Brandenburgische Technische Universität Cottbus
Institut für Informatik

Ein Analysewerkzeug für zeitbehaftete Automaten

Diplomarbeit
von

Dirk Beyer

1. Gutachter:	Prof. Dr. Claus Lewerentz
2. Gutachter:	Prof. Dr.-Ing. Monika Heiner
Betreuer:	Dr. Heinrich Rust

Cottbus, 30. Juni 1998

Erklärung

Die vorliegende Diplomarbeit wurde von mir selbständig angefertigt. Die verwendeten Hilfsmittel und Quellen sind im Literaturverzeichnis vollständig aufgeführt.

Cottbus, 30. Juni 1998

„Wenn Du ein Schiff bauen willst, dann trommle nicht Deine Männer zusammen, um Holz zu beschaffen und um Arbeit zu verteilen, sondern lehre sie die Sehnsucht nach dem weiten, endlosen Meer.“

Antoine de Saint-Exupéry

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung in die Thematik

Aufgrund des ständig größer werdenden Bedarfs an Software auch in sicherheitskritischen Anwendungen sind Werkzeuge zur Beherrschung der steigenden Komplexität und zum Nachweis der zu garantierenden Eigenschaften gefragt. Dabei gewinnen Methoden der formalen Spezifikation immer mehr an Bedeutung.

Bei der Modellierung von Systemen besteht ein wesentliches Problem in der Abstraktion von der realen Welt. Zur Modellierung von reaktiven Systemen existieren leistungsfähige und einsatztaugliche Werkzeuge. Daß bei vielen Werkzeugen jedoch von der Zeit abstrahiert werden muß, stellt einen wesentlichen Nachteil dieser Methoden dar, da zusätzlich zur Korrektheit des Systems auch die Korrektheit der Abstraktion von der Zeit bewiesen werden muß.

Am Lehrstuhl Software-Systemtechnik der BTU Cottbus wurden Methoden untersucht, die es ermöglichen, Realzeitsysteme zu modellieren, ohne von der Zeit abstrahieren zu müssen. Dazu werden die Konzepte ausgehend von einigen bereits existierenden Methoden überarbeitet und erweitert. Dem steigenden Bedarf an Steuerungssoftware für eingebettete Systeme entsprechend wird Wert auf die Modellierung von hybriden Systemen gelegt. Ziel ist die konzeptionelle Erarbeitung einer gut handhabbaren und die Möglichkeiten der automatischen Verifikation unterstützenden Methode.

In der vorliegenden Arbeit wurde eine Klassenbibliothek zur rechnerinternen Repräsentation von Realzeitmodellen entsprechend der am Lehrstuhl entwickelten Konzepte entworfen und implementiert. Diese Klassenbibliothek dient der Entwicklung eines Spezifikations- und Analysewerkzeugs in einer späteren Phase.

In den folgenden Abschnitten dieses ersten Kapitels wird der derzeitige Stand der Entwicklung auf dem Gebiet der Modellierung hybrider Systeme wiedergegeben. Dazu dient zunächst eine kurze Einführung in die Modellierung von Realzeitsystemen. Einige bestehende Werkzeuge werden vorgestellt und verglichen; Erfahrungen im Umgang mit einem dieser Werkzeuge werden dargestellt.

Das zweite Kapitel beinhaltet eine informelle Beschreibung des Aufbaus, die Konzepte und die Notation des neuen Automatenmodells. Um zusätzliche Eigenschaften sowie Unterschiede zu den anderen Notationen zu illustrieren, wird das Transportband, welches im ersten Kapitel als Modellierungsbeispiel dient, nun mit der eingeführten Notation beschrieben.

Durch das im dritten Kapitel beschriebene Compilerfrontend wird die textuelle Repräsentation des Modells in eine rechnerinterne Repräsentation überführt und zur weiteren Verarbeitung bereitgestellt. Die verschiedenen Compilerphasen werden vorgestellt; es wird der Aufbau erklärt sowie auf einige Implementierungsdetails eingegangen.

Die Transformationen zum Beseitigen einiger abkürzender Schreibweisen und zur Auflösung der Modulhierarchie werden im Kapitel 4 erklärt. Dadurch werden die Voraussetzungen für die Codeerzeugung geschaffen.

Die Beschreibung der Testautomatenanalyse sowie der Codeerzeugung für die HyTech-Notation erfolgt anschließend im fünften Kapitel. Nach dem kurzen Eingehen auf die Benutzung der Komponenten und auf die Durchführung einer Analyse am Beispielmodell werden einige Aussichten zur möglichen Weiterentwicklung des Werkzeugs gegeben.

Im sechsten Kapitel wird die Organisation des Projektes beschrieben: Neben dem Begründen einiger Entscheidungen bezüglich des Projektmanagements und der Diskussion der bei der Durchführung des Projektes aufgetretenen Probleme werden wichtige persönliche Erfahrungen wiedergegeben.

1.1 Modellierung von Realzeitsystemen

Bei einem *reaktiven System* kommt es auf die *logisch richtigen* Reaktionen auf Ereignisse aus der Umwelt an. Als theoretisches Grundkonzept wird oft ein endlicher Automat mit Ein- und Ausgabealphabet benutzt [1]. Darüber hinaus gehört bei einem *Realzeitsystem* der *richtige Zeitpunkt* der Reaktion zur Korrektheit des Systems. Zur Modellierung von Realzeitsystemen dient z. B. der Timed Automaton von Alur und Dill als theoretischer Hintergrund [2]. Werden bei einem System außer der Zeit noch weitere kontinuierliche Größen betrachtet, deren Ableitung nach der Zeit Werte ungleich eins annehmen darf, so wird von einem *hybriden System* gesprochen. Der Begriff „hybrides System“ wird hier nicht in seiner allgemeinen Bedeutung, also als System, in dem zwei unterschiedliche Konzepte vereint werden, verwendet, sondern hier ist konkret die Modellierung einerseits von Zeiten und andererseits von Größen, deren Ableitung nach der Zeit nicht konstant eins ist, gemeint. Da bei der Modellierung von hybriden Systemen also sowohl diskrete als auch kontinuierliche Aspekte zu beachten sind, bietet sich als Grundlage dazu der Hybrid Automaton von Henzinger an [3].

Somit bilden reaktive Systeme, Realzeitsysteme und hybride Systeme eine Hierarchie von Verfeinerungen (Abb. 1.1).

Bei der Entwicklung eines Modellierungskonzeptes für Realzeitsysteme stehen mehrere theoretische Grundkonzepte zur Verfügung. Zum einen können Realzeitsysteme mittels Prozeßalgebren (z. B. ACSR [4]) beschrieben werden. Die Prozeßalgebren gestatten die modulare Verifikation und die hierarchische Entwicklung von Systemen, wobei die Anforderungsspezifikation in der gleichen Sprache als abstrakter Prozeß beschrieben werden kann. Die Verifikation erfolgt über den Nachweis der Äquivalenz der beiden Prozesse.

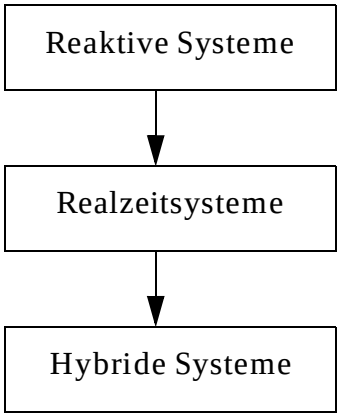

Abbildung 1.1: Hierarchie der reaktiven Systeme (nach [9])

Ein weiteres theoretisches Grundkonzept für die Realzeitmodellierung bilden die Petri-Netze mit zeitbehafteten Transitionen nach Merlin und Farber [5]. Dieses Konzept wurde von Popova-Zeugmann und Heiner erweitert zum Duration Interval Petri Net [6]. In diesen Netzen ist es möglich, Ausführungszeiten durch Intervallangaben an den Transitionen zu modellieren, wobei eine nicht-unterbrechbare Feuerregel angewendet wird, d. h. ein eventueller Konflikt wird aufgelöst, bevor überhaupt Zeit vergangen ist. Zur Analyse wird dieses Netz in ein äquivalentes Intervall-Petri-Netz oder Duration-Petri-Netz transformiert. Für Petri-Netze stehen mehrere Werkzeuge zur automatischen Verifikation zur Verfügung. Eine Methode zum Nachweis von Eigenschaften ist die Erreichbarkeitsanalyse, jedoch besteht bei der Erzeugung des Erreichbarkeitsgraphen das Problem des exponentiellen Aufwandes.

Der Hybrid Automaton stellt eine Erweiterung des Timed Automaton dar, wobei zusätzlich zu den normalen Zeitvariablen (*clocks*) noch Variablen möglich sind, bei denen ein beliebiges Intervall zur Festlegung der Ableitung nach der Zeit angegeben werden kann. Dadurch können veränderliche Größen, deren Ableitung nach der Zeit stückweise konstant ist, direkt modelliert werden. Die Verifikation des Modells erfolgt anhand von Erreichbarkeitsanalysen über Regionen. Unter einer Region wird hier die Vereinigung von konvexen Polyedern verstanden, die durch lineare Relationen gegeben sind [7].

Im folgenden wird eine informelle Beschreibung des Automatenmodells gegeben. Zur Illustration dient die Abb. 1.2, in der ein Sensor durch einen hybriden Automaten modelliert wurde, der aus drei Zuständen besteht. Der Sensor unterscheidet dabei, ob das Werkstück noch nicht am Sensor war, ob es gerade erkannt wird oder ob es sich bereits hinter dem Sensor befindet.

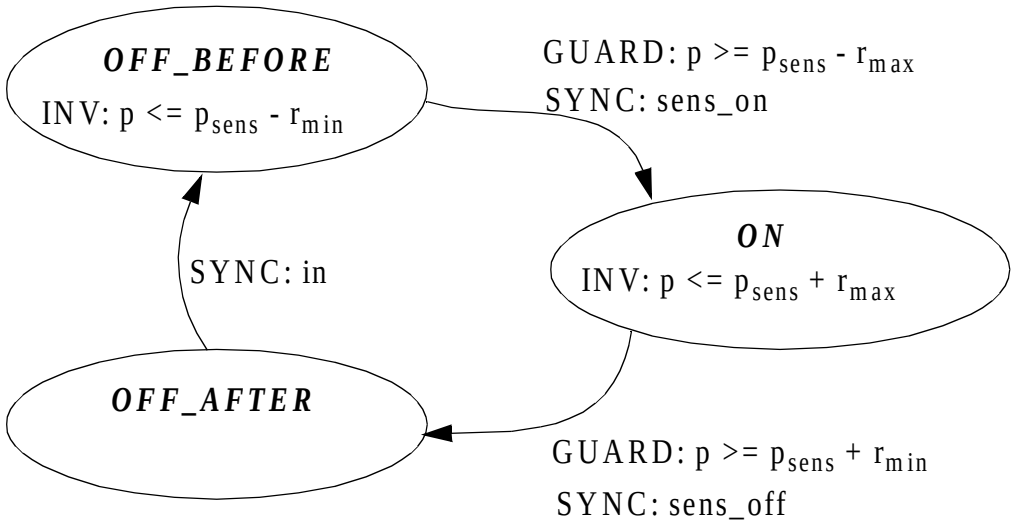

Abbildung 1.2: Hybrider Automat zur Modellierung eines Sensors

Ein hybrider Automat besteht aus:

- einer endlichen Menge von reellwertigen Variablen; zu jeder Variablen *X* existiert eine Variable *X'*, die der Ableitung von *X* nach der Zeit entspricht,
- einem Kontrollgraphen (*V, E*), der aus einer Menge von Kontrollzuständen *V* und einer Menge von Kontrollübergängen (Transitionen) *E* besteht,
- der Funktion *initial*, welche jedem Zustand eine Bedingung für die Werte der Variablen zuordnet, die erfüllt sein muß, wenn dieser Zustand Initialzustand ist,
- der Funktion *invariant*, die jedem Zustand eine Bedingung über die Werte der Variablen zuordnet, und der Funktion *flow*, die jedem Zustand eine Bedingung über die Werte der Variablen und deren Ableitungen zuordnet,
- einer Übergangsbedingung an jeder Transition, die aus einem Prädikat über den Variablen und deren Ableitungen besteht,
- einer endlichen Menge von Ereignissen und einer Funktion *event*, die jeder Transition ein Ereignis zuordnet, von dem die Auswahl der Transition abhängt.

Dieses Automatenmodell dient u. a. dem Werkzeug HyTech [8], das im folgenden Abschnitt beschrieben wird, und dem Cottbus Timed Automaton als theoretische Grundlage.

1.2 Vorstellung einiger bestehender Werkzeuge

In diesem Abschnitt werden aus mehreren existierenden Werkzeugen zur Modellierung von Realzeitsystemen die Möglichkeiten derjenigen Werkzeuge näher beschrieben, bei denen vollautomatische[1] Verifikationswerkzeuge vorhanden sind und die als Grundlage den Timed Automaton bzw. den Hybrid Automaton benutzen. Die Eigenschaften der Werkzeuge Kronos, Uppaal und HyTech werden zunächst dargestellt und danach miteinander verglichen. Eine ausführlichere Gegenüberstellung auch weiterer bestehender Methoden und Werkzeuge zur Modellierung von Realzeitsystemen gibt Lötzbeyer [9]. Verschiedene Methoden zur Modellierung von reaktiven Systemen sind durch die Anwendung auf die Fallstudie „Production Cell“ von Lewerentz und Lindner vergleichend gegenübergestellt worden [10].

1.2.1 Kronos

Kronos ist ein Werkzeug zur Verifikation von Realzeitsystemen und wurde im Forschungslabor VERIMAG in Grenoble, Frankreich, entwickelt [11].

In der Notation von Kronos gibt es keine Modularisierungsmöglichkeiten, da das gesamte System in einem Automaten modelliert wird. Diskrete Größen werden als explizite Zustände modelliert. Zur Modellierung von analogen Größen stehen nur rücksetzbare Uhren zur Verfügung, also analoge Veränderliche, deren Ableitung konstant eins ist und die bei Bedarf auf null gesetzt werden dürfen. Da es nur einen Automaten gibt, sind nur globale Variablen möglich.

Nichtdeterminismus kann durch die Entscheidungsmöglichkeit, ob ein Übergang ausgewählt oder im selben Zustand verblieben wird, entstehen. Andererseits kann Nichtdeterminismus durch die Auswahl einer von mehreren möglichen Transitionen ausgedrückt werden.

In den Bedingungsausdrücken sind lineare Einschränkungen über Uhren erlaubt.

Die Spezifikation wird in einer TCTL-Formel angegeben und mittels Model-Checking analysiert.

1.2.2 Uppaal

Mit Uppaal steht ein Werkzeug für die Validierung und Verifikation von Realzeitsystemen zur Verfügung. Es wurde an den Universitäten Uppsala und Aalborg entwickelt [13].

1. Unter einem vollautomatischen Verifikationswerkzeug wird hier ein Werkzeug verstanden, welches bei Bereitstellen von Modell und Spezifikation ohne Einbeziehung des Benutzers das Analyseresultat liefert.

Ein Modell in Uppaal beinhaltet eine Systembeschreibung, die aus einzelnen miteinander kommunizierenden Prozessen zusammengesetzt ist. Es ist also nur eine Kompositionsebene möglich, keine Hierarchie.

Alle Variablen werden global deklariert. Es sind nicht nur Uhren, sondern auch ganzzahlige Variablen und kontinuierliche Variablen möglich, deren Ableitung in der Zeit veränderlich ist und durch Intervallangaben spezifiziert werden kann. Dadurch können auch hybride Systeme beschrieben werden. Zur Modellierung des diskreten Teils können also sowohl explizite Zustände als auch ganzzahlige Variablen benutzt werden. In den Zuweisungen von ganzzahligen Variablen sind nur lineare Berechnungen möglich.

Die Synchronisation der einzelnen Prozesse erfolgt nach dem CCS-Konzept von Milner [14]. Dabei erfolgt die Synchronisation über Kanäle, die jeweils in ihrer Richtung festgelegt sind.

Die in den Bedingungen zugelassenen Ausdrücke sind auf Vergleiche von Variablen untereinander oder mit natürlichen Zahlen sowie Addition bzw. Subtraktion von natürlichen Zahlen beschränkt.

Mengen von Konfigurationen werden bei Uppaal durch konvexe Bereiche dargestellt. Dadurch wurde die Beschränkung auf lineare Ausdrücke notwendig. Die Idee dabei ist eine symbolische Repräsentation der Konfigurationen. Die einfachere Handhabung z. B. eines Vergleiches einer Uhr mit einer Konstanten, bei dem nur einige Fallunterscheidungen notwendig sind, ist vorteilhaft.

Die Spezifikation erfolgt als Formel einer eingeschränkten Form von TCTL. Als Analysemöglichkeiten stehen Erreichbarkeitstests über diesen temporallogischen Formeln und die Simulation zur Verfügung.

1.2.3 HyTech

Dieses Werkzeug zur automatischen Analyse von eingebetteten Systemen wurde von der Universität Berkeley in Zusammenarbeit mit dem Intel-Entwicklungslabor Hillsboro entwickelt [8].

HyTech ermöglicht außer der Modellierung von Realzeitsystemen auch die Modellierung von hybriden Systemen.

In einer HyTech-Modellierung wird ein System als Gruppe von kommunizierenden Automaten beschrieben. Eine Hierarchie ist dabei nicht möglich. Auch hier werden alle Variablen global deklariert. Die Variablentypen Diskret, Uhr, Stoppuhr, Parameter und Analog sind zugelassen.

Durch eine analoge Variable kann eine physikalische Größe modelliert werden, die kontinuierlich ihren Wert ändert. Diese Wertänderung ist bestimmt durch die Ableitung der Variablen nach der Zeit, die auf ein bestimmtes Intervall festgelegt wird. Beim Übergang in einen anderen Zustand kann die Ableitung verändert werden. Uhren stellen analoge

Variablen dar, bei der die Ableitung immer konstant eins betragen muß. Die Stoppuhren sind analoge Variablen, bei denen die Ableitung entweder null oder eins ist. Größen, deren Ableitung konstant null sein soll, werden durch diskrete Variablen modelliert. Alle Variablen dürfen in Zuweisungen bei den Zustandsübergängen beliebig gesetzt werden.

Die diskreten Teile der Modellierung werden in HyTech als explizite Zustände und durch diskrete Variablen formuliert. Der kontinuierliche Teil wird anhand der analogen Variablen, Uhren und Stoppuhren modelliert.

In den Bedingungsausdrücken sind lineare Einschränkungen über den Variablen möglich, also lineare Ausdrücke, die mit den Relationsoperatoren verknüpft werden.

Die Synchronisation der Automaten wird über Synchronisationsmarken nach dem CSP-Konzept von Hoare realisiert [15]. Ein Übergang mit einer Synchronisationsmarke kann nur ausgewählt werden, wenn alle Automaten, für die die gleiche Synchronisationsmarke deklariert wurde, auch einen Übergang mit dieser Synchronisationsmarke auswählen, d. h. alle anderen Automaten müssen „einverstanden“ bzw. bereit sein.

HyTech beinhaltet Werkzeuge zur Erreichbarkeitsanalyse und zur parametrischen Analyse. Die Erreichbarkeitsanalyse erfolgt unter Benutzung des Regionen-Kalküls, das die Berechnung von erreichbaren Regionen und logische Operationen darüber ermöglicht.

1.2.4 Vergleich der Werkzeuge

Allen diesen Werkzeugen sind einige Eigenschaften gemeinsam:

- Die diskreten Teile des Systems können als explizite Zustände respräsentiert werden.
- In den expliziten Zuständen wird eine Invariante angegeben, die einen Zustandsübergang erzwingt, sobald die Invariante falsch wird.
- Wenn in einem Zustand weder ein Übergang möglich noch die Invariante erfüllt ist, dann ist eine verbotene Konfiguration entstanden: Die Zeit bleibt stehen. Diese Situation wird als Zenoverhalten bezeichnet [12].
- Sämtliche Variablen werden global deklariert.
- In den Transitionen sind Bedingungen und Variablenzuweisungen enthalten.

In Tabelle 1.1 ist eine zusammenfassende Übersicht der Eigenschaften der drei Werkzeuge dargestellt.

Die mit Uppaal oder Kronos beschreibbaren Systeme sind auch mit HyTech beschreibbar; darüber hinaus stellt HyTech weitere Modellierungsmöglichkeiten zur Verfügung (z. B. die zusätzlichen Variablentypen Diskret, Analog und Stoppuhr).

Eigenschaft	Kronos	Uppaal	HyTech
Komposition	keine	Prozesse	Automaten
Hierarchie	keine	eine Ebene	eine Ebene
Sichtbarkeits-bereiche	global	global	global
Kommunika-tion	Signalausgabe	Kanäle mit Rich-tung	CSP
diskrete Zustände	expl. Zustände	expl. Zustände u. diskrete Variablen	expl. Zustände u. diskrete Variablen
Nichtdeter-minismus	1. wann Übergang 2. Auswahl der Transition	1. wann Übergang 2. Auswahl der Transition 3. Wert der Ableitung	1. wann Übergang 2. Auswahl der Transition 3. Wert der Ableitung
Variablen-typen	clock	clock, int	clock, stopwatch, discrete, analog, parameter
Wert der Ableitung	1	Angabe eines Intervalls	*clock:* 1, *stopwatch:* 0 oder 1, *analog:* Intervall
Ausdrücke	linear	linear	linear
Analyse	TCTL	Erreichbarkeit	Erreichbarkeit, parametrisch

Tabelle 1.1: Vergleich der bestehenden Werkzeuge

1.3 Fallstudie: Modellierung mit HyTech

In diesem Abschnitt werden Ergebnisse und Erfahrungen vorgestellt, die in einer von Rust angefertigten Fallstudie erlangt wurden [16]. Ziel dieser Fallstudie war es, Erfahrungen im Umgang mit HyTech zu sammeln und daraus resultierend Konzepte zu entwickeln, die eine besser handhabbare Modellierung gestatten.

Als Modellierungsobjekt diente ein Transportband der Modellfertigungs-anlage des Lehrstuhls. Eine genauere Beschreibung der Fertigungsanlage erfolgt in [17]. Dabei wurden sowohl die physikalischen Bestandteile als

auch die Steuerung und der Kontext des Bandes berücksichtigt. Im Anschluß an die Modellierung wurden einige Sicherheitseigenschaften spezifiziert und die Analysemöglichkeiten von HyTech erprobt.

Während der Modellierung und Spezifikation sind die folgenden Verbesserungsideen entstanden:

- Die Synchronisation erfolgt in HyTech mittels Synchronisationsmarken. Ein Signal arbeitet dabei nicht in einer bestimmten Richtung. Bei der Modellierung wäre es günstig, **Eingabe- und Ausgabesignale** zu unterscheiden. Eingabesignale dürfen den Rest des Systems nicht einschränken, d. h. in jedem Zustand muß für jedes Eingabesignal eine Transition mit dem entsprechenden Signal auswählbar sein. Die vom Entwickler nicht berücksichtigten aber notwendigen Transitionen sollten automatisch generiert werden und zu einen **Fehlerzustand** führen. Dieser zeigt dann, daß mit dem Auftreten dieser Signale nicht gerechnet wurde und daß das System durch die Einschränkung dieser Signale beeinflußt worden wäre. Durch diese Erweiterung ist leicht nachweisbar, ob ein Testautomat (Eingabemodul) wirklich nur „abhorcht“.
- Im Normalfall sollte ein Signal oder eine Variable nur von einem Automaten eingeschränkt (also nur in einem Modul als Ausgabesignal oder Ausgabevariable deklariert) werden dürfen. **Mehrfach einschränkbare** Signale (oder Variablen) sollten als solche deklariert werden.
- Im zu modellierenden System existieren einige ähnliche Komponenten. Um die Modellierung übersichtlicher und strukturierter zu gestalten, wäre ein **Schablonen-Mechanismus** hilfreich, der es erlaubt, von einer Komponente durch Angabe von Parametern leicht geänderte Instanzen zu erzeugen.
- In einigen Situationen soll durch ein („ankommendes“) Signal als **sofortige Reaktion** ein („erzeugtes“) anderes Signal ausgelöst werden. Dieses Verhalten könnte zur Vereinfachung der Modellierung in einer einzigen Transition notiert werden.
- Zur besseren Strukturierung ist es günstig, eine hierarchische Anordnung von Modulen als **Komposition von Untermodulen** zu unterstützen. Weiterhin ist die Einführung von **Gültigkeitsbereichen** für Namen von Signalen und Variablen wichtig, um eine Kapselung zu erreichen.
- Mit einer **Analyse auf Determinismus** könnte kontrolliert werden, ob für einen Automaten eine automatische Generierung von ausführbarem Code (z. B. für das Steuerprogramm) möglich ist.

1.4 Aufgabenstellung

Ausgehend von den Erfahrungen mit HyTech bei der im vorherigen Abschnitt beschriebenen Fallstudie wurde von Rust am Lehrstuhl Software-Systemtechnik der BTU Cottbus ein erweitertes Automatenmodell entwickelt [18]. Dabei wurden die in der Fallstudie erlangten Verbesse-

rungsideen berücksichtigt. Zu diesem Automatenmodell wird im Rahmen dieser Arbeit eine Notation entworfen und durch eine Grammatik definiert. Um einen automatischen Nachweis von Eigenschaften, die für das Modell gefordert werden, führen zu können, ist eine Werkzeugunterstützung notwendig. Insbesondere ist die Möglichkeit der Erreichbarkeitsanalyse bereitzustellen, da mit dieser Methode sehr viele Eigenschaften nachweisbar sind.

Um CTA-Modelle aus der textuellen Repräsentation in die rechnerinternen Strukturen überführen zu können, ist zunächst ein Compilerfrontend zu entwickeln, welches die weitere Verarbeitung der Modelle ermöglicht.

Wie die Abb. 1.1 zeigt, werden am CTA-Modell Transformationen vorgenommen, die abkürzende Schreibweisen und implizite Eigenschaften auflösen. Um zu einer Normalform zu gelangen, in der keine Hierarchie mehr vorhanden ist, dienen die Algorithmen zur Auflösung der Enthaltenseinshierarchie. Die Beseitigung der modularen Strukturen ist notwendig, um das Modell in die HyTech-Notation transformieren zu können.

Anhand dieser Normalform kann der Produktautomat gebildet werden, der direkt durch die Erreichbarkeitsanalyse untersucht werden kann, da es sich hierbei um einen endlichen Automaten handelt, bei dem keine Synchronisation mehr berücksichtigt werden muß. In der vorliegenden Diplomarbeit wurde jedoch dieser Schritt nicht realisiert. Es wurde eine Codeerzeugung implementiert, die aus dem in der Normalform befindlichen CTA-Modell ein Modell in der HyTech-Notation erzeugt. Durch diesen Schritt können sofort die Analysemöglichkeiten von HyTech benutzt und erste Erfahrungen im Umgang mit CTA-Modellen gesammelt werden.

Die in diesem Projekt realisierten Ablaufschritte sind in der Abb. 1.1 in dem von der gestrichelten Kurve umgebenen Bereich dargestellt.

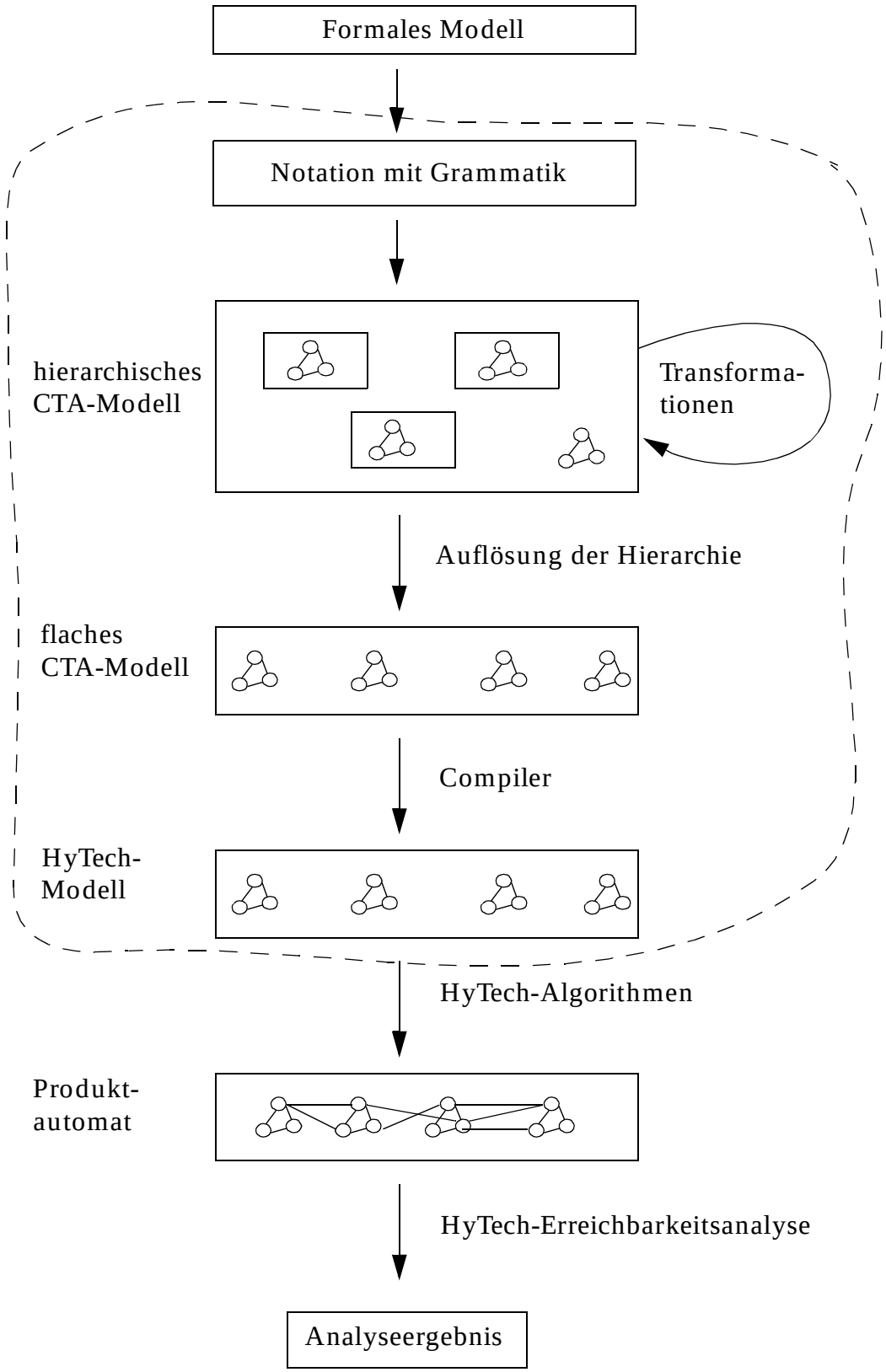

Abbildung 1.1: Ablauf bei der Arbeit mit CTA-Modellen

2 Cottbus Timed Automata

In Anlehnung an HyTech ist auch der Cottbus Timed Automaton (im folgenden CTA abgekürzt) zur Modellierung von hybriden Systemen bestimmt. Die wesentlichen Konzepte von HyTech finden sich im CTA wieder, jedoch wurden die in der Fallstudie erlangten Erfahrungen zur Entwicklung von Verbesserungen genutzt und neue Konzepte eingebracht. Der Cottbus Timed Automaton wurde von Rust formal definiert [18]. Mit diesem Konzept wird eine komfortablere und übersichtlichere Modellierung erlaubt. Im folgenden Abschnitt wird der Aufbau eines CTA informell beschrieben.

2.1 Bestandteile des CTA

Ein Cottbus Timed Automaton ist ein Modul. Dieses Modul besteht aus:

- einem Namen, der zur Identifizierung dient,
- einem Satz von Schnittstellenkomponenten:
  - Eingabevariablen und -signale,
  - Ausgabevariablen und -signale,
  - mehrfach einschränkbare Variablen und Signale,
- einem Satz von lokalen Komponenten:
  - lokale Variablen und Signale,
  - eine Menge von Automaten, durch die das Verhalten des Moduls beschrieben wird,
  - eine Menge von Instanzen anderer Module,
  - eine Initial-Konfiguration, die die Werte der lokal kontrollierten Variablen und den Zustand des Automaten zu Beginn festlegt.
  - der Ausdruck DER(*var*) zu jeder lokalen und Ausgabevariablen *var* des Typs ANALOG und STOPWATCH, durch den die Ableitung der Variablen eingeschränkt wird.

Jeder Automat wird durch eine Menge von Zuständen beschrieben, die jeweils die folgenden Komponenten enthalten:

- einen Namen zur Identifikation,
- eine Invariante, die bestimmt, daß der Automat sich nur im beschriebenen Zustand befinden darf, solange die Invariante erfüllt ist,
- eine Menge von Einschränkungen der Ableitungen von Variablen,
- eine Menge von Transitionen.

Sobald die Invariante eines Zustands falsch wird, muß der Automat in einen anderen Zustand übergehen.

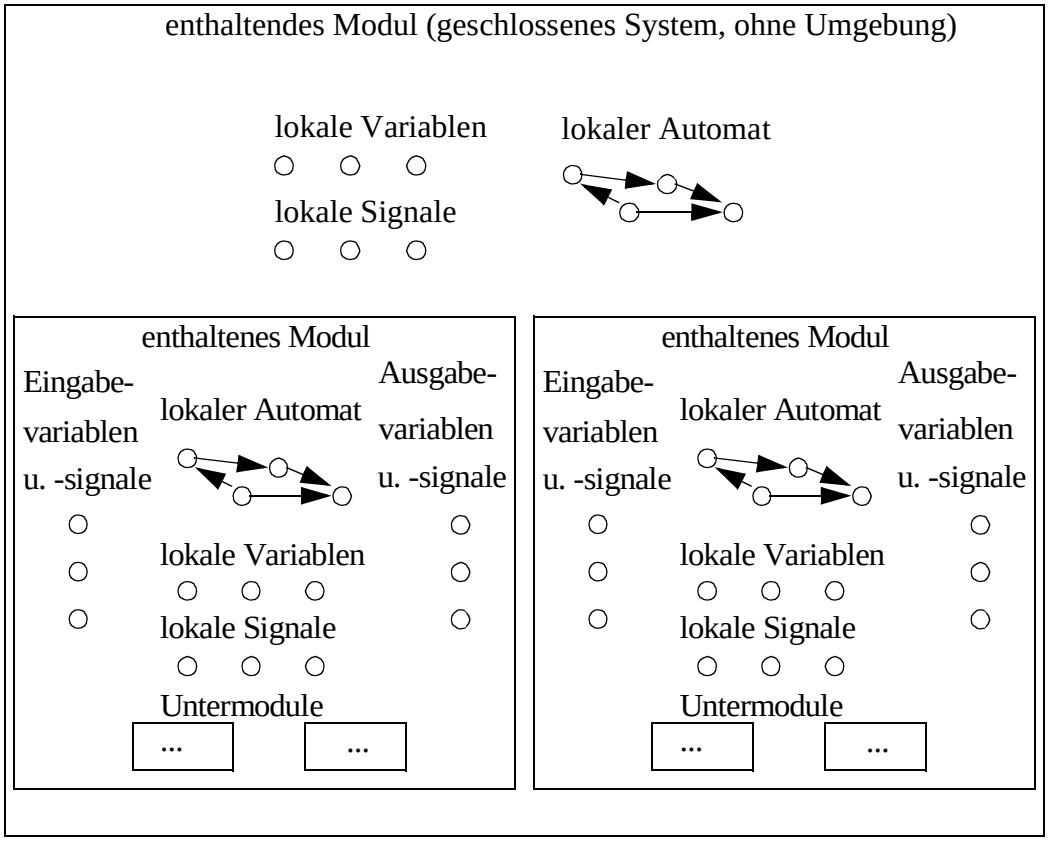

Abbildung 2.1: Modulstruktur eines CTA-Systems

Jede Transition besteht aus:

- einer Menge von möglichen Folgezuständen, die bestimmt, in welchen Zustand übergegangen werden kann,
- einem Wächter, der einen booleschen Ausdruck über die Variablen bildet und welcher die Bedingung zum Zustandsübergang ist,
- einer Synchronisationsanweisung, die anhand von Signalen die Synchronisation mit anderen Automaten ermöglicht,
- einer Menge von Variablenzuweisungen.

Die Struktur eines mit dem CTA-Konzept modellierten Systems zeigt die Abb. 2.1. Jeder Schnittstellenkomponente eines Submoduls, die außerhalb dieses Submoduls verwendet werden soll, wird eine Komponente des enthaltenden Moduls zugeordnet. Diese Zuordnungen werden bei der Instanziierung vorgenommen. Jedes Modul hat einen eigenen Namensraum für Komponentenbezeichner. Zur syntaktischen Veranschaulichung der Bestandteile eines Moduls dient Abb. 2.2.

```
MODULE MSensor
{     INPUT
            in:          SYNC;
            pos:         ANALOG;
            sens_pos:    CONST;
      OUTPUT
            sens_on:     SYNC;
            sens_off:    SYNC;
      LOCAL
            sensor_radius_min = 3: CONST;
            sensor_radius_max = 5: CONST;
      INITIALIZATION
      {
            STATE = off_after;
      }
      AUTOMATON
      {
            STATE off_before
            {           ...
            }
            ...
      }
}
```

Abbildung 2.2: Syntaktischer Aufbau eines Moduls

2.2 Konzepte und Notation für den CTA

Die Konzepte des CTA und die im Rahmen der Diplomarbeit entwickelte **Modellierungssprache** zur Notation von CTA-Modellen werden in den folgenden Abschnitten vorgestellt. Diese Notation soll - ausgehend von der HyTech-Notation - die Ergänzungen beinhalten, die sich in der Fallstudie als günstig erwiesen haben.

2.2.1 Komponentendeklaration

In der hier beschriebenen Notation wird zwischen vier Kontrollbereichen, welche die Zugriffsrechte regeln, unterschieden. In der Abb. 2.2 wird eine Variablendeklaration verdeutlicht, wobei drei Eingabe-, zwei Ausgabe- und zwei lokale Komponenten deklariert werden.

- LOCAL

  Lokale Variablen bzw. Signale sind nur innerhalb des Moduls verfügbar, in dem sie deklariert wurden. Die im Modul enthaltenen Submodule können über ihre Schnittstelle auf diese lokalen Komponenten zugreifen.

- INPUT

  Eingabekomponenten dürfen im Modul nicht eingeschränkt werden. Sie dienen lediglich zur Informationsübermittlung von außen nach innen; es wird also vom Modul eine Garantie an die Umgebung gegeben.

- OUTPUT

  Ausgabevariablen und -signale, die nur innerhalb des Moduls (und der darin enthaltenen Module) verändert werden sollen, müssen mit OUTPUT deklariert werden. Äußere Module dürfen auf solche Variablen also nicht einschränkend wirken; vom Modul wird eine Anforderung an die Umgebung gestellt.

  Zweck der Ausgabesignale ist die Absicherung, daß innerhalb eines Moduls davon ausgegangen werden kann, daß alle anderen Module dieses Signal jederzeit „empfangen“ können.

  Ausgabevariablen hingegen stellen sicher, daß die innerhalb des Moduls in dieser Variablen abgelegte Information außerhalb des Moduls garantiert nicht verändert wird.

- MULTREST (multiply restricted)

  Werden Variablen oder Signale mit MULTREST als mehrfach eingeschränkt deklariert, können in verschiedenen Modulen Zuweisungen für die Variablen (und deren Ableitung) sowie Einschränkungen der Signale stattfinden.

2.2.2 Variablentypen und Konstanten

Um die Verwendungsmöglichkeiten von Variablen eingrenzen und nützliche Redundanz einbringen zu können, wird jeder Variablen ein Datentyp zugeteilt. Als Datentypen stehen (in Anlehnung an HyTech) die im folgenden beschriebenen Möglichkeiten zur Verfügung. Die Deklaration von einer analogen Variablen und einigen Konstanten wird in Abb. 2.2 illustriert.

- CONST

  Eine Konstante dient zur Parametrisierung eines Moduls und ist nicht veränderbar.

- DISCRETE

  Die Ableitung einer diskreten Variablen ist zu jeder Zeit gleich null. Sie wird zur Speicherung von reelwertigen Zahlen benutzt.

```
STATE start
{
        -- Wait for start of transfer and start belt.
        TRANS    in_transfer;
                SYNC      ?start_in_transfer THEN !go;
}
```

Abbildung 2.3: Signale in einer Transition

- CLOCK

  Zur Modellierung der Zeit dient eine Uhrenvariable. Der Wert der Ableitung beträgt immer eins; die Uhr kann also nicht angehalten werden.
- STOPWATCH

  Im Gegensatz zur normalen Uhr kann der Wert der Ableitung entweder eins oder null betragen; die Uhr läuft oder ist angehalten. Beide Uhrentypen sind beliebig setzbar.
- ANALOG

  Zur Modellierung einer stückweise kontinuierlichen Größe stehen analoge Variablen zur Verfügung, deren Ableitungen nach der Zeit auf beliebige Intervalle festgesetzt werden dürfen.

2.2.3 Synchronisation

In der CTA-Notation werden die in HyTech verwendeten theoretischen Grundlagen des CSP-Konzepts zur Synchronisation weiter verwendet. Auch die Kommunikation über gemeinsam benutzte Variablen ist möglich.

Synchronisationsmarken (im folgenden als Signale bezeichnet) repräsentieren keinen Wert, sondern sie schränken die Transitionen der Automaten in der Weise ein, daß eine Transition mit einem bestimmten Signal nur dann gewählt werden kann, wenn alle anderen Automaten, die dieses Signal kennen, auch eine Transition mit demselben Signal auswählen.

Wie in Abb. 2.2 dargestellt, wird ein Signal mit dem Schlüsselwort *SYNC* deklariert. Durch das Schlüsselwort *THEN* in der *SYNC*-Anweisung hat der Entwickler in dieser Notation zusätzlich die Möglichkeit, in einer Transition als Reaktion auf ein Signal gleich wieder ein Signal auszulösen. Dabei werden zuerst alle Automaten, die das Signal vor *THEN* kennen, synchronisiert und im Anschluß daran die Automaten, die das Signal nach *THEN* kennen.

Im Beispiel der Abb. 2.3 wird folgendes Verhalten des Steuerprogrammes modelliert: Erhält das Steuerprogramm von der Umgebung das Signal zum Start des Transports, soll es sofort den Motor anschalten. Damit im Zustand

start die Transition zum Folgezustand *in_transfer* ausgewählt werden kann, müssen alle Automaten, die dieses Signal *start_in_transfer* kennen, auch eine Transition mit demselben Signal auswählen. Danach wird durch die Transition das Signal *go* zum Start des Motors ausgelöst.

Diese Ergänzung vereinfacht Modellierungssituationen, bei denen sonst durch ein Signal in einen Zustand übergegangen werden muß, in dem - ohne Zeit vergehen zu lassen - die Transition mit dem zu erzeugenden (Reaktions-) Signal gewählt wird.

Für Signale gelten bezüglich der Gültigkeitsbereiche die gleichen Regeln wie für Variablen. Ein Signal wird im Normalfall von nur einem Automaten eingeschränkt und dort als Ausgabesignal deklariert. Sollen mehrere Module ein Signal einschränken können, so muß das Signal als mehrfach eingeschränkt deklariert sein. Zur besseren Übersicht werden in den Automaten Eingabesignale mit „*?*“, Ausgabesignale mit „*!*“ und mehrfach eingeschränkte Signale mit „*#*“ gekennzeichnet.

2.2.4 Testautomaten

Ein Testautomat ist ein Automat, der das Verhalten seiner Umgebung beobachtet, aber nicht beeinflußt, also ein reiner Eingabeautomat. Dabei ist sicherzustellen, daß jedes Signal, das für den Automaten deklariert wurde, in jedem Zustand zu jeder Zeit verarbeitet werden kann. Trifft diese Forderung für ein Signal in einem Zustand nicht zu, so kann der Automat in einen Testautomaten transformiert werden. Eine Transition mit diesem Signal wird eingefügt , die immer dann auswählbar ist, wenn keine andere Transition mit diesem Signal ausgewählt werden kann. Solche Transitionen führen in einen Fehlerzustand, von dem aus mit allen Signalen nur wieder in denselben Fehlerzustand übergegangen werden kann. Der Automat gelangt also immer dann in den Fehlerzustand, wenn er ohne die eingefügten Transitionen die Umgebung beeinflußt hätte, d. h. wenn sich die Umgebung anders als vom Automaten erwartet verhalten hat.

In dem im Anhang A aufgelisteten Transportbandmodell ist ein Testautomat (*MTestTime*) dargestellt, bei dem diese Situation auftritt. Um überprüfen zu können, ob ein Modul nur lokale und Eingabekomponenten enthält, wird vom Analysewerkzeug für jedes Modul eine Funktion *IsObserver()* bereitgestellt.

```
INST iTestIn
  FROM MTestTime
  WITH
  {
    begin AS in;
    end AS stop_in_transfer;
  }
```

Abbildung 2.4: Modulinstanziierung innerhalb eines Moduls

2.2.5 Modulkonzept: Schablonen, Instanzen und Komposition

Da der Modulbegriff sehr vielseitig Verwendung findet, wird zur Abgrenzung das hier benutzte Modulkonzept vorgestellt, welches die folgenden Prinzipien beinhaltet:

- Kapselung und Geheimnisprinzip,
- Schablonen und Instanziierungen,
- Abstraktion und Komposition.

Kapselung und Geheimnisprinzip. Ein CTA-Modul ist ein Modellierungsbaustein, der zur Kapselung von Verhalten und Daten der im Modul enthaltenen Komponenten dient. CTA-Module bestehen aus einer Schnittstelle und einer Implementierung. Die Schnittstelle ist durch die Deklaration der Kommunikations- und Parametrisierungskomponenten definiert, die Implementierung besteht aus einem Automaten und einer Menge von Modul-Instanziierungen.

Module haben eine *Konfiguration*, also ein „Gedächtnis“. Diese Konfiguration *C = (S, A(V))* legt fest, in welchem *Zustand* sich der Automat zu einem bestimmten Zeitpunkt befindet und wie die im Modul deklarierten Variablen belegt sind.

Mit einer Instanz eines Moduls kann nur über die Schnittstelle kommuniziert werden, d. h. das Verhalten einer Modulinstanz kann nur durch Kommunikation über die Schnittstellenkomponenten beeinflußt werden.

Schablonen und Instanziierungen. Um die Modellierung ähnlicher Module zu vereinfachen und den Grad der Wiederverwendung zu erhöhen, wird jedes Modul als Schablone beschrieben. In einer Moduldefinition kann von einem anderen Modul eine Instanz erzeugt werden. Bei der Instanziierung erfolgt die Zuordnung der Schnittstellenkomponenten eines enthaltenen Moduls zu den Variablen bzw. Signalen des enthaltenden Moduls. Die Syntax ist im Beispiel der Abb. 2.4 verdeutlicht. Dort wird eine Instanz (*iTestIn*) des Moduls *MTestTime* erzeugt, wobei die Signale denen des enthaltenden Moduls zugeordnet werden. Zur graphischen Ver-

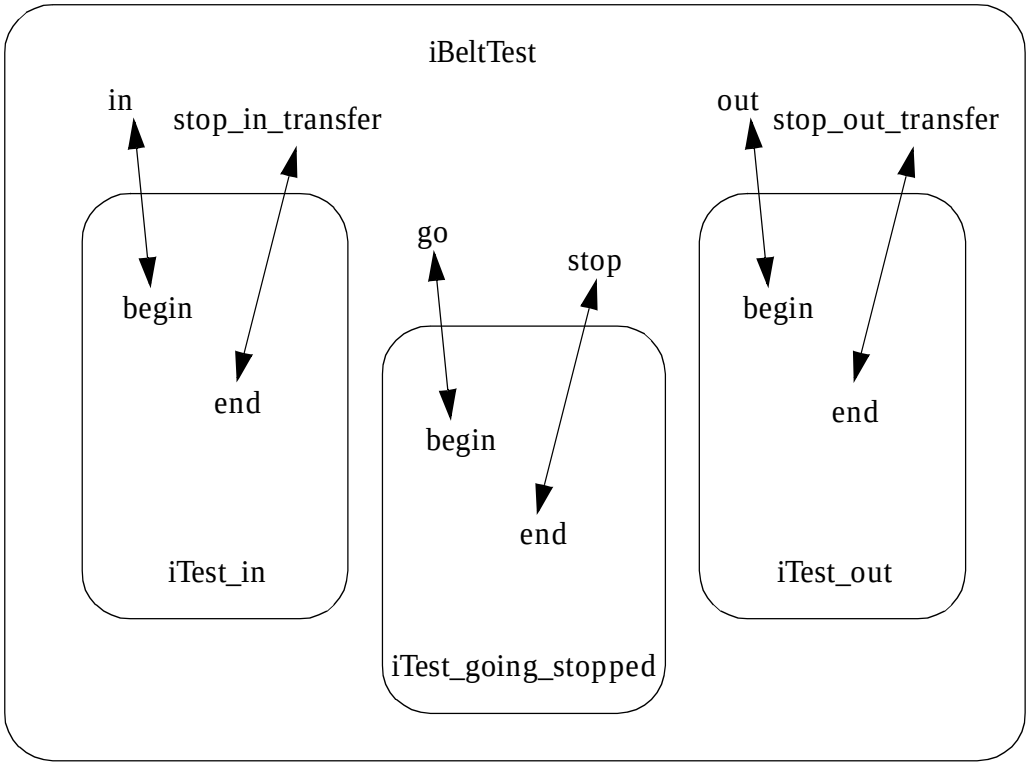

Abbildung 2.5: Innere Struktur der Instanz iBeltTest

anschaulichung wurde in der Abb. 2.5 durch Pfeile verdeutlicht, wie die Signale der Untermodule mit den Signalen des enthaltenden Moduls korrespondieren.

Zur Veranschaulichung kann eine Instanz eines Moduls mit einem Exemplar eines bestimmten Schaltkreistyps (Hardware) verglichen werden. Wie bei einer Modulinstanz wird auch bei einer integrierten Schaltung das Verhalten und die konkrete Verwendung durch die „Beschaltung“ von außen festgelegt. Auch bei einer integrierten Schaltung ist eine Analyse sowohl mit als auch ohne Beschaltung möglich.

Ein CTA-Modul wird entweder zur Modellierung eines geschlossenen Systems (ohne Schnittstelle) oder als Schablone zur Erzeugung von parametrisierten Instanzen (Black-Box-Wiederverwendung) genutzt. Sowohl Systeme als auch Schablonen können unabhängig voneinander analysiert werden.

Abstraktion und Komposition. Um die Architektur eines Modells beschreiben zu können, ist es notwendig, überschaubare Abstraktionsebenen definieren und Gruppierung von Untereinheiten vornehmen zu können. Deshalb stehen CTA-Module zueinander in einer Kompositionsbeziehung; Instanzen eines Moduls können in anderen Modulen enthalten sein und es entsteht somit eine baumartige streng hierarchische Struktur ohne Zyklen.

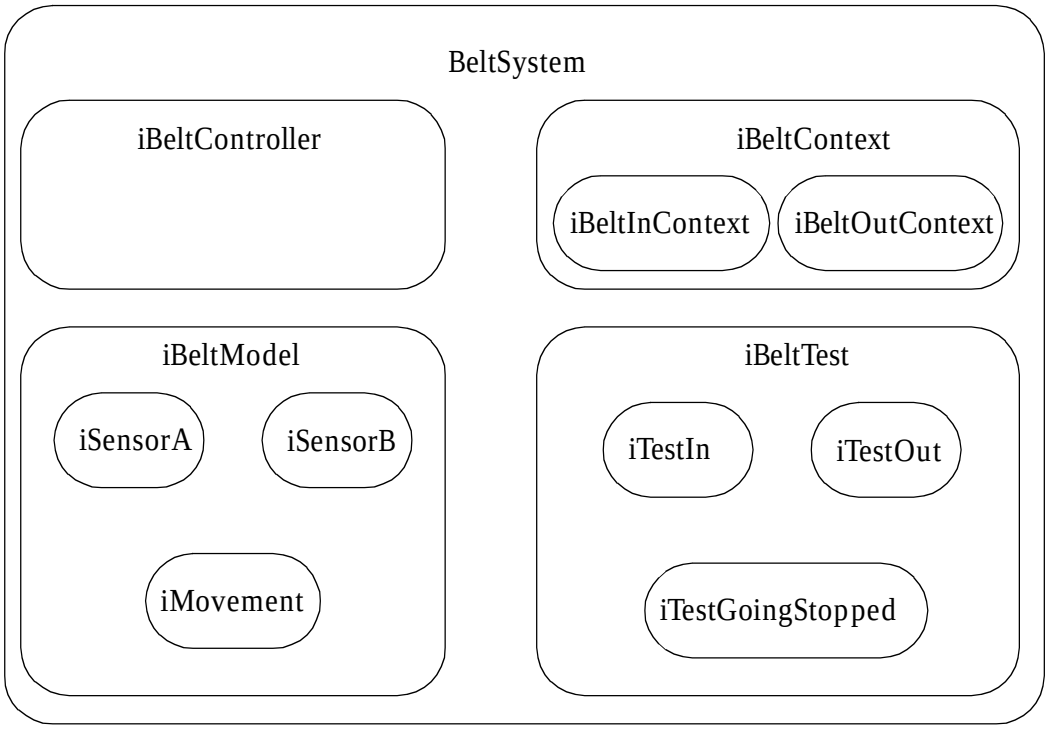

Abbildung 2.6: Instanzenstruktur des Förderband-Systems

In der Abb. 2.6 wird illustriert, wie ein System aus Modulen zusammengesetzt werden kann.

2.2.6 Code-Generierung

Aus der Modellierung der Steuerung (*iBeltController* in Abb. 2.6) soll ein ausführbares Programm generiert werden können. Dazu muß dieser Teil des Systems aus deterministischen Automaten bestehen. Für die Vorbereitung der Code-Generierung wäre es eine sinnvolle Ergänzung, wenn ein solcher Test auf Determinismus für jedes Modul, z. B. durch eine Funktion *IsDeterministic()*, vom Analysewerkzeug bereitgestellt wird.

2.3 CTA-Modellierung eines Transportbandes

Um mit dem im vorherigen Abschnitt vorgestellten Konzept Erfahrungen zu sammeln und ein Beispiel zur Veranschaulichung der Konzepte bereitzustellen, wird in diesem Abschnitt das gleiche Förderband der Fertigungsanlage modelliert wie in der Fallstudie zu HyTech. Auf diese Fallstudie wird in der vorliegenden Arbeit zur Illustration von Zusammenhängen Bezug genommen. Die Struktur des Systems wird in der Abb. 2.6 gezeigt. Das System besteht aus dem physikalischen Modell, der Steuerung, dem Umgebungskontext des Förderbandes und einem Testmodul.

In den folgenden Abschnitten wird der Aufbau der Komponenten des Systems kurz vorgestellt. Den vollständigen Quelltext der Förderband-Modellierung enthält Anhang A. Eine ausführlichere Beschreibung des CTA-Modells bieten Beyer und Rust [19].

2.3.1 Physikalisches Modell

Das physikalische Modell (*iBeltModel*) des Transportbandes besteht aus dem Motor (*iBeltMovement*) und zwei Sensoren (*iBeltSensorA, iBeltSensorB*). Diese Bestandteile sind in je einer Modulinstanz modelliert, in der das Verhalten des jeweiligen Anlagenteils spezifiziert und eine Schnittstelle zur Kommunikation nach außen zur Verfügung gestellt wird. Weil sich die beiden Sensoren voneinander nur durch ihre Position auf dem Transportband unterscheiden, sind sie von der gleichen Moduldefinition durch Angabe des Parameters für die Position instanziiert worden. Durch die hierarchische Gliederung präsentiert sich das physikalische Modell nach außen als Einheit.

2.3.2 Steuerung

In der Steuerung (*iBeltController*) wird der Ablauf der Aktionen, die zum Transport auf dem Band notwendig sind, festgelegt. Die Steuerung dient als Bindeglied zwischen der Umwelt und dem physikalischen Modell und kann als „Software“-Komponente aufgefaßt werden.

2.3.3 Kontext

Die Umgebung des Transportbandes (*iBeltContext*) ist in zwei Module gegliedert. Zum einen existiert eine Umgebung zum eingabeseitigen Nachbarn (*iBeltInContext*), der mitteilt, ob das nächste Werkstück zum Transport bereitsteht, und zum anderen existiert eine ausgabeseitige Umgebung (*iBeltOutContext*), an die das transportierte Werkstück übergeben werden muß. Durch die Zusammenfassung zum Gesamtkontext (*iBeltContext*) entsteht ein Teilsystem.

2.3.4 Testmodul

Das Testmodul (*iBeltTest*) besteht aus drei Submodulen, die jeweils eine parametrisierte Instanz des als Schablone dienenden Moduls *MTestTime* sind. Dieses Modul mißt die Zeit zwischen dem Auftreten zweier Signale (*begin* und *end*). In der ersten Instanz (*iTestIn*) wird die Zeit zwischen dem Eintreffen des Signals *in* und dem Eintreffen des Signals *stop_in_transfer* gemessen. Wie in der Abb. 2.5 durch die Pfeile verdeutlicht, werden bei der Instanziierung die Schnittstellensignale des enthaltenen Moduls (*begin* und

end) mit den Signalen der enthaltenden Modulinstanz *iBeltTest* verknüft. Analog dazu erfolgt die Instanziierung der beiden anderen enthaltenen Module.

3 Compiler-Frontend

Das Fernziel dieses Projektes ist ein Werkzeug zur Unterstützung bei der Modellierung und Analyse von Realzeitsystemen. Die Abb. 3.1 stellt den Ablauf bei der Benutzung des Werkzeugs dar. Ausgangspunkt ist die Modellierung in der CTA-Notation. Diese wird als Textdatei zur Verfügung gestellt und durch das Compiler-Frontend in ein Objektsystem transformiert, welches den abstakten Syntaxbaum für alle angebotenen Analysen als Laufzeit-Datenbasis bereitstellt.

In diesem und den folgenden Kapiteln wird die praktische Realisierung der im Kapitel 2 beschriebenen Konzepte dargestellt. Im Rahmen dieser Arbeit wurde zunächst eine Klassenbibliothek für CTA-Modelle entwikkelt, durch die alle notwendigen Bausteine zum Scannen, zum Parsen, zur flexiblen Repräsentation der CTA-Modelle und zur Überprüfung der Kontextbedingungen bereitgestellt werden.

Durch das im Rahmen dieser Arbeit entwickelte Werkzeug wird die praktische Erprobung der Modellierung mittels CTA möglich, von welcher Aussagen über die Analysierbarkeit von CTA-Modellen erwartet werden. Es soll festgestellt werden, inwieweit sich diese flexibel in einer internen Repräsentation verwalten lassen.

3.1 Entwurfs- und Programmierentscheidungen

In diesem Abschnitt werden Festlegungen bezüglich des Entwurfs und der Implementierung des Werkzeugs beschrieben. Diese Richtlinien sollen die Entwicklung eines verständlichen Klassensystems unterstützen, indem auf den konsistenten und sinnvollen Einsatz der Konzepte der objektorientierten Programmierung und der verwendeten Programmiersprache Wert gelegt wird.

Da die Entwicklung des CTA-Konzepts noch nicht abgeschlossen ist und somit noch Veränderungen zu erwarten sind, müssen diese leicht in die Implementierung überführbar sein. Weil der Parser ein CTA-Modell mit neuer Syntax durch die neue Grammatik in denselben abstrakten Syntaxbaum übersetzt, sind Änderungen an der Syntax der Notation ohne Änderung der Klassenbibliothek möglich.

Um diesen Anforderungen an Flexibilität zu entsprechen, werden die Konzepte und Methoden der objektorientierten Systementwicklung verwendet [20]. Das vielseitige Verwenden von Vererbung beim Entwurf erschwert dessen nachfolgende Umsetzung in die Implementierung, weil sehr viele Abhängigkeiten zu beachten sind. Die Verständlichkeit des Klassensystems wird geringer. Daher werden die Möglichkeiten der Vererbung in diesem Projekt nur eingeschränkt benutzt.

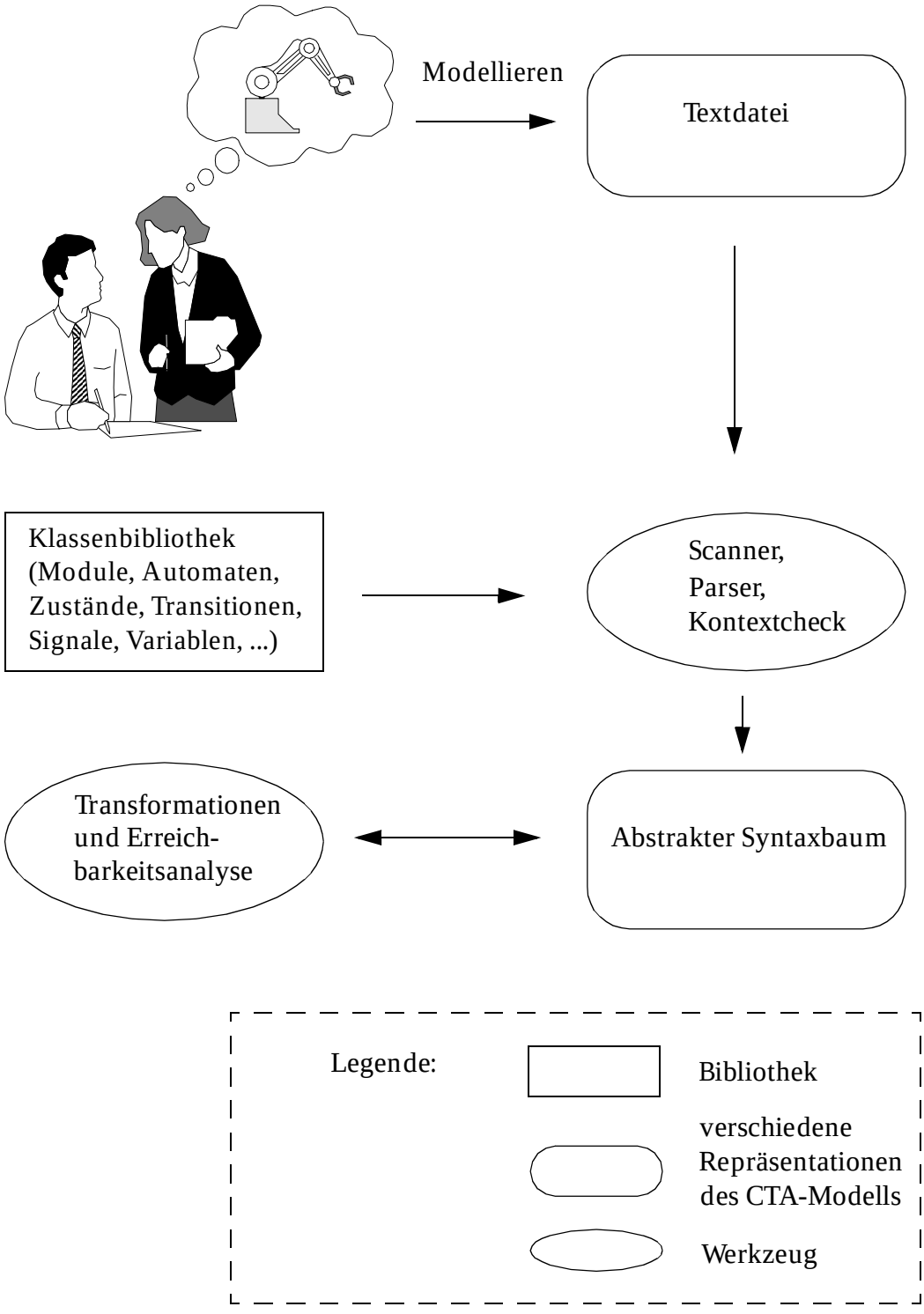

Abbildung 3.1: Ablauf bei Benutzung des Analysewerkzeugs

Um eine gute Qualität des Klassensystems, vor allem Portierbarkeit und Verständlichkeit der Quelltexte, zu erreichen, wurden für die Entwicklungsarbeiten die folgenden Festlegungen getroffen:

- Als Implementierungssprache wird die objektorientierte Programmiersprache C++ in der Standardisierung von ANSI benutzt [21], weil diese sehr weit verbreitet ist und das System somit portierbar wird. Diese Überprüfung erfolgt durch den in diesem Projekt verwendeten Compiler (*g++*) durch Benutzung der Compileroptionen *-ansi* und *-Wall*.
- Die Verwendung von Vererbung wird in diesem Projekt auf die Modellierung von Ist-Ein-Beziehungen eingeschränkt.
- Um eine robuste Klassenbibliothek zu entwickeln, die eine polymorphe Einsetzbarkeit von Untertypen gewährleistet, wird konforme Vererbung verwendet [22]. Da in C++ kein Überschreiben einer Funktion durch eine gleichnamige Funktion mit einer anderen Parameterliste möglich ist (dies wird von C++ als Überladen verstanden), tritt bei Parametertypen (Vorbedingungen) nur eine Äquivalenz auf. Jedoch ist darauf zu achten, daß beim Rückgabewert (Nachbedingungen) Kontravarianz ausgeschlossen wird.
- Es wird immer *public*-Vererbung eingesetzt, damit eine Untertypenbeziehung sichergestellt wird. Bei *protected*- oder *private*-Vererbung würde eine Unterklasse eine kleinere Schnittstelle haben als die Basisklasse.
- Mehrfachvererbung wird im beschriebenen Projekt nicht benutzt, da durch den Einsatz von Mehrfachvererbung die Verständlichkeit der Implementierung verringert wird.
- Alle Methoden sind *public* oder *private*, es gibt also keine *protected*-Methoden. *private*-Methoden erlauben eine funktionale Verfeinerung innerhalb einer Klasse, wobei die verfeinerten Methoden nicht nach außen zur Verfügung gestellt werden. Die Verwendung von *protected*-Methoden würde jedoch die Benutzung der Methoden denjenigen Benutzern der Bibliothek gestatten, die eine Wiederverwendung der Struktur durch Vererbung realisieren; den Black-Box-Wiederverwendern hingegen bleiben sie verborgen.
- Alle Attribute sind *private*, es gibt also weder *protected*- noch *public*-Attribute. Die Verwendung von *protected*-Attributen wird aus dem selben Grund wie bei den Methoden vermieden. Auf *public*-Attribute kann die Umgebung der Klasse beliebig zugreifen, was dem Geheimnisprinzip widerspricht. Soll dennoch ein Attribut zur öffentlichen Verwendung bereitgestellt werden, so muß der Zugriff immer via Methodenaufruf erfolgen.
- Gemäß den oben aufgeführten Gründen wird das Schlüsselwort *protected* nicht benutzt.
- Das Schlüsselwort *friend* wird nicht benutzt, weil damit das Konzept des Geheimnisprinzips für bestimmte Dienstnutzer aufgeweicht wird.
- Alle Klassen (außer *ctaObject*) erben von einer anderen Klasse; als Stan-

dard-Basisklasse wird *ctaObject* verwendet.

- Die Destruktoren der Basisklassen werden in C++ automatisch nach dem Destruktor der freizugebenden Klasse aufgerufen. Bei der Verwendung von polymorphen Objektbezeichnern wird die Speicherfreigabe der Objekte von abgeleiteten Klassen durch den virtuellen Destruktor der Klasse *ctaObject* sichergestellt. „Der Destruktor einer Klasse, die von einer Klasse mit einem virtuellen Destruktor abgeleitet ist, ist selbst virtuell“ ([23], S. 156).
- Zur Realisierung des Projektes sollen möglichst nur Werkzeuge und Bibliotheken der GNU-Distribution verwendet werden (Bibliotheken, Compiler, Linker, flex, bison). Insbesondere wird auf die Verwendung der C++-Standardbibliothek nach der ANSI/ISO-Schnittstelle Wert gelegt. Bei der Benutzung dieser Bibliothek wurde die Veröffentlichung zur Implementierung von Silicon Graphics [24] und die Einführung in die C++-Standardbibliothek von Josuttis verwendet [25].
- Alle von Fremdbibliotheken benötigten Klassen werden nur über Adapterklassen verwendet, um einen späteren Austausch der benutzten Klassen zu ermöglichen. Zunächst wird nach dem Entwurfsmuster *Klassenadapter* [26] vorgegangen, wobei jedoch die Schnittstellendefinition *Target* wegen Mehrfachvererbung nicht benutzt werden kann. Die Schnittstelle des vom *Client* benutzten *Adapter* wird durch die Schnittstelle des *Adaptee* festgelegt. Dadurch wird das Klassensystem auf andere Versionen der Fremdbibliotheken und andere Plattformen portierbar. Ist bei der Benutzung einer anderen Bibliothek viel anzupassen, wird gegebenenfalls der *Klassenadapter* in einen *Objektadapter* umgewandelt.
- Alle Klassenbezeichner haben das Präfix *cta*.
- Zu jeder Klasse existieren eine Schnittstellendatei (*.h*) sowie eine Implementierungsdatei (*.cpp*).
- Die Dateien haben den Namen <*klasse*>.<*endung*>.
- Die Namen der Attribute sind so aufgebaut, daß das erste Wort klein geschrieben wird und jedes weitere Wort im Namen mit einem großen Buchstaben beginnt. Bei Methodennamen beginnt auch das erste Wort mit einem großen Buchstaben. Die mit *define*-Anweisungen deklarierten Bezeichner beginnen mit dem String *_cta*.
- Ist für ein Klassenattribut der öffentliche Zugriff bereitzustellen, so wird dazu eine Set- bzw. eine Get-Methode implementiert; der Name ergibt sich aus dem Attributnamen: *Set*<*attribut*> bzw. *Get*<*attribut*>.
- Alle Get-Methoden sind als *const* zu deklarieren, d. h. sie dürfen die Attribute nicht ändern. Weiterhin werden bei allen als *const* deklarierten Methoden alle Komponenten in der Parameterliste als *const* deklariert. Somit entstehen echte seiteneffektfreie Funktionen im mathematischen Sinne.
- In klasseninternen Service-Methoden ist bei lesenden Attributzugriffen

die entsprechende Get-Methode zu benutzen, da durch diese sichergestellt und dokumentiert wird, daß auf das Attribut nur ein Lese-Zugriff erfolgt.

- Alle Bezeichner für Objekte sind Pointer auf die Objekte. Die einzige Ausnahme bilden *Stream*-Objekte. Diese werden nicht durch Pointer, sondern durch direkte Objektbezeichner adressiert, weil sonst durch die Benutzung des Dereferenzierungsoperators die Übersichtlichkeit bei der Anwendung von *Stream*-Objekten mit der Funktion *operator<<()* verloren geht.
- Die Parameter von *return* und *delete* werden einheitlich in Klammern angegeben.
- Vom Gebrauch der *Type-Casts* ist möglichst abzusehen.
- Bei der Programmierung wird die Programmierrichtlinie der Firma Ellemtel verwendet [27].

Bei der Implementierung waren in diesem Projekt keine Abweichungen von den vorliegenden Richtlinien notwendig.

3.2 Definition der Sprache

Für die Definition der Sprache zur Beschreibung eines CTA-Modells ist eine kontextsensitive Grammatik erforderlich. Da ein effizienter Parser für eine kontextsensitive Grammatik nur schwer entwickelt werden kann, wird die Syntax der Sprache mittels einer kontextfreien Grammatik definiert. Die Kontextbedingungen werden getrennt davon beschrieben und in einem separaten Durchlauf durch den Syntaxbaum überprüft.

Im folgenden wird die Sprache anhand der Grammatik in einer um reguläre Ausdrücke erweiterten Backus-Naur-Form beschrieben [28]. Reguläre Ausdrücke werden in der bei Unix-Werkzeugen üblichen Syntax[1] notiert. Die kontextabhängigen oder schwer in die Grammatik einzubringenden Bedingungen werden jeweils unterhalb der betreffenden Regel der Grammatik verbal beschrieben. Die Kontextbedingungen ergeben sich aus der formalen Beschreibung der CTA-Modelle sowie aus der Notation [18]. Die Semantik wurde im Kapitel 2 beschrieben. Bei der angegebenen Grammatik handelt es sich nicht um die Nutzergrammatik, sondern um die vom Parsergenerator verwendbare Implementierungsgrammatik.

Terminalsymbole werden in Großbuchstaben oder Sonderzeichen, zusätzlich von Hochkommata eingeschlossen, notiert; Metasymbole hingegen mit einem Großbuchstaben am Anfang eines Wortes und von den Zeichen

1. [] ... mindestens ein Zeichen aus einer Alternative; ? ... 0 oder 1 Auftreten eines Ausdrucks; + ... mindestens ein Auftreten des Ausdruckes; * ... Ausdruck kann beliebig oft auftreten

< und > umgeben. Da der zu entwickelnde Parser nach dem Left-to-right-bottom-up-Prinzip arbeiten soll, wird eine linksrekursive Grammatik benutzt.

Modulaufbau.

<System>	*::=*	*<Module>*
	|	*<System> <Module>*

Ein CTA-Modell besteht aus einer nichtleeren Menge von CTA-Modulen.

<Modul>	*::=*	*'MODULE' <Identifier> '{' <ModuleContent> '}'*

Durch das zweite Metasymbol wird ein eindeutiger Name zur Identifizierung des Moduls angegeben.

<ModuleContent>	*::=*	ε
	|	*<ModuleContent> <Interface>*
	|	*<ModuleContent> <Instantiation>*
	|	*<ModuleContent> <Initial>*
	|	*<ModuleContent> <Automaton>*

Die Modulinhalte können in beliebiger Reihenfolge und beliebig oft angegeben werden. Ein *Interface* muß immer deklariert werden. Wird ein Automat angegeben, so ist auch eine *Initial*-Konfiguration notwendig, um den Startzustand festzulegen.

Deklarationen.

<Interface>	*::=*	*'INPUT' <DeclList>*
	|	*'OUTPUT' <DeclList>*
	|	*'MULTREST' <DeclList>*
	|	*'LOCAL' <LocalDeclList>*

Wird eine Komponente als *LOCAL* deklariert, kommt eine Konstantendeklaration mit Initalisierung in Frage.

Schnittstellenkomponenten können bei der Instanziierung den Komponenten des enthaltenden Moduls zugeordnet werden bzw. unbenutzt bleiben.

Als *INPUT* deklarierte Komponenten dürfen innerhalb des Moduls nicht eingeschränkt werden, was eine Garantie an die Umgebung darstellt.

Wurde eine Schnittstellenkomponente als *OUTPUT* deklariert, so darf diese nur innerhalb des Moduls eingeschränkt werden. Der Kontext des Moduls darf diesen Bezeichner nur als *INPUT* benutzen. Dies stellt eine Anforderung des Moduls an die Umgebung dar. Daraus ergeben sich folgende konkrete Bedingungen innerhalb eines Moduls:

- Wurde eine Komponente einmal als *OUTPUT* benutzt, darf sie in keinem anderen Automaten und keiner anderen Instanz als *OUTPUT* oder *MULTREST* benutzt werden.

- Umgekehrt gilt: Wurde eine Komponente einmal als *MULTREST* benutzt, so scheidet die Benutzung als *OUTPUT* in allen anderen Automaten und Instanzen aus.

DeclList ::= ε
| *<DeclList> <Identifier> ':' Type ';'*

Der Name für eine Komponente darf nur ein einziges Mal vergeben werden.

LocalDeclList ::= ε
| *<LocalDeclList> <Identifier> ':'*
<TypeWithoutConst> ';'
| *<LocalDeclList> <Identifier> ,='*
<LinearExpression> ':' 'CONST' ';'

Wie bei der Regel *DeclList* darf auch hier der Name für eine Komponente nur ein einziges Mal vergeben werden. Wird eine Konstante als *LOCAL* deklariert, so muß auch eine Initialisierung mit einem Wert erfolgen.

Type ::= *'CONST'*
| *<TypeWithoutConst>*
TypeWithoutConst ::= *'DISCRETE'*
| *'CLOCK'*
| *'STOPWATCH'*
| *'ANALOG'*
| *'SYNC'*

Wie auch in den Abschnitten 2.2.2 und 2.2.3 beschrieben, ist bei den Datentypen zunächst zwischen diskreten und kontinuierlichen Komponenten zu unterscheiden. Bezeichner für diskrete Komponenten, also Signale bzw. Synchronisationsmarken, werden durch das Schlüsselwort *SYNC* deklariert. Die Bezeichner für kontinuierliche Komponenten unterscheidet man in der Ableitung nach der Zeit. Als *ANALOG* deklarierte Variablen können beliebige Einschränkungen der Ableitungen haben. Bei den *STOPWATCH*-Variablen sind nur die Werte null und eins der Ableitungen erlaubt. Eine *CLOCK*-Variable muß eine unveränderliche Ableitung mit dem Wert eins haben und bei einer *DISCRETE*-Variable darf der Wert der Ableitung nur null sein; dies entspricht einer normalen Variablen in einer sequentiellen Programmiersprache. Ein *CONST*-Bezeichner darf nur eine Ableitung vom Wert null besitzen, darüber hinaus ist der Wert selbst nicht veränderbar. *CONST*-Bezeichner müssen als *LOCAL* oder *INPUT* deklariert werden.

Jedes deklarierte Signal muß von mindestens einem im Modul enthaltenen Automaten benutzt werden. Unbenutzte Signale deuten auf Modellierungsfehler hin und sind deshalb nicht erlaubt. Das Alphabet der einem Automaten bekannten Signale ergibt sich aus allen in den Transitionen benutzten Signalen.

Instanziierungen.

Instantiation ::= *'INST' <Identifier> 'FROM' <Identifier>*
'WITH' '{' <Connections> '}'

Der erste Bezeichner gibt den Namen der zu erzeugenden Instanz an. Er darf noch nicht für ein anderes Modul vergeben worden sein. Der zweite Bezeichner bestimmt das Modul, welches bei der Instanziierung als Schablone gilt. Es muß ein Modul mit diesem Namen definiert worden sein.

Durch das Metasymbol *Connections* erfolgt die Produktion der Identifizierungsrelation $ident \subseteq C' \times C$ mit C' = Menge der in der Schablone deklarierten Komponenten und C = Menge der im enthaltenden Modul deklarierten Komponenten [18]. Jeder Komponente der Schablone darf nur einmal eine Komponente des enthaltenden Moduls zugeordnet werden:

- *ident* muß eine Funktion sein.

Jeder Komponente der Schablone darf höchstens eine Komponente des enthaltenden Moduls zugeordnet werden:

- *ident* muß injektiv sein.

In der Enthaltenseinsbeziehung der Moduldefinitionen darf kein Zyklus auftreten. Die Beziehung wird durch die Enthaltenseinsrelation E beschrieben:

- E^+ muß irreflexiv sein ($\forall(n \in M)$:(n,n) $\notin E^+$, M = Menge der definierten Module).

Connections ::= ε
| *<Connections> <Identifier> 'AS' <Identifier> ';'*

In dieser Regel werden den Schnittstellenkomponenten der Modulinstanz (erster Bezeichner) die entsprechenden Komponenten des enthaltenden Moduls zugeordnet. Schnittstellenkomponenten sind Komponenten, die nicht als *LOCAL* deklariert worden sind:

- $dom(ident) \subseteq C' - C'L$, C' = Menge der deklarierten Komponenten, $C'L$ = Menge der LOCAL-Komponenten mit $C'L \subseteq C'$.

Der Bezeichner auf der linken Seite muß im Schablonenmodul und der Bezeichner auf der rechten Seite im enthaltenden Modul deklariert worden sein. Beide Bezeichner müssen im Datentyp übereinstimmen.

Die Zuordnung darf nur unter Beachtung einer verträglichen Benutzung der Restriktionstypen erfolgen:

- $x \in C'O \rightarrow ident(x) \in CO \cup CL$, $C'O$ = *OUTPUT*-Komponenten mit $C'O \subseteq C'$, CO = *OUTPUT*-Komponenten mit $CO \subseteq C$, CL = *LOCAL*-Komponenten mit $CL \subseteq C$,
- $x \in C'M \rightarrow ident(x) \notin CI$, $C'M$ = *MULTREST*-Komponenten mit $C'M \subseteq C'$, CI = *MULTREST*-Komponenten mit $CI \subseteq C$,
- $x \in C'I \rightarrow ident(x) \in C$, $C'I$ = *INPUT*-Komponenten mit $C'I \subseteq C'$,

Tabelle 3.1: Erlaubte Zuordnungen in den Instanziierungen

Im Schablonenmodul deklariert als:	Im enthaltenden Modul deklariert als:
LOCAL	---
MULTREST	LOCAL, MULTREST, OUTPUT
OUTPUT	LOCAL, OUTPUT
INPUT	LOCAL, MULTREST, OUTPUT, INPUT

- seien I' und I'' zwei Instanzen und I.M das Schablonenmodul, von dem eine Instanz I instanziiert wurde, dann muß gelten:

  $(x \in \text{I'.M.CO} \wedge y \in \text{I''.M.C} \wedge \text{I'.ident(x)} = \text{I''.ident(y)}) \rightarrow y \in \text{I''.M.CI}$

Aus diesen Bedingungen ergibt sich Tab. 3.1, wobei alle nicht in der Tabelle enthaltenen Zuordnungen verboten sind.

Initialisierungen.

<Initial> *::=* *'INITIALIZATION' '{' <Configuration> '}'*

In einer Konfiguration werden die Anfangswerte der Variablen und die Startzustände bestimmt.

<Configuration> *::=* *'STATE' '=' <Identifier> <IdentList> ';'*
<LinearRestriction>

Der Bezeichner und alle Bezeichner in der nachfolgenden Bezeichnerliste müssen Namen von in den lokalen Automaten definierten Zuständen sein. Da von allen lokalen Automaten der gleiche Namensraum benutzt wird, ist die Angabe des Automaten, in dem die Zustände definiert sind, nicht notwendig. Kein Bezeichner, der in der Regel *LinearRestriction* einschränkend benutzt wird, darf als *INPUT* deklariert worden sein.

Automaten.

<Automaton> *::=* *'AUTOMATON' '{' <States> '}'*

Jeder Automat enthält eine Menge von Zuständen.

<States> *::=* ε
| *<States> 'STATE' <Identifier>*
'{' <Invariant> <Derivation> <Transitions> '}'

Der Bezeichner gibt den eindeutigen Namen des Zustandes an. Der Name darf noch nicht für einen anderen Zustand in einem Automaten verwendet worden sein, der im gleichen Modul enthalten ist, da alle Automaten eines Moduls den gleichen Namensraum benutzen.

<Invariant> *::=* ε
| *'INV' '{' <LinearRestriction> '}'*

Tabelle 3.2: Signalbenutzung in einem Automaten

Deklaration im Modul	erlaubte Benutzung im Automaten
LOCAL	?, !, #
INPUT	?
OUTPUT	?, !, #
MULTREST	?, #

<Derivation> *::=* ε
| *'DERIV' '{' <DerRestriction> '}'*
<DerRestriction> *::=* ε
| *<DerRestriction> 'DER' '(' <Identifier> ')'*
<RateRelOp> <LinearExpression> ';'

Der Bezeichner muß vom Typ *ANALOG* oder *STOPWATCH* sein und darf nicht als *INPUT* deklariert worden sein. Im linearen Ausdruck der Regel *LinearExpression* dürfen nur konstante Werte und Konstantenbezeichner verwendet werden.

<Transitions> *::=* ε
| *<Transitions> 'TRANS' <Identifier> <IdentList>*
'{' <Guard> <Sync> <Do> '}'

Alle als Namen für Zustände verwendeten Bezeichner müssen auch Namen von Zuständen in dem diese Transition enthaltenden Automat sein.

<IdentList> *::=* ε
| *<IdentList> '|' <Identifier>*
<Guard> *::=* ε
| *'GUARD' '{' <LinearRestriction> '}'*
<Sync> *::=* ε
| *'SYNC' <SignalType> <Identifier> <SyncThen> ';'*

Der Bezeichner muß vom Typ *SYNC* sein und mit dem Benutzungszeichen der Regel *SignalType* entsprechend Tab. 3.2 kompatibel sein.

Wird in einem Automaten ein Signal mit dem Zeichen *'?'* benutzt, so ist dieses Signal für diesen Automaten ein Eingabesignal. Das bedeutet, daß in allen Zuständen jederzeit die Auswahl einer Transition mit diesem Signal möglich sein muß. Diese Bedingung wird gegebenenfalls durch die automatische Vervollständigung mit Übergängen in einen Fehlerzustand erfüllt.

Aus der Benutzung eines Signals mit dem Zeichen *'!'* folgt die Benutzung im Automaten als Ausgabesignal. Dies impliziert die Bedingung, daß alle anderen Automaten und Modulinstanzen dieses Signal nicht einschränken dürfen, es also für diese ein Eingabesignal darstellt.

<SyncThen> *::=* ε
| *'THEN' <SignalType> <Identifier>*

Hier gelten die gleichen Kontextbedingungen wie in der Regel *Sync*.

<SignalType> *::=* *'?'*
| *'!'*
| *'#'*
<Do> *::=* ε
| *'DO' '{' <LinearConstraint> '}'*
<LinearConstraint> *::=* ε
| *<LinearConstraint> <Identifier>*
'ASSIGN' <LinearExpression> ';'

Der benutzte Bezeichner muß deklariert worden sein. Es darf sich dabei nicht um eine *INPUT*-Variable handeln. Der Typ darf nicht *SYNC* und *CONST* sein.

Variableneinschränkungen und Ausdrücke.

<LinearRestriction> *::=* ε
| *<LinearRestriction> <Identifier>*
<RelOp> <LinearExpression> ';'

Der in dieser Regel verwendete Bezeichner muß deklariert und darf nicht vom Typ *SYNC* sein.

<LinearExpression> *::=* *<LinearTerm>*
| *<ChooseExpression>*
| *<LinearExpression> <LinOp> <LinearTerm>*
<LinearTerm> *::=* *<PosLinearTerm>*
| *'+' <PosLinearTerm>*
| *'-' <PosLinearTerm>*
<PosLinearTerm> *::=* *<Constant>*
| *<Identifier>*

Der Bezeichner muß deklariert worden sein, darf jedoch nie vom Typ SYNC sein.

<Constant> *::=* *<FloatingConstant>*
| *<IntegerConstant>*
<ChooseExpression> *::=* *'CHOOSE' '(' '[' <LinearExpression> ','*
<LinearExpression> ']' ')'

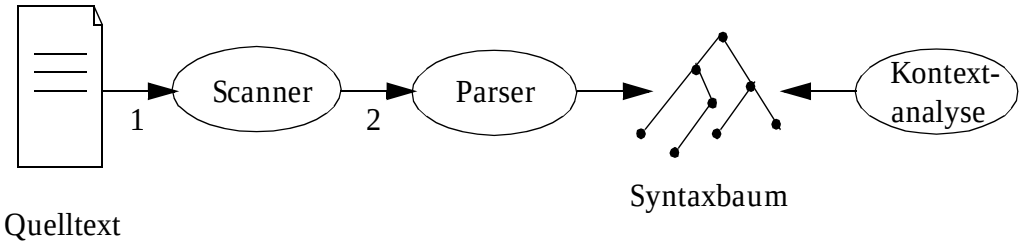

Abbildung 3.1: Compiler-Frontend

```
<RelOp>          ::=  '='
                  |   '<'
                  |   '>'
                  |   'LE'
                  |   'GE'
<RateRelOp>      ::=  '='
                  |   'LE'
                  |   'GE'
<LinOp>          ::=  '+'
                  |   '-'
<Identifier>     ::=  [a-zA-Z][0-9a-zA-Z]*
```

Der Unterstrich in Bezeichnern ist nicht erlaubt, wird jedoch für die interne Verwendung benötigt und daher vom Parser akzeptiert.

```
<FloatingConstant>   ::=  ( ([0-9]*“.“[0-9]+) | ([0-9]+“.“) | ([0-9]+) )
                          ([eE][-+]?[0-9]+)?
<IntegerConstant>    ::=  [0-9]+
```

Erlaubt sind beliebig lange Dezimalbrüche und ganzzahlige Werte.

3.3 Beschreibung der Compilerbestandteile

Alle Programmteile, die zur internen Repräsentation der CTA-Modelle notwendig sind, werden in den folgenden Abschnitten beschrieben. Die Abb. 3.1 stellt die Zusammenarbeit von Scanner, Parser und Kontextanalyse dar: Der Quelltext wird vom Scanner zeichenweise aus der Eingabedatei eingelesen und in Morpheme aufgeteilt, die dann vom Parser gemäß der Grammatik auf syntaktische Korrektheit überprüft und zum Syntaxbaum zusammengesetzt werden. Die Kontextanalyse schließlich überprüft die Kontextbedingungen im CTA-Modell, die durch eine kontextfreie Grammatik nicht beschrieben werden können.

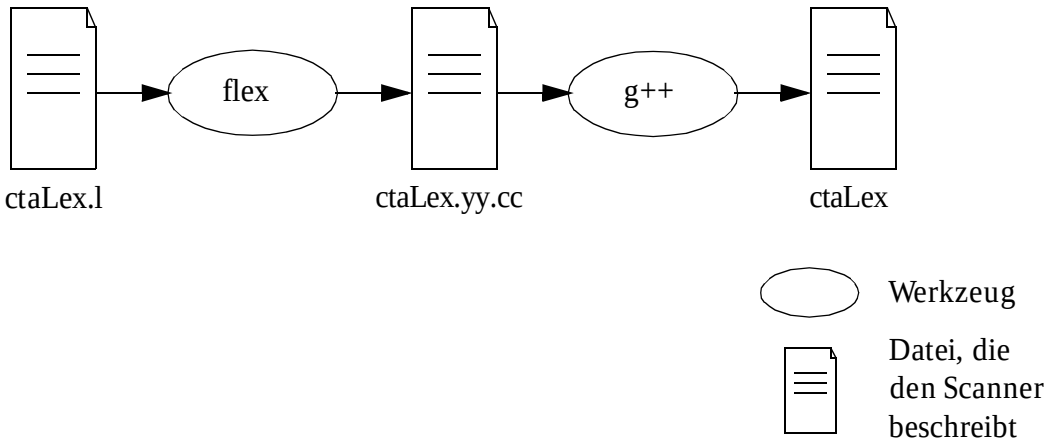

Abbildung 3.2: Erzeugung des Scanners

Um die geforderte hohe Flexibilität bei Erweiterungen zu erreichen, wird in der Entwurfsphase die objektorientierte Modellierung benutzt. Implementiert werden sämliche Projektteile in der objektorientierten Programmiersprache C++.

3.3.1 Lexikalische Analyse

Zur Entwicklung des Scanners wurde der in der GNU-Distribution mitgelieferte Scannergenerator *flex* benutzt [29]. Der Scanner wird vom Scannergenerator aus einer Eingabedatei erzeugt, die den Scanautomaten beschreibt. Die Abb. 3.2 stellt den Ablauf bei der Arbeit mit *flex* dar. Dabei werden die Morpheme, in die der Scanner den Eingabestrom aufteilen soll, als reguläre Ausdrücke angegeben und die jeweils auszuführenden Aktionen beschrieben. Von *flex* wird nun eine *C++*-Datei generiert, woraus mit einem *C++*-Compiler ein ausführbares Programm erzeugt wird. Die Scanner-Definition befindet sich im Anhang C.

In der GNU-Distribution sind verschiedene Klassen zur Einbindung des Scanners enthalten. Diese werden in Abb. 3.3 veranschaulicht. Die Klasse *FlexLexer* bietet eine Schnittstelle für einen Scanner an, welche durch die Klasse *yyFlexLexer* verfeinert wird. Dadurch werden alle von *flex* zur Verfügung gestellten Komponenten verkapselt. Die Klasse *ctaScanner* realisiert eine Adapterklasse, so daß beim Einsatz eines anderen Scanners nur die Adapterklasse angepaßt werden muß. Zusätzlich stellt die Klasse *ctaScanner* einen Konstruktor zur Verfügung, dem der bereits initialisierte Dateistrom des Quelltextes als Parameter übergeben wird.

Vom Parser kann somit ein Scannerobjekt erzeugt werden, welches per Funktionsaufruf *yylex()* das nächste Morphem bzw. mit *YYText()* die zuletzt gescannte Zeichenkette an den Parser liefert.

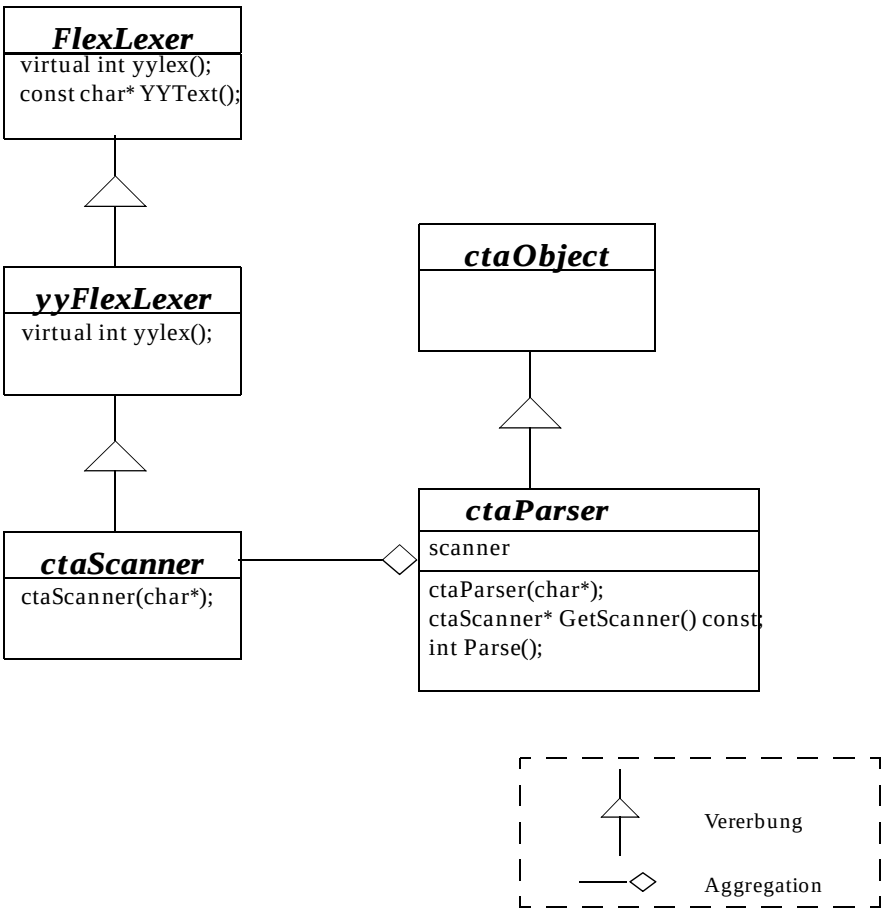

Abbildung 3.3: Klassendiagramm für Scanner und Parser

3.3.2 Syntaktische Analyse

Zur Erzeugung des Parsers wird der Parsergenerator *bison* aus der GNU-Distribution verwendet [30]. Der Parsergenerator erhält (ähnlich dem Scanner in Abb. 3.2) als Eingabe eine Datei (*ctaYacc.y*, siehe Anhang C), in der einerseits eine LALR(1)-Grammatik notiert ist und in der sich andererseits für jede Regel der Grammatik ein Anweisungsblock zum Aufbau des abstrakten Syntaxbaumes sowie Teile der Kontextanalyse (Abschnitt 3.3.5) befinden.

Die kontextfreie Grammatik, die die Syntax einer CTA-Modellierung definiert, wurde im Abschnitt 3.2 vorgestellt. Da *bison* einen Shift-Reduce-Parser (Bottom-Up-Syntaxanalyse) generiert, wurde eine linksrekursive Grammatik entwickelt. Ein Shift-Reduce-Parser legt die eingelesenen Token bzw. schon reduzierten Metasymbole auf den Parser-Stack. Erst wenn eine rechte Seite einer Regel der Grammatik mit den *n* oberen Stack-Elementen übereinstimmt, erfolgt ein Reduce-Schritt [31]. Bei einer rechtsrekursiven Gramatik hätte ein Shift-Reduce-Parser einen sehr großen

Stack-Bedarf, weil ein Reduce-Schritt erst sehr spät bei der letztmaligen Anwendung der rechtsrekursiven Produktionsregel angewendet werden kann.

Zur Kapselung der von *bison* bereitgestellten Komponenten und zur Bereitstellung einer Schnittstelle wird die Klasse *ctaParser* eingeführt, welche ein Objekt der Klasse *ctaScanner* enthält. Durch die Methode *Parse()* wird das Parsen gestartet, und anhand der Methode *GetScanner()* kann auf den Scanner zugegriffen werden.

bison bietet die Möglichkeit des *Error-Recovery*, wodurch nach einem Syntaxfehler unter Ausgabe von Fehlermeldungen fortgefahren werden kann. Dazu ist es notwendig, eine zusätzliche Alternative mit dem Metasymbol *error* einzuführen, die mit einem Symbol endet, dessen Auftreten das Ende des aus dem aktuellen Metasymbol produzierbaren Textes sicherstellt. So kann das aktuelle Metasymbol reduziert und so fortgefahren werden, als ob es den Fehler nicht gab. In diesem Projekt wird das Error-Recovery nur beim Deklarationsteil zur Demonstration der Möglichkeiten angewendet.

In allen Syntaxbaumknoten, in denen Pointer auf Mengen existieren, soll aus technischen Gründen wenigstens eine leere Menge erzeugt werden; der Pointer darf nach dem Parsen also nie auf *NULL* gesetzt oder undefiniert sein.

Der Parser besteht aufgrund der Benutzung des Parsergenerators aus einer Sammlung von globalen Komponenten. Das hat den Nachteil, daß für alle Syntaxbaumknoten Methoden zum Setzen der Inhalte definiert werden mußten. Eine Alternative dazu war, die Klasse, die den Parser implementiert, bei allen Syntaxbaumknoten als *friend* einzutragen. Dies war hier jedoch nicht möglich, da die Parserfunktionen nicht in einer Klasse implementiert wurden, sondern lediglich eine Schnittstelle bereitgestellt wurde.

Eine Struktur zur Speicherung der Namen - die Symboltabelle - wurde in diesem Projekt nicht implementiert, sondern es wurde auf die Datenstruktur *Abbildung* aus der C++-Standardbibliothek zurückgegriffen. Es wurde die Implementierung der Abbildung als balancierten Binärbaum verwendet. Die Zeitkomplexität beim Finden eines Namens in dieser Datenstruktur ist *O(log n)*. Die Implementierung der Abbildung als *Hashtabelle* hat bei entsprechend guter *Hash*-Funktion eine noch bessere Zeitkomplexität. Da die Dokumentation zu diesem Teil der C++-Standardbibliothek zum Zeitpunkt der Entwicklung des Parsers noch nicht vorlag, konnte diese Struktur nicht verwendet werden. Die Flexibilisierung durch des Benutzen der Adapterklasse *ctaMap* ermöglicht jedoch jederzeit den nachträglichen Wechsel zu einer besseren Implementierung der Abbildung.

3.3.3 Abstrakter Syntaxbaum

Der abstrakte Syntaxbaum dient zur internen Repräsentation der Aufbaustruktur des CTA-Modells. Die notwendigen Komponenten zum Aufbau des abstrakten Syntaxbaumes werden hier auf Entwurfsebene beschrieben.

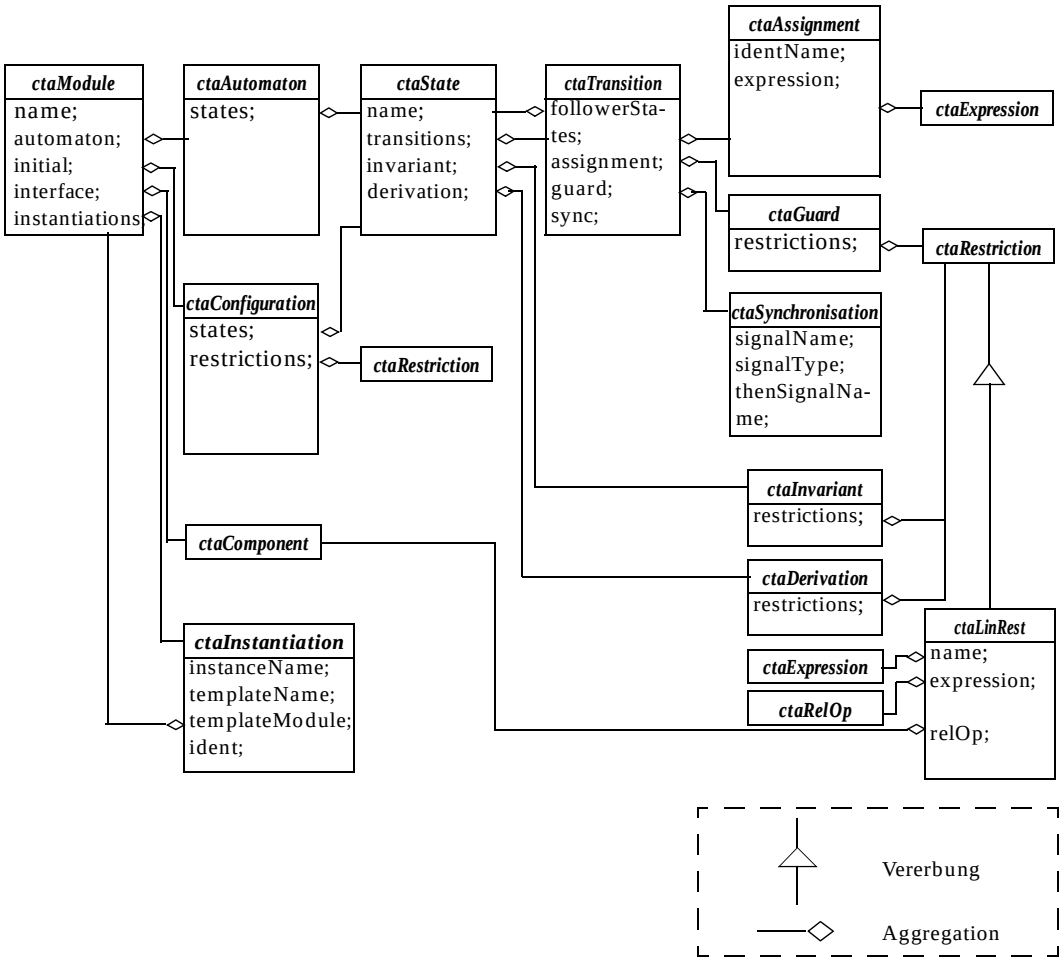

Abbildung 3.4: Klassendiagramm für Modulbestandteile

Die interne Repräsentation benötigt den fünffachen Speicherplatzbedarf gegenüber der textuellen Repäsentation, d. h. eine CTA-Modellierung in einer 17 KB-großen Textdatei entspricht einem Syntaxbaum der Größe von 85 KB. Beim Einlesen der Beispielmodellierung des Förderbandes wurden ca. 350 Objekte erzeugt.

Bei allen Syntaxbaumklassen werden für alle Attribute Zugriffsmethoden bereitgestellt. Diese Entscheidung liegt darin begründet, daß sowohl die *stream*-Klasse, die zur Ausgabe benutzt wird, als auch die Klassen, die bei der Kontextanalyse Informationen über die Syntaxbaumknoten benötigen, auf die Inhalte lesend zugreifen. Der Parser muß beim Setzen der Inhalte der Syntaxbaumknoten über die Zugriffsmethoden schreibend auf die Attribute der Klassen zugreifen.

Modulbestandteile. Die in Abschnitt 2.1 beschriebenen Komponenten eines CTA-Moduls finden sich im Klassendiagramm in Abb. 3.4 wieder. Die Komponenten werden durch Klassen modelliert, die sich im wesentlichen mit den Metasymbolen der Grammatik decken, so daß bei (fast)

jeder Regel ein dem Syntaxbaumknoten entsprechendes Objekt angelegt wird. Dabei werden die auf dem Parser-Stack abgelegten Objekte durch die Set-Methoden des behandelten Objektes als Komponenten eingetragen.

Die Variableneinschränkungen, die in den Klassen *ctaGuard*, *ctaInvariant* und *ctaDerivation* benötigt werden, sind in der Klasse *ctaLinRest* enthalten. Um bei späteren Erweiterungen auf nichtlineare Einschränkungen reagieren zu können, werden von den oben genannten Klassen jeweils Pointer auf die Schnittstellenklasse *ctaRestriction* genutzt, welche durch die Klasse *ctaLinRest* implementiert wird (Polymorphie).

Die Klasse *ctaInstantiation* speichert alle Informationen, die für die Instantiierung notwendig sind. Dies sind zum einen der Name der neuen Instanz und der Name des Schablonenmoduls. Der Pointer auf das Modulobjekt der Schablone kann erst im zweiten Durchlauf gesetzt werden. Die Zuordnungsfunktion *ident* bildet die Namen der Komponenten im Schablonenmodul auf Komponenten im enthaltenden Modul ab.

Alle Bestandteile mit einem eindeutigen Namen (also Module, Zustände, Instanziierungen und die Identifizierungsfunktion) werden in der Datenstruktur *Abbildung (map)* abgelegt. Die Einträge erfolgen, indem der Name als Schlüssel und ein Pointer auf das zugehörige Objekt als Wert eingetragen wird.

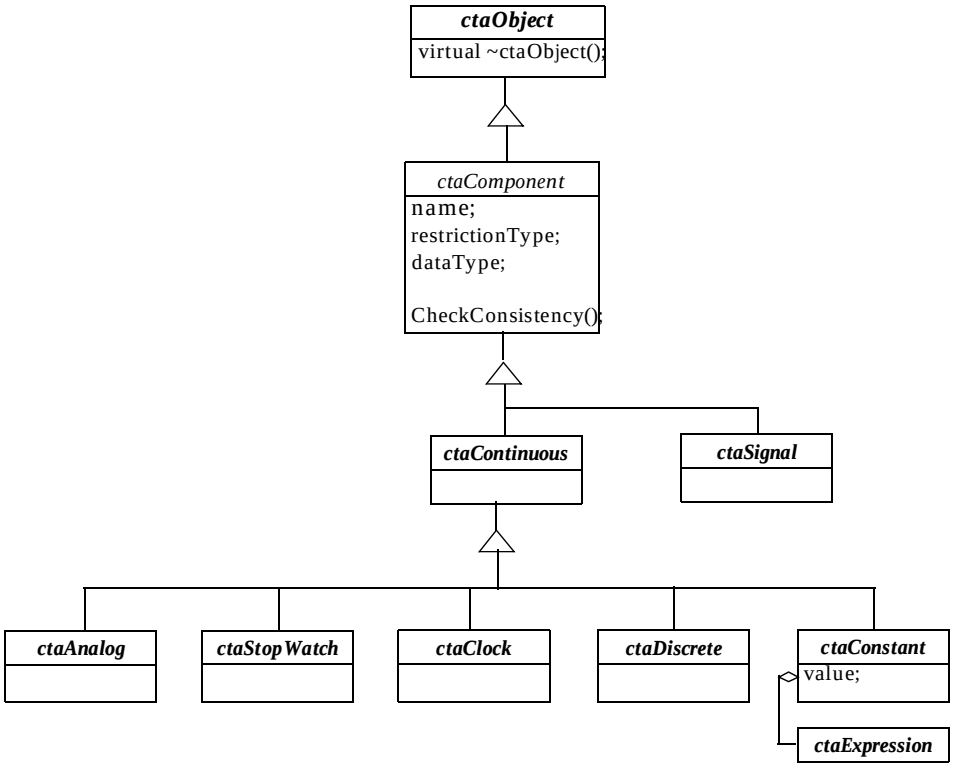

Abbildung 3.5: Klassendiagramm für Kommunikationskomponenten

Kommunikationskomponenten. Der Entwurf der Variablen- und Synchronisationskomponenten ist in Abb. 3.5 dargestellt. Eine Komponente kann entweder eine Synchronisationskomponente (*ctaSignal*) oder eine kontinuierliche Komponente (*ctaContinuous*) sein.

Wie in Abschnitt 2.2.2 beschrieben, unterscheiden sich die kontinuierlichen Komponenten nach den möglichen Werten der Ableitungen in analoge, Stoppuhr-, Uhr-, diskrete und konstante Komponenten, welche durch von *ctaContinuous* abgeleitete Klassen modelliert werden. Der bei lokalen Konstanten zu speichernde Initialisierungswert ist im Attribut *value* der Klasse *ctaConstant* abgelegt.

In jedem Modul werden sämtliche Kommunikationskomponenten in einem Container als Objekte der Klasse *ctaComponent* abgelegt. Um eine hohe Flexibilität, vor allem in Hinblick auf die Zuordnung bei der Modulinstanziierung, zu erreichen, wird als Container eine Abbildung benutzt. Bei dieser Abbildung bildet die Menge der Komponentennamen den Definitionsbereich (also die Schlüssel in der Datenstruktur) und die Menge der Komponentenobjekte (bzw. Pointer darauf) den Wertebereich.

Ausdrücke und Operatoren. Ausdrücke werden durch Objekte der Klasse *ctaExpression* repräsentiert. Um einen flexiblen Entwurf zu erhalten, wird das Entwurfsmuster *Composite* verwendet [26]. Wie in Abb. 3.6 dargestellt, entspricht dabei die Klasse *ctaExpression* der Klasse *Component* im Muster, *ctaTerm* der Klasse *Leaf*, und die Klassen *ctaLinExpr* und *ctaChooseExpr* ent-

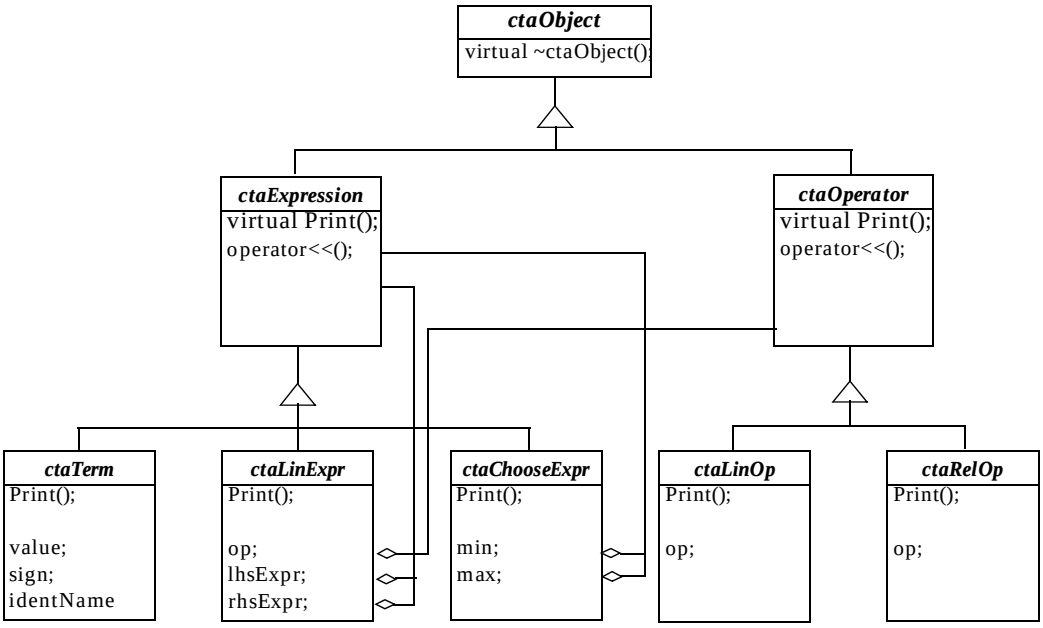

Abbildung 3.6: Klassendiagramm für Ausdrücke und Operatoren

sprechen der Klasse *Composite*. Durch die Verwendung dieses Entwurfsmusters wird eine sehr hohe Flexibilität in Bezug auf Erweiterbarkeit des Klassenteilsystems erreicht, so daß jederzeit weitere notwendige Ausdrucksformen (Klammerung, boolesche Ausdrücke, etc.) hinzukommen können.

Durch ein Objekt der Klasse *ctaTerm* wird entweder ein Bezeichner oder ein konstanter Wert modelliert. Ob es sich bei dem Term um einen Bezeichner oder um einen Wert handelt, wird anhand des Attributs zur Beschreibung des Namens entschieden: Ist der String (*identName*) leer, so handelt es sich um eine Konstante, deren Wert im Attribut *value* enthalten ist, sonst ist in *identName* der Bezeichner enthalten.

Die Ausdrücke werden immer in der ausführlichen Struktur im Speicher gehalten, also nicht ausgerechnet. Dadurch ist zwar ein größerer Aufwand bei Entwurf und Implementierung notwendig, aber nur so kann der ursprüngliche Ausdruck von der Ausgabe-Funktion bereitgestellt und eine Erweiterung der Modellierungsmöglichkeiten beim CTA-Konzept einfach eingebracht werden.

Service-Klassen. Für die Aufbewahrung einer Menge von Objekten werden die Container-Klassen *ctaSet* und *ctaMap* bereitgestellt. Um auf spätere Veränderungen der benutzten Bibliotheken flexibel reagieren zu können, wurden diese Klassen von der Klasse *set* bzw. von der Klasse *map* aus der

C++-Standardbibliothek nach dem Vorbild eines Klassenadapters abgeleitet. Sollte später eine Veränderung notwendig sein, kann diese Struktur gegebenenfalls in einen Objektadapter umgewandelt werden [26].

Zur Repräsentation von Zeichenketten dient die Klasse *ctaString*. Diese wurde wegen der gleichen Argumentation wie bei *ctaSet* von der Klasse *string* der C++-Standardbibliothek abgeleitet.

In der C++-Standardbibliothek sind die Implementierungen der Datenstrukturen von der Implementierung der Algorithmen, die mit den Datenstrukturen arbeiten, getrennt. Dazu wurde im Gegensatz zur Idee des Entwurfsmusters *Strategy*, welches eine dynamische Zuordnung von verschiedenen Algorithmen zu einer Datenstruktur ermöglicht [32], nach dem *Template*-Konzept vorgegangen.

Bei der benutzten Datenstruktur wird nicht nur der zu benutzende Datentyp durch ein *Template* parametrisiert (z. B. int, double, char), sondern allgemeine Algorithmen über die Struktur (z. B. Sortieren, Mengenoperationen) werden ebenfalls durch Templates flexibel gehalten. Zu jeder Datenstruktur existiert ein spezieller Iteratortyp, der als Positionszeiger für die entsprechende Datenstruktur dient. Durch die Benutzung dieser Iteratoren ist die Implementierung der Algorithmen in der Bibliothek nur einmal notwendig.

3.3.4 Ausgabe des Syntaxbaumes

Zur Ausgabe der internen Repräsentation des CTA-Modells besitzt jeder Syntaxbaumknoten eine Ausgabemethode. Diese Ausgabe, welche die Möglichkeit der Erweiterung um eine wohlformatierte Ausgabe zu einem Pretty-Printer bietet, ist besonders zu Testzwecken während der Implementierungsarbeiten sehr nützlich.

Weil die Funktion *operator<<()* nicht als *virtual* deklariert werden kann, müssen Klassen, deren Objekte unter Anwendung von Polymorphie benutzt werden, die Ausgabe in der Methode *Print()* bereitstellen. In der Funktion *operator<<()* der Basisklasse wird die Methode *Print()* mit dem zu benutzenden Ausgabe-Stream als Parameter aufgerufen. Dieses Vorgehen ist speziell bei den Klassen *ctaTerm*, *ctaLinExpr*, *ctaBraceExpr* und *ctaChooseExpr* durch den Einsatz des Kompositor-Musters notwendig geworden.

Eine Alternative zu dieser Technik bestand in der konsequenten Implementierung der Ausgabe als *Print()*-Funktion in allen Syntaxbaumknoten unter entsprechender Angabe des Ausgabestromes als Parameter. Diese Variante wurde jedoch nicht angewendet, da der in C++ gebräuchliche Ausgabemechanismus an einen Ausgabestrom mit der Methode *operator<<()* der Klasse *ostream* bereitgestellt werden sollte.

Bei der Ausgabe der Namen von Variablen, Signalen, Konstanten und Zuständen wird zunächst die entsprechende Komponente in der Deklaration (*Interface* bzw. *Automat*) gesucht, wobei der Name als Schlüssel dient.

Danach wird der in dieser Komponente gespeicherte Name ausgegeben. Der Vorteil liegt in einer hohen Flexibilität bei der Ausgabe von absoluten Namen für die benutzten Komponenten, bei denen ein Präfix benutzt werden muß, welches nur in den Objekten der Komponentendefinitionen abgespeichert ist.

3.3.5 Kontextanalyse

Die Kontextanalyse dient der Überprüfung aller kontextabhängigen Bedingungen für CTA-Modelle. Im folgenden wird darauf eingegangen, wie die im Abschnitt 3.2 aufgeführten Bedingungen implementiert wurden.

Einerseits gibt es Kontextbedingungen, die sofort beim Aufbau des Syntaxbaumes überprüft werden können. Dazu zählt z. B. die Bedingung, daß kein Bezeichner doppelt deklariert werden darf. Solche Kontextüberprüfungen werden im ersten Durchlauf von den in der Grammatik enthaltenen Programmteilen angestoßen.

Andererseits existieren Bedingungen, die eine vollständige Syntaxanalyse der gesamten Eingabedatei voraussetzen. Zum Beispiel kann beim Parsen einer Transition noch keine Aussage darüber getroffen werden, ob der angegebene Folgezustand im noch zu verarbeitenden CTA-Text definiert wird oder ob es sich um einen undefinierten Bezeichner handelt. Aus diesem Grund ist ein zweiter Durchlauf durch den Syntaxbaum notwendig, bei dem die noch offenen Kontextbedingungen überprüft werden. Diese Traversierung erfolgt durch den rekursiven Aufruf der Methode *CheckConsistency()*, die in den Syntaxbaumknoten enthalten ist und den zweiten Teil der Kontextanalyse implementiert. Diese Vorgehensweise hat den Vorteil, daß die Kontextüberprüfung lediglich in den Modulen aufgerufen werden muß und alle enthaltenen Syntaxbaumknoten automatisch ebenfalls die Kontextüberprüfung aufrufen.

Bei der Deklaration von Bezeichnern ist zu beachten, daß kein Bezeichner wiederholt benutzt werden darf. Dieser Bedingung, die bei Syntaxbaumknoten zur Definition von Modulen, Zuständen, Variablen-, Signal- und Konstantenbezeichnern, Instanziierungen und der zugehörigen Ident-Funktion auftritt, wird durch die Implementierung als Abbildung (Datenstruktur *map*) entsprochen. Da der Definitionsbereich einer Abbildung eine Menge ist, kann kein Bezeichner doppelt eingetragen werden. Beim Versuch des wiederholten Eintrags in die durch die Abbildung implementierte Symboltabelle wird die entsprechende Fehlermeldung ausgegeben.

Die im folgenden aufgelisteten Punkte wurden in der vorliegenden Arbeit nicht berücksichtigt:

- In einer Automatendefinition können entartete Automaten (z. B. zwei unabhängige Automaten oder eine Menge von Zuständen ohne Transitionen) definiert werden, die durch eine fehlerhafte Modellierung ent-

standen sind und eigentlich gar nicht analysiert werden sollen. Diese Automaten müssen von einer weiterführenden Analyse erkannt werden.

- Als Konstanten sind zur Zeit nur Integer-Werte zugelassen. Bei der Weiterentwicklung des Werkzeugs müssen dafür rationale Zahlen mit uneingeschränkter Größe von Zähler und Nenner verwendet werden.
- In der Einschränkung der Ableitung bei Bezeichnern des Typs *STOPWATCH* muß beachtet werden, daß als Werte für die Ableitung nur null oder eins erlaubt sind. Dazu ist es notwendig, zur Compile-Zeit den Ausdruck in der Ableitungseinschränkung zu interpretieren.
- Zur Einschränkung der Ableitung dürfen nur Ausdrücke über konstante Werte und Konstantenbezeichner verwendet werden. Diese Bedingung wurde nicht implementiert, da noch eine Erweiterung um die Benutzung der Werte von Ableitungen anderer Variablen bevorsteht.
- Alle deklarierten Signale müssen benutzt werden. Diese Bedingung ist nicht zwingend zu analysieren, da sie lediglich der Vermeidung von Modellierungsfehlern dient.

4 Erzeugung der Normalform

Als theoretische Grundlage für die CTA-Konzepte dient der hybride Automat. Demnach wird ein System als eine Menge von Automaten modelliert. Um weiterführende Analysen durchführen zu können, muß das Modell, welches die CTA-Erweiterungen benutzt, in eine Normalform gebracht werden. Ein CTA-Modell in dieser Normalform besteht aus einem einzigen Modul, welches mehrere Automaten enthalten kann. Dabei wird die im CTA-Konzept enthaltene syntaktische Abkürzung der *THEN*-Synchronisation sowie die implizierte Erweiterung der Automaten beim Benutzen von Eingabesignalen durch entsprechende Transformationen beseitigt.

Um die Enthaltenseinshierarchie aufzulösen, werden die Automaten der Untermodule unter Anwendung der Umbenennungs- bzw. Identifizierungsfunktion aus dem jeweiligen Schablonenmodul ins enthaltende Modul, in der Enthaltenseinshierarchie also nach oben, übertragen.

4.1 Auflösung der *THEN*-Synchronisation

Als verkürzende Schreibweise sind in der CTA-Notation Transitionen zugelassen, die mit einer *THEN*-Synchronisation versehen sind. Das bedeutet, daß die Transition ausgewählt werden darf, wenn das erste Signal „empfangen“ wird. Beim Schalten dieser Transition wird, ohne Zeit vergehen zu lassen, sofort das zweite Signal „gesendet“.

Da ein Automat mit solchen Transitionen nicht direkt analysiert werden kann, wird der Automat dahingehend transformiert, daß es keine solchen Transitionen mehr gibt. Aus der Transition wird das zweite Signal gestrichen und der Zielzustand auf einen neuen Zustand gesetzt. Dessen Invariante ist per Konstruktion immer falsch. Daher muß sofort eine Transition ausgewählt werden, um den Zustand zu verlassen. Die einzige Transition, die es in dem neuen Zustand gibt, ist die, die in den ursprünglichen Folgezustand der alten Transition übergeht. Der Wächter dieser Transition ist immer erfüllt, und als Synchronisationssignal wird das zweite (also das *THEN*-Signal) „gesendet“.

Um die Codeerzeugung für HyTech zu erleichtern, wurde in der vorliegenden Realisierung ein Verfahren in leicht veränderter Form implementiert. Diese Transformation wird anhand des Beispiels in der Abb. 4.1 erklärt. Aus der ursprünglichen Transition wird wie oben das zweite Signal gestrichen und der Folgezustand der Transition auf einen neuen einzufügenden Zustand (*then_start*) gesetzt. Jedoch wird in der Zuweisungsklausel eine im Deklarationsteil vorher neu eingefügte Uhrenvariable auf den Wert null gesetzt. Die Invariante des neuen Zustandes besagt, daß der Automat nur solange in diesem Zustand bleiben darf, wie die Uhrenvariable noch den Wert null enthält. Da die Ableitung einer Uhrenvariablen immer eins ist,

vor der Transformation:

```
STATE start
{
  -- Wait for start of transfer and start belt.
  TRANS in_transfer
  {
    SYNC #start_in_transfer THEN !go;
  }
}
```

nach der Transformation:

```
LOCAL then_startClock: CLOCK;
. . .
STATE start
{
  -- Wait for start of transfer.
  TRANS then_start
  {
    SYNC #start_in_transfer;
    DO
    {
      then_startClock := 0;
    }
  }
}
STATE then_start
{
  -- We urgently have to use a transition to next state.
  INV
  {
    then_startClock = 0;
  }
  -- Start belt.
  TRANS in_transfer
  {
    SYNC !go;
  }
}
```

Abbildung 4.1: Transformation einer THEN-Synchronisation

verändert sich der Wert der Variablen, sobald Zeit vergeht. Durch die Invariante muß sofort eine Transition ausgewählt werden, um den Zustand zu verlassen. Die einzige Transition im neuen Zustand ist wie oben beschrieben wieder die, die in den ursprünglichen Folgezustand übergeht. Die Bedingung des Wächters ist per Konstruktion immer wahr; als Synchronisationssignal wird das ursprüngliche *THEN*-Signal „gesendet“.

Diese Transformation kann durch die Methode *ThenTransformation* in der Klasse *ctaModule* ausgeführt werden. In der Implementierung muß beim Namen der Uhrenvariablen als eindeutige Kennzeichnung der Name des Zustands enthalten sein, da bei unterschiedlichen Automaten im selben Modul nicht die gleiche Uhrenvariable benutzt werden darf.

4.2 Automatische Vervollständigung der Automaten

In der CTA-Notation bedeutet ein Eingabesignal für einen Automaten, daß dieses Signal im Automaten nicht eingeschränkt werden darf. Immer dann, wenn ein anderer Automat eine Transition mit diesem Signal auswählt, muß auch im betrachteten Automaten eine Transition mit dem Eingabesignal ausgewählt werden können. Ist dies nicht der Fall, so schränkt der Automat das Signal ein; eine solche Situation darf nie auftreten.

Um solche Fehler aufdecken zu können, wird in jedem Automaten, der ein Eingabesignal verwendet, ein Fehlerzustand *sync_error* eingefügt. Für jedes Eingabesignal des Automaten wird in jedem Zustand eine Transition zum Fehlerzustand generiert. Diese Transition soll immer dann ausgewählt werden, wenn keine andere Transition des Automaten, die mit dem Eingabesignal synchronisiert ist, auswählbar ist. Daher wird bei der neuen Transition in der Synchronisationsanweisung das behandelte Eingabesignal eingetragen. Als Wächter der neuen Transition zum Fehlerzustand wird die Konjunktion aller Negate der Wächter von den Transitionen desselben Zustandes gewählt, die das Eingabesignal in der Synchronisationsanweisung benutzen.

Wenn sich der Automat einmal im Fehlerzustand *sync_error* befindet, konsumiert der Automat alle Eingabesignale durch Transitionen, die wieder in den Fehlerzustand führen.

Die Abb. 4.2 zeigt den Automaten des Sensor-Moduls nach dem Einfügen des Fehlerzustandes und der notwendigen Transitionen. Die Wächter der eingefügten Transitionen in den ursprünglichen Zuständen sind jeweils jederzeit wahr, da keine andere Transition vorhanden ist, die das Eingabesignal *in* benutzt.

```
INPUT  in: SYNC
...
AUTOMATON
{
      STATE off_before
      {
         INV
         {   pos <= sens_pos - sensor_radius_min;   }
         TRANS on
         {
            GUARD
            {  pos >= sens_pos - sensor_radius_max;}
            SYNC !sens_on;
         }
         TRANS sync_error
         {
            SYNC ?in;
         }
      }
      STATE on
      {
         INV
         {   pos <= sens_pos + sensor_radius_max;  }
         TRANS off_after
         {
            GUARD
            {  pos >= sens_pos + sensor_radius_min;}
            SYNC !sens_off;
         }
         TRANS sync_error
         {
            SYNC ?in;
         }
      }
      STATE off_after
      {
         TRANS off_before
         {
            SYNC ?in;
         }
      }
      STATE sync_error
      {
         TRANS sync_error
         {
            SYNC ?in;
         }
      }
}
```

Abbildung 4.2: Fehlerzustand für Eingabesignale

HyTech bietet in der zugehörigen Notation nur Konjunktionen von Ungleichungen bzw. Gleichungen als Ausdrücke für Wächter an, wobei die logischen Operationen Negation und Disjunktion nicht unterstützt werden. Da in der oben beschriebenen Transformation jedoch Negate benutzt werden, müssen diese sukzessive nach den folgenden Regeln umgeformt werden:

- Besteht die Formel aus dem Negat einer Relation, so wird die NOT-Operation „ausgerechnet“; es wird also die invertierte Relation verwendet.
- Das Negat einer Konjunktion wird nach der De-Morgan’schen Regel in eine Disjunktion einzelner Negate umgeformt.
- Zur Auflösung einer Disjunktion wird die Transition dupliziert. Jeder Wächter erhält genau einen Teilausdruck der ursprünglichen Disjunktion.

Diese Funktionalität wurde in der vorliegenden Implementierung aus Aufwandsgründen nicht realisiert.

4.3 Auflösung der Hierarchie

Für alle weiteren Analysen und Transformationen ist es notwendig, die Modulstruktur der CTA-Modelle aufzuheben. Zu diesem Zweck wurde die Methode *Flatten()* in den Klassen *ctaSystem* und *ctaModule* implementiert.

Die Flatten-Funktion überträgt alle lokalen Komponenten (also Automaten, Deklarationen und Initialisierungen) aus den Instanzen jeweils unter Anwendung der *ident*-Funktion in das Modul, welches die Instanz enthält. Die *ident*-Funktion identifiziert die Schnittstellenkomponenten mit den entsprechend der Instanziierungsvorschrift zugeordneten Komponenten des enthaltenden Moduls. Alle lokalen Komponenten werden mit einem Präfix versehen, so daß ein global eindeutiger Name entsteht. Das Präfix besteht aus der Aneinanderreihung der Namen aller Vaterinstanzen und des Wurzelmoduls in der Enthaltenseinshierarchie. Die Funktion traversiert durch rekursive Aufrufe die ganze Modulstruktur.

Die Hierarchieauflösung beginnt mit dem Aufruf der Funktion *Flatten()* des Objektes der Klasse *ctaSystem*, in dem das ganze Modell enthalten ist. In dieser Funktion wird das zu benutzende Präfix zur Konstruktion des absoluten Namens auf den Namen des Wurzelmoduls gesetzt. Das interne Präfix aller Komponentenobjekte in der Schnittstelle des Wurzelmoduls wird mit diesem Präfix belegt. Danach wird die Funktion *Flatten()* des Wurzelmoduls aufgerufen. Als Parameter wird das bisherige Präfix und die leere Identifizierungsfunktion *ident* übergeben.

Der Algorithmus der Funktion *Flatten()* in der Klasse *ctaModule* wird in der Abb. 4.3 zum besseren Verständnis in Pseudocode-Notation dargestellt. Zunächst werden alle Variablen- und Signaldeklarationen ausgegeben, die nicht umbenannt werden müssen, d. h. die nicht im Definitionsbereich der Funktion *ident* liegen. Die Vergabe des Präfixes erfolgte schon in der *Flat-*

```
module.Flatten(prefix, ident)
    // module.Flatten() traverses the composition and puts all automata
       into the root module.
    // module.interface is a function that returns the component
       object for a given name.
    // module.initial is the initial configuration of the module..
    // module.automata is a set contatining all the local automata.
    // module.instantiations is a set of instantiations.
    // instantiation.ident is the renaming function for a given instantiation.
    // instantiation.templateModule is the module which is instantiated by
       instantiation.
    // instantiation.instanceName is the name of the instantiated module.

    // The Print-Procedures uses the interface for output if the component is
      not in domain
    // of ident. If it is in domain of ident then use the value in range of ident
      for output.

    // Output of interface components which we have not to rename
    Print(module.interface)
    // Renaming states
    for each automaton in module.automata do
        for each state in automaton.states do
            set prefix of state to prefix
        end
    end
    // Output of initialisation.
    Print(module.initial)
    // Output of automata.
    Print(module.automata)
    // Output of instantiations.
    for each instantiation in module.instantiations do
        newPrefix = prefix + instantiation.instanceName
        // Rename components in interface.
        for each component in instantiation.templateModule.interface do
           set prefix of component to newPrefix
        end
        // Set new components in range of interface belonging to ident.
        for each d in dom(instantiation.ident) do
           set instantiation.templateModule.interface(d) to instantiation.ident(d
        end
        // for each component apply the ident from parents
        newIdent = ident( instantiation.ident )
        instantiation.templateModule.Flatten(newPrefix, newIdent)
    end
end
```

Abbildung 4.3: Algorithmus zur Auflösung der Enthaltenseinshierarchie

ten-Funktion des übergeordneten Moduls. Danach werden bei allen Zuständen in den Automaten die Präfixe gesetzt. Nun können alle Initialisierungen und alle Automaten ausgegeben werden.

Das Umbenennen bei den Instanziierungen erfolgt in mehreren Schritten. Zunächst wird das neue Präfix konstruiert, indem der Name der Instanz (getrennt durch einen Unterstrich) an das aktuelle Präfix angehängt wird. Dieses Präfix wird bei allen Komponenten im Deklarationsteil des Schablonenmoduls eingetragen.

Durch die Identifizierung der Schnittstellenkomponenten der Submodule mit den Komponenten des enthaltenden Moduls mittels *ident*-Funktion enthält nun der Wertebereich der *ident*-Funktion die Namen der Komponenten im enthaltenden Modul. Da dieses Modul wiederum eine Instanz sein kann, muß diese *ident*-Funktion mit den *ident*-Funktionen der übergeordneten, schon vorgenommenen, Instanziierung verknüpft werden, welche der Funktion *Flatten()* als Parameter übergeben wurde. Die Verknüpfung erfolgt dadurch, daß im Wertebereich der aktuellen *ident*-Funktion alle Werte, die im Definitionsbereich der übergebenen *ident*-Funktion enthalten sind, durch den entsprechenden Wert im Wertebereich der übergebenen Funktion *ident* ersetzt werden, also:

$$\forall x\colon ((\text{inst.ident}(x) \in \text{dom}(\text{ident})) \rightarrow (\text{newIdent}(x) = \text{ident}(\text{inst.ident}(x)))) .$$

Sind diese Vorbereitungen abgeschlossen, wird die *Flatten*-Funktion unter Angabe des aktuellen Präfixes und der nunmehr für die auszugebende Modulinstanz gültigen *ident*-Funktion aufgerufen. Damit wird der Enthaltenseinsbaum entsprechend einer Tiefensuche traversiert; die entsprechenden Automaten werden ausgegeben.

5 Analysen und Werkzeuge

Die Testautomaten-Analyse, die anhand des Syntaxbaumes durchgeführt wird, sowie die weitere Verarbeitung und die Codeerzeugung für HyTech werden in diesem Kapitel beschrieben.

Analysen, die über die Kontextüberprüfung hinausgehen, wurden - mit Ausnahme der Testautomaten-Analyse - in der vorliegenden Arbeit nicht implementiert. Vielmehr wird für die Erreichbarkeitsanalyse das Werkzeug HyTech verwendet; das in die Normalform transformierte CTA-Modell wird dafür in die HyTech-Notation übersetzt.

Als Ausblick werden am Ende dieses Kapitels einige nützliche Erweiterungen für das implementierte Werkzeug vorgestellt.

5.1 Testautomaten-Analyse

Eine Erweiterung der CTA-Module liegt in der Möglichkeit, überprüfen zu können, ob ein gegebenes Modul seine Umgebung nur beobachtet, d. h. ob der enthaltene Automat ein reiner Testautomat ist. Diese Überprüfung muß anhand der hierarchischen Struktur des CTA-Modells durchgeführt werden, also vor dem Auflösen der Enthaltenseinshierarchie.

Weil in einem Beobachtungsmodul alle Variablen und Signale als *INPUT* oder *LOCAL* deklariert sein müssen, ist es lediglich notwendig, alle Restriktionstypen zu kontrollieren. Durch die Kontextanalyse wird die Überprüfung der korrekten Benutzung der Eingabevariablen vorgenommen. Daß die Eingabevariablen nicht eingeschränkt werden, wird durch die im Abschnitt 4.1 erläuterte Erweiterung um den Fehlerzustand sichergestellt.

In der Implementierung wurde die Überprüfung, ob es sich um ein reines Beobachtungsmodul handelt, in der Methode *IsObserver()* der Klasse *ctaModule* realisiert.

5.2 HyTech-Codeerzeugung

Da das Werkzeug HyTech die Erreichbarkeitsanalysen bereits zur Verfügung stellt, wird in diesem Projekt die Erzeugung des Produktautomaten und die Erreichbarkeitsanalyse nicht implementiert, sondern es wird eine Codeerzeugung für HyTech zur Verfügung gestellt, um dessen Analysemöglichkeiten nutzen zu können.

Voraussetzungen für die Benutzung von HyTech sind das Aufheben der modularen Struktur und das Übersetzen des flachen Moduls in die HyTech-Notation. Dieses kann erst im Anschluß an alle auf der modularen

Struktur durchzuführenden Analysen der CTA-Eigenschaften und alle notwendigen Transformationen erfolgen. Die im vorherigen Kapitel vorgestellten Transformationen zur Erzeugung der Normalform werden benutzt.

Die Umwandlung der Hierarchie in eine flache Struktur erfolgt, indem das gesamte Modell durch die *Flatten*-Funktion flach in einen Puffer ausgegeben wird. Nach dem Abschluß des Funktionsaufrufs wird das gesamte interne Modell verworfen. Anschließend wird das Modell durch das Compilerfrontend aus dem Puffer neu eingelesen. Dadurch liegt das Modell in Form eines einzigen Moduls, welches nur noch Automaten enthält, intern zur weiteren Verarbeitung vor. Diese Struktur kann durch die Codeerzeugung in der HyTech-Notation ausgegeben werden.

Ein HyTech-Modell besteht aus einer Menge von miteinander über Signale kommunizierenden Automaten. Da die Reihenfolge des Aufbaus einer HyTech-Modellierung von der CTA-Notation abweicht, sind bei der Codeerzeugung folgende Schritte notwendig:

- **Konstanten:** Die Konstanten werden bei HyTech mittels *define*-Anweisungen global deklariert. Es müssen alle Konstanten aus dem Deklarationsteil des Moduls gesucht und ausgegeben werden.
- **Variablen:** Alle Variablen werden vor der Auflistung der Automaten in einem mit *var* eingeleiteten Deklarationsteil ausgegeben. Dazu müssen alle Variablen aus dem Deklarationsteil des CTA-Moduls ausgegeben werden.
- **Automaten:** Die Automaten werden in der HyTech-Notation in ähnlicher Weise dargestellt wie in der CTA-Notation. Allerdings müssen alle in einem Automaten benutzten Synchronisationsmarken zu Beginn des Automaten aufgelistet werden, um das Alphabet des Automaten zu definieren. Daraus ergeben sich die folgenden Schritte bei der Ausgabe eines jeden Automaten:
  - Das Alphabet der Synchronisationsmarken ist definiert durch die Menge der im Automaten benutzten Marken. Dazu werden alle Synchronisationsanweisungen der Transitionen durchlaufen und anschließend die Liste aller Synchronisationsmarken ausgegeben.
  - Weiterhin wird die Initial-Konfiguration des Automaten ausgegeben. Im CTA-Modul ist eine Menge der Startzustände für alle im Modul enthaltenen Automaten angegeben. Aus dieser Menge wird der zum jeweiligen Automaten gehörende Startzustand gewählt. Die Einschränkungen für die Variablen werden für jeden der erzeugten Automaten in der *initially*-Klausel angegeben.
  - Die Menge der Zustände wird analog zum CTA-Aufbau ausgegeben, wobei lediglich die Abweichungen in der Syntax zu beachten sind.

Die Implementierung der HyTech-Ausgabe wird in den Methoden *PrintHyTech()* der Syntaxbaumknoten realisiert. Bei einigen Syntaxbaumknoten kann auch die schon implementierte CTA-Ausgabe benutzt werden, da die Syntax, die hier verwendet wird, der von HyTech benutzten sehr ähnlich ist (z. B. bei Ausdrücken).

Da einige CTA-Konstrukte nicht unmittelbar nach HyTech übersetzt werden können und die entsprechenden Transformationen in diesem Projekt aufgrund des hohen Aufwandes nicht implementiert worden sind, wurde eine Spracheinschränkung von CTA zu CTA* vorgenommen. Die Menge der CTA*-Modelle, für die in der vorliegenden Arbeit die Codeerzeugung für HyTech realisiert wurde, ist durch die im folgenden aufgeführten Einschränkungen gekennzeichnet:

- Bei CTA* kann in der Initialisierungsanweisung nur ein Initialzustand angegeben werden. Bei der Transformation wird ein Startzustand eingefügt, von dem aus, ohne Zeit vergehen zu lassen, in einen der beiden angegebenen Initialzustände übergegangen werden muß.
- In einer CTA*-Transition darf nur ein Folgezustand angegeben werden. Die CTA-Schreibweise, bei der das Eintragen von zwei alternativen Folgezuständen in einem Zustandsübergang möglich ist, muß durch Kopieren der Transition für jeden Zustand in die Normalform überführt werden.
- Für die in Abschnitt 4.2 beschriebene Transformation zur Vervollständigung der Automaten, in denen Eingabesignale benutzt wurden, liegt noch keine Implementierung vor. Daher ist in CTA*-Modellen der Übergang in einen Fehlerzustand bei einer potentiell einschränkenden Benutzung von Eingabesignalen nicht sichergestellt.
- In der CTA-Notation sind Übergänge in Zustände, in denen die Invariante falsch ist, prinzipiell erlaubt. In einem solchen Zustand darf keine Zeit vergehen, d. h. der Zustand muß sofort durch Auswahl einer Transition verlassen werden. Solche Situationen sind bei CTA* nicht erlaubt, da in der HyTech-Semantik keine Übergänge in Zustände mit falscher Invariante erlaubt sind.

5.3 Benutzung der zur Verfügung gestellten Komponenten

Die in den letzten Kapiteln und Abschnitten vorgestellten Komponenten wurden nicht zu einem fixierten Programm zusammengestellt, sondern sollen durch entsprechende Funktionsaufrufe wie bei einer Bibliothek flexibel benutzt werden. Sie stehen den Entwicklern, die an dem Projekt weiterarbeiten, zur Verwertung bzw. Einbindung in anderen Werkzeugen frei zur Verfügung.

Um eine kurze Einführung in die Benutzung der Komponenten zu geben, wird im folgenden der Standardablauf der einzelnen Schritte vorgestellt. Dabei dient die in den Quelltexten enthaltene *main*-Funktion als Beispiel zur Formulierung der Funktionsaufrufe.

Voraussetzung für die Arbeit mit den zur Verfügung gestellten Klassen sind die globalen Objektbezeichner *pubParsedSystem* und *pubParser*, für die zunächst die entsprechenden Objekte erzeugt werden müssen. Die Datei, welche die textuelle Repräsentation des CTA-Modells enthält, wird geöffnet und als *stream*-Objekt bereitgestellt.

Durch den Aufruf des Parsers, bei dem der Dateistrom als Parameter übergeben wird (*pubParser->Parse(inStream)*), wird jetzt das gesamte Modell aus der Datei eingelesen und der Syntaxbaum aufgebaut. Die Wurzel des Syntaxbaumes ist das Objekt *pubParsedSystem*, welches global sichtbar sein muß, damit die globale, von *bison* generierte Parserfunktion den Syntaxbaum in *pubParsedSystem* eintragen kann.

Als nächster Schritt muß die abkürzende Schreibweise für die *THEN*-Synchronisation transformiert werden. Diese Transformation muß vor der Kontextanalyse vorgenommen werden, da für einige Ergänzungen alle Strukturen aufgebaut sein müssen.

Der Aufruf der Kontextanalyse wird durch die Methode *CheckConsistency()* vorgenommen. Dabei wird das ganze Modell ein zweites Mal durchlaufen und z. B. auch die Abhängigkeiten der Module voneinander überprüft. Sind diese Arbeiten abgeschlossen, befindet sich ein konsistentes CTA-Modell im Arbeitsspeicher.

Anhand dieser Struktur kann nun z. B. die Testautomatenanalyse durchgeführt werden oder das ganze Modell ausgegeben werden.

Mit dem Aufruf der Funktion *Flatten()* des Objektes *pubParsedSystem* kann die Enthaltenseinshierarchie aufgelöst werden. Nach diesem Aufruf befindet sich das CTA-Modell in der Normalform, d. h. das Modell besteht aus einem Modul, welches nur noch Automaten und keine Instanzen mehr enthält.

Als letzer möglicher Schritt kann durch die Methode *PrintHyTech()* unter Angabe des Ausgabestromes für das nunmehr flache CTA-Modell die Codegenerierung für HyTech ausgeführt werden. Dadurch besteht die Möglichkeit, weiterführende Analysetechniken, die von HyTech angeboten werden, auf das Modell anzuwenden.

5.4 Analyse des Beispiel-Modells

Wurde die HyTech-Codeerzeugung für ein CTA-Modell erfolgreich abgeschlossen, so können die von HyTech angebotenen Analysemöglichkeiten genutzt werden. Im folgenden wird die Übersetzung für das im Abschnitt 2.3 vorgestellte CTA-Modell untersucht, wobei prinzipiell die gleichen Analysen wie in der HyTech-Fallstudie durchgefürt werden [16]. Der CTA-Text dieses Modells ist im Anhang A und der HyTech-Text im Anhang B enthalten.

In der HyTech-Notation folgt die Formulierung der Analysen im Anschluß an die Definition des zu untersuchenden Modells in der gleichen Datei. Im Analyseabschnitt wird die Arbeit mit Regionen ermöglicht: Für die zu benutzenden Regionen werden Variablen eingeführt, in denen Regionen zur Bearbeitung gespeichert werden können und mit denen Auswertungsausdrücke formuliert werden können.

Wie in Abb. 5.1 dargestellt, werden beim behandelten Beispiel folgende Variablen eingeführt: *reg_start* für die Initialregion, *reg_reachable* für die Region, die von *reg_start* aus erreichbar ist, *reg_interest* für die Region, in die das Modell gelangen soll, und die Region *reg_test_error*, in die das System im Fehlerfall gerät.

Der Variablen *reg_start* wird die Initialisierungsregion zugewiesen. Diese besteht aus aus einer Konjunktion von linearen Einschränkungen über die globalen Variablen und Festlegungen bezüglich der Zustände, in denen sich die Automaten befinden müssen.

Plausibilitätstest. Zunächst wird anhand einer Plausibilitätsüberprüfung bewiesen, daß ein empfangenes Werkstück in den Bereich hinter dem Sensor B und vor dem Ende des Transportbandes transportiert wird. Die dazu erforderlichen Analysekommandos sind in der Abb. 5.1 dargestellt: In der Variablen *reg_interest* wird die zu erreichende Region spezifiziert. Durch den *reach-forward*-Operator wird mittels Vorwärtsschritten eine Region erzeugt, die alle von der Startregion erreichbaren Konfigurationen enthält. Im Anschluß daran wird die Menge der Konfigurationen, die sowohl in *reg_reachable* als auch in *reg_interest* vorkommen, bestimmt. Ist diese Menge leer, so kann die interessante Region nicht erreicht werden, der Plausibilitätstest ist also fehlgeschlagen. Anderenfalls wird ein Lauf des von HyTech erzeugten Produktautomaten von der Startregion in eine Konfiguration dieser Schnittmenge ausgegeben. Dabei wird z. B. auch die Reihenfolge des Auftretens der Signale deutlich.

Erreichbarkeitstest für verbotene Zustände. Die Region, die vom betrachteten System nie erreicht werden darf, wird in der Variablen *reg_test_error* bereitgestellt. Wie in Abb. 5.2 definiert, liegt ein Fehlerfall dann vor, wenn entweder die Signale in der falschen Reihenfolge aufgetreten sind oder zeitliches Fehlverhalten beim Senden der Signale festgestellt wird. Die Reihenfolge der Signale wird durch die Zustandsübergänge der Testautomaten verfolgt. Tritt ein unerwartetes Signal auf, wird in den Fehlerzustand übergegangen.

Die Bedingung, daß keine Zeit vergehen darf, während am Transportband z. B. gerade ein Werkstück vom Eingabenachbarn eintrifft und der Motor ausgeschaltet ist, wird durch das Messen der Zeit mit der Uhrenvariablen *critical_time* im jeweiligen Zustand kontrolliert. Hat der Automat der Instanz *iTestIn* gerade das Signal *in* erhalten, wenn der Automat der Instanz *iTestGoingStopped* noch nicht das Signal *go* bekam und haben die Uhren der beiden Automaten größere Werte als null, so liegt ein Fehlverhalten vor.

```
var reg_start,
    reg_reachable,
    reg_interest,
    reg_test_error: region;

-- The starting region.
reg_start :=
      MBeltSystem_iBeltModel_pos >=
MBeltSystem_iBeltModel_length+MBeltSystem_plate_radius
    & loc[a4] =
MBeltSystem_iBeltModel_iBeltMovement_stopped_and_unloaded
    & loc[a5] = MBeltSystem_iBeltModel_iSensorA_off_after
    & loc[a6] = MBeltSystem_iBeltModel_iSensorB_off_after
    & loc[a1] = MBeltSystem_iBeltContext_iBeltInContext_start
    & loc[a2] =
MBeltSystem_iBeltContext_iBeltOutContext_wait_for_plate
    & loc[a3] = MBeltSystem_iBeltController_start
    & MBeltSystem_iBeltTest_iTestGoingStopped_critical_time = 0
    & MBeltSystem_iBeltTest_iTestIn_critical_time = 0
    & MBeltSystem_iBeltTest_iTestOut_critical_time = 0
    & loc[a7] =
MBeltSystem_iBeltTest_iTestGoingStopped_wait_for_begin
    & loc[a8] = MBeltSystem_iBeltTest_iTestIn_wait_for_begin
    & loc[a9] = MBeltSystem_iBeltTest_iTestOut_wait_for_begin
    ;
-- Define interesting region: b1_pos is behind sensorB
--   and before the end of the belt.
reg_interest :=
      (MBeltSystem_iBeltModel_pos >
              MBeltSystem_iBeltModel_sensB_pos
              + MBeltSystem_plate_radius)
    & (MBeltSystem_iBeltModel_pos <
              MBeltSystem_iBeltModel_length
              -MBeltSystem_plate_radius);

-- Compute from reg_start reachable region.
reg_reachable := reach forward from reg_start endreach;

-- Print trace from start to region of interest.
if empty( reg_interest & reg_reachable )
then
    prints "No way to region of interest";
else
    prints "Trace to region of interest:";
    print trace to reg_interest using reg_reachable;
endif;
```

Abbildung 5.1: Analysekommandos für den Plausibilitätstest

```
-- Use the testing automaton to check for erroneous configurations.

reg_test_error :=
        -- Wrong sequence of signals for transfer from bA to b1.
      loc[a7] = MBeltSystem_iBeltTest_iTestGoingStopped_error
        -- Wrong sequence of signals for transfer from b1 to bB.
    | loc[a8] = MBeltSystem_iBeltTest_iTestIn_error
        -- Wrong sequence of start/stop-signals.
    | loc[a9] = MBeltSystem_iBeltTest_iTestOut_error
        -- Belt stopped in critical time interval during
        --  transfer from bA to b1.
    | (  loc[a8] = MBeltSystem_iBeltTest_iTestIn_wait_for_end
       & loc[a7] =
MBeltSystem_iBeltTest_iTestGoingStopped_wait_for_begin
       & MBeltSystem_iBeltTest_iTestIn_critical_time > 0
      & MBeltSystem_iBeltTest_iTestGoingStopped_critical_time > 0
      )
        -- Belt stopped in critical time interval during
        --  transfer from b1 to bB.
    | (  loc[a9] = MBeltSystem_iBeltTest_iTestOut_wait_for_end
       & loc[a7] =
MBeltSystem_iBeltTest_iTestGoingStopped_wait_for_begin
       & MBeltSystem_iBeltTest_iTestOut_critical_time > 0
       & MBeltSystem_iBeltTest_iTestOut_critical_time > 0
      )
    ;

-- Check reachability of error region.
-- Print trace from start to region of interest.
if empty( reg_test_error & reg_reachable )
then
    prints "No way to error region of testing automaton.";
else
    prints "Trace to error region of testing automaton:";
    print trace to reg_test_error using reg_reachable;
endif;
```

Abbildung 5.2: Analysekommandos für die Fehlerregion

Eine genauere Beschreibung des Zusammenspiels der Testautomaten gibt [16].

5.5 Ausblick

Im folgenden werden Ideen, Konzepte und Aussichten dargestellt, die unter Verwendung des realisierten Projektes zu einem vollständigen Analysewerkzeug für CTA-Modelle führen.

Produktautomat. Ausgangspunkt ist die Repräsentation des Modells durch eine Menge von miteinander über Synchronisationsmarken kommunizierenden nebenläufigen Automaten.

Um die Erreichbarkeitsanalyse durchführen zu können, ist die Transformation des Modells, bestehend aus einer Menge von Automaten, in ein Modell notwendig, welches durch einen einzigen Automaten repräsentiert wird. Diese Transformation erfogt durch die sukzessive Zusammenfassung zweier Automaten zu einem Produktautomaten, bis nur noch ein Automat übrig ist. Dieser enthält in den Transitionen zwar noch die Synchronisationsmarken, jedoch spielt in dieser Automaten-Normalform die Synchronisation nach dem CSP-Konzept keine Rolle mehr. Für diese Automatenklasse existieren bereits untersuchte Algorithmen [33].

Erreichbarkeitsanalyse. Die Erreichbarkeitsanalyse erbringt den Nachweis, ob eine spezielle Konfiguration in einem Lauf des Automaten erreicht werden kann.

Die Erreichbarkeitsanalyse wird mit Hilfe des Regionen-Konzeptes realisiert. Eine Region ist eine Menge von Konfigurationen, die durch die in den Wächtern und Invarianten angegebenen linearen Einschränkungen (Gleichungen bzw. Ungleichungen) sowie durch den Zustand des Automaten angegeben werden. Regionen stellen eine symbolische Repräsentation der Zustände des Erreichbarkeitsgraphen dar. Die Erzeugung des Erreichbarkeitsgraphen wird dadurch effizienter. In der Implementierung werden Regionen durch die von einer speziellen Bibliothek bereitgestellten Datenstruktur *Polyeder* dargestellt. Die beiden Operationen Vereinigung und Durchschnitt, die sehr häufig in den Erreichbarkeitsberechnungen verwendet werden, sind bei der Datenstruktur *Polyeder* möglich.

Wichtige zu implementierende Algorithmen sind Vorwärts- und Rückwärtsschritte sowie der *reach*-Operator. Durch einen Vorwärtsschritt wird der Erreichbarkeitsgraph durch alle von der aktuellen Region aus möglichen Übergänge erweitert (Rückwärtsschritt analog). Durch einen *reach*-Operator kann gefragt werden, ob von einer bestimmten Region aus eine andere Region im betrachteten Automaten erreichbar ist.

Spezifikation. Zur Formulierung der Spezifikation muß eine kontextfreie Grammatik entwickelt werden, um Aussagen über das System machen zu können. Dazu wird eine Sprache benötigt, die es ähnlich HyTech gestattet, erreichbare Regionen und logische Operationen darüber zu spezifizieren.

Determinismus-Check. Eine sehr nützliche Analyse wäre ein Test eines Automaten auf Determinismus. Dieser ist besonders interessant bei den Automaten, die das Steuerprogramm modellieren, da hier ein Interesse an einer Codeerzeugung vorliegt. Bei korrekter Übersetzung in die Zielsprache bleiben dann die bewiesenen Eigenschaften erhalten - vorausgesetzt die Umgebung erfüllt ebenfalls ihr entsprechendes Modell.

6 Vorgehensweise

Wie der Arbeitsprozeß organisiert wurde, welche Probleme auftraten und welche Maßnahmen zur Qualitätssicherung genutzt wurden, wird im folgenden Abschnitt beschrieben.

Um aus der CTA-Modellierung eine Laufzeit-Datenbasis zu erhalten, wurde in der ersten Arbeitsphase ein Klassensystem zur internen Repräsentation von CTA-Modellen entwickelt. Dieses Klassensystem beinhaltet die Klassen für Module, Automaten, Zustände, Transitionen, Variablen und Signale; daraus erzeugte Objekte werden vom Parser zum abstrakten Syntaxbaum zusammengesetzt. Ebenfalls in der ersten Arbeitsphase entstand die kontextfreie Grammatik, welche die Sprache der Notation der CTA-Modelle definiert.

Der Aufgabenteil des Compiler-Frontends war ursprünglich in einer späteren Phase des Projektes vorgesehen, damit genügend Zeit zur Untersuchung der algorithmischen Möglichkeiten bleibt. Jedoch wurde bereits zum Aufbau der Strukturen, die zum Testen benötigt wurden, ein automatisches Erzeugen der internen Repräsentation unerläßlich.

In der zweiten Phase wurde die Kontextanalyse realisiert. Die erforderlichen Transformationen sowie die HyTech-Codeerzeugung waren Bestandteile der dritten Phase.

6.1 Zeitplanung und Ablauf

Da in dieser Diplomarbeit neben der schriftlichen Ausarbeitung besonders die Implementierung der ersten Teile des Analysewerkzeugs im Vordergrund stand, mußte abgeschätzt werden, wie weit die Entwicklungsarbeiten in dem zur Verfügung stehenden Zeitraum durchgeführt werden können. Dazu wurden die anstehenden detaillierten Aufgaben jeweils zu Arbeitspaketen zusammengefaßt und der Zeitplanung zugeführt. Zunächst wurden Zeitschätzungen abgegeben. Während der Entwicklungsarbeiten wurden die benötigten Zeiten notiert, um nach dem Anlauf der betrachteten Phase diese Schätzungen aufgrund der nun vorliegenden genaueren Erfahrungswerte korrigieren zu können.

Im folgenden werden einige der oben genannten Aspekte geschildert. Dabei wird die Durchführung der erste Phase genauer beschrieben. Auf die Zeitplanung für die restlichen Phasen des Projektes wird lediglich kurz eingegangen.

Organisation der ersten Phase

Für die erste Phase sind folgende Aufgaben definiert worden, wobei die Zahlen in den Klammern die benötigte Arbeitszeit bezogen auf die Programmentwicklung in Tagen darstellt.

a) Klassen für die interne Repräsentation (Syntaxbaumknoten) (8)

b) Kontextfreie Grammatik zur Notation (4)

Grammatik aufschreiben, also Syntax definieren; immer wieder an Syntaxbaumknoten anpassen

c) Scanner (3)

Morpheme identifizieren (Schlüsselwörter, Konstanten, Wertübergabe an Parser);

Schnittstelle definieren (Pseudo-„Kapselung“ nach außen)

d) Parser (5)

Syntaxbaumknoten erzeugen und mit Werten füllen

Parser als Klasse mit Schnittstelle bereitstellen (Der Parser wird so behandelt, als wäre er tatsächlich gekapselt.)

e) Pretty-Printer und Aufbau des Syntaxbaumes (8)

(Da der Pretty-Printer eine Ausgabe der internen Struktur ermöglicht, ist er die ideale Komponente zum Test aller anderen Teilaufgaben.)

Auf der Grundlage des Phasenmodells Analyse, Entwurf, Implementierung und Test wurden die Aufgaben verfeinert und die Zeitabschätzungen gebildet.

Zuerst stand die Entscheidung an, ob Scanner und Parser von Hand oder mittels Scanner- und Parsergeneratoren (hier standen flex und bison zur Verfügung) entwickelt werden sollten. Für das System, für das ein reiner OO-Entwurf vorgesehen war, schienen *flex* und *bison* nicht geeignet zu sein, da diese C-Code (inklusive globaler Komponenten) erzeugen. Dadurch wurde die Von-Hand-Implementierung von Scanner und Parser in Betracht gezogen.

Für die Entscheidung zwischen Von-Hand- und Werkzeug-Erzeugung des Compiler-Frontends wurde folgende Vorgehensweise gewählt: Um den Zeitaufwand abschätzen zu können, sollte zwei Tage lang begonnen werden, den Scanner und den Parser von Hand zu implementieren. Beim anschließenden Durchführen der gleichen Arbeiten mit den Werkzeugen sollte beobachtet werden, ob die werkzeugunterstützte Herangehensweise so viel effizienter ist, daß man den Verzicht auf eine reine objekt-orientierte Implementierung in Kauf nimmt. Als Ergebnis stand nach diesen vier Tagen fest, daß durch die Werkzeuge *flex* und *bison* eine Unterstützung gegeben wird, auf die nicht verzichtet werden kann.

Wie bei der Durchführung der Teilaufgaben vorgegangen wurde und welche Zeiten tatsächlich benötigt wurden, ist in Abb. 6.1 dargestellt. Alle aufgelisteten Teilaufgaben beziehen sich immer auf eine Menge von Syntaxelementen, die zu repräsentieren sind. Deshalb wurden einige einfache Syntaxbaumknoten ausgewählt, für die zunächst alle Teilaufgaben erfüllt wurden (Teil 1). Dabei stand im Vordergrund, den Umgang mit den einzelnen Phasen zu erproben und einmal alle Phasen vollständig zu durchlaufen. Ein weiterer Vorteil dieser Herangehensweise liegt darin, nach der Realisierung des ersten Teils über ein zwar unfertiges, aber lauf- und testfähiges System zu verfügen. Im Gegensatz zur prototypischen

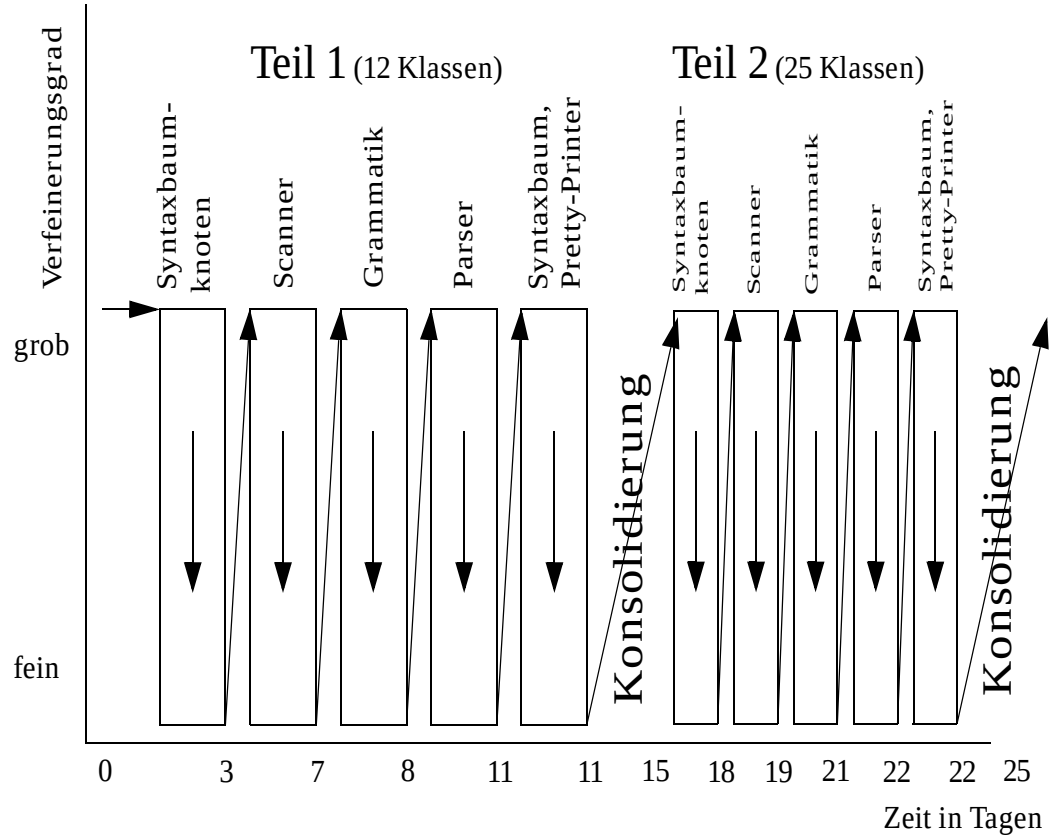

Abbildung 6.1: Prozeßmodell für die erste Phase

Implementierung wurde hier für einen bestimmten Teil des Systems der Entwurf vollständig in die Implementierung umgesetzt. Weil dabei die Erfahrung erlangt wurde, daß diese Vorgehensweise für das vorgesehene Projekt günstig ist, wurde sie auf den restlichen, größeren Teil 2 angewendet.

Es wurde die Entscheidung getroffen, innerhalb eines Teils nicht alle Aufgaben gleichzeitig durch ein Phasenmodell laufen zu lassen, sondern für jede Aufgabe einzeln den Entwurf und so weit wie nötig die Verfeinerungen zu realisieren. Innerhalb des ersten Teils wurden somit alle Knoten analysiert, entworfen, implementiert und getestet. Danach wurden die Morpheme identifiziert, die Grammatik für dieselben Syntaxbaumknoten entwickelt und in Form eines Bison-Quelltextes mit den Anweisungen zum Aufbau des abstrakten Syntaxbaumes angereichert. Zum Test der Komponenten wurde der Pretty-Printer stufenweise entwickelt.

Weil die Erfahrungen aus diesem ersten Projektabschnitt zur Planung des weiteren Projektverlaufes herangezogen werden sollten, wurde der Arbeitsplan nach Abschluß des ersten Arbeitspaketes einer Überprüfung und Überarbeitung unterzogen. Als Konsequenz daraus wurde festgestellt, daß die Entwicklungsarbeiten in dieser Arbeit nur bis zum Abschluß der wesentlichen Kontextanalysen und der Auflösung der Enthaltenshierar-

Monat	Arbeitsphase	Soll	Ist
April	Phase 1: Schreiben der Ausarbeitung	4	10
	Phase2: Kontextanalyse	10	5
	Qualitätssicherung	2	0
	Einarbeitung in StdC++Lib, Portierung, Portierung auf DOS (99% Installationsarbeiten)	0	5
	Schreiben der Ausarbeitung	4	0
Mai	Kontextanalyse	0	1
	Schreiben der Ausarbeitung	0	4
	Qualitätssicherung	2	2
	Phase 3: Flatten-Funktion	10	5
	Schreiben der Ausarbeitung	2	3
	HyTech-Ausgabe / THEN-Auflösung	5	4
Juni	Konsolidierung	5	4
	Schreiben der Ausarbeitung	0	4
	Einleitung/Zusammenfassung	5	3
	Layout/Korrektur	12	11

Tabelle 6.1: Zeitplanung der weiteren Phasen

chie fortgeführt werden. Um jedoch erste Erfahrungen im Umgang mit den CTA-Modellen zu erlangen, war die Entwicklung einer eingeschränkten HyTech-Codeerzeugung wichtig.

Organisation der weiteren Phasen

Die Tab. 6.1 zeigt die Zeitschätzungen für die einzelnen Arbeitsphasen sowie die tatsächlich benötigte Zeit (Angabe in Tagen). Die größten Abweichungen von der Planung gab es bei der Implementierung der Flatten-Funktion, die durch eine vereinfachte Implementierung unter Nutzung der normalen Ausgabefunktion nur die halbe Zeit benötigt hat, und bei der Implementierung der Kontextanalyse, bei der durch die Nutzung geeigneter Datenstukturen auf einem abstrakteren Niveau implementiert werden konnte. Jedoch wurden bei der unerwarteten Umstellung auf die C++-

Standardbibliothek und der damit verbundenen Einarbeitungsphase 5 Tage benötigt, die nicht vorgesehen waren. Die Umstellung wurde einerseits notwendig, da durch eine Systemumstellung am Institut die alte Bibliothek langfristig nicht mehr zur Verfügung stehen wird. Andererseits bietet die neue Bibliothek neuere und bessere Implementierungen der Datenstrukturen.

6.2 Qualitätssicherung

In allen Entwurfsphasen werden zur Qualitätssicherung Reviews vorgenommen. In der Implementierungsphase wurden regelmäßig, mindestens aller zwei Wochen, **Konsolidierungsmaßnahmen** eingeleitet (Code-Review, Redesign, Vermessung), um den Entwurf mit der Implementierung konsistent zu halten. Dazu wurden jeweils zwei Arbeitstage eingeplant.

6.2.1 Konsolidierungsphasen

In den Konsolidierungsphasen wurden Code- und Design-Reviews durchgeführt, die zur Sicherstellung der Qualität der Software erforderlich wurden. In der ersten Konsolidierungsphase wurden mehrere Schwachstellen im Entwurf festgestellt, die größtenteils darin begründet waren, daß der Entwurf wegen unzureichender Erfahrung mit der Programmiersprache C++ nicht genau so wie vorgesehen implementiert werden konnte.

In der zweiten Konsolidierungsphase wurden Fehler vor allem in der Implementierung gefunden. Zum Beispiel wurden in einigen Klassen bei der Deallokierung des Speichers nicht alle aggregierten Objekte richtig freigegeben, d. h. es mußten einige Destruktoren überarbeitet werden. Die Ursache dafür war eine teilweise unvollständige Anpassung der Implementierung bei nachträglichen Veränderungen des Entwurfs. Weiterhin stellte sich heraus, daß bei der Benutzung von Polymorphie nicht berücksichtigt wurde, daß auch Destruktoren als *virtual* deklariert werden müssen, damit alle Attribute der konkreten Objekte der Unterklassen auch deallokiert werden.

Bei der Übersetzung der C++-Dateien wird der Compiler angewiesen, alle Hinweise und Warnungen auszugeben und Abweichungen vom ANSI-Standard zu signalisieren. Im vorliegenden Projekt werden stets die Optionen *-Wall* und *-ansi* des Compilers *g++* benutzt. Zur Überprüfung einiger Programmierrichtlinien wurde im Review-Prozeß ein Shell-Skript (*chkstyle*) verwendet.

Um einen Überblick über die Anzahl der angelegten Syntaxbaumknoten zu bekommen, wurde in der Klasse *ctaObject* ein Objektzähler implementiert, der im Konstruktor inkrementiert und im Destruktur dekrementiert wird. Dadurch können auch Informationen über den Zustand der Speicherverwaltung gewonnen werden. Auf diese Art wurde festgestellt, daß

40 Objekte in Wirklichkeit gar nicht freigegeben worden waren, obwohl die *delete*-Anweisungen für die Referenzen aufgerufen wurden. Die Ursache dafür lag in der unvollständigen Änderung der Destruktoren, nachdem andere Teile der entsprechenden Klasse einer Reorganisation unterzogen worden waren.

6.2.2 Vermessung und Inspektion

In der Konsolidierungsphase wurde das ganze Klassensystem vermessen, um Rückschlüsse auf die Qualität der Implementierung zu erhalten. Dabei wurde ein Teil der im Abschnitt 3.1 aufgeführten Richtlinien auf Realisierung überprüft.

Die verwendeten Maße wurden im wesentlichen von Erni und Lewerentz übernommen [34]. Insgesamt wurden bei vier von acht Metriken Ausreißer festgestellt. Eine dieser Klassen mußte überarbeitet werden. Für eine andere Klasse wurde festgestellt, daß sie mit besonderer Vorsicht zu verändern bzw. zu erweitern ist.

Bei der Vermessung wurde ein Problem nicht entdeckt, das in der nachfolgenden Inspektion auffiel: Existiert bei einer Klasse zu jedem Attribut eine Set- und eine Get-Methode, dann ist zwar die Flexibilität bei der Änderung klasseninterner Datenstrukturen, nicht jedoch das Geheimnisprinzip gewahrt. Es ist also ein Maß wünschenswert, welches über die Beachtung des Geheimnisprinzips Aufschluß gibt bzw. auf kritische Klassen aufmerksam macht.

Eine weitere sehr nützliche Ergänzung wäre eine Möglichkeit zur Lokalisierung von Klassenbestandteilen, die überdurchschnittlich oft verändert werden. Solche Bestandteile stellen aufgrund der häufigen Änderungen „Problemkinder“ dar, die einer genaueren Überprüfung zugeführt werden sollten. In diesem Projekt wurde bei der Inspektion nur die Tatsache benutzt, daß Klassendateien, die häufig geändert worden sind, vom Versionsmanagementsystem eine höhere Versionsnummer als normale Klassen erhalten.

6.2.3 Benutzte Werkzeuge

Zur Erzeugung des ausführbaren Programmes wurden der Compiler *g++* und das Werkzeug *make* verwendet. Als Versionsmanagementsystem wurde das Unix-Werkzeug *CVS* benutzt. Der *Emacs* diente in Zusammenarbeit mit dem Debugger *gdb* als Programmierumgebung. Beim Vermessen des Softwareprojektes wurde das CASE-Tool *SNiFF+* mit dem vom Lehrstuhl Software-Systemtechnik der BTU Cottbus entwickelten integrierten Meßwerkzeug *Crocodile* verwendet.

6.3 Reflexion

Ziel der Zeitplanung war die zuverlässige Angabe von Zeiträumen für die Bearbeitung der jeweiligen Teile der Diplomarbeit. Der ständige Überblick über den Fortschritt der Arbeiten hatte einerseits positive arbeitspsychologische Folgen, z. B. das Vermeiden von Qualitätseinbußen durch extreme Streßsituationen, und andererseits den Vorteil einer gut handhabbaren Rechenschaftslegung.

Der Nutzen der Zeitplanung wurde einerseits dadurch deutlich, daß bereits im April die Aussage darüber getroffen werden konnte, daß der ursprünglich angedachte Umfang der Entwicklungsarbeiten den Rahmen einer Diplomarbeit sprengen würde. Deshalb erfolgte eine Reduzierung des Umfangs auf das Compilerfrontend inklusive der auf der Modulstruktur durchzuführenden Kontextanalysen und Ansätze zur Weiterverarbeitung des internen Modells. Dieser Umfang konnte in genau der gegebenen Zeit realisiert werden.

Andererseits wurde die Bedeutung der Zeitplanung dadurch sichtbar, daß die terminlichen Absprachen zu Meilensteinen eingehalten werden konnten. Nur bei wenigen Arbeitspaketen wichen die tatsächlich benötigten Zeiten bis zu 100 Prozent von den Schätzungen ab. Dieser Wert war für die vorliegende Arbeit völlig ausreichend, zumal der größere Teil der Abweichungen unter 30 % lag. Diese Abweichungen hatten ihre Ursache in nicht vorhersehbaren Verlagerungen von Inhalten zwischen den verschiedenen Arbeitspaketen innerhalb einer Phase. D. h. es wurden einige Arbeitpakete in kürzerer Zeit erledigt, da bei anderen Arbeitpaketen indirekt oder unbewußt Zuarbeiten in der dort mehr benötigten Zeit geleistet worden sind.

Diese positiven Erfahrungen kamen dadurch zustande, daß in der Arbeitsatmosphäre am Lehrstuhl eine besondere Motivation zu Aspekten des Projektmanagements vorlag und daß ein individuell angepaßter Softwareprozeß verwendet wurde. In diesem Softwareprozeß wurde das Projekt in verschiedene Phasen unterteilt. In jeder Phase wurde ein für die entsprechenden Aufgaben speziell geeignetes Vorgehen gewählt, welches in Diskussionen jeweils validiert und entsprechend verworfen bzw. verbessert wurde. Das in der ersten Phase verwendete Modell wurde im Abschnitt 6.1 vorgestellt, ebenso die Begrenzung des zeitlichen Aufwandes beim Experimentieren mit den Werkzeugen *flex* und *bison*. Bei der Kontextanalyse in der zweiten Phase, die sich nicht durch komplizierte Realisierung, sondern durch viel Tipparbeit auszeichnet, hingegen wurde jede Funktionalität einzeln implementiert und getestet, da es sich hier im wesentlichen um die Implementierung nicht automatisiert erzeugbarer Analysen handelt.

Ein Problem, welches oft zu Unsicherheiten führte, war die Entscheidung, die ersten Phasen des Werkzeugs ausführlich zu implementieren. Eine Alternative dazu wäre die prototypische Implementierung der wesentlichen Komponenten des Werkzeugs mit dem Ziel, sich tiefergehend mit den Analysetechniken für die Automaten zu beschäftigen. Dadurch wäre die

vorliegende Arbeit sicher noch interessanter geworden. Weil aber das Werkzeug am Lehrstuhl weiterentwickelt werden soll, war es erforderlich, auf die ausführliche Implementierung Wert zu legen. Da die Codeerzeugung für HyTech lediglich dem Sammeln von Erfahrungen im Umgang mit den CTA-Modellen dient und nicht weiterentwickelt wird, wurde diese letzte Phase prototypisch durchgeführt.

7 Zusammenfassung

Als Ergebnis einer Fallstudie zur Modellierung einer Fertigungsanlagenkomponente mittels hybrider Automaten stellte sich heraus, daß sich diese Automaten prinzipiell sehr gut zur Modellierung kleiner hybrider Systeme eignen. Für die Entwicklung großer Systeme ist dieses Verfahren jedoch nicht geeignet, da weder die Modularität noch die Wiederverwendung durch Exemplarerzeugungsmechanismen unterstützt werden.

Die Cottbus Timed Automata, welche auf Erweiterungen der hybriden Automaten basieren, bieten eben diese und weitere Konzepte an, die den Entwurf von großen Systemen unterstützen.

Nach einer informellen Beschreibung dieser Konzepte wurde im Rahmen dieser Diplomarbeit eine Notation zur textuellen Repräsentation der CTA-Modelle entwickelt, für die eine kontextfreie Grammatik entworfen wurde. Durch das Bereitstellen eines Compilerfrontends wurde die textuelle Repräsentation in eine konsistente interne Repräsentation transformiert, wobei alle erforderlichen Analysen an der Modulstruktur durchgeführt wurden.

Durch verschiedene Transformationen zur Aufhebung der abkürzenden Schreibweisen und der modularen Strukturen wurde das CTA-Modell in eine Normalform gebracht, die im wesentlichen einer Menge von hybriden Automaten entspricht. Diese Normalform dient als Grundlage für weitere Transformationen und Analysen.

Da die zusätzliche Implementierung von weiterführenden Analysetechniken den Rahmen einer Diplomarbeit gesprengt hätte, ist eine Codeerzeugung realisiert worden, die eine Analyse anhand des Werkzeugs HyTech ermöglicht. HyTech bietet sowohl eine Sprache zur Formulierung von Spezifikationen als auch die Erreichbarkeitsanalyse an.

Damit ist in einer sechsmonatigen Entwicklungszeit ein Werkzeug entstanden, durch das es ermöglicht wird, sowohl eine hierarchische Modellierung mit Konzepten zur Wiederverwendung von ähnlichen Modulen anzubieten, als auch die konzeptuellen Grundlagen und die bereits gründlich untersuchten und implementierten Algorithmen für hybride Automaten zu nutzen.

Für die bevorstehende Weiterentwicklung des Werkzeugs stehen die Möglichkeiten der Implementierung der Verarbeitung und Analyse der CTA-Modelle offen. Durch das Nutzen der Konzepte der objektorientierten Programmierung und durch einen flexiblen Entwurf, der unter anderem wegen der Anwendung von Entwurfsmustern und der Benutzung der neuen ANSI/ISO-Standardbibliothek entstand, wurde den hohen Anforderungen an Wiederverwendbarkeit und Wartbarkeit entsprochen.

Neben den in Abschnitt 5.5 aufgezeigten noch zu realisierenden Erweiterungen des Werkzeugs stehen Erfahrungen bei der Modellierung größerer Systeme mit dem CTA-Konzept aus.

Literaturverzeichnis

[1] G. H. Mealy. A method for synthesizing sequential circuits. In *Bell System Technical Journal 34*, pages 1045-1079, 1955.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. In *Theoretical Computer Science 126*, pages 183-235, 1994.

[3] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pages 278-292, 1996.

[4] Insup Lee, Hanene Ben-Abdallah and Jin-Young Choi. A Process Algebraic Method for the Spezifikation and Analysis of Real-Time Systems. In *Constance Heitmeyer and Dino Mandrioli, editors, Formal Methods for Real-Time Computing*, pages 167-194, Jon Wiley & Sons Ltd, Chichester, 1996.

[5] P. M. Merlin and D. J. Farber. Recoverability of communication protocols - implications of a theoretical study. *IEEE Transactions on Communications*, 1976.

[6] Louchka Popova-Zeugmann and Monika Heiner. Worst-case Analysis of Concurrent Systems with Duration Interval Petri Nets. In *Proceedings of the EKA '97*, pages 162-179, 1997.

[7] Nicolas Halbwachs. Delay Analysis in synchronous Programs. In C. Courcoubetis, editor, *Proceedings of the 5th Ann. Conf. on Computer-Aided Verifikation (LNCS 697)*, pages 333-346, Springer-Verlag, Berlin, 1993.

[8] Thomas A. Henzinger, Pei-Hsin Ho und Howard Wong-Toi. A user guide to HyTech. In *Proceedings of the First Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS 1019)*, pages 41-71, Springer-Verlag, Berlin, 1996.

[9] Annette Lötzbeyer. Spezifikation, Modellierung und Verifikation von Realzeitsystemen. FZI-Publication 6/96, Forschungszentrum Informatik, Karlsruhe, 1996.

[10] Claus Lewerentz and Thomas Lindner.Case Study Production Cell: A Comparative Study in Formal Software Development. FZI-Publication 1/94, Forschungszentrum Informatik, Karlsruhe, 1994.

[11] C. Daws, A. Olivero, S. Tripakis and S. Yovine. The tool Kronos. In Rajeev Alur, Thomas A. Henzinger and Eduardo D. Sontag, editors, *Hybrid Systems III (LNCS 1066)*, pages 208-219, Springer-Verlag, Berlin, 1996.

[12] Martin Abadi and Leslie Lamport. An old-fashioned recipe for real time. In J. W. de Bakker, C. Huizing, W. P. de Roever and G. Rozenberg, editors, *Real Time: Theory in Practice (LNCS 600)*, pages 1-27, Springer-Verlag, Berlin, 1992.

[13] Johan Brengtsson, Kim Larsen, Frederik Larsson, Paul Peterson und Wang Yi. Uppal - a tool suite for verifikation of real-time systems. In Rajeev Alur, Thomas A. Henzinger and Eduardo D. Sontag, editors, *Hybrid Systems III (LNCS 1066)*, pages 232-243, Springer-Verlag, Berlin, 1996.

[14] Robin Milner. Communication and Concurrency. Prentice Hall, Hemel Hemlstead, 1989.

[15] C. A. R. Hoare. Communicating Sequential Processes. Prentice Hall, Hemel Hemlstead, 1985.

[16] Heinrich Rust. Modelling a Production Cell Component as a Hybrid Automaton: A Case Study. Unveröffentlichter Bericht, BTU Cottbus, Lehrstuhl Software-Systemtechnik, 1998.

[17] Dirk Beyer. Beschreibung der Modell-Fertigungsanlage. Unveröffentlichter Bericht, BTU Cottbus, Lehrstuhl Software-Systemtechnik, 1998.

[18] Heinrich Rust. A Formal Definition for a Modular Hybrid Modelling Notation. Unveröffentlichter Bericht, BTU Cottbus, Lehrstuhl Software-Systemtechnik, 1998.

[19] Dirk Beyer and Heinrich Rust. Modelling a Production Cell as a Distributed Real-Time System with Cottbus Timed Automata. In *Proceedings of FBT'98*, pages 148-159, Shaker-Verlag, Aachen, 1998.

[20] Heide Balzert. Methoden der objektorientierten Systemanalyse. BI-Wissenschaftsverlag, Mannheim, 1995.

[21] Bjarne Stroustrup. Die C++ Programmiersprache. Addison-Wesley, Bonn, 1998.

[22] Arne Frick, Walter Zimmer and Wolf Zimmermann. Konstruktion robuster und flexibler Klassenbibliotheken. In *Informatik – Forschung*

und Entwicklung (Bd. 11), pages 168-178, 1996.

[23] Borland GmbH. Borland C++ 4.5 – Programmierhandbuch. Borland GmbH, Langen, 1994.

[24] Alexander Stepanov and Meng Lee. The Standard Template Library. Hewlett-Packard Laboratories, Palo Alto, 1995.

[25] Nicolai Josuttis. Die C++ Standardbibliothek: Eine detaillierte Einführung in die vollständige ANSI/ISO-Schnittstelle. Addison-Wesley, Bonn, 1996.

[26] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, 1994.

[27] Mats Henricson and Erik Nyquist. Programming in C++: Rules and Recommendations. Ellemtel Telecommunication Systems Laboratories, Älvsjö, 1992.

[28] Niklaus Wirth. Compilerbau. B. G. Teubner, Stuttgart, 1986.

[29] Vern Paxson. Flex: A fast Scanner Generator. User Guide, University of California, Berkley, 1995.

[30] Charles Donnelly and Richard Stallman. Bison: The YACC-compatible Parser Generator. User Guide, Free Software Foundation, Cambridge, 1995.

[31] Alfred V. Aho and Jeffrey D. Ullman. The Theory of Parsing, Translation, and Compiling. Volume I: Parsing. Prentice-Hall, London, 1972.

[32] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *ECOOP '93 Conference Proceedings (LNCS)*, Springer-Verlag, Berlin, 1993.

[33] Rajeev Alur et al. The algorithmic analysis of hybrid systems. In *Theoretical Computer Science 138*, pages 3-34, 1995.

[34] Karin Erni and Claus Lewerentz. Applying Design-Metrics to Object-Oriented Frameworks. In *Proc. 3rd Intl Software Metrics Symposium*, pages 64-74, IEEE CS Press, Los Alamitos, 1996.

ANHANG A

Modellierung eines Transportbandes mittels CTA (Fallstudie)

```
-- This model is after transformations for input signals.
-- We need it as CTA*-model to test HyTech code generation.
-- Interesting issues are in testing automata.

-- A hybrid automaton model of a belt of the production cell.

-- This module describes the continuous movement of the belt.
--   It defines the derivative of the position of the plate
--   at every moment.
--   It has to listen for the „go“ and the „stop“-signal
--   as well as for the „load“ and the „unload“-signals.
--   For a loaded and moving belt, the plate position changes,
--   For an unloaded or stopped belt, this is not the case.
--   Additionally, the transfer of the plate from belt to the
--   input belt is modeled.
MODULE MBeltMovement
{
  INPUT
    -- Start the belt.
    go: SYNC;
    -- Stop the belt.
    stop: SYNC;
    -- Belt is being loaded.
    in: SYNC;
    -- Distance to input belt.
    distance_to_in: CONST;
    -- Radius of plate.
    plate_radius: CONST;
    -- Length of belt.
    length: CONST;

  OUTPUT
    -- This signal is generated: Plate starts to leave belt.
    out: SYNC;
    -- Position of the plate in the belt.
    pos: ANALOG;

  LOCAL
    -- Speed for movement.
    speed=1: CONST;

  -- Initially, we assume the belt is stopped and unloaded.
  INITIALIZATION
  {
    STATE = stopped_and_unloaded;
  }
  AUTOMATON
  {
    -- The belt is stopped and unloaded.
    STATE stopped_and_unloaded
    {
      -- There is no piece which might move.
      DERIV
```

```
    {
      DER(pos) = 0;
    }

    -- We react on a start signal.
    TRANS moving_and_unloaded
    {
      SYNC ?go;
    }

    -- We react on a load signal.
    TRANS stopped_and_loaded
    {
      SYNC ?in;
      DO
      {
        pos := - distance_to_in - plate_radius;
      }
    }
  }

  -- The belt is moving and unloaded.
  STATE moving_and_unloaded
  {
    -- There is no piece which might move.
    DERIV
    {
      DER(pos) = 0;
    }

    -- We react on a stop signal.
    TRANS stopped_and_unloaded
    {
      SYNC ?stop;
    }

    -- We react on a load signal.
    TRANS moving_and_loaded
    {
      SYNC ?in;
      DO
      {
        pos := - distance_to_in - plate_radius;
      }
    }
  }

  -- The belt is stopped and loaded.
  STATE stopped_and_loaded
  {
    -- The piece does not move.
    DERIV
    {
      DER(pos) = 0;
    }

    -- We react on a go signal.
    TRANS moving_and_loaded
    {
      SYNC ?go;
    }
  }

  -- The belt is moving and loaded.
  STATE moving_and_loaded
  {
```

```
        -- We may stay here until the plate reaches the end.
        INV
        {
          pos < length-plate_radius;
        }
        -- In this location, a piece moves with the
        --   given speed.
        DERIV
        {
          DER(pos) = speed;
        }

        -- We react on a stop signal
        TRANS stopped_and_loaded
        {
          SYNC ?stop;
        }

        -- We generate the transfer signal.
        TRANS moving_and_unloaded
        {
          GUARD
          {
            pos = length - plate_radius;
          }
          SYNC !out;
        }
      }
    }
}


-- This module computes the output of a sensor on the belt.
MODULE MSensor
{
  INPUT
    in: SYNC;
    pos: ANALOG;
    sens_pos: CONST;

  OUTPUT
    sens_on: SYNC;
    sens_off: SYNC;

  LOCAL
    -- Some local constants to model the range of sensor.
    sensor_radius_min = 3: CONST;
    sensor_radius_max = 5: CONST;

  -- Initially, we assume to be in the off-state after the sensor
  --   and the plate is not on the belt.
  INITIALIZATION
  {
    STATE = off_after;
  }
  AUTOMATON
  {
    -- In this state, the sensor is off because the plate is too far
    --   before the sensor.
    STATE off_before
    {
      -- We stay here as long as the plate has not
      --   reached the minimal range of the sensor.
      INV
      {
        pos <= sens_pos - sensor_radius_min;
```

```
        }
        -- After the plate has reached the maximal sensor range, we
        --   can generate an on-signal and go to the on-state.
        TRANS on
        {
          GUARD
          {
            pos >= sens_pos - sensor_radius_max;
          }
          SYNC !sens_on;
        }
      }

      -- We can stay here as long as we are in the on-state, that is,
      --   as long as the plate is in the maximal sensor range.
      STATE on
      {
        INV
        {
          pos <= sens_pos + sensor_radius_max;
        }

        -- We can leave this state whenever the plate leaves the
        --   minimal sensor range.
        TRANS off_after
        {
          GUARD
          {
            pos >= sens_pos + sensor_radius_min;
          }
          SYNC !sens_off;
        }
      }

      -- This location represents the situations where the plate
      --   is far after the sensor.
      STATE off_after
      {
        -- We leave this state when another plate is put onto
        --   the belt.
        TRANS off_before
        {
          SYNC ?in;
        }
      }
    }
}

-- Hardware description of the belt.
MODULE MBeltModel
{
  INPUT
    -- Interface signals from controller.
    go: SYNC;
    stop: SYNC;
    -- Interface signal from environment.
    in: SYNC;
    -- Radius of Plate.
    plate_radius: CONST;

  OUTPUT
    -- Sensor outputs.
    sens_A_on: SYNC;
    sens_A_off: SYNC;
    sens_B_on: SYNC;
    sens_B_off: SYNC;
```

```
      -- Plate starts to leave this belt.
      out: SYNC;

   LOCAL
      -- Position of plate on belt.
      pos: ANALOG;
      -- Distance to input belt.
      distance_to_in=1: CONST;
      -- Length of belt.
      length=80: CONST;
 sensA_pos=30: CONST;
 sensB_pos=50: CONST;


   -- Instantiation of the needed modules.
   INST iBeltMovement
   FROM MBeltMovement
   WITH
   {
      go AS go;
      stop AS stop;
      in AS in;
      out AS out;
      pos AS pos;
      distance_to_in AS distance_to_in;
      length AS length;
      plate_radius AS plate_radius;
   }

   -- The difference between iSensorA and iSensorB is
   --   only the position.
   INST iSensorA
   FROM MSensor
   WITH
   {
      in AS in;
      pos AS pos;
      sens_pos AS sensA_pos;
      sens_on AS sens_A_on;
      sens_off AS sens_A_off;
   }

   INST iSensorB
   FROM MSensor
   WITH
   {
      in AS in;
      pos AS pos;
      sens_pos AS sensB_pos;
      sens_on AS sens_B_on;
      sens_off AS sens_B_off;
   }
}

-- This module describes the controller of belt.
MODULE MBeltController
{
   INPUT
      -- Receive signal when plate has reached sensorA or sensorB.
      sens_A_on: SYNC;
      sens_B_on: SYNC;
      -- Get the stop-transfer-signal from output belt.
      stop_out_transfer: SYNC;

   OUTPUT
      -- Generate the stop-transfer-signal for input belt.
```

```
   stop_in_transfer: SYNC;
   -- Generate the start and stop signals for the motors.
   go: SYNC;
   stop: SYNC;

MULTREST
   -- Synchronize on start-transfer-signal with input belt.
   start_in_transfer: SYNC;
   -- Synchronize on start-transfer-signal with output belt.
   start_out_transfer: SYNC;

INITIALIZATION
{
   STATE = start;
}
AUTOMATON
{
   -- Wait for a start-transfer-signal.
   STATE start
   {
      -- Wait for start of transfer and start belt.
      TRANS in_transfer
      {
         SYNC #start_in_transfer THEN !go;
      }
   }

   -- Wait till plate reaches sensorA.
   STATE in_transfer
   {
      -- Wait till plate reaches sensorA and tell input belt
      --   that transfer can stop.
      TRANS internal_transfer
      {
         SYNC ?sens_A_on THEN !stop_in_transfer;
      }
   }

   -- Wait till plate reaches sensorB.
   STATE internal_transfer
   {
      TRANS wait_for_out
      {
         -- Wait till plate reaches sensorB and stop belt.
         SYNC ?sens_B_on THEN !stop;
      }
   }

   -- Wait till output belt wants to take the plate.
   STATE wait_for_out
   {
      -- Wait for start-signal of transfer to
      --    output belt and start belt again.
      TRANS out_transfer
      {
         SYNC #start_out_transfer THEN !go;
      }
   }

   -- Transfer the plate to output belt.
   STATE out_transfer
   {
      -- Wait till output belt acknowledges receipt and stop belt.
      TRANS start
      {
         -- We finished the control circle.
```

```
            SYNC ?stop_out_transfer THEN !stop;
          }
        }
      }
    }
}

-- This module models the neighbour producing plates.
MODULE MBeltInContext
{
  INPUT
    stop_in_transfer: SYNC;

  OUTPUT
    in: SYNC;

  MULTREST
    start_in_transfer: SYNC;

  INITIALIZATION
  {
    STATE = start;
  }
  AUTOMATON
  {
    -- Wait some time, then try to start a transfer from
    --    the input belt to modeled belt.
    STATE start
    {
      -- After some time, send a start-transfer-signal
      --   to input belt.
      TRANS in_transfer
      {
        SYNC #start_in_transfer;
      }
    }

    -- Accomplish a transfer from input belt to belt.
    STATE in_transfer
    {
      -- After some indeterminate time, output a load-message.
      TRANS wait_for_stop
      {
        SYNC !in;
      }
    }

    -- Wait for message about the accomplished transfer.
    STATE wait_for_stop
    {
      TRANS start
      {
         SYNC ?stop_in_transfer;
      }
    }
  }
}

MODULE MBeltOutContext
{
  INPUT
    out: SYNC;

  OUTPUT
    stop_out_transfer: SYNC;

  MULTREST
```

```
    start_out_transfer: SYNC;

  INITIALIZATION
  {
    STATE = wait_for_plate;
  }
  AUTOMATON
  {
    -- Wait till belt wants to transfer to output belt.
    STATE wait_for_plate
    {
      -- When belt wants to start the transfer, allow it, but
      --   possibly after some time.
      TRANS get_from_belt
      {
        SYNC #start_out_transfer;
      }
    }

    -- Get plate from belt.
    STATE get_from_belt
    {
      TRANS wait_rest
      {
        SYNC ?out;
      }
    }

    -- Wait some time until plate is sure to have left belt,
    --  e.g., because it has reached a sensor in output belt.
    STATE wait_rest
    {
      -- Tell belt, after some time, that transfer has been
      --  accomplished, and continue the loop.
      TRANS wait_for_plate
      {
        SYNC !stop_out_transfer;
      }
    }
  }
}

-- This module describes the context of belt.
MODULE MBeltContext
{
  INPUT
    -- This signal is received by output belt from belt1
    --  when the plate leaves belt.
    out: SYNC;
    -- This signal is received from input belt when the transfer
    --  has been accomplished.
    stop_in_transfer: SYNC;

  OUTPUT
    -- This signal is generated by input belt when the plate
    --  leaves it.
    in: SYNC;
    -- This signal is generated by output belt when the transfer
    --  has been accomplished.
    stop_out_transfer: SYNC;

  MULTREST
    -- On this signal synchronize the input belt and
    --  the modeled belt when a transfer may start.
    start_in_transfer: SYNC;
    -- On this signal synchronize the modeled belt and
```

```
      --  the output belt when a transfer may start.
      start_out_transfer: SYNC;


   -- Here we instantiate the two context modules.
   INST iBeltInContext
   FROM MBeltInContext
   WITH
   {
      in AS in;
      start_in_transfer AS start_in_transfer;
      stop_in_transfer AS stop_in_transfer;
   }

   INST iBeltOutContext
   FROM MBeltOutContext
   WITH
   {
      out AS out;
      start_out_transfer AS start_out_transfer;
      stop_out_transfer AS stop_out_transfer;
   }
}

-- An observer automaton which helps to test if time passes between
--  two signals.
MODULE MTestTime
{
   -- Observe if time passes between signal begin and signal end.
   INPUT
      begin: SYNC;
      end: SYNC;
   LOCAL
      -- Time in critical situation.
      critical_time: CLOCK;

   -- Initially, we assume that we are in wait_for_begin state.
   INITIALIZATION
   {
      STATE = wait_for_begin;
   }

   AUTOMATON
   {
      -- Wait till begin signal for transition.
      STATE wait_for_begin
      {
         TRANS wait_for_end
         {
            SYNC ?begin;
            DO
            {
               critical_time := 0;
            }
         }

         TRANS error
         {
            SYNC ?end;
         }
      }

      -- Wait till end signal for transition.
      STATE wait_for_end
      {
         TRANS wait_for_begin
```

```
        {
          SYNC ?end;
          DO
          {
            critical_time := 0;
          }
        }

        TRANS error
        {
          SYNC ?begin;
        }
      }
      STATE error
      {
        TRANS error
        {
          SYNC ?begin;
        }

        TRANS error
        {
          SYNC ?end;
        }
      }
    }
}

-- A testing automaton which helps to check that in time intervals
--   between ‚in‘ and ‚stop_in_transfer‘ and in time intervals
--   between ‚out‘ and ‚stop_out_transfer‘, the belt is
--   always moving.
MODULE MBeltTest
{
  INPUT
    go: SYNC;
    stop: SYNC;
    in: SYNC;
    out: SYNC;
    stop_in_transfer: SYNC;
    stop_out_transfer: SYNC;

  -- Check if we are in an in-transfer.
  INST iTestIn
  FROM MTestTime
  WITH
  {
    begin AS in;
    end AS stop_in_transfer;
  }

  -- Check if we are in an out-transfer.
  INST iTestOut
  FROM MTestTime
  WITH
  {
    begin AS out;
    end AS stop_out_transfer;
  }

  -- Check if the belt is going or stopped.
  INST iTestGoingStopped
  FROM MTestTime
  WITH
  {
    begin AS go;
```

```
    end AS stop;
  }
}

-- This is the top module.  We do not have to instantiate this.
MODULE MBeltSystem
{
  -- No interface.  We have only local identifiers.
  LOCAL
    go: SYNC;
    stop: SYNC;
    in: SYNC;
    out: SYNC;
    sens_A_on: SYNC;
    sens_B_on: SYNC;
    start_in_transfer: SYNC;
    stop_in_transfer: SYNC;
    start_out_transfer: SYNC;
    stop_out_transfer: SYNC;
    plate_radius=5: CONST;

  -- Instantiation of submodules.
  INST iBeltModel
  FROM MBeltModel
  WITH
  {
    go AS go;
    stop AS stop;
    in AS in;
    out AS out;
    sens_A_on AS sens_A_on;
    sens_B_on AS sens_B_on;
    plate_radius AS plate_radius;
  }

  INST iBeltController
  FROM MBeltController
  WITH
  {
    go AS go;
    stop AS stop;
    sens_A_on AS sens_A_on;
    sens_B_on AS sens_B_on;
    start_in_transfer AS start_in_transfer;
    stop_in_transfer AS stop_in_transfer;
    start_out_transfer AS start_out_transfer;
    stop_out_transfer AS stop_out_transfer;
  }

  INST iBeltContext
  FROM MBeltContext
  WITH
  {
    in AS in;
    out AS out;
    start_in_transfer AS start_in_transfer;
    stop_in_transfer AS stop_in_transfer;
    start_out_transfer AS start_out_transfer;
    stop_out_transfer AS stop_out_transfer;
  }

  INST iBeltTest
  FROM MBeltTest
  WITH
  {
    go AS go;
```

```
      stop AS stop;
      in AS in;
      out AS out;
      stop_in_transfer AS stop_in_transfer;
      stop_out_transfer AS stop_out_transfer;
    }
}
```

ANHANG B

Transportbandmodell in HyTech-Notation

Das beschriebene CTA-Modell kann zur Nutzung der Analysemöglichkeiten von HyTech in die HyTech-Notation übersetzt werden. Der dabei vom Werkzeug generierte Text wird im folgenden aufgelistet. Am Ende des HyTech-Dokuments befinden sich die Anweisungen zur Analyse des Modells.

```
-- HyTech-code generated from CTA-Module „MBeltSystem“.

define(MBeltSystem_iBeltModel_distance_to_in, 1)
define(MBeltSystem_iBeltModel_iBeltMovement_speed, 1)
define(MBeltSystem_iBeltModel_iSensorA_sensor_radius_max, 5)
define(MBeltSystem_iBeltModel_iSensorA_sensor_radius_min, 3)
define(MBeltSystem_iBeltModel_iSensorB_sensor_radius_max, 5)
define(MBeltSystem_iBeltModel_iSensorB_sensor_radius_min, 3)
define(MBeltSystem_iBeltModel_length, 80)
define(MBeltSystem_iBeltModel_sensA_pos, 30)
define(MBeltSystem_iBeltModel_sensB_pos, 50)
define(MBeltSystem_plate_radius, 5)

var
 MBeltSystem_iBeltController_then_in_transferClock: clock;
 MBeltSystem_iBeltController_then_internal_transferClock: clock;
 MBeltSystem_iBeltController_then_out_transferClock: clock;
 MBeltSystem_iBeltController_then_startClock: clock;
 MBeltSystem_iBeltController_then_wait_for_outClock: clock;
 MBeltSystem_iBeltModel_pos: analog;
 MBeltSystem_iBeltTest_iTestGoingStopped_critical_time: clock;
 MBeltSystem_iBeltTest_iTestIn_critical_time: clock;
 MBeltSystem_iBeltTest_iTestOut_critical_time: clock;

automaton a1
 synclabs:
  MBeltSystem_in,
  MBeltSystem_start_in_transfer,
  MBeltSystem_stop_in_transfer;

 initially MBeltSystem_iBeltContext_iBeltInContext_start;

 loc MBeltSystem_iBeltContext_iBeltInContext_in_transfer:
  while
   True
  wait
  {

  }
  when
   True
   sync MBeltSystem_in
   goto MBeltSystem_iBeltContext_iBeltInContext_wait_for_stop;

 loc MBeltSystem_iBeltContext_iBeltInContext_start:
  while
   True
```

```
   wait
   {

   }
   when
    True
    sync MBeltSystem_start_in_transfer
    goto MBeltSystem_iBeltContext_iBeltInContext_in_transfer;

 loc MBeltSystem_iBeltContext_iBeltInContext_wait_for_stop:
   while
    True
   wait
   {

   }
   when
    True
    sync MBeltSystem_stop_in_transfer
    goto MBeltSystem_iBeltContext_iBeltInContext_start;

end

automaton a2
 synclabs:
   MBeltSystem_out,
   MBeltSystem_start_out_transfer,
   MBeltSystem_stop_out_transfer;

 initially MBeltSystem_iBeltContext_iBeltOutContext_wait_for_plate;

 loc MBeltSystem_iBeltContext_iBeltOutContext_get_from_belt:
   while
    True
   wait
   {

   }
   when
    True
    sync MBeltSystem_out
    goto MBeltSystem_iBeltContext_iBeltOutContext_wait_rest;

 loc MBeltSystem_iBeltContext_iBeltOutContext_wait_for_plate:
   while
    True
   wait
   {

   }
   when
    True
    sync MBeltSystem_start_out_transfer
    goto MBeltSystem_iBeltContext_iBeltOutContext_get_from_belt;

 loc MBeltSystem_iBeltContext_iBeltOutContext_wait_rest:
   while
    True
   wait
   {

   }
   when
    True
    sync MBeltSystem_stop_out_transfer
    goto MBeltSystem_iBeltContext_iBeltOutContext_wait_for_plate;
```

```
end

automaton a3
 synclabs:
  MBeltSystem_go,
  MBeltSystem_sens_A_on,
  MBeltSystem_sens_B_on,
  MBeltSystem_start_in_transfer,
  MBeltSystem_start_out_transfer,
  MBeltSystem_stop,
  MBeltSystem_stop_in_transfer,
  MBeltSystem_stop_out_transfer;

 initially MBeltSystem_iBeltController_start;

 loc MBeltSystem_iBeltController_in_transfer:
  while
   True
  wait
  {

  }
  when
   True
   sync MBeltSystem_sens_A_on
   do
   {
    MBeltSystem_iBeltController_then_in_transferClock‘ = 0
   }
   goto MBeltSystem_iBeltController_then_in_transfer;

 loc MBeltSystem_iBeltController_internal_transfer:
  while
   True
  wait
  {

  }
  when
   True
   sync MBeltSystem_sens_B_on
   do
   {
    MBeltSystem_iBeltController_then_internal_transferClock‘ = 0
   }
   goto MBeltSystem_iBeltController_then_internal_transfer;

 loc MBeltSystem_iBeltController_out_transfer:
  while
   True
  wait
  {

  }
  when
   True
   sync MBeltSystem_stop_out_transfer
   do
   {
    MBeltSystem_iBeltController_then_out_transferClock‘ = 0
   }
   goto MBeltSystem_iBeltController_then_out_transfer;

 loc MBeltSystem_iBeltController_start:
  while
```

```
   True
  wait
  {

  }
  when
   True
   sync MBeltSystem_start_in_transfer
   do
   {
    MBeltSystem_iBeltController_then_startClock‘ = 0
   }
   goto MBeltSystem_iBeltController_then_start;

 loc MBeltSystem_iBeltController_then_in_transfer:
  while
   MBeltSystem_iBeltController_then_in_transferClock = 0
  wait
  {

  }
  when
   True
   sync MBeltSystem_stop_in_transfer
   goto MBeltSystem_iBeltController_internal_transfer;

 loc MBeltSystem_iBeltController_then_internal_transfer:
  while
   MBeltSystem_iBeltController_then_internal_transferClock = 0
  wait
  {

  }
  when
   True
   sync MBeltSystem_stop
   goto MBeltSystem_iBeltController_wait_for_out;

 loc MBeltSystem_iBeltController_then_out_transfer:
  while
   MBeltSystem_iBeltController_then_out_transferClock = 0
  wait
  {

  }
  when
   True
   sync MBeltSystem_stop
   goto MBeltSystem_iBeltController_start;

 loc MBeltSystem_iBeltController_then_start:
  while
   MBeltSystem_iBeltController_then_startClock = 0
  wait
  {

  }
  when
   True
   sync MBeltSystem_go
   goto MBeltSystem_iBeltController_in_transfer;

 loc MBeltSystem_iBeltController_then_wait_for_out:
  while
   MBeltSystem_iBeltController_then_wait_for_outClock = 0
  wait
```

```
  {

  }
  when
   True
   sync MBeltSystem_go
   goto MBeltSystem_iBeltController_out_transfer;

 loc MBeltSystem_iBeltController_wait_for_out:
  while
   True
  wait
  {

  }
  when
   True
   sync MBeltSystem_start_out_transfer
   do
   {
    MBeltSystem_iBeltController_then_wait_for_outClock‘ = 0
   }
   goto MBeltSystem_iBeltController_then_wait_for_out;

end

automaton a4
 synclabs:
  MBeltSystem_go,
  MBeltSystem_in,
  MBeltSystem_out,
  MBeltSystem_stop;

 initially MBeltSystem_iBeltModel_iBeltMovement_stopped_and_unloaded;

 loc MBeltSystem_iBeltModel_iBeltMovement_moving_and_loaded:
  while
   MBeltSystem_iBeltModel_pos < MBeltSystem_iBeltModel_length - MBeltSystem_plate_radius
  wait
  {
   dMBeltSystem_iBeltModel_pos = MBeltSystem_iBeltModel_iBeltMovement_speed
  }
  when
   True
   sync MBeltSystem_stop
   goto MBeltSystem_iBeltModel_iBeltMovement_stopped_and_loaded;

  when
      MBeltSystem_iBeltModel_pos = MBeltSystem_iBeltModel_length - MBeltSystem_plate_radius
   sync MBeltSystem_out
   goto MBeltSystem_iBeltModel_iBeltMovement_moving_and_unloaded;

 loc MBeltSystem_iBeltModel_iBeltMovement_moving_and_unloaded:
  while
   True
  wait
  {
   dMBeltSystem_iBeltModel_pos = 0
  }
  when
   True
   sync MBeltSystem_stop
   goto MBeltSystem_iBeltModel_iBeltMovement_stopped_and_unloaded;

  when
   True
```

```
    sync MBeltSystem_in
    do
    {
      MBeltSystem_iBeltModel_pos‘ = - MBeltSystem_iBeltModel_distance_to_in - MBeltSystem_plate_radius
    }
    goto MBeltSystem_iBeltModel_iBeltMovement_moving_and_loaded;

 loc MBeltSystem_iBeltModel_iBeltMovement_stopped_and_loaded:
  while
    True
  wait
  {
    dMBeltSystem_iBeltModel_pos = 0
  }
  when
    True
    sync MBeltSystem_go
    goto MBeltSystem_iBeltModel_iBeltMovement_moving_and_loaded;

 loc MBeltSystem_iBeltModel_iBeltMovement_stopped_and_unloaded:
  while
    True
  wait
  {
    dMBeltSystem_iBeltModel_pos = 0
  }
  when
    True
    sync MBeltSystem_go
    goto MBeltSystem_iBeltModel_iBeltMovement_moving_and_unloaded;

  when
    True
    sync MBeltSystem_in
    do
    {
      MBeltSystem_iBeltModel_pos‘ = - MBeltSystem_iBeltModel_distance_to_in - MBeltSystem_plate_radius
    }
    goto MBeltSystem_iBeltModel_iBeltMovement_stopped_and_loaded;

end

automaton a5
 synclabs:
  MBeltSystem_iBeltModel_sens_A_off,
  MBeltSystem_in,
  MBeltSystem_sens_A_on;

 initially MBeltSystem_iBeltModel_iSensorA_off_after;

 loc MBeltSystem_iBeltModel_iSensorA_off_after:
  while
    True
  wait
  {

  }
  when
    True
    sync MBeltSystem_in
    goto MBeltSystem_iBeltModel_iSensorA_off_before;

 loc MBeltSystem_iBeltModel_iSensorA_off_before:
  while
                              MBeltSystem_iBeltModel_pos    <=    MBeltSystem_iBeltModel_sensA_pos    -
MBeltSystem_iBeltModel_iSensorA_sensor_radius_min
```

```
  wait
  {

  }
  when
                              MBeltSystem_iBeltModel_pos  >=  MBeltSystem_iBeltModel_sensA_pos  -
MBeltSystem_iBeltModel_iSensorA_sensor_radius_max
   sync MBeltSystem_sens_A_on
   goto MBeltSystem_iBeltModel_iSensorA_on;

 loc MBeltSystem_iBeltModel_iSensorA_on:
  while
                           MBeltSystem_iBeltModel_pos  <=  MBeltSystem_iBeltModel_sensA_pos  +
MBeltSystem_iBeltModel_iSensorA_sensor_radius_max
  wait
  {

  }
  when
                              MBeltSystem_iBeltModel_pos  >=  MBeltSystem_iBeltModel_sensA_pos  +
MBeltSystem_iBeltModel_iSensorA_sensor_radius_min
   sync MBeltSystem_iBeltModel_sens_A_off
   goto MBeltSystem_iBeltModel_iSensorA_off_after;

end

automaton a6
 synclabs:
  MBeltSystem_iBeltModel_sens_B_off,
  MBeltSystem_in,
  MBeltSystem_sens_B_on;

 initially MBeltSystem_iBeltModel_iSensorB_off_after;

 loc MBeltSystem_iBeltModel_iSensorB_off_after:
  while
   True
  wait
  {

  }
  when
   True
   sync MBeltSystem_in
   goto MBeltSystem_iBeltModel_iSensorB_off_before;

 loc MBeltSystem_iBeltModel_iSensorB_off_before:
  while
                           MBeltSystem_iBeltModel_pos  <=  MBeltSystem_iBeltModel_sensB_pos  -
MBeltSystem_iBeltModel_iSensorB_sensor_radius_min
  wait
  {

  }
  when
                              MBeltSystem_iBeltModel_pos  >=  MBeltSystem_iBeltModel_sensB_pos  -
MBeltSystem_iBeltModel_iSensorB_sensor_radius_max
   sync MBeltSystem_sens_B_on
   goto MBeltSystem_iBeltModel_iSensorB_on;

 loc MBeltSystem_iBeltModel_iSensorB_on:
  while
                           MBeltSystem_iBeltModel_pos  <=  MBeltSystem_iBeltModel_sensB_pos  +
MBeltSystem_iBeltModel_iSensorB_sensor_radius_max
  wait
  {
```

```
  }
  when
                                MBeltSystem_iBeltModel_pos   >=   MBeltSystem_iBeltModel_sensB_pos   +
MBeltSystem_iBeltModel_iSensorB_sensor_radius_min
    sync MBeltSystem_iBeltModel_sens_B_off
    goto MBeltSystem_iBeltModel_iSensorB_off_after;

end

automaton a7
 synclabs:
  MBeltSystem_go,
  MBeltSystem_stop;

 initially MBeltSystem_iBeltTest_iTestGoingStopped_wait_for_begin;

 loc MBeltSystem_iBeltTest_iTestGoingStopped_error:
  while
   True
  wait
  {

  }
  when
   True
   sync MBeltSystem_go
   goto MBeltSystem_iBeltTest_iTestGoingStopped_error;

  when
   True
   sync MBeltSystem_stop
   goto MBeltSystem_iBeltTest_iTestGoingStopped_error;

 loc MBeltSystem_iBeltTest_iTestGoingStopped_wait_for_begin:
  while
   True
  wait
  {

  }
  when
   True
   sync MBeltSystem_go
   do
   {
    MBeltSystem_iBeltTest_iTestGoingStopped_critical_time‘ = 0
   }
   goto MBeltSystem_iBeltTest_iTestGoingStopped_wait_for_end;

  when
   True
   sync MBeltSystem_stop
   goto MBeltSystem_iBeltTest_iTestGoingStopped_error;

 loc MBeltSystem_iBeltTest_iTestGoingStopped_wait_for_end:
  while
   True
  wait
  {

  }
  when
   True
   sync MBeltSystem_stop
   do
```

```
    {
      MBeltSystem_iBeltTest_iTestGoingStopped_critical_time‘ = 0
    }
    goto MBeltSystem_iBeltTest_iTestGoingStopped_wait_for_begin;

  when
    True
    sync MBeltSystem_go
    goto MBeltSystem_iBeltTest_iTestGoingStopped_error;

end

automaton a8
 synclabs:
  MBeltSystem_in,
  MBeltSystem_stop_in_transfer;

 initially MBeltSystem_iBeltTest_iTestIn_wait_for_begin;

 loc MBeltSystem_iBeltTest_iTestIn_error:
  while
    True
  wait
  {

  }
  when
    True
    sync MBeltSystem_in
    goto MBeltSystem_iBeltTest_iTestIn_error;

  when
    True
    sync MBeltSystem_stop_in_transfer
    goto MBeltSystem_iBeltTest_iTestIn_error;

 loc MBeltSystem_iBeltTest_iTestIn_wait_for_begin:
  while
    True
  wait
  {

  }
  when
    True
    sync MBeltSystem_in
    do
    {
      MBeltSystem_iBeltTest_iTestIn_critical_time‘ = 0
    }
    goto MBeltSystem_iBeltTest_iTestIn_wait_for_end;

  when
    True
    sync MBeltSystem_stop_in_transfer
    goto MBeltSystem_iBeltTest_iTestIn_error;

 loc MBeltSystem_iBeltTest_iTestIn_wait_for_end:
  while
    True
  wait
  {

  }
  when
    True
```

```
    sync MBeltSystem_stop_in_transfer
    do
    {
      MBeltSystem_iBeltTest_iTestIn_critical_time‘ = 0
    }
    goto MBeltSystem_iBeltTest_iTestIn_wait_for_begin;

  when
    True
    sync MBeltSystem_in
    goto MBeltSystem_iBeltTest_iTestIn_error;

end

automaton a9
 synclabs:
  MBeltSystem_out,
  MBeltSystem_stop_out_transfer;

 initially MBeltSystem_iBeltTest_iTestOut_wait_for_begin;

 loc MBeltSystem_iBeltTest_iTestOut_error:
  while
    True
  wait
  {

  }
  when
    True
    sync MBeltSystem_out
    goto MBeltSystem_iBeltTest_iTestOut_error;

  when
    True
    sync MBeltSystem_stop_out_transfer
    goto MBeltSystem_iBeltTest_iTestOut_error;

 loc MBeltSystem_iBeltTest_iTestOut_wait_for_begin:
  while
    True
  wait
  {

  }
  when
    True
    sync MBeltSystem_out
    do
    {
      MBeltSystem_iBeltTest_iTestOut_critical_time‘ = 0
    }
    goto MBeltSystem_iBeltTest_iTestOut_wait_for_end;

  when
    True
    sync MBeltSystem_stop_out_transfer
    goto MBeltSystem_iBeltTest_iTestOut_error;

 loc MBeltSystem_iBeltTest_iTestOut_wait_for_end:
  while
    True
  wait
  {

  }
```

```
   when
    True
    sync MBeltSystem_stop_out_transfer
    do
    {
     MBeltSystem_iBeltTest_iTestOut_critical_time‘ = 0
    }
    goto MBeltSystem_iBeltTest_iTestOut_wait_for_begin;

   when
    True
    sync MBeltSystem_out
    goto MBeltSystem_iBeltTest_iTestOut_error;

end

-- ************************************************************
-- **** Start of analysis commands ****
-- ************************************************************

-- We define a starting region and a region reachable from
--   the starting region.
var reg_start,
   reg_reachable,
   reg_interest,
   reg_test_error: region;

-- The starting region.
reg_start :=
    MBeltSystem_iBeltModel_pos >= MBeltSystem_iBeltModel_length+MBeltSystem_plate_radius
   & loc[a4] = MBeltSystem_iBeltModel_iBeltMovement_stopped_and_unloaded
   & loc[a5] = MBeltSystem_iBeltModel_iSensorA_off_after
   & loc[a6] = MBeltSystem_iBeltModel_iSensorB_off_after
   & loc[a1] = MBeltSystem_iBeltContext_iBeltInContext_start
   & loc[a2] = MBeltSystem_iBeltContext_iBeltOutContext_wait_for_plate
   & loc[a3] = MBeltSystem_iBeltController_start
   & MBeltSystem_iBeltTest_iTestGoingStopped_critical_time = 0
   & MBeltSystem_iBeltTest_iTestIn_critical_time = 0
   & MBeltSystem_iBeltTest_iTestOut_critical_time = 0
   & loc[a7] = MBeltSystem_iBeltTest_iTestGoingStopped_wait_for_begin
   & loc[a8] = MBeltSystem_iBeltTest_iTestIn_wait_for_begin
   & loc[a9] = MBeltSystem_iBeltTest_iTestOut_wait_for_begin
   ;

-- Define interesting region: plate is behind sensor B
--   and before the end of the belt.
reg_interest :=
    (MBeltSystem_iBeltModel_pos > MBeltSystem_iBeltModel_sensB_pos+MBeltSystem_plate_radius)
   & (MBeltSystem_iBeltModel_pos < MBeltSystem_iBeltModel_length-MBeltSystem_plate_radius);

-- Print start region.
prints „Start region:“;
print reg_start;
prints „“;

-- Print region of interest.
prints „Region of interest:“;
print reg_interest;
prints „“;

-- Compute from reg_start reachable region.
reg_reachable := reach forward from reg_start endreach;

-- Print reachable region.
prints „Reachable region:“;
print reg_reachable;
```

```
prints „“;

-- Print trace from start to region of interest.
if empty( reg_interest & reg_reachable )
then
   prints „No way to region of interest“;
else
   prints „Trace to region of interest:“;
   print trace to reg_interest using reg_reachable;
endif;

-- Use the testing automaton to check for erroneous configurations.

reg_test_error :=
       -- Wrong sequence of signals for transfer from bA to b1.
     loc[a7] = MBeltSystem_iBeltTest_iTestGoingStopped_error
       -- Wrong sequence of signals for transfer from b1 to bB.
   | loc[a8] = MBeltSystem_iBeltTest_iTestIn_error
       -- Wrong sequence of start/stop-signals.
   | loc[a9] = MBeltSystem_iBeltTest_iTestOut_error
       -- Belt stopped in critical time interval during
       --  transfer from bA to b1.
   | (  loc[a8] = MBeltSystem_iBeltTest_iTestIn_wait_for_end
     & loc[a7] = MBeltSystem_iBeltTest_iTestGoingStopped_wait_for_begin
     & MBeltSystem_iBeltTest_iTestIn_critical_time > 0
     & MBeltSystem_iBeltTest_iTestGoingStopped_critical_time > 0
     )
       -- Belt stopped in critical time interval during
       --  transfer from b1 to bB.
   | (  loc[a9] = MBeltSystem_iBeltTest_iTestOut_wait_for_end
     & loc[a7] = MBeltSystem_iBeltTest_iTestGoingStopped_wait_for_begin
     & MBeltSystem_iBeltTest_iTestOut_critical_time > 0
     & MBeltSystem_iBeltTest_iTestOut_critical_time > 0
     )
   ;

-- Check reachability of error region.
-- Print trace from start to region of interest.
if empty( reg_test_error & reg_reachable )
then
   prints „No way to error region of testing automaton.“;
else
   prints „Trace to error region of testing automaton:“;
   print trace to reg_test_error using reg_reachable;
endif;

-- End of description.
```

ANHANG C

Quelltexte

Um einen Einblick in die Implementierung zu geben, wurden im folgenden einige Quelldateien aufgeführt. In der Datei ctaLex.l ist die Definition des finiten Automaten enthalten, aus der der Scanner generiert wird. Die Beschreibung des Parsers ist in der Datei ctaYacc.y enthalten. Im Anschluß daran werden von den verwendeten Klassen die Schnittstellen-Dateien aufgelistet.