In *Tagungsband zur 5. Fachtagung Entwurf komplexer Automatisierungssysteme*, Seiten 243-264, Braunschweig, Mai 1997

Kompositionale Constraints hybrider Systeme

Peter Herrmann, Heiko Krumm
Fachbereich Informatik
Universität Dortmund
D-44221 Dortmund
Tel.:(02 31) 755 - 4836
Fax: (02 31) 755 - 4730
E-Mail: {herrmann|krumm}@ls4.informatik.uni-dortmund.de

Abstract: Modulare Spezifikationen hybrider Systeme modellieren häufig sehr direkt die Ressourcen der realen Systeme als Komponenten und die Verbindungen dazwischen als Kopplung. Wir plädieren hier dafür, reale Systeme zu Verifikations- und Analysezwecken zusätzlich auch in anderer Struktur zu modellieren, nämlich als Verbund einzelner Constraints zu im Systeminneren geforderten dynamischen Eigenschaften. Somit treten Constraints als modulare Komponenten einer Systemspezifikation auf. Gelingt eine geeignete Gliederung in Constraints, kann die Verifikation einer gewünschten Eigenschaft des Gesamtsystems sich auf die Betrachtung weniger Constraints beschränken, so daß die Komplexität der Verifikation erheblich reduziert wird und logische Zusammenhänge verständniserleichternd hervorgehoben werden. Diese hier sogenannte strukturierte Verifikation wird formal durch die Kompositionalität der Spezifikationsmodule möglich. Sie gewährleistet, daß Eigenschaften eines Moduls auch Eigenschaften eines einbettenden Systems sind. Wir stellen die Spezifikationssprache cTLA+ vor, die diese Konzepte unterstützt, und verdeutlichen ihre Anwendung an einem einfachen Beispiel.

Stichworte: Hybride Systeme, formale Spezifikation, Verifikation

1 Einleitung

Modulare Spezifikationstechniken unterstützen Beschreibungen, die aus Komponenten so zusammengesetzt sind, daß sie ausschließlich über wohldefinierte Schnittstellen mit ihrer Umgebung im System interagieren. Modularität erleichtert die deutlich ausgeprägte Strukturierung von Spezifikationen. Ferner öffnet

sie Möglichkeiten zur Wiederverwendung von Spezifikationskomponenten. Modulare Spezifikationstechniken heißen darüberhinaus kompositional, wenn die leichte Lesbarkeit und Verständlichkeit von Spezifikationen sowie ihre Analyse und formale Verifikation direkt davon profitieren, daß die modularen Bausteine Abstraktionen unterstützen. An die Stelle einer umfassenden und mit der Komplexität eines Gesamtsystems konfrontierten Behandlung kann eine Serie jeweils in der Komplexität erheblich reduzierter Teilbehandlungen treten.

Im wesentlichen können zwei Arten von Kompositionalität unterschieden werden, die wir hier Subsystem- und Teilsystem-Kompositionalität nennen. Unter Subsystem soll dabei ein Teilsystem verstanden werden, das hierarchisch so gekapselt werden kann, daß es zum umgebenden System hin als modulare Einheit auftritt. Nur als Teilsystem bezeichnen wir eine beliebige Untermenge der Komponenten eines Systems mit ihrer wechselseitigen Kopplung. Ein Teilsystem muß nicht notwendigerweise eingekapselt werden können. Bei einer Strukturierung eines Systems in Teilsysteme müssen die Teilsysteme nicht disjunkt sein. Dieselben Komponenten können in verschiedenen Teilsystemen enthalten sein.

- Die Subsystem-Kompositionalität unterstützt die Abstraktion vom inneren Aufbau einer Systemkomponente. Auch komplexere Subsysteme können in ihren für das Gesamtsystem relevanten Eigenschaften (z.B. in der Art eines Ersatzschaltbilds) so spezifiziert werden, daß die Komponentenspezifikation eine wesentlich geringere Komplexität als das detaillierte Subsystem besitzt. Eine formale Systemverifikation kann somit in zwei Phasen gegliedert werden. Zunächst werden die einzelnen Subsysteme jeweils separat behandelt und es wird nachgewiesen, daß die abstrakten Subsystemspezifikationen zutreffen. Erst die zweite Phase widmet sich der Behandlung des Gesamtsystems. Hier werden alle Subsysteme nur noch durch ihre abstrakten Spezifikationen vertreten.
- Die Teilsystem-Kompositionalität unterstützt die Abstraktion von übrigen Komponenten bei der Behandlung eines Gesamtsystems. Sie basiert darauf, daß Beiträge einzelner Komponenten zum Gesamtverhalten des Systems isoliert von anderen Systemkomponenten betrachtet werden können. So müssen in die Verifikation eines einzelnen Aspekts des Gesamtsystems nicht mehr alle Komponenten eingehen, sondern die Betrachtung kann sich auf ein aus wenigen Komponenten bestehendes Teilsystem konzentrieren. Die Teilsystem-Kompositionalität greift auch bei nicht-hierarchischen Struktu-

ren. Sie ermöglicht die Dekomposition bei der oben angesprochenen zweiten Phase einer Verifikation, der Behandlung des Gesamtsystems.

Beide Arten der Kompositionalität ergänzen sich und werden von unserem Ansatz unterstützt. Der vorliegende Beitrag konzentriert sich auf die Teilsystem-Kompositionalität.

Als modulare Komponenten der Spezifikationssprache werden Prozesse verwendet. Ein Prozeß ist ein offenes Subsystem, das ein aktives dynamisches Verhalten besitzt und mit seiner Umgebung, d.h. den übrigen Prozessen des Systems und der Systemumgebung, interagieren kann. Es unterliegt eine Sicht, die an der Spezifikationssprache LOTOS orientiert ist. Wie bei LOTOS lassen sich ressourcen- und constraintorientierte Prozesse [VSvS88] darstellen und zu Systemen koppeln:

- Systeme aus ressourcenorientierten Prozessen spiegeln Implementierungsstrukturen wider, d.h. ein Prozeß entspricht in der Regel auch einer modularen Einheit der Implementierung (z.B. einer Baugruppe). Die Kopplung der Prozesse modelliert die Kopplung der Implementierungseinheiten (z.B. durch Verbindungen).
- Systeme aus constraintorientierten Prozessen strukturieren ein System rein logisch. Ein Prozeß entspricht einer logischen Anforderung an das System (z.B. der Festlegung der erlaubten Reihenfolge einer Menge von Aktionen). Die Kopplung der Prozesse entspricht einer logischen UND- Verknüpfung der Anforderungen.

Dasselbe reale System kann sowohl ressourcen- als auch constraintorientiert modelliert werden. Darüberhinaus sind auch Mischformen möglich. Ein System kann z.B. so modelliert werden, daß es ressourcenorientiert in Implementierungseinheiten strukturiert ist, daß aber einzelne Implementierungseinheiten jeweils constraintorientiert durch eine Menge logischer Anforderungen repräsentiert sind.

Constraints hybrider Systeme können in verschiedene Typen klassifiziert werden. Im ereignisdiskreten Teil beschreiben Safety-Constraints zulässige Reihenfolgen und Parametrisierungen von Aktionen. Ein System erfüllt, wenn es in einem erlaubten Zustand verharrt, alle Safety-Anforderungen. Fortschrittsgarantien werden separiert davon durch Liveness-Constraints ausgedrückt. Sie

verbieten in der Regel, daß ein System in bestimmten Zuständen unendlich lange verharrt. Im realzeitbewerteten ereignisdiskreten System sind darüberhinaus Realzeit-Constraints wesentlich, die sich auf die Zeitdauer des Verharrens in Zuständen, d.h. die Zeit zwischen zwei Aktionen, beziehen. Es sind Constraints möglich, die die Aktivität durch Bezugnahme auf minimale Wartefristen drosseln, und solche, die durch Bezugnahme auf maximale Reaktionszeiten die Aktivität forcieren. Im kontinuierlichen Teil eines Systems bringen Constraints entsprechend einzelne Abhängigkeiten zum Ausdruck, denen kontinuierliche Größen unterliegen. Für unseren Ansatz ist es sehr wesentlich, daß einzelne konstituierende Eigenschaften eines Systems bei Bedarf auch in sehr feiner Granularität durch Constraint-Prozesse modular beschrieben werden können.

Weiterhin ist die Teilsystem-Kompositionalität wichtig. Die Kopplung von Constraint-Prozessen zu einem System entspricht der logischen UND-Verknüpfung der Prozesse. Die durch einen Prozeß ausgedrückte Eigenschaft ist direkt auch in jedem System vorhanden, das den Prozeß enthält. Um dies von der Spezifikationstechnik her zu gewährleisten, muß sichergestellt sein, daß sich einzelne Prozesse nicht widersprechen. Dies ist bei Safety-Constraints vergleichsweise einfach, bei Constraints, die Aktivitäten unbedingt forcieren, aber im allgemeinen unmöglich, denn eine unbedingt erzwungene Aktion könnte z.B. im Widerspruch zum Safety-Spielraum des Systems stehen. Wir beschränken uns deshalb auf die Unterstützung geeignet bedingter forcierender Constraints und geben Strategien an, mit denen gewünschte unbedingte Systemeigenschaften durch bedingte Constraints eingebracht werden können.

Da ein Teilsystem einer UND-Verknüpfung eines Teils dieser Prozesse entspricht, wird jede Eigenschaft, die von einem Teilsystem impliziert wird, auch vom Gesamtsystem impliziert. Die formale Verifikation dieser Eigenschaft kann sich also auf die Betrachtung des Teilsystems beschränken und ist somit in ihrer Komplexität u.U. wesentlich reduziert. Die Teilsystem-Kompositionalität unterstützt deshalb die strukturierte Verifikation. Gelingt es, die abstrakten Anforderungen an ein System in einzelne Eigenschaften so zu spalten, daß jede Eigenschaft durch ein echtes Teilsystem sichergestellt wird, so wird die Verifikation in eine Menge von Teilsystem-Verifikationen gegliedert.

Die Kompositionalität der in unserer Spezifikationstechnik spezifizierbaren Prozesse unterstützt diese Strukturierung der Verifikation von den formalen Vor-

aussetzungen her. Es ist allerdings im Einzelfall einer Anwendung offen, ob für ein spezielles System geeignete Eigenschaften, Prozesse und Teilsysteme gefunden werden können. Hier spielt nun der Ansatz der Constraintorientierung eine wichtige Rolle. In einem ressourcenorientiert gegliederten System können in der Regel kaum geeignete Strukturen gefunden werden. Wichtige Systemeigenschaften stellen sich dort oft nur dadurch ein, daß sehr viele Systemkomponenten einen Teilbeitrag dazu liefern. Das zur Verifikation der Eigenschaft benötigte Teilsystem wäre deshalb kaum weniger komplex als das Gesamtsystem. Wird das System aber constraintorientiert gegliedert, z.B. dadurch, daß jede Implementierungskomponente durch eine Menge von Constraints zu ihrem Verhalten eingebracht wird, dann lassen sich bedeutende Gewinne durch die strukturierte Verifikation erzielen.

Wir verwenden als Spezifikationssprache cTLA+ (compositional TLA+ [HMK96]). Es wurde zur Spezifikation und Verifikation ereignisdiskreter Systeme entwickelt (z.B. zur Modellierung von Datentransferprotokollen [HK95]). cTLA+ wird im Rahmen eines von der DFG im Schwerpunktprogramm KONDISK geförderten Projekts für die Behandlung hybrider Systeme erweitert. cTLA+ baut auf TLA (Temporal Logic of Actions [L94a]) auf, einer Linearzeit-Temporallogik zur Behandlung von Zustandstransitionssystemen und der zugeordneten Spezifikationssprache TLA+ [L94b]. cTLA+ ergänzt TLA+ um ein kompositionales Prozeßkonzept. Prozesse kapseln hierzu private Zustandsgrößen ein. Aktionen definieren Transitionsklassen. Sie können mit Datenparametern parametrisiert sein. Wie in der Sprache LOTOS werden Interaktionen zwischen Prozessen durch gemeinsame Aktionen modelliert, so daß Kommunikation über die Aktionsparameter beschrieben werden kann. Die formale Semantik von cTLA+ ist durch eine Abbildung definiert, die cTLA+- Spezifikationen in semantisch äquivalente Formeln der Temporallogik TLA überführt.

Die in cTLA+ eingeführten Realzeit-Erweiterungen und Konzepte zur Beschreibung kontinuierlicher Abhängigkeiten basieren ebenfalls auf TLA bzw. auf dafür vorgeschlagenen Lösungen [AL91b, L93]. Realzeit wird durch eine Zustandsgröße `now` repräsentiert, die von einer Aktion `tick` in sehr kleinen Schritten erhöht wird. Zur Modellierung kontinuierlicher Zustandsübergänge kann ein Prozeß eine Aktion `cont` enthalten, deren Parameter den kontinuierlichen Ein- und Ausgangsgrößen des Prozesses entsprechen. Die `cont`-Aktionen der Prozesse eines Systems sind auf Systemebene miteinander und mit der Uhr-Aktion `tick` gekoppelt. Mit cTLA+ wurde eine Anpassung dieser Konzepte an das

Prozeßkonzept vorgenommen und es wurden Lösungen zur Wahrung der Kompositionalität, insbesondere bei aktivitätsforcierenden Constraints, entwickelt.

Einige weitere Ansätze betrachten die kompositionale Spezifikation und Verifikation von Realzeit- und hybriden Systemen. Alur et al. schlagen die Benutzung hybrider Automaten vor [ACHH93, AHH93], die die Spezifikation kontinuierlicher Zustandsänderungen von Variablen im Rahmen diskreter Hauptzustände (Lokationen) erlauben. Die Komposition hybrider Systeme erfolgt durch Produktautomatenbildung. Zur Verifikation können automatentheoretische Methoden wie Modell-Checking verwendet werden. Benveniste [B96] spezifiziert hybride Systeme in einer Erweiterung der Spezifikationssprache VHDL und verifiziert mit Hilfe von Prozeßalgebren, wobei die Teilsystemkompositionalität von Systemen ausgenutzt wird. Nach Aussage des Autors können Spezifikationen nicht in ein zustandsorientiertes Modell überführt werden. Hooman et al. verwenden temporale Logik [H93, ZHK96]. Detaillierte Systeme werden in einer Zuweisungssyntax auf der Basis von Hoare-Tripeln spezifiziert. Abstraktere Systemeigenschaften werden in einer expliziten Realzeit-Temporallogik beschrieben. Systemverifikationen erfolgen durch Anwendung spezieller Deduktionsregeln. Dabei kann die Subsystemkompositionalität von Systemen ausgenutzt werden. Ähnliche Ansätze auf der Basis impliziter Realzeit-Temporallogiken verfolgen Quiwen und Weidong sowie Sintzoff [QW96, S96]. Abadi und Lamport schlagen dagegen eine kompositionale Methode vor, die auch Teilsystemkompositionalität berücksichtigt [AL91b, L93]. TLA wird dazu um eine virtuelle Uhrenvariable erweitert. Systeme können unter Verwendung eines speziellen Kompositionstheorems [AL93] miteinander gekoppelt werden. Im Gegensatz zu dem hier vorgestellten Ansatz garantiert die Komposition nicht die Erfüllbarkeit der zusammengesetzten Systeme, die explizit nachgewiesen werden muß.

In diesem Beitrag werden die theoretischen Grundlagen des Ansatzes nur kurz angesprochen. Insbesondere die formale Semantik von cTLA+, d.h. der Bezug zur temporalen Logik TLA, wird nur angerissen. Der Beitrag konzentriert sich auf Grundzüge und auf Aspekte der praktischen Anwendung, die anhand eines einfachen Beispiels beleuchtet werden. So wird im nächsten Kapitel zunächst das Beispiel vorgestellt. Die drei danach folgenden Kapitel erläutern cTLA+, die Realzeiterweiterungen und die Erweiterungen zu hybriden Systemen aus der Sicht der Anwendung heraus anhand von Modulen des Beispiels. Anschließend wird die Anwendung der strukturierten Verifikation am Beispiel gezeigt.

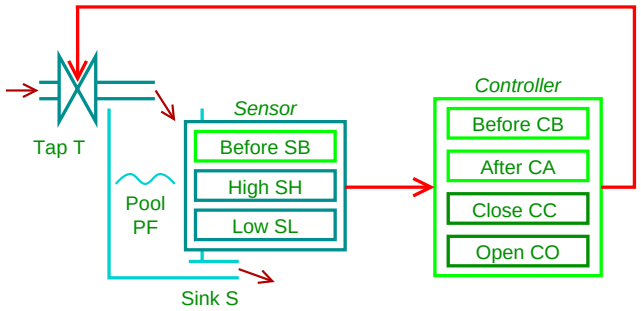

Abbildung 1: Das einfache Beispielsystem

2 Ein einfaches Beispiel

Das hier benutzte einfache Beispiel ist in Abb. 1 skizziert. Es besteht aus einem Flüssigkeitsbehälter PF. PF besitzt einen Abfluß S und einen über ein Ventil T kontrollierbaren Zufluß. Ein Sensor detektiert den aktuellen Pegel und signalisiert das Überschreiten des oberen und das Unterschreiten des unteren Grenzwerts an einen Regler. Der Regler sendet seinerseits Befehle zum Öffnen oder Schließen an das Ventil. Das Beispielsystem ist hybrid. Zwischen Zufluß, Pegel und Abfluß bestehen kontinuierliche Abhängigkeiten. Der Regler arbeitet hingegen diskret. Das Ventil wird bei Überschreiten des oberen Schwellwerts geschlossen, bei Unterschreiten des unteren Schwellwerts geöffnet. Die Reaktionen des Reglers und des Sensors unterliegen Realzeit-Anforderungen. Die beiden Komponenten Sensor und Regler sollen im folgenden jeweils durch eine Menge von Verhaltensconstraints dargestellt werden. Die Sensor-Constraints SL und SH sind Safety-Constraints derart, daß nur dann ein `"high"` bzw. `"low"` Signal an den Regler gegeben werden darf, wenn der aktuelle Pegel dem auch entspricht. SB ist ein Constraint, das durch Angabe einer maximalen Reaktionszeit eine fristgerechte Pegel-Weitergabe an den Regler erzwingt. Ähnlich erzwingt CB eine fristgerechte Regler- Reaktion. CA verlangt umgekehrt, daß eine minimale Wartezeit eingehalten wird, bevor das Ventil signalisiert wird. CC und CO sind Safety-Constraints des Reglers, die besagen, daß Schließ- bzw. Öffne-Befehle an das Ventil nur nach entsprechender Signalisierung des Sensors erfolgen dürfen.

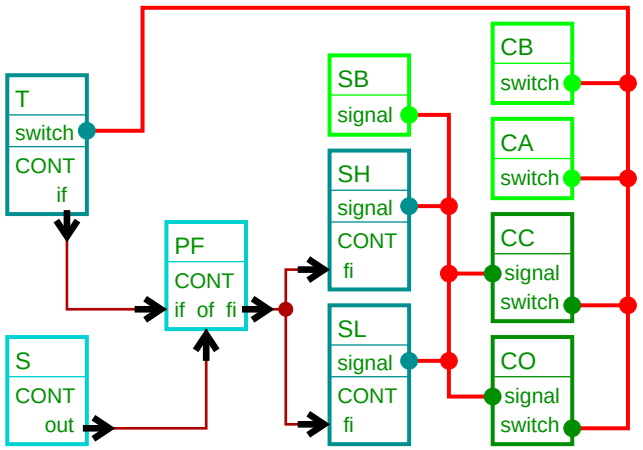

Abbildung 2: Beispielspezifikation — Systemübersicht

Abb. 2 zeigt die Struktur der Beispielspezifikation. Das System besteht aus 10 Prozessen T, S, PF, SH, SL, SB, CO, CC, CA und CB, die bereits in Abb. 1 skizziert sind. Da das Beispiel sehr einfach ist, wurde der Regler nur durch vier Constraints dargestellt und auch der Sensor wird nur durch drei Constraints modelliert. Die übrigen Komponenten sind sogar so einfach, daß jeweils nur ein Prozeß in der Spezifikation benutzt wird.

Die Prozesse CO und CC sind rein ereignisdiskrete Prozesse. Sie können jeweils mit den Aktionen `signal` und `switch` diskrete Zustandsübergänge ausführen. Mittels `signal` interagieren die Regler-Constraints mit den Sensor-Constraints. Aufgrund der Kopplung nach dem Prinzip gemeinsamer Aktionen dürfen die `signal`-Aktionen der fünf Prozesse SB, SH, SL, CO und CC nur gleichzeitig stattfinden. Dies modelliert die Signalisierung des Reglers durch den Sensor. Die `switch`-Aktionen der Regler-Constraints sind mit dem Ventil gekoppelt. Eine `switch`-Interaktion, dargestellt durch das gleichzeitige Schalten aller fünf `switch`-Aktionen, modelliert die Übergabe eines Stellsignals vom Regler an das Ventil. Die beiden Sensor-Prozesse SL und SH sind hybrid. Sie haben eine kontinuierliche Eingangsgröße `fi`, den aktuellen Pegel im Behälter. Der Behälter PF seinerseits ist ein rein kontinuierlicher Prozeß mit den Eingängen `if` und `of` — sie notieren den Zufluß und den Abfluß — sowie dem Ausgang `fi`, dem Pegel. Der Senkenprozeß S ist ein Constraint zu den möglichen Abfluß-Werten, der

sich als kontinuierlicher Prozeß mit einem Ausgang darstellt. Der Ventilprozeß T wiederum ist hybrid. Er besitzt eine kontinuierliche Ausgangsgröße und die diskrete Aktion `switch` koppelt ihn, wie schon erwähnt, mit dem Regler.

3 cTLA$^+$

Module der Spezifikationssprache cTLA+ definieren Prozeßtypen, aus denen Instanzen erzeugt werden können. Es gibt einfache Prozeßtypen und solche, die der Bildung von Systemen oder Subsystemen dienen. Die Syntax lehnt sich an Modula 2 und TLA+ [L94b] an.

```
PROCESS  Controller ( pc : {"open", "close"} )
  VARIABLES  react : ( "yes", "no" ) ;    ! Controller reaction necessary?
  INIT ≙     react = "no" ;               ! Initial state condition
  ACTIONS
    signal ( pst : ("high","low") ) ≙     ! sensor signals controller
        react′ = IF (pst = "high"  ∧ pc = "close") ∨
                    (pst = "low"   ∧ pc = "open")  THEN "yes" else "no" ;

    switch ( sw : ("open","close") ) ≙    ! controller signals tap
        ( (sw = pc  ∧ react = "yes"  ∧ react′ = "no") ∨
          (sw ≠ pc  ∧ react′ = react) ) ;

    WF: switch ;
  END
```

Der oben gezeigte Prozeßtyp Controller ist ein Beispiel eines einfachen Prozeßtyps, der direkt die Vorschrift für ein einfaches Zustandstransitionssystem liefert. Unmittelbar nach dem Prozeßtyp-Namen Controller wird in diesem Beispiel ein generischer Prozeßtyp-Parameter, der Parameter pc, vereinbart. pc ist vom Datentyp `{"open", "close"}`, eine aktuelle Parametrisierung kann also die Werte `"open"` oder `"close"` annehmen.

Unter VARIABLES werden die Zustandsgrößen des Systems vereinbart. INIT definiert die Initialzustandsbedingung, d.h. ein Prozeß startet in einem Zustand, der INIT erfüllt. Unter ACTIONS werden die Aktionen notiert. Eine Aktion (z.B. `signal`) ist eine Bedingung über Aktionsparametern (z.B. `pst`), Zustandsvariablen (z.B. `react`) und sogenannten gestrichenen Zustandsvariablen (z.B. `react'`), die den Folgezustand referenzieren. Alle Zustandsübergänge, die die Bedingung einer Aktion `a` unter einer Parametrisierung `p` erfüllen, entsprechen einem `a(p)`-Schritt. Die Menge der Aktionen definiert die Zustandsübergangsrelation des Transitionssystems: Es sind nur Übergänge erlaubt, die einer Aktion entsprechen.

Mit diesen bisher besprochenen Elementen werden Safety-Eigenschaften beschrieben. Liveness-Eigenschaften werden durch zusätzliche Fairness-Anforderungen an Aktionen notiert. Im Beispiel soll die Aktion `switch` mindestens weak fair gefeuert werden. Zur Gewährleistung der Kompositionalität sind Fairnessanforderungen in cTLA+ (im Gegensatz zu TLA/TLA+) bedingt, `WF: switch` verlangt deshalb nur, daß die bedingte Aktion `"switch` ∧ die Umgebung des Prozesses erlaubt momentan ein Schalten von `switch"` weak fair behandelt wird. Fairness bezieht sich dabei darauf, daß sich das System nicht unendlich lange in Zuständen befinden darf, in denen ein Schalten der Aktion möglich wäre. Weak Fairness erzwingt hierbei das Schalten nur, wenn ansonsten unendlich lange schaltbare Zustände in ununterbrochener Folge vorliegen würden. Strong Fairness erzwingt das Schalten auch, wenn die Folge schaltbarer Zustände ab und zu unterbrochen ist.

Mittels eines Prozeßinstanz-Aufrufs (z.B. `CC : Controller("close")`) kann eine Instanz eines Prozeßtyps erzeugt werden. Solche Prozeßinstanzen treten in Prozeßtypen auf, die der Bildung echter Systeme dienen. So kann die in Abb. 2 gezeigte Konfiguration (bezogen auf den ereignisdiskreten Teil) durch den folgenden Prozeßtyp ExampleSystem beschrieben werden.

```
PROCESS  ExampleSystem
  PROCESSES  T : Tap ;  S : Sink ; PF : Pool ;
             SH : Sensor("high") ; SL : Sensor("low") ; SB : SensorBefore;
             CC : Controller("close") ; CO : Controller("open") ;
             CB : ControllerBefore ; CA : ControllerAfter ;

  ACTIONS
    sensorSignals ( pst : {"high","low"} ) ≙
             T.stutter ∧ S.stutter ∧ PF.stutter ∧
             SH.signal(pst) ∧ SL.signal(pst) ∧ SB.signal ∧
             CC.signal(pst) ∧ CO.signal(pst) ∧
             CB.stutter ∧ CA.stutter ;

    controllerSwitchs ( sw : {"open","close"} ) ≙
             T.switch(sw) ∧ S.stutter ∧ PF.stutter ∧
             SH.stutter ∧ SL.stutter ∧ SB. stutter ∧
             CC.switch(sw) ∧ CO.switch(sw) ∧
             CB.switch ∧ CA.switch ;
    ! add the description of the continuous coupling here !
  END
```

Das System besteht aus den Prozessen T, S, PF, SH, SL, SB, CC, CO, CB und CA. Sie werden nach dem Prinzip gemeinsamer Aktionen so gekoppelt, daß im System nur die beiden Aktionen `sensorSignals` und `controllerSwitchs` möglich sind. `sensorSignals` besteht aus der UND-Verknüpfung der `signal`-Prozeßaktionen der Prozesse SH, SL, SB, CC und CO. Die übrigen Prozesse T, S und PF müssen dabei einen in TLA sogenannten Stotterschritt ausführen, d.h. sie verändern ihren Zustand nicht. Dies wird durch die Pseudoaktion `stutter` zum Ausdruck gebracht. Ähnlich definiert die Systemaktion `controllerSwitchs` eine gemeinsame Aktion der Prozesse T, CC, CO, CB und CA.

4 Realzeit-Constraints

Die Realzeit wird entsprechend der in [AL91b] vorgeschlagenen Lösung durch eine reellwertige Zustandsgröße `now` repräsentiert. In cTLA+-Spezifikationen muß `now` nicht explizit deklariert werden und kann — anders als übrige Zustandsvariablen, die immer nur im deklarierenden Prozeß sichtbar sind — in beliebigen Prozessen — allerdings nur lesend — angesprochen werden. Es wird angenommen, daß `now` durch eine Aktion `tick` in lebendiger Weise erhöht wird. Es darf dabei eine beliebig kleine aber echt positive Schranke für den Maximalwert der Inkremente angenommen werden.

Realzeit-Anforderungen werden (ebenfalls auf [AL91b] aufbauend) — wie Liveness-Anforderungen — in Begriffen des Schaltens von Aktionen ausgedrückt:

- `V MIN TIME a : t;` verletzliche minimale Wartezeit
- `P MIN TIME a : t;` persistente minimale Wartezeit
- `V MAX TIME a : t;` verletzliche maximale Reaktionszeit
- `P MAX TIME a : t;` persistente maximale Reaktionszeit

Alle vier Anforderungstypen beziehen sich auf eine Aktion `a` und eine Zeitspanne `t`. Dabei bezieht sich `t` auf Zeitspannen, in denen die Aktion a schaltbar ist, aber nicht schaltet. Die Angabe einer minimalen Wartezeit verlangt, daß jedem Schalten der Aktion `a` eine solche Zeitspanne mit der Mindestlänge `t` vorausgehen muß. Die Angabe einer maximalen Reaktionszeit verlangt, daß die

Aktion `a` schalten muß, spätestens nachdem sie die angegebene Zeitspanne `t` lang schaltbar war. Verletzliche Warte- oder Reaktionszeiten beziehen sich auf Zeitspannen, in denen die Aktion ununterbrochen schaltbar ist. Werden persistente Warte- oder Reaktionszeiten notiert, kann die Zeitspanne `t` von Perioden, in denen die Aktion `a` nicht schaltbar ist, unterbrochen werden; `t` bezieht sich hier auf die Summe der Intervall-Längen, in denen `a` seit dem letzten Schalten schaltbar war. Man beachte die Korrespondenz wischen Weak Fairness und verletzlichen Zeitspannen sowie zwischen Strong Fairness und persistenten Zeitspannen.

Zur Gewährleistung der Teilsystem-Kompositionalität weicht die formale Semantik der mit cTLA+ verfolgten Lösung von der in [AL91b] vorgestellten ab. cTLA+-Realzeit-Anforderungen zu maximalen Reaktionszeiten sind schwächer. Sie beziehen sich nur auf solche Zeitspannen, in denen die Aktion `a` schaltbar ist und gleichzeitig auch alle diejenigen Aktionen der Umgebung schaltbar sind, mit denen `a` nach dem Prinzip gemeinsamer Aktionen gekoppelt ist.

```
PROCESS  ControllerAfter
  ACTIONS
    switch ; ! signal to tap

    V MIN TIME: switch 2.4 ;
  END
```

```
PROCESS  ControllerBefore
 ACTIONS
   switch ; ! signal to tap

   V MAX TIME: switch 2.5 ;
 END
```

```
PROCESS  SensorBefore
  ACTIONS
    signal ; ! signal to controller

    V MAX TIME: signal 0.5 ;
  END ;
```

Als Beispiele zu Prozeßtypen, die mit Realzeit-Anforderungen behaftet sind, finden sich oben die drei in unserem Systembeispiel benutzten Prozeßtypen ControllerAfter, ControllerBefore und SensorBefore. Alle drei Prozeßtypen sind sehr einfach. Es wird lediglich der Name der betreffenden Aktion eingeführt und die Realzeitanforderung der Aktion genannt.

Die modulare Verifikation eines Systems kann dadurch unterstützt werden, daß aktivitätsforcierende Constraints mit einem Safety-Modell ihrer Umgebung kombiniert werden (vgl. Liveness-Constraints gemäß [HK95]). Dies soll hier nicht weiter ausgeführt werden.

5 Hybride Constraints

Zur Beschreibung hybrider Constraints wird in einen Prozeßtyp eine Aktion namens CONT (für continuous) eingeführt, die das zeitkontinuierliche Verhalten des Prozesses modelliert. Hierzu wird angenommen, daß die CONT-Aktionen aller Prozesse eines Systems miteinander und mit der tick-Aktion der Realzeituhr gekoppelt sind, so daß kontinuierliche Beziehungen in Form von Differenzen-Gleichungen formuliert werden können, die sich auf beliebig kleine, aber echt positive und bei allen Prozessen eines Systems gleichartig auftretende Zeitschritte beziehen können.

```
PROCESS  ExampleSystem  ! cf. PROCESS ExampleSystem in Sect. 3.
  PROCESSES  ...
  ACTIONS
    sensorSignals ...
    controllerSwitchs ...
    CONT ( OUTPUT if, of, fi : Real ) ≙  ! the continous coupling
            T.CONT(if) ∧ S.CONT(of) ∧ PF.CONT(if,of,fi) ∧
            SH.CONT(fi) ∧ SL.CONT(fi) ∧ SB. stutter ∧
            CC.stutter ∧ CO.stutter ∧ CB.stutter ∧ CA.stutter ;
  END
```

Alle kontinuierlichen Ein- und Ausgangsgrößen eines Prozesses werden als Parameter der CONT-Aktion aufgeführt. So kann die Kopplung der kontinuierlichen Größen eines Systems in der CONT-Aktion einer Systembeschreibung durch die aktuelle Parametrisierung der CONT-Aktionen der Prozesse definiert werden. Dies wird im vorstehenden Beispiel des Prozeßtyps ExampleSystem gezeigt. Es bezieht sich auf das in Abb. 1 beschriebene System.

```
PROCESS Sensor ( ps : {"high", "low"} )
  VARIABLES  st : {"ready", "alarm", "wait"} ; ! the sensor state
  INIT ≙ st = "ready" ;
  ACTIONS
    CONT ( INPUT  fi : Real ) ≙  ! continuos detection of level
      st′ = CASE  ps = "low"   ∧ fi<40 ∧ st ≠ "wait" → "alarm";
                  ps = "high"  ∧ fi>60 ∧ st ≠ "wait" → "alarm";
                  ps = "low"   ∧ fi≥40               → "ready";
                  ps = "high"  ∧ fi≤60               → "ready";
                  OTHERWISE                           → "wait" ;
              ENDCASE ;
    signal ( pst : {"high", "low" } ) ≙  ! signal to controller
      ( (pst=ps ∧ st = "alarm" ∧ st′ = "wait") ∨
```



```
        (pst≠ps ∧ st′ = st) ) ;
  END
```

Im Beispiel nach Abb. 1 beobachten Prozesse des vorstehenden Typs Sensor die kontinuierliche Eingangsgröße `fi` (INPUT-Parameter der Aktion CONT) und belegen in Abhängigkeit dazu die diskrete Zustandsgröße `st`. Je nach Prozeßparameter bringt der Prozeß ein Constraint zum Ausdruck, das Signalisierungen des Reglers mit einer oberen bzw. einer unteren Grenzwertüberschreitung einschränkt. So wird die obere Überschreitung (`signal("high")`) nur unter der Parametrisierung `ps="high"` eingeschränkt (dann gilt `pst=ps` in der Aktion `signal`). Es darf unter diesen Umständen nur signalisiert werden, wenn der Prozeß vorher ein Überschreiten des Grenzwerts beobachtete (`st="alarm"`). Nach einer Signalisierung gilt `st="wait"` und der Pegel muß vor der nächsten Signalisierung den Grenzwert wieder unterschritten haben (`st="normal"`).

```
PROCESS Sink
  ACTIONS
    CONT ( OUTPUT out : Real ) ≙ 4 ≤ out ∧ out ≤ 6 ;  ! continuous flow out
  END
```

Ein Prozeß des vorstehenden Typs Sink beschreibt den Abfluß des Behälters. Der Prozeß besitzt eine kontinuierliche Ausgangsgröße `out`. Sie modelliert den Wert des Abflusses und wird in ihrem exakten zeitlichen Verlauf nicht weiter spezifiziert. Der Prozeß bringt lediglich zum Ausdruck, daß sich die Abflußwerte immer innerhalb des Intervalls $[4, 6]$ bewegen.

```
PROCESS Pool
  VARIABLES
    f : Real ; ! level
  INIT ≙ f = 50 ; ! initial level
  ACTIONS
    CONT ( INPUT if, of : Real ;    ! flow in, flow out
           OUTPUT fi : Real ) ≙     ! level
      f′ = f + (if-of) * (now′-now) ! state change
      ∧  fi = f ;                   ! output constraint
  END
```

Ein Prozeß des vorstehenden Typs Pool beschreibt den Flüssigkeitsbehälter des Beispiels. Es ist ein rein kontinuierlicher Prozeß. Die reellwertige Variable f soll den aktuellen Pegel modellieren. Es wird nur die kontinuierliche Aktion CONT vereinbart. CONT enthält zwei Bedingungen. Die erste beschreibt das Zustandsübergangsverhalten in Form einer Differenzengleichung (f' steht für

den Wert von f zum Zeitpunkt now', now notiert den Wert der aktuellen Zeit entsprechend Abschnitt 4). Die zweite Bedingung legt den Wert der Ausgangsgröße fest.

```
PROCESS Tap
  VARIABLES
    fl : Real ;                          ! the flow of the tap
    tap : {"open", "closed" }            ! the state of the tap
  INIT  ≙  fl = 0  ∧ tap = "closed"      ! initially no flow, tap closed
  ACTIONS
    CONT ( OUTPUT if : Real ) ≙          ! if: flow to pool
      fl′ = IF tap = "open"
              THEN  Min (10, fl + 10 * (now′-now))
              ELSE  Max ( 0, fl - 10 * (now′-now))
            ∧ if = fl ;
    switch ( sw : {"open", close"} ) ≙ ! signal from controller
      tap′ = sw  ∧ fl′ = fl ;            ! switch tap
  END
```

Der vorstehende Prozeßtyp Tap ist der letzte noch nicht gezeigte Typ unseres Beispielsystems. Der entsprechende Prozeß modelliert das Ventil des Beispiels, das von der diskreten Aktion `switch` geöffnet oder geschlossen werden kann, in seinen Auswirkungen auf den kontinuierlichen Zufluß des Behälters. Bei geöffnetem Ventil steigt der Fluß mit konstanter Steigung bis zum Maximalwert von 10. Bei geschlossenem Ventil verringert sich der Fluß mit konstantem Gefälle bis zum Versiegen.

6 Strukturierte Verifikation

Die Semantikfestlegungen von cTLA+ ordnen cTLA+-Prozessen und Prozeßsystemen äquivalente Formeln der Temporallogik TLA zu. So können formale Verifikationsmaßnahmen mit den Mitteln von TLA durchgeführt werden. Der Beweis, daß ein System I eine Spezifikation S implementiert, d.h., daß I alle in S beschriebenen Eigenschaften besitzt, ist äquivalent zum Beweis der Gültigkeit der TLA-Implikation $I \Rightarrow S$. Der Beweis kann durch syntaktische Ableitung im TLA-Kalkül erbracht werden. Er kann auch im Beweis der Existenz einer Refinement Mapping genannten Zustandsabbildung rm aufgehen [AL91a]. Ein Refinement Mapping rm bildet dabei Zustände von I auf Zustände von S ab. Es besitzt strukturerhaltende Eigenschaften (Initialzustandstreue und Transitionentreue). Die in S geforderten Liveness-Eigenschaften müssen von den rm-

Bildern der in I möglichen Zustandsfolgen erfüllt werden. Und die in I und S zueinander korrespondierenden Zustandsvariablen müssen von rm per Identität abgebildet werden. Sowohl im TLA-Kalkül als auch beim Nachweis der Existenz eines Refinement Mappings ist es zentral, daß Systemeigenschaften zustandsbasiert ausgedrückt werden. Der Beweis von Safety-Eigenschaften kann durch Invariantenbeweise erbracht werden. Wichtig ist dafür der Entwurf einer geeigneten Invariante. Der Beweis von Liveness-Eigenschaften wird im groben derart geführt, daß zunächst mithilfe von Safety-Beweisen die Menge zu berücksichtigender Zustandsfolgen eingeschränkt wird, um dann für die verbliebenen Folgen aus den Fairness-Annahmen auf die gewünschten Liveness-Eigenschaften zu schließen (vgl. [L94a]). Obwohl TLA mächtige und geeignete Mittel anbietet, sind Beweise, daß komplexere Systeme (Systeme mit großem Zustandsraum, aufgespannt durch zahlreiche Zustandsvariablen, und mit großer Anzahl von Transitionstypen) gewünschte abstrakte Eigenschaften besitzen, aber in der Regel kaum noch mit vertretbarem Aufwand zu führen. Benötigte Invarianten müssen z.B. häufig in vielen Iterationsschritten verschärft werden. Es fällt sehr schwer, feinschrittige Argumentationsketten als Beweisideen zu entwerfen und die Übersicht darüber zu bewahren.

Mit unserem Ansatz der strukturierten Verifikation kann nun die Beweisführung wesentlich erleichtert werden. Für den Beweis einer abstrakten Eigenschaft muß nicht mehr das ganze komplexe System herangezogen werden, es genügt die Betrachtung eines Teilsystems [HK94]. In Kombination mit der Gliederung von Beweisen anhand geeigneter Zwischenziele (dargestellt durch Hilfssätze) lassen sich durch die strukturierte Verifikation leicht-verständliche Argumentationsketten in übersichtliche und mit vertretbarem Aufwand zu entwickelnde formale Beweise umsetzen. Die soll im folgenden am Beispiel des in Abschnitt 3 vorgestellten Systems ExampleSystem verdeutlicht werden.

Es soll die abstrakte Eigenschaft PUB des Systems bewiesen werden:

- PUB: Der Pegel im Behälter wird nie die Schranke 80 überschreiten.

PUB läßt sich in cTLA+ als Instanz des folgenden Prozeßtyps PoolUpperBound darstellen, wenn die Variable f des Prozesses PUB mit der Variablen f des Prozesses PF aus ExampleSystem korrespondiert.

```
PROCESS  PoolUpperBound
  VARIABLES  f : Real ;   ! the level
  INIT ≙ f = 50 ;         ! initial value of f
```

```
  ACTIONS
    CONT  ≙  f′≤80 ;     ! f may arbitrarily change but never exceed 80
  END
```

Wir führen vier Hilfssätze ein. Sie drücken Eigenschaften aus, die im Abstraktionsgrad zwischen dem detaillierten System und der abstrakten Eigenschaft PUB liegen.

- CUB: Ein Ventilschließbefehl erfolgt nur, wenn der Füllstand unter 78,1 liegt.
- ROA: Auf ein Sensorsignal für hohen Pegel (60) folgt ein Ventilschließbefehl, bevor der Füllstand 78,1 erreicht hat.
- ATC: Nach einem Ventilschließbefehl steigt der Pegel bis zum nächsten Öffne-Befehl um höchstens 1,9 Einheiten an.
- TOU: Ein Ventil-Öffne-Befehl ist nur möglich, wenn der Füllstand unter 60 liegt.

Es gilt, daß die Komposition aus CUB, ROA, ATC und TOU unser Beispiel PUB impliziert. Anstelle des einfachen formalen Beweises skizzieren wir nur die Beweisidee. Wir unterscheiden dabei abhängig vom Ventilzustand zwei Fälle: Wenn das Ventil geschlossen ist, liegt der Füllstand immer unter 80. Zum Zeitpunkt der Ventilschließung liegt er nämlich unter der Marke 78,1 (CUB). Danach kann er nur noch um 1,9 steigen (ATC). Bei offenem Ventil unterschreitet der Füllstand stets die Marke 78,1. Zum Zeitpunkt der Öffnung liegt der Füllstand unter 60 (TOU). Wenn er anschließend diesen Wert überschreitet, löst er ein Sensorsignal aus, so daß das Ventil vor Erreichen von 78,1 wieder geschlossen wird (ROA).

Anschließend muß gezeigt werden, daß die vier Hilfssätze durch Teilsysteme von ExampleSystem impliziert werden. Als Beispiel skizzieren wird den Beweis von ROA. ROA wird in cTLA+ durch eine Prozeßinstanz des Typs ReactOnAlarm dargestellt:

```
PROCESS ReactOnAlarm
  VARIABLES
    st : { "limit", "ready", "wait" }; ! lemma states
    f : Real;                          ! level
  INIT = st = "ready"  ∧  f = 50;      ! initial state
  ACTIONS
    CONT ≙
```



```
      f′ ∈ IF (st = "limit")
           THEN { k | k ∈ Real : k < 78.1 }
           ELSE Real ∧
      ! If increase is limited, level must not exceed 78.1
      st′ = CASE f ≤ 60                → "ready";
                 f > 60  ∧  st ≠ "wait" → "limit";
                 OTHERWISE              →  st;
      ! Level increase is limited if mark 60 is passed
    switch (sw : {"open","close"}) ≙
      st′ = IF (sw = "close"  ∧  st = "alarm")
            THEN "wait" ELSE st ∧
      ! Release of level increase at tap closing switch
      f′ = f;
END
```

Modelliert wird, daß ein Ventilschließbefehl (diskrete Aktion `switch`) erfolgen muß, bevor der Füllstand des Tanks den Wert 78,1 erreicht, wenn zuvor ein Sensorsignal für hohen Pegel ausgelöst wurde. Die verschiedenen Zustände des Systems werden durch die Variable `st` und der Füllstand des Tanks durch `f` beschrieben. Wenn der Füllstand unter dem Alarmpegel 60 liegt, gilt `st="ready"` und an den Flüssigkeitsanstieg werden keine Anforderungen gestellt. Nach dem Überschreiten des Alarmpegels gilt `st="limit"`. Der Füllstand `f` darf zunächst nur Werte annehmen, die unter dem kritischen Pegel 78,1 liegen. Erst nach einem Ventilschließbefehl (Aktion `switch("close")`) wird diese Beschränkung aufgehoben und der Füllstand darf weiter ansteigen (`st="wait"`).

Zum Beweis des Hilfssatzes ROA wird ein Subsystem TS von ExampleSystem verwendet. Es ist komponiert aus den Constraints SH und CC, die das funktionale Verhalten des Sensors und des Reglers bei hohem Pegel modellieren. Außerdem wird zum Beweis die Reaktionszeit des Sensors und des Reglers verwendet, so daß auch die Realzeitconstraints SB und CB in TS komponiert sind. Ferner ist zur Beweisführung ein weiterer Hilfssatz notwendig, der die Steigung des Pegels angibt:

- PFM: Die maximale Steigung des Pegels ist 6.

Dieser Hilfssatz wird durch ein Teilsystem von ExampleSystem impliziert, das sich aus den Constraints PF, T und S zusammensetzt. Der Beweis wird wiederum nur anhand der Beweisidee veranschaulicht. Die Füllstandsänderung des Tanks ergibt sich aus der Differenz von Flüssigkeitszufluß und -abfluß (PF). Da der maximale Zufluß 10 (T) und der minimale Abfluß 4 (S) beträgt, kann der Füllstand um maximal 6 wachsen.

Im folgenden skizzieren wir den Beweis TS $\Rightarrow$ ROA, wobei TS für das aus SH, CC, SB, CB und PFM zusammengesetzte System steht. Zunächst wird die Zustandsabbildung zwischen TS und ROA durch die Formel

```
ROA.st = IF ((CC.react = "yes" ∧ SH.st = "wait") ∨ SH.st = "alarm")
         THEN "limit" ELSE SH.st ∧
ROA.f = PFM.f
```

festgelegt. Die Variable `ROA.st` wird also im Teilsystem TS durch die Variablen `SH.st` von Constraint SH und `CC.react` von Constraint CC implementiert. `ROA.f` entspricht der Variable `PFM.f` im Hilfssatz PFM.

Der Beweis wird als TLA-Safetybeweis geführt (vgl. [L94a]). Dazu wird zunächst bewiesen, daß die Initialbedingung von TS die von ROA impliziert. Anschließend wird nachgewiesen, daß jede Aktion von TS mit jeder erlaubten Parameterbelegung entweder eine Aktion in ROA mit einer erlaubten Parameterbelegung oder einen Stotterschritt in ROA impliziert. Unter einem Stotterschritt versteht man einen Zustandsübergang in denselben Zustand. Zum Beweis eines Teils dieser Implikationen ist die Verwendung einer Invariante des Systems TS notwendig. Eine Invariante wird durch den Nachweis verifiziert, daß sie zum einen im Initialzustand von TS gilt und zum anderen bei der Schaltung aller Aktionen von TS erhalten bleibt.

Der Beweis, daß die Initialbedingung von ROA aus der von TS folgt, ist trivial. Da die Variable `st` von SH initial den Wert `"ready"` annimmt, folgt nach der Zustandsabbildung sofort die Initialbedingung `st = "ready"` von ROA. Außerdem wird `f = 50` durch die Bedingung `PFM.f = 50` impliziert.

Der Beweis, daß jede Aktion des Teilsystems TS entweder eine Aktion oder einen Stotterschritt in ROA impliziert, kann ohne Verwendung einer Invariante nur für die Aktionen `signal` und `switch` in TS geführt werden. Als Beispiel zeigen wir den Beweis der Aktion `signal("high")`, die in ROA einem Stotterschritt entspricht. Die Variable `PFM.f` verändert ihren Wert beim Schalten von `signal("high")` nicht. Die Variable `st` in ROA hat immer den Wert `"limit"`, wenn `signal("high")` ausgeführt wird (`SH.st = "alarm"`). Nach der Ausführung gilt `SH.st="wait"` und `CC.react="yes"`, so daß `ROA.st` nach der Zustandsabbildung weiterhin den Wert `"limit"` besitzt.

Für den Beweis, daß die Aktion `CONT` des Teilsystems TS die Aktion `CONT` von ROA impliziert, wird die folgende Invariante von TS aufgestellt und verifiziert:


```
( ( CC.react = "no" ∧ SH.st = "wait" ) ∨ PFM.f < 78,1 ) ∧
( SH.st = "wait" ∨ PFM.f < 63,1 ) ∧
( SH.st = "ready" ∨ PFM.f ≥ 59,9 ) ∧
( SL.st = "ready" ∨ PFM.f ≤ 40,1 )
```

Mit Hilfe des ersten Konjunkts der Invariante kann leicht bewiesen werden, daß die Aktion `CONT` in TS die Aktion `CONT` in ROA impliziert. Die anderen Konjunkte dienen zum Beweis der Invariante.

Außerdem muß noch gezeigt werden, daß die Reaktionszeitanforderungen in den Realzeitconstraints SB und CB nicht durch Constraints von ExampleSystem verletzt werden, die nicht in TS komponiert sind. Da beim Beweis von ROA nur die Reaktionszeiten für die Aktion `signal("high")` und `switch("low")` verwendet werden, deren Schaltbedingungen nicht durch Constraints außerhalb von TS beschränkt sind, ist dieser Nachweis jedoch trivial.

7 Schlußbemerkungen

Der Beitrag stellte den Ansatz vor, hybride Systeme in constraintorientierter Struktur zu beschreiben, um eine erleichterte formale Verifikation, nämlich die hier sogenannte strukturierte Verifikation zu ermöglichen. Die strukturierte Verifikation kann bei geeigneter Gliederung eines Systems in Constraints zu sehr deutlich reduziertem Verifikationsaufwand und zu leicht nachvollziehbaren Systembeweisen führen, da sich die Betrachtung auf Teilsysteme beschränken kann. Der Ansatz wurde in Verbindung mit der Spezifikationssprache cTLA+ vorgestellt, die die hierzu notwendige Teilsystemkompositionalität besitzt. Obwohl hier nicht deutlich herausgestellt, ist cTLA+ eine formale Spezifikationssprache mit exakt definierter formaler Semantik. Auch existieren bereits rechnergestützte Werkzeuge (z.B. Transformation von cTLA+-Spezifikationen in TLA+-Spezifikationen, TLA+-Frontend für den Zugang zu einem Prädikatenlogik-Beweiser, TLA+-Interpreter für die Ablaufsimulation von Spezifikationen, TLA+-Model Checker für die Erreichbarkeitsanalysebasierte automatische Verifikation endlicher Systeme), die teilweise zur Zeit an die besprochenen Erweiterungen von cTLA+ für hybride Systeme angepaßt werden. Ausgehend von den Erfahrungen mit der formalen korrektheitssichernden Behandlung verteilter ereignisdiskreter Systeme (nämlich von Kommunikationsprotokollen) gehen wir aber davon aus, daß das Arbeiten mit einer formalen Spezifikations- und Verifikationstechnik durch rechnergestützte Werkzeuge

zwar stark erleichtert werden kann, daß sich als Hauptproblem der Anwendung aber die vom Menschen zu leistende adäquate Modellierung herausstellt. Dies kann durch Bereitstellung wiederverwendbarer Spezifikationsbausteine und daran angepaßter Theoreme zur Verifikationsunterstützung wesentlich erleichtert werden. Die Modularität und Kompositionalität von cTLA+ bietet dafür sehr gute Voraussetzungen und weitergehende Arbeiten werden diese Richtung vertiefen.

Literatur

[ACHH93] Alur, R., Courcoubetis, C., Henzinger, Th.A., Ho, P.-H.: *Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems.* Hybrid Systems, 209–229, Lecture Notes in Computer Science 736, Springer Verlag, 1993.

[AHH93] Alur, R., Henzinger, Th.A., Ho. P.-H.: *Automatic Symbolic Verification of Embedded Systems.* Proc. 14th Annual IEEE Real-time Systems Symposium, 2–11, 1993.

[AL91a] Abadi, M., Lamport, L.: *The Existence of Refinement Mappings.* Theoretical Computer Science **82** (2), 253–284, 1991.

[AL91b] Abadi, M., Lamport, L.: *An Old-Fashioned Recipe for Real Time.* Aus J. W. de Bakker, C. Huizing, W. P. de Roever, und G. Rozenberg, Real-Time: Theory in Practice, Lecture Notes in Computer Science 600, Springer-Verlag, 1991.

[AL93] Abadi, M., Lamport, L.: *Composing Specifications.* ACM Transactions on Programming Languages and Systems **15** (1), 73–132, 1993.

[B96] Benveniste, A.: *Compositional and Uniform Modelling of Hybrid Systems.* Hybrid Systems III, Lecture Notes in Computer Science, 41–51, Springer-Verlag, 1996.

[H93] Hooman, J.: *A Compositional Approach to the Design of Hybrid Systems.* Hybrid Systems, Lecture Notes in Computer Science 736, 121–148, Springer Verlag, 1993.

[HK94] Herrmann, P., Krumm, H.: *Compositional Specification and Verification of High-Speed Transfer Protocols.* Proc. Protocol Specification, Testing, and Verification XIV, 339–346, IFIP, Chapmann & Hall, Vancouver, 1994.

[HK95] Herrmann P., Krumm H.: *Re-Usable Verification Elements for High-Speed Transfer Protocol Configurations.* Proc. Protocol Specification, Testing, and Verification XV, 171–186, IFIP, Chapmann & Hall, Warschau, 1995.

[HMK96] Heyl, C., Mester, A., Krumm, H.: *cTc — A Tool Supporting the Construction of cTLA-Specifications.* Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 1055, 407–411, Springer-Verlag, 1996.

[L93] Lamport, L.: *Hybrid Systems in TLA$^+$.* Hybrid Systems, Lecture Notes in Computer Science 736, 77–102, Springer-Verlag, 1993. Springer-Verlag.

[L94a] Lamport, L.: *The Temporal Logic of Actions.* ACM Transactions on Programming Languages and Systems **16** (3), 872–923, 1994.

[L94b] Lamport, L.: *TLA+.* DEC Digital Systems Research Center, 1994.

[QW96] Qiwen, X., Weidong, H.: *Hierarchical Design of a Chemical Concentration Control System.* Hybrid Systems III, Lecture Notes in Computer Science, 270–281, Springer-Verlag, 1996.

[S96] Sintzoff, M.: *Abstract Verification of Structured Dynamical System.* Hybrid Systems III, Lecture Notes in Computer Science, 126–137, Springer-Verlag, 1996.

[VSvS88] Vissers, C.A., Scollo, G., van Sinderen, M.: *Architecture and specification style in formal descriptions of distributed systems.* Proc. Protocol Specification, Testing and Verification VIII, 189–204, IFIP, Elsevier, 1988.

[ZHK96] Zhou, P., Hooman, J., Kuiper, R.: *Compositional Verification of Real-Time Systems with Explicit Clock Temporal Logic.* Formal Aspects of Computing, 294–323, 1996.