# Nachweis der Verhaltensgleichheit zwischen Prozessmodellen auf unterschiedlichen Abstraktionsebenen

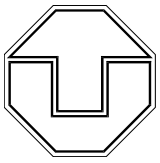
## Diplomarbeit

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



24. Juni 2004

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# 1. Einführung

## 1.1. Einordnung dieser Arbeit

Diese Arbeit fügt sich in den den Bereich „Verifikation von Komponenten der Automatisierungstechnik“ ein. Sie kann als Fortführung meiner Belegarbeit [Pit03] verstanden werden, die den formalen automatischen Nachweis von gewünschten Eigenschaften in Netzwerken von Feldbus-Komponenten diskutierte und implementierte. Diese Diplomarbeit widmet sich der Aufgabe, die Möglichkeiten formaler Verifikation zu erweitern und praxistauglicher zu machen, indem sie sich mit folgenden Problemen beschäftigt:

- In [Pit03] wurde das Verhalten einer Komponente durch Ausdrücke einer Prozess-Algebra „CCMB“ beschrieben. Für konstruktive Verfahren – d. h. ein System wird zuerst in CCMB beschrieben, verifiziert und dann evtl. mit Hilfe von Synthesetools implementiert – ist diese Methode auch durchaus geeignet.

  Allerdings sind viele Komponenten schon als fertige Implementierung vorhanden. Um auch Systeme mit solchen fertigen Komponenten zu verifizieren, muss analytisch vorgegangen werden, d. h. die CCMB-Prozesse müssen aus dem Quelltext ermittelt werden. Die Automatisierung dieser Modellgenerierung ist ein Ziel dieser Arbeit.

- Bei vielen praktischen Netzwerken sind deren Zustandsräume wegen der kombinatorischen Vielfalt so groß, dass eine automatische Verifikation nicht mehr möglich ist (sogenannte „Zustandsexplosion“). Deshalb wurde vorgeschlagen, in der aktiven Bibliothek für eine Komponente mehrere Modelle zu hinterlegen: eine „Implementierung“, die die Komponente vollständig in allen Einzelheiten beschreibt und beliebig viele „Abstraktionen“, in denen Details versteckt werden die den System-Entwickler auf gewissen Entwurfsebenen nicht mehr interessieren.

  Diese Abstraktionen erleichtern sowohl die formale Verifikation von Eigenschaften als auch die Arbeit des Entwicklers, der dann auch große Netzwerke besser überschauen und verstehen kann.

  Um die Konsistenz dieser verschiedenen Beschreibungen sicherzustellen, soll ein Werkzeug entstehen, das die Verhaltensgleichheit einer Abstraktion und der Implementierung prüft.

Der Nachweis der Verhaltensgleichheit zweier Implementierungen ermöglicht auch abgeleitete Anwendungen, z. B. die qualifizierte Suche möglicher Implementierungs-Kandidaten bzw. Parameter-Einstellungen, wenn eine „Spezifikations-Komponente“ vorliegt.

Die Idee dazu entstand bei der Diskussion über die Implementierung einer Sitzsteuerung in PKWs. Diese liegt derzeit in C#-Klassen in einem Framework (CCM) vor, an denen

sich diese Arbeit orientieren soll. Die Komponenten-Klassen dienen daher als Beispiele und werden auch zur Demonstration größerer Anwendungsfälle verwendet.

## 1.2. Gliederung

Das folgende Kapitel 2 beschreibt die Aufgabenstellung und Lösungsansätze der automatischen Modell-Generierung aus C#-Programmen. Es fasst auch die hier benötigten formalen Definitionen und Sprachen aus meiner Belegarbeit [Pit03] zusammen. An einem kleinen Beispiel wird die Funktion des Programms `cs2ccmb` illustriert, das diese Modellgenerierung implementiert.

Die Architektur und interne Arbeitsweise von `cs2ccmb` dokumentiert Kapitel 3. Es richtet sich weniger an Anwender, sondern vor allem an Entwickler, die den Funktionsumfang von `cs2ccmb` erweitern oder ändern möchten.

Kapitel 4 formalisiert das Konzept „Verhaltensgleichheit“. Es stellt die in der Literatur üblichen Methoden und Definitionen und die für diese Arbeit notwendigen Erweiterungen vor und beschreibt den verwendeten Algorithmus. Dieser wurde in dem Programm `sim` implementiert, dessen Benutzung und Architektur kurz dokumentiert wird.

Kapitel 5 stellt einige größere Anwendungsfälle aus der PKW-Sitzsteuerung vor, die diese Arbeit motiviert hat.

Schließlich folgt in Kapitel 6 noch ein Ausblick über mögliche Fortführungen dieser Arbeit, danach einige technische Hinweise zum Projektarchiv, die Selbstständigkeitserklärung und das Copyright.

## 1.3. Danksagung

Mein Dank gilt den Mitarbeitern des Lehrstuhls für technische Informationssysteme, an dem ich in sechs Semestern als studentische Hilfskraft gearbeitet habe. In dieser Zeit habe ich sehr viel gelernt, neue spannende Aufgabenstellungen der Informatik entdeckt und viel Spaß gehabt. Aus der Zusammenarbeit mit den Mitarbeitern ergaben sich auch die Themen meiner Beleg- und dieser Diplomarbeit. Besonders möchte ich mich bei folgenden Personen bedanken:

- Prof. Klaus Kabitzsch, der mir diese schöne Zeit und viele Freiheiten ermöglicht hat,
- Dipl.-Inf. Gunnar Stein, der mir während meiner Arbeit am Lehrstul, dem Großen Beleg und dieser Diplomarbeit durch viele Ideen, Diskussionen und Hilfe zur Seite gestanden hat,
- Prof. Horst Reichel für viele interessante Vorlesungen und Hinweise zu dieser Diplomarbeit,
- Annett Kittel, Mathias Wellner und Stefan Vogelsang für das Korrekturlesen,
- und ganz besonders meinen Eltern, die mir dieses Studium ermöglicht haben.

## 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 vorzubeugen, werden die verwendeten Operationen hier kurz definiert. Dabei seien $A, B, x, y \in \Omega$ Mengen und $R$ eine binäre Relation.

| | | | |
|---|---|---|---|
| $\varnothing$ | $=_{def}$ | $\{x \mid x \neq x\}$ | (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) |
| $xRy$ | $=_{def}$ | $(x, y) \in R$ | (Enthaltensein in einer binären Relation $R$) |

## 1.5. Warenzeichen

Alle in dieser Arbeit genannten Unternehmens- und Produktbezeichnungen sind in den meisten Fällen geschützte Marken- oder Warenzeichen. Die Wiedergabe von Marken- oder Warenzeichen in dieser Diplomarbeit berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass diese als frei von Rechten Dritter zu betrachten wären. Alle erwähnten Marken- oder Warenzeichen unterliegen uneingeschränkt den länderspezifischen Schutzbestimmungen und den Besitzrechten der jeweiligen eingetragenen Eigentümer.

# 2. Modellgenerierung

Dieses Kapitel präzisiert die Aufgabenstellung der automatischen Modellerzeugung, diskutiert verschiedene vorhandene Systeme, führt die verwendete Modellierung ein und beschreibt die Lösungsansätze der automatischen Modellgenerierung.

## 2.1. Aufgabenstellung

Um das Verhalten zweier Komponenten (einer Implementierung und einer Abstraktion) vergleichen zu können, muss von beiden eine formale Beschreibung des Verhaltens vorliegen. Diese liegt momentan in C#-Klassen vor, die in dem Framework „CCM“ implementiert wurden. Abb. 2.1 zeigt eine C#-Klasse eines sehr einfachen Schalters, der im Folgenden als Beispiel dienen soll (siehe Abschnitt 2.3 für die detaillierte Erläuterung).

```
1  using System;
2  using CCM;
3
4  namespace CCM.Library {
5
6      public class Switch : Comp {
7          VFS OUT = new VFS(2);
8          bool on, check;
9
10         public Switch( bool check ) {
11             on = false;
12             this.check = check;
13             OUT.init();
14         }
15
16         [Event] public void setState( bool on ) {
17             this.on = on;
18             EventOccured();
19         }
20
21         protected override void Run() {
22             // check for invalid state
23             if( check && OUT.StateFb > 1 )
24                 Common.log( this, "invalid StateFB!" );
25
26             if( on )
27                 OUT.State = 1;
28             else
29                 OUT.State = 0;
30         }
31     }
32 }
```

**Abb. 2.1**: C#-Beispiel eines einfachen Schalters `Switch`

Allerdings ist der Beweis von Aussagen über Programmcode unpraktikabel und auch ungeeignet:

- Der Programmcode beschreibt die komplette Funktionsweise einer Komponente; um zu entscheiden, ob eine Komponente durch eine andere austauschbar (äquivalent) ist oder eine andere abstrahiert, ist jedoch nur das von außen beobachtbare Schnittstellenverhalten relevant.
- Für Programmiersprachen geeignete Logiken (Higher Order Logic, Hoare-Kalkül) sind unentscheidbar – somit höchstens interaktiv beweisbar – und nur mit recht großem Aufwand erlernbar. Ziel dieser Arbeit ist jedoch ein einfacher nichtinteraktiver Nachweis (oder Gegenbeweis) der Äquivalenz zweier Komponentenbeschreibungen.

Deshalb muss ein Modell der Komponente generiert werden, das nur noch das relevante Verhalten beschreibt, hier also die Kommunikation mit anderen Komponenten. Dazu wird die in [Pit03] eingeführte Prozessbeschreibungssprache „CCMB“ verwendet, die für die Schnittstellen- und Verhaltensbeschreibung von Feldbuskomponenten entwickelt wurde. Abb. 2.2 zeigt eine CCMB-Beschreibung der schon erwähnten Komponente `Switch` (mit Konstruktorparameter `true`); ein Modell dieser Art soll durch ein zu entwickelndes Werkzeug automatisch aus dem Quelltext ermittelt werden[1].

## 2.2. Ähnliche Systeme

Die Erzeugung formaler Modelle direkt aus Programmcode ist noch nicht sehr weit verbreitet, jedoch gibt es einige Systeme, die ähnliche Ansätze verfolgen. Einige Beispiele:

- Visser et al. beschreiben in [VHB+03] die Verifikation von Java-Programmen, bei der der erzeugte Bytecode in einer speziellen virtuellen Maschine läuft, die während der Laufzeit ein Zustandsmodell des Programms erzeugt.
- Das System „SPIN“ [Hol97] beherrscht ebenfalls verschiedene Methoden zur Verifikation eines gegebenen Modells. Neuere Versionen bieten auch die automatische Modellerzeugung aus C- oder Java-Code an.

Diese Systeme haben jedoch Ziele, die von jenen dieser Arbeit konzeptionell stark abweichen: sie verifizieren die Korrektheit des Codes selbst, hingegen soll in dieser Arbeit ein Modell des von außen sichtbaren Schnittstellenverhaltens generiert werden.

Ein Ansatz von einer anderen Seite ist das sogenannte „Code Reverse Engineering“ in vielen UML-Werkzeugen. Diese erlauben es, schon fertigen Code zu parsen und daraus eine Schnittstellenbeschreibung in Form eines UML-Klassendiagramms zu erstellen. Jedoch berücksichtigen diese Werkzeuge nicht das Verhalten einer Komponente.

Es gibt recht viele Systeme (z. B. [TMTG03]), die in der „konstruktiven“ Richtung arbeiten, also Programmcode aus einem gegebenen Zustandsautomaten erzeugen. Die umgekehrte (analytische) Richtung scheint jedoch nur sehr wenig verbreitet zu sein (es wurde keine relevante Literatur gefunden).

[1] In der vorliegenden Implementierung werden Aufrufe von `Common.log()` *nicht* mit in das Prozessmodell eingefügt, da es sich nicht um einen Stub-Aufruf, sondern nur um eine Debug-Meldung handelt. Der Aufruf wurde nur zur leichteren Nachvollziehbarkeit mit in das Prozess-Modell eingefügt.

```
Switch = OUT.init() ->
            ( OUT.StateFb=0 -> OUT.State:=0 -> SwOff
            | OUT.StateFb=1 -> OUT.State:=0 -> SwOff
            | OUT.StateFb=2 -> Common.log(Switch, "invalid StateFB!") -> OUT.State:=0 -> SwOff
            )

SwOff = ( setState(False) ->
            ( OUT.StateFb=0 -> OUT.State:=0 -> SwOff
            | OUT.StateFb=1 -> OUT.State:=0 -> SwOff
            | OUT.StateFb=2 -> Common.log(Switch, "invalid StateFB!") -> OUT.State:=0 -> SwOff
            )
        | setState(True) ->
            ( OUT.StateFb=0 -> OUT.State:=1 -> SwOn
            | OUT.StateFb=1 -> OUT.State:=1 -> SwOn
            | OUT.StateFb=2 -> Common.log(Switch, "invalid StateFB!") -> OUT.State:=1 -> SwOn
            )
        | OUT.StateFb=0 -> OUT.State:=0 -> SwOff
        | OUT.StateFb=1 -> OUT.State:=0 -> SwOff
        | OUT.StateFb=2 -> Common.log("Switch", "invalid StateFB!") -> OUT.State:=0 -> SwOff
        )

SwOn = ( setState(False) ->
            ( OUT.StateFb=0 -> OUT.State:=0 -> SwOff
            | OUT.StateFb=1 -> OUT.State:=0 -> SwOff
            | OUT.StateFb=2 -> Common.log(Switch, "invalid StateFB!") -> OUT.State:=0 -> SwOff
            )
        | setState(True) ->
            ( OUT.StateFb=0 -> OUT.State:=1 -> SwOn
            | OUT.StateFb=1 -> OUT.State:=1 -> SwOn
            | OUT.StateFb=2 -> Common.log(Switch, "invalid StateFB!") -> OUT.State:=1 -> SwOn
            )
        | OUT.StateFb=0 -> OUT.State:=1 -> SwOn
        | OUT.StateFb=1 -> OUT.State:=1 -> SwOn
        | OUT.StateFb=2 -> Common.log("Switch", "invalid StateFB!") -> OUT.State:=1 -> SwOn
        )
```

**Abb. 2.2**: CCMB-Modell des Schalters `Switch`

## 2.3. Design und Kontrollfluss einer CCM-Komponente

Das CCM-Framework stellt abstrakte Klassen bereit, um Komponenten und Stubs eines CCM-Netzwerkes leicht und konsistent zu implementieren. Es gibt auch den Kontrollfluss für alle enthaltenen Komponenten vor, so dass sich die Aufgabe einer Komponenten-Klasse auf die Ereignisbehandlung beschränkt.

Anhand des `Switch`-Beispiels (Abb. 2.1) werden die Bestandteile solch einer Klasse zusammen mit ihrer Interaktion mit dem Framework beschrieben, so dass der Leser die Semantik von Komponentenklassen verstehen kann. Diese Beschreibung soll auch als minimale Anforderung angesehen werden, die das CCM-Framework für korrekte Modelle erfüllen muss; daher wird hier darauf verzichtet, das gesamte CCM-Framework zu dokumentieren.

Eine C#-Quelltextdatei einer Komponente muss das CCM-Framework importieren (Zeile 2 im `Switch`-Beispiel) und eine von `CCM.Comp` abgeleitete Komponentenklasse definieren (Zeile 6). Die Klasse enthält folgende Bestandteile:

1. Instanzen aller benötigten Stubs (Zeile 7 deklariert einen Stub `OUT` als Instanz des Stub-Typs `VFS` [*value feedback sender*]); die Parameter eines Stubs werden mit Konstruktor-Argumenten übergeben (im Beispiel gibt die Zahl die maximale Zustandsnummer (2) an)

2. Felder, die den Zustand der Komponente speichern (Zeile 8, der Schalter speichert in `on`, ob er ein- oder ausgeschaltet ist und in `check` den Konstruktorparameter, ob illegale `StateFb`-Werte gemeldet werden sollen)
3. beliebig viele Konstruktoren, die den Komponentenzustand (aus 2.) und die Stubs initialisieren (Zeilen 10-14)
4. Methoden, die Eingabeereignisse repräsentieren und verarbeiten; sie müssen `public` und mit dem Attribut `[Event]` versehen sein (siehe auch Abschnitt 2.4). Diese Methoden aktualisieren anhand ihrer Parameter den Klassenzustand und teilen dem Framework mit, dass ein Ereignis stattgefunden hat, indem am Ende die geerbte Methode `EventOccurred()` aufgerufen wird. (Zeilen 16 bis 19)
5. Methoden, die nur intern verwendet und nicht von außen aufgerufen werden und daher kein externes Ereignis repräsentieren; diese dürfen nicht mit `Event` attributiert werden und sollten auch nicht öffentlich (`public`) sein. (im Beispiel nicht vorhanden)
6. Die Ereignis-Bearbeitung `protected override void Run()`, die durch den Zustandswechsel anstehende Ausgabe-Ereignisse (Aufruf von Stub-Methoden, Schreiben von Stub-Propertys) generiert. `Run()` wird zu folgenden Zeitpunkten aufgerufen:
   - einmalig nach dem Start der Komponente nach Registrierung im CCM-Framework (aus Komponentensicht nach der Ausführung eines Konstruktors)
   - nachdem eine `[Event]`-Methode `EventOccurred()` aufruft
   - beim Eintreffen eines Kommunikations-Ereignisses von anderen Komponenten
   - nach Ablauf eines Timers; da diese mit einer statischen Analyse schwer zu modellieren sind, bleiben Timer in dieser Arbeit unberücksichtigt. Wenn eine Komponente Timer enthält, ist das generierte Modell somit (möglicherweise) unvollständig!

## 2.4. Metainformationen

Da die Analyse nicht ein vollständiges konkretes Netzwerk inklusive dem CCM-Framework betrachten soll, sondern nur die Quelltexte einer konkreten Komponente und ihrer Stubs, ist es nötig, die ursprünglichen Klassen mit Metainformationen anzureichern. C# erlaubt das Anheften solcher Informationen an Klassen, Methoden, Felder und Funktionsparameter in Form von „Attributen“, die in eckige Klammern vor eine Deklaration geschrieben werden.

Ein solches Attribut `[Event]` wurde in der `Switch`-Klasse schon verwendet, um Ereignis- und Hilfsmethoden voneinander zu unterscheiden.

Eine andere wichtige Information, die vom Komponenten-Entwickler bereitgestellt werden muss, ist der real verwendete Wertebereich der Stub-Propertys und -Methoden sowie der Parameter von `[Event]`-Methoden. Diese Ereignisse werden außerhalb der Komponente generiert, daher können die möglichen Werte nicht durch eine Analyse ermittelt werden, die sich nur auf die Komponente beschränkt.

Eine Wertebereichsangabe erfolgt durch das Attribut `[Range(`*min*`,`*max*`,`*step*`)]`. *min* und *max* geben dabei den kleinst- und größtmöglichen Wert an, *step* die zu verwendende Schrittweite. Fehlt der *step*-Parameter, wird `1` bzw. `1.0` angenommen. Alle Parameter werden entweder als (numerische) Konstanten angegeben oder als String, der den Namen eines Klassenfeldes enthält; im letzteren Falle wird bei der Auswertung des Attributes der aktuelle Wert des Klassenfeldes verwendet. C# verbietet die Verwendung nichtkonstanter Ausdrücke in Attributen, so dass der Umweg über einen String nötig ist.

Damit die attributierten Komponenten kompiliert werden können, müssen alle verwendeten Attribute definiert werden. Dies geschieht in der Datei `CCMAttributes.cs`, die im Projekt-Archiv im Verzeichnis `src/` zu finden ist (siehe Kapitel 7).

## 2.5. Stub-Klassen

Stubs sind Klassen, die einen funktionell zusammengehörigen Teil einer Schnittstelle kapseln und implementieren. So kann das Kommunikations-Verhalten einer Komponente abstrakter und einfacher implementiert werden.

Das ursprüngliche Ziel war, die Analyse auf die Komponentenklasse zu beschränken. Dies führt allerdings dazu, dass Wertebereiche von Stub-Propertys und -Methoden irgendwo in der Komponente angegeben werden müssten, was umständlich und fehlerträchtig ist. Zum anderen müssten diese Angaben in jeder analysierten Komponente wiederholt werden; dies wäre redundant, weil die Wertebereiche eines Stub-Members (meist) nicht von der ihn verwendenden Komponente, sondern von den Stub-Parametern abhängen. Der natürliche Ort zur Angabe dieser Wertebereiche ist somit die Stubklasse selbst.

Ein anderer Nachteil wäre, dass zur Analysezeit keinerlei Fehlerprüfung bezüglich Typkompatibilität, Methodenexistenz etc. möglich wäre.

Aus diesen Gründen wurde entschieden, Stubs mit in die Analyse einzubinden. Von diesen Stub-Klassen werden jedoch nur die Typen, Namen und Wertebereiche der öffentlichen Felder und Methoden benötigt, die eigentliche Implementierung bleibt dabei unberücksichtigt. Es genügt, eine stark vereinfachte Form der Stubklasse zur Analyse zu verwenden, in der die Methodenkörper und alles Unwesentliche entfernt sind (siehe dazu auch Abschnitt 3.1.2). Abb. 2.3 zeigt als Beispiel den von `Switch` verwendeten Stub `VFS` mit vollständiger Wertebereichs-Attributierung; diese Klasse ist die zum Zweck der Analyse vereinfachte „Spezifikation“ des in der PKW-Sitzsteuerung verwendeten Stubs.

## 2.6. Beschreibung der Modelle

### 2.6.1. Kripke-Struktur

Als mathematische Beschreibung eines Verhaltensmodells wird die in der Informatik und Ingenieurswelt weit verbreitete Kripke-Struktur (auch bekannt als „markiertes Transitionssystem“, „labeled transition system“ (LTS), „Zustandsübergangsmodell“ oder „endlicher Automat“) verwendet.

```
1  using System;
2  using System.Collections;
3
4  using CCM;
5
6  namespace CCM.Seat
7  {
8      public class VFS: StubS
9      {
10         byte cfgStates = 3;
11
12         public VFS(byte cfgStates) {
13             this.cfgStates = cfgStates;
14         }
15
16         override public void init() {}
17
18         public void setStub(VFR vfr) {}
19
20         [Range(0, "cfgStates")] public byte CfgStates;
21
22         [Range(0, "cfgStates")] public byte State;
23
24         [Range(0, "cfgStates")] public byte StateFb;
25
26         [Range (−1.5, 1.5, 0.5)] public float ContFb;
27
28         [Range (−1.5, 1.5, 0.5)] public float Cont;
29
30         public bool checkStateFeedback(int time) {}
31
32         public bool stateFbChanges(byte from, byte to) {}
33
34         public bool stateFbChanges() {}
35
36         public override void Snapshot() {}
37     }
38 }
```

**Abb. 2.3**: C#-Beispiel des vereinfachten Stubs `VFS`

Eine KRIPKE-Struktur ist ein über einer Menge möglicher Transitionen $\Delta$ definiertes Tripel

$$\mathcal{F}^{\Delta} = (S, s^0 \in S, next \subseteq S \times \Delta \times S)$$

aus einer Zustandsmenge $S$, einem Startzustand $s^0$ und einer Relation der möglichen Zustandsübergänge $next$ [2]. $(s, \delta, s') \in next$ bedeutet, dass im Zustand $s$ die Transition $\delta$ möglich ist und in den Nachfolgezustand $s'$ führt. Oft wird in der Literatur dafür die abkürzende (und intuitive) Schreibweise $s \xrightarrow{\delta} s'$ verwendet.

Über dieser Struktur wird später die Definition und der Nachweis der Verhaltensgleichheit durchgeführt. Allerdings sollen die generierten Modelle nicht direkt in dieser mengentheoretischen Form, sondern wie im `Switch`-Beispiel (Abb. 2.2) gezeigt in der Prozessalgebra CCMB notiert werden, da diese wesentlich übersichtlicher und leichter zu lesen ist.

[2] Die in [Pit03] eingeführte Abbildung $V$ zur Zuordnung von atomaren Zustandsprädikaten wird hier nicht berücksichtigt, da sie für den Nachweis der Verhaltensgleichheit nicht benötigt wird.

### 2.6.2. Transitionen $\Delta$

Transitionen sind entweder interne Operationen oder Kommunikationsschritte mit anderen Komponenten; sie lassen die Komponente in einen anderen Zustand übergehen. Die Kommunikation von Komponenten erfolgt ausschließlich über Stubs.

Die in CCMB definierten Transitionsarten manifestieren sich daher im CCM-Framework als Methodenaufrufe und Lesen/Setzen von Propertys eines Stub-Objekts:

**Ausgabe:** Die Komponente setzt eine Property eines Stubs (z. B. `OUT.State = 1;`) oder ruft eine Stub-Methode auf (z. B. `OUT.init();`)

**Eingabe:** Von außen wird eine `[Event]`-Methode aufgerufen (z. B. `setState(true)`) oder die Komponente fragt eine Stub-Property ab (z. B. `if(Out.StateFb > 1) {...}` oder `int n = OUT.StateFb;`) oder verwendet den Rückgabewert eines Methoden-Aufrufs. Dabei ist zu beachten, dass bei der Analyse *alle* möglichen Parameter-, Property- und Rückgabewerte zu berücksichtigen sind. Um die Anzahl der Kombinationen zu begrenzen und unmögliche Werte auszuschließen, müssen die Wertebereiche durch Attribute eingeschränkt werden.

In der aktuellen Implementierung des CCM-Frameworks werden vor dem Aufruf von `Run()` die Werte aller Stub-Propertys in einem „Schnappschuss“ festgehalten. Dies bedeutet, dass sich die Property-Werte innerhalb der `Run()`-Methode nicht ändern und dass keine Fehler beim gleichzeitigen Zugriff mehrerer Komponenten auf einen Stub (“race conditions“) auftreten können.

**Interne Operation `tick`:** Da bei der statischen Analyse keine Aussagen über Laufzeiten gemacht werden können, werden interne Ereignisse fester Dauer in dieser Arbeit nicht berücksichtigt.

Die Menge aller möglichen Transitionsausdrücke wird als $\Delta$ bezeichnet.

### 2.6.3. Prozessalgebra CCMB

Die Sprache CCMB wurde in [Pit03] definiert; hier soll nur eine Zusammenfassung der Syntax und Intuition gegeben werden. Da atomare Zustandsprädikate in dieser Arbeit nicht benötigt werden, sind sie hier auch nicht mit aufgeführt.

Eine Prozessalgebra ist eine mathematische Sprache zur einfacheren Beschreibung einer Kripke-Struktur. Das Verhalten einer Komponente wird beschrieben durch die Definition einer Menge von „Prozesskonstanten“ $KON$. Eine Prozesskonstante kann als Name eines Zustands verstanden werden und ihr definierter Wert als Definition aller möglichen Transitionen, die in diesem Zustand möglich sind. Da es in der Verhaltensbeschreibung (meist) sehr viele „Zwischenzustände“ gibt, auf die man sich nicht namentlich beziehen muss (z. B. als Zielzustand mehrerer Transitionen), können diese Zwischenzustände anonym bleiben, d. h. sie benötigen keine explizite Prozesskonstante. Daher erlaubt eine Prozessalgebra nicht nur Ausdrücke in Form von $Zustand = Transition \rightarrow Folgezustand$, sondern eine beliebige Anzahl nacheinander auszuführender Transitionen mit anonymen Zwischenzuständen.

Die Syntax von Prozessausdrücken ist wie folgt definiert (ISO-EBNF):

$$Process \; = KON \;\Big|\; \text{„0“} \;\Big|\; \Delta \text{ „}\rightarrow\text{“ } Process \;\Big|\; Process \text{ „}|\text{“ } Process$$

- 0 ist der „Nullprozess“ (Stop, Deadlock), der keine Transitionen ausführen kann.
- Der Prozess $\delta \rightarrow P$ führt die Transition $\delta$ aus und verhält sich dann wie die Prozesskonstante $P \in KON$.
- $P_1|P_2$ verhält sich entweder wie $P_1$ oder wie $P_2$. Beginnen die beiden Prozesse mit unterschiedlichen Transitionen, so ist die Alternativen-Auswahl deterministisch (dies ist in dieser Arbeit immer der Fall).

Außerdem ist die in der Mathematik übliche Klammerung von Teilausdrücken erlaubt.

Genutzt werden diese Prozessausdrücke in definierenden Gleichungen $KON = Process$. Eine Menge dieser Gleichungen stellt eine Verhaltensbeschreibung einer Komponente dar, wenn alle in den Prozessen referenzierten Konstanten eine Definition besitzen.

Das schon erwähnte Modell für `Switch` (Abb. 2.2) zeigt ein Beispiel einer kompletten Prozessbeschreibung mit den Prozesskonstanten $KON_{\texttt{Switch}} = \{Switch, SwOff, SwOn\}$.

### 2.6.4. Semantik von C#-Klassen

Wie bestimmt nun ein im CCM-Framework implementiertes C#-Programm die Elemente einer Kripke-Struktur bzw. einer Prozessbeschreibung?

Der Zustandsraum einer Komponente ist bestimmt durch die möglichen Werte ihrer Klassenfelder und durch die aktuelle Position in einem Kommunikationsablauf mit anderen Komponenten. In der Prozessbeschreibung ist es zweckmäßig, als Prozesskonstante (Element der Menge $KON$) ein Tupel der Werte aller Klassenfelder zu verwenden; dies sichert eine eindeutige Benennung. Da sich in einem Kommunikationsablauf die Klassenfelder nicht unbedingt nach jedem Schritt ändern (z. B. nach einer Ausgabe), bleiben diese Zwischenzustände anonym.

Im Startzustand sind sämtliche Klassenfelder nicht zugewiesen, d. h. mit ihren Default-Werten (`0`, `false`, `null`) belegt.

Transitionen und Wertänderungen der Klassenfelder werden durch C#-Anweisungen in Methodenkörpern erzeugt. Der Klassenzustand (d. h. der Wert ihrer Felder) nach Ausführung einer Methode ist der Folgezustand der Transitionen des jeweils berechneten Prozesses.

Grundlage der Methodenauswertung ist die im ECMA-Standard 334 [ECM02] definierte Semantik von C#. Dabei ist allerdings zu beachten, dass Eingabetransitionen (Lesen von Stub-Propertys, Rückgabewerte von Stub-Methoden) kein eindeutiges Ergebnis, sondern eine Menge möglicher Resultate zurückliefern; diese Alternativen müssen in dem zu berechnenden Prozess durch den |-Operator miteinander kombiniert werden.

Ausgehend vom Startzustand wird zunächst der passende Konstruktor aufgerufen, der die Stubs und Klassenfelder initialisiert. Dies generiert eine Menge von Zuständen, in denen sich die Klasse nach Abarbeitung des Konstruktors befinden kann. In jedem dieser Zustände kann nun eine `[Event]`-Methode oder direkt die `Run()`-Methode aufgerufen werden, was wiederum neue oder schon bekannte Zustände erzeugen kann.

## 2.7. Modellgenerierung durch `cs2ccmb`

Das Programm „`cs2ccmb`“ (C#-zu-CCMB-Konverter) ist ein im Rahmen dieser Arbeit entstandenes Werkzeug, das die oben beschriebene Modellgenerierung automatisch ausführt. Dieser Abschitt dokumentiert die zu Grunde liegenden Konzepte und die Verwendung; die detaillierte Beschreibung der Implementierung (die für Anwender nicht unbedingt wichtig und nötig ist), ist in Kapitel 3 zu finden.

### 2.7.1. Designentscheidungen

#### Eingabe

- zu analysierende Komponenten-Klasse
- alle Stub-Klassen, die von der Komponente benötigt werden
- konkrete Konstruktor-Argumente: Da es Komponenten gibt, die anhand dieser Argumente unterschiedliche Stubs instanziieren, wäre es unübersichtlich, ein Modell zu generieren, das alle möglichen Kombinationen erfasst. Zum anderen benötigt der Vergleich zweier Komponenten ohnehin eine kompatible Instantiierung, da zwei unterschiedlich parametrisierte Komponenten-Instanzen im Allgemeinen nicht verhaltensgleich sind.
- Ausgabeformat und -datei

**Parsing** Der erste Verarbeitungsschritt ist das Parsen des gesamten Quellcodes in einen die Klassenstruktur repräsentierenden Objektbaum. Dies benötigt zwar mehr Speicherplatz als die direkte Interpretation des Quellcodes, erlaubt aber eine wesentlich höhere Verarbeitungsgeschwindigkeit, weil die syntaktische Fehlerbehandlung nur einmal durchgeführt werden muss. Zum anderen ist so die Portierung auf eine andere Programmiersprache wesentlich einfacher.

**Analyse** Grundsätzlich existieren zwei verschiedene Semantiken von Programmiersprachen: die denotationale („Kompilierung“) und die operationale („Interpretation“).

Im Kontext dieser Arbeit würde Kompilierung bedeuten, eine Komponentenklasse direkt auf eine Kripke-Struktur (bzw. eine Prozessbeschreibung) abzubilden. Dies muss nur einmal getan werden, ist also sehr schnell. Diese Methode hat auch den Vorteil, dass sie unabhängig von konkreten Variablen-, Feld- und Stub-Property-Werten (die zur Analysezeit nicht ermittelt werden können) ist. Viele C#-Anweisungen wie Wertzuweisungen, If-Then-Bedingungen oder Schleifen mit fester Wiederholungsanzahl lassen sich auch direkt auf eine Prozessbeschreibung abbilden.

Es existieren aber auch viele Anweisungen, die sich nicht statisch zur Kompilierungs-Zeit entscheiden lassen, wie dynamische Verwaltung von Arrays, While-Schleifen, von Konstruktor-Parametern abhängige Stub-Instantiierungen, etc. Die schon existierenden Komponenten der PKW-Sitzsteuerung verwenden auch viele dieser nicht übersetzbaren Anweisungen, so dass kompilieren nicht möglich ist.

Die Interpretation des Codes kann problemlos sämtliche C#-Anweisungen verarbeiten, da nicht die Anweisungen selbst, sondern nur deren Wirkung auf den Klassenzustand (die Werte der Klassenfelder) in der Prozessbeschreibung gespeichert werden. Dabei muss der Code allerdings für jede betrachtete Alternative und für jeden Funktionsaufruf erneut interpretiert werden, die Laufzeit ist somit proportional zur Anzahl der möglichen Zustände (bei Endlosschleifen im Programmcode terminiert die Analyse überhaupt nicht). Zum anderen erfordert Interpretation, dass die Werte sämtlicher Variablen, Klassenfelder, Stub-Propertys und Methoden-Rückgabewerte genau bekannt sind; somit muss über alle möglichen Werte der Stub-Propertys und -methoden iteriert werden, was die Angabe von Wertebereichen erfordert.

**Modellspeicherung** Die generierte Prozessbeschreibung wird während der Analyse in einer Objekt-Netzwerkstruktur gespeichert und erst zum Schluss in eine ASCII-Form übersetzt. Dies erfordert wiederum mehr Speicherplatz als die sofortige CCMB-Ausgabe, erlaubt aber die direkte Weiterverwendung der Prozessbeschreibungen bei der Prüfung der Verhaltensgleichheit. Zudem ist das Hinzufügen anderer Ausgabeformate (z. B. in Prolog-kompatibler Syntax zur Prüfung von Netzwerkeigenschaften wie in [Pit03]) leicht möglich.

**Ausgabe** Enthalten die Klassen syntaktische oder andere Fehler, bricht das Programm mit einer entsprechenden Meldung, die Art und Ort des Fehlers beschreibt, sofort ab. Im Erfolgsfall wird das generierte Prozessmodell im festgelegten Format in eine Ausgabedatei geschrieben. Folgende Formate stehen zur Verfügung:

- **CCMB:** ASCII-Notation der Prozessalgebra CCMB (siehe Abschnitt 2.6.3).
- **Prolog:** Dieses Format kann in dem in [Pit03] vorgestellten Theorembeweiser verwendet werden, um Eigenschaften formal zu beweisen/zu widerlegen.
- **Binär:** Die generierte Objektstruktur, die das Prozessmodell speichert, wird in binäre Dateien serialisiert; dies ermöglicht ein einfaches Einlesen der Daten in Programmen wie dem Bisimulations-Prüfer `sim` (s. Kapitel 4).

### 2.7.2. Ablauf der Modellgenerierung

Abb. 2.4 zeigt den detaillierten Ablaufplan der Modellgenerierung durch `cs2ccmb`. Aus Gründen der Übersichtlichkeit sind Parsing und Ausgabe nicht mit enthalten.

„TODO“ ist dabei eine Warteschlange für noch zu bearbeitende Zustände. Ein Folgezustand wird grundsätzlich am Ende der `Run()`-Methode erzeugt, wenn die Bearbeitung eines Ereignisses abgeschlossen ist. Wenn dieser Folgezustand noch nicht in der Menge der schon berechneten Prozesse vorhanden ist, wird er in diese Warteschlange gestellt.

Irgendwann erzeugen die `Run()`-Methodenaufrufe nur noch Zustände, die schon berechnet wurden und die Warteschlange wird geleert. Da die Auswertung erst terminiert, wenn die Warteschlange leer ist, sind somit sämtliche durch die Ereignisse erreichbaren Zustände als Prozesse erfasst.

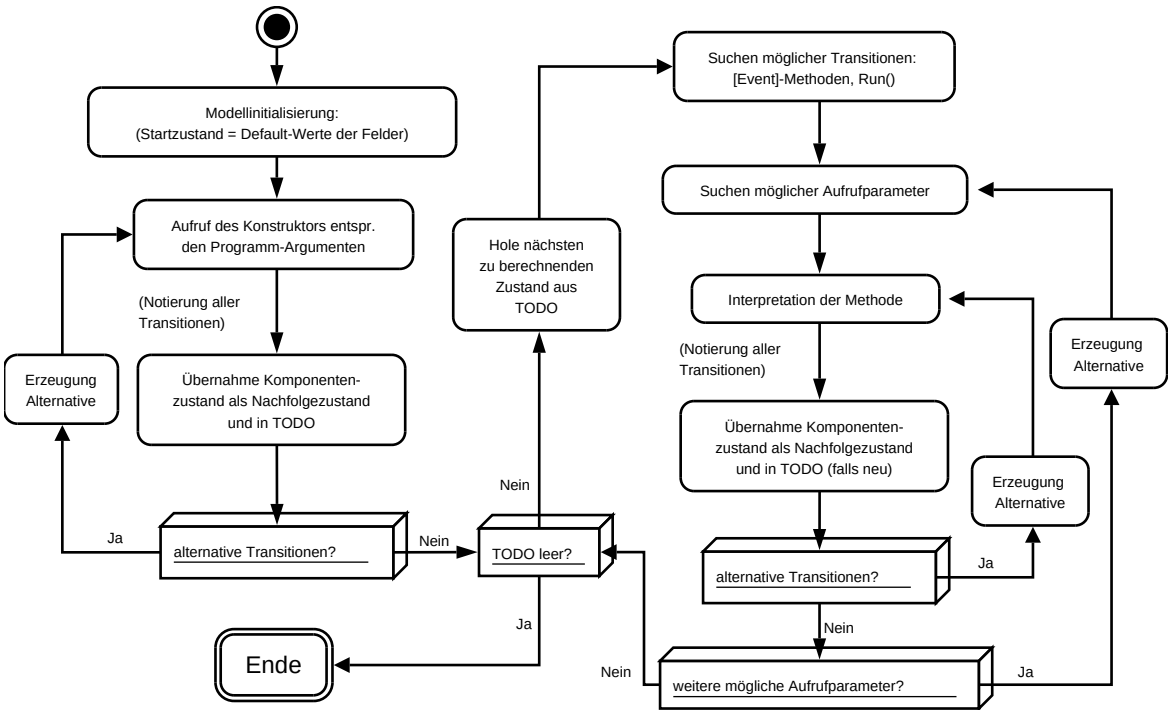



**Abb. 2.4**: Ablauf der Modellgenerierung mit `cs2ccmb`

### 2.7.3. Aufruf

`cs2ccmb` benötigt zur Ausführung eine installierte CLI (Common Language Infrastructure, eine von Microsoft definierte plattformunabhängige virtuelle Maschine), also eine installierte .NET[3]- oder Mono[4]-Plattform.

Momentan existiert für das Programm keine Benutzeroberfläche, der Aufruf und die Ausgabe erfolgen auf der Kommandozeile:

`cs2ccmb.exe` *Output StubFile StubFile ... CompFile* `--` *CtorPar CtorPar ...*

Zuerst wird der Ausgabemodus und die Ausgabedatei festgelegt. *Output* hat dabei eine der folgenden Formen: `-ccmb` *File* für CCMB-Syntax, `-prolog` *File* für Prolog-Syntax und `-bin` *File* für das binäre Serialisations-Format. *File* gibt den Dateinamen an; bei „`-`“ erfolgt die Ausgabe auf der Konsole.

Danach werden die Dateien übergeben, die die benötigten Stub-Spezifikationsklassen enthalten, und die Quelltextdatei *CompFile* mit der zu analysierenden Komponenten-Klasse.

Soll ein Konstruktor aufgerufen werden, der Parameter erwartet, so werden diese nach den zwei Strichen „`--`“ angegeben.

Als Beispiel sei die Analyse des schon zitierten Schalters demonstriert: Enthält die Datei `VFS.cs` den Stub `VFS` aus Abb. 2.3 und die Datei `Switch.cs` die Komponente `Switch` aus Abb. 2.1, dann erzeugt der Aufruf

```
cs2ccmb.exe -ccmb Switch.ccmb VFS.cs Switch.cs
```

[3] `http://www.microsoft.com/net` (Stand: 24. Juni 2004)
[4] `http://www.go-mono.org` (Stand: 24. Juni 2004)

die Fehlermeldung „Class.construct(): given arguments do not match any constructor“, weil es keinen Konstruktor ohne Argumente gibt. Mit

```
cs2ccmb.exe -ccmb Switch.ccmb VFS.cs Switch.cs -- true
```

wird ein Prozessmodell ähnlich der von Abb. 2.2 erzeugt und in die Datei `Switch.ccmb` geschrieben. Allerdings ignoriert `cs2ccmb` Aufrufe von `Common.log()`, da diese keine Stub-Kommunikation darstellen. Natürlich kann die automatische Analyse keine sinnvollen Zustandsnamen wie *SwOn* generieren, sondern die Zustandsnamen sind Wertetupel der Klassenfelder. Als Interpretationshilfe gibt `cs2ccmb` als erste Zeile die Namen der Klassenfelder in den Wertetupeln aus.

Deaktiviert man die `StateFb`-Prüfung durch den Konstruktor-Parameter `false`, vereinfachen sich die generierten Prozesse, da `StateFb` gar nicht mehr abgefragt wird, und `cs2ccmb` erzeugt folgende Ausgabe (die direkt auf die Konsole geschrieben wird):

```
$ cs2ccmb.exe -ccmb - VFS.cs Switch.cs -- false
State name format: ["check", "on", "OUT"]

Switch = OUT.init() -> OUT.State:=0 -> [False,False,"VFS"]

[False,True,"VFS"] = (
          setState(False) -> OUT.State:=0 -> [False,False,"VFS"]
        | setState(True) -> OUT.State:=1 -> [False,True,"VFS"]
        | OUT.State:=1 -> [False,True,"VFS"])

[False,False,"VFS"] = (
          setState(False) -> OUT.State:=0 -> [False,False,"VFS"]
        | setState(True) -> OUT.State:=1 -> [False,True,"VFS"]
        | OUT.State:=0 -> [False,False,"VFS"])
```

Abschließend sei das gleiche Beispiel in Prolog-Syntax gezeigt:

```
$ cs2ccmb.exe -prolog - VFS.cs Switch.cs -- false
% State name format: ("check", "on", "OUT")

processdef( switch, [send(out_init), out_State:=0, st(false,false,'VFS')] ).

processdef( st(false,true,'VFS'), [
          [send(setstate(False)), out_State:=0, st(false,false,'VFS')]
        + [send(setstate(True)), out_State:=1, st(false,true,'VFS')]
        + [out_State:=1, st(false,true,'VFS')]] ).

processdef( st(false,false,'VFS'), [
          [send(setstate(False)), out_State:=0, st(false,false,'VFS')]
        + [send(setstate(True)), out_State:=1, st(false,true,'VFS')]
        + [out_State:=0, st(false,false,'VFS')]] ).
```

Wird diese Ausgabe in eine Datei geschrieben, kann diese direkt mit dem existierenden CCMB-Theorembeweiser weiterverarbeitet werden. Dabei ist folgendes zu beachten:

- Zustandsnamen und der erste Bezeichner von Transitionen werden in Kleinbuchstaben konvertiert, da in Prolog ein Bezeichner, der mit einem Großbuchstaben anfängt, eine Variable darstellt.

- Anonyme Zustände (also alle außer dem Startzustand) sind mit dem Präfix „`st`“ versehen.
- Bei qualifizierten Bezeichnern (wie `OUT.State`) ist der Punkt durch einen Unterstrich (`_`) ersetzt, da der Punkt in Prolog eine andere syntaktische Bedeutung hat.
- Die ursprüngliche Definition von CCMB in [Pit03] sieht keine Methodenaufrufe ohne Rückgabewert vor. Diese werden deshalb in eine `send(...)`-Transition verpackt.

# 3. Implementierung von `cs2ccmb`

Dieses Kapitel beschreibt detailliert die Implementierung von `cs2ccmb`. Es ist deshalb primär für Entwickler interessant, die dieses Werkzeug erweitern oder ändern wollen; für Anwender reicht es aus, dessen Zweck und die zugrunde liegenden Konzepte zu kennen, die in Kapitel 2 beschrieben wurden.

## 3.1. Stand der Implementierung

Aus Zeitgründen konnte dieses Werkzeug nicht vollständig implementiert werden; es erlaubt momentan die Verarbeitung der Komponenten und Stubs der PKW-Sitzsteuerung. Um auch für zukünftige und andere Komponenten verwendbar zu sein, muss das Programm an vielen Stellen noch vervollständigt werden.

### 3.1.1. Sprachumfang

Folgende Elemente der Sprache werden momentan nicht unterstützt:

- Vorwärts-Referenzen; Das bedeutet, dass – im Unterschied zum C#-Standard – alle Bezeichner vor ihrer ersten Verwendung deklariert werden müssen. Dies ermöglicht das Parsen und die Typprüfung in einem Durchgang und vereinfacht den Parser beträchtlich.
- Pointer
- Typecasts
- mehrdimensionale Arrays
- anonyme Methoden
- Parameter-Arrays als Methodenargumente
- Ausdrücke, die folgende Konstruktionen enthalten: `base`, `typeof`, `sizeof`, `checked`, `unchecked`
- die Anweisungen `using`, `fixed`, `switch`, `foreach`
- die Typdeklarationen `struct`, `interface`, `enum`, `delegate`
- folgende Memberdeklarationen: Konstanten, Events, Indexer, Operatoren, Destruktoren, Propertys (letztere müssen in Stub-Spezifikationsklassen in einfache Felder umgewandelt werden)

### 3.1.2. Anforderungen an die Stub-Klassen

Die Stubklassen müssen für die Analyse mit Metainformationen versehen werden (`Range`-Attribute), die eigentliche Implementierung der Stub-Methoden soll unberücksichtigt bleiben und die momentan implementierten Klassen enthalten viele C#-Konstruktionen, die `cs2ccmb` momentan noch nicht versteht. Deshalb arbeitet die Analyse nicht mit den Stubklassen selbst, sondern mit „Spezifikationsklassen“ der Stubs, die sich von den eigentlichen Implementierungen wie folgt unterscheiden:

- alle Klassenfelder und -methoden sind durch Attribute in ihrem Wertebereich beschränkt,
- alle Propertys sind durch normale Klassenfelder ersetzt,
- und alle Methoden haben einen leeren Körper (bis auf Anweisungen im Konstruktor, die die Klassenfelder initialisieren).

Wird `cs2ccmb` später so erweitert, dass es den kompletten C#-Sprachumfang abdeckt, dann können die Spezifikationsklassen natürlich wegfallen und die Angabe der Wertebereiche direkt in den Stub-Implementierungsklassen vorgenommen werden.

## 3.2. Architektur-Übersicht

Das UML-Klassendiagramm 3.1 zeigt eine Übersicht der Module und Klassen, die eine Komponente repräsentieren und das Modell speichern (die zum Parsen benötigten Klassen sind nicht mit aufgeführt).

`cs2ccmb` besteht aus fünf großen Teilen:

**Parser** (`cs-parser.jay`, `cs-tokenizer.cs`, `report.cs`, `location.cs`): Scanner und Parser, die eine Quelltextdatei in eine repräsentierende Objektstruktur übersetzen.

**Code-Baum** (`class.cs`, `block.cs`, `expr.cs`, `decls.cs`): Klassen, die die Bestandteile eines C#-Programmes repräsentieren; werden vom Parser instanziiert.

**Iteratoren** (`iterator.cs`): Klassen, die für verschiedene Datentypen über einen gegebenen Wertebereich iterieren; dies schließt auch die Iteration über die möglichen Eingabewerte aller Stubs ein.

**Modell** (`process.cs`): speichert die schon berechneten und die noch zu berechnenden Prozesse („TODO“ im Ablaufdiagramm 2.4) der zu analysierenden Komponente.

**Hauptprogramm** (`cs2ccmb.cs`): steuert die Eingabe, Modellgenerierung und Ausgabe.

Die Klassen der Module „Code-Baum“ und „Modell“ werden in einer separaten Bibliothek `csmodel.dll` gehalten, da sie auch für das Programm zur Bisimulations-Prüfung (siehe Kapitel 4) benötigt werden. Diese gemeinsame Bibliothek ermöglicht auch die Serialisierung des generierten Prozessmodells.

Alle Klassen, Methoden und öffentlichen Felder wurden im Quellcode in javadoc-Syntax dokumentiert. Dieses Format hat ist als Quasi-Standard zur Quellcode-Dokumentation weit

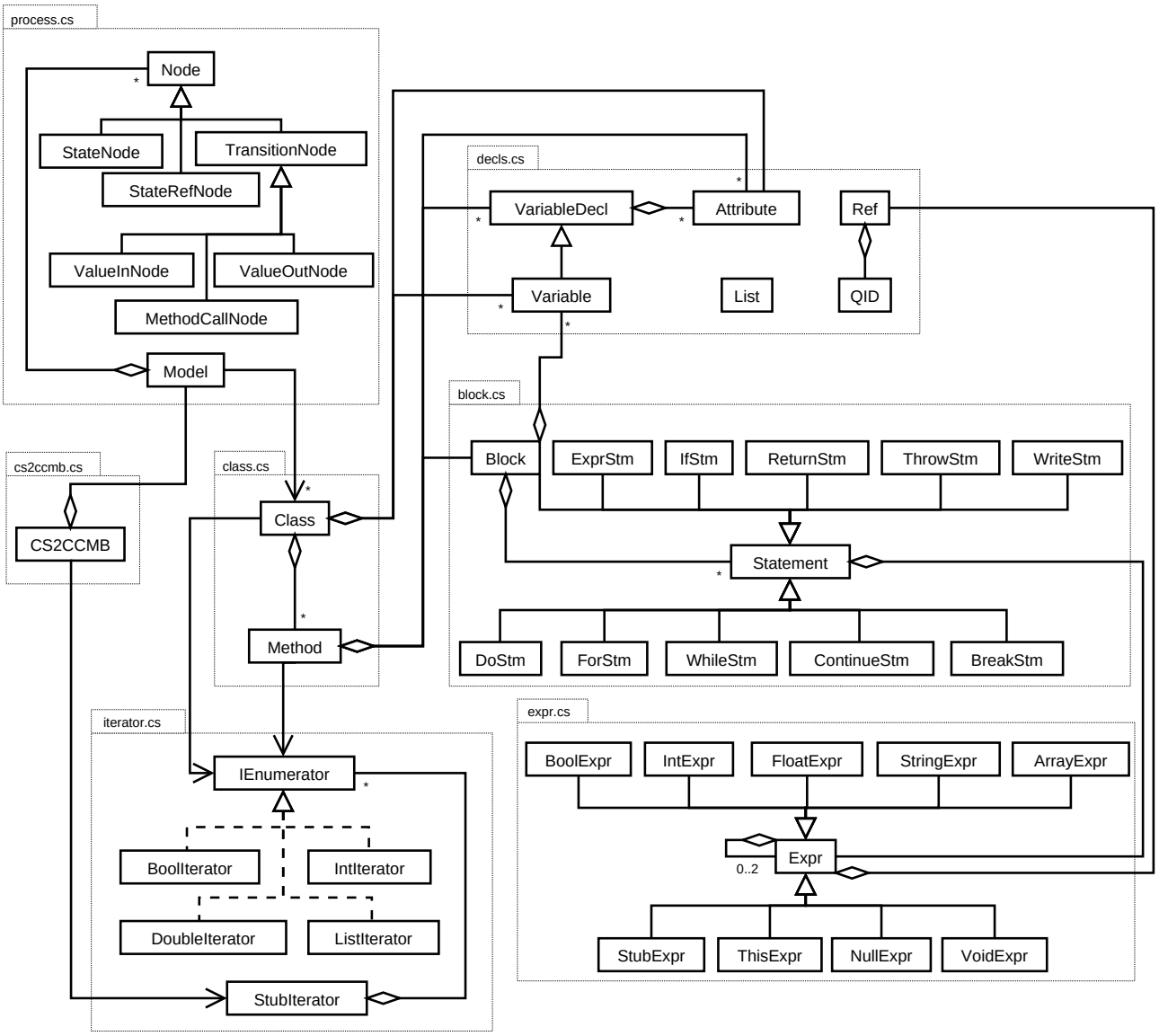



**Abb. 3.1**: cs2ccmb-Klassenübersicht

verbreitet und viele Programme (javadoc, doc++, kdoc, doxygen, cxref, etc.) verstehen es. Das Projektarchiv enthält im Verzeichnis `html/` eine durch doxygen[1] erzeugte HTML-Dokumentation, die eine Referenz für die weitergehende Entwicklung bietet.

Da diese Referenz recht umfangreich (die generierte PDF-Version umfasst über 200 Seiten!) und zum linearen Lesen ohnehin nicht geeignet und gedacht ist, wird darauf verzichtet, sie hier abzudrucken. Als Nachschlagewerk bei der Entwicklung ist die mit Hyperlinks versehende HTML-Version sicher bequemer und schneller zu verwenden.

[1] `http://www.doxygen.org` (Stand: 24.06.2004)

## 3.3. Parser

Um die lexikalische und grammatische Analyse einer C#-Klasse nicht von Grund auf neu entwickeln zu müssen, wurde auf die Quellen des freien Mono-Projekts zurückgegriffen. Dieses ist unter der GNU General Public License lizensiert; da `cs2ccmb` ebenfalls unter dieser Lizenz steht, ist die Wiederverwendung des Mono-Codes möglich.

Mono verwendet und enthält den Parser-Generator „Jay“, eine von den Mono-Entwicklern auf C# portierte Variante des bekannten „YACC“. Dieser übersetzt eine BNF-Grammatik mit Semantik in kompilierbaren C#-Code. Jay besitzt keine eigene Dokumentation, hat aber die gleiche Schnittstelle wie Berkeley YACC und dem dazu kompatiblen GNU Bison. Zu letzterem existiert ein ausführliches und sehr gut geschriebenes Handbuch [Fre02], das zum Verständnis von Jay sehr gut geeignet ist.

Konkret wurden von Mono folgende Dateien unverändert (bis auf die Umbenennung des Namespaces) übernommen:

- `cs-tokenizer.cs` stellt die Klasse `Tokenizer` für die lexikalische Analyse bereit. Die öffentliche Schnittstelle dieser Klasse ist von Jay vorgegeben; dem erzeugten Parser muss nur eine Instanz übergeben werden, um die Verwendung kümmert sich der von Jay erzeugte Code.
- `location.cs` enthält die Klasse `Location`, die eine Position im Quellcode speichert; diese ist für die Angabe der Position von Fehlern hilfreich.
- `report.cs` formatiert mit der Klasse `Report` Fehlermeldungen und Warnungen mit ihrer Position im Quellcode. `cs2ccmb` verwendet nur die Methode

  ```
  static void Error( int code, Location l, string text )
  ```

  und den Zähler `static public int Errors`.

Von Monos Parser in `cs-parser.jay` wurden nur die Grammatik-Regeln übernommen und die Semantik (der zu den Regeln gehörige Programmcode) vollständig neu geschrieben. Jede Regel, die ein C#-Konzept (Klassen, Methoden, Deklarationen, Anweisungen, Ausdrücke) beschreibt, erzeugt ein repräsentierendes Objekt der entsprechenden Klasse (`Class`, `Method`, `Variable`, `Statement`, `Expression`).

Die momentan vorliegende Grammatik enthält leider noch drei shift/reduce-Konflikte[2], deren Beseitigung äußerst aufwändig wäre. Umgangen wurde das Problem durch die Angabe von Präzedenzregeln[3] und spezifische Tests, ob Ausdrücke der betreffenden Regeln korrekt erkannt werden. Wird beim Aufruf von Jay die Option `-v` angegeben, wird eine Datei `y.output` erzeugt, in der der generierte Parsing-Automat und die auftretenden Konflikte detailliert aufgeführt werden.

[2] Dies sind beim Parsen auftretende Mehrdeutigkeiten, die entweder inhärent in der Grammatik enthalten sind oder die durch die konkrete Implementation des Parsers entstehen. Das schon erwähnte Bison-Handbuch enthält eine Einführung in Konflikte, die beim Parsen auftreten können: `http://www.gnu.org/software/bison/manual/html_node/Shift-Reduce.html` (Stand: 24.06.2004)
Das klassische – auch hier auftretende – Beispiel ist die mehrdeutige Interpretation der verschachtelten if-Anweisung „`if( e1 ) if( e2 ) s1; else s2;`“: Es muss in so einem Fall definiert werden, auf welches `if` sich das `else` bezieht.

[3] D. h. die Festlegung, wie stark Operatoren und Schlüsselwörter ihre Operanden binden; vergleichbar mit der „Punkt-vor-Strich“-Regel in der Mathematik.

## 3.4. Repräsentation des Komponenten-Programms

Alle Bestandteile des C#-Programms der zu analysierenden Komponente werden in zugehörigen Klassen gespeichert, die das Programm auch ausführen können. Die Struktur dieser Klassen ist in Abb. 3.1 und deren Funktionsweise in der HTML-Referenz dokumentiert, weshalb hier nicht jede Klasse einzeln beschrieben wird, sondern nur die für die Analyse wichtigen Aspekte.

### 3.4.1. Zustandsmanipulation

Der Zustand einer Klasse (d. h. deren Felder) wird initialisiert durch den Aufruf eines Konstruktors mit passenden Parametern (`Class.construct(IList args)`). Danach sind für die Analyse folgende Aktionen möglich:

- Abfrage des aktuellen Zustandes (`StateNode Class.getState()`); dieser wird als Zustandsknoten zurückgeliefert
- Setzen des aktuellen Zustandes (`void Class.setState(StateNode)`); dies ist erforderlich, um nach einer analysierten Alternative der Stub-Eingabewerte die nächste Variante auszuführen
- Aufruf einer Stub-Methode: Zunächst wird durch

  ```
  Method Class.getMethod(string name, IList args)
  ```

  eine passende Methode gesucht (die Angabe der Argumente ist wegen der Möglichkeit überladener Methoden nötig); diese kann dann durch `Method.execute(IList args)` ausgeführt werden.

### 3.4.2. Auflösung von Referenzen

Referenzen auf Variablen, Klassenfelder und Methoden werden in Objekten der Klasse `Ref` gespeichert. Diese ist zuständig für die Auflösung eines Namens zu einem `Variable`- bzw. `Method`-Objekt. Variablenwerte können direkt über die Property `Value` abgefragt und manipuliert werden, der Aufruf einer Methode erfolgt über `call(List args)`.

Referenzen auf lokale Variablen und Komponentenfelder werden dabei direkt bearbeitet, Referenzen auf Stub-Felder und -Methoden dagegen zur Klasse `StubIterator` delegiert (s. u.), die die aktuelle Alternative der Stub-Eingabewerte bereitstellt.

## 3.5. Stub-Iteration

Bei der Analyse werden sämtliche Kombinationen erlaubter Stub-Eingabewerte berücksichtigt, um sämtliche Verhaltensweisen einer Komponente zu erfassen.

Zum einen gibt es solche Alternativen beim Aufruf einer `[Event]`-Methode (Schritt „Suchen möglicher Transitionen“ in Abb. 2.4). Die Iteration über einen einzelnen Parameter erfolgt mit entsprechenden Instanzen von `BoolIterator`, `IntIterator` und `DoubleIterator`, die

Iteration über die komplette Parameterliste führt `ListIterator` durch. Diese Klassen implementieren alle die `IEnumerator`-Schnittstelle, sie können also wie alle anderen aufzählbaren Objekte z. B. in `foreach`-Konstrukten benutzt werden.

Zum anderen entstehen Alternativen bei der Abfrage einer Stub-Property oder des Rückgabe-Wertes einer Stub-Methode bei der Auflösung von Referenzen (s. o.). Die möglichen und aktuellen Werte werden durch die Singleton-Klasse `StubIterator` verwaltet. Die Iteration erfolgt über `void start()` und `bool next()`. `Ref` delegiert Stub-Aufrufe an die Methoden `stubInput()`, `stubOutput()` und `call()`, die die entsprechende Alternative zurückliefern und auch einen entsprechenden Knoten in das Prozessmodell einfügen (falls nicht schon einer mit dem aktuellen Wert existiert). Stub-Aufrufe werden dabei dynamisch in die Iteration eingefügt; damit ist sichergestellt, dass nur über wirklich verwendete Werte iteriert wird.

## 3.6. Hauptprogramm

Das Hauptprogramm `cs2ccmb.cs` implementiert schließlich den in Abb. 2.4 gezeigten Algorithmus. Zuerst werden die Kommandozeilen-Optionen ausgewertet und sämtliche Quelltexte durch den Parser in eine ausführbare Klassenstruktur übersetzt. Die Komponente wird durch Aufruf des Konstruktors initialisiert und alle auftretenden Stub-Alternativen bearbeitet. Danach berechnet die `Main`-Methode solange neu auftretende Zustände in der `TODO`-Warteschlange, bis diese leer ist.

# 4. Nachweis der Verhaltensgleichheit

Nachdem nun die Verhaltensbeschreibung der Komponenten als Prozess-Algebra-Ausdrücke und damit auch als Automaten-Struktur vorliegt, wird nun der Vergleich zweier solcher Beschreibungen behandelt. Dazu werden zunächst kurz die üblichen Methoden in der Literatur diskutiert, eine Methode ausgewählt und genau definiert und die Implementierung beschrieben. Die Methode wird wiederum anhand eines kleinen Anwendungsbeispiels demonstriert.

## 4.1. Mögliche Vergleichsrelationen

Es gibt mittlerweile eine sehr große Anzahl an Prozess-Algebren, die für verschiedene Anwendungsbereiche entwickelt wurden, darunter auch CCMB. Jedoch wird die Semantik (fast) aller dieser Prozess-Beschreibungssprachen auf KRIPKE-Strukturen wie in Abschnitt 2.6.1 abgebildet. Somit ist es sinnvoll, Eigenschaften nicht auf der syntaktischen, sondern auf der semantischen Ebene zu definieren und zu vergleichen; dies wird in der Literatur auch durchweg so getan, um Definitionen und Algorithmen für alle Arten von Prozess-Algebren verwenden zu können.

Daher wird nun vorausgesetzt, dass die Verhaltensbeschreibung von den zu vergleichenden Komponenten in KRIPKE-Strukturen vorliegt, so wie sie die CCMB-Semantik in [Pit03] liefert.

Allgemein werden Prozesse durch Betrachtung ihres beobachtbaren Verhaltens verglichen: Jedem Prozess $p$ wird eine Menge $\mathcal{O}(p)$ zugeordnet, deren Elemente als mögliche Verhaltensweisen betrachtet werden können. Damit lassen sich zwei Arten des Vergleichs betrachten:

- $q$ „implementiert“ die Spezifikation $p$, wenn $q$ sämtliche Verhaltensweisen von $p$ erlaubt. Dies bestimmt eine Ordnung

$$p \sqsubseteq_{\mathcal{O}} q \quad \Leftrightarrow \quad \mathcal{O}(p) \subseteq \mathcal{O}(q)$$

- Zwei Prozesse $p$ und $q$ sind verhaltensgleich, wenn ihre beobachtbaren Verhaltensweisen gleich sind. Dies bestimmt eine Äquivalenzrelation

$$p =_{\mathcal{O}} q \quad \Leftrightarrow \quad p \sqsubseteq_{\mathcal{O}} q \ \wedge \ q \sqsubseteq_{\mathcal{O}} p \quad \Leftrightarrow \quad \mathcal{O}(p) = \mathcal{O}(q)$$

Die konkrete Konstruktion von $\mathcal{O}(p)$ wird durch die Wahl der gewünschten Art der Verhaltensgleichheit bestimmt. In [vG01] werden 13 Möglichkeiten der Definition betrachtet und nach „Feinheit“ geordnet. Je „feiner“ eine Definition ist, desto weniger Prozesse werden von ihr identifiziert, d. h. als verhaltensgleich bewertet. Die Palette reicht dabei von der sehr

einfachen „Trace-Semantik“ (gröbste Definition), bei der nur die Menge aller möglichen Transitionsfolgen betrachet wird, bis zur feinsten Definition der „Bisimulation“, die als Kriterium die lokale Ununterscheidbarkeit in jedem möglichen Zustand verwendet.

Robin Milner verwendet in [Mil89] die Bisimulation als Standard-Vergleichsrelation. Da CCMB von der von Milner verwendeten Prozessalgebra „CCS“ inspiriert wurde, es sehr viele Arbeiten gibt, die sich speziell mit Bisimulation beschäftigen, und da es die „genauestmögliche“ (feinste) Definition der Verhaltensgleichheit ist, soll sie in dieser Arbeit verwendet werden.

## 4.2. Definition der (Bi)simulation

Der Name „Simulation“ ergibt sich aus der Tatsache, dass ein Prozess in jedem Zustand die möglichen Transitionen eines anderen Prozesses erzeugen, also den anderen Prozess simulieren kann. Die einfache Simulation entspricht einer konkreten Form der Implementierungsrelation $\sqsubseteq_{\mathcal{O}}$. Wenn die Eigenschaft in beiden Richtungen gilt, d. h. wenn beide Prozesse die Transitionen des jeweils anderen simulieren können, dann sind die Prozesse äquivalent und man spricht von „Bisimulation“ (als konkrete Instanz der allgemeinen Äquivalenzrelation $=_{\mathcal{O}}$). Die formale Definition wurde aus [vG01] übernommen und an die hier verwendete Symbolik und an die explizite Unterscheidung von zwei verschiedenen Kripke-Strukturen angepasst.

Die vorgestellten Definitionen setzen voraus, dass $\mathcal{F}_1 = (S_1, s_1^0, next_1 \subseteq S_1 \times \Delta \times S_1)$ und $\mathcal{F}_2 = (S_2, s_2^0, next_2 \subseteq S_2 \times \Delta \times S_2)$ zwei Kripke-Strukturen über derselben Transitionsmenge $\Delta$ sind.

---

Eine Relation $R \subseteq S_1 \times S_2$ ist eine **Simulation**, wenn für alle Transitionen $\delta \in \Delta$ und Zustände $s_1, s_1' \in S_1$ und $s_2, s_2' \in S_2$ gilt:

$$(s_1, \delta, s_1') \in next_1 \ \wedge s_1 R s_2 \quad \Rightarrow \quad \exists s_2'.\ (s_2, \delta, s_2') \in next_2 \ \wedge \ s_1' R s_2'$$

Zustand $s_2$ **simuliert** $s_1$ (notiert als $s_1 \lesssim s_2$), wenn es eine Simulation $R$ gibt mit $s_1 R s_2$. Eine Kripke-Struktur $\mathcal{F}_2$ simuliert $\mathcal{F}_1$ ($\mathcal{F}_1 \lesssim \mathcal{F}_2$), wenn der Startzustand von $\mathcal{F}_2$ den Startzustand von $\mathcal{F}_1$ simuliert, d. h. $s_1^0 \lesssim s_2^0$.

---

---

Eine Relation $R \subseteq S_1 \times S_2$ ist eine **Bisimulation**, wenn für alle Transitionen $\delta \in \Delta$ und Zustände $s_1, s_1' \in S_1$ und $s_2, s_2' \in S_2$ gilt:

- $(s_1, \delta, s_1') \in next_1 \ \wedge s_1 R s_2 \quad \Rightarrow \quad \exists s_2'.\ (s_2, \delta, s_2') \in next_2 \ \wedge \ s_1' R s_2'$
- $(s_2, \delta, s_2') \in next_2 \ \wedge s_1 R s_2 \quad \Rightarrow \quad \exists s_1'.\ (s_1, \delta, s_1') \in next_1 \ \wedge \ s_1' R s_2'$

Zwei Zustände $s_1$, $s_2$ sind **bisimular** (notiert als $s_1 \sim s_2$), wenn es eine Bisimulation $R$ gibt mit $s_1 R s_2$. Zwei Kripke-Strukturen $\mathcal{F}_1$ und $\mathcal{F}_2$ sind bisimular ($\mathcal{F}_1 \sim \mathcal{F}_2$), wenn ihre Startzustände bisimular sind, d. h. $s_1^0 \sim s_2^0$.

---

Bisimularität bedeutet somit Simularität „in beide Richtungen“:

$$s_1 \lesssim s_2 \ \wedge \ s_2 \lesssim s_1 \quad \Leftrightarrow \quad s_1 \sim s_2$$

Intuitiv kann eine KRIPKE-Struktur eine andere simulieren, wenn sie alle Transitionen der anderen Struktur ausführen kann. Zwei KRIPKE-Strukturen sind bisimular, wenn sie in einem Zustand die gleichen Transitionen ausführen können und dies auch für alle möglichen Folgezustände gilt.

Diese Standard-Definitionen müssen für die Zwecke dieser Arbeit noch um einige Konzepte erweitert werden:

1. Die Prüfung der Verhaltensgleichheit soll nicht *irgendeine* Bisimulation ermitteln, sondern der Entwickler der abstrahierten Komponente soll im voraus spezifizieren, welche abstrakten Zustände welchen konkreten Zuständen bzw. Zustandsbereichen entsprechen; z. B. „`hot=true` abstrahiert den Bereich `temp=60` bis `temp=110`“. Im formalen Sinn definiert er damit eine Äquivalenzrelation $ZA$, die im folgenden **Zustands-Abstraktion** genannt wird.

2. Es ist sinnvoll, nicht nur die Zustände, sondern auch die Werte von Transitionen zu abstrahieren: wenn z. B. eine abstrahierende Komponente konkrete Eingabewerte nur noch in zwei Äquivalenzklassen einteilt (z. B. „schnell“ und „langsam“ oder „heiß“ und „kalt“), dann können Ausgabetransitionen nicht wieder den konkreten Wert liefern, da dieser im Zustand der Komponente nicht gespeichert wird; stattdessen sollte ein repräsentativer Wert der Äquivalenzklasse ausgegeben werden.

   Eine vom Komponenten-Entwickler anzugebende **Transitions-Abstraktion** muss beschreiben, welche Transitionswerte der Abstraktion welche Transitionswertebereiche der Implementierung repräsentieren. Diese Äquivalenzrelation wird mit $TA$ bezeichnet.

3. Es ist wünschenswert, für Analyse- und Testzwecke abstrakte Komponenten zu schreiben, die nur einen Teil der Funktionalität der implementierenden Komponente enthalten. Dazu wird ein **Zustandsfilter** genanntes Prädikat $\Phi$ über den Zuständen der implementierenden Komponente eingeführt und alle Zustände, die *nicht* in $\Phi$ liegen, müssen die (Bi)simulationsbedingungen nicht erfüllen.

Die formale Definition dieser Konzepte erfolgt separat, um sie besser mit der ursprünglichen Fassung vergleichen zu können und die Definition übersichtlich zu halten.

Die Berücksichtigung der extern vorgegebenen Zustands-Abstraktion erfolgt auf Basis der Standard-Bisimulation durch eine zusätzliche Einschränkung (nach [CS01]):

---

Sei $ZA \subseteq S_1 \times S_2$ eine Äquivalenzrelation.

Eine Relation $R \subseteq S_1 \times S_2$ ist eine $ZA$**-Bisimulation**, wenn $R$ eine Bisimulation ist und $R \subseteq ZA$.

---

Intuitiv sind somit zwei Zustände nur dann äquivalent (bisimular), wenn sie in der vom Komponentenentwickler spezifizierten Zustandsabstraktion enthalten sind.

Um die Transitions-Abstraktion zu integrieren, muss die ursprüngliche Definition so modifiziert werden, dass eine Transition in beiden Prozessen nicht mehr identisch, sondern äquivalent in $TA$ sein muss:

---

Sei $TA \subseteq \Delta \times \Delta$ eine Äquivalenzrelation über den Transitionen.

Eine Relation $R \subseteq S_1 \times S_2$ ist eine **$TA$-Bisimulation**, wenn für alle Transitionen $\delta_1, \delta_2 \in \Delta$ und Zustände $s_1, s_1' \in S_1$ und $s_2, s_2' \in S_2$ gilt:

- $(s_1, \delta_1, s_1') \in next_1 \ \wedge \ s_1 R s_2 \ \Rightarrow$
  $\exists s_2', \delta_2.\ (s_2, \delta_2, s_2') \in next_2 \ \wedge \ s_1' R s_2' \ \wedge \ \delta_1 \, TA \, \delta_2$
- $(s_2, \delta_2, s_2') \in next_2 \ \wedge \ s_1 R s_2 \ \Rightarrow$
  $\exists s_1', \delta_1.\ (s_1, \delta_1, s_1') \in next_1 \ \wedge \ s_1' R s_2' \ \wedge \ \delta_1 \, TA \, \delta_2$

---

Ein Zustandsfilter ist eine Vorbedingung für die Bisimulationsanforderungen:

---

Sei $\Phi \subseteq S_2$ ein Zustandsfilter-Prädikat.

Eine Relation $R \subseteq S_1 \times S_2$ ist eine **$\Phi$-Bisimulation**, wenn für alle Transitionen $\delta \in \Delta$ und Zustände $s_1, s_1' \in S_1$ und $s_2, s_2' \in S_2$ gilt:

- $s_2 \notin \Phi \ \wedge \ (s_1, \delta, s_1') \in next_1 \ \wedge \ s_1 R s_2 \ \Rightarrow \ \exists s_2'.\ (s_2, \delta, s_2') \in next_2 \ \wedge \ s_1' R s_2'$
- $s_2 \notin \Phi \ \wedge \ (s_2, \delta, s_2') \in next_2 \ \wedge \ s_1 R s_2 \ \Rightarrow \ \exists s_1'.\ (s_1, \delta, s_1') \in next_1 \ \wedge \ s_1' R s_2'$

---

Diese Konzepte ergeben zusammen dann die Definition der **$\Phi$-$TA$-$ZA$-Bisimulation**, die alle in dieser Arbeit notwendigen Konzepte enthält. Die erweiternden Konzepte lassen sich natürlich analog auf die einfache Simulation übertragen.

Es sei angemerkt, dass auch diese Definition noch erweitert werden kann. Hier wird die Abstraktion auf die Zusammenfassung von Werten des Zustands- und Transitionsraums zu Äquivalenzklassen beschränkt. Denkbar wäre auch eine Abstraktion der Zustandsübergänge, bei der sich eine abstrakte Transition in der Implementierung als Folge und/oder Alternative von Transitionen manifestiert (also wiederum als Prozess); diese Abstraktionsart wird „vertikale Bisimulation“ genannt (im Gegensatz zu der hier vorgestellten „horizontalen“ Bisimulation, bei der die Menge erlaubter Transitionen $\Delta$ beider Prozesse gleich sind) und in [RG98] vorgestellt. Die Verifikation ist dabei allerdings wesentlich aufwändiger.

Denkbar und praktisch interessant ist auch die Spezifikation ganzer Teilsysteme durch eine abstrahierende Komponente. Wie das Verhalten eines Netzwerkes aus den Spezifikationen der einzelnen Komponenten zu ermitteln ist, wurde schon in [Pit03] gezeigt. Dabei tritt das Problem auf, dass die Transitionen der einzelnen Komponenten oft parallel stattfinden können, was sich in der Prozessbeschreibung des ganzen Netzwerkes als alternative Ausführung aller

möglichen Reihenfolgen der Transitionen niederschlägt („Interleaving“). [HNW98] schlägt eine Methode vor, die Untersuchung auf einen „Repräsentanten“ der alternativen Transitionsabfolgen zu beschränken, was die Modelle drastisch verkleinern und damit die Verifikation beschleunigen kann und bei einer möglichen zukünftigen Implementierung dieser Erweiterung genutzt werden sollte.

## 4.3. Spezifikation in C#

Das Verhalten einer abstrakten Komponente soll genauso wie ihre Implementierung in einer C#-Klasse des CCM-Frameworks angegeben werden. Dies gewährleistet die Austauschbarkeit des Quellcodes und erlaubt mehrstufige Abstraktionen.

Der Unterschied zu einer Implementierungs-Klasse besteht nur darin, dass abstrahierende Klassen zusätzlich die Transitions- und Zustands-Abstraktionen enthalten müssen.

### 4.3.1. Spezifikation der Zustands-Abstraktion

Die Angabe der Äquivalenzrelation erfolgt wiederum durch Attribute. Ein Klassenfeld der abstrakten Klasse trägt eine Menge von `Eq(`$x$`,`$rg$`)`-Attributen, die jeweils einen repräsentativen Wert $x$ einem implementierenden Wertebereich $rg$ zuordnen, und optional ein Attribut `Abstracts("`$fieldname$`")`, das dem abstrakten Feld das implementierende Feld $fieldname$ zuordnet, falls die Namen nicht identisch sein sollten.

Der Wertebereich $rg$ hat die folgende Form:

- eine einfache Konstante (`0`, `2.5`, etc.); dies beschreibt einen Wertebereich mit nur einem Element
- der Name eines Klassenfeldes (das sich während der Abarbeitung nicht ändern sollte, d. h., das ein Komponenten-Parameter ist)
- ein geschlossenes Intervall `"[`$min$`,`$max$`]"`, d. h. den Bereich zwischen einschließlich $min$ und $max$; $min$ und $max$ dürfen dabei wiederum Konstanten oder Klassenfeld-Namen sein
- ein offenes Interval `"(`$min$`,`$max$`)"`, d. h. den Bereich zwischen $min$ und $max$, aber ohne die Grenzen selbst; hier darf $min$ zusätzlich den Wert `-oo` und $max$ den Wert `oo` annehmen, die „plus/minus unendlich“ symbolisieren
- ein halboffenes Intervall `"(`$min$`,`$max$`]"` bzw. `"[`$min$`,`$max$`)"` (wiederum mit offenen unendlichen Grenzen)

Nicht zusammenhängende Äquivalenzklassen werden durch eine Menge von Wertebereichen mit jeweils gleichem Repräsentanten beschrieben.

Für alle nicht attributierten Felder und nicht erfassten Wertebereiche wird die Identitäts-Relation angenommen.

### 4.3.2. Spezifikation der Transitions-Abstraktion

Das `Abstracts`-Attribut gibt wiederum bei ungleicher Feldbenennung den entsprechenden Namen des Feldes der Implementierung an, das den zu abstrahierenden Stub enthält.

Da Stubs mehrere Propertys enthalten können, die jede für sich eine Äquivalenzrelation benötigen, gibt es zusätzlich das Attribut `Property("`*name*`")`. Nach diesem Attribut beziehen sich die folgenden `Eq`-Attribute auf die Property bzw. die Methode *name* des deklarierten Stubs. Auf diese Weise kann jede Property und auch jeder Methoden-Rückgabewert nacheinander spezifiziert werden. Die Angabe der eigentlichen Wertebereiche erfolgt genauso wie in der Zustands-Abstraktion.

## 4.4. Anwendungsbeispiel

Die PKW-Sitzsteuerung enthält eine Komponente `Speed`, mit der die Geschwindigkeit des Fahrzeugs eingestellt werden kann. Der originale Quelltext wurde mit den notwendigen Attributen ergänzt; um das Beispiel einfach und übersichtlich zu halten, wurde der Geschwindigkeitsbereich auf 0 bis 80 festgelegt, mit Zwischenschritten von 20, um den hier abgedruckten Zustandsraum nicht zu groß werden zu lassen. In einer praktischen Analyse kann dies natürlich wesentlich verfeinert werden (Parameter `maxspeed`, Schritte von 1 km/h usw.). Abb. 4.1 zeigt die Implementierung und das durch `cs2ccmb` generierte Prozessmodell von `Speed`.

Eine sehr grobe Abstraktion `SpeedAbstr` dieser Komponente, die nur noch zwischen „Stop“ (`go=false`) und „fahrend“ (`go=true`, mit dem Transitions-Repräsentanten `OUT.Cont = 10`) unterscheidet, zeigt Abb. 4.2.

Das `Abstracts`-Attribut vor `bool go` bedeutet somit, dass `go` das Feld `speed` abstrahiert (wären die Namen identisch, könnte das Attribut wegfallen). Die `Eq`-Attribute geben die Zustandsabstraktion an: `go=false` gilt nur nur bei `speed=0` und `go=true` gilt im Bereich von nicht einschließlich 0 bis unendlich.

Die Attibutierung von `OUT` beschreibt, dass bei der Property `Cont` der Wert `0` ein Repräsentant für den (einelementigen) Wertebereich `0` ist und `10` den Wertebereich aller strikt positiven Geschwindigkeiten repräsentiert. Die Werte sind als Fließkommazahl angegeben, um zu verdeutlichen, dass `Cont` ein reeller Wert ist. `OUT` benötigt kein `Abstracts`-Attribut, da der entsprechende Stub in `Speed` ebenfalls `OUT` heißt.

## 4.5. Algorithmus

Cleaveland und Sokolsky stellen in [CS01] verschiedene Algorithmen zur effizienten Entscheidung der Bisimularität vor. Der Algorithmus „On-the-fly Preorder“ ist dabei für diese Arbeit als Grundlage am besten geeignet: er entscheidet $\Phi$-$ZA$-Bisimularität und erfordert nicht, dass die Kripke-Strukturen im voraus komplett vorliegen, sondern der jeweils benötigte Teil des Zustandsraumes kann lokal aus der jweiligen Prozessalgebra-Darstellung berechnet werden.

```
1  using System;
2  using CCM;
3
4  namespace CCM.Seat
5  {
6          public class Speed : Comp
7          {
8                  public VFS OUT = new VFS(0);
9
10                 float speed;
11
12                 protected override void Run()
13                 {
14                         OUT.Cont = speed;
15                 }
16
17                 [Event] public void setSpeed( [Range(0.0,80.0,20.0)] float speed)
18                 {
19                         this.speed = speed;
20                         EventOccured();
21                 }
22         }
23 }
```

```
State name format: ["speed", "OUT"]

Speed = OUT.Cont:=0 -> [0,"VFS"]

[80,"VFS"] = (
          setSpeed(0) -> OUT.Cont:=0 -> [0,"VFS"]
        | setSpeed(20) -> OUT.Cont:=20 -> [20,"VFS"]
        | setSpeed(40) -> OUT.Cont:=40 -> [40,"VFS"]
        | setSpeed(60) -> OUT.Cont:=60 -> [60,"VFS"]
        | setSpeed(80) -> OUT.Cont:=80 -> [80,"VFS"]
        | OUT.Cont:=80 -> [80,"VFS"])

[60,"VFS"] = (
          setSpeed(0) -> OUT.Cont:=0 -> [0,"VFS"]
        | setSpeed(20) -> OUT.Cont:=20 -> [20,"VFS"]
        | setSpeed(40) -> OUT.Cont:=40 -> [40,"VFS"]
        | setSpeed(60) -> OUT.Cont:=60 -> [60,"VFS"]
        | setSpeed(80) -> OUT.Cont:=80 -> [80,"VFS"]
        | OUT.Cont:=60 -> [60,"VFS"])

[20,"VFS"] = (
          setSpeed(0) -> OUT.Cont:=0 -> [0,"VFS"]
        | setSpeed(20) -> OUT.Cont:=20 -> [20,"VFS"]
        | setSpeed(40) -> OUT.Cont:=40 -> [40,"VFS"]
        | setSpeed(60) -> OUT.Cont:=60 -> [60,"VFS"]
        | setSpeed(80) -> OUT.Cont:=80 -> [80,"VFS"]
        | OUT.Cont:=20 -> [20,"VFS"])

[40,"VFS"] = (
          setSpeed(0) -> OUT.Cont:=0 -> [0,"VFS"]
        | setSpeed(20) -> OUT.Cont:=20 -> [20,"VFS"]
        | setSpeed(40) -> OUT.Cont:=40 -> [40,"VFS"]
        | setSpeed(60) -> OUT.Cont:=60 -> [60,"VFS"]
        | setSpeed(80) -> OUT.Cont:=80 -> [80,"VFS"]
        | OUT.Cont:=40 -> [40,"VFS"])

[0,"VFS"] = (
          setSpeed(0) -> OUT.Cont:=0 -> [0,"VFS"]
        | setSpeed(20) -> OUT.Cont:=20 -> [20,"VFS"]
        | setSpeed(40) -> OUT.Cont:=40 -> [40,"VFS"]
        | setSpeed(60) -> OUT.Cont:=60 -> [60,"VFS"]
        | setSpeed(80) -> OUT.Cont:=80 -> [80,"VFS"]
        | OUT.Cont:=0 -> [0,"VFS"])
```

**Abb. 4.1**: C#-Klasse und Prozessmodell der Komponente Speed

```
1  using System;
2  using CCM;
3
4  namespace CCM.Seat
5  {
6          /// Abstraction of speed: only two states "stop" (go==false) and
7          /// "driving" (go==true)
8          public class SpeedAbstr : Comp
9          {
10                 // Stubs
11                 [Property("Cont"), Eq ( 0.0, 0.0 ),  Eq( 10.0,  "(0.0,oo)" )]
12                 public VFS OUT = new VFS(0);
13
14                 [Abstracts("speed"), Eq( false , 0.0 ),  Eq( true, "(0.0,oo)")]
15                 bool go;
16
17                 protected override void Run()
18                 {
19                         if ( go )
20                             OUT.Cont = 10.0;
21                         else
22                             OUT.Cont = 0.0;
23                 }
24
25                 [Event] public void setSpeed( [Range(0.0,80.0,20.0)] float speed)
26                 {
27                         this.go = (speed != 0.0);
28                         EventOccured();
29                 }
30         }
31 }
```

```
State name format: ["go", "OUT"]

SpeedAbstr = OUT.Cont:=0 -> [False,"VFS"]

[True,"VFS"] = (
          setSpeed(0) -> OUT.Cont:=0 -> [False,"VFS"]
        | setSpeed(20) -> OUT.Cont:=10 -> [True,"VFS"]
        | setSpeed(40) -> OUT.Cont:=10 -> [True,"VFS"]
        | setSpeed(60) -> OUT.Cont:=10 -> [True,"VFS"]
        | setSpeed(80) -> OUT.Cont:=10 -> [True,"VFS"]
        | OUT.Cont:=10 -> [True,"VFS"])

[False,"VFS"] = (
          setSpeed(0) -> OUT.Cont:=0 -> [False,"VFS"]
        | setSpeed(20) -> OUT.Cont:=10 -> [True,"VFS"]
        | setSpeed(40) -> OUT.Cont:=10 -> [True,"VFS"]
        | setSpeed(60) -> OUT.Cont:=10 -> [True,"VFS"]
        | setSpeed(80) -> OUT.Cont:=10 -> [True,"VFS"]
        | OUT.Cont:=0 -> [False,"VFS"])
```

**Abb. 4.2**: C#-Klasse und Prozessmodell der abstrahierenden Komponente `SpeedAbstr`

In dem dort präsentierten Algorithmus ist das Konzept des Zustandsfilters allerdings allgemeiner gefasst: es gibt zwei Relationen $\Phi_1, \Phi_2 \subseteq S_1 \times S_2 \times \Delta$ für relevante Zustände und Transitionen der Abstraktion und Implementierung. Diese ergeben sich aus der hier vorgestellten Variante mit nur einem $\Phi \subseteq S_2$ durch $\Phi_1 = S_1 \times \Phi \times \Delta$ und $\Phi_2 = \Phi_1$ für Bisimulation bzw. $\Phi_2 = \varnothing$ für einfache Simulation (Implementierung). Die Auswahl der zu prüfenden Beziehung ($\lesssim$ oder $\sim$) erfolgt hier durch den globalen Parameter $bisim \in \{yes, no\}$.

Zum anderen unterstützt obiger Algorithmus Nichtdeterminismus, d. h. eine Transition kann in unterschiedliche Folgezustände führen. Die Konstruktion von `cs2ccmb` schließt nichtdeterministische Alternativen aber prinzipiell aus, da die Auswahl eines Folgezustandes zum spätestmöglichen Zeitpunkt vorgenommen wird: `cs2ccmb` erzeugt Prozesse der deterministischen Form $a \rightarrow (b|c)$, und nicht die (nicht äquivalente, aber ähnliche) indeterministische Form $(a \rightarrow b)|(a \rightarrow c)$. Dies vereinfacht den Algorithmus erheblich, da z. B. keine alternativen Folgezustände von schon ausgewerteten Transitionen berücksichtigt werden müssen.

Zu beachten ist bei der Implementierung die Anwendung der Transitions-Abstraktion $TA$ anstatt des direkten Vergleiches der Transitionen der beiden Komponenten.

Der Algorithmus arbeitet mit folgenden Datenstrukturen:

- Bisimulationsgraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ mit $\mathcal{V} \subseteq S_1 \times S_2$ und $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$:

  In den Knoten (Menge $\mathcal{V}$) werden die simularen bzw. äquivalenten Zustandspaare gespeichert. Am Ende des Algorithmus ist $\mathcal{V}$ eine (Bi)simulation.

  Die Knoten $(p', q')$, von denen Kanten aus $\mathcal{E}$ zu einem Zustandspaar $(p, q) \in \mathcal{V}$ führen, sind die „Rechtfertigung“ für die (Bi)simularität von $(p, q)$, denn wenn alle Nachfolgezustände $(p', q')$ (bi)simular sind, sind es auch $(p, q)$. (Natürlich müssen alle Knoten auch die Transitions- bzw. Zustandsabstraktion erfüllen.)

- $\overline{R} \subseteq S_1 \times S_2$: diese Menge enthält alle schon untersuchten nicht-(bi)simularen Zustandspaare.

- $A \subseteq S_1 \times S_2$: wird bei der Untersuchung festgestellt, dass ein Zustandspaar $(p, q)$ nicht (bi)simular ist, werden in dieser Menge alle Knoten aus $\mathcal{V}$ gespeichert, die aus dem Bisimulationsgraph wieder entfernt werden müssen, also alle Knoten, zu denen transitiv Kanten aus $\mathcal{E}$ von $(p, q)$ führen.

Der Algorithmus besteht aus drei Funktionen:

- $similar : (p, q) \rightarrow \{related, not\_related\}$ prüft, ob $p$ und $q$ in der durch $bisim$ festgelegten Beziehung stehen. Üblicherweise wird diese Funktion mit den Startzuständen der zu vergleichenden Komponenten aufgerufen.

- $search\_impl : (p \stackrel{\delta}{\rightarrow} p', q) \rightarrow \{related, not\_related\}$ versucht, zum gegebenen Zustandsübergang der Abstraktion einen entsprechenden Nachfolgerzustand $q'$ der Implementierung zu finden, so dass $p'$ und $q'$ in Beziehung stehen. Wird ein solcher Zustand gefunden, wird die Kante $((p', q'), (p, q))$ zu $\mathcal{E}$ hinzugefügt.

- $search\_abstr : (q \stackrel{\delta}{\rightarrow} q', p) \rightarrow \{related, not\_related\}$ ist analog zu $search\_impl$ zur Suche eines passenden Nachfolgerzustandes $p'$ der Abstraktion zu einem gegebenen Zustandsübergang der Implementierung.

Abb. 4.3 zeigt die Definition der Funktion *similar*, Abb. 4.4 die *search_...*-Funktionen. Die Algorithmen verwenden folgende Notationen:

$\{p \xrightarrow{\bullet}\} \quad =_{def} \quad \{\delta |\ (p, \delta, p') \in next\}$ (alle möglichen Transitionen von $p$)

$(p \xrightarrow{\delta}) \quad =_{def} \quad p'$ mit $(p, \delta, p') \in next$ (Folgezustand von $p$ über $\delta$)

Da wie schon erwähnt die zu untersuchenden Modelle stets deterministisch sind, ist $(p \xrightarrow{\delta})$ stets eindeutig bestimmt.

## 4.6. Implementierung

Das Programm „`sim`“ implementiert den vorgestellten Algorithmus. Es erhält als Parameter Optionen (s. u.) und die beiden Prozessmodelle der abstrakten und der implementierenden Komponente im Binärformat. Diese enthalten die serialisierten Objektstrukturen der durch `cs2ccmb` erzeugten Prozessmodelle.

Optional können danach noch beliebig viele Zustandsprädikate der Form „$Wert\ Op\ Wert$“ angegeben werden, die Bedingungen angeben, unter denen ein Zustand der Implementierung die Bisimulations-Bedingungen erfüllen muss. $Wert$ ist dabei eine Konstante (`true`, `false` oder eine Ganz- oder Fließkomma-Zahl) oder der Name eines Klassenfeldes der Implementations-Komponente. $Op$ gibt eine der möglichen Vergleichsoperation an:

```
<   <=   =   >=   >
```

Standardmäßig prüft `sim` die Implementierungs-Relation $\lesssim$ (Simulation). Soll vollständige Äquivalenz (Bisimulation $\sim$) geprüft werden, muss die Option `-b` angegeben werden.

Außerdem steht eine Option `-d` (Debugging) zur Verfügung, die den Verlauf der Prüfung auf der Konsole hierarchisch darstellt. Dies ermöglicht das schrittweise Nachvollziehen jedes Tests. Wenn diese Option nicht angegeben wird, werden lediglich nicht(bi)simulare Knotenpaare ausgegeben, um die Inkonsistenz leichter lokalisieren zu können.

`sim` besteht aus folgenden Modulen:

**Hauptprogramm** (`sim.cs`): Implementiert den Bisimulations-Algorithmus; verwendet die Bibliothek `csmodel.dll`, die die Klassen der deserialisierten Objekte bereitstellt.

**Zustandsfilter** (`statepredicate.cs`): Zustandsprädikat `StatePredicate` für die Selektion relevanter Zustände der implementierenden Komponente.

**Mengen** (`pairset.cs`): `PairSet` speichert eine Menge von Objektpaaren; realisiert durch kaskadierte Hash-Tabellen, so dass die Operationen $(x, y) \in M$, $M := M \cup \{(x, y)\}$ und $M := M \setminus \{(x, y)\}$ sehr effizient sind (maximal $\mathcal{O}(\log |M|)$, bei optimaler Balancierung sogar nur $\mathcal{O}(\log \log |M|)$.

**Debugging** (`debugging.cs`): Stellt die Klasse `Logger` bereit, die eine hierarchische Ausgabe von Debug-Meldungen implementiert.

Die einzelnen Klassen, Methoden und Felder sind wiederum in javadoc-Syntax dokumentiert, doxygen erzeugte wiederum die im Projektarchiv zu findende HTML-Referenz.

$\overline{R} := \varnothing$
$\mathcal{V} := \varnothing$
$\mathcal{E} := \varnothing$
**function** SIMILAR$(p, q)$
  **if** $(p, q) \in \overline{R}$ **then** ▷ wurde schon als nicht simular erkannt
    **return** *not_related*
  **end if**
  **if** $(p, q) \notin ZA$ **then** ▷ prüfen, ob Zustandsabstraktion erfüllt
    $\overline{R} := \overline{R} \cup \{(p, q)\}$
    **return** *not_related*
  **end if**
  **if** $(p, q) \in \mathcal{V} \ \vee \ q \notin \Phi$ **then** ▷ schon geprüft oder nicht relevant
    **return** *related*
  **end if**
  ▷ $(p, q)$ wird zum ersten Mal geprüft
  $\mathcal{V} := \mathcal{V} \cup \{(p, q)\}$
  $status := related$
  **for all** $\delta \in \{p \xrightarrow{\bullet}\}$ **do** ▷ für jeden $p$-Nachfolger passenden $q$-Nachfolger suchen
    **if** $status = related$ **then**
      $p' := (p \xrightarrow{\delta})$
      $status :=$SEARCH_IMPL$(p \xrightarrow{\delta} p', q)$
    **end if**
  **end for**
  **if** $bisim = yes$ **then** ▷ diese Richtung nicht bei Relation $\lesssim$ prüfen
    **for all** $\delta \in \{q \xrightarrow{\bullet}\}$ **do** ▷ für jeden $q$-Nachfolger passenden $p$-Nachfolger suchen
      **if** $status = related$ **then**
        $q' := (q \xrightarrow{\delta})$ ▷ Nachfolgerzustand von $q$ über $\delta$
        $status :=$SEARCH_ABSTR$(q \xrightarrow{\delta} q', p)$
      **end if**
    **end for**
  **end if**

  **if** $status = not\_related$ **then** ▷ alle Knoten löschen, zu denen transitiv Kanten führen
    $A := \{(p, q)\}$
    **while** $A \neq \varnothing$ **do**
      **Choose** $(r, s) \in A$ ▷ beliebiges Element aus A auswählen
      $A := A \setminus \{(r, s)\}$
      $A := A \cup \{(rr, ss) |\ ((r, s), (rr, ss)) \in \mathcal{E}\}$ ▷ alle Vorgängerzustandspaare suchen
      $\mathcal{V} := \mathcal{V} \setminus \{(r, s)\}$ ▷ $(r, s)$ aus Bisimulationsgraph entfernen
      $\mathcal{E} := \mathcal{E} \setminus \{(e_1, e_2) |\ e_1 = (r, s) \ \vee \ e_2 = (r, s)\}$
      $\overline{R} := \overline{R} \cup \{(s, s)\}$ ▷ als nicht (bi)simular markieren
    **end while**
  **end if**
  **return** *status*
**end function**

**Abb. 4.3**: Definition der Funktion *similar*

```
function SEARCH_ABSTR(q -δ->q', p)
    for all ε ∈ {p -•->} do
        p' := (p -ε->)
        if (ε, δ) ∈ TA ∧ SIMILAR(p', q') then          ▷ simular, Kante hinzufügen
            E := E ∪ {((p', q'), (p, q))}
            return related
        end if
    end for
    return not_related
end function

function SEARCH_IMPL(p -δ->p', q)
    for all ε ∈ {q -•->} do
        q' := (q -ε->)
        if (δ, ε) ∈ TA ∧ SIMILAR(p', q') then          ▷ simular, Kante hinzufügen
            E := E ∪ {((p', q'), (p, q))}
            return related
        end if
    end for
    return not_related
end function
```

**Abb. 4.4**: Definition der Funktionen *search_abstr* und *search_impl*

## 4.7. Beispiel

Der Aufruf und die Ausgabe von `sim` wird wiederum anhand der schon vorgestellten Beispiel-Komponenten `Speed` (Abb. 4.1) und `SpeedAbstr` (Abb. 4.2) erfolgen. Zunächst werden die Prozessmodelle beider Komponenten im Binär-Format erzeugt:

```
cs2ccmb.exe -bin Speed.bin VFS.cs Speed.cs

cs2ccmb.exe -bin SpeedAbstr.bin VFS.cs SpeedAbstr.cs
```

Erwartungsgemäß ist `SpeedAbstr` auch eine konsistente Abstraktion von `Speed`: Der Aufruf

```
sim.exe -b SpeedAbst.bin Speed.bin
```

erzeugt die Meldung „`Models are bisimilar`“.

Bei so einfachen Komponenten gibt es natürlich kaum nichttriviale Fehler, aber zur Demonstration soll in der Abstraktion einmal in `setSpeed()` die Zeile

```
this.go = (speed != 0.0);
```

durch die (offensichtlich falsche) Zeile

```
this.go = (speed > 20.0);
```

ersetzt werden, d. h. `go` unterscheidet nicht zwischen „Halt“ und „Fahrt“, sondern eher zwischen „langsam“ (weniger oder gleich 20 km/h) und „schnell“ (mehr als 20 km/h). Nun erzeugt obiger `sim`-Aufruf:

```
--------------------------------------------------
Not bisimilar:
Abstract      : setSpeed(20)
Implementation: setSpeed(20)
--------------------------------------------------
Not bisimilar:
Abstract      : [False,"VFS"]
Implementation: [0,"VFS"]
--------------------------------------------------
Not bisimilar:
Abstract      : OUT.Cont:=0
Implementation: OUT.Cont:=0
--------------------------------------------------
Not bisimilar:
Abstract      : SpeedAbstr
Implementation: Speed

Models are not bisimilar
```

Im Knoten `setSpeed(20)` sind die beiden Prozessmodelle inkonsistent (nicht bisimular), deshalb müssen alle vorhergehenden Knotenpaare rückwirkend ebenfalls als nicht bisimular markiert werden; diese werden dann ebenfalls ausgegeben. Liest man die Liste von unten nach oben, erhält man die Schritte, die von den jeweiligen Startzuständen aus zu dem Knoten führen, die nicht mehr verhaltensgleich sind. Dieser Weg lässt sich anhand der vorliegenden CCMB-Beschreibung gut nachvollziehen.

Zur Demonstration der Prüfung der Relation „implementiert“ (einfache Simularität) kann in `SpeedAbstr.cs` die Zeile

```
[Event] public void setSpeed( [Range(0.0,80.0,20.0)] float speed)
```

durch

```
[Event] public void setSpeed( [Range(0.0,60.0,20.0)] float speed)
```

ersetzt werden; dadurch existiert die Transition `setSpeed(80)` nicht mehr, alle anderen Transitionen werden jedoch nicht verändert:

```
$ cs2ccmb.exe -ccmb - VFS.cs SpeedAbstr60.cs
State name format: ["go", "OUT"]

SpeedAbstr = OUT.Cont:=0 -> [False,"VFS"]

[True,"VFS"] = (
          setSpeed(0) -> OUT.Cont:=0 -> [False,"VFS"]
        | setSpeed(20) -> OUT.Cont:=10 -> [True,"VFS"]
        | setSpeed(40) -> OUT.Cont:=10 -> [True,"VFS"]
        | setSpeed(60) -> OUT.Cont:=10 -> [True,"VFS"]
        | OUT.Cont:=10 -> [True,"VFS"])

[False,"VFS"] = (
          setSpeed(0) -> OUT.Cont:=0 -> [False,"VFS"]
        | setSpeed(20) -> OUT.Cont:=10 -> [True,"VFS"]
        | setSpeed(40) -> OUT.Cont:=10 -> [True,"VFS"]
        | setSpeed(60) -> OUT.Cont:=10 -> [True,"VFS"]
        | OUT.Cont:=0 -> [False,"VFS"])
```

Erwartungsgemäß schlägt

```
sim.exe -b SpeedAbstr60.bin Speed.bin
```

fehl, aber

```
sim.exe SpeedAbstr60.bin Speed.bin
```

gibt die Erfolgsmeldung „`First model is implemented by the second one.`“ aus.

# 5. Anwendungsfälle

In diesem Kapitel werden die verschiedenen Einsatzmöglichkeiten der (Bi)simulationsprüfung zusammengefasst und jede anhand einer Komponente der PKW-Sitzsteuerung demonstriert. Es werden auch die Modifikationen erläutert, die an den Original-Quellcodes erforderlich sind, um die Analyse durchzuführen.

## 5.1. Reduktion des Zustandsraums

Die Hauptmotivation dieser Arbeit war die Reduktion des Zustandsraumes einer Komponente, um eine hierarchische Verifikation zu ermöglichen. Dies wurde schon anhand der Komponente `Speed` in Abschnitt 4.7 durchgeführt.

Zur Erstellung einer abstrakten Komponente und deren Analyse sind folgende Schritte notwendig:

1. Attributierung der Implementierungs-Komponente: Im Fall `Speed.cs` musste die Methode `setSpeed()` durch `[Event]` gekennzeichnet und ihr Parameter `speed` durch ein `Range`-Attribut in Wertebereich und Auflösung beschränkt werden.
2. Entscheidung über die zu abstrahierenden Zustandsgrößen und Zusammenfassung von Intervallen zu Äquivalenzklassen: Im vorliegenden Beispiel wurde zwischen „Stop“ und „Fahrt“ unterschieden, es sind jedoch auch andere Varianten denkbar (z. B. „Stop“, „langsam“ und „schnell“).
3. Implementierung der abstrakten Komponente und Angabe der Äquivalenzklassen durch `Eq`-Attribute an den abstrahierenden Feldern.
4. Falls notwendig, müssen auch die Transitionen abstrahiert werden. In der abstrakten Komponente sind die Äquivalenzklassen wiederum durch `Property`- und `Eq`-Attribute der Stub-Instanzen abzugeben.
5. Generierung der Modelle (im Binärformat) durch Aufruf von `cs2ccmb` mit entsprechenden Konstruktorparametern.
6. Die Abstraktion ist korrekt implementiert, wenn die Modelle bisimular sind, d. h. wenn

   `sim.exe -b` *(Abstraktes Modell) (Konkretes Modell)*

   erfolgreich ist.

## 5.2. Qualifizierte Auswahl und Parameter-Konfiguration

Die Sitzsteuerung enthält eine Schalterkomponente `Button`, die recht vielseitig und komplex ist (Unterstützung von mehreren Tastern, Zähler für Flanken, etc.). Der hier in Anhang A.1 gezeigte und auch im Projektarchiv enthaltene Quellcode wurde schon für die Analyse vorbereitet:

- Die Methode `setPressed` wurde mit `[Event]` und dem zulässigen Bereich für `button` attributiert (Zeile 42).
- Der Type des Klassenfelds `lockvar` wurde von `Object` auf `string` geändert, da die momentane Implementierung von `cs2ccmb` den Typ `Object` nicht kennt (Zeile 38).
- Die Zählung der Flanken wurde auf 10 beschränkt, um das generierte Modell klein zu halten; zum anderen unterscheidet `cs2ccmb` momentan nicht zwischen verschiedenen Integer-Typen, sondern rechnet intern immer mit `int` (Zeilen 51 und 52).

Nun soll diese Komponente durch eine einfachere, speziellere Komponente ersetzt werden. Ein erster Entwurf einer einfachen Komponente mit nur einem Knopf und ohne Flankenzählung könnte dem schon gezeigten `Switch` (Abb. 2.1) ähneln:

```
1  using System;
2  using CCM;
3
4  namespace CCM.Seat
5  {
6      public class SimpleButton1 : CCM.Comp
7      {
8          // Coops
9          public VFS OUT = new VFS(2);
10         bool[] pressed = new bool[1];
11
12         [Event] public void setPressed( [Range(1,1)] byte button,bool val)
13         {
14             pressed[0] = val;
15             EventOccured();
16         }
17
18         protected override void Run()
19         {
20             if (pressed[0])
21             {
22                 OUT.acquireCooperation();
23                 OUT.Cont = -1.0F;
24                 OUT.State = 1;
25             } else {
26                 OUT.Cont = -1.0F;
27                 OUT.State = 0;
28                 OUT.releaseCooperation();
29             }
30         }
31
32     }
33 }
```

Der aktuelle Zustand wird – etwas umständlich – in einem einelementigen Array gespeichert, da `sim` (momentan) einfache Werte nicht mit Arrays vergleichen kann. Jedoch schlägt dieser erste Versuch fehl:

```
$ cs2ccmb.exe -bin Button1ff.bin VFS.cs Button.cs -- 1 false false
$ cs2ccmb.exe -bin SimpleButton1.bin VFS.cs SimpleButton1.cs
$ sim.exe -b SimpleButton1.bin Button1ff.bin
Loaded abstract model.
Loaded implementation model.
--------------------------------------------------
Not bisimilar:
Abstract      : SimpleButton1
Implementation: Button
```

Ein Blick auf die CCMB-Modelle (die hier nicht mit abgedruckt sind) verrät auch den Grund: `Button` sendet nur einen neuen Zustand an den Stub `OUT`, wenn sich der Tasterzustand auch wirklich geändert hat und nicht wie `SimpleButton1` bei jedem Aufruf der `Run`-Methode.

In einem zweiten Versuch wird der vorherige Zustand des Tasters gespeichert und der Stub nur bei einer tatsächlichen Zustandsänderung kontaktiert. Nun sind die Modelle äquivalent:

```
1  using System;
2  using CCM;
3
4  namespace CCM.Seat
5  {
6      public class SimpleButton2 : CCM.Comp
7      {
8          // Coops
9          public VFS OUT = new VFS(2);
10         bool[] pressed = new bool[1];
11
12         bool on = false;
13
14         [Event] public void setPressed( [Range(1,1)] byte button,bool val)
15         {
16             pressed[0] = val;
17             EventOccured();
18         }
19
20         protected override void Run()
21         {
22             if( on == pressed[0] )
23                 return;
24
25             if (pressed[0])
26             {
27                 OUT.acquireCooperation();
28                 OUT.Cont = -1.0F;
29                 OUT.State = 1;
30             } else {
31                 OUT.Cont = -1.0F;
32                 OUT.State = 0;
33                 OUT.releaseCooperation();
34             }
35             on = pressed[0];
36         }
37
38     }
39 }
```

```
$ cs2ccmb.exe -bin SimpleButton2.bin VFS.cs SimpleButton2.cs
$ sim.exe -b SimpleButton2.bin Button1ff.bin
Loaded abstract model.
Loaded implementation model.
Models are bisimilar.
```

Umgekehrt ist es natürlich auch möglich, zu einer gegebenen Komponente („Spezifikations-Komponente“) eine passende allgemeine Implementierung zu suchen und zu verifizieren, dass deren Parameter korrekt eingestellt werden. Für die Parameter `(1, false, false)` wurde dies soeben gezeigt; ein `Button` mit z. B. aktivierter Flankenzählung (Konstruktor-Parameter `(1, true, false)`) ist jedoch keine geeignete Implementierung für `SimpleButton2`:

```
$ cs2ccmb.exe -bin Button1tf.bin VFS.cs Button.cs -- 1 true false
$ ./sim.exe -b SimpleButton2.bin Button1tf.bin
Loaded abstract model.
Loaded implementation model.
--------------------------------------------------
Not bisimilar:
Abstract      : setPressed(1, True)
Implementation: setPressed(1, True)
--------------------------------------------------
Not bisimilar:
Abstract      : [[False],False,"VFS"]
Implementation: [False,[False],True,"",0,"VFS",False,0,False,1,True,False,0,False]
--------------------------------------------------
Not bisimilar:
Abstract      : SimpleButton2
Implementation: Button

Models are not bisimilar.
```

## 5.3. Verifikation Teilfunktionalität

Durch die Angabe von Filterprädikaten ist es auch möglich, Teilfunktionalitäten einer Komponente zu prüfen. Um gleichzeitig auch die Grenzen von `cs2ccmb` auszutesten, soll dies an der Komponente `Courtesy` demonstriert werden. Sie steuert das automatische Zurückfahren des Fahrersitzes um ein einfacheres Ein- und Aussteigen zu ermöglichen. Der Quellcode ist in Anhang A.2 zu finden; gegenüber der Originalimplementation wurde lediglich die Property `float CourtesyPoint` entfernt, da diese für die Analyse nicht benötigt wird und `cs2ccmb` noch keine Propertys unterstützt.

Das Modell dieser Komponente umfasst etwa 3 Millionen Zustände und Transitionen, die generierte Binärdatei ist ca. 160 MB groß. Daher ist die CCMB-Beschreibung hier nicht mit abgedruckt, da sie etwa 12.000 Seiten umfassen würde und Modelle dieser Größenordnung ohnehin nicht mehr zur manuellen Auswertung geeignet sind.

Gemäß der Spezifikation würde der Sitz beim wiederholten Einstecken und Herausziehen des Zündschlüssels ständig vor- und zurückfahren. Daher sieht die Spezifikation eine Beschränkung dieses Zykluses auf maximal 3 Wiederholungen vor. Eine vereinfachte Komponente, die diese Sicherheitsabschaltung nicht enthält (d. h. der Zähler `count` wurde entfernt) ist `CourtesyNoCount` (s. Anhang A.3).

Nun ist offensichtlich, dass sich `CourtesyNoCount` nicht wie `Courtesy` verhält:

```
$ cs2ccmb.exe -bin Courtesy.bin VFR.cs VFS.cs Courtesy.cs
[Transitionszähler]
Finished. Calculated 2975525 nodes.
```

```
$ cs2ccmb.exe -bin CourtesyNoCount.bin VFR.cs VFS.cs CourtesyNoCount.cs
[Transitionszähler]
Finished. Calculated 803177 nodes.

$ sim.exe -b CourtesyNoCount.bin Courtesy.bin
Loaded abstract model.
Loaded implementation model.
Not bisimilar:
Abstract      : KL.State=0
Implementation: KL.State=0
--------------------------------------------------
Not bisimilar:
Abstract      : MotorLA.StateFb=0
Implementation: MotorLA.StateFb=0
--------------------------------------------------
Not bisimilar:
Abstract      : Door.stateChanges(1, 0)=False
Implementation: Door.stateChanges(1, 0)=False
--------------------------------------------------
[...]
--------------------------------------------------
Not bisimilar:
Abstract      : Courtesy[null,0,0,False,False,False,null,null,False]
Implementation: Courtesy[null,0,0,0,False,False,False,null,null,False]

Models are not bisimilar.
```

Die vereinfachte Variante verhält sich jedoch wie `Courtesy` in allen Zuständen, in denen `count` kleiner als 3 ist:

```
$ sim.exe -b CourtesyNoCount.bin Courtesy.bin "count < 3"
Loaded abstract model.
Loaded implementation model.
Models are bisimilar.
```

Die Komponente `CourtesyNoCount` reduziert somit den Zustandsraum von 3 Millionen auf etwa 800.000 Zustände und kann somit für darauf aufbauende Komponenten, die nicht speziell mit dieser Sicherheits-Abschaltung arbeiten, äquivalent verwendet werden. Weitere Vereinfachungen können den Zustandsraum sicher noch weiter verkleinern.

Einige technische Anmerkungen:

Analysen dieser Größenordnung sollten auf Rechnern mit viel Hauptspeicher durchgeführt werden (mindestens 1 GB), da ansonsten durch Auslagerung von Speicher auf die Festplatte („swapping“) die Laufzeit leicht mehrere Stunden betragen kann.

Die CPU-Leistung ist relativ uninteressant, der Hauptspeicher ist die kritische Größe: so dauerte die erste Bisimulationsprüfung (ohne Zustandsfilter) nur wenige Sekunden, da nur etwa 200.000 Transitionen berechnet wurden. Die ersten 1,7 Millionen Zustände der zweiten Analyse wurden in wenigen Minuten geprüft, aber die Prüfung der restlichen 1,3 Millionen Zustände benötigte mehrere Stunden, da bei dieser Grenze der Hauptspeicher voll war.

Beide Programme zeigen daher den Fortschritt der Berechnung an, indem sie die Anzahl schon berechneter Transitionen ausgeben.

# 6. Zusammenfassung und Ausblick

Diese Arbeit hat gezeigt, dass eine automatische Konsistenzprüfung abstrakter und konkreter Feldbuskomponenten-Implementierungen möglich ist. Die bereitgestellten Werkzeuge `cs2ccmb` und `sim` erlauben die praktische Durchführung dieser Analysen bis zu einer Größenordnung, die in Abschnitt 5.3 erkennbar geworden ist.

`cs2ccmb` erweitert auch das Ergebnis von [Pit03] um die Möglichkeit, die Verifikation von Eigenschaften mit dem Prolog-Theorembeweiser auch über schon existierende C#-Implementierungen durchzuführen, ohne eine CCMB-Spezifikation manuell anfertigen zu müssen.

Jedoch sollte diese Arbeit in diesem interessanten Forschungsgebiet keinesfalls als abgeschlossen angesehen werden. Mögliche Fortführungen und Erweiterungen sind:

- Die Vervollständigung des C#-Parsers und -Interpreters um die nicht unterstützten Sprachelemente (siehe Abschnitt 3.1.1).
- Möglichkeiten der Verkleinerung des Prozessmodells: Bei großen Modellen ist ersichtlich, dass die generierten Modelle eine sehr regelmäßige Struktur und viel Redundanz haben. Es wäre wünschenswert, die Prozessmodelle in einer effizienteren Datenstruktur zu speichern. Eine Idee wäre die Verwendung von Entscheidungsbäumen (OBDDs – „ordered binary decision diagrams“).
- Konsistenzprüfung für Abstraktionen ganzer Teilsysteme (siehe Abschnitt 4.2).

# 7. Das Projekt-Archiv

Alle relevanten Dateien wurden in ein Archiv gepackt, das von

`http://www.piware.de/projects/bisim-sharp.zip`

heruntergeladen werden kann (ca. 500 KB).

## 7.1. Inhalt

**bin/:** die fertig kompilierten Programme `cs2ccmb`, `sim` und `bindump` und die Bibliothek `csmodel.dll`.

`bindump` ist ein Hilfsprogramm, dass ein Modell im Binärformat einliest und die Prozesse in CCMB-Syntax auf die Standard-Ausgabe schreibt.

**html/:** die durch **doxygen** erzeugte Referenz aller Quelltext-Dateien, Namensräume, Klassen, Methoden und Felder im HTML-Format. Wie üblich ist `index.html` die Startseite.

**components/:** enthält die in dieser Arbeit erwähnten CCM-Komponenten.

**src/** C#-Quellen aller Programme; dieses Verzeichnis enthält auch die **doxygen**-Steuerdatei `Doxyfile` und eine **make**-Steuerdatei `Makefile`.

Hier befindet sich auch die Datei `CCMAttributes.cs`, die die verwendeten Attribute (`Event`, `Range`, `Abstracts`, `Property` und `Eq`) definiert und zur Kompilierung attributierter Komponenten erforderlich ist.

**LICENSE:** Der vollständige Text der GNU General Public License, unter der das Projekt veröffentlicht wird.

## 7.2. Hinweise zur Kompilierung

Die Programme wurden ausgiebig sowohl mit Microsofts .NET SDK 1.1 unter Windows XP als auch mit Novells Mono Beta 1 unter Linux getestet. Fast alle Regeln des `Makefile` funktionieren mit `nmake` aus dem .NET-SDK und mit GNU `make`, allerdings müssen wegen unterschiedlicher Befehls- und Pfadnamens-Konventionen einige Variablen angepasst werden.

In der ausgelieferten Fassung sind die für Unix passenden Werte aktiviert und die für Windows zu verwendenden Werte auskommentiert; für die Kompilation unter Windows ist das `Makefile` also entsprechend zu modifizieren.

Das `Makefile` bietet folgende Ziele:

**release (Default-Ziel):** kompiliert die Programme `cs2ccmb`, `sim` und `bindump`.

**debug:** kompiliert ebenfalls alle Programme, jedoch wird `cs2ccmb` in einer Variante erstellt, die sehr viele Informationen über den Verlauf der Analyse zu Debugging-Zwecken ausgibt.

Dies wird durch das Setzen des Symbols `DEBUG` erreicht. Im Quellcode werden die Debug-Meldungen durch ein `#if DEBUG ... #endif` nur bedingt übersetzt.

**doc:** erzeugt die HTML-Referenz durch Aufrufen von `doxygen Doxyfile`.

Dies ist die einzige Regel, die unter Windows nicht funktioniert. Die Referenz umfasst auch die Dokumentation des von Jay generierten C#-Parsers (`cs-parser.cs`). **doxygen** erfordert, dass die Dokumentation der Datei ganz am Anfang steht, Jay setzt jedoch an den Anfang des generierten Quellcodes einen eigenen Kommentar. Dieser wird von der `Makefile`-Regel durch `tail +4` weggefiltert. Der `tail`-Befehl steht jedoch unter Windows nicht zur Verfügung, weshalb das Entfernen des Jay-Kommentars und der Aufruf zur Generierung der Dokumentation dort manuell durchgeführt werden muss.

**clean:** löscht alle kompilierten Programme.

**cleandoc:** löscht die generierte HTML-Referenz.

**distclean:** ruft `clean` und `cleandoc` auf und löscht zusätzlich `cs-parser.cs`.

**testswitch:** ruft `cs2ccmb` zur Generierung des CCMB-Modells für `Switch.cs` auf; erzeugt eine Ausgabe ähnlich Abb. 2.2.

**testspeed:** generiert das CCMB-Modell für `Speed.cs` (s. Abb. 4.1).

**testprolog:** generiert das Prolog-Modell für `Speed.cs` (s. Abschnitt 2.7.3).

**testbisim:** generiert Modelle für `Speed.cs` und `SpeedAbstr.cs` (in der „falschen“ Variante mit bis zu 60 km/h) und prüft Simulation und Bisimulation (s. Abschnitt 4.7).

Alle Regeln prüfen und generieren ihre Abhängigkeiten, d. h. jede Regel kann zu jeder Zeit erfolgreich aufgerufen werden. Wird z. B. in einem „sauberen“ Quellbaum `make testspeed` aufgerufen, wird zunächst `cs2ccmb` kompiliert (jedoch nicht `sim`, weil dies für diesen Test nicht benötigt wird).

# 8. Selbstständigkeits-Erklärung, Copyright und Lizenz

Hiermit versichere ich, dass ich diese Diplomarbeit, die die vorliegende schriftliche Ausarbeitung, den Quellcode und die Dokumentation der beschriebenen Programme umfasst, selbstständig verfasst und keine anderen als die im Literaturverzeichnis und im Abschnitt 3.3 angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche durch Verweise ins Literaturverzeichnis oder Fußnoten kenntlich gemacht.

Ferner erkläre ich, dass diese Arbeit in keinem anderen Studiengang und keiner anderen Institution als Prüfungsleistung verwendet wurde und wird.

Martin Pitt
Dresden, 24. Juni 2004

# A. Listings

Dieser Anhang enthält Quellcodes und Modelle, die zu lang waren, um sie als Abbildungen in den laufenden Text einzufügen.

## A.1. Button.cs

```
1  using System;
2  using System.Collections;
3  using System.ComponentModel;
4  using System.Drawing;
5  using System.Data;
6  using System.Windows.Forms;
7  using System.Threading;
8
9  using CCM;
10
11 namespace CCM.Seat
12 {
13
14         public class Button : CCM.Comp
15         {
16                 // Coops
17                 public VFS OUT = new VFS(3);
18
19                 byte buttons = 1;
20                 bool flank = false;
21                 bool locked = false;
22                 bool on = false;
23                 bool stopOnFbFail = false;
24                 bool[] pressed;
25
26                 public Button(byte buttons,bool flank,bool stopOnFbFail)
27                 {
28                         if (buttons == 0) throw new Exception();
29                         if (flank && (buttons != 1)) throw new Exception();
30                         this.buttons = buttons;
31                         this.flank = flank;
32                         this.stopOnFbFail = stopOnFbFail;
33                         if (flank) OUT = new VFS(1);
34                         else OUT = new VFS((byte) (buttons+1));
35                         pressed = new bool[buttons];
36                 }
37
38                 string lockvar = "";
39                 byte flanks = 0;
40                 byte oldFlanks = 0;
41
42                 [Event] public void setPressed( [Range(1,"buttons")] byte button,bool val)
43                 {
44                         lock (lockvar)
45                         {
46                                 if ((button<1) || (button > buttons)) throw new Exception();
47                                 pressed[button-1] = val;
48
```

```
49                          // Sonderbehandlung Flankensteuerung
50                          if (val && (oldFlanks == flanks))
51                              if( ++flanks == 10 )
52                                  flanks = 0;
53                  }
54
55                  EventOccured();
56          }
57
58          bool moreThenOnePressed = false;
59          bool onePressed = false;
60          bool nothingPressed = false;
61          byte onePressedIndex; // nur ü gltig , wenn onePressed == true;
62
63          void evalPressed()
64          {
65                  nothingPressed = true;
66                  moreThenOnePressed = false;
67                  for (int i=0; i<buttons; i++)
68                  {
69                          if (!pressed[i]) continue;
70                          onePressedIndex = (byte) i;
71                          if (!nothingPressed) moreThenOnePressed = true;
72                          nothingPressed = false;
73                  };
74                  onePressed = !nothingPressed && !moreThenOnePressed;
75          }
76
77          protected override void Run()
78          {
79                  evalPressed();
80
81                  if (moreThenOnePressed) locked = true;
82
83                  if (on && stopOnFbFail)
84                          if (!OUT.checkStateFeedback(1500)) locked = true;
85
86                  if (nothingPressed) locked = false;
87
88                  if (on && locked )
89                  {
90                          Common.log(this,"Stop due extra condition");
91                          locked = true;
92                  }
93
94                  if (on && (locked || nothingPressed))
95                  {
96                          OUT.Cont = -1.0F; OUT.State = 0;
97                          on = false;
98                          OUT.releaseCooperation();
99                          Common.log(this,"off");
100                 }
101
102                 if (!on && !flank && onePressed && !locked)
103                 {
104                         on = true;
105                         OUT.acquireCooperation();
106                         OUT.Cont = -1.0F;
107                         OUT.State = (byte) (onePressedIndex + 1);
108                         Common.log(this,"on");
109                 }
110
111                 if (!on && flank && (oldFlanks != flanks) && !locked)
112                 {
113                         Common.log(this,"Update flank signal");
114                         OUT.Cont = -1.0F;
115                         OUT.State = flanks;
116                         oldFlanks = flanks;
```

```
117                         }
118                         active = on;
119                 }
120
121         }
122 }
```

## A.2. Courtesy.cs

```
1   using System;
2   using System.Collections;
3   using System.ComponentModel;
4   using System.Drawing;
5   using System.Data;
6   using System.Windows.Forms;
7   using System.Threading;
8
9   namespace CCM.Seat
10  {
11          public class Courtesy : Comp
12          {
13                  // Coops
14                  public VFS MotorLA = new VFS(3);
15                  public VFR KL = new VFR(5);
16                  public VFR Door = new VFR(2);
17
18                  bool on = false;
19                  bool activeRestore = false;
20                  float orgPos = 0.0F;
21                  bool orgPosValid = false;
22
23                  bool meanwhile = false;
24
25                  float courtesyPoint = 60.0F;
26                  byte count = 0;
27
28                  protected override void Run()
29                  {
30                          // 1. Determine flanks
31                          bool keyOut = KL.stateChanges(1,0);
32                          bool keyIn  = KL.stateChanges(0,1);
33                          bool turnedToK15R = KL.stateChanges(1,2) || KL.stateChanges(3,2);
34
35                          bool doorOpened = Door.stateChanges(0,1);
36                          bool doorClosed = Door.stateChanges(1,0);
37
38                          if (doorOpened || turnedToK15R) count = 0;
39
40                          if (MotorLA.StateFb != 0) meanwhile = true;
41
42                          // 2. Determine action
43                          bool startGettingIn =
44                                  doorOpened &&
45                                  (KL.State == 0) &&
46                                  (MotorLA.ContFb <= courtesyPoint);
47
48                          bool startGettingOut =
49                                  (keyOut || (((KL.State == 1) || (KL.State == 2)) && doorOpened)) &&
50                                  (MotorLA.ContFb <= courtesyPoint);
51
52                          bool stopGettingInOut =
53                                  on && (
54                                          !MotorLA.checkStateFeedback(1000) ||
55                                          MotorLA.Lose ||
56                                          (MotorLA.ContFb > courtesyPoint) ||
57                                          (KL.State == 3) ||
```

```
58                                      doorClosed);
59
60                              bool startRestore =
61                                      !meanwhile &&
62                                      (count<3) &&
63                                      (MotorLA.ContFb>orgPos) &&
64                                      orgPosValid &&
65                                      (keyIn  || ((( KL.State == 1) || (KL.State == 2)) && doorClosed));
66
67                              bool stopRestore =
68                                      activeRestore &&
69                                      !MotorLA.checkStateFeedback(1000) ||
70                                              MotorLA.Lose ||
71                                              ((MotorLA.ContFb <= orgPos) && (orgPosValid)) ||
72                                              (KL.State == 3);
73
74                              // Start GettingInOut
75                              if ((startGettingIn  ||  startGettingOut) && !on)
76                              {
77                                      //Console.WriteLine("Start GettingInOut");
78                                      on = true;
79
80                                      MotorLA.acquireCooperation();
81                                      orgPos = MotorLA.ContFb; orgPosValid = true;
82                                      //Console.WriteLine("Save org pos to "+orgPos.ToString());
83                                      MotorLA.Cont = courtesyPoint;
84                                      MotorLA.State = 2;
85                              }
86
87                              // Stop GettingInOut
88                              if (on && stopGettingInOut)
89                              {
90                                      //Console.WriteLine("Stop GettingInOut");
91                                      on = false; meanwhile = false;
92                                      if (!MotorLA.Lose)
93                                      {
94                                              MotorLA.State = 0;
95                                              MotorLA.releaseCooperation();
96                                              meanwhile = false;
97                                      }
98                              }
99
100                             // Start Restore
101                             if (!activeRestore && startRestore)
102                             {
103                                     count++;
104                                     //Console.WriteLine("Start Restore to "+orgPos.ToString());
105                                     MotorLA.acquireCooperation();
106                                     MotorLA.State = 1;
107                                     MotorLA.Cont = orgPos;
108                                     activeRestore = true;
109                             }
110
111                             if (activeRestore && stopRestore)
112                             {
113                                     //Console.WriteLine("Stop Restore");
114                                     activeRestore = false;
115                                     if (!MotorLA.Lose)
116                                     {
117                                             MotorLA.State = 0;
118                                             MotorLA.releaseCooperation();
119                                     }
120                             }
121                     }
122             }
123     }
```

## A.3. CourtesyNoCount.cs

```
1  using System;
2  using System.Collections;
3  using System.ComponentModel;
4  using System.Drawing;
5  using System.Data;
6  using System.Windows.Forms;
7  using System.Threading;
8
9  namespace CCM.Seat
10 {
11         public class Courtesy : Comp
12         {
13                 // Coops
14                 public VFS MotorLA = new VFS(3);
15                 public VFR KL = new VFR(5);
16                 public VFR Door = new VFR(2);
17
18                 bool on = false;
19                 bool activeRestore = false;
20                 float orgPos = 0.0F;
21                 bool orgPosValid = false;
22
23                 bool meanwhile = false;
24
25                 float courtesyPoint = 60.0F;
26
27                 protected override void Run()
28                 {
29                         // 1. Determine flanks
30                         bool keyOut = KL.stateChanges(1,0);
31                         bool keyIn  = KL.stateChanges(0,1);
32                         bool turnedToK15R = KL.stateChanges(1,2) || KL.stateChanges(3,2);
33
34                         bool doorOpened = Door.stateChanges(0,1);
35                         bool doorClosed = Door.stateChanges(1,0);
36
37                         if (MotorLA.StateFb != 0) meanwhile = true;
38
39                         // 2. Determine action
40                         bool startGettingIn =
41                                 doorOpened &&
42                                 (KL.State == 0) &&
43                                 (MotorLA.ContFb <= courtesyPoint);
44
45                         bool startGettingOut =
46                                 (keyOut || (((KL.State == 1) || (KL.State == 2)) && doorOpened)) &&
47                                 (MotorLA.ContFb <= courtesyPoint);
48
49                         bool stopGettingInOut =
50                                 on && (
51                                         !MotorLA.checkStateFeedback(1000) ||
52                                         MotorLA.Lose ||
53                                         (MotorLA.ContFb > courtesyPoint) ||
54                                         (KL.State == 3) ||
55                                         doorClosed);
56
57                         bool startRestore =
58                                 !meanwhile &&
59                                 (MotorLA.ContFb>orgPos) &&
60                                 orgPosValid &&
61                                 (keyIn  || ((( KL.State == 1) || (KL.State == 2)) && doorClosed));
62
63                         bool stopRestore =
64                                 activeRestore &&
65                                 !MotorLA.checkStateFeedback(1000) ||
```

```
66                                      MotorLA.Lose ||
67                                      ((MotorLA.ContFb <= orgPos) && (orgPosValid)) ||
68                                      (KL.State == 3);
69
70                              // Start GettingInOut
71                              if ((startGettingIn || startGettingOut) && !on)
72                              {
73                                      //Console.WriteLine("Start GettingInOut");
74                                      on = true;
75
76                                      MotorLA.acquireCooperation();
77                                      orgPos = MotorLA.ContFb; orgPosValid = true;
78                                      //Console.WriteLine("Save org pos to "+orgPos.ToString());
79                                      MotorLA.Cont = courtesyPoint;
80                                      MotorLA.State = 2;
81                              }
82
83                              // Stop GettingInOut
84                              if (on && stopGettingInOut)
85                              {
86                                      //Console.WriteLine("Stop GettingInOut");
87                                      on = false; meanwhile = false;
88                                      if (!MotorLA.Lose)
89                                      {
90                                              MotorLA.State = 0;
91                                              MotorLA.releaseCooperation();
92                                              meanwhile = false;
93                                      }
94                              }
95
96                              // Start Restore
97                              if (!activeRestore && startRestore)
98                              {
99                                      //Console.WriteLine("Start Restore to "+orgPos.ToString());
100                                     MotorLA.acquireCooperation();
101                                     MotorLA.State = 1;
102                                     MotorLA.Cont = orgPos;
103                                     activeRestore = true;
104                             }
105
106                             if (activeRestore && stopRestore)
107                             {
108                                     //Console.WriteLine("Stop Restore");
109                                     activeRestore = false;
110                                     if (!MotorLA.Lose)
111                                     {
112                                             MotorLA.State = 0;
113                                             MotorLA.releaseCooperation();
114                                     }
115                             }
116                     }
117             }
118     }
```

# Literaturverzeichnis


[CS01] Cleaveland, Rance und Oleg Sokolsky: *Equivalence and Preorder Checking for Finite-State Systems.* In: *Handbook of Process Algebra*, Seiten 391–424, 2001.

[ECM02] ECMA International: *C# Language Specification*, 2002. Verfügbar unter `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`.

[Fre02] Free Software Foundation: *Bison – the YACC compatible Parser Generator*, 2002. Verfügbar unter `http://www.gnu.org/software/bison/manual/`

[HNW98] Huhn, Michaela, Peter Niebert und Heike Wehrheim: *Partial Order Reductions for Bisimulation Checking.* In: *Foundations of Software Technology and Theoretical Computer Science*, Seiten 271–282, August 1998.

[Hol97] Holzmann, Gerard J.: *The Model Checker Spin.* IEEE Transactions on Software Engineering, 23(5):279–295, May 1997. Verfügbar unter `http://spinroot.com/spin/Doc/ieee97.pdf`

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

[Pit03] Pitt, Martin: *Verifikation des dynamischen Verhaltens beim Entwurf von Feldbuskomponenten hinsichtlich ihrer Passfähigkeit.* Großer Beleg, Technische Universität Dresden, 2003.

[RG98] Rensink, Arend und Roberto Gorrieri: *Vertical Bisimulation.* Hildesheimer Informatik-Berichte, September 1998.

[TMTG03] Tauber, Joshua, Peter Musial, Michael Tsai und Stephen Garland: *Verifiable Code Generation from Abstract I/O Automata Models.* MIT Laboratory for Computer Science, March 2003. Verfügbar unter `http://www.csail.mit.edu/research/abstracts/abstracts03/theory/59tauber.pdf`.

[vG01] Glabbeek, Rob J. van: *The Linear Time – Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes.* In: *Handbook of Process Algebra*, 2001. Verfügbar unter `Boole.stanford.edu/pub/spectrum1.ps.gz`

[VHB+03] Visser, W., K. Havelund, G. Brat, S. Spark und F. Lerda: *Model checking programs.* Automated Software Engineering Journal, 10(2), April 2003. Verfügbar unter `http://ase.arc.nasa.gov/visser/ase00FinalJournal.pdf`