Beispiel ist der beste Lehrmeister"

Deutsches Sprichwort

Um Beispiele für die Anwendung automatischer Verifikation zu haben, werden in diesem Kapitel zwei einfache Szenarien mit verschiedenen Fehlerklassen vorgestellt. Die auftretenden Fehler sind in diesem Kapitel zunächst verbal beschrieben, da eine formale Sprache zur Beschreibung von Stubs und Anforderungen und das zugehörige Beweiskalkül erst später eingeführt werden.

Die Beispiele sind absichtlich sehr einfach, um die Funktionsweise und auch die Fehler offensichtlich und einfach nachvollziehbar zu halten. Die Praxis zeigt aber erstens, dass reale Anwendungen erstens wesentlich komplexer sind und die hier noch trivialen Verschaltungsfehler dann auch über mehrere Indirektionen auftreten und zweitens oft Komponenten verschiedener Hersteller verwenden. Die formale Beschreibung zeigt dort ihre Stärken, da sie im Gegensatz zu einer Dokumentation in Prosa vollständig und exakt sein muss.

Die Anwendungsfälle werden hier zunächst als Blockschaltbild dargestellt, dabei entspricht ein Stub einem Block. In realen Fällen wird man einem Block sicherlich eher ein Device zuordnen, aber der Einfachheit halber wird hier die Ebene der Devices nicht betrachtet. Die Bezeichnungen an den Stubs folgen dabei der Konvention, dass bei Ausgängen der Name ausserhalb und bei Eingängen der Name innerhalb des Blocks steht.

2.1 Ungeeignete Komponente

Bei Hausbeleuchtungen gibt es oft mehrere Lichtschalter, die ein und dieselbe Lampe $LAMP$ von verschiedenen Orten aus steuern sollen. Jeder Lichtschalter S_i fungiert dabei als Taster, der den aktuellen Zustand der Lampe (an/aus) wechselt. Dabei möchte man aber auch einen zentralen Ein-/Aus-Schalter $MAIN$ (z. B. an der Wohnungstür), der unabhängig von allen anderen Schaltern den Lampenzustand festsetzt:

Die Anforderungen sind:

1. Solange $MAIN$ „aus" ist, geht und bleibt $LAMP$ aus.
2. Wenn $MAIN$ auf „an" gestellt wird, dann geht $LAMP$ an.
3. Wenn $MAIN$ an ist, dann ändert jeder Druck auf die Taster den Lampenzustand.

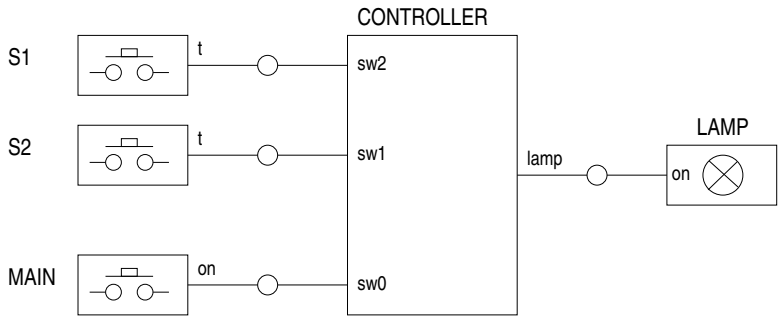


Abb. 2.1: Anwendungsfall: Beleuchtung mit mehreren Schaltern/Tastern

Werden nun die Taster falsch an den Controller angeschlossen, oder ein unpassender Controller verwendet, der einen separaten Ein-/Aus-Schalter nicht unterstützt, so werden diese Anforderungen nicht erfüllt. Das gleiche Problem entsteht, wenn die Schaltermodule einen Parameter besitzen, der regelt, ob das Modul als Taster oder Umschalter fungiert, und die Parameter falsch eingestellt werden (Konfigurationsfehler).

2.2 Echtzeitanalyse

Dieses Szenario modelliert eine Temperaturüberwachung, wie man sie z. B. in Heizungen oder Reaktoren findet. Der $CONTROLLER$ entscheidet anhand der Daten vom Temperatursensor $SENSOR$, ob der aktuelle Zustand kritisch ist und löst bei Bedarf einen Alarm aus. Der Einfachheit halber wird der Alarm wiederum durch eine Lampe angezeigt.

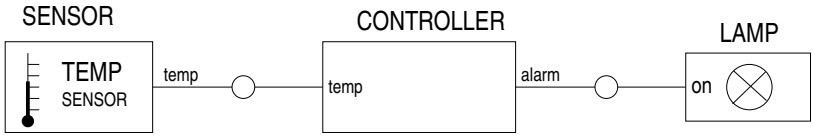


Abb. 2.2: Anwendungsfall: Temperaturalarm

Üblicherweise stellt man an ein solches System die Forderung, dass der Alarm im kritischen Fall spätestens nach einer vorgegebenen Zeitschranke ausgelöst werden muss. Ist nun der Controller zu langsam, so ist diese Forderung nur manchmal oder überhaupt nicht erfüllt.

3 Beschreibung der Stubs

„Eine Definition ist das Erfassen der Wildnis einer Idee mit einem Wall von Worten“

Samuel Butler d. Ä. (1612-1680), englischer Satiriker

3.1 Schnittstelle

3.1.1 Kanäle und Parameter

Die Interaktion und Kommunikation der Stubs mit der Umgebung erfolgt allgemein über Eingabe- (die Menge *Inputs*) und Ausgabekanäle (die Menge *Outputs*). Bei LON-Netzen werden diese Kanäle z. B. in Form von Netzwerkvariablen dargestellt, in IP-Netzen wie dem Internet werden Pakete beliebigen Inhalts versendet, die ebenfalls als Nachrichten an verschiedene Kanäle interpretiert werden können.

Zu beachten ist, dass ein Stub mit zwei verschiedenen Umgebungen kommuniziert: zum einen mit dem Gerät, an dass er gekoppelt ist („device“) und mit anderen Stubs über ein Netzwerk (in der Rolle eines „port“).

Daneben kann ein Stub interne Parameter besitzen (die Menge *Params*), mit denen sein Verhalten zur Laufzeit konfiguriert werden kann.

3.1.2 Datentypen

In konkreten Systemen sind die Kanäle üblicherweise getypt, so dass auch in dieser Arbeit eine typisierte Sprache verwendet wird. An elementaren Datentypen steht dabei zur Verfügung:

- `int` für Ganzzahlen
- `real` für reelle Zahlen
- `bool` für Wahrheitswerte (mit dem Wert `true` oder `false`)
- `enum` für endliche Aufzählungstypen

Zusätzlich definiert CCM noch strukturierte Datentypen (`struct`), die aber ihrerseits auf unterster Ebene aus diesen elementaren Datentypen zusammengesetzt werden, so dass deren Elemente sich für die dynamische Verifikation nur hinsichtlich ihrer Notation unterscheiden (Qualifikation durch `Structname.Elementname`).

3.1.3 Notation

Die Syntax der Beschreibungen der *Inputs*, *Outputs* und *Params* eines Stubs werden hier in mit Hilfe der ISO-EBNF [ISO96] definiert:

$$
\begin{aligned}
&Identifier = (Letter \mid \text{„_“})\ \{Letter \mid \text{„_“} \mid Digit\} \\
&Type = \text{„int“} \mid \text{„real“} \mid \text{„bool“} \mid \text{„enum“}\ EnumValues \\
&EnumValues = \text{„\{“}\ Identifier\ \{\ \text{„,“}\ Identifier\ \}\ \text{„\}“} \\
&Inputs = \left(\ \text{„portin“} \mid \text{„devin“}\ \right)\ Type\ Identifier \\
&Outputs = \left(\ \text{„portout“} \mid \text{„devout“}\ \right)\ Type\ Identifier \\
&Params = \text{„param“}\ Type\ Identifier
\end{aligned}
$$

Abb. 3.1: Syntax der Schnittstellenbeschreibung von Stubs

Die Klasse *Letter* umfaßt dabei alle Groß- und Kleinbuchstaben, die Klasse *Digit* die Ziffern (0 bis 9).

Inputs und *Outputs* berücksichtigen in der Syntax die Unterscheidung der Umgebung, mit der der Stub kommuniziert: mit dem den Stub exportierenden Gerät (`devin` und `devout`) und mit anderen Stubs (`portin` und `portout`).

3.2 Verhalten

3.2.1 Modellierung

Das von außen beobachtbare Verhalten wird folgendermaßen abstrahiert:

1. Stubs kommunizieren ausschließlich über ihre Ein- und Ausgabekanäle. Die Beschreibungssprache muss also zwischen Ein- und Ausgabe unterscheiden können und Variablennamen und -werte unterstützen.
2. Es wird zwischen zwei grundsätzlichen Nachrichtenklassen unterschieden: Der **Kontrollfluss** dient der Steuerung der Komponenten und wird auf endlich viele konkrete Werte beschränkt; dagegen wird von den konkreten Werten im **Datenfluss** abstrahiert und stattdessen eine funktionale Schreibweise verwendet.
3. Das Senden einer Nachricht hat eine Latenz, geht also nicht beliebig schnell; die Dauer einer Nachricht wird als die Zeitspanne „1 tick“ definiert.
4. Ein Stub befindet sich - zumindest aus Sicht seines beobachtbaren Schnittstellenverhaltens - immer in genau einem von drei Zuständen:

- Senden einer Nachricht, d. h. Setzen eines Ausgangs auf einen neuen Wert; im folgenden als **Aktion** bezeichnet
- Warten auf eine Nachricht, d. h. Warten, bis ein Eingang einen bestimmten Wert erhält oder (mit einem beliebigen Wert) aktualisiert wird; im folgenden als **Ereignis** bezeichnet
- interne Operationen, in denen weder Ein- noch Ausgabe vorgenommen wird

Das Verhalten eines Stubs wird durch einen endlichen Automaten (Zustandsübergangsmodell, state machine) beschrieben. Zustandsdiagramme sind in der Ingenieurswelt sehr weit verbreitet und die Arbeit damit ist gut erforscht und zudem sehr intuitiv.

Die damit einhergehende Beschränkung auf nur endlich viele verschiedene Zustände und Kommunikationsabläufe muss sowieso in Kauf genommen werden, da eine vollautomatische Verifikation sonst nicht mehr unter allen Umständen möglich ist. Es stört auch in der praktischen Anwendung nicht: die Intention ist, Stubs sehr einfach und klein zu halten, da die Komplexität sonst nicht mehr beherrschbar und überschaubar ist.

Die Beschreibung der Zustandsübergänge (auch „Transitionen“ genannt) erfolgt in einer textuellen Sprache, die die oben genannte Abstraktion möglichst gut modelliert. Da es bereits einige weit verbreitete Prozessbeschreibungssprachen gibt, soll nach Möglichkeit auch eine schon existierende verwendet werden; deshalb werden die populärsten Vertreter kurz diskutiert:

- CSP (Communicating Sequential Processes) [Hoa85] ist eine der ältesten Sprachen zur Beschreibung von parallelen Prozess-Systemen und bietet eine sehr reichhaltige Syntax. Die Modellierung der Kommunikation bietet gleich zwei unterschiedliche Konzepte: benannte Kommunikationskanäle (gerichtet) und global synchronisierte ungerichtete atomare Aktionen.
- Magee und Kramers FSP (Finite State Processes) [MK99] ist eine kleine Teilmenge von CSP, die sehr einfach und intuitiv zu erlernen ist. Leider unterscheidet FSP nicht zwischen Ein- und Ausgabe und das Modell globaler Synchronisation entspricht nicht üblichen Technologien wie z. B. LonWorks-Netzen.
- Robin Milner's CCS (Calculus of Communication Systems) [Mil89] definiert atomare Ein- und Ausgabekanäle als einzige Form der Kommunikation zwischen zwei Prozessen und scheint deshalb für die Modellierung am besten geeignet zu sein.

Ausgehend davon wurde CCS als Vorbild für CCMB gewählt, das jetzt um zwei Aspekte erweitert wird: anstatt atomarer Transaktionsnamen sollen Gleichungen über den Schnittstellenvariablen und funktionale Variablenwerte verwendet werden und die Kommunikation soll nicht auf einen einzigen Empfänger beschränkt bleiben. Zudem ist das Konzept einer Kanalbindung in Netzwerken zu formalisieren.

3.2.2 Definition der Transitionen

Eine **Transition** ist die Beschreibung eines Ereignisses (Eingabe) bzw. einer Aktion (Ausgabe) auf der Ebene des endlichen Automaten einer Stub-Beschreibung. Die Sprache aller Transitionen wird im folgenden als Δ definiert.

Ausgehend von der in 3.2.1 gewählten Modellierung wird zwischen folgenden Nachrichten unterschieden:

- **Kontrollfluss:** $out := c$ mit $out \in Outputs$ und einem dazu typkompatiblen Wert c beschreibt die Aktion „Der Wert c wird an den Ausgabekanal out geschickt.“ Die entsprechende Eingabetransition wird als $in = c$ notiert ($in \in Inputs$).
- **Datenfluss:** Die Aktion $send(out)$ und ihr entsprechendes Ereignis $update(in)$ abstrahieren von konkreten Werten und sind für untstrukturierte Daten zuständig. Einen funktionalen Zusammenhang zwischen Aus- und Eingabekanälen drückt ein Funktionssymbol $f \in Identifier$ mit Eingabekanälen oder Konstanten als Parameter aus:
$$out := f(in_1, c_2, in_3, \ldots)$$
Empfangen wird diese Nachricht ebenfalls durch $update(in)$.
- **Parameter:** Im gewissen Sinne ähneln Parameter Eingabekanälen des Kontrollflusses. Ein wichtiger Unterschied ist, dass Stub-Parameter nur „von außen“ durch den Benutzer geändert werden, niemals durch andere Stubs. Die Integration der Parameter gestaltet sich deshalb etwas schwierig, weil diese Unterscheidung nur ungenügend formalisiert werden kann. Es wurde deshalb folgender Ansatz gewählt: Parameter verhalten sich auf Stub-Ebene wie Eingabekanäle, sie können jedoch bei der Zusammenschaltung von Stubs nicht gebunden werden. Zudem soll eine Verifikation nicht dadurch fehlschlagen, dass willkürlich Parameter geändert werden.
- **Zeit:** Um auch – in beschränktem Ausmaß – Echtzeiteigenschaften untersuchen zu können, wird das Ereignis „tick“ definiert, welches das Ereignis „es vergeht ein Zeitschritt“ (wie oben gesagt: die Dauer des Versendens einer Nachricht) darstellt.

In der folgenden Tabelle ist aufgeführt, welche Ausgabenachrichten (Aktionen) welche Ereignisse in an ihn gebundene Stubs auslösen (mit ✓ markiert) bzw. welche Kombinationen unmöglich (weil nicht entscheidbar) sind (ϟ).

Aktion	**Ereignis**	$in = c$	$update(in)$	$param = c$	$tick$
$out := c$		✓	✓		✓
$out := f(\ldots)$		ϟ	✓		✓
$send(out)$		ϟ	✓		✓
$tick$					✓

Damit definieren wir die Sprache der Nachrichten Δ in ISO-EBNF:

$$
\begin{array}{rcl}
\Delta & = & Event \;\Big|\; Action \;\Big|\; Waiting \\
Event & = & Inputs_{Type} \text{ „=“ } Const_{Type} \;\Big|\; \text{„update(“ } Inputs \text{ „)“} \\
 & & \Big|\; Params_{Type} \text{ „=“ } Const_{Type} \\
Action & = & Outputs_{Type} \text{ „:=“ } Const_{Type} \;\Big|\; \\
 & & Outputs \text{ „:=“ } Identifier \text{ „(“ } Arg \text{ \{„,“ } Arg \text{ \} „)“} \;\Big|\; \text{„send(“ } Outputs \text{ „)“} \\
Arg & = & Const \;\Big|\; Inputs \\
Waiting & = & \text{„tick“} \;\Big|\; \text{„tick(“ } PositiveNaturalNumber \text{ „)“}
\end{array}
$$

Abb. 3.2: Syntax der Transitionen

Um die Typkorrektheit darzustellen, sind die Produktionsregeln als Metaregeln zu verstehen, die für jeden Datentyp aus $Type$ instantiiert werden. $Const_{Type}$ ist dabei eine Konstante vom Typ $Type$.

Relationen wie $>$ oder $\leq$ werden für numerische Variablen nicht definiert, da mit ihnen eine Abbildung auf eine endliche Menge atomarer Transitionen nicht mehr möglich ist. Wenn man den Wertebereich der Variablen stark beschränken würde, wäre der Einsatz eines Constraint Solvers denkbar; da diese Arbeit aber nicht zum Ziel hat, die Funktionalität des Netzes vollständig zu verifizieren, sondern auf der gröberen Ebene der Passfähigkeit arbeitet, wurde darauf verzichtet.

Die Erweiterung „tick(n)“ bedeutet „nach n Zeitschritten“. $PositiveNaturalNumber$ ist dabei eine positive natürliche Zahl.

3.2.3 Definition der Prozessalgebra

Die Beschreibung des Verhaltens eines Stubs wird üblicherweise „Prozess“ genannt. Diese werden in einer textuellen Sprache beschrieben, die einerseits formal definiert, andererseits auch einfach und intuitiv zu verstehen ist.

Basis der Prozessbeschreibungs-Sprache von CCMB sind die Menge Δ aller möglichen Transitionen und eine Menge KON von Prozesskonstanten. Die Definition einer Konstanten kann als Definition der möglichen Zustandsübergänge verstanden werden und die Konstante selbst als Name für den Zustand. Allerdings muss nicht jeder Zustand einen eigenen Namen (also eine explizite Prozesskonstante) tragen, da in vielen Prozessen Zustände existieren, auf die man sich nicht extern beziehen muss.

Um Zustände gegenüber anderen auszeichnen zu können, wird eine Menge Φ atomarer Prädikate eingeführt. Prädikate sind Zustandsmarkierungen, auf die man sich bei der Formulierung von Eigenschaften beziehen kann.

$Process \;=\; KON \;\Big|\; „0“ \;\Big|\; „[“\ \Phi\ „]“\ Process \;\Big|\; \Delta\ „\rightarrow“\ Process \;\Big|\; Process\ „|“\ Process$
$ProcessDef \;=\; KON\ „:=“\ Process$

Abb. 3.3: Syntax der Prozessalgebra zur Verhaltensbeschreibung von Stubs

0 ist der Nullprozess, der keine Aktionen ausführen kann.

Der Pfeiloperator beschreibt eine Transition in einen Folgezustand. $tr \rightarrow P$ reagiert also auf das Ereignis bzw. führt die Aktion tr aus und verhält sich dann wie Prozess P.

Der |-Operator beschreibt eine Alternative, d. h. $P1|P2$ verhält sich *entweder* wie $P1$ *oder* wie $P2$. Beginnen $P1$ und $P2$ mit unterschiedlichen Transitionen, so ist die Auswahl deterministisch. Die Definition erlaubt aber auch indeterministische Prozesse, wenn die ersten Transitionen beider Alternativen gleich sind.

Prädikate eines Zustandes werden als Präfix in eckige Klammern geschrieben. Ein Beispiel sei die Unterscheidung der Zustände in „gefährlich“ und „sicher“: Mit $danger, safe \in \Phi$ drückt der Prozess

$$[danger]\ stop = 1 \rightarrow [safe]\ OFF$$

aus, dass er anfänglich in einem gefährlichen Zustand ist und nach Eintreffen der Nachricht $stop = 1$ in einen ungefährlichen Zustand OFF übergeht.

Genutzt werden die Prozessausdrücke in definierenden Gleichungen $K := P$.

Um die Notwendigkeit der Klammersetzung zu beschränken und Formeln übersichtlicher zu halten, erhalten die Operatoren nun noch je eine Priorität und Assoziativität. Die Zahlen der Prioritätsspalte stellen eine Rangfolge dar, d. h., eine kleine Zahl bedeutet höhere Priorität, bzw. eine stärkere Bindung:

Operator	**Priorität**	**Assoziativität**
$\rightarrow$	1	rechts
$[\varphi]$	2	ja
$\vert$	3	links (willkürlich, weil irrelevant)

Abb. 3.4: Präzedenz und Assoziativität der Prozessalgebra-Operatoren

3.2.4 Semantik

Die Semantik der soeben definierten Prozessausdrücke wird nun durch eine Abbildung auf die bei temporalen Logiken üblichen Kripke-Strukturen ähnlich wie in [Gol92] definiert:

Eine Kripke-Struktur (Δ-Rahmen, Δ-labeled transition system) ist eine Struktur

$$\mathcal{F} = \Big(S, s^0 \in S, next \subseteq S \times \Delta \times S, V : \Phi \rightarrow \mathcal{P}(S)\Big)$$

die eine Menge S der Zustände, den Startzustand s^0 und eine Relation $next$ der möglichen Zustandsübergänge umfasst. Ein Element $(s, \delta, s') \in next$ ist dabei zu verstehen als „im Zustand s ist die Aktion (Nachricht) δ möglich und diese führt in den Nachfolgezustand s'“.

Wie schon oben erläutert ist Φ die Menge atomarer Zustandsprädikate. $V(\varphi)$ mit $\varphi \in \Phi$ wird dabei interpretiert als Menge der Zustände, die das Prädikat φ haben.

Aus den Prozesskonstanten KON und einer Menge $PD \subseteq ProcessDef$ von Prozessdefinitionen wird nun eine Kripke-Struktur konstruiert. Als Zustände wird die Menge $Process$ aller Prozesse verwendet und die $next$-Relation ist die kleinste Menge, die folgende Bedingungen (entsprechend den verbalen Beschreibungen in den Abschnitten 3.2.2 und 3.2.3) erfüllt:

- 0 hat keine Nachfolger:

  $\nexists \delta \in \Delta, P \in Process.\ (0, \delta, P) \in next$
- Eine erwartete Transition wird ausgeführt:

  $\forall \delta \in \Delta, P \in Process.\ (\delta \rightarrow P, \delta, P) \in next$
- $update()$ reagiert auf einen beliebigen Eingabewert:

  $\forall \tau \in Type.\ \forall i \in Inputs_\tau, c \in Const_\tau, P \in Process.$

  $\quad (update(i) \rightarrow P, i = c, P) \in next$
- $tick(n)$ entspricht n-maligem tick:

  $\forall P \in Process, n \in \mathbb{N}^+.$

  $\quad (tick(1) \rightarrow P, tick, P) \in next$

  $\wedge\ (tick(n+1) \rightarrow P, tick, tick(n) \rightarrow P) \in next$
- Ein Prozess ist unabhängig von allen anderen Alternativen:

  $\forall \delta \in \Delta, P_1, P_1', P_2 \in Process.$

  $\quad (P_1, \delta, P_1') \in next \quad \rightarrow \quad (P_2|P_1, \delta, P_1') \in next \wedge (P_1|P_2, \delta, P_1') \in next$
- Eine Prozessdefinition bedeutet, dass die Konstante äquivalent zum definierten Prozess ist:

  $\forall \delta \in \Delta, P, P' \in Process, K \in KON.$

  $\quad (P, \delta, P') \in next \wedge (K := P) \in PD \quad \rightarrow \quad (K, \delta, P') \in next$
- atomare Eigenschaften bleiben bei einem $tick$ erhalten, bei anderen Transitionen nicht:

  $\forall \varphi \in \Phi; \delta \in Event \cup Action; P, P' \in Process.$

  $\quad (P, tick, P') \in next \quad \rightarrow \quad ([\varphi]P, tick, [\varphi]P') \in next$

  $\wedge\ (P, \delta, P') \in next \quad \rightarrow \quad ([\varphi]P, \delta, P') \in next$

Abb. 3.5: Semantik der Prozessausdrücke für Stubs

Die Semantik bezüglich der atomaren Prädikate, d. h. die Definition der Funktion V zeigt folgende Abbildung:

- Prozess mit Prädikatspräfix. Ein Prozess kann mehrere Präfixe haben, die rekursiv durchsucht werden:

  $\forall \varphi, \psi \in \Phi, P \in Process.\ [\psi]P \in V(\varphi) \ \Leftrightarrow\ \varphi = \psi \vee P \in V(\varphi)$
- 0 und eine Transition haben keine Prädikate:

  $\forall \varphi \in \Phi.\ 0 \notin V(\varphi)$

  $\forall \varphi \in \Phi, P \in Process, \delta \in \Delta.\ (\delta \rightarrow P) \notin V(\varphi)$
- Die Prädikate einer Alternative ist die Vereinigung der Prädikate der alternativen Prozesse:

  $\forall \varphi \in \Phi; P_1, P_2 \in Process.\ (P_1|P_2) \in V(\varphi) \ \Leftrightarrow\ P_1 \in V(\varphi) \vee P_2 \in V(\varphi)$
- Die Prädikate einer Prozesskonstanten entsprechen den Prädikaten ihrer Definition:

  $\forall \varphi \in \Phi, K \in Kon, P \in Process.\ K \in V(\varphi) \ \Leftrightarrow\ (K := P) \in PD \wedge P \in V(\varphi)$

Abb. 3.6: Semantik der Prozessausdrücke für atomaren Prädikate

3.3 Notation einer Stub-Beschreibung

Die Schnittstelle und das Verhalten eines Stubs s werden zusammengefasst als ein Tupel

$$s = (Inputs_s, Outputs_s, Params_s, \mathcal{F}_s)$$

Ist im folgenden ein Stub s gegeben, so sei im folgenden die Notation $Element_s$ vereinbart, die das entsprechende Element des Beschreibungstupels des Stubs s referenziert. Z. B. bezeichnet $Inputs_s$ die Eingangskanäle von s. Weiterhin bezeichnen $next_s$ und V_s die $next$-Relation und die Abbildung V in $\mathcal{F}_s$.

3.4 CCMB-Syntax

In CCMB wird ein Stub mit folgender Syntax definiert:

$$StubDesc \ = \ \text{„stub“}\ \underbrace{Identifier}_{Stubname}\ \text{„\{“}\ \{Inputs|Outputs|Params\}\ \{ProcessDef\}\ \text{„\}“}$$

Es sei vereinbart, dass die erste auftretende Definition einer Prozesskonstanten der Startzustand ist.

3.5 Beispiele

Zur Demonstration der Syntax und Semantik sollen hier nur sehr einfache Stubs dienen. Kapitel 6, wo die Anwendungsfälle aus Kapitel 2 implementiert werden, bietet wesentlich komplexere Stubs.

Ein sehr einfacher Stub ist die Schnittstelle einer Uhr bzw. eines Taktgebers, die nichts anderes tut, als fortlaufend „tock“-Nachrichten zu senden:

```
stub CLOCK {
    portout bool tock

    CL := send(tock)-> CL
}
```

Dieser Stub trägt den Namen $CLOCK$, besitzt nur einen einzigen Ausgang *tock* vom Typ *bool* (d. h., ein Wahrheitswert), hat den Startzustand CL und verhält sich nach dem Senden von $send(tock)$ wieder wie CL. Die dazu äquivalente Semantik ist dann (vgl. Abb. 3.5):

$$\begin{aligned} CLOCK &= (\varnothing, \{tock_{bool}\}, \varnothing, \mathcal{F}_{CLOCK}) \\ \mathcal{F}_{CLOCK} &= (\{CL\}, CL, next_{CLOCK}, \varnothing) \\ next_{CLOCK} &= \{(CL, send(tock), CL)\} \end{aligned}$$

Jetzt soll die Uhr um einen Eingang *stop* erweitert werden; bei „stop=true“ soll die Uhr (für immer) anhalten:

```
stub HCLOCK {
    portin bool stop
    portout bool tock

    CL := stop=true -> 0 | send(tock) -> CL
}
```

Die dazugehörige Semantik:

$$\begin{aligned} HCLOCK &= (\{stop_{bool}\}, \{tock_{bool}\}, \varnothing, \mathcal{F}_{HCLOCK}) \\ \mathcal{F}_{HCLOCK} &= (\{CL, 0\}, CL, next_{HCLOCK}, \varnothing) \\ next_{HCLOCK} &= \{(CL, send(tock), CL), (CL, stop = true, 0)\} \end{aligned}$$

Um das ganze zu einer richtigen Stopp-Uhr zu erweitern, darf die Uhr nicht für immer anhalten (im Zustand 0), sondern muss nach einer Nachricht „stop=false“ weiterticken:

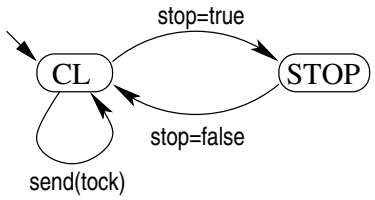
```
stub SCLOCK {
    portin bool stop
    portout bool tock

    CL := stop=true -> STOP | send(tock) -> CL
    STOP := stop=false -> CL
}
```

Die Semantik ergibt sich dann wie folgt:

$$\begin{aligned} SCLOCK &= (\{stop_{bool}\}, \{tock_{bool}\}, \varnothing, \mathcal{F}_{SCLOCK}) \\ \mathcal{F}_{SCLOCK} &= (\{CL, STOP\}, CL, next_{SCLOCK}, \varnothing) \\ next_{SCLOCK} &= \{(CL, send(tock), CL), (CL, stop = true, STOP), \\ &\quad (STOP, stop = false, CL)\} \end{aligned}$$

Zur Veranschaulichung sei hier auch einmal das Zustandsübergangsdiagramm gegeben:



4 Verbindung von Stubs zu Netzwerken

„TEAM means:
Together
Everybody
Achieves
More“

Norman Frische

Bei der Integration eines Systems werden mehrere Einzelkomponenten zu einem Netzwerk zusammengeschaltet, in dem die Ein- und Ausgänge ihrer Stubs physisch oder logisch über Nachrichtenkanäle verbunden werden.

Die Definition eines solchen Netzwerkes erfolgt in den „Network Profiles“ oder in „Stub Groups“, wie in Abschnitt 1.2 eingeführt. Solch ein Netzwerk N definiert demnach eine Menge von kooperierenden Stubs $Stubs_N$ und eine Bindungsliste $bind_N$.

$Stubs_N$ sei dabei eine Menge von Stub-Instanzen, die durch einen Namen bezeichnet werden. Die Elemente dieser Menge sind dabei Tupel $(name, stub)$, wobei $name$ ein Bezeichner und $stub$ eine Stub-Schnittstelle in Tupelform entsprechend Abschnitt 3.3 ist. Zur besseren Lesbarkeit wird ein Element als $name : stub$ notiert.

Ein Netzwerk definiert auch eine Menge von Anforderungen die es erfüllen soll. Darauf wird im Kapitel 5 eingegangen, das die Logik und das Verifikationskalkül einführt.

Da in einem Netzwerk mehrere Stubs mit gleichen Input-, Outputnamen oder Parametern existieren können, ist es notwendig, diese mit dem zugehörigen Stubnamen zu qualifizieren. Daher wird die Schnittstelle (Ein- und Ausgänge und Parameter) einer Zusammenschaltung einer Menge von Stubs $Stubs_N$ wie folgt definiert:

- $Inputs_N := \bigcup_{(name,s) \in Stubs_N} \{name\} \times Inputs_s$
- $Outputs_N := \bigcup_{(name,s) \in Stubs_N} \{name\} \times Outputs_s$
- $Params_N := \bigcup_{(name,s) \in Stubs_N} \{name\} \times Params_s$

Für $(stubname, channel)$ wird die Notation $stubname.channel$ verwendet, die in Programmiersprachen zur Qualifikation von Variablen sehr verbreitet ist.

4.1 Modellierung

Eine fundamentale Entscheidung ist das Verhalten von Ausgabekanälen in dem Fall, dass in einem Netzwerk nicht alle an einen Ausgang gebundenen Eingänge gerade im Zustand „warte auf Eingabe von diesem Ausgang“ sind. Es eröffnen sich drei Möglichkeiten, auf deren Vor- und Nachteile zur Entscheidungsfindung eingegangen werden:

1. **vollständige Synchronisation:** die Ausgabe wird solange verzögert, bis alle gebundenen Eingänge im Zustand „warte auf Eingabe“ sind.

   + kein Verlust von Nachrichten (entspricht meist der Realität)

   – ein nicht bereiter Stub blockiert gesamtes Netzwerk (unrealistisch)

2. **Nachrichtenwarteschlangen:** Nachrichten werden nicht direkt an einen Stubeingang geschickt, sondern in einer Warteschlange (Queue) gespeichert, bis der Stub eine Eingabe-Operation durchführt.

   • sehr implementationsspezifisch; wird oft so implementiert und entspricht dann auch der Realität (insbes. Echtzeitverhalten), bringt aber zusätzliche Parameter wie Queue-Größen, Vorhaltezeiten usw. ins Spiel

   + Nachrichten gehen nicht (sofort) verloren

   – komplizierte formale Semantik

3. **halbseitige Synchronisation:** eine Nachricht wird sofort gesendet; jeder Stub, der nicht im Zustand „warten auf Eingabe“ ist, erhält die Nachricht nicht.

   + Stubs können andere Stubs nicht blockieren (realistisch)

   + einfache formale Semantik, entspricht CSS-Standard

   – Nachrichten können verloren gehen

   • vollständige Synchronisation kann durch „Handshaking“ (gegenseitiges Senden einer Nachricht) modelliert werden

Somit wäre die Modellierung eines konkreten Stubs bei einer Warteschlangensemantik am direktesten. Leider muss sie auf jedes konkrete Feldbus-System angepasst werden und ist in diesem Rahmen zu kompliziert. Deshalb wird im Rahmen dieser Arbeit die dritte Alternative (halbseitige Synchronisation) verwendet.

Dies ist aber keine endgültige Entscheidung. Es ist ohne Probleme möglich, in einer Implementation des Verifikationssystemes alternative Semantiken anzubieten. Das Verifikationskalkül (siehe Kapitel 5) ist davon unabhängig und auch die Syntax der Prozessalgebra muss dafür nicht geändert werden.

4.2 Definition von Bindungen

Eine **Bindungsliste** eines Netzwerkes ist eine Abbildung

$$bind_N : Outputs_N \rightarrow \mathcal{P}(Inputs_N)$$

mit folgenden Eigenschaften:

- Alle Variablen einer Bindung sind vom gleichen Typ
- Ein Eingang darf nicht an mehr als einen Ausgang gebunden werden:

$$\forall o_1, o_2 \in Outputs_N.\ o_1 \neq o_2 \Rightarrow bind_N(o_1) \cap bind_N(o_2) = \varnothing$$

$bind_N$ weist einem Ausgangskanal alle an ihn gebundenen Eingänge zu. Dies ermöglicht das Binden mehrerer Eingänge an denselben Ausgang, was technisch möglich und sinnvoll ist oder die Abbildung auf die leere Menge, was bedeutet, dass der Ausgang nicht gebunden ist.

Das Binden mehrerer Ausgänge an einen Eingang ist in manchen Fällen sicher auch sinnvoll; es wirft aber die Frage auf, was passiert, wenn zwei Ausgänge gleichzeitig zwei verschiedene Werte an einen Eingang senden. Das Verhalten eines praktischen Systems in diesen Falle ist implementierungsspezifisch. Denkbar wäre zum Beispiel eine zufällige Entscheidung; die Semantik könnte das berücksichtigen, in dem in so einem Falle mehrerer alternative Nachfolgerzustände mit je einem Eingangswert möglich wären.

4.3 Freie und gebundene Kanäle

$bind_N$ partitioniert die Kommunikationskanäle eines Netzwerkes in **freie** und **gebundene** Ein- und Ausgänge. Gebundene Kanäle sind auf der Ebene der Netzwerke nicht mehr zugänglich, die noch freien Kanäle stellen die Schnittstelle von N dar.

$$BoundInputs_N := \{i \in Inputs_N \mid \exists o \in Outputs_N.\ i \in bind_N(o)\} \tag{4.1}$$

$$FreeInputs_N := Inputs_N \setminus BoundInputs_N \tag{4.2}$$

$$FreeOutputs_N := \{o \in Outputs_N \mid bind_N(o) = \varnothing\} \tag{4.3}$$

$$BoundOutputs_N := Outputs_N \setminus FreeOutputs_N \tag{4.4}$$

Wenn ein Ausgang o gebunden ist, d. h., er in $BoundOutputs_N$ ist, dann sollten alle an ihn gebundenen Eingänge in $BoundInputs_N$ liegen. Diese Konsistenz stellt der folgende Satz sicher:

Unter den Definitionen (4.1) bis (4.4) gilt:

$$o \in BoundOutputs_N \quad \Rightarrow \quad \forall i \in bind_N(o).\ i \in BoundInputs_N$$

Beweis (durch Widerspruch): Sei $out \in BoundOutputs_N$. Annahme: es gibt ein ungebundenes $in \in bind_N(out)$, d. h. $in \notin BoundInputs_N$. Daraus folgt nach (4.1):

$$\nexists o \in Outputs_N.\ in \in bind_N(o)$$

Aus (4.4) folgt $BoundOutputs_N \subseteq Outputs_N$ und deshalb gilt auch

$$\nexists o \in BoundOutputs_N.\ in \in bind_N(o)$$

was der ursprünglichen Annahme $out \in BoundOutputs_N$ widerspricht. □

4.4 Semantik

Gestellte Anforderungen beziehen sich meist auf das gesamte Netzwerk, nicht auf die einzelnen Stubs. Deshalb muss das Gesamtverhalten $\mathcal{F}_N$ von N definiert werden, um die Eigenschaften daran verifizieren zu können. Die gewählte Modellierung verlangt folgende Semantik:

1. Eine von „außen“ eintreffende Transition, d. h. ein Ereignis oder eine Aktion eines ungebundenen Kanals wird entsprechend seiner Qualifizierung durch den Stubnamen an den entsprechenden Stub weitergeleitet. Die anderen Stubs führen eine „tick“-Transition durch, um zeitsynchron zu bleiben; können sie das nicht (weil sie z. B. selbst eine Ausgabe vornehmen wollen), dann bleiben sie im aktuellen Zustand.
2. Neben den äußeren Ereignissen kann das System auch innere Aktionen auslösen: wenn in einem Stub eine Ausgabeaktion möglich ist, dann wird sie zusammen mit den entsprechenden Eingabeereignissen der an ihn gebundenen Stubs durchgeführt. Die anderen Stubs führen wiederum – sofern möglich – einen „tick“ durch.
3. Das ganze Netzwerk kann einen „tick“ durchführen, wenn keines seiner Stubs eine Aktion durchführen kann.

In dieser Modellierung kann immer nur eine Nachricht zu einer Zeit gesendet werden. Bei Feldbussystemen entspricht dies auch der Realität. Es werden aber trotzdem alle Reihenfolgen der aktuell möglichen Nachrichten betrachtet und verifiziert, weil die Formulierung von *next* als Relation (und nicht als Funktion) mehrere mögliche Nachfolgezustände erlaubt.

Sei $Stubs_N = \{name1 : s1, \ldots, namek : sk\}$ die Menge der im Netzwerk enthaltenen Stubs. Ausgehend von deren Kripke-Strukturen $\mathcal{F}_{s1}, \ldots, \mathcal{F}_{sk}$ und den Bindungen $bind_N$ wird nun das Gesamtverhalten von N

$$\mathcal{F}_N = (S_N, s_N^0, next_N \subseteq S_N \times \Delta \times S_N, V_N)$$

definiert durch die „Kreuzstruktur“:

- $S_N = S_{s1} \times \cdots \times S_{sk}$
- $s_N^0 = (s_{s1}^0, \ldots, s_{sk}^0)$
- $V_N(namei.\varphi) = \left\{ (s_1, \ldots, s_k) \middle| \; s_i \in V_{si}(\varphi) \right\}$
- $V_N(\varphi) = \left\{ (s_1, \ldots, s_k) \middle| \; \exists i.\; s_i \in V_{si}(\varphi) \right\}$

Der Zustandsraum ergibt sich demnach als kartesisches Produkt der Stub-Zustände, im Anfangszustand des Netzwerkes ist jeder Stub in seinem Anfangszustand.

Bei der Frage, ob eine atomare Eigenschaft in einem Zustand wahr ist, gibt es zwei Möglichkeiten: man kann testen, ob sie in einem ganz bestimmten Stub wahr ist (in der qualifizierten Variante) oder ob sie in irgendeinem Stub wahr ist (unqualifizierte Variante).

Die Definition der *next*-Relation ergibt sich aus der Formalisierung der oben beschriebenen Modellierung. Dazu werden zunächst einige Notationen und Hilfsprädikate eingeführt:

- Ein Netzwerkzustand $(P_1, \ldots, P_k) \in S_N$ wird kurz mit $\vec{P}$ bezeichnet. Entsprechend ist $\vec{P'} := (P'_1, \ldots, P'_n)$. Der Bezeichner P anstelle von s deutet an, dass hier ein spezieller Prozessausdruck als Zustandsbeschreibung verwendet wird.
- $\vec{P_i} := (P_1, \ldots, P_{i-1}, P_{i+1}, \ldots, P_n)$ ist der Netzwerkzustand $\vec{P}$ ohne den Zustand P_i des Stubs i.
- Das Prädikat $PossTick_{st}(P, P')$ ordnet dem Zustand P des Stubs st den *tick*-Nachfolger P' zu. Wenn st keinen *tick* ausführen kann, dann bleibt er im aktuellen Zustand, d. h. $P' = P$:

  $$PossTick_{st}(P, P') \;\equiv\; (P, tick, P') \in next_{st} \;\vee\; [\nexists P''.\; (P, tick, P'') \notin next_{st} \wedge P' = P]$$

- Das Prädikat $PossTick(\vec{P}, \vec{P'})$ ordnet den Stubzuständen P_i die *tick*-Nachfolger P'_i zu; Stubs, die keinen *tick* durchführen können, bleiben dabei im aktuellen Zustand:

  $$PossTick((P_1, \ldots, P_n), (P'_1, \ldots, P'_n)) \;\equiv\; \forall i \in \{1 \ldots n\}.\; PossTick_i(P_i, P'_i)$$

- $BindEv_{st}(P, a, P')$ weist für die Aktion $a \in Action_N$ dem Zustand P des Stubs st den entsprechenden Nachfolgerzustand P' zu, wenn st auf ein entsprechendes Ereignis wartet:

  $$BindEv_{st}(P, os.out := c, P') \;\equiv\; \exists in.\; st.in \in bind(os.out) \wedge (P, in = c, P') \in next_{st}$$

  $$BindEv_{st}(P, send(os.out), P') \;\equiv\; BindEv_{st}(P, os.out := f(\vec{x}), P') \;\equiv$$
  $$\exists in.\; st.in \in bind(os.out) \;\wedge\; (P, update(in), P') \in next_{st}$$

- $BindEv(\vec{P}, a, \vec{P'})$ erledigt die Zuordnung der gebundenen Nachfolgerzustände für alle Stubs eines Netzwerkes. Wenn ein gebundener Stub nicht auf das entsprechende Ereignis wartet, führt er stattdessen einen *tick* durch:

  $$BindEv((P_1, \ldots, P_n), a, (P'_1, \ldots, P'_n)) \;\equiv$$
  $$\forall i \in \{1 \ldots n\}.\; BindEv(P_i, a, P'_i) \;\vee\; [\nexists P'.\; BindEv(P_i, a, P') \wedge PossTick(P_i, P'_i)]$$

Nun wird $next_N$ entsprechend der drei Modellierungsanforderungen definiert:

Aktionen gebundener und freier Ausgänge:

$$(\vec{P}, st.out := c, \vec{P'}) \in next_N \;\Leftrightarrow\; (P_{st}, out := c, P'_{st}) \in next_{st} \;\wedge$$
$$st.out \in FreeOutputs_N \;\wedge\; PossTick(\vec{P}_{(st)}, \vec{P'}_{(st)})$$
$$\vee\; st.out \in BoundOutputs_N \;\wedge\; BindEv(\vec{P}_{(st)}, st.out := c, \vec{P'}_{(st)})$$

$$(\vec{P}, st.out := f(\vec{x}), \vec{P'}) \in next_N \;\Leftrightarrow\; (P_{st}, out := f(\vec{x}), P'_{st}) \in next_{st} \;\wedge$$
$$st.out \in FreeOutputs_N \;\wedge\; PossTick(\vec{P}_{(st)}, \vec{P'}_{(st)})$$
$$\vee\; st.out \in BoundOutputs_N \;\wedge\; BindEv(\vec{P}_{(st)}, st.out := f(\vec{x}), \vec{P'}_{(st)})$$

$$(\vec{P}, send(st.out), \vec{P'}) \in next_N \;\Leftrightarrow\; (P_{st}, send(out), P'_{st}) \in next_{st} \;\wedge$$
$$st.out \in FreeOutputs_N \;\wedge\; PossTick(\vec{P}_{(st)}, \vec{P'}_{(st)})$$
$$\vee\; st.out \in BoundOutputs_N \;\wedge\; BindEv(\vec{P}_{(st)}, send(st.out), \vec{P'}_{(st)})$$

Ereignisse gebundener und freier Eingänge: (Ein gebundenes Ereignis kann nur stattfinden, wenn die entsprechende auslösende Aktion möglich ist.)

$$(\vec{P}, st.in = c, \vec{P'}) \in next_N \;\Leftrightarrow\; (P_{st}, in = c, P'_{st}) \in next_{st} \;\wedge$$
$$st.in \in FreeInputs_N \;\wedge\; PossTick(\vec{P}_{(st)}, \vec{P'}_{(st)})$$
$$\vee\; st.in \in BoundInputs_N \;\wedge\; (\exists st'.out\,.\; st.in \in bind_N(st'.out)) \;\wedge$$
$$(\vec{P}, st'.out := c, \vec{P'}) \in next_N$$

$$(\vec{P}, update(st.in), \vec{P'}) \in next_N \;\Leftrightarrow\; (P_{st}, update(in), P'_{st}) \in next_{st} \;\wedge$$
$$st.in \in FreeInputs_N \;\wedge\; PossTick(\vec{P}_{(st)}, \vec{P'}_{(st)})$$
$$\vee\; st.in \in BoundInputs_N \;\wedge\; (\exists st'.out\,.\; st.in \in bind_N(st'.out)) \;\wedge$$
$$\Big((\vec{P}, send(st'.out), \vec{P'}) \in next_N \;\vee\; \exists c.\; (\vec{P}, st'.out := c, \vec{P'}) \in next_N\Big)$$

Parameteränderungen:

$$(\vec{P}, st.param = c, \vec{P'}) \in next_N \;\Leftrightarrow$$
$$st.param \in Params_N \;\wedge\; (P_{st}, param = c, P'_{st}) \in next_{st} \;\wedge\; PossTick(\vec{P}_{(st)}, \vec{P'}_{(st)})$$

„tick“ des ganzen Netzwerkes: (Wenn ein Stub nicht explizit „tick“ ausführen kann, dann darf auch keine Aktion möglich sein)

$$\Big((P_1, \ldots, P_n, tick, (P'_1, \ldots, P'_n)\Big) \in next_N \;\Leftrightarrow\; \forall i \in \{1, \ldots, n\}.$$
$$(P_i, tick, P'_i) \in next_i \;\vee\; (P'_i = P_i \wedge \nexists a \in Action_{P_i}, P'.\; (P_i, a, P') \in next_i)$$

4.5 Beschreibung und Kompositionen

Nach den obigen Definitionen ist die Schnittstelle eines Stub-Netzwerkes N analog zur Beschreibung eines Stubs (siehe Abschnitt 3.3) ein Tupel

$$N = (FreeInputs_N, FreeOutputs_N, Params_N, \mathcal{F}_N)$$

Diese strukturelle Gleichheit ermöglicht eine beliebig tiefe Komposition von Stubs als „Stub Group“ (Komposition von Stubs) oder „Network Profile“ (Komposition von Stubs und Stub Groups).

4.6 Initialisierung

In der syntaktischen Definition von Stubs wurden Parameter eingeführt. Werden Stubs zu einem Netzwerk zusammengeschlossen, ist es wünschenswert, die Parameter entsprechend den Anforderungen des Netzwerkes einzustellen. Die momentane Formalisierung erlaubt die Berücksichtigung von Konfigurationseinstellungen durch die Wahl des Startzustandes.

Um die Handhabbarkeit zu verbessern und realistischer zu machen, scheint für die Implementation aber ein anderer Ansatz geeignet: Zu jedem Netzwerk N existiert eine Liste von Aktionen $Init_N \in (Event_N)^*$, die Parameterzuweisungen enthält und die bei der Instantiierung eines Netzwerkes abgearbeitet werden. Dann kann ein Netzwerk einen kanonischen Startzustand erhalten, der erst bei der Instantiierung des Netzwerkes durch die Parameter verändert wird.

Diese Idee wurde absichtlich nicht mit in die syntaktische Definition aufgenommen, weil sie die Syntax- und Semantikdefinition nur unnötig erschweren würde, ohne prinzipiell neue Funktionalität hinzuzufügen. Zusätzlich bleibt das Konzept einer Initialisierungs-Aktionsliste damit optional und muss nicht zwingend verwendet werden.

Zur bequemen Handhabung empfiehlt es sich, zu einem Netzwerk N eine Relation

$$next_N^* \subseteq S_N \times \Delta_N^* \times S_N$$

zu definieren, die die Folgezustände für eine Liste von Aktionen abbildet:

$\forall s \in S_N.\ (s, \varepsilon, s) \in next_N^*$ (Basisfall für leere Liste)

$\forall s, s'' \in S_N; \delta_1, \ldots, \delta_k \in \Delta_N.$
$\quad (s, (\delta_1, \delta_2, \ldots, \delta_k), s'') \in next_N^* \ \Leftrightarrow$
$\quad \exists s' \in S_N.\ (s, \delta_1, s') \in next_N \ \wedge\ (s', (\delta_2, \ldots, \delta_k), s'') \in next_N^*$ (Rekursion)

Sind alle Parametereinstellungen deterministisch – was in praktischen Systemen wohl oft der Fall ist – dann ist diese Relation auch funktional, so dass es nur ein s^{Init} mit

$$(s_N^0, Init_N, s_N^{Init}) \in next_N^*$$

gibt.

5 Verifikation von Anforderungen

„Logik ist der Beginn der Weisheit –
nicht das Ende“

Spock (Filmfigur aus „Star Trek“)

In diesem Kapitel wird für CCMB eine formale Sprache zur Beschreibung von Anforderungen und gewünschten Eigenschaften an Stub-Netze entwickelt. Mit Hilfe eines Verifikationskalküles soll es möglich sein, diese Anforderungen stets nichtinteraktiv verifizieren zu können, bzw. Gegenbeispiele zu finden.

Diese Anforderungen erfordern, dass die verwendete Logik entscheidbar ist. Man kann deshalb nicht beliebig ausdrucksstarke Logiken und auch keine unendlichen Datentypen verwenden.

Die einfachste Logik, mit denen Verhalten beschrieben und verifiziert werden kann, ist die modale Aussagenlogik. Diese kann allerdings nur lokale Eigenschaften beschreiben, weil aussagenlogische modale Formeln nur Aussagen machen können, die eine maximale Anzahl von „ticks“ in der Zukunft liegen. Sicherheitseigenschaften wie „die Temperatur darf nie über 70 [Grad] steigen“ oder „kein Stub des Netzes darf jemans in einen mit `error` ausgezeichneten Zustand gelangen“ oder Lebendigkeitseigenschaften wie „der Prozess muss irgendwann terminieren“ und „jede Anfrage wird nach endlicher Zeit beantwortet“ sind damit nicht ausdrückbar.

Deshalb wird als Grundlage der μ-Kalkül verwendet, der eine sehr reichhaltige Logik bietet und für endliche Systeme auch ein entscheidbares (d. h. korrektes und vollständiges) Beweiskalkül besitzt. Ein Nachteil des μ-Kalküls ist, dass die Formulierung von Sicherheits- und Lebendigkeitseigenschaften nicht sehr intuitiv ist; Formeln in Logiken wie CTL oder CTL^* sind wesentlich verständlicher. Da aber diese und andere temporale Logiken im μ-Kalkül enthalten sind, soll dieser Nachteil durch die Definition oft gebrauchter und allgemein verwendbarer Makros aufgehoben werden.

5.1 Modaler μ-Kalkül

Die Syntax beruht wiederum auf der Menge Φ, deren Elemente aber hier allgemeiner als Variablennamen betrachtet werden:

Sei Φ eine Menge von Variablen und Δ eine Menge von Transitionen (vgl. Abschnitt 3.2.4). Mit $Z \in \Phi$ und $A \subseteq \Delta$ ist die Menge der Formeln F des μ-Kalküls durch folgende EBNF-Beschreibung definiert:

$$F = Z \;\Big|\; F \text{ „}\wedge\text{“ } F \;\Big|\; F \text{ „}\vee\text{“ } F \;\Big|\; \text{„[“ } A \text{ „]“ } F \;\Big|\; \text{„}\langle\text{“ } A \text{ „}\rangle\text{“ } F \;\Big|\; \text{„}\nu\text{“ } Z \text{ „.“ } F \;\Big|\; \text{„}\mu\text{“ } Z \text{ „.“ } F$$

Zu beachten ist die Unterscheidung freier und durch die Operatoren μ oder ν gebundene Variablen: freie Variablen werden als atomare Prädikate betrachtet, diese haben daher auch außerhalb einer Formel eine Bedeutung. Der Geltungsbereich gebundenener Variablen beschränkt sich dagegen auf den durch den Operator gebundenen Teil der Formel.

Die Tatsache „In der Kripke-Struktur $\mathcal{F}$ gilt die Formel F im Zustand $s \in S_{\mathcal{F}}$“ wird wie in Logiken üblich notiert als $s \models_{\mathcal{F}} F$ und die Negation als $s \not\models_{\mathcal{F}} F$. Der Übersichtlichkeit halber wird der Index $\mathcal{F}$ von nun an weggelassen, solange sich keine Mehrdeutigkeiten ergeben.

Dabei sei die informelle Bedeutung:

- $s \models Z$ (mit $Z \in \Phi$), wenn s das Prädikat „Z“ besitzt
- $s \models F_1 \wedge F_2$, wenn sowohl F_1 als auch F_2 in s wahr sind
- $s \models F_1 \vee F_2$, wenn F_1 oder F_2 (oder beide) in s wahr sind
- $s \models [A]F$, wenn F nach allen in s möglichen Aktionen aus A gilt
- $s \models \langle A \rangle F$, wenn F nach mindestens einer in s möglichen Aktion aus A gilt
- Die Rekursionsoperatoren $\mu Z.F$ und $\nu Z.F$ beschreiben andauernde Eigenschaften durch Fixpunkte; deren Bedeutung wird weiter unten erklärt.

Zusätzlich sind noch zwei Konstanten $\top$ (wahr) und $\bot$ (falsch) als geschlossene Ausdrücke definiert: (Beweis siehe unten)

$$\top := \nu Z.Z \qquad \bot := \mu Z.Z \qquad \text{mit } Z \in \Phi$$

Für die modalen Operatoren [ ] und $\langle\rangle$ werden noch einige Abkürzungen verwendet, um die Lesbarkeit zu erhöhen:

- $\Box \equiv [\Delta]$ (“nach allen möglichen Aktionen“)
- $\Diamond \equiv \langle \Delta \rangle$ („nach einer beliebigen Aktion“)
- $[\,\overline{A}\,] \equiv [\Delta \setminus A]$ („für alle Aktionen außer denen in A“)
- $\langle a \rangle \equiv \langle \{a\} \rangle$ (einelementige Aktionsmenge; „a ist möglich und danach gilt...“)

entsprechend für jeweils beide Operatoren und für alle Kombinationen.

5.2 Semantik

Um mit rekursiven modalen Eigenschaften, wie sie bei der Beschreibung andauernder Eigenschaften gebraucht werden, umzugehen, wird deren *intensionale Semantik* definiert. Dies ist hilfreich, um Formeln des μ-Kalküls gut verstehen und mit ihnen umgehen zu können.

Jede Formel F teilt die Zustände S einer Kripke-Struktur in zwei Teilmengen:

$$\|F\| = \{s \in S |\ s \models F\} \qquad S \setminus \|F\|$$

als die Menge von Zuständen $\|F\|$, in denen F wahr ist und $S \setminus \|F\|$, in denen sie nicht wahr ist. Damit ist die Intension für eine gegebene Kripke-Struktur $(S, s^0, next, V)$ wie folgt definiert:

$$\begin{array}{lcl}
\|Z\| & = & V(Z) \\
\|F \wedge G\| & = & \|F\| \cap \|G\| \\
\|F \vee G\| & = & \|F\| \cup \|G\| \\
\|[A]\|, \|\langle A \rangle\| & : & \mathcal{P}(S) \to \mathcal{P}(S) \\
\|[A]\|(X) & = & \{s \in S |\ \forall s' \in S.\ \forall a \in A.\ (s, a, s') \in next \to s' \in X\} \\
\|\langle A \rangle\|(X) & = & \{s \in S |\ \exists s' \in X.\ \exists a \in A.\ (s, a, s') \in next\} \\
\|[A]F\| & = & \|[A]\|(\|F\|) \\
\|\langle A \rangle F\| & = & \|\langle A \rangle\|(\|F\|) \\
\|\nu Z.F\| & = & gfp(\|F\|^Z) \\
\|\mu Z.F\| & = & lfp(\|F\|^Z)
\end{array}$$

Die Abbildungen $\|[\,]|$ und $\|\langle\rangle\|$ beschreiben die Semantik der modalen Operatoren $[\,]$ und $\langle\rangle$. Zu beachten ist dabei, dass diese Abbildungen – bildlich gesprochen – einen Zeitschritt zurück gehen: Wenn eine Formel F in einem Zustand s gilt, dann bezieht sich eine Formel $[A]F$ auf einen Vorgängerzustand von s (denn $[A]F$ heißt ja gerade „*nach* A gilt in allen *Folge*zuständen F). Wenn $X \subseteq S$ eine Menge von Zuständen ist, dann gilt $s \in \|[A]\|(X)$ genau dann, wenn alle A-Nachfolger von s in X liegen. Analog dazu gilt $s \in \|\langle A \rangle\|(X)$, wenn es einen A-Nachfolger von s gibt, der in X liegt.

Die Funktionen gfp und lfp werden im nächsten Abschnitt definiert, werden aber aus Gründen der Vollständigkeit hier schon verwendet.

Eine mengentheoretische Formulierung, wie sie die intensionale Semantik darstellt, ist für die manuelle Berechnung zwar sehr anschaulich, jedoch für die maschinelle Verifikation eher ungeeignet. Für letzere ist eine direkte Formulierung der Erfüllbarkeitsrelation $\models$ geeigneter, welche im allgemeinen *extensionale* Semantik genannt wird. Da diese aber hier nicht von Interesse ist, wird sie hier auch nicht aufgeführt; der interessierte Leser kann sie in [Sti92] finden.

5.3 Beschreibung andauernder Eigenschaften durch Fixpunkte

Durch Konjunktion, Disjunktion und die modalen Operatoren [] und $\langle\rangle$ (das entspricht der modalen Aussagenlogik) lassen sich nur lokale Eigenschaften eines Prozesses beschreiben. Es zeigt sich, dass (potentiell unendlich lange) andauernde Eigenschaften mit Hilfe von Fixpunktgleichungen beschreibbar sind.

Dabei sei zunächst der Begriff „Fixpunkt“ definiert:

Ein **Fixpunkt** einer Abbildung $f : X \to X$ ist ein Element $x^F \in X$ welches die Gleichung

$$f(x^F) = x^F$$

erfüllt.

5.3.1 Beispiel: Uhren

In einer Kripke-Struktur $\mathcal{F}_{Cl=}(\{Clock\} \cup \{Clock_i|\ i \in \mathbb{N}\}, s^0, next, V)$ seien folgende Prozesse gegeben, die das Verhalten von Uhren darstellen:

$Clock_0 = \ 0$

$Clock_n = \ send(tock) \to Clock_{n-1}$

$Clock = \ send(tock) \to Clock$

D. h., $Clock_n$ tickt genau n mal und bleibt dann stehen, während $Clock$ unendlich oft tickt.

Es gibt keine Formel A der modalen Aussagenlogik (das sind die obigen Formeln ohne die Operatoren μ und ν und mit den Konstanten $\top$ und $\bot$), die zwischen den beiden Eigenschaften „endlich tickend“ und „unendlich tickend“ unterscheiden kann, d. h., für die $\|A\| = \{Clock\}$ gilt. Der genaue Beweis dazu ist in [Sti92] nachzulesen.

Betrachtet man aber die rekursive Formel

$$Z =_{def} \langle send(tock)\rangle Z$$

mit $Z \in \Phi$ (intuitiv: „Die Eigenschaft Z ist, dass ein $send(tock)$ möglich ist und danach Z gilt“), dann gibt es zwei Lösungen, die deren Semantik, d. h. die Gleichung

$$\|Z\| = \|\langle send(tock)\rangle\|_{\mathcal{F}_{Cl}}(\|Z\|)$$

erfüllen, nämlich $\|Z\| = \varnothing$ und $\|Z\| = \{Clock\}$, wie man durch Einsetzen in die Definitionen leicht überprüft:

$$
\begin{aligned}
\|\langle send(tock)\rangle\|(\varnothing) &= \{s \in S|\ \underbrace{\exists s' \in \varnothing}_{\text{keine Lsg.}}\ .\ \exists a \in \{send(tock)\}.\ (s, a, s') \in next\} \\
&= \varnothing \\
\|\langle send(tock)\rangle\|(\{Clock\}) &= \{s \in S|\ \underbrace{\exists s' \in \{Clock\}.\ \exists a \in \{send(tock)\}.}_{\text{einsetzen der einzigen Lsg.}}\ (s, a, s') \in next\} \\
&= \{s \in S|\ (s, send(tock), Clock) \in next\} \\
&= \{Clock\}
\end{aligned}
$$

Andere Lösungen gibt es nicht: Sobald ein Element $Clock_k$ in X enthalten ist, so muss auch dessen $\langle send(tock)\rangle$-Nachfolger $Clock_{k-1}$ in X enthalten sein; dieser bedingt wiederum $Clock_{k-2} \in X$ usw. bis $Clock_0 \in X$. $Clock_0$ hat aber keinen $\langle send(tock)\rangle$-Nachfolger, deshalb kann es nicht Element des Fixpunktes sein, und deshalb alle anderen $Clock_n$-Prozesse auch nicht.

5.3.2 Kleinste und größte Fixpunkte

Wie oben demonstriert wurde, lassen sich Fixpunkte rekursiver modaler Gleichungen zur Charakterisierung von potentiell unendlich lang andauernden Eigenschaften nutzen. Von besonderem Interesse sind dabei der *kleinste* (lfp) und der *größte* (gfp) Fixpunkt einer Abbildung (relativ zur Ordnung $\subseteq$).

Ein allgemeines Resultat von Tarski und Knaster, was die Existenz und Eindeutigkeit dieser Fixpunkte sicherstellt, ist das **Fixpunkttheorem:**

Sei $g : \mathcal{P}(S) \to \mathcal{P}(S)$ eine in Bezug auf die $\subseteq$-Ordnung monotone Funktion. Dann besitzt g

- einen kleinsten Fixpunkt $lfp(g) = \bigcap\{X \subseteq S|\ g(X) \subseteq X\}$
- einen größten Fixpunkt $gfp(g) = \bigcup\{X \subseteq S|\ X \subseteq g(X)\}$

Siehe [Sti92] für den Beweis.

In einer Formel wie $\nu Z.V \wedge [send(tock)]Z$ mit $Z, V \in \Phi$ ist V als freie Variable zu betrachten, Z dagegen ist durch den Operator νZ „gebunden“. Um diese Bindung zu berücksichtigen und das Fixpunkttheorem anzuwenden, wird zunächst eine Notation definiert: Die Funktion

$$
f_{p \mapsto A}(x) := \begin{cases} A & \text{wenn } x = p \\ f(x) & \text{sonst} \end{cases}
$$

bezeichnet die Funktion, die das Argument p auf A abbildet und an allen anderen Stellen identisch mit $f(x)$ ist.

Nun kann $\|F\|^p : \mathcal{P}(S) \to \mathcal{P}(S)$ definiert werden als

$$\|F\|^p(X) := \|F\|_{p \mapsto X}$$

Somit wird nach der Semantikdefinition in 5.2 die durch den jeweiligen Operator gebundene Variable rekursiv wieder auf das Funktionsargument abgebildet (was dem Einsetzen der Funktion in sich selbst entspricht).

Diese Abbildungen sind monoton, wie in [Sti92] bewiesen wird, was die Anwendung des Fixpunkttheorems erlaubt.

Diese Definition vervollständigt auch die Semantikdefinition in 5.2.

Nun kann insbesondere die Semantik der oben definierten Symbole $\top$ und $\bot$ berechnet werden:

$$\begin{aligned}
\|\bot\| &= \|\mu Z.Z\| \\
&= lfp(\|Z\|^Z) \\
&= lfp(\lambda X \subseteq S.\ \|Z\|_{Z \mapsto X}) \\
&= lfp(\lambda X \subseteq S.\ X) \\
&= \bigcap\{X \subseteq S \mid \underbrace{X \subseteq X}_{\text{wahr}}\} \\
&= \bigcap\{X \subseteq S\} \\
&= \varnothing
\end{aligned}
\qquad
\begin{aligned}
\|\top\| &= \|\nu Z.Z\| \\
&= gfp(\|Z\|^Z) \\
&= gfp(\lambda X \subseteq S.\ \|Z\|_{Z \mapsto X}) \\
&= gfp(\lambda X \subseteq S.\ X) \\
&= \bigcup\{X \subseteq S \mid \underbrace{X \subseteq X}_{\text{wahr}}\} \\
&= \bigcup\{X \subseteq S\} \\
&= S
\end{aligned}$$

Somit ist die Interpretation von $\top$ als „wahr“ und $\bot$ als „falsch“ gerechtfertigt.

5.4 Entfaltung

Nun sind alle Definitionen vorhanden, um die Semantik jeder Formel des μ-Kalküls zu berechnen. Nichtdestotrotz sind diese Definitionen nicht sehr gut intuitiv erfassbar: schon die Bedeutung relativ kleiner Formeln wie $\nu Z.(F \wedge \Box Z)$ oder $\mu Z.\Diamond\top \wedge [\overline{x = 1}]Z$ ist daran nicht mehr gut einsichtig.

Deshalb wird hier ein Verfahren vorgestellt, mit denen man sich die Bedeutung der Formeln schrittweise durch „auffalten“ klar machen kann. Die Entfaltungen

$$\nu Z^k.F \quad \text{und} \quad \mu Z^k.F$$

sind wie folgt definiert:

$$
\begin{array}{lcll}
\nu Z^0.F & = & \top & \\
\nu Z^{k+1}.F & = & F[Z := \nu Z^k.F] & \text{(Substitution von } Z\text{)} \\
 & & & \\
\mu Z^0.F & = & \bot & \\
\mu Z^{k+1}.F & = & F[Z := \mu Z^k.F] & \text{(Substitution von } Z\text{)}
\end{array}
$$

Mit Hilfe des folgenden Satzes (Beweis in [Sti92])

$$
\begin{array}{l}
s \models \nu Z.F \;\Leftrightarrow\; \forall k \in \mathbb{N}.\; s \models \nu.Z^k.F \\
s \models \mu Z.F \;\Leftrightarrow\; \exists k \in \mathbb{N}.\; s \models \mu.Z^k.F
\end{array}
$$

---

ist nun die Bedeutung von Fixpunktformeln direkter darstellbar.

Zur Illustration wird zunächst noch einmal das Uhrenbeispiel herangezogen. Durch Auffalten erschließt sich die Bedeutung von $\nu Z.\langle send(tock)\rangle Z$ wie folgt:

$$
\begin{array}{lcl}
\nu Z^0.\langle send(tock)\rangle Z & = & \top \\
\nu Z^1.\langle send(tock)\rangle Z & = & \langle tock := true\rangle \top \\
 & \vdots & \\
\nu Z^k.\langle tock := true\rangle Z & = & \underbrace{\langle tock := true\rangle \cdots \langle send(tock)\rangle}_{k-\text{mal}} \top \\
 & \vdots &
\end{array}
$$

Diese Formeln sind wegen des vorherigen Satzes und dem Operator ν als konjunktiv zu betrachten, d. h.: für alle k muss gelten, dass $\langle send(tock)\rangle$ mindestens k-mal möglich ist. Dies trifft, wie schon gezeigt, nur für $Clock$, aber für kein $Clock_n$ zu. Mit dieser Formelgestalt können also Sicherheitseigenschaften über Aktionen definiert werden.

Um auch eine Lebendigkeitseigenschaft zu demonstrieren und auch ein Beispiel für eine Sicherheitseigenschaft über Zuständen zu haben, werden nun noch die am Anfang des Abschnittes angeführten Formeln entfaltet:

$$
\begin{array}{lcll}
\nu Z^0.(F \wedge \Box Z) & = & \top & \\
\nu Z^1.(F \wedge \Box Z) & = & F \wedge \Box\top & (F \text{ gilt im akt. Zustand}) \\
\nu Z^2.(F \wedge \Box Z) & = & F \wedge \Box(F \wedge \Box\top) & (F \text{ gilt im akt. Zustand und} \\
 & & & \text{für alle Nachf. gilt } F) \\
\nu Z^3.(F \wedge \Box Z) & = & F \wedge \Box(F \wedge \Box(F \wedge \Box\top)) & (F \text{ gilt im akt. Zustand und} \\
 & & & \text{für alle Nachf. gilt } F \text{ und für} \\
 & & & \text{alle deren Nachfolger gilt } F) \\
 & \vdots & &
\end{array}
$$

Führt man dies weiter, so ergibt sich „F gilt in allen möglichen Zuständen“.

$$\begin{aligned}
\mu Z^0.\Diamond\top \wedge [\,\overline{x=1}\,]Z) &= \bot \\
\mu Z^1.\Diamond\top \wedge [\,\overline{x=1}\,]Z) &= \Diamond\top \wedge [\,\overline{x=1}\,]\bot \\
\mu Z^2.\Diamond\top \wedge [\,\overline{x=1}\,]Z) &= \Diamond\top \wedge [\,\overline{x=1}\,](\Diamond\top \wedge [\,\overline{x=1}\,]\bot) \\
&\vdots
\end{aligned}$$

(es ist eine Aktion möglich und andere Aktionen als $x = 1$ führen zum Fehlschlag)

(es ist eine Aktion möglich und für alle Aktionen außer $x = 1$ muss eine Aktion möglich sein, für die gilt: nach ihr muss eine Aktion möglich sein...)

Die Formeln sind wegen μ disjunktiv zu betrachten, also eine muss gelten. Dies kann man also interpretieren als Lebendigkeitseigenschaft „Das Ereignis $x = 1$ muss irgendwann eintreffen“.

5.5 Formulierung von Eigenschaften

Formeln des μ-Kalküls sind sehr ausdrucksstark, aber auch unintuitiv. Deshalb werden für oft gebrauchte Klassen von Eigenschaften Makrodefinitionen zur Verfügung gestellt, die bei der Formulierung von Anforderungen durch Konstruktion von Formeln sehr hilfreich sind.

Die hier definierten Operatoren stellen nur wenige Beispiele der Mächtigkeit des μ-Kalküls dar. Eine konkrete Implementation kann noch andere Makrooperatoren implementieren.

5.5.1 Sicherheits- und Lebendigkeitseigenschaften

Zwei sehr allgemeine Eigenschaftsklassen sind die Sicherheits- und Lebendigkeitseigenschaften. Eine Sicherheitseigenschaft drückt aus, dass niemals etwas „schlechtes“ passiert (Temperatur zu hoch, Deadlock), während eine Lebendigkeitseigenschaft fordert, dass irgendwann etwas „gutes“ passieren muss (Algorithmus terminiert, Zugriff auf Ressource gewährt).

Zur einfachen Beschreibung werden die entsprechenden Operatoren aus CTL übernomen. Ein „Lauf“ ist dabei eine (möglicherweise unendliche) Folge von Aktionen oder Ereignissen, die ein Prozess durchführt:

- $\mathbb{A}$ für alle Läufe
- $\mathbb{E}$ es gibt einen Lauf
- $\mathbb{G}$ für alle Zustände (eines bestimmten Laufs)
- $\mathbb{F}$ es gibt einen Zustand (eines bestimmten Laufs)

Diese können dann folgendermaßen interpretiert werden:

$s \models \mathbb{AG}.F$ wenn $s \models F$ und für alle von s erreichbaren Zustände s' aller möglichen Läufe gilt $s' \models F$. (**starke Sicherheitseigenschaft**)

$s \models \mathbb{EG}.F$ es gibt einen in s beginnenden Lauf, auf dem für alle Zustände s' gilt: $s' \models F$ (**schwache Sicherheitseigenschaft**)

$s \models \mathbb{AF}.F$ für alle in s beginnenden Läufe gibt es einen Zustand s' mit $s' \models F$, d. h. F *wird* irgendwann erfüllt sein (**starke Lebendigkeitseigenschaft**)

$s \models \mathbb{EF}.F$ es gibt einen in s beginnenden Lauf, auf dem es einen Zustand s' mit $s' \models F$ gibt, d. h. F *kann* irgendwann erfüllt sein (**schwache Lebendigkeitseigenschaft**)

Diese Operatoren können im μ-Kalkül wie folgt ausgedrückt werden:

$$\begin{array}{rcl} \mathbb{AG}.F & \equiv & \nu Z.F \wedge \Box Z \\ \mathbb{EG}.F & \equiv & \nu Z.F \wedge (\Box\bot \vee \Diamond Z) \\ \mathbb{AF}.F & \equiv & \mu Z.F \vee (\Diamond\top \wedge \Box Z) \\ \mathbb{EF}.F & \equiv & \mu Z.F \vee \Diamond Z \end{array}$$

Durch Auffalten kann man sich von der Bedeutung überzeugen (mit $\mathbb{AG}$ wurde das bereits demonstriert).

Solche Eigenschaften sollen manchmal nicht über Zustandseigenschaften, sondern mit Aktionen definiert werden, z. B. „Die Aktionen aus A dürfen nie passieren“. Dies läßt sich äquivalent ausdrücken als „ein Zustand, in dem Aktionen aus A nicht möglich sind“, was im μ-Kalkül als $[A]\bot$ ausgedrückt wird.

Entsprechendes läßt sich „Eine Aktion aus A ist möglich“ äquivalent ausdrücken als „ein Zustand, in dem Aktionen aus A möglich sind“: $\langle A\rangle\top$. Die strengere Form „eine der Aktionen aus A *muss* eintreffen“ ist äquivalent zu „es ist eine Aktion möglich und andere Aktionen als die in A dürfen nicht passieren“: $\Diamond\top \wedge [\overline{A}]\bot$.

Da diese Forderung in Form der Lebendigkeitseigenschaft „Eine Aktion aus A muss irgendwann eintreffen“ sehr allgemein gebräuchlich ist, bekommt sie ein eigenes Makro:

$$\mathbb{H}(A) = \mu Z.\Diamond\top \wedge [\,\overline{A}\,]Z$$

Diese Bausteine sind für sich allein schon sehr nützlich. Durch ihre Kombination lassen sich jedoch noch andere, komplexere Eigenschaften ausdrücken. Zum Beispiel beschreibt $\mathbb{AG}.\mathbb{H}(a)$ die starke Sicherheitseigenschaft: für jeden Lauf gilt in jedem Zustand, dass die Aktion a irgendwann eintrifft. Dies bedeutet also: die Aktion a wird unendlich oft ausgeführt.

5.5.2 Until-Eigenschaften

Eine andere Klasse von Eigenschaften ist unter dem Namen „until-Eigenschaften“ bekannt: diese beschränken die Gültigkeit einer Formel in Abhängigkeit des (Nicht-) Eintreffens einer anderen Anforderung.

Die Eigenschaft „F ist wahr bis G wahr wird“ wird als F until G bezeichnet und läßt sich im μ-Kalkül als

$$F \text{ until } G \equiv \mu Z.\, G \vee (F \wedge \Diamond\top \wedge \Box Z)$$

Als Lebendigkeitseigenschaft bedingt diese Formel, dass G irgendwann wahr werden muss. Soll dies nicht der Fall sein, muss daraus eine Sicherheitseigenschaft gemacht werden. Diese Anforderung „F ist wahr, solange G nicht eintrifft“ wird bezeichnet als

$$F \text{ unless } G \equiv \nu Z.\, G \vee (F \wedge \Box Z)$$

5.5.3 Echtzeiteigenschaften

Oft ist die Aussage $\mathbb{H}(a)$ – “Aktion a muss irgendwann eintreffen“ – zu schwach. Statt dessen fordert man, dass a auch in einem bestimmten Zeitraum, d. h. im spätestens n-ten Schritt passiert. Zu diesem Zweck wird das Makro $\mathbb{H}(a, n)$ eingeführt, das genau diese Aussage formalisiert. Dabei bedeutet „im 1. Schritt“, dass a sofort stattfindet, „im 2. Schritt“, dass a sofort oder nach dem folgenden Ereignis auftritt usw.

Als Grundlage kann die Definition von $\mathbb{H}(A)$ herangezogen werden, die dann auf die ersten n Entfaltungen beschränkt wird:

$$\begin{aligned} \mathbb{H}(A, 0) &\equiv \bot \\ \mathbb{H}(A, n+1) &\equiv \Diamond\top \wedge [\,\overline{A}\,]\mathbb{H}(A, n) \end{aligned}$$

5.6 Negation

5.6.1 Voraussetzungen

Die Negationsoperation kommt in der Definition des μ-Kalküls nicht vor und wurde bis jetzt auch nicht erwähnt. Das hat den Grund, weil man die Verwendung der Negation im μ-Kalkül nicht allgemein erlauben darf, weil sonst Fixpunktformeln nicht mehr monoton sein müssen.

Ein einfaches Gegenbeispiel ist die Gleichung $Z = \neg Z$; Ein Fixpunkt X, der diese Gleichung erfüllt, müsste der Eigenschaft $X = S \setminus X$ genügen, was bei $S \neq \varnothing$ unmöglich ist.

Die Verwendung der Negation ist möglich unter der Bedingung, dass durch Fixpunktoperatoren gebundene Variablen unnegiert auftreten. Stellt man sich die Entfaltung vor, bedeutet das also, dass die Formel auf jeder Stufe der Entfaltung gleich bleibt und nicht (logisch) „alterniert“.

5.6.2 Normalform

Da Beweise mit Negationen i. A. schwierig zu führen sind und das unten vorgestellte Beweiskalkül die Negation auch nicht enthält (außer bei atomaren Prädikaten), bringt man Formeln vor der Verifikation in die Negationsnormalform (NNF), d. h. dass Negation nur vor atomaren Prädikaten erlaubt ist. Die folgenden Regeln erlauben – rekursiv angewandt – die äquivalente Transformation einer μ-Formel mit Negation in die entsprechende NNF-Formel:

$$\begin{array}{rcl} \neg\top & \equiv & \bot \\ \neg(F \wedge G) & \equiv & \neg F \vee \neg G \\ \neg[A]F & \equiv & \langle A\rangle\neg F \\ \neg\mu Z.\, F & \equiv & \nu Z.\, \neg(F[Z := \neg Z]) \end{array} \qquad \begin{array}{rcl} \neg\bot & \equiv & \top \\ \neg(F \vee G) & \equiv & \neg F \wedge \neg G \\ \neg\langle A\rangle F & \equiv & [A]\neg F \\ \neg\nu Z.\, F & \equiv & \mu Z.\, \neg(F[Z := \neg Z]) \end{array}$$

Die Beweise ergeben sich aus der Äquivalenz der Semantik und sind in [Sti92] zu finden.

Einige Worte zur Erklärung der Transformation der Fixpunktoperatoren: Da zuerst die gebundene Variable Z in F durch ihr Negat $\neg Z$ ersetzt wird und anschließend die gesamte Formel F negiert werden muss, tritt Z schließlich wieder unnegiert auf.

5.6.3 Beispiel

Die Negation der schwachen Sicherheitseigenschaft „es gibt einen Lauf, in dem die Aktion *crash* nie ausgeführt wird“ demonstriert das Verfahren:

$$\begin{array}{rll} & \neg(\mathbb{E}\mathbb{G}.[crash]\bot) & \\ \equiv & \neg(\nu Z.[crash]\bot \wedge (\Box\bot \vee \Diamond Z)) & \\ \equiv & \mu Z.\neg([crash]\bot \wedge (\Box\bot \vee \Diamond\neg Z)) & \text{(Subst. der gebundenen Var. durch Negat)} \\ \equiv & \mu Z.\neg([crash]\bot) \vee \neg(\Box\bot \vee \Diamond\neg Z) & \\ \equiv & \mu Z.\langle crash\rangle(\neg\bot) \vee (\neg(\Box\bot) \wedge \neg(\Diamond\neg Z)) & \\ \equiv & \mu Z.\langle crash\rangle\top \vee (\Diamond(\neg\bot) \wedge \Box(\neg\neg Z)) & \\ \equiv & \mu Z.\langle crash\rangle\top \vee (\Diamond\top \wedge \Box Z) & \\ \equiv & \mathbb{A}\mathbb{F}.\langle crash\rangle\top & \end{array}$$

Neben der Funktion als Beispiel offenbart diese Transformation einen interessanter Zusammenhang: wie schon auf der sprachlichen Ebene ersichtlich (die Negation würde bedeuten „es gibt keinen Lauf, in dem die Aktion *crash* nie ausgeführt wird“, sie wird also in jedem Lauf ausgeführt), stellt die Negation einer schwachen Sicherheitseigenschaft eine starke Lebendigkeitseigenschaft dar. Tatsächlich gilt für alle Formeln F

$$\neg\mathbb{A}\mathbb{F}.F \equiv \mathbb{E}\mathbb{G}.\neg F \qquad\qquad \neg\mathbb{A}\mathbb{G}.F \equiv \mathbb{E}\mathbb{F}.\neg F$$

wie man sich durch die obigen Transformationsregeln leicht überzeugt.

5.6.4 Weitere Operatoren

Mit Hilfe der Negation lassen sich nun alle aus der Aussagenlogik bekannten Operationen definieren. Sehr häufig gebraucht wird die Implikation $A \rightarrow B$ ("Wenn A dann B", if A then B):

$$A \rightarrow B := \neg A \vee B$$

Analog dazu können auch Äquivalenz, exklusives Oder usw. definiert werden.

5.7 Deadlock

Bediensysteme mit begrenzten Ressourcen zu neigen zu Deadlocks. Diese Eigenschaft kann unabhängig von einem konkreten System beschrieben werden. In CCMB liegt ein Deadlock vor, wenn ein Netzwerk in einem Zustand ist, in dem es nur noch eine (unendliche oder endliche) Folge von „ticks" durchführen kann oder überhaupt keine Transitionen mehr möglich sind (im 0-Prozess). Anders ausgedrückt liegt ein Deadlock vor, wenn außer „tick" nichts mehr möglich ist:

$$Deadlock \equiv \mathbb{AG}.[\overline{tick}]\bot$$

Umgekehrt gilt also: ein Zustand ist kein Deadlock, wenn es einen Lauf gibt, auf dem eine Transition verschieden von „tick" möglich ist:

$$NotDead \equiv \neg Deadlock \equiv \mathbb{EF}.\langle\, \overline{tick}\, \rangle\top$$

Damit das ganze Netzwerk Deadlock-frei ist, muss dies für alle Zustände gelten und somit die Forderung $\mathbb{AG}.NotDead$ (bzw. $\neg\mathbb{EF}.Deadlock$) erfüllen.

5.8 Präzedenzregeln

Auch für die logischen Formeln werden für die Operatoren Vorrang- und Assoziativitätsregeln festgelegt, um unnötige Klammersetzung möglichst zu vermeiden:

Operator	Priorität	Assoziativität
$\neg$	1	ja
$[\,]$, $\langle\,\rangle$	2	ja
$\wedge$	3	links
$\vee$	4	links
$\rightarrow$	5	rechts
until, unless	6	nein*
μ, ν	7	ja
$\mathbb{H}$	8	nein
$\mathbb{AF}, \mathbb{AG}, \mathbb{EF}, \mathbb{EG}$	9	ja

* Es wäre natürlich möglich, hier willkürlich eine Assoziativität zu definieren. Da es aber keine intuitive Richtung gibt, sollten diese Operatoren bei Aneinanderreihung immer geklammert werden.

Abb. 5.1: Präzedenz und Assoziativität der Operatoren des μ-Kalküls

5.9 Verifikationskalkül

Die Berechnung von Fixpunkten mit Hilfe der Semantik ist im Allgemeinen keine einfache Aufgabe. Wünschenswert wäre eine Verifikationsmethode, die direkt auf der syntaktischen Ebene arbeitet und anschaulich ist, was den „Umweg“ über die Semantik erspart.

Dieser Aufgabe wird ein Tableau-Beweissystem gerecht, das nun vorgestellt wird. Dieses ist aus Sequenzen der Form $s \vdash F$ aufgebaut. Die Relation $\vdash$ ist dabei als syntaktisches Analogon zur Erfüllbarkeitsrelation $\models$ (welche auf der semantischen Ebene definiert ist) anzusehen.

Um zu testen, ob ein Zustand s einer Kripke-Struktur die Eigenschaft F besitzt, versucht man, ein erfolgreiches Tableau aufzustellen. Ein Tableau für F ist ein Beweisbaum, dessen Wurzel mit $s \vdash F$ markiert ist, dessen Knotenerweiterungen den Regeln für die $\vdash$-Relation folgen und dessen Blätter mit (wahren oder falschen) terminalen Sequenzen versehen sind. Ein Tableau wird **erfolgreich** genannt, wenn sämtliche seiner Blätter wahre terminale Sequenzen sind.

Eine Tableauregel hat die folgende Form:

$$\textit{Regelname}: \qquad \frac{s_1 \vdash F_1 \ \ldots \ s_n \vdash F_n}{s \vdash F}$$

Das bedeutet, dass aus den Prämissen $s_i \vdash F_i$ die Implikation $s \vdash F$ gefolgert werden kann. Im Tableau ist das folgendermaßen anzuwenden: besitzt ein Knoten die Form $s \vdash F$, dann kann er um alle Nachfolgerknoten $s_1 \vdash F_1$ bis $s_n \vdash F_n$ (nach oben) erweitert werden. Um eine Formel zu beweisen, die ja die „äußerste“ Implikation ist, sind die Tableauregeln also von unten nach oben zu befolgen.

Abb. 5.2 zeigt die Tableauregeln für Formeln des μ-Kalküls in Negationsnormalform.

Das Tableauverfahren ist korrekt, was folgender Satz sicherstellt:

Korrektheit des Tableau-Beweissystems:

Wenn $s \vdash F$ ein erfolgreiches Tableau besitzt, dann gilt $s \models F$.

Die Umkehrung des Satzes gilt aber nur eingeschränkt:

Vollständigkeit des Tableau-Beweissystems:

Wenn $s \models F$ gilt und S endlich ist, dann besitzt $s \vdash F$ ein erfolgreiches Tableau.

(Die Beweise sind in [Sti92] zu finden und werden hier nicht abgedruckt.)

Nichtterminale Regeln:

Regel $\wedge$: $$\frac{s \vdash F \quad s \vdash G}{s \vdash F \wedge G}$$

Regel $\vee_1$: $$\frac{s \vdash F}{s \vdash F \vee G}$$

Regel $\vee_2$: $$\frac{s \vdash G}{s \vdash F \vee G}$$

Regel $[A]$: $$\frac{s_1 \vdash F \ \ldots \ s_n \vdash F}{s \vdash [A]F} \qquad \{s_1, \ldots, s_n\} = \{s' |\ (s, a, s') \in next \wedge a \in A\}$$

Regel $\langle A \rangle$: $$\frac{s' \vdash F}{s \vdash \langle A \rangle F} \qquad (s, a, s') \in next \wedge a \in A$$

Regel σZ: $$\frac{s \vdash U}{s \vdash \sigma Z.F} \qquad \text{für eine neue Variable } U \text{ mit } U =_{def} \sigma Z.F$$

Regel U: $$\frac{s \vdash F[Z := U]}{s \vdash U} \qquad \text{wenn } U \text{ definiert wurde als } U = \sigma Z.F$$

Wahre (erfolgreiche) Terminale:

1. $s \vdash \top$
2. $s \vdash \varphi$ und $s \in V(\varphi)$ bzw. $s \vdash \neg\varphi$ und $s \notin V(\varphi)$
3. $s \vdash [A]G$ und $\{s' |\ (s, a, s') \in next \wedge a \in A\} = \varnothing$
4. $s \vdash U$ und $U = \nu Z.G$ und es gibt einen Knoten darunter (auf dem Pfad zur Wurzel), der ebenfalls mit $s \vdash U$ ausgezeichnet ist

Falsche (fehlschlagende) Terminale:

1. $s \vdash \bot$
2. $s \vdash \varphi$ und $s \notin V(\varphi)$ bzw. $s \vdash \neg\varphi$ und $s \in V(\varphi)$
3. $s \vdash \langle A \rangle G$ und $\{s' |\ (s, a, s') \in next \wedge a \in A\} = \varnothing$
4. $s \vdash U$ und $U = \mu Z.G$ und es gibt einen Knoten darunter (auf dem Pfad zur Wurzel), der ebenfalls mit $s \vdash U$ ausgezeichnet ist

Abb. 5.2: Tableau-Regeln für das μ-Kalkül

5.10 Intuition des Tableauverfahrens

Das Tableauverfahren basiert darauf, eine zu beweisende Formel schrittweise in ihre atomaren Bestandteile zu zerlegen und deren Gültigkeit aus dem Modell zu ermitteln. Die meisten Regeln sind sicherlich recht intuitiv.

Die Ableitungen für $\top$, $\bot$, $\wedge$ und $\vee$ entsprechen denen der Aussagenlogik, auch die Terminale für die atomaren Eigenschaften sind offensichtlich.

Die $\langle\,\rangle$- und $[\,]$-Regeln erledigen den Beweis für einen bzw. alle Nachfolgerzustände. Auch deren Definition sollte einsichtig sein.

Die Intuition der σZ- und U-Regeln ist die ausschöpfende Durchmusterung des Zustandsraumes für die Fixpunktoperatoren. Trifft der Beweis auf einen μ bzw. ν-Operator, dann wird für diese Formel in der σZ-Regel eine neue Variable eingefürt; so kann sich der Tableaubeweis „merken“, welche Formeln im Beweis schon einmal aufgetreten sind; trifft der Beweis irgendwann erneut auf diese einzelne Variable im selben Zustand, dann wurde ein kompletter Zyklus von Aktionen durch den Zustandsraum geprüft und die Gültigkeit kann anhand der terminalen Regeln entschieden werden.

Erreicht der Beweis hingegen eine einzelne U-Variable in einem noch nicht aufgetretenen Zustand, dann führt die U-Regel einen Entfaltungsschritt wie in Abschnitt 5.4 beschrieben durch.

6 Demonstration an den Anwendungsfällen

„Longum iter est per praecepta,
breve et efficax per exempla“

Lucius Annaeus Seneca (ca. 4 v. Chr - 65 n. Chr.),
römischer Politiker, Rhetor, Philosoph und Schriftsteller

Dieses Kapitel zeigt die Anwendung der eingeführten Formalismen mit Beispielen. Dazu werden die in Kapitel 2 beschriebenen Anwendungsfälle verwendet.

Der erste Schritt der Formalisierung ist immer die Modellierung der Devices und Stubs. Da für die Verifikation der Eigenschaften jedoch nur die Stubs interessant sind, wird hier auf eine Modellierung der Devices verzichtet.

Sind alle benötigten Stubs definiert, so wird das Netzwerk erstellt, indem die Stubs instantiiert und gebunden werden. Die in Kapitel 2 verbal beschriebenen Anforderungen werden nun in Formeln des μ-Kalküls ausgedrückt und die Anwendung des Tableau-Verfahrens aus Abschnitt 5.9 an einigen Anforderungen demonstriert. Da die Anwendungsfälle Fehler beinhalten, wird dies zunächst fehlschlagen; aus der „Antwort“ der Verifikation wird dann ein Gegenbeispiel konstruiert, welches zur Korrektur des Systems hilfreich ist.

Da der zweite Anwendungsfall (Echtzeitanalyse) am einfachsten – im Sinne der Tableaugröße – zu verifizieren ist, wird mit diesem begonnen.

6.1 Echtzeitanalyse

6.1.1 Stubs

`LAMP` – Lampe

Mit nur zwei von außen beobachtbaren Zuständen (an und aus) ist eine Lampe ein sehr einfacher Stub. Die folgende Abbildung zeigt eine einfache Beschreibung als Prozessbeschreibung und dessen entsprechendes Zustandsdiagramm. Die Kästchen mit den abgerundeten Ecken stellen dabei die Zustände dar, die gestrichelten Kästen zeigen, welche atomaren Prädikate in dem verbundenen Zustand gelten.

```
stub LAMP {
    portin bool on

    LAMP := [nolight] (on=true -> ON |
                       on=false -> LAMP)

    ON := [light] (on=false -> LAMP |
                   on=true -> ON)
}
```

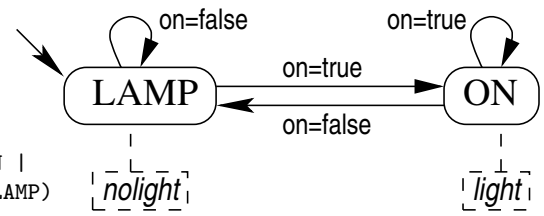


Abb. 6.1: Prozessbeschreibung und enstprechendes Zustandsdiagramm für LAMP

SENSOR – Temperatursensor

Der Sensor wird sehr einfach gehalten. Er tut nichts anderes als beständig den Ausgangswert `temp` zu aktualisieren:

```
stub SENSOR {
    portout int temp

    SENSOR := send(temp) -> tick(2) -> SENSOR
}
```

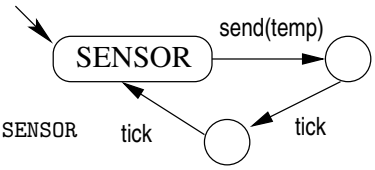


Abb. 6.2: Prozessbeschreibung und entsprechendes Zustandsdiagramm für SENSOR

Zu beachten ist, dass Sensor nicht als `SENSOR = send(temp) -> SENSOR` definiert werden sollte, da er dann pausenlos Nachrichten schicken und den Kommunikationsbus total blockieren würde. Bei der Verifikation fallen solche Fehler allerdings sofort auf.

Controller

Es sollen zwei Alternativen zur Verfügung stehen: Die Variante `SLOW_CTR` benötigt ein oder zwei ticks zur Auswertung des Sensorergebnisses und danach den Alarmzustand senden; dagegen braucht die schnelle Variante `FAST_CTR` dafür immer nur einen tick:

```
stub SLOW_CTR {
    portin int temp
    portout bool alarm

    SLOW_CTR := update(temp) -> tick ->
        ( COUT | tick -> COUT )

    COUT := (alarm:=true | alarm := false) -> SLOW_CTR
}
```

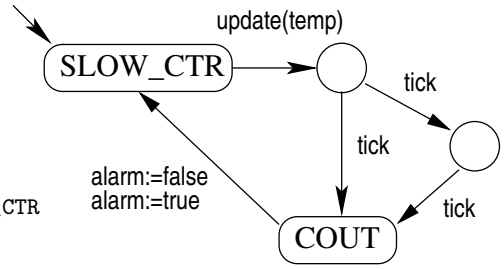


Abb. 6.3: Prozessbeschreibung und entsprechendes Zustandsdiagramm für SLOW_CTR

```
stub FAST_CTR {
    portin int temp
    portout bool alarm

    FAST_CTR := update(temp) -> tick -> COUT

    COUT := ( alarm:=true | alarm := false ) -> FAST_CTR
}
```

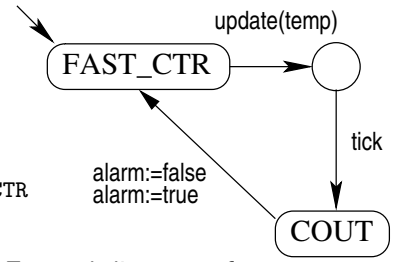


Abb. 6.4: Prozessbeschreibung und entsprechendes Zustandsdiagramm für `FAST_CTR`

6.1.2 Netzwerk

Nun wird zur Bildung des Netzwerkes TC von jedem definierten Stub ein Exemplar instantiiert und die Bindung entsprechend Abb. 2.2 vorgenommen:

$$Stubs_{TC} = \{sens : SENSOR, ctr : SLOW_CTR, lamp : LAMP\}$$

$$\begin{aligned} bind_{TC}(sens.temp) &= \{ctr.temp\} \\ bind_{TC}(ctr.alarm) &= \{lamp.on\} \end{aligned}$$

6.1.3 Anforderungen

Die Echtzeitanforderung sei, dass nach einer Aktualisierung der Temperatur im spätestens 2. tick das Alarmsignal $A := \{lamp.on = true, lamp.on = false\}$ ausgelöst wird:

$$[send(sens.temp)].\mathbb{H}(A, 2)$$

(Für alle Folgezustände von $send(sens.temp)$ gilt: im spätestens 2. tick erfolgt das Ereignis $lamp.on = true$ oder $lamp.on = false$.)

Um den Beweisbaum noch beherrschbar klein zu halten, wird diese Forderung hier nicht als Sicherheitseigenschaft formuliert, sie bezieht sich also nur auf den Startzustand. Mit Hilfe der Implementation ist dies jedoch durchführbar, die Formel erhält dazu noch den Präfix $\mathbb{AG}$.

6.1.4 Verifikation

Wie man bei diesem kleinen Beispiel auch durch Hinschauen sieht, wird mit `SLOW_CTR` das Ereignis `on:=true` bzw. `on:=false` im 2. oder erst im 3. tick nach Aktualisierung der Temperatur ausgelöst. Der Tableaubeweis bestätigt dies:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{TC2 \vdash \top}}{TC1 \vdash \Diamond\top}\,\langle\rangle,\star^2 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{TC3 \vdash \top}}{TC2 \vdash \Diamond\top}\,\langle\,\rangle,\star^3 \quad \dfrac{\lightning}{TC3 \vdash \bot}}{TC2 \vdash \Diamond\top \wedge [\overline{A}]\bot}\,\langle\,\rangle,\star^4}{TC2 \vdash \Diamond\top \wedge [\overline{A}]\bot}\,\wedge,\bigstar}{TC2 \vdash \mathbb{H}(A,1)}\,Def.\ \mathbb{H}}{TC1 \vdash [\overline{A}]\mathbb{H}(A,1)}\,[\,],\star^2}{TC1 \vdash \Diamond\top \wedge [\overline{A}]\mathbb{H}(A,1)}\,\wedge}{TC1 \vdash \mathbb{H}(A,2)}\,Def.\ \mathbb{H}}{TC \vdash [send(sens.temp)].\mathbb{H}(A,2)}\,[\,],\star^1
$$

$\star^1$ Bindung löst $update(ctr.temp)$ bei $send(sens.temp)$ aus, andere Ereignisse sind nicht möglich
$\rightsquigarrow TC1 := \{sens : tick(2) \rightarrow SENS, lamp : LAMP, ctr : tick \rightarrow (tick \rightarrow COUT|COUT)\}$

$\star^2$ (nur) tick ist möglich
$\rightsquigarrow TC2 := \{sens : tick \rightarrow SENS, lamp : LAMP, ctr : (tick \rightarrow COUT|COUT)\}$

$\star^3$ tick, $ctr.alarm := true$, $ctr.alarm := false$ (1. Aktionen von COUT), $lamp.on = true$ und $lamp.on = false$ (durch Bindung) sind möglich
$\rightsquigarrow TC3 := \{sens : SENS, lamp : LAMP, ctr : COUT)\}$ (nach tick)

$\star^4$ $\overline{A} = \{tick\}$, d. h. außer $ctr.alarm := \ldots$ (die A auslösen), ist auch noch tick möglich $\rightarrow$ $TC3$

Es gab bei diesem Tableau keine Alternativen (durch Operatoren $\vee$ oder $\langle\,\rangle$) und das mit $\lightning$ bezeichnete Blatt ist ein fehlschlagendes Terminal. An der mit $\star^4$ markierten Stelle sieht man auch gleich, wo das Problem liegt: an diesem Punkt sind nicht nur Aktionen möglich, die A auslösen, sondern auch eine andere (`tick`).

Setzt man stattdessen für ctr `FAST_CTR` ein, dann ist

$$TC2 := \{sens : tick \rightarrow SENS, lamp : LAMP, ctr : COUT\}$$

und das Tableau verändert sich ab der Stelle $\bigstar$ wie folgt:

$$
\dfrac{\dfrac{\dfrac{\checkmark}{TC3 \vdash \top}}{TC2 \vdash \Diamond\top}\,\langle\,\rangle,\star^1 \quad \dfrac{\checkmark}{TC2 \vdash [\overline{A}]\bot}\,\langle\,\rangle,\star^2}{TC2 \vdash \Diamond\top \wedge [\overline{A}]\bot}\,\wedge
$$

$\star^1$ $ctr.alarm := true$ ist möglich
$\rightsquigarrow TC3 := \{sens : SENS, lamp : ON, ctr : FAST_CTR\}$

$\star^2$ Nun sind nur noch $ctr.alarm := true/false$ möglich welche durch die Bindung A auslösen. Somit ist $\overline{A}$ leer und damit $[\overline{A}]\bot$ ein erfolgreiches Terminal.

Somit erfüllt `FAST_CTR` die Echtzeitanforderung.

6.2 Ungeeignete Komponente

6.2.1 Stubs

Hier wird die schon im vorigen Abschnitt definierte Lampe `LAMP` wiederverwendet.

`SWITCH` - ein Schalter

Der Schalterstub `SWITCH` besitzt einen Device-Eingang *button*, der bei jedem Tastendruck eine Nachricht empfängt.

`SWITCH` arbeitet in zwei möglichen Betriebsmodi, die durch den Parameter `toggle` festgelegt werden: einmal als Umschalter, in dem der Ausgang `s` nach jedem Tastendruck *update*(*button*) zwischen „Ein“ (1) und „Aus“ (0) wechselt (bei `toggle=false`), und zum anderen als gewöhnlicher Taster, der bei jedem Druck eine Aktualisierungs-Nachricht verschickt (bei `toggle=true`).

```
stub SWITCH {
    devin bool button
    portout bool s
    param bool toggle

    SWITCH := [swoff]
              (update(button) -> s:=true -> ON |
                 toggle=true -> TOGGLE)

    ON := [swon]
             (update(button) -> s:=false -> SWITCH |
             toggle=true -> TOGGLE)

    TOGGLE := update(button) -> send(s) -> TOGGLE |
              toggle=false -> SWITCH
}
```

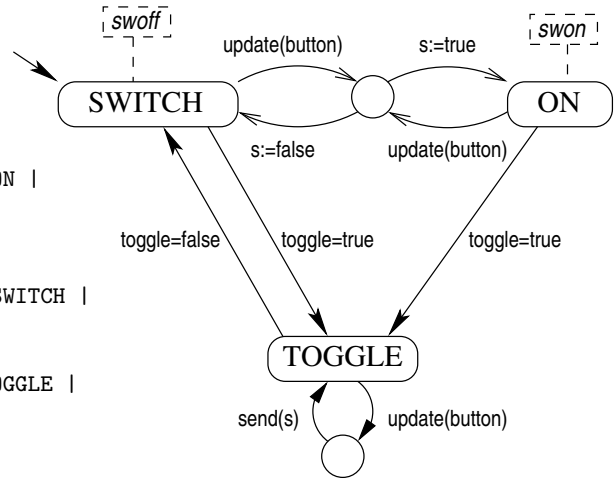


Abb. 6.5: Prozessbeschreibung und entsprechendes Zustandsdiagramm für `SWITCH`

Verschiedene Controller

Zu Demonstrationszwecken wird zunächst ein ungeeigneter Controller `TOGGLE_CTR` definiert, der keinen speziellen Hauptschalter, sondern nur Taster zum Umschalten des Lampenzustandes erlaubt:

```
stub TOGGLE_CTR {
    portin bool sw0
    portin bool sw1
    portin bool sw2
    portout bool lamp

    CTR := lamp:=false -> (
             update(sw0) -> CTRON |
             update(sw1) -> CTRON |
             update(sw2) -> CTRON )

    CTRON := lamp:=true -> (
             update(sw0) -> CTR |
             update(sw1) -> CTR |
             update(sw2) -> CTR )
}
```

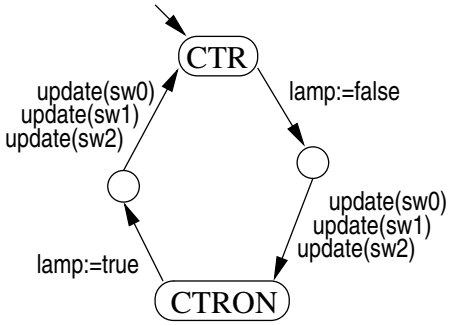


Abb. 6.6: Prozessbeschreibung und entsprechendes Zustandsdiagramm für TOGGLE_CTR

Als geeignete Alternative steht noch MAIN_CTR zur Verfügung, dessen Eingang sw0 einen Umschalter in Hauptschalterfunktion unterstützt:

```
stub MAIN_CTR {
    portin bool sw0
    portin bool sw1
    portin bool sw2
    portout bool lamp

    MOFF := update(sw1) -> MOFF | update(sw2) -> MOFF |
            sw0=true -> MON1

    MON1 := lamp:=true -> (
              update(sw1) -> MON0 | update(sw2) -> MON0 |
              sw0=false -> lamp:=false -> MOFF )

    MON0 := lamp:=false -> (
              update(sw1) -> MON1 | update(sw2) -> MON1 |
              sw0=false -> MOFF )
}
```

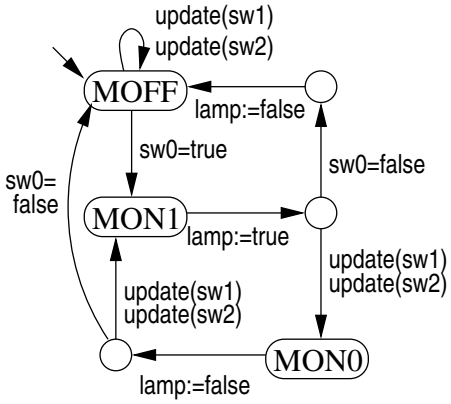


Abb. 6.7: Prozessbeschreibung und entsprechendes Zustandsdiagramm für MAIN_CTR

6.2.2 Netzwerk

Nun wird für beide Controller je ein Netzwerk LT (mit TOGGLE_CTR) bzw. LM (mit MAIN_CTR) aus drei SWITCHes, einer LAMP und einem Controller angelegt:

$$\begin{aligned} Stubs_{LT} &= \{main : SWITCH, s1 : SWITCH, s2 : SWITCH, ctr : TOGGLE_CTR, l : LAMP\} \\ Stubs_{LM} &= \{main : SWITCH, s1 : SWITCH, s2 : SWITCH, ctr : MAIN_CTR, l : LAMP\} \end{aligned}$$

Die Bindung ist für beide Varianten gleich, deshalb trägt *bind* hier keinen Index:

$$\begin{aligned} bind(main.s) &= \{ctr.sw0\} \\ bind(s1.s) &= \{ctr.sw1\} \\ bind(s2.s) &= \{ctr.sw2\} \\ bind(ctr.lamp) &= \{l.on\} \end{aligned}$$

Es folgen noch die Initialisierungen. Für die Taster `s1` und `s2`, von denen nur ein *update*() verlangt wird, spielt die Konfiguration eigentlich keine Rolle. Der Form halber werden sie aber korrekt eingestellt:

$$\begin{aligned} Init_{LT} &= (s1.toggle = true, s2.toggle = true) \\ Init_{LM} &= (s1.toggle = true, s2.toggle = true) \end{aligned}$$

Gemäß dem Abschnitt 4.6 („Initialisierung“) sind die Startzustände in den initialisierten Netzwerken dann [1]

$$\begin{aligned} s^0_{LT} &= \{main : SWITCH, s1 : TOGGLE, s2 : TOGGLE, ctr : CTR, l : LAMP\} \\ s^0_{LM} &= \{main : SWITCH, s1 : TOGGLE, s2 : TOGGLE, ctr : MOFF, l : LAMP\} \end{aligned}$$

Die Erwartung ist somit, dass LM die Anforderungen erfüllt, LT dagegen nicht.

6.2.3 Anforderungen

Die Anforderungen an die Hausbeleuchtung sind in Abschnitt 2.1 aufgeführt. Zur besseren Übersichtlichkeit werden sie hier noch einmal erwähnt und an ihnen die Formalisierung vorgenommen:

1. *Solange* `MAIN` *„aus“ ist, dann geht und bleibt* `LAMP` *aus.*

   Die Verwendung der atomaren Prädikate erlaubt eine sehr einfache Formalisierung:

   $$\mathbb{AG}.\ main.swoff \rightarrow nolight$$

   `swoff` ist ein atomares Prädikat des Schalters `SWITCH`. In der Formel wird die qualifizierte Form verwendet, weil nicht *irgendein* Schalter aus sein soll (wie es die unqualifizierte Schreibweise ausdrücken würde), sondern speziell der Schalter `main`.

   Der Präfix $\mathbb{AG}$ macht aus der Implikation eine starke Sicherheitseigenschaft, d. h., die Implikation gilt nicht nur im Ausgangszustand, sondern in allen erreichbaren.

2. *Wenn* `MAIN` *auf „an“ gestellt wird, dann geht* `LAMP` *an.*

   Mit Hilfe des atomaren Prädikates `light` ist die zugehörige Formel:

   $$\mathbb{AG}.\ [main.s := true]light$$

   Auch diese Formel soll in allen erreichbaren Zuständen gelten, ist also eine starke Sicherheitseigenschaft.

[1] Man beachte, dass hier die Prozesskonstanten (Namen) von *Zuständen* Verwendung finden, bei der Netzwerk-Definition am Anfang dieses Abschnittes sind es jedoch *Stubnamen*.

3. *Wenn* `MAIN` *an ist, dann ändert jeder Druck auf die Taster den Lampenzustand.*

   Die Änderung ist nicht direkt übersetzbar, sondern es muss jede Möglichkeit (von an nach aus und umgekehrt) separat betrachtet werden:

   Wenn `MAIN` *an ist, dann gilt: wenn die Lampe aus ist (*`nolight`*), dann geht sie nach einem Tastendruck an und wenn die Lampe an ist (*`light`*), dann geht sie nach einem Tastendruck aus.*

   Dies läßt sich nun für einen konkreten Taster i direkt formulieren:

$$\mathbb{AG}.\, main.swon \rightarrow \Big(\begin{array}{l} (nolight \rightarrow [update(t.s)]light) \wedge \\ (light \rightarrow [update(t.s)]nolight) \end{array} \Big)$$

   Diese Formel muss für alle verwendeten Taster – hier: $t \in \{s1, s2\}$ – geprüft werden. Man beachte, dass es im μ-Kalkül keine Quantisierungsoperatoren wie $\forall$ gibt. In einer konkreten Implementation kann dies natürlich für eine endliche Menge von Alternativen als äußerster Operator definiert werden.

Aufmerksame Leser haben hier vielleicht schon einen Fehler in den Anforderungen gefunden. Dieser wird im nächsten Kapitel auffallen, wenn die Verifikation mit Hilfe der Implementierung durchgeführt wird.

6.2.4 Verifikation

Obwohl das Beispiel auf den ersten Blick recht klein anmutet, ist es doch schon viel zu groß, um einen kompletten Beweisbaum abzudrucken. Denn es gibt zum Beispiel schon 80.459 verschiedene Läufe der Länge ≤ 6, d. h. Folgen von maximal sechs Transitionen. Daher kann die vollständige Verifikation dieses Beispiels nur noch mit maschineller Unterstützung durchgeführt werden. Deshalb wird hier nur die Verifikation der ersten Eigenschaft demonstriert und auch diese nur exemplarisch an ausgewählten Zuständen getestet.

So wird zunächst die erste Eigenschaft bei Verwendung des `TOGGLE_CTR` (also im Netzwerk LT) geprüft:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{LT_I \vdash nolight}}{LT_I \vdash \neg main.swoff \vee nolight}\,\vee_2}{LT_I \vdash main.swoff \rightarrow nolight}\,Def.\rightarrow \quad \dfrac{\vdots}{LT_I \vdash \Box U}}{LT_I \vdash (main.swoff \rightarrow nolight) \wedge \Box U}\,\wedge}{LT_I \vdash U}\,U}{LT_I \vdash \nu Z.(main.swoff \rightarrow nolight) \wedge \Box Z}\,\sigma Z}{LT_I \vdash \mathbb{AG}.\, main.swoff \rightarrow nolight}\,Def.\ \mathbb{AG}$$

Das atomare Prädikat ist im aktuellen Zustand von l (der Lampe) erfüllt, die Anforderung gilt also im Ausgangszustand. Nun wird an der mit ... bezeichneten Stelle fortgefahren. Im Zustand LT_I sind folgende Transitionen möglich: $ctr.lamp := false$ (und

damit $l.on = false$) als Startaktion von ctr und ein Druck auf einen Schalter, also $update(main.button)$, $update(s1.button)$ und $update(s2.button)$. Hier wird der Baum für die Ausgabeaktion weiterentwickelt (die anderen werden aus Platzgründen weggelassen). Nach $ctr.lamp := false$ sieht das Netz folgendermaßen aus:

$$\begin{aligned} LT1 \;=\; & \{main : SWITCH, s1 : TOGGLE, s2 : TOGGLE, l : LAMP, \\ & ctr : (update(sw0) \rightarrow CTRON | \ldots | update(sw2) \rightarrow CTRON)\} \end{aligned}$$

Auch in $LT1$ sind wieder die drei $update(\cdots.button)$-Aktionen möglich, daneben auch $tick$. Hier wird nur der Folgezustand $LT2$ nach $update(s1.button)$ betrachtet.

$$\begin{aligned} LT2 \;=\; & \{main : SWITCH, s1 : send(s) \rightarrow TOGGLE, s2 : TOGGLE, l : LAMP, \\ & ctr : (update(sw0) \rightarrow CTRON | \ldots | update(sw2) \rightarrow CTRON)\} \end{aligned}$$

Damit sieht die Fortsetzung des Tableaus wie folgt aus:

$$\dfrac{\vdots \qquad \dfrac{\dfrac{\dfrac{\dfrac{\checkmark\ \star^1}{LT1 \vdash (main.swoff \rightarrow nolight)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\checkmark\ \star^1}{LT2 \vdash (main.swoff \rightarrow nolight)} \quad \dfrac{\vdots\ \star^2}{LT2 \vdash \Box U}}{LT2 \vdash (main.swoff \rightarrow nolight) \wedge \Box U}\ \wedge}{LT2 \vdash U}\ U \qquad \vdots}{LT1 \vdash \Box U}\ [\,]}{LT1 \vdash (main.swoff \rightarrow nolight) \wedge \Box U}\ \wedge}{LT1 \vdash U}\ U \qquad \vdots}{LT_I \vdash \Box U}\ [\,]$$

Die mit $\star^1$ bezeichneten Blätter sind analog zum vorherigen Tableau wahre Terminale, die Anforderung ist also auch in diesen beiden Zuständen erfüllt. In $LT2$ sind nun folgende Aktionen möglich: $send(s1.s)$ (und damit $update(ctr.sw1)$), $update(main.button)$ und $update(s2.button)$. Die $send$-Aktion führt nun zum Zustand

$$LT3 = \{main : SWITCH, s1 : TOGGLE, s2 : TOGGLE, ctr : CTRON, l : LAMP\}$$

in dem (u. a.) die Aktion $ctr.lamp := true$ (und damit $l.on = true$) möglich ist, die zum Zustand

$$\begin{aligned} LT4 \;=\; & \{main : SWITCH, s1 : TOGGLE, s2 : TOGGLE, l : ON, \\ & ctr : (update(sw0) \rightarrow CTR | \ldots | update(sw2) \rightarrow CTR)\} \end{aligned}$$

führt. An der Stelle $\star^2$ des letzten Tableaus werden diese beiden Zwischenschritte übersprungen und nur mit $LT4$ fortgefahren:

$$\dfrac{\dfrac{\dfrac{\lightning}{LT4 \vdash \neg main.swoff}\ \vee_1 \qquad \text{(alternative Regeln)} \qquad \dfrac{\lightning}{LT4 \vdash nolight}\ \vee_2}{LT4 \vdash \neg main.swoff \vee nolight}}{LT4 \vdash (main.swoff \rightarrow nolight)}\ Def.\ \rightarrow$$

In $LT4$ ist $main$ im Zustand $SWITCH$, welcher das Prädikat $swoff$ erfüllt. Somit erfüllt $LT4$ nicht die Formel $\neg main.swoff$. Auch die andere $\vee$-Alternative schlägt fehl, weil kein Stub (insbesondere nicht l im Zustand ON) das Prädikat $nolight$ besitzt.

Prüft man dagegen die Transitionsfolge $update(s1.button) \rightarrow send(s1.s)$ unter Verwendung des `MAIN_CTR` (also im Netzwerk LM), dann geht LM in den Zustand

$$LM1 = \{main : SWITCH, s1 : TOGGLE, s2 : TOGGLE, ctr : MOFF, l : LAMP$$

der wegen $l : LAMP$ das Prädikat *nolight* besitzt und deshalb die Anforderung in diesem Zustand erfüllt.

7 Implementation

Im letzten Kapitel zeigte sich, dass eine manuelle Verifikation schon bei relativ kleinen Szenarien wegen des exponentiell wachsenden Zustandsraumes versagt. Dieses Problem tritt natürlich auch bei maschineller Unterstützung auf, diese vergrößert die Menge von praktisch verifizierbaren Netzwerken jedoch erheblich.

Momentan liegt eine prototypische Implementation in ProLog vor[1]. Diese Sprache bietet den Vorteil, dass man in ihr die Regeln für die Prozess-Semantik und die Tableauverifikation direkt ausdrücken kann und sich um viele technische Belange wie das Einlesen der Netzwerkbeschreibung oder die Verwaltung von Datenstrukturen nicht zu kümmern braucht. So ist das Programm überschaubar klein (etwa 400 Zeilen) und ist damit auch nicht sehr anfällig für Fehler. Um portabel zu sein, verwendet das Programm nur ISO-Direktiven und -Operatoren [Hod99].

Die Implementation in ProLog ist allerdings recht ineffizient und langsam. Viele Berechnungen werden auch redundant durchgeführt und dadurch, dass Datenstrukturen nicht von Hand erstellt, sondern von ProLog generiert werden, haben kritische Operationen eine höhere Komplexität als nötig. Deshalb ist das hier vorgestellte Programm eher als Machbarkeitsstudie und zum Experimentieren gedacht.

Dieses Kapitel beschreibt die Struktur und Anwendung der Implementation. Es setzt ein grundsätzliches Verständnis der Programmiersprache ProLog voraus. Einführungen dazu gibt es sowohl in Buchform als auch im Internet sehr viele, siehe z. B. [SL95].

Das System besteht aus drei Teilen, die hier zusammen mit ihrer Schnittstelle beschrieben werden:

Prozess- und Netzwerkdef.	Semantik der Prozessalgebra	μ-Kalkül

`processdef(name,Def)`
`net(name,Def,Bind,Init)`
	`next(P1, Action, P2)`
	`has_property(P, Prop)`
	`run(Process, Depth)`
		`failure(F, P, Run)`

Der Anwender stellt dabei den ersten Block zur Verfügung (Prozess- und Netzwerkdefinitionen), das vorliegende Programm implementiert die Semantiken und die Tableauregeln.

Zu beachten ist, dass das Programm nur mit den Prozessausdrücken arbeitet, es beachtet keine Kanaldeklarationen. Somit setzt es voraus, dass Kanäle typkorrekt gebunden sind und es auch keine Schreibfehler bei Kanal- und Prozessnamen gibt.

[1] Diese kann von `http://www.piware.de/docs.shtml` heruntergeladen und unter den Bedingungen der GNU Lesser General Public License verwendet werden. Der vollständige Lizenztext ist auf `http://www.gnu.org/copyleft/lgpl.html` einsehbar.

7.1 Prozess- und Netzwerkdefinition

Um die Definitionen von Prozesskonstanten und Netzwerken ProLog zugänglich zu machen, muss die mathematische Notation in ProLog-Syntax übersetzt werden:

Syntax der Transitionen		
CCMB	ProLog	Anmerkungen
$out := c$	`out:=c`	
$out := f(x_1, \ldots, x_n)$	`out:==f(x1,...,xn)`	(ProLog benötigt diese Unterscheidung)
$send(out)$	`send(out)`	
$in = c$	`in=c`	
$update(in)$	`update(in)`	
$par = c$	`par==c`	(zur Unterscheidung Param./Eingabe)
$tick$	`tick`	
$stub.ch$	`stub^ch`	(Qualifikation von Kanälen)

Syntax der Prozessausdrücke	
CCMB	ProLog
0	`0`
$a \rightarrow b \rightarrow P$	`[a, b, p]`
$P_1 \mid P_2$	`p1 + p2`
$[prop]P$	`[prop] * p`
$K := P$	`processdef( k, p )`
$K(x) := P(x, y)$	`processdef( k, p, y is f(x) )`

Abb. 7.1: Prolog-Syntax der Transitionen und Prozesse

Die Drei-Parameter-Form von `processdef` ist nützlich, um parametrisierte Prozesskonstanten zu verwenden: enthalten der Prozessname oder die Definition Variablen, so können diese im dritten Parameter instantiiert werden.

Zur Illustration werden die beiden Uhrentypen (endlich und unendlich tickend) aus dem Unterabschnitt 5.3.1 (s. Seite 31) und auch noch eine abgewandelte Form, in der die Uhr jederzeit kaputt gehen kann, in ProLog-Syntax notiert:

```
processdef( clock, [send(tock), clock] ).

processdef( clock(0), 0 ).
processdef( clock(N), [send(tock), clock(N1)], N1 is N-1 ).

processdef( bclock, [tock:=true, block] + [tock:=false, [error] * 0] ).
```

Hier beinhaltet die dritte Klausel die beiden Variablen `N` und `N1`. `N` ist als bekannt vorauszusetzen, die Instantiierung von `N1` erfolt dann im dritten Parameter von `processdef`.

Zu beachten ist hierbei, dass eventuell vorhandene Prozesse ohne Variablen (hier also `clock(0)`) immer *vor* den Definitionen mit Variablen stehen, da ProLog die Klauseln von oben nach unten durchsucht.

Neben den so zu beschreibenden Stub-Prozessen sind nun noch die Netzwerke zu deklarieren. Ein gegebenes Netzwerk namens N hat in CCMB die folgende Form:

$$\begin{aligned} Stubs_N &= \{name : Stub, \ldots\} \\ bind_N &= \Big\{(name_1.out_1, \{name_{11}.in_{11}, name_{12}.in_{12}, \ldots\}), (name_2, \ldots), \ldots\Big\} \\ Init_N &= (name_1.par = c_1, name_2.par = c_2, \ldots) \end{aligned}$$

Bei der Stub-Instantiierung werden für die Implementation nicht alle vier Komponenten von $Stub = (Inputs, Outputs, Params, \mathcal{F})$ verwendet, sondern nur der Startzustand s^0 aus $\mathcal{F}$ in Form eines Prozessausdrucks. Die *next*-Relation und die Abbildung V wurden ja schon durch die Prozessausdrücke in `processdef` definiert, der Typ eines Kanals lässt sich mittels des verwendeten Operators (`:=`, `=`, `==`) bestimmen und es wird statische Typkorrektheit vorausgesetzt (so dürfen z. B. nicht zwei Eingänge oder ein *int*-Ausgang an einen *bool*-Eingang gebunden werden).

Die ProLog-Definition dieses Netzwerkes hat folgende Form:

```
net( n, [name1:P1, name2:P2,...],
        [name1^out1/name11^in11, name1^out1/name12^in12, ...],
        [name1^par == c1, name2^par == c2, ...]
   ).
```

`P1` und `P2` sind dabei die Startzustände der Stubinstanzen `name1` und `name2`. Jede Bindung eines Einganges wird einzeln in der Form `out/in` notiert. Zu beachten ist, dass der Ausgang *immer* als erstes geschrieben wird.

Zur Illustration finden sich weiter unten im Abschnitt 7.4 komplexe Beispiele, in denen auch Netzwerkdefinitionen vorgenommen werden. Das Archiv der Implementierung enthält ausserdem die ProLog-Quelltexte der hier besprochenen Beispiele und der Anwendungsfälle.

Um ein ganzes Netzwerk als Prozess zu repräsentieren, reicht die Stub-Liste nicht aus, da man zu vielen Gelegenheiten auch den Netzwerknamen benötigt. Daher hat ein Netzwerk-Prozessausdruck die Form `n( netname, [name1:p1, name2:p2, ...] )`.

7.2 Prozess-Semantik

In der Datei `processes.pl` befinden sich die Klauseln, die die Prozess- und Netzwerksemantik implementieren. Zum Experimentieren und zur Fehlersuche sind auch noch einige Debugging-Funktionen vorhanden.

Dieses Modul implementiert die zentrale Klausel `next(P1, trans, P2)`, die wahr ist, wenn $(P_1, trans, P_2) \in next$ gilt, also in `P1` die Transition `trans` möglich ist und diese zum Prozess `P2` führt. Dies funktioniert sowohl für einzelne Stubs als auch für Netzwerke. `P1` muss vollständig instantiiert sein, `trans` und `P2` dürfen beliebig variabel sein und werden dann mit gültigen Werden instantiiert. Verschiedene Aufrufarten seien hier demonstriert:

- Frage, ob ein Element in *next* ist oder nicht:

```
?- next( clock, send(tock), clock ).

yes

?- next( clock, send(tock), 0 ).

no
```

- Bestimmung der möglichen Nachfolgerzustände:

```
?- next( clock, send(tock), P ).

P = clock

?- next( bclock, tock:=false, P ).

P = [error]*0
```

- Bestimmung der möglichen Werte einer Transition:

```
?- next( bclock, tock:=X, _ ).

X = true ? ;
X = false ? ;

no
```

- Bestimmung aller möglichen Transitionen und ihrer Nachfolger:

```
?- next( bclock, T, P ).

P = bclock
T = tock:=true ? ;

P = [error]*0
T = tock:=false ? ;

no
```

Es ist manchmal interessant, die Läufe eines Prozesses genauer zu untersuchen. Dazu steht die Klausel `run( P, N )` zur Verfügung, die alle möglichen Läufe von `P` bis zur Länge `N` als Baumstruktur auflistet:

```
?- run( clock, 3 ).
--> clock
    send(tock)
    --> clock
        send(tock)
        --> clock
            send(tock)
            --> clock
                send(tock)
```

Zeilen, die mit `-->` anfangen, bezeichnen einen Prozess. In der selben Einrückungstiefe stehen alle möglichen Transitionen und – wiederum eingerückt – deren Nachfolgerzustände.

```
?- run( bclock, 2 ).
--> bclock
    tock:=true
    --> bclock
        tock:=true
        --> bclock
            tock:=true
            tock:=false
        tock:=false
        --> [error]*0
    tock:=false
--> [error]*0

?- run( clock(3), 5 ).
--> clock(3)
    send(tock)
    --> clock(2)
        send(tock)
        --> clock(1)
            send(tock)
            --> 0
```

Möchte man nicht alle möglichen Läufe wissen, sondern hat man einen konkreten Lauf z. B. als Gegenbeispiels gegeben, dann ist die Klausel `trace( P, [tr1, tr2, ...] )` nützlich, die die gegebenen Transitionen in der Liste nacheinander ausführt und nach jedem Schritt den aktuellen Zustand des Prozesses ausgibt.

Die zweite zentrale Definition in `processes.pl` ist `has_property( P, Prop )`, die wahr ist, wenn `P` das atomare Prädikat `Prop` besitzt, also $P \in V(Prop)$ gilt. `P` muss instantiiert sein, `Prop` dagegen darf auch eine Variable sein, um alle Prädikate zu finden, die ein Prozess besitzt:

```
?- has_property( [error,p1] * 0, p2 ).

no

?- has_property( [error,p1] * 0, P ).

P = error ? ;
P = p1
```

Um von den konkreten Schnittstellen zur Prozess- und Netzwerkdefinition zu abstrahieren, exportiert dieses Modul die Klausel `expand_proc_const( K, P )`, die wahr ist, wenn `P` die Definition des Stubs bzw. Netzwerkes `K` ist. Sie wertet dazu die vom Anwender bereitgestellten `processdef`- und `net`-Klauseln aus.

Ein Netzwerk kann verschiedene syntaktische Repräsentationen haben: die Reihenfolge der Stubs kann geändert oder Prozesskonstanten durch ihre Definition ersetzt werden ohne dass sich die Semantik ändert. Die Entscheidung ob zwei gegebene Prozesse äquivalent sind erledigt die Klausel `proc_equiv( P1, P2 )`.

Alle anderen Klauseln sind Hilfsprädikate und nicht als Teil der Schnittstelle anzusehen. Die meisten haben einen einfachen Zweck und eine einfache Implementation und sind kommentiert.

Nur das Prädikat `poss_trans_but( P, Excl, Trans )` bedarf einer Erläuterung: Es hat die Aufgabe, alle möglichen Transitionen des Prozesses `P` außer denen in `Excl` zu finden und in `Trans` zurückzugeben. Dabei müssen auch die durch Bindungen entstehenden Komplementärereignisse, die nicht in `Excl` aufgeführt sind, herausgefiltert werden. Ansonsten würden Anfragen der Art $P \models [\,\overline{s.out := true}\,]F$ falsch verifiziert werden: Wenn $s.out$ an einen Eingang $s'.in$ gebunden ist und $s.out := true$ in P möglich ist, dann kann auch $s'.in = true$ möglich sein. Es ist aber nicht im Sinne obiger Formel zu testen, ob F nach der Transition $s'.in = true$ gilt, weil $s.out := true$ und $s'.in = true$ eigentlich ein und dasselbe Ereignis – nur von verschiedenen Stubs aus betrachtet – sind.

7.3 Tableau-Beweiser

Auch die Formeln des μ-Kalküls müssen in eine ProLog-kompatible Syntax übersetzt werden, um damit Berechnungen durchzuführen:

μ-Kalkül	ProLog	Anmerkungen
$\top$	`true`	
$\bot$	`false`	
Z	`z`	(atomares Prädikat / μ-Variable)
$\neg F$	`~ F`	
$F \wedge G$	`F /\ G`	
$F \vee G$	`F \/ G`	
$[\overline{\{a_1, a_2, \ldots\}}]F$	`allbut( [a1,a2,...], F )`	
$[\{a_1, a_2, \ldots\}]F$	`all( [a1,a2,...], F )`	
$\Box F$	`all( F )`	
$\langle\overline{\{a_1, a_2, \ldots\}}\rangle F$	`exbut( [a1,a2,...], F )`	
$\langle\{a_1, a_2, \ldots\}\rangle F$	`ex( [a1,a2,...], F )`	
$\Diamond F$	`ex( F )`	
$\mu Z.F$	`mu( z, F )`	
$\nu Z.F$	`nu( z, F )`	
$\mathbb{AG}.F$	`ag( F )`	(entsprechend für $\mathbb{EF}$ etc.)
$\mathbb{H}(F)$	`h( F )`	
$\mathbb{H}(F, n)$	`h( F, n )`	
F until G	`until( F, G )`	
F unless G	`unless( F, G )`	
$I \rightarrow T$	`ifthen( I, T )`	
$(I \wedge T) \vee (\neg I \wedge E)$	`ifthenelse( I, T, E )`	

Abb. 7.2: Prolog-Syntax der μ-Formeln

Das Modul `muc.pl` implementiert einen Theorembeweiser für das μ-Kalkül und die in Abschnitt 5.5 eingeführten Makros. Es setzt voraus, dass die Klauseln `next`, `has_property`, `expand_proc_const` und `proc_equiv` definiert sind, es baut also auf `processes.pl` auf.

ProLog sucht solange nach einer Lösung und verwirft fehlschlagende terminale Klauseln, bis es einen Unifikator gefunden hat, der die Anfrage wahr macht. Daher ist es ungünstig, die Tableau-Regeln direkt in ProLog zu übertragen und zu fragen, ob ein gegebener Prozess P eine Formel F erfüllt. Stattdessen wird die Anfragelogik negiert: auf die Frage, ob P die Formel F *nicht* erfüllt, sucht ProLog solange, bis es einen Fehlschlag gefunden hat und verwirft dabei Terminale, die F erfüllen.

Somit heißt die den Anwender interessierende Klausel `failure( F, P, R )`. Schlägt sie – auf eine Formel `F` und einen Prozess `P` angesetzt – fehl (Antwort „no“), dann gilt $P \models F$. Wird eine Lösung `R` gefunden, dann enthält `R` ein Gegenbeispiel in Form einer Liste von Transitionen, die zu einem fehlschlagendem Blatt des Tableaus führt. Die Klauseln für `failure` entsprechen somit der logischen Negation der μ-Tableauregeln.

Da die Anzahl zu testender Zustände bei realen Systemen recht groß werden kann, ist `failure` als Tiefensuche implementiert. Das hat den Nachteil, dass ein gefundenes Gegenbeispiel nicht unbedingt das kürzestmögliche ist, kommt aber mit viel weniger Speicherplatz als die Breitensuche (welche einen exponentiellen Speicherbedarf hat) aus.

Für die eigentliche Berechnung wird die vierstellige Form von `failure` verwendet, in dem zusätzlich noch die bisher definierten Tableauvariablen ($U_i := \ldots$ bei der Regel σZ) gespeichert werden.

Da sich die Verwaltung der Liste schon geprüfter Variablen-Zustands-Paare für die U-Regel als Klauselparameter sehr schwierig gestaltet, wird sie in einem dynamischen Prädikat `occurred_u` gespeichert. Wenn verschiedene Netzwerke in einer interaktiven ProLog-Sitzung getestet werden, sollte diese Liste zwischendurch durch Aufrufen von `?- clean.` gelöscht werden.

Trifft `failure` während der Verifikation auf ein fehlschlagendes Terminal, so wird es mit Hilfe des Prädikats `printfail` ausgegeben. Dies erleichtert die Fehlersuche sowohl im Verifikationssystem selbst als auch in kleineren Netzwerken. Die Ausgabe dauert aber sehr lange, so dass sich die Verifikation damit wesentlich verlangsamt. Deshalb sollte die Ausgabe normalerweise unterdrückt werden; dazu ist in `muc.pl` die vorletzte Klausel durch Entfernen des Kommentarzeichens zu aktivieren.

Man beachte dabei, dass die Existenz fehlschlagender Terminale nicht automatisch bedeutet, dass die Formel im getesteten Zustand nicht gilt! Die Regeln $\vee$ und $\langle\,\rangle$ lassen Alternativen zu, von denen nur eine zu einem wahren Terminal führen muss.

Das Prädikat `neg(F, NF)` realisiert einen Schritt der Transformation zur Negationsnormalform (siehe Unterabschnitt 5.6.2). Es ist wahr, wenn `NF` die Normalform des Negats von `F` ist.

Mit `macro( M, Def )` werden schließlich die in Abschnitt 5.5 eingeführten Makrooperationen definiert. Dabei ist `Def` die Definition des Makros `M`.

Der Rest des Moduls besteht aus kleineren Hilfsprädikaten, die nur eine kleine und eng begrenzte Aufgabe haben. Sie sind im Quellcode kommentiert und nicht als Teil der Schnittstelle anzusehen, für Anwender also uninteressant.

7.4 Beispiele

7.4.1 Uhren

Um mit einem ganz einfachen Fall anzufangen, werden die oben definierten Uhren `clock`, `clock(N)` und `bclock` untersucht:

- „unendlich oft tickend“? Die Formel $\mathbb{AG}.\langle send(tock)\rangle\top$ wird erwartungsgemäß von `clock` erfüllt:

  ```
  ?- failure( ag( ex([send(tock)]) ), clock, R ).

  no
  ```

  `clock(4)` und `bclock` schlagen dagegen fehl. An ersterem wird auch die Klausel `trace` demonstriert:

  ```
  ?- failure( ag( ex([send(tock)]) ), clock(3), R ), trace( clock(3), R ).
  ex([],[send(tock)]) does not hold in 0

  send(tock)
  clock(2)
  send(tock)
  clock(1)
  send(tock)
  0

  R = [send(tock),send(tock),send(tock)]

  ?- failure( ag( ex([send(tock)]) ), bclock, R ).
  ex([],[send(tock)]) does not hold in [error]*0

  R = [tock:=false|_]
  ```

  Bei beiden wurde eine Lösung `R` gefunden, die den Lauf eines Gegenbeispiels angibt. Die ersten Ausgabezeilen wurden von `printfail` generiert und zeigen, an welchem Punkt die Verifikation auf ein fehlschlagendes Terminal gestoßen ist.

- „Frei von Fehlerzuständen“? Das heißt, dass kein Zustand das Prädikat `error` tragen darf ($\mathbb{AG}.\neg error$):

  ```
  ?- failure( ag( ~error ), clock, R ).

  no

  ?- failure( ag( ~error ), bclock, R ).
  ~error does not hold in [error]*0

  R = [tock:=false]
  ```

7.4.2 Anwendungsfall Temperaturkontrolle

Nun wird als komplexes Beispiel noch dieser Anwendungsfall maschinell verifiziert. Dazu wird er zunächst in ProLog umgesetzt, Abb. 7.3 zeigt den Code des Moduls.

```
1  :- include( 'processes' ).
2  :- include( 'muc' ).
3
4  processdef( lamp, [on=false, lamp] + [on=true, lampon] ).
5  processdef( lampon, [on=true, lampon] + [on=false, lamp] ).
6
7  processdef( sensor , [ send(temp), tick (2), sensor ] ).
8
9  processdef( slowctr , [ update(temp), tick , [ tick , screact] + screact ] ).
10 processdef( screact , [ alarm := true, slowctr] +
11                        [alarm := false , slowctr ] ).
12
13 processdef( fastctr , [ update(temp), tick,
14     [alarm:=true,fastctr]+[alarm:=false, fastctr ]] ).
15
16 net( tcslow , [ s:sensor , c:slowctr , l:lamp],
17      [s^temp/c^temp, c^alarm/l^on], [] ).
18
19 net( tcfast , [ s:sensor , c: fastctr , l:lamp],
20      [s^temp/c^temp, c^alarm/l^on], [] ).
21
22 checks( Run ) :-
23     failure ( all ([send(s^temp)], h([l^on=true,l^on=false], 2) ), tcslow , Run ).
24 checkf( Run ) :-
25     failure ( all ([send(s^temp)], h([l^on=true,l^on=false], 2) ), tcfast , Run ).
26 checkfa( Run ) :-
27     failure ( ag( all ([send(s^temp)], h([l^on=true,l^on=false ], 2) ) ), tcfast , Run ).
```

Abb. 7.3: Anwendungsfall „Temperaturkontrolle“ in ProLog

Die Zeilen 1 und 2 binden die beiden benötigten Module ein. Die drei letzten Klauseln (Zeilen 22 – 29) kapseln Anfragen: `checks` prüft das Netz mit dem `slowctr`, `checkf` entsprechend mit dem `fastctr`. `checkfa` testet, ob die Forderung in allen Zuständen (nicht nur dem Startzustand) erfüllt ist:

```
?- checks(R).
false does not hold in n(tcslow,[s:[sensor],c:screact,l:lamp])

R = [send(s^temp),tick,tick]

?- checkf(R).

no

?- checkfa(R).

(270 ms) no
```

Das Netz mit `fastctr` erfüllt also die Echtzeitforderung sogar als Sicherheitseigenschaft. Dabei wird die `next`-Klausel immerhin schon 2.756 mal aufgerufen, was zeigt, dass die manuelle Verifikation dieser Formel schon sehr schwierig ist.

7.4.3 Anwendungsfall Hausbeleuchtung

Die Umsetzung des Szenarios in ProLog zeigt Abb. 7.4.

```
1  :- include( 'processes' ).
2  :- include( 'muc' ).
3
4  processdef( lamp, [nolight ] * ( [ on=false, lamp] + [on=true, lampon] ) ).
5  processdef( lampon, [light ] * ( [ on=true, lampon] + [on=false, lamp] ) ).
6
7  processdef( switch , [ swoff ] * ( [ update(button), s:=true, switchon] +
8      [toggle==true, switchtoggle] ) ).
9  processdef( switchon , [swon ] * ( [ update(button), s:=false , switch] +
10     [toggle==true, switchtoggle] ) ).
11 processdef( switchtoggle , [ update(button), send(s), switchtoggle] +
12     [toggle==false, switch ] ).
13
14 processdef( togglectr , [ lamp:=false , [update(sw0), togglectron] +
15     [update(sw1), togglectron] + [update(sw2), togglectron ]] ).
16 processdef( togglectron , [ lamp:=true, [update(sw0), togglectr] +
17     [update(sw1), togglectr] + [update(sw2), togglectr ]] ).
18
19 processdef( mainctr, [update(sw1), mainctr] + [update(sw2), mainctr] +
20     [sw0=true, mainctr_on1] ).
21 processdef( mainctr_on1, [lamp:=true, [update(sw1),mainctr_on0] +
22     [update(sw2),mainctr_on0] + [sw0=false, lamp:=false, mainctr] ] ).
23 processdef( mainctr_on0, [lamp:=false , [update(sw1),mainctr_on1] +
24     [update(sw2),mainctr_on1] + [sw0=false, mainctr]] ).
25
26 net( lt , [ main:switch, s1:switch , s2:switch , ctr : togglectr , l:lamp],
27      [main^s/ctr^sw0, s1^s/ctr^sw1, s2^s/ctr^sw2, ctr^lamp/l^on],
28      [s1^toggle==true, s2^toggle==true] ).
29
30 net( lm, [main:switch, s1:switch , s2:switch , ctr:mainctr, l:lamp],
31      [main^s/ctr^sw0, s1^s/ctr^sw1, s2^s/ctr^sw2, ctr^lamp/l^on],
32      [s1^toggle==true, s2^toggle==true] ).
```

Abb. 7.4: Anwendungsfall „Hausbeleuchtung“ in ProLog

Die Überprüfung der ersten Eigenschaft ($\mathbb{AG}$. $main.swoff \rightarrow nolight$) im Netzwerk mit dem $TOGGLE_CTR$ schlägt erwartungsgemäß fehl:

```
?- failure( ag( ifthen(main^swoff,nolight) ), lt, R ).

R = [ctr^lamp:=false,update(s2^button),send(s2^s),ctr^lamp:=true]
```

Nun ist die Frage spannend, ob die Anforderung im Netzwerk `LM` wirklich erfüllt ist:

```
?- failure( ag( ifthen(main^swoff,nolight) ), lm, R ).

 [update(s2^button),update(s1^button),send(s2^s),update(main^button),
 main^s:=true,ctr^lamp:=true,send(s1^s),ctr^lamp:=false,
 update(s1^button),send(s1^s),ctr^lamp:=true,update(main^button),
 main^s:=false]
```

Es gibt also ein Gegenbeispiel. Auch wenn das gefundene sicherlich nicht das kürzeste ist, so zeigen doch die letzten beiden Transitionen, wo das Problem liegt: unmittelbar nach

dem die Nachricht $main.s := false$ gesendet wird, ist die Lampe natürlich noch nicht an, da der `mainctr` dies erst im nächsten tick erledigt. Eine Abschwächung besteht darin, zu fordern, dass entweder die Lampe aus ist oder dies im nächsten tick geschieht:

```
?- failure( ag( ifthen(main^swoff,nolight\/h(l^on=false,1) ) ), lm, R ).

(530 ms) no
```

Dieses Problem tritt auch bei den beiden anderen Anforderungen auf, die analog korrigiert werden müssen.

7.5 Laufzeitverhalten

Eine praktikable Maßzahl für die Beurteilung der Größe eines Netzwerkes ist die Anzahl der Zustände, die während der Verifikation getestet wurden. Diese werden wie schon erläutert in dem dynamischen Prädikat `occurred_u` gespeichert; die Klausel `numstates(N)` im Modul `muc.pl` ermittelt die Anzahl der gespeicherten Zustände und gibt sie in `N` zurück.

Auf dem Testsystem – ein Athlon 1.3 GHz unter Linux (Kernel 2.4.21) und dem Compiler GNU Prolog[2] – wurde die Laufzeit verschiedener Anfragen an Netze der Größenordnung des Anwendungsfalles „Hauslichtbeleuchtung“ gemessen:

Zustände	Dauer in s	ms/Zustand
96	0.52	5.4
315	2.16	6.9
368	2.35	6.4
630	2.93	4.7

Ausserdem wurde die Laufzeit an folgendem pathologischen Netzwerk mit bekannter Anzahl von Zuständen ermittelt:

```
:- include( 'processes' ).
:- include( 'muc' ).

processdef( t(0), 0 ).
processdef( t(N), [a:=N,t(N1)], N1 is N-1 ).

net( bench,
  [a:t(9), b:t(9), c:t(9)],
  [], [] ).

b :- failure( ag(true), bench, _ ).
```

[2] http://gnu-prolog.inria.fr

Durch Variation der Anzahl der „Ticks“ des Stubs `t(N)` und der Anzahl k der Stub-Instanzen (hier: $N = 9$ und $k = 3$) lässt sich die Größe des Zustandsraumes (die sich aus $(N + 1)^k$ ergibt) verändern. Die ermittelten Geschwindigkeiten sind zwar nicht sehr repräsentativ für praktische Netzwerke, jedoch lässt sich daran die Komplexität des Algorithmus ableiten:

Zustände	**Dauer in s**	**ms/Zustand**
100	0.11	1.1
500	2.04	4.1
1.000	9.92	10
3000	78.2	26
5.000	210	42
7000	404	57
10.000	1030	103

Die benötigte Zeit pro Zustand wächst linear mit der Anzahl der Zustände. Der primäre Grund dafür ist die Implementation der U-Regel in `muc.pl`: bei jedem Aufruf von ihr muss getestet werden, ob die aktuelle Formel schon einmal im aktuellen Zustand geprüft wurde. Da die Liste schon geprüfter Zustände mit der Zeit immer länger wird, dauert auch deren Durchsuchen immer länger.

Auch werden die Elemente dieser Liste nicht direkt mit dem aktuellen Zustand verglichen, sondern deren Äquivalenz durch das Prädikat `proc_equiv` bestimmt, was sehr aufwändig ist. Zudem sind ProLog-Programme generell wenig performant, da die Datenstrukturen und der Lösungsalgorithmus vom Compiler erzeugt und nicht speziell auf das Problem optimiert werden. Daher ist auch schon bei kleinen Netzen die Geschwindigkeit im Hinblick auf die Leistung der Testplattform äußerst gering.

Diese Implementation ist somit für Probleme bis zur Größenordnung von 10.000 Zuständen brauchbar. Praktische Szenarien sind aber oft wesentlich umfangreicher, so dass hier Verbesserungen nötig sind. Sinnvoll wären hier z. B.:

- die Implementation des Programms in C oder C++; So kann etwa die Komplexität der Suchoperationen von $\mathcal{O}(n)$ auf $\mathcal{O}(\log n)$ verringert werden, in dem mit Hash-Tabellen und optimierten Datenstrukturen gearbeitet wird.
- das Festlegen einer Normalform für Prozessausdrücke; Dadurch kann die Operation `proc_equiv` entfallen, die sehr zeitaufwändig ist.

8 Zusammenfassung und Ausblick

Diese Arbeit hat gezeigt, dass die Formalisierung und automatische Verifikation von Feldbus-Systemen möglich und bis zu gewissen Grenzen auch praktikabel ist. Die Hauptleistung bestand in der Anforderungsanalyse und Entwicklung einer formalen Sprache und Semantik für solche Systeme und der Erstellung einer Referenzimplementation.

Die bereichsübergreifende Arbeit zwischen theoretischer Informatik, in der Werkzeuge wie das μ-Kalkül schon lange bekannt sind, und angewandter Informatik, in der formale Methoden (noch?) nicht allzugroße Verbreitung gefunden haben, war für mich sehr spannend und interessant. Es zeigte sich, dass die praktische Anwendung theoretischer Erkenntnisse zwar nicht ohne Schwierigkeiten und Abstriche gelingt, jedoch mit moderatem Aufwand zu verwertbaren Ergebnissen führt.

Die hier beschriebenen, aber auch andere Anwendungsfälle haben jedoch gezeigt, dass die Mächtigkeit der Prozessalgebra bei der Modellierung vieler Systeme noch zu schwach ist und auch die Netzwerksemantik auf konkrete Systeme wie z. B. LON-Netze besser angepasst werden sollte. Somit ist diese Arbeit eher als Grundsteinlegung und weniger als fertiges Produkt zu betrachten. Denkbare Fortsetzungen dieser Arbeit wären z. B.:

- Entwicklung alternativer Semantiken für konkrete Feldbussysteme
- Erweiterung der Prozessbeschreibungssprache um mehr Operatoren wie Negation und „syntaktischen Zucker“ (z. B. eingeschränkte Vergleichsoperatoren wie $<$ und $\geq$)
- Erweiterung der Semantik auf N:M-Bindungen, d. h., dass auch mehrere Eingänge an einen Ausgang gebunden werden können
- eine schnellere Implementation
- Integration des Theorembeweisers in das CCM-System, zusammen mit der Bereitstellung eines Frontends
- Anbindung an schon existierende Systeme; z. B. Programme, die schon vorliegende Zustandsübergangsbeschreibungen in die CCMB-Syntax übersetzen

Abschließend möchte ich allen Personen danken, die mir bei der Erstellung dieser Belegarbeit in langen Diskussionen ihr Ohr und ihre Zeit geopfert und mich unterstützt haben; insbesondere sind das Prof. Klaus Kabitzsch, Gunnar Stein und Holger Preiss. Danken möchte ich an dieser Stelle auch Prof. Horst Reichel, dessen Vorlesungen mir die interessante und vielfältige Welt der theoretischen Informatik offenbarten.

Martin Pitt
Dresden, im September 2003

Literaturverzeichnis

[Gol92] Goldblatt, Robert: *Logics of time and computation.* Center for the study of language and information, second edition, 1992.

[Hoa85] Hoare, Tony: *Communicating Sequential Processes.* Prentice Hall International, 1985.

[Hod99] Hodgson, J. P. E.: *Prolog: The ISO directives, control constructs and builtins*, 1999. Verfügbar unter `http://pauillac.inria.fr/~deransar/prolog/bips.html`

[ISO96] ISO/IEC: *Extended BNF*, final draft edition, 1996. Verfügbar unter `http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`

[Mil89] Milner, Robin: *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[MK99] Magee, Jeff and Jeff Kramer: *Concurrency - State models & Java Programs.* Wiley, 1999.

[SL95] Schwab, Klaus und Heiko Ludwig: *Programmierkurs Prolog*, 1995. Verfügbar unter `http://www.fto.de/~hschaefer/prolog95/intro.html`

[Sti92] Stirling, Colin: *Modal and temporal logics.* Clarendon Press, Oxford, 1992.