
BERICHTE

AUS DEM DEPARTMENT FÜR INFORMATIK
der Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften

Herausgeber: Die Professorinnen und Professoren
des Departments für Informatik

Combination of Processes, Data, and Time

Jochen Hoenicke

Dissertation

9/06 - Juli 2006

ISSN 0946-2910

Gutachter:

Prof. Dr. E.-R. Olderog (Univ. Oldenburg)

Prof. Dr. A. P. Ravn (Univ. Aalborg)

eingereicht: 5. Mai 2006

Tag der Disputation: 12. Juli 2006

© 2006 by the author

Author's address:
Jochen Hoenicke
Fakultät II, Department für Informatik
Abteilung „Entwicklung korrekter Systeme“
26111 Oldenburg
Germany

E-mail: hoenicke@gmail.com



Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Combination of Processes, Data, and Time

Dissertation zur Erlangung des Grades eines
Doktors der Naturwissenschaften

vorgelegt von

Dipl.-Inform. Jochen Hoenicke

Oldenburg, July 12, 2006

Abstract

Nowadays, complex computing systems control safety critical systems like nuclear power plants, aeroplanes, and modern cars. Errors in safety critical systems can have catastrophic consequences. The best way to ensure that a system is error free is by using formal methods. However, no existing formal method covers all aspects of these systems.

In this work, we present a combination of the formal methods CSP, Object-Z and Duration Calculus. Each method can describe certain aspects of a system: CSP can describe behavioural aspects, such as sequential and concurrent behaviour and synchronous communication, Object-Z complex data operations, and Duration Calculus real-time requirements. It is challenging to combine these to a unified language, *CSP-OZ-DC*, that takes advantage of the individual strengths of the underlying formalisms.

The semantics of CSP-OZ-DC needs to cover the basic entities of the constituent languages: events for CSP, operations and data spaces for Object-Z, and time dependent observables for Duration Calculus. We describe a particular behaviour of the system as a trajectory mapping each point in time to a valuation that provides values for all state variables. Instantaneous events are represented by rising and falling edges of Boolean state variables. The *trace semantics* of the full system is given as the set of all behaviours that are allowed by the CSP, the Object-Z, and the Duration Calculus part.

To facilitate the use of model checkers, we also provide an *operational semantics*. To this end, we introduce *phase event automata*, a new class of timed automata. The components of a combined specification can be translated individually into phase event automata that run in parallel to build the full system. The full system permits those behaviours that are permitted by each component automaton. We prove the soundness of the translation by showing that a behaviour is in the trace semantics of the CSP, the Object-Z, or the DC part if and only if it is accepted by the corresponding automaton.

We present two different model checking approaches for phase event automata. For systems having a finite data space, an abstraction from phase event automata into timed automata can be used. Timed automata can

be checked with tools like Uppaal [BDL04], which can prove properties like reachability. For systems with an infinite data space, we present a different approach. The phase event automata are translated into transition constraint systems that represent the transition relation by a first-order formula. A bounded model checker can be used to find counterexamples that violate the formula. The absence of counterexamples in a system can be proven by the abstraction refinement model checker ARMC [Ryb02]. There is no guarantee that this model checker terminates, since the problem it strives to solve is not decidable. However, we demonstrate the effectiveness by a case study of a real-time system with unbounded integer variables in its data space.

Zusammenfassung

Heutzutage steuern komplexe Computersysteme sicherheitskritische Anwendungen wie Kernkraftwerke, Flugzeuge und Autos. Fehler in sicherheitskritischen Systemen können katastrophale Auswirkungen haben. Die beste Art diese Fehler zu vermeiden ist der Einsatz formaler Methoden. Allerdings gibt es keine formale Methode die alle Aspekte eines komplexen Systems gut beschreiben kann.

In dieser Arbeit präsentieren wir eine Kombination der formalen Methoden CSP, Object-Z und Duration Calculus. Jede Methode kann gewisse Aspekte eines Systems beschreiben. CSP kann Verhaltensaspekte, wie sequentielle und parallele Abläufe sowie Kommunikation, Object-Z kann komplexe Operationen auf Daten und Duration Calculus kann Realzeitanforderungen beschreiben. Es ist eine Herausforderung diese Techniken in einer einheitlichen Sprache, CSP-OZ-DC, zu vereinigen.

Die Semantik von CSP-OZ-DC muss die Primitiven der zugrunde liegenden Sprachen unterstützen: zeitlose Ereignisse für CSP, Operationen und komplexe Datentypen für Object-Z und zeitabhängige Observablen für Duration Calculus. Ein konkretes Verhalten wird daher durch eine Trajektorie beschrieben, die jedem Zeitpunkt eine Belegung aller Zustandsvariablen zuordnet. Ereignisse werden durch steigende und fallende Flanken einer booleschen Variable dargestellt. Die *Tracesemantik* des Gesamtsystems ist die Menge der Verhalten, die vom CSP-Teil, vom Object-Z-Teil und vom Duration-Calculus-Teil erlaubt werden.

Um die Benutzung von Model-Checkern zu ermöglichen, wird auch eine *operationelle Semantik* angegeben. Dazu führen wir eine neue Klasse von zeitbehafteten Automaten, *Phasen-Event-Automaten*, ein. Die Komponenten einer kombinierten CSP-OZ-DC-Spezifikation werden getrennt in Automaten übersetzt. Diese Automaten laufen parallel ab, um das Verhalten des Gesamtsystems abzubilden. Die Parallelkomposition erlaubt dabei genau das Verhalten, das von allen Automaten erlaubt wird. Wir beweisen die Korrektheit der Übersetzung, indem wir zeigen, dass ein Verhalten genau dann in der Tracesemantik des CSP-, des Object-Z oder des Duration-Calculus-Teils liegt, wenn der zugehörige Automat das Verhalten akzeptiert.

Wir stellen zwei verschiedene Ansätze für Model-Checking auf Phasen-Event-Automaten vor. Für Systeme mit endlichem Datenraum geben wir eine Übersetzung von Phasen-Event-Automaten nach zeitbehafteten Automaten an, die von Ereignissen und Datenvariablen abstrahiert. Diese Automaten können mit Werkzeugen, wie zum Beispiel Uppaal [BDL04], auf bestimmte Eigenschaften, wie die Erreichbarkeit von Zuständen, überprüft werden. Für Systeme mit unendlichem Datenraum stellen wir einen weiteren Ansatz vor. Die Phasen-Event-Automaten werden in Transition-Constraint-Systeme (TCS) übersetzt, die die erlaubten Übergänge durch eine Formel in Prädikatenlogik erster Stufe angeben. Solche Systeme lassen sich dann mit einem Bounded-Model-Checker verifizieren. Wenn ein System nicht korrekt ist, kann der Bounded-Model-Checker ein Gegenbeispiel finden. Um allerdings die Abwesenheit von beliebig langen Gegenbeispielen zu beweisen, benötigt man einen anderen Ansatz, zum Beispiel den Abstraction-Refinement Model-Checker ARMC [Ryb02]. Weil das Erreichbarkeitsproblem für TCS nicht entscheidbar ist, gibt es keine Garantie, dass ARMC terminiert. Allerdings zeigen wir die Wirksamkeit dieses Ansatzes anhand einer Fallstudie eines Realzeitsystems, das ganzzahlige Variablen benutzt, deren Werte nicht beschränkt sind.

Acknowledgements

I thank Ernst-Rüdiger Olderog, who gave me a position in his working group and gave me the inspiration for this thesis. He provided me with the freedom to develop the theory behind CSP-OZ-DC and ensured that this thesis came into existence by providing the necessary resources. Furthermore, I thank Anders Ravn for working as co-referee and improving this thesis with his comments. Also his research on the Duration Calculus forms an important basis. His formula class of implementables were the starting point for the counterexample traces.

I have enjoyed working in the group “Correct System Design” (formerly “Semantics”). I would like to thank the current and former members of the group for contributing to the positive atmosphere: Andrea Göken, Clemens Fischer, Josef Tapken, Heike Wehrheim, Henning Dierks, Michael Möller, Holger Rasch, Andreas Schäfer, Ingo Brückner, Johannes Faber, André Platzer, Roland Meyer, and Margarete Muhle. Especially I thank Clemens for establishing the basis CSP-OZ and giving me useful hints, Andreas for interesting discussions, for example about axiomatisability of Duration Calculus, and Michael, who was always helpful in creating nice looking graphics and layouts. Roland has extended the model-checking technique for Phase Event Automata in his diploma thesis. Andreas, Michael, Roland and Johannes were giving feedback for earlier versions of this thesis.

The AVACS subproject “R1” gave a lot of new ideas that have partly found their way into this thesis. The members are Andreas Podelski, Ernst-Rüdiger Olderog, Patrick Maier, Andrey Rybalchenko, Bernd Finkbeiner, Heike Wehrheim, Ingo Brückner, Uwe Waldmann, and Viorica Sofronie-Stokkermans. I have to thank Patrick Maier for introducing me to abstraction-refinement model-checking and Andrey Rybalchenko for writing the tool ARMC.

A good theory is not useful without tool support. Therefore, I thank all the people who implemented or extended tools for CSP-OZ-DC and phase event automata: Christian Ohler, Johannes Faber, Roland Meyer, Casjen Schnars, and the members of the students’ project «Syspect».

Last but not least, I thank my parents for their continuous support and for weekends with full board and lodging.

Contents

1. Introduction	1
2. CSP, Object-Z and Duration Calculus	5
2.1. CSP	5
2.1.1. Syntax	6
2.1.2. Operational Semantics	6
2.2. Z	7
2.2.1. Syntax	9
2.2.2. Type Checking	11
2.2.3. Semantics	12
2.2.4. Boolean Type	13
2.2.5. Object-Z	14
2.2.6. Reference and Value Semantics	15
2.3. Duration Calculus	16
2.3.1. Syntax of Duration Calculus	17
2.3.2. Semantics of Duration Calculus	18
2.3.3. Abbreviations	19
2.3.4. Embedding Z into Duration Calculus	20
2.3.5. Embedding Events Into State Variables	20
2.3.6. Counterexample Traces	21
2.3.7. Satisfiability of DC is Semi-Decidable	24
3. CSP-OZ-DC	29
3.1. CSPz	29
3.1.1. Syntax	30
3.1.2. Semantics	37
3.2. CSP-OZ-DC Classes	48
3.2.1. Syntax	48
3.2.2. Case Study	51
3.2.3. Semantics	54
3.3. Parallel Composition of Systems	59
3.4. Discussion and Related Work	61

3.4.1.	CSPz	61
3.4.2.	Semantics of CSP-OZ-DC	64
3.4.3.	Parallel Composition	65
3.4.4.	Related Work	65
4.	Phase Event Automata	67
4.1.	Prerequisites	69
4.2.	Syntax of Phase Event Automata	72
4.3.	Operational Semantics	75
4.4.	Automata and Formulae	81
4.5.	Deterministic Automata	83
4.6.	Case Study: Audio Control Protocol	86
4.7.	Discussion and Related Work	90
4.7.1.	Discussion	90
4.7.2.	Other Timed Automata Models	92
5.	From CSP-OZ-DC to Phase-Event-Automata	95
5.1.	Translating CSP	95
5.2.	Translating Object-Z	98
5.3.	Translating DC	101
5.3.1.	Power Set Construction for Counterexamples	103
5.3.2.	Creating the Accepting Automaton	135
5.3.3.	Case Study: Elevator	138
5.4.	Discussion and Related Work	139
6.	Model Checking	143
6.1.	Implementation of Phase Event Automata	144
6.1.1.	Representation of Formulae	144
6.1.2.	Computing the Power Set Automaton	146
6.2.	Reachability and Phase Event Automata	149
6.3.	Translation to Uppaal Automata	151
6.3.1.	Case Study: Audio Protocol	154
6.4.	A Constraint-based Semantics for PEA	159
6.4.1.	Transition Constraint Systems	160
6.4.2.	Translation of PEA to TCS	161
6.4.3.	Bounded Model Checking	165
6.4.4.	ARMC	166
6.4.5.	Case Study: Elevator	167
6.5.	Related Work	171

6.5.1. Audio Protocol	171
6.5.2. Model Checking Duration Calculus	172
7. Conclusion	175
7.1. Summary	175
7.2. Future Work	176
A. Syntax of CSP-OZ-DC	179
A.1. New constructs in CSPz	179
A.2. New constructs in CSP-OZ-DC	180
A.3. DC formulae	181
Bibliography	183
Index	193
Curriculum Vitae	197
Technical Reports	199

List of Figures

3.1.	Type-checking rule for a CSPz process declaration	40
3.2.	Elevator specification	55
4.1.	Phase event automaton for a watchdog	70
4.2.	Manchester coding of ‘101100100’	87
4.3.	The sender and receiver automata	89
4.4.	Gaps between two rising edges in Manchester encoding	90
4.5.	Comparison of timed automata models	94
5.1.	Translation of CSP part of elevator	99
5.2.	Translation of Object-Z part of elevator	102
5.3.	Automaton for $\mathbf{true} \wedge [A] \wedge \ell \geq 4 \wedge [B] \wedge \ell < 6$	109
5.4.	Automaton for $[A] \wedge [B]$	111
5.5.	Automaton for $[A] \wedge \ell > 1 \wedge [B]$	112
5.6.	Automaton for $[\mathbf{true}] \wedge [B] \wedge \ell \geq 2 \wedge [\neg B]$	113
5.7.	Automaton for $[A] \wedge \ell < 2$	113
5.8.	Automaton for $[A] \wedge \ell < 1 \wedge \ell \leq 2$	114
5.9.	Definition of <i>guard</i>	116
6.1.	Decision Diagram representation	145
6.2.	Java code to manage Constraint Decision Diagrams	146
6.3.	Pseudo code to compute successors and guard	148
6.4.	Pseudo code to build power set automaton	150
6.5.	Product automaton for the Audio Protocol	154
6.6.	Product automaton without unreachable edges and locations	155
6.7.	Algorithm to semi-decide reachability for a TCS	166

1. Introduction

In recent years, computing systems have become ubiquitous. So-called embedded systems control technical equipment. They are widely used in consumer applications like digital cameras, mobile phones, and handhelds. They also control safety critical systems like nuclear power plants, aeroplanes, and modern cars. Errors in safety critical systems can have catastrophic consequences. The best way to ensure that a system is error free is by applying formal methods. These methods enable the manufacturer to actually prove that the system contains no software errors.

However, no existing formal method covers all aspects of these systems. There are techniques to describe the behavioural aspects like CSP [Hoa85] and CCS [Mil89] that provide methods to analyse the behaviour of communicating concurrent processes. With state-based techniques like Z [ISO02] and B [Abr96], it is possible to model large state spaces and complex operations. For real-time systems that have to react within certain time bounds, timed automata [AD94] and logics like TCTL [ACD93], Timed LTL [AH89], and Duration Calculus [ZH04] provide techniques to model these systems and specify their properties.

Using these techniques in isolation can lead to inconsistent specifications. If parts of the system are described by a behavioural language and other parts by a state-based language, a clean semantics is needed that describe how these parts interact with each other. Therefore, the combination of behavioural and state-based techniques has been researched. The results are languages like CSP-OZ [Fis00], TCOZ [MD98], and RT-Z [Süh99], that have their own syntax and semantics. Ideally, the definition of the semantics reuses the semantics of the individual languages, and describes only the way they are combined.

To extend state-based and behavioural techniques with real-time aspects, different approaches exist. One approach is unifying a state based language and Timed CSP [DS95], an extension of CSP with a time-out operator. In this thesis we take a different approach. We combine CSP-OZ, which integrates CSP and Object-Z, with Duration Calculus [ZH04], a technique to specify real-time systems.

Communicating Sequential Processes (CSP) is a specification language

for communicating sequential processes. It was introduced by Hoare [Hoa78, Hoa85]. The central concepts of CSP are synchronous communications over channels between different components and parallel compositions. Different operational and denotational semantics are defined, and their consistency is backed up by a mathematical theory in [Ros97]. The model checker FDR [Ros94] enables automatic verification for CSP specification.

The language Z [Spi88] is a mathematical notation developed by Abrial and others at the Programming Research Group of the Oxford University Computing Laboratory (OUCL) since the late 1970s. It defines notations for logical operations, quantifiers, sets, and functions. Furthermore, a schema calculus is defined, which provides a structured way to represent large state space and operations. An object-oriented extension is Object-Z by Smith [Smi00]. Tool support for Z and Object-Z exists in form of type checkers [Joh96] and theorem provers like Z/EVES [Saa97] and Isabelle/HOL-Z [Kol97, SKS02]. The Community Z Tools (CZT) provide a library for parsing and type checking Z specifications.

Duration Calculus (abbreviated to DC) is a real-time interval logic with an extensive proof system (calculus). It was initially developed in the context of the ProCoS (Provably Correct Systems) project to describe the behaviour of real-time systems. To represent state variables that change over time, the notion of observables is used. Their semantics is given by functions $\text{Time} \rightarrow D$ providing a value from the data domain D for each point in time. The time domain can be discrete ($\text{Time} = \mathbb{N}$) or dense ($\text{Time} = \mathbb{R}$). Verification support for discrete time is given through the validity checker DCVALID [Pan01]. For dense time, embeddings of Duration Calculus into the theorem provers PVS and Isabelle are provided by Skakkebak [Ska94] and Heilmann [Hei99].

The combination CSP-OZ-DC should allow reusing existing tools and verification methods for each of the individual languages. For example, refinement proofs for one part should be transferable to the full CSP-OZ-DC specification. Therefore, the combined language must allow modular reasoning. We define the semantics of the combined language based on the semantics of the constituent languages. Modular reasoning is also useful to verify complex specification consisting of many CSP-OZ-DC classes.

In this thesis, we give two semantics for the combined specification language CSP-OZ-DC. In chapter 3 we define a trace semantics. Each of the individual languages CSP, Object-Z, and DC already has a trace seman-

tics, but the traces come from different domains. Therefore, they have to be transformed to a common shape. The traces permitted by a combined CSP-OZ-DC specification are those that are permitted by the CSP, the Object-Z, and the Duration Calculus part. This definition of trace semantics allows modular reasoning.

In chapter 5, we give an equivalent operational semantics for CSP-OZ-DC. The semantics is based on a new automaton model, called *phase event automata*. When these automata run in parallel, they synchronise on both their state space and their events. To implement real-time properties, these automata use the concept of clocks from Alur and Dill [AD94]. The operational semantics of a CSP-OZ-DC class is given by the parallel composition of three automata, each implementing one of the parts of the class: the CSP part, the Object-Z part and the DC part. We show the correctness of the translation by comparing the runs of the automata with the trace semantics of CSP-OZ-DC.

The main technical result is the translation of a subset of Duration Calculus, called *counterexample formulae*, to phase event automata. In 5.3 we give an explicit construction of the automaton. We prove the correctness of the translation by showing that each interpretation accepted by the automaton satisfies the DC formulae and vice versa.

The operational semantics enables model checking techniques. Model checking [CES86] is a verification method that has been successfully applied on finite state systems to prove their correctness. It is fully automatic and does not require ingenious proofs. The first model checkers used explicit state techniques, enumerating all reachable states of the system. If a state is found that violates a property, a counterexample is produced that shows how to reach this state. When all reachable states have been visited without finding a state that violates the property, the model checker assures that the system is safe with respect to the property.

The state of the art in automatic verification is dramatically increased by symbolic techniques. Instead of enumerating all reachable states by giving exact values of the state variables, symbolic techniques [BCM⁺90, McM93] use formulae, which represent sets of reachable states. In some cases, these symbolic techniques may be employed even for infinite state systems. A popular example is model checking of timed automata [YPD94] where zones, which are infinite sets of clock valuations, are described by a finite formula.

In CSP-OZ-DC the sources of infinity are threefold: CSP processes can introduce infinitely many control locations, Object-Z can introduce an infi-

nite data space by declaring variables in an infinite data domain, and Duration Calculus introduces real-valued time. We do not examine systems with an infinite CSP part in this thesis, but for systems with infinite data space and real-valued time, we present model checking techniques in chapter 6. We consider two different approaches. For some systems, we can abstract from the data space of a phase event automaton. This produces a timed automata, which can be checked using popular tools like Uppaal [BDL04] and Kronos [Yov97]. These tools prove properties like reachability. For systems with infinite data space, we present a different approach. The phase event automaton is translated into transition constraint systems that represent the transition relation by a first-order formula. A bounded model checker finds counterexamples that violate the formula. Another model checker for transition constraint systems is the abstraction refinement model checker ARMC [Ryb02]. It creates a finite abstraction of the system, which is refined based on spurious counterexamples. These are counterexamples in the abstract system that are not present in the concrete systems. We show that this model checker can be applied to phase event automata built from a CSP-OZ-DC class.

This thesis is organised as follows. In chapter 2, we introduce the specification techniques CSP, Object-Z, and Duration Calculus. In chapter 3, we present the combined specification technique CSP-OZ-DC, and give the trace semantics for this language. Chapter 4 introduces phase event automata, their parallel composition, and their semantics. In chapter 5, we present the translation of CSP-OZ-DC classes to phase event automata. The main result is the translation of a subclass of Duration Calculus, called counterexample formulae, to phase event automata. In chapter 6, we sketch the implementation of the translation, and present the two different model checking approaches. In chapter 7 we summarise the results of this thesis and discuss topics for future work. The appendix contains grammar rules for CSP-OZ-DC extending the syntax rules for Z.

2. CSP, Object-Z and Duration Calculus

Contents

2.1. CSP	5
2.1.1. Syntax	6
2.1.2. Operational Semantics	6
2.2. Z	7
2.2.1. Syntax	9
2.2.2. Type Checking	11
2.2.3. Semantics	12
2.2.4. Boolean Type	13
2.2.5. Object-Z	14
2.2.6. Reference and Value Semantics	15
2.3. Duration Calculus	16
2.3.1. Syntax of Duration Calculus	17
2.3.2. Semantics of Duration Calculus	18
2.3.3. Abbreviations	19
2.3.4. Embedding Z into Duration Calculus	20
2.3.5. Embedding Events Into State Variables	20
2.3.6. Counterexample Traces	21
2.3.7. Satisfiability of DC is Semi-Decidable	24

This chapter introduces the formal techniques CSP, Object-Z, and Duration Calculus.

2.1. CSP

CSP is a specification language for communicating sequential processes. It was introduced by Hoare [Hoa78, Hoa85]. The language is used to describe behaviour of sequential and parallel processes in terms of the events they communicate. The events can be structured as (c, v) where c is a communication channel and v the communicated value. The set of events a process can communicate is also called its alphabet and denoted by A .

Events are instantaneous; the real-time a communication needs is abstracted into a single moment. The communication between processes running in parallel is synchronous; both the sending and the receiving processes take the associated transition at the same time.

2.1.1. Syntax

The following simple grammar outlines the basic syntax of CSP:

$$P ::= Stop \mid Skip \mid a \rightarrow P \mid P_1 \sqcap P_2 \mid P_1 \square P_2 \mid \\ P_1 \circlearrowleft P_2 \mid P_1 \parallel_A P_2 \mid P \setminus A \mid X$$

The process *Stop* represents a deadlock, i. e., a process that never participates in any communication. The process *Skip* terminates immediately. Termination is represented by communicating a reserved termination event \checkmark . The process $a \rightarrow P$ communicates an event a and behaves like process P afterwards. The construct $P_1 \sqcap P_2$ represents a process that behaves either as P_1 or as P_2 . The choice is done internally. In contrast there is the external choice $P_1 \square P_2$ where the first visible event of a component running in parallel determines which alternative, P_1 or P_2 , has to be taken. Sequential composition is represented as $P_1 \circlearrowleft P_2$. Whenever the first process P_1 communicates the termination event \checkmark , process P_2 is activated.

The constructs $P_1 \parallel_A P_2$ represents the parallel composition of the two processes P_1 and P_2 with synchronisation alphabet A . The alphabet A is the set of events that P_1 and P_2 have to take synchronously. Only if both processes agree on those events, they can take a simultaneous transition. Both processes can communicate events that are not in the alphabet A without having the other process cooperating.

2.1.2. Operational Semantics

The semantics is given as structured operational semantics (SOS) in the style of Plotkin [Plo81]. It is represented as a labelled transition system $L = (Q, \Sigma, q_0, \longrightarrow)$. The sets of states Q are CSP processes, q_0 is an initial CSP process. The set Σ is the set of all events occurring in the alphabet of q_0 including the reserved events \checkmark and τ . The transition relation $\longrightarrow \subseteq (Q \times \Sigma \times Q)$ is inductively defined by rules. We write $P \xrightarrow{a} P'$ for $(P, a, P') \in \longrightarrow$. Examples of rules for \longrightarrow are

$$\frac{}{(a \rightarrow P) \xrightarrow{a} P} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \underset{A}{\parallel} Q \xrightarrow{a} P' \underset{A}{\parallel} Q'} \text{ where } a \in A.$$

The first rule allows the prefixing process $a \rightarrow P$ to communicate the event a . It then continues executing process P . The second rule describes synchronised parallel communication: If process P and process Q can both evolve with an a -transition and a is in the synchronisation alphabet A then $P \underset{A}{\parallel} Q$ can evolve with an a -transition.

2.2. Z

The language Z is a mathematical notation developed by Abrial and others at the Programming Research Group of the Oxford University Computing Laboratory (OUCL) since the late 1970s. It defines rigid notations for logical operations, quantifiers, sets, and functions. Furthermore, a schema calculus is defined, which is useful to encapsulate related variables into a schema, and to define operations over schemas that change the variables inside. In [Spi88], a first step towards standardisation of Z was done in form of a reference manual. In 2002 the Z language has become an ISO/IEC standard [ISO02].

The mathematical toolkit provides a standard notation for the usual set operators, e. g., $\cup, \cap, \subseteq, \in, \emptyset$. The power set of a set X is denoted by $\mathbb{P}X$, the Cartesian product of X and Y as $X \times Y$. Furthermore different symbols for total function $X \rightarrow Y$, injective functions $X \mapsto Y$, partial functions $X \mapsto Y$, relations $X \leftrightarrow Y$ are provided. All functions in Z are represented as sets of tuples, thus their type is $\mathbb{P}(X \times Y)$.

For relations $F : X \leftrightarrow Y$ (note that a function is also a relation) and $A \subseteq X, B \subseteq Y$, the Z language uses the notation $A \triangleleft F$ to denote the restriction of F on the domain A . The restriction of F on the complement of A is denoted by $A \triangleleft F$. Similar notations, $F \triangleright B$ and $F \triangleright B$, are used for restrictions on the range B and its complement. These operators are defined as follows.

$$\begin{aligned} A \triangleleft F &== F \cap (A \times Y) \\ A \triangleleft F &== (X \setminus A) \triangleleft F \\ F \triangleright B &== F \cap (X \times B) \\ F \triangleright B &== F \triangleright (Y \setminus B) \end{aligned}$$

The schema calculus is used to group related variables together, just as the record construct does in many programming languages. A schema is written as $[SchemaText]$ where $SchemaText$ consists of a *DeclPart* and an optional *Predicate* followed by a vertical bar. The *DeclPart* is used to declare new variables. The *Predicate* acts as a global invariant restricting the allowed values of the variables. An example of a schema is:

$PolarCoord$
$\phi, r : \mathbb{R}$
$r \geq 0$
$0 \leq \phi < 2\pi$
$r = 0 \Rightarrow \phi = 0$

This schema is equivalent to

$$PolarCoord == [\phi, r : \mathbb{R} \mid r \geq 0 \wedge 0 \leq \phi < 2\pi \wedge (r = 0 \Rightarrow \phi = 0)]$$

which defines *PolarCoord* as the set of all *bindings* (mappings from the names ϕ and r to their respective types) that satisfy the condition given after the \mid symbol. For example, one element of *PolarCoord* is the binding $\langle \phi == 0, r == 0 \rangle$. The variables of the schema can be accessed by their names. For a binding $p : PolarCoord$, the variables are accessed by the notation $p.r$ and $p.\phi$.

The syntax of the schema calculus is integrated into the mathematical notation. For example, universal quantification in *Z* has the following syntax:

$$\forall SchemaText \bullet Predicate \quad [\text{universal quantification}]$$

Its semantics is **true** if and only if the *Predicate* is true for all variable bindings defined by *SchemaText*. Wherever a *DeclPart* is expected, one can also include a previously defined schemas, e. g., $\forall PolarCoord \mid r = 0 \bullet \phi = 0$.

An operation on a schema type S is a schema that relates the pre and post states of S . Conventionally, the post state of S is denoted S' , which is the schema S with all variables primed, e. g.,

$$PolarCoord' == [\phi', r' : \mathbb{R} \mid r' \geq 0 \wedge 0 \leq \phi' < 2\pi \wedge (r' = 0 \Rightarrow \phi' = 0)].$$

An operation can be written as a schema that includes the pre and post states. An operation schema can have further input and output values that are conventionally decorated by ? and ! respectively. The inclusion of pre and post states can be abbreviated by ΔS , which is defined as $\Delta S == [S; S']$. The following paragraph defines the operation *Scale* on *PolarCoord*. It takes an input parameter *factor?* and scales the coordinate by multiplying the *r* parameter.

$\begin{array}{l} \textit{Scale} \\ \Delta \textit{PolarCoord} \\ \textit{factor?} : \mathbb{R} \\ \hline r' = r * \textit{factor?} \\ \phi' = \text{if } \textit{factor?} = 0 \text{ then } 0 \text{ else } \phi \end{array}$
--

There is a large tool support base for Z, ranging from typesetting Z specifications over type-checkers [Joh96] up to theorem provers, e. g., HOL-Z [Kol97] and Z/EVES [Saa97].

2.2.1. Syntax

The basic entities in a Z specification are paragraphs. A paragraph in Z declares new symbols and fixes their meaning. In [ISO02] the following paragraphs among others are defined:

$\textit{Paragraph} ::= [\textit{NAME}]$	– given types
$\left\{ \begin{array}{l} \textit{DeclPart} \\ \hline \textit{Predicate} \] \end{array} \right.$	– axiomatic description
$\left\{ \begin{array}{l} \textit{NAME} \hline \textit{DeclPart} \\ \hline \textit{Predicate} \] \end{array} \right.$	– schema definition
$\textit{NAME} == \textit{Expression}$	– horizontal definition
$\textit{FreeType} \ \& \ \dots \ \& \ \textit{FreeType}$	– free types

A given type $[\textit{NAME}]$ paragraph declares a type *NAME*. Semantically, a type is a non-empty set of objects. There is no restriction on the number of objects in the set (can be finite or infinite), nor the representation of the

objects. An axiomatic description can be used to introduce new variables, namely those declared by *DeclPart*. The variables are assigned a value such that *Predicate* holds. This value is fixed through the remainder of the specification.

As explained earlier the schema

<i>NAME</i>
<i>DeclPart</i>
<i>Predicate</i>

is just another syntax for $NAME == [DeclPart \mid Predicate]$, which is a horizontal definition. Furthermore, a horizontal definition $NAME == Expression$ is equivalent to the axiomatic description

$$\mid NAME : \{Expression\}$$

which introduces the new variable *NAME*. Its value must be the only value in the singleton set, thus the value of *Expression*.

A *FreeType* paragraph declares one or more types in a BNF like syntax:

$$\begin{aligned} FreeType &::= NAME \text{ ' ::= ' } Branch \text{ ' | ' } \dots \text{ ' | ' } Branch \\ Branch &::= NAME [\langle \langle Expression \rangle \rangle] \end{aligned}$$

For example, a binary tree with numbers stored in its leaves can be expressed as a free type as follows:

$$Tree ::= branch \langle \langle Tree \times Tree \rangle \rangle \mid leaf \langle \langle \mathbb{Z} \rangle \rangle$$

A free type declaration is equivalent to a given type paragraph, followed by an axiomatic description that restricts the structure of the type. For example, the free type *Tree* is equivalent to

$$[Tree]$$

<i>branch</i> : $Tree \times Tree \mapsto Tree$
<i>leaf</i> : $\mathbb{Z} \mapsto Tree$
$\text{ran } branch \cap \text{ran } leaf = \emptyset$
$\forall T : \mathbb{P} Tree \mid branch(T \times T) \cup leaf(\mathbb{Z}) \subseteq T \bullet T = Tree$

The first paragraph declares *Tree* as given type, the axiomatic description declares the constructors of the free type: the functions *branch* and *leaf*. These functions must be injective (denoted by \mapsto) and the range of these functions must be disjoint. Furthermore *Tree* is the smallest set which is closed under the application of *branch* and *leaf*.

2.2.2. Type Checking

In *Z* each variable and expression has a type that can be determined statically. In [ISO02] types are constructed by the following grammar:

$$\begin{aligned}
 \textit{Type} ::= & [NAME, \dots, NAME] \textit{Type2} && \text{-- generic type} \\
 & | \textit{Type2} \\
 \textit{Type2} ::= & \text{GIVEN } NAME && \text{-- given type} \\
 & | \text{GENTYPE } NAME && \text{-- generic parameter type} \\
 & | \mathbb{P} \textit{Type2} && \text{-- power set type} \\
 & | \textit{Type2} \times \dots \times \textit{Type2} && \text{-- Cartesian product type} \\
 & | [\textit{Signature}] && \text{-- schema type} \\
 \\
 \textit{Signature} ::= & [NAME : \textit{Type}; \\
 & \dots; NAME : \textit{Type}] && \text{-- type signature}
 \end{aligned}$$

The basic operations on types are as follows: \mathbb{P} creates the power set type, whose objects are all sets of elements with the basic type. The operator \times creates the Cartesian product type. The type **GIVEN** *NAME* is the type of the elements in the set declared by the given type paragraph [*NAME*]. Besides declaring a new type the paragraph also creates a new constant *NAME* that is the set of all objects of type **GIVEN** *NAME*. Hence the type of *NAME* is \mathbb{P} **GIVEN** *NAME*. In [ISO02] the set of all numbers is declared as a given type $[\mathbb{A}]$, where \mathbb{A} stands for “arithmos”. Hence, the type of \mathbb{R} is \mathbb{P} **GIVEN** \mathbb{A} .

Schema types are used for *bindings* of a *Z* schema. For example the elements of *PolarCoord* are bindings with the type $[\phi : \text{GIVEN } \mathbb{A}; r : \text{GIVEN } \mathbb{A}]$. The schema *PolarCoord* is a set of bindings therefore its type is $\mathbb{P}[\phi : \text{GIVEN } \mathbb{A}; r : \text{GIVEN } \mathbb{A}]$.

Generic types are used for generic set operators like $\subseteq, \cup, \cap, \in$, etc. For example, the operator \in is a relation between objects and the corresponding

sets, which can be expressed as $[X] \mathbb{P}(\text{GENTYPE } X \times \mathbb{P} \text{GENTYPE } X)$. Here the first part $[X]$ denotes that \in has a generic type parameter X that need to be instantiated by a concrete type when \in is used in another expression. The types $\text{GENTYPE } X$ are placeholders and replaced by the concrete type when the type is instantiated. A relation such as \in is represented as set of tuples. In this case \in is defined as relation between X and $\mathbb{P} X$, the power set of X .

The symbols $\vdash^{\mathcal{E}}$, $\vdash^{\mathcal{P}}$ and $\vdash^{\mathcal{D}}$ are used to denote that an expression, a predicate, or a paragraph is well-typed. For an expression $\Sigma \vdash^{\mathcal{E}} e \circ \tau$ denotes that e is well-typed and has type τ in the type environment Σ . The type environment is a mapping from *NAME* to *Type* and assigns a type to the variables that are declared outside the expression. The Z standard uses rules as the following one to define the process of type checking:

$$\frac{\Sigma \vdash^{\mathcal{E}} e \circ \tau_1}{\Sigma \vdash^{\mathcal{E}} \mathbb{P} e \circ \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P} \alpha \\ \tau_2 = \mathbb{P} \mathbb{P} \alpha \end{array} \right)$$

This rule states that $\mathbb{P} e$ is well typed and its type is $\tau_2 = \mathbb{P} \mathbb{P} \alpha$ if e is well typed and its type is $\tau_1 = \mathbb{P} \alpha$. The type variable α can be instantiated with any type.

2.2.3. Semantics

The ISO/IEC standard for Z [ISO02] defines the semantics for Z in a meta language based on Zermelo-Fraenkel set theory. The underlying set theory is not typed. It uses the following sets to give semantics for Z:

- *NAME* denotes the set of all valid Z identifiers. Names can be strings of letters, optionally decorated with subscripts, ?, ! or ', or mathematical symbols. For example, *PolarCoord*, *factor?*, and \subseteq are valid names.
- \mathbb{U} denotes the universe, a set of all possible semantic values. It is closed under construction of power set and Cartesian product. This set also provides semantic values for generic symbols that can be applied on arbitrary types, e.g., \subseteq .
- \mathbb{W} denotes the world, the set of all values from universe that are not values of generic symbols.

- **Model** denotes the set of models, $\text{Model} == \text{NAME} \mapsto \mathbb{U}$. A model is thus a finite function, that assigns to variables values from the universe.

The Z standard uses the symbols $\llbracket \cdot \rrbracket^{\mathcal{E}}$, $\llbracket \cdot \rrbracket^{\mathcal{T}}$, $\llbracket \cdot \rrbracket^{\mathcal{P}}$, $\llbracket \cdot \rrbracket^{\mathcal{D}}$, to denote the semantics of expressions, types, predicates, and paragraphs. The semantics of an expression $\llbracket e \rrbracket^{\mathcal{E}} M$ denotes the semantic value of e in the model M . This is a value from the set \mathbb{W} . The semantics of the type τ in the model M is denoted by $\llbracket \tau \rrbracket^{\mathcal{T}} M$. This is the carrier set, i. e., the set of all elements from \mathbb{W} that have type τ . The semantics of a predicate $\llbracket p \rrbracket^{\mathcal{P}}$ is the set of all models in which the predicate is true. The semantics of a paragraph $\llbracket D \rrbracket^{\mathcal{D}}$ is a relation from models to models. A tuple (M, M') is in the relation if M' is an extension of the model M defining values for the variables declared in D . There may be multiple consistent extensions of M or none. Therefore $\llbracket D \rrbracket^{\mathcal{D}}$ is a relation, not a function.

2.2.4. Boolean Type

The Z standard [ISO02] has no notion of Boolean variables. However, it is often useful to have a Boolean type in the specification. A declaration $\mathbb{B} == \{true, false\}$ is not syntactically correct in Z because the elements of a set must be expressions but *true* and *false* are predicates. Although an expression cannot be a predicate, the Z standard explicitly allows the reverse. A predicate can be a schema expression e . The truth value of this expression is $\theta e \in e$. The operator θ creates a binding for the schema denoted by e . The values for the variables declared in the schema e are taken from the current context. Therefore, the expression $\theta e \in e$ means that the current values of the variables occurring in the schema e satisfies the predicate part of this schema.

If we use the simplest schema, i. e., the empty schema $e == []$, then the semantic value for the expression e is the set $\{\langle \rangle\}$ containing the empty binding. The predicate $\theta e \in e$ is true because θe is the empty binding $\langle \rangle$. Falseness can be expressed by the schema $[] \text{ false}$, which denotes the empty set (with the same type as $[]$). The Boolean type can be defined as the set of all subsets of the empty schema $[]$:

$$\mathbb{B} == \mathbb{P}[]$$

Note that \mathbb{B} has only two elements namely $[]$ and $[] \text{ false}$. The Boolean type can be mostly used as expected, e. g.

$odd : \mathbb{Z} \rightarrow \mathbb{B}$ $b_1, b_2, b_3 : \mathbb{B}$
$\forall z : \mathbb{Z} \bullet odd(z) \Leftrightarrow z \bmod 2 \neq 0$ $b_2 \Leftrightarrow true$ $b_1 \Leftrightarrow (b_2 \vee b_3) \wedge \neg (b_2 \wedge b_3)$

Care has to be taken because an expression cannot be a predicate. The equations $x = (n > 3)$ and $odd = \lambda z : \mathbb{Z} \bullet (z \bmod 2 \neq 0)$ are not syntactically correct. However, they can be written as $x = [| n > 3]$ and $odd = \lambda z : \mathbb{Z} \bullet [| z \bmod 2 \neq 0]$.

2.2.5. Object-Z

A Z specification can define several operations for a schema type. In the plain Z syntax these operations are scattered all over the specification. Modern programming languages often use the object-oriented paradigm where operations are directly declared along with the data-type. For Z an object-oriented extension is Object-Z [Smi00]. It was developed at the Software Verification Research Center in the University of Queensland.

The following example shows the definition of *PolarCoord* as Object-Z class.

<i>PolarCoord</i>	
$\phi, r : \mathbb{R}$	
$r \geq 0$ $0 \leq \phi < 2\pi$ $r = 0 \Rightarrow \phi = 0$	
<i>Scale</i>	
$\Delta(r, \phi)$ $factor? : \mathbb{R}$	
$r' = r * factor?$ $\phi' = \text{if } factor? = 0 \text{ then } 0 \text{ else } \phi$	

The state schema declaring the variables ϕ and r and the operation *Scale* are combined in a larger class declaration. The class declaration gives the

name of the class, here *PolarCoord*. The state schema does not have a name in Object-Z. All named schemas in a class are operations, e. g., *Scale* in the example above. Every operation schema implicitly includes the pre and post states of state schema. The Δ list is used to list all variables that are changed by the operations; variables that are not listed are not changed.

2.2.6. Reference and Value Semantics

In [DKRS91] and [Smi92] Object-Z classes are given value semantics. A class was identified by its state schema. For example, the semantic value of *PolarCoord* as defined above is the same as if it had been defined as a schema $PolarCoord == [\phi, r : \mathbb{R} \mid \dots]$. Two classes $a, b : PolarCoord$ are equal if $a.r = b.r$ and $a.\phi = b.\phi$. The expression $a.r'$ is the same as $a'.r$; the schema $a.Scale$ is an abbreviation for the following schema:

$a.Scale$ <hr/> $a, a' : PolarCoord$ $factor? : \mathbb{R}$ <hr/> $a'.r = a.r * factor?$ $a'.\phi = \text{if } factor? = 0 \text{ then } 0 \text{ else } a.\phi$
--

Later, a reference semantics for Object-Z was proposed [Gri98, Smi00] where an object is identified by its reference. Two objects with equal contents are no longer equal. On the other hand, even if the contents of an object changes due to an operation, the object reference does not change. With the reference semantics, the expressions $a.r'$ and $a'.r$ represent different values: $a.r'$ denotes the post state of r in the object that is referenced by a in the pre state, while $a'.r$ denotes the pre state of r in the object that is referenced by a' , the post state value of a .

For example, consider a polygon class that stores its vertices as a sequence of polar coordinates.

$Polygon$ <hr/> <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> $vertices : \text{seq } PolarCoord$ </div> <hr/> \dots

Using the value semantics, an operation that scales the polygon can be specified as follows:

$$\frac{\text{Scale} \quad \Delta(\text{vertices}) \quad \text{factor?} : \mathbb{R}}{\forall i : \text{dom } \text{vertices} \bullet \text{vertices}(i).\text{Scale}}$$

With the reference semantics there is a important difference. The sequence *vertices* does not change anymore, as its contents, the object references, are still the same. Only their contents change.

$$\frac{\text{Scale} \quad \Delta() \quad \text{factor?} : \mathbb{R}}{\forall i : \text{dom } \text{vertices} \bullet \text{vertices}(i).\text{Scale}}$$

With the reference semantics it is possible that two different objects *A* and *B* share a reference to the same object *C*. If *B* changes object *C*, this change also has effects on object *A*. This makes modular reasoning much more difficult. A prerequisite for modular reasoning is that the operation on object *C* by object *B* is globally visible.

We follow the approach of Fischer [Fis00] and use the reference semantics only for CSP-OZ-DC classes. In these classes the operations are visible by means of CSP events. Thus synchronisation issues caused by shared parallel access to the same class can be resolved with CSP methods. If Object-Z classes are used, the underlying semantics is the simple value semantics. Thus Object-Z classes can be used to implement structured data types, e. g., stacks, lists, records, etc.

2.3. Duration Calculus

Duration Calculus [ZH04] (abbreviated to DC) is a real-time interval logic accompanied with a calculus. It was initially developed in the context of the ProCoS project (ESPRIT BRA 3104 and 7071) to describe the behaviour of real-time systems.

2.3.1. Syntax of Duration Calculus

We define the basic syntax and semantics in accordance to [HZ97, ZH04]. The syntax of Duration Calculus is constructed from the following set of symbols:

SVar The set of Boolean-valued *state variables* P, Q, R, \dots , also called *observables*. The values of these variables depend on the point in time.

GVar The set of real-valued x, y, z, \dots . The meaning of these variables is time-independent.

FSymb The set of global f^n, g^m, \dots with arities $n, m \geq 0$. The function f^n takes n real values as arguments and returns a real value.

RSymb The set of global p^n, q^m, \dots with arities $n, m \geq 0$. The relation q^n takes n real values as arguments and returns a boolean value.

The set of *state expressions* is defined by the following syntax

$$StateExpr ::= 0 \mid 1 \mid SVar \mid \neg StateExpr \mid StateExpr \vee StateExpr$$

The symbols 0 and 1 stand for **false** and **true** respectively. A state expression is a boolean connection of state variables and thus is time-dependent. The set of Duration Calculus *terms* is defined by the following syntax:

$$Term_{DC} ::= \int StateExpr \mid GVar \mid f(Term_{DC}, \dots, Term_{DC})$$

Here \int is the integration operator of Duration Calculus, which measures the duration the state expression evaluates to true. The meaning of a term is a real value computed using a given time interval. The set of Duration Calculus *formulae* is defined by the following syntax:

$$\begin{aligned} Formula_{DC} ::= & p(Term_{DC} \times \dots \times Term_{DC}) \\ & \mid \neg Formula_{DC} \mid Formula_{DC} \vee Formula_{DC} \\ & \mid Formula_{DC} \wedge Formula_{DC} \\ & \mid \exists GVar \bullet Formula_{DC} \mid \exists SVar \bullet Formula_{DC} \end{aligned}$$

We allow relations over terms, Boolean connectives (the operators \wedge, \Rightarrow , and \Leftrightarrow are defined as abbreviations as usual), and quantification over global

and state variables. Furthermore, formulae may be connected by the chop operator, denoted by \frown . Like for terms, the meaning of a formula also depends on a time interval. The chop operator is used to chop time intervals into two parts where each part needs to fulfil a different formula. In [HZ97] quantification over state variables is not allowed, however, it can be easily added to the language.

2.3.2. Semantics of Duration Calculus

The semantics of a Duration Calculus formula depends on the meaning of state variables, global variables, function symbols and relation symbols. The meaning of function and relation symbols is fixed and defined by total functions $\tilde{f}^n : \mathbb{R}^n \rightarrow \mathbb{R}$ and $\tilde{p}^n : \mathbb{R}^n \rightarrow \mathbb{B}$. For our purpose, it suffices to use constants (functions with arity zero) $k : \mathbb{R}$, the functions $+$, $-$, and the relations $<$, \leq , $=$, \geq , $>$ with their standard meaning.

The values for the state variables $P \in SVar$ is given by an interpretation $\mathcal{I}[[P]] : \mathbf{Time} \rightarrow \{0, 1\}$, that assigns to each point in time a value 0 or 1 denoting whether P holds at that time. We use dense time: $\mathbf{Time} = \mathbb{R}^+$. For $\mathcal{I}[[P]]$, finite variability is required, i. e., on every finite interval $[0, t]$ the value of $\mathcal{I}[[P]]$ may change only a finite number of times. The values for the global variables $X \in GVar$ is given by a valuation $\mathcal{V} : GVar \rightarrow \mathbb{R}$.

The interpretation \mathcal{I} is canonically extended to state expressions. The extension is defined inductively:

$$\begin{aligned} \mathcal{I}[[0]](t) &= 0 && \text{for all } t \in \mathbf{Time}, \\ \mathcal{I}[[1]](t) &= 1 && \text{for all } t \in \mathbf{Time}, \\ \mathcal{I}[[\neg S]](t) &= 1 - \mathcal{I}[[S]](t) && \text{for all } t \in \mathbf{Time}, \\ \mathcal{I}[[S_1 \vee S_2]](t) &= \max(\mathcal{I}[[S_1]](t), \mathcal{I}[[S_2]](t)) && \text{for all } t \in \mathbf{Time}. \end{aligned}$$

Let \mathbf{Val} denote the set of all valuations $GVar \rightarrow \mathbb{R}$ and \mathbf{Intv} denote the set of all intervals $[b, e]$ with $b, e \in \mathbf{Time}$, $b \leq e$. The meaning of a *term* depends on the interpretation \mathcal{I} , the valuation of global variables $\mathcal{V} \in \mathbf{Val}$, and a time interval $[b, e] \in \mathbf{Intv}$. The semantics of a term is expressed by $\mathcal{I}[-] : Term_{DC} \rightarrow (\mathbf{Val} \times \mathbf{Intv}) \rightarrow \mathbb{R}$ and is defined inductively by:

$$\begin{aligned} \mathcal{I}[[f S]](\mathcal{V}, [b, e]) &= \int_b^e \mathcal{I}[[S]](t) dt \\ \mathcal{I}[[x]](\mathcal{V}, [b, e]) &= \mathcal{V}(x) \\ \mathcal{I}[[f^n(t_1, \dots, t_n)]](\mathcal{V}, [b, e]) &= \tilde{f}^n(\mathcal{I}[[t_1]](\mathcal{V}, [b, e]), \dots, \mathcal{I}[[t_n]](\mathcal{V}, [b, e])) \end{aligned}$$

The f operator is defined by the Riemann integral of the interpretation for the state expression over the given interval. The Riemann integral is well-defined, because the finite variability of the state variables propagates to state expressions. The remaining definition is straightforward.

The semantics of a formula also depends on the interpretation, the valuation, and the interval. We write $\mathcal{I}, \mathcal{V}, [b, e] \models F$ iff the formula F holds for interpretation \mathcal{I} , valuation \mathcal{V} , and interval $[b, e]$. The inductive definition is as follows.

$\mathcal{I}, \mathcal{V}, [b, e] \models p^n(t_1, \dots, t_n)$	iff $\tilde{p}^n(\mathcal{I}[[t_1]](\mathcal{V}, [b, e]), \dots, \mathcal{I}[[t_n]](\mathcal{V}, [b, e]))$
$\mathcal{I}, \mathcal{V}, [b, e] \models \neg F$	iff not $\mathcal{I}, \mathcal{V}, [b, e] \models F$
$\mathcal{I}, \mathcal{V}, [b, e] \models F_1 \vee F_2$	iff $\mathcal{I}, \mathcal{V}, [b, e] \models F_1$ or $\mathcal{I}, \mathcal{V}, [b, e] \models F_2$
$\mathcal{I}, \mathcal{V}, [b, e] \models F_1 \wedge F_2$	iff there is $m \in \mathbb{R}$ with $b \leq m \leq e$ satisfying $\mathcal{I}, \mathcal{V}, [b, m] \models F_1$ and $\mathcal{I}, \mathcal{V}, [m, e] \models F_2$
$\mathcal{I}, \mathcal{V}, [b, e] \models \exists x \bullet F$	iff there is $v_x \in \mathbb{R}$ with $\mathcal{I}, \mathcal{V} \oplus \{x \mapsto v_x\}, [b, e] \models F$
$\mathcal{I}, \mathcal{V}, [b, e] \models \exists P \bullet F$	iff there is $I_P : \text{Time} \rightarrow \{0, 1\}$ with $\mathcal{I} \oplus \{P \mapsto I_P\}, \mathcal{V}, [b, e] \models F$

The formula F holds on an interval $[b, e]$ for \mathcal{I} (denoted by $\mathcal{I}, [b, e] \models F$) iff $\mathcal{I}, \mathcal{V}, [b, e] \models F$ holds for all valuations \mathcal{V} . An interpretation \mathcal{I} satisfies F (denoted by $\mathcal{I} \models F$) iff for all intervals $[0, t]$ starting at time zero $\mathcal{I}, \mathcal{V}, [0, t] \models F$ holds.

2.3.3. Abbreviations

The special symbol ℓ refers to the length of an interval and is defined by $\ell == f 1$. The formula $\lceil S \rceil == (f S = \ell \wedge \ell > 0)$ abbreviates that state expression S holds almost everywhere in an interval. Here “almost everywhere” means “for all but finitely points”. Furthermore, the length of such an interval can be restricted, e.g., $\lceil S \rceil^{\leq r} == \lceil S \rceil \wedge \ell \leq r$ and $\lceil S \rceil^r == \lceil S \rceil \wedge \ell = r$.

The abbreviation $\diamond F == \mathbf{true} \wedge F \wedge \mathbf{true}$ states that there is a subinterval satisfying F . The dual operator $\square F == \neg \diamond \neg F$ states that all subintervals satisfy F .

2.3.4. Embedding Z into Duration Calculus

As seen in the previous section, one of the basic entities in plain Duration Calculus are state variables, which are mappings from time to the values $\{0, 1\}$. But in our combined specification we want to refer to the entities of the Object-Z class in the Duration Calculus part. To achieve this we will embed Z expressions and predicates into the Duration Calculus.

In an Object-Z class we have the state space, which consists of several variables. As the values of these variables changes over time, we can regard them as functions from time to their domain. Likewise we can regard predicates over Object-Z as functions from time to the boolean values, which matches the interpretation of state expressions in the plain Duration Calculus. As predicates are already closed against logical operators we can use this syntax:

$$\textit{StateExpr} ::= \textit{Predicate}_Z$$

To give meaning to such state expressions an interpretation $\mathcal{I}_Z : \textit{Time} \rightarrow \textit{Model}$ is needed, which assigns a complete Z-model to every single point in time. We require finite variability, i. e., for every interpretation and every finite interval $[0, t]$ there is a partition $0 < t_1 < \dots < t_n = t$, such that \mathcal{I} is constant on each subinterval $]t_i, t_{i+1}[$. We can define the semantics of a state expression S for some point in time t by evaluating the Z predicate S under the model $\mathcal{I}_Z(t)$:

$$\mathcal{I}[\![S]\!](t) = \begin{cases} 1 & \mathcal{I}_Z(t) \models S \\ 0 & \text{otherwise} \end{cases}$$

2.3.5. Embedding Events Into State Variables

State variables in Duration Calculus can only be used for *stable* variables, that keep their value for a nonempty interval. By contrast, an *instant* event that we need for CSP only occurs at a single point in time. Although it is impossible to observe variables at a single point it is possible to observe the exact time a variable changes. For example, in the formula $(\lceil S \rceil \wedge \ell = x) \cap \lceil \neg S \rceil$ the state variable S changes x time units after the beginning of the interval. If an event is identified with the change of a state variable it is possible to express the occurrence of this event.

The above formula involving two non-zero length intervals and a duration x is not convenient. Therefore, we follow the approach suggested in [ZH04] and introduce new formulae that hold for the point intervals where the state variable changes. To simplify the definition we introduce a notation from [ZH96]. The formulae $\nwarrow S$ and $\nearrow S$ describe that a state expression is satisfied before resp. after a given point in time:

$$\begin{aligned} \mathcal{I}, \mathcal{V}, [b, e] \models \nwarrow S &\text{ iff } b = e \wedge \exists m : \mathbb{R} \mid m < b \bullet \mathcal{I}, \mathcal{V}, [m, b] \models [S] \\ \mathcal{I}, \mathcal{V}, [b, e] \models \nearrow S &\text{ iff } b = e \wedge \exists m : \mathbb{R} \mid m > e \bullet \mathcal{I}, \mathcal{V}, [e, m] \models [S] \end{aligned}$$

So $\nwarrow S$ holds on a point interval iff that interval can be extended to the left such that S holds on this extension. Likewise $\nearrow S$ holds on point intervals that can be extended to the right such that S holds on the extension. With this notation one can specify that the variable S changes as follows:

$$\Downarrow S \quad == \quad (\nwarrow \neg S \wedge \nearrow S) \vee (\nwarrow S \wedge \nearrow \neg S)$$

To express that S does not change at some point we can use the negated formula and require that the interval has zero duration.

$$\not\Downarrow S \quad == \quad \neg \Downarrow S \wedge \ell = 0$$

Now we can identify CSP events with a Boolean variables that changes every time the event occurs. We introduce a state variable e for every CSP event. The formulae $\uparrow e$ denotes that an event occurs at a certain point in time. To express that an event does not occur during an interval we can use the formula

$$\Box e \quad == \quad \neg (\ell > 0 \wedge \uparrow e \wedge \ell > 0).$$

This formula states that there is no change of e inside the interval. It allows a change of e at the end points of the interval, though. It also holds for all point intervals.

2.3.6. Counterexample Traces

The full language of Duration Calculus is too expressive to be implemented. As an example consider the formula

$$\Box((\uparrow ev \wedge \ell = 1) \Rightarrow (\mathbf{true} \wedge \not\Downarrow ev)) \quad (2.1)$$

This formula states that if an event ev occurs it does not occur again exactly one second later. It may occur earlier or later though. Unfortunately, there is no way to implement this behaviour. The reason is that there is no restriction how often the event ev can occur within one second. One has to set a new timer whenever this event occurs which requires an unbounded number of timers. It also requires exact measurement of the duration as it is allowed to occur after 0.9999 and after 1.0001 seconds. Therefore this formula is not useful in practise.

As we cannot handle a formula like this we allow only formulae that are of a certain shape that makes it possible to implement them. One class of formulae that was designed to meet this requirement is the class of implementables [Rav94, SO99]. Ravn introduces the *followed-by* operator $F \longrightarrow [P]$, which states that every interval satisfying F is always followed by an interval where P holds. It is defined as follows¹:

$$F \longrightarrow [P] == \neg \diamond (F \wedge [\neg P]).$$

The operator can be further modified by time constraints to restrict the length of the interval where F holds:

$$F \xrightarrow{t} [P] == \neg \diamond (F \wedge \ell = t \wedge [\neg P]) \quad (\text{leads-to})$$

$$F \xrightarrow{\leq t} [P] == \neg \diamond (F \wedge \ell \leq t \wedge [\neg P]) \quad (\text{bounded followed-by})$$

This operator is useful to define patterns for progress, stability and synchronisation. For example, the progress pattern, $[P \wedge Q] \xrightarrow{t} [\neg P]$, states that if $P \wedge Q$ holds for t seconds, it must be followed by an interval where $\neg P$ holds. Thus $P \wedge Q$ cannot hold longer than t seconds. The following class of DC implementables are taken from [SO99].

Definition 2.1. Let P , P_1 , and Q be state expressions. The following formulae are *implementables*.

1. *Initialisation*: $[\top] \vee ([P] \wedge \mathbf{true})$ demands that the system starts in a state satisfying P .
2. *Sequencing*: $[P] \longrightarrow [P \vee P_1]$ demands that the system can evolve from state P only to a state satisfying P_1 .

¹In [Rav94] it is defined as $\square(F \wedge \ell = r \wedge \ell > 0 \Rightarrow \ell = r \wedge [P] \wedge \mathbf{true})$. The definitions are logically equivalent.

3. *Progress*: $[P] \xrightarrow{t} [\neg P]$ demands that state P must be left after at most t seconds.
4. *Stability*: $[\neg P] \wedge [P \wedge Q] \xrightarrow{\leq t} [P \vee P_1]$ demands that if state P is entered while Q holds it is stable or only evolves to a state satisfying P_1 during the next t seconds. The annotation $\leq t$ can be dropped to allow unbounded stability. If P_1 is **false** state P is stable.
5. *Synchronisation*: $[P \wedge Q] \xrightarrow{t} [\neg P]$ demands that state P must be left within t seconds if Q holds. Note that the progress pattern is a special case of this pattern.

Except for the initialisation pattern, the implementables are built from the followed-by and leads-to operators. Most implementables are therefore abbreviations for formulae of the shape

$$\neg (\text{phase}_1 \wedge \dots \wedge \text{phase}_n),$$

where phase_i is either **true** or $[P]$ for some state expression P . Additionally, the duration of phase_i may be restricted by $\ell \sim t$. We call a formula of the above shape a *counterexample formula* as it forbids bad behaviour by giving a counterexample. A counterexample describes a single behaviour that must not occur. Using this pattern limits the danger of over-specification, because it forbids only a single undesired behaviour.

However, not all formulae of this shape are implementable. For example formula (2.1) can be reformulated as

$$\neg (\mathbf{true} \wedge \downarrow \text{ev} \wedge \ell = 1 \wedge \downarrow \text{ev} \wedge \mathbf{true}). \quad (2.2)$$

It turns out that the problem is the exact length $\ell = 1$. If the length restriction was $\ell < t$ or $\ell \leq t$ the formula would only demand a stable behaviour (event ev should not occur too often). If the restriction was $\ell \geq t$ or $\ell > t$ it would demand that event ev has to occur often enough. These two cases correspond to the *stability* and *progress* implementable. Thus, we only allow length restriction $\ell \sim t$ with $\sim \in \{<, \leq, \geq, >\}$. The syntax of counterexample formulae is defined as follow.

$$\begin{aligned} \text{ce_formula} & ::= \neg (\text{phase} \wedge (\text{phase} \mid \text{events}) \\ & \quad \wedge \dots \wedge (\text{phase} \mid \text{events}) \wedge \mathbf{true}) \\ \text{phase} & ::= (\mathbf{true} \mid [\text{Predicate}])[\wedge \ell \sim t] \end{aligned}$$

$$\begin{aligned}
& [\wedge \boxplus NAME \dots \wedge \boxplus NAME] \\
\sim & ::= \leq | < | > | \geq \\
events & ::= \uparrow NAME \mid \downarrow NAME \\
& \mid events \vee events \mid events \wedge events
\end{aligned}$$

Although the leads-to operator as defined above seems to require equality, it can be replaced by \geq in most cases. It turns out that all implementables can be reformulated as counterexample traces:

Lemma 2.2 (Implementables as Counterexample Traces). *The implementables can be reformulated as equivalent counterexample traces (c.f. [Tap01], lemma 4.2).*

1. $\Box \vee ([P] \wedge \mathbf{true}) \equiv \neg (\neg [P] \wedge \mathbf{true})$
2. $[P] \longrightarrow [P \vee P_1] \equiv \neg (\mathbf{true} \wedge [P] \wedge \neg [P \wedge \neg P_1] \wedge \mathbf{true})$
3. $[P] \xrightarrow{t} \neg [P] \equiv \neg (\mathbf{true} \wedge ([P] \wedge \ell > t) \wedge \mathbf{true})$
4. $\neg [P] \wedge [P \wedge Q] \xrightarrow{\leq t} [P \vee P_1] \equiv$
 $\neg (\mathbf{true} \wedge \neg [P] \wedge ([P \wedge Q] \wedge \ell < t) \wedge \neg [P \wedge \neg P_1] \wedge \mathbf{true})$
5. $[P \wedge Q] \xrightarrow{t} \neg [P] \equiv \neg (\mathbf{true} \wedge ([P \wedge Q] \wedge \ell \geq t) \wedge [P] \wedge \mathbf{true})$

Proof. Most formulae are expansions of the abbreviations for the operators \longrightarrow and \diamond . The formula for stability uses the equivalence

$$\mathbf{true} \wedge ([Q] \wedge [P]) \wedge \ell \leq t \equiv \mathbf{true} \wedge [Q] \wedge ([P] \wedge \ell < t),$$

the formula for synchronisation uses the equivalence

$$\mathbf{true} \wedge ([P] \wedge \ell = t) \equiv \mathbf{true} \wedge ([P] \wedge \ell \geq t).$$

□

2.3.7. Satisfiability of DC is Semi-Decidable

It is well-known that satisfiability for Duration Calculus is not decidable [ZH04]. The proof provides a reduction of the termination problem for two-counter machines [SS63] to the satisfiability problem of Duration Calculus. The two-counter machine terminates if and only if the translated

formula is satisfiable. Although the termination problem is semi-decidable, this reduction does not yield semi-decidability of the satisfiability problem. We will now show that under certain assumptions it is semi-decidable.

The first assumption is: It is semi-decidable, whether a conjunction $P_1 \wedge \dots \wedge P_n$ of state expressions P_i occurring in F is satisfiable. This is a necessary assumption because otherwise satisfiability of the formula $\lceil P_1 \rceil \wedge \dots \wedge \lceil P_n \rceil$ cannot be semi-decided. The second assumption is: The first order logic part, i. e., the logic over the reals with the function symbols and predicates that are used to build terms and formulae from smaller terms, is semi-decidable. With the method of quantifier elimination [Col75] the first order logic $(\mathbb{R}, +, \cdot)$ is even decidable. This is the logic built from the function symbols $+$, \cdot and the predicates $\geq, \leq, >, <, =$.

Let $\Sigma_F = \mathbb{P} \text{StateExpr}_F$ be the power set of state expressions occurring in a DC formula F . Since there are only finitely many state expressions occurring in F , Σ_F is also finite. For every interpretation \mathcal{I} on every finite interval $[b, e]$ there are only finitely many changes of \mathcal{I} . Let $b = t_0 < \dots < t_n = e$ the points of time where the interpretation changes. Then \mathcal{I} can be represented by the word $w = w_1 \dots w_n \in \Sigma_F^*$ where $w_i = \{P : \Sigma_F \mid \mathcal{I}, [t_{i-1}, t_i] \models \lceil P \rceil\}$.

We call w the untimed part of \mathcal{I} , denoted by $Untime(\mathcal{I}, [b, e])$. Furthermore, for a fixed word w we can translate the question of satisfiability of a formula F into a first-order formula over real numbers using the predicates $=, \leq$, the function $+$, and other predicates and functions that directly occur in F . This observation can be used to semi-decide satisfiability of a DC formula F . A non-deterministic algorithm that accepts the satisfiable formulae works as follows.

1. Guess a word $w \in \Sigma_F^*$ which is the untimed part of some satisfying interpretation \mathcal{I} .
2. Check for each w_i that $(\bigwedge P \in w_i \bullet P) \wedge (\bigwedge P \in \Sigma_F \setminus w_i \bullet \neg P)$ is satisfiable. Abort if it is unsatisfiable.
3. Translate the DC formula F and the word $w = w_1 \dots w_n$ to a first order formula $xlat(F, w, t_0, t_n)$ (defined below) over real numbers.
4. Check that $xlat(F, w, t_0, t_n)$ is satisfiable. Abort if it is unsatisfiable.
5. Accept F .

By $xlat(F, w, t_0, t_n)$ we denote the translation of the DC formula F . The variables occurring free in $xlat(F, w, t_0, t_n)$ are t_0, t_1, \dots, t_n and the free variables of F . The formula $xlat(F, w, t_0, t_n)$ is satisfiable, if and only if there is an interpretation \mathcal{I} with $Untime(\mathcal{I}, [t_0, t_n]) = w$ that satisfies F on the interval $[t_0, t_n]$. The formula is translated into a formula over real numbers referencing the variables t_0, \dots, t_n , the timed part of the interpretation. We define $xlat$ inductively over the syntax of formulae.

$$\begin{aligned}
 xlat(F_1 \wedge F_2, w, b, e) &= xlat(F_1, w, b, e) \wedge xlat(F_2, w, b, e) \\
 xlat(\neg F, w, b, e) &= \neg xlat(F, w, b, e) \\
 xlat(F_1 \hat{\ } F_2, w, b, e) &= \exists m. b \leq m \wedge m \leq e \\
 &\quad \wedge xlat(F_1, w, b, m) \wedge xlat(F_2, w, m, e) \\
 &\quad \text{where } m \text{ is a fresh variable} \\
 xlat(\exists X.F, w, b, e) &= \exists X. xlat(F, w, b, e)
 \end{aligned}$$

Now consider the DC formula $p(T_1, \dots, T_{k_p})$ where p is a predicate of arity k_p and T_i are terms. The terms are translated by $xlat(T_i, w)$ to terms over real numbers referencing the variables t_1, \dots, t_n . The basic idea is to translate $\int P$ to $\sum i : 1..n \mid P \in w_i \bullet t_i - t_{i-1}$, the sum of the lengths of the intervals where P holds. Because w and P are known when $xlat$ is invoked, the resulting term is a fixed sum of terms $t_i - t_{i-1}$. The following diagram illustrates this translation.

$$\begin{array}{c}
 w : \boxed{\begin{array}{|c|c|c|c|} \hline \{P, Q\} & \{Q\} & \{P\} & \{P, Q\} \\ \hline \end{array}} \\
 \begin{array}{ccccccc}
 & t_0 & & t_1 & & t_2 & & t_3 & & t_4 \\
 & & & & & & & & & \\
 xlat(\int P, w) & = & (t_1 - t_0) & + & (t_3 - t_2) & + & (t_4 - t_3)
 \end{array}
 \end{array}$$

However, the translation needs to consider the current interval $[b, e]$. Therefore, we introduce new variables t'_i , which are set depending on t_i and the interval $[b, e]$ to

$$t'_i = \begin{cases} b & t_i < b, \\ t_i & b \leq t_i \leq e, \\ e & t_i > e. \end{cases} \quad (2.3)$$

Hence, the values for t'_i are set to t_i if t_i is inside the considered interval $[b, e]$. Otherwise it is set to b or e . The following diagram illustrates this.

$$\begin{array}{c}
w : \begin{array}{|c|c|c|c|} \hline \{P, Q\} & \{Q\} & \{P\} & \{P, Q\} \\ \hline \end{array} \\
\begin{array}{ccccccc}
t_0 & & t_1 & & t_2 & & t_3 & & t_4 \\
& & & & \text{---} & & \text{---} & & \\
& & & & [b, e] & & & & \\
& & & & | & & | & & \\
& & & & t'_0 = t'_1 & & t'_2 & & t'_3 & & t'_4
\end{array} \\
xlat(f P, w) = (t'_1 - t'_0) + (t'_3 - t'_2) + (t'_4 - t'_3)
\end{array}$$

The variables t'_0, \dots, t'_n are bound with an existential quantifier and their value is fixed according to (2.3).

$$\begin{aligned}
xlat(p(T_1, \dots, T_{k_p}), w, b, e) = \\
\exists t'_0 \dots t'_n \bullet p(xlat(T_1, w), \dots, xlat(T_{k_p}, w)) \wedge \\
\bigwedge_{i=0}^n ((t_i < b \Rightarrow t'_i = b) \wedge (b \leq t_i \leq e \Rightarrow t'_i = t_i) \wedge (t_i > e \Rightarrow t'_i = e))
\end{aligned}$$

Because n , the length of the word w is known, the operator \bigwedge can be expanded when translating the formula. For example, for $n = 1$ it is expanded to

$$\begin{aligned}
&\exists t'_0, t'_1 \bullet p(xlat(T_1, w), \dots, xlat(T_{k_p}, w)) \wedge \\
&((t_0 < b \Rightarrow t'_0 = b) \wedge (b \leq t_0 \leq e \Rightarrow t'_0 = t_0) \wedge (t_0 > e \Rightarrow t'_0 = e)) \wedge \\
&((t_1 < b \Rightarrow t'_1 = b) \wedge (b \leq t_1 \leq e \Rightarrow t'_1 = t_1) \wedge (t_1 > e \Rightarrow t'_1 = e))
\end{aligned}$$

The inductive definition of $xlat$ for terms is as follows.

$$\begin{aligned}
xlat(f P, w) &= \sum i : 1..n \mid P \in w_i \bullet t'_i - t'_{i-1} \\
xlat(X, w) &= X \\
xlat(f(T_1, \dots, T_{k_f}), w) &= f(xlat(T_1, w), \dots, xlat(T_{k_f}, w))
\end{aligned}$$

Using the translation function $xlat$ the algorithm given above semi-decides the satisfiability problem for F .

Proof. Assume that F is satisfiable. Then there is an interpretation \mathcal{I} , a variable valuation \mathcal{V} and an interval $[b, e]$ with $\mathcal{I}, \mathcal{V}, [b, e] \models F$. The algorithm can guess the untimed behaviour $w = \text{Untime}(\mathcal{I}, [b, e])$. By construction of w_i , the formula $(\bigwedge P \in w_i \bullet P) \wedge (\bigwedge P \in \Sigma_F \setminus w_i \bullet \neg P)$ is satisfiable. The formula $xlat(F, w, t_0, t_n)$ is satisfiable by construction: The satisfying valuation assigns the values given by \mathcal{V} to the variables occurring free in F , b to t_0 , e to t_n , and the points in time when \mathcal{I} changes on the

interval $[b, e]$ to the variables t_1, \dots, t_{n-1} . Hence, the algorithm accepts F .

If the algorithm accepts F , there is a untimed word w and a valuation for t_0, \dots, t_n and the variables occurring free in F such that $xlat(F, w, t_0, t_n)$ evaluates to true. For each w_i the formula $(\bigwedge P \in w_i \bullet P) \wedge (\bigwedge P \in \Sigma_F \setminus w_i \bullet \neg P)$ is satisfiable. Hence, there is a satisfying valuation M_i for the variables occurring in the state expressions P . Now, choose

$$\mathcal{I}(t) = \begin{cases} M_1 & t < t_1, \\ M_2 & t \in [t_1, t_2], \\ \vdots & \\ M_{n-1} & t \in [t_{n-2}, t_{n-1}], \\ M_n & t > t_{n-1}. \end{cases}$$

Choose the valuation \mathcal{V} of the free variables according to the satisfying valuation of $xlat(F, w, t_0, t_n)$. Then $\mathcal{I}, \mathcal{V}, [t_0, t_n] \models F$. \square

3. CSP-OZ-DC

Contents

3.1. CSPz	29
3.1.1. Syntax	30
3.1.2. Semantics	37
3.2. CSP-OZ-DC Classes	48
3.2.1. Syntax	48
3.2.2. Case Study	51
3.2.3. Semantics	54
3.3. Parallel Composition of Systems	59
3.4. Discussion and Related Work	61
3.4.1. CSPz	61
3.4.2. Semantics of CSP-OZ-DC	64
3.4.3. Parallel Composition	65
3.4.4. Related Work	65

In this chapter, we present the combined language CSP-OZ-DC. In the first section, we present CSPz the CSP syntax we use. Then, we present CSP-OZ-DC itself.

3.1. CSPz

The textbook language of CSP [Hoa85, Ros97] uses a minimal syntax and semantics such as the one described in section 2.1. There is no formal syntax for declaring data types and for common functions, e. g., arithmetic operators over integers. Instead, a mathematical notation with intuitive semantics is used.

When tool support for CSP arose, this gap had to be closed. The FDR model checker [For05] for CSP comes with its own input language that provides a complete syntax for expressions, types, and channels. It supports enumeration types, integers, tuples, sets, and functions. Scattergood [Sca94] defined a semantics for the input language and named it CSP_M for ‘machine readable CSP’.

Although this syntax would be expressive enough for a combination of CSP and Object-Z, it is inconvenient to have expressions in CSP processes written in one syntax and expressions in Z parts in a different syntax. Also, using two different type systems makes it more difficult to combine the languages. Therefore, Fischer [Fis00] defined a new concrete syntax for CSP, named CSPz, that provides the same constructs as CSP_M but reuses Z syntax where applicable. We follow his approach with small modifications.

3.1.1. Syntax

The language CSPz is a proper extension of the Z language. A Z specification consists of several paragraphs. In CSPz two new types of paragraphs exists: *channel declarations* and *process declarations*:

$$\textit{Paragraph} ::= \textit{Paragraph}_Z \mid \textit{Channel} \mid \textit{Process}$$

Furthermore, the syntax for Z expressions is extended in CSPz by *process expressions*.

$$\textit{Expression} ::= \textit{Expression}_Z \mid \textit{Expression}_{CSP}$$

In the remainder of this section, we introduce the new constructs *Channel*, *Process* and *Expression_{CSP}*.

Channel Declaration

A CSPz *channel* is declared with the keywords `chan` or `local_chan`. Local channels are only visible within a single CSP-OZ-DC class, while channels declared with `chan` are visible globally. Multiple channel declarations are separated by newline characters (*NL*). The syntax for a channel declaration is

$$\begin{aligned} \textit{Interface} &::= \textit{ChannelDecl} \textit{NL} \dots \textit{NL} \textit{ChannelDecl} \\ \textit{ChannelDecl} &::= \text{chan } \textit{NAME}, \dots, \textit{NAME} [: \textit{Expression}] \\ &\quad \mid \text{local_chan } \textit{NAME}, \dots, \textit{NAME} [: \textit{Expression}] \end{aligned}$$

The latter declares one or more channels of the same type. The type is given by *Expression*. This expression must be a schema type, i. e., a set of bindings that can be communicated over this channels. A binding in Z is a function mapping names to values. Thus every value that is communicated

over a channel has a name that can be used to reference the value. If *Expression* is omitted, it defaults to the empty schema $[]$. These channels are also called *signals*.

A channel corresponds to a set of *events* that can be communicated over this channel. An event is identified by its channel name and a binding that provides values for all parameters that appear in the channel declaration. A channel can be represented as a function that maps a binding to an event.

Example 3.1. The channel declaration

```
chan open, close
chan request : [x : ℤ]
```

declares three new channels *open*, *close*, and *request*. The channels *open* and *close* are signals. The channel *request* has a single parameter x of type \mathbb{Z} . There are infinitely many events corresponding to the channel *request*, e. g., $request\langle x == -1 \rangle$, $request\langle x == 0 \rangle$, $request\langle x == 1 \rangle$. For each signal channel there is exactly one corresponding event denoted by $open\langle \rangle$, $close\langle \rangle$.

Process Declaration

Traditionally, process declarations in CSP are recursive equations, each binding a name to a process term. The names occurring on the left hand side of the equation can appear again on the right hand side. An example of a simple recursive process is:

$$\begin{aligned} P &= a \rightarrow Q \\ Q &= b \rightarrow P \end{aligned}$$

In Z the operator $==$ is used to define constants, e. g.,

$$\mathbb{N} == \{n : \mathbb{Z} \mid n \geq 0\}.$$

However, this operator cannot be used for recursive equations. We therefore follow Fischer and use a new operator $\stackrel{c}{=}$ that allows defining a recursive CSP process. It comes with its own type-checking rule and has its own semantics tailored for recursive process equations.

The BNF syntax of a *process declaration* is

$$\begin{aligned} \text{ProcessDeclaration} &::= \text{ProcessEquation } NL \dots NL \text{ ProcessEquation} \\ \text{ProcessEquation} &::= \text{NAME}[(\text{SchemaText})] \stackrel{c}{=} \text{Expression} \end{aligned}$$

A process declaration consists of one or more equations separated by new-line characters. Each equation declares a new process *NAME* optionally taking parameters described by *SchemaText*. It is defined by an expression, which must evaluate to some process term. The latter is ensured by the type-checking rules of Z.

Process Expressions

The syntax of Z expressions is extended by process expressions. The design of the Z language makes it possible to define new operators without changing the syntax of Z expressions. For example, the external choice operator \square can be declared as a Z operator:

function 30 leftassoc ($_ \square _$)

$$\left| \begin{array}{l} _ \square _ = \text{Process} \times \text{Process} \rightarrow \text{Process} \\ \hline \forall P, Q : \text{Process} \bullet P \square Q = \dots \end{array} \right.$$

The first line defines \square as left-associative infix function with the precedence of 30 (the precedence of the $+$ operator). The axiomatic paragraph declares the operator as a function that takes two processes and returns a new process. The definition of $P \square Q$ is omitted here. This paragraph could be included in a CSP toolkit that takes a similar role as the mathematical toolkit for Z. However, some syntactic constructs cannot be introduced without changing the Z syntax.

Therefore, we follow [Fis00] and define process expressions as an extension to the core syntax of Z.

Basic Processes The simplest processes are *Stop*, *Skip*, *Chaos(A)*, and *Diver*.

Expression ::= *Stop* | *Skip* | *Chaos(Expression)* | *Diver* – basic processes

The process *Stop* represents a deadlock, the process that cannot evolve further. The process *Skip* terminates immediately. For the process *Chaos(A)*,

the value represented by A must be a set of events. The process can either refuse any communication or communicate any event from the set A . The process *Diver* is the diverging process executing an infinite sequence of τ events.

External and Internal Choice Beside the usual binary operators \square and \sqcap for external and internal choice, the syntax of CSPz also supports the *replicated* choice operators. With the replicated operators, a choice over an infinite number of processes is possible. The syntax for external and internal choice is as follows:

$$\begin{aligned}
 \textit{Expression} ::= & \textit{Expression} \square \textit{Expression} && \text{-- external choice} \\
 & | \textit{Expression} \sqcap \textit{Expression} && \text{-- internal choice} \\
 & | \square \textit{SchemaText} \bullet \textit{Expression} && \text{-- replicated external choice} \\
 & | \sqcap \textit{SchemaText} \bullet \textit{Expression} && \text{-- replicated internal choice}
 \end{aligned}$$

The syntax for the replicated choice operators is similar to the Z syntax of the λ -operator or set comprehension. The *SchemaText* declares new variables, which can be referenced in the process expression.

Example 3.2. The following construct builds an external choice over all processes $P(x, y)$ where x and y are integer variables and $x < y$:

$$\square x, y : \mathbb{Z} \mid x < y \bullet P(x, y)$$

Prefixing and Multi-Prefixing In the standard literature for CSP [Hoa78, Hoa85, Ros97] the basic syntax only defines one prefix operator $a \rightarrow P$. Here a is an event and P is a process. The resulting process $a \rightarrow P$ communicates the event a and afterwards proceed as P . The formal syntax is as follows.

$$\textit{Expression} ::= \textit{Expression} \rightarrow \textit{Expression} \quad \text{-- event or schema prefixing}$$

The first expression must be an event and the second expression a process expression. Fischer also defines schema prefixing in case the first expression is a schema. He uses it to give semantics for Z operations in CSPz. The details of this operator can be found in [Fis00].

However, to capture values that are communicated over a channel, there is the need for another syntax allowing to receive the communicated values into local variables. In [Ros97] this is defined as syntactic sugar. If c is a channel that can communicate the values $\{1, 2, 3\}$ then the process $c?a \rightarrow P(a)$ is an abbreviation for $c.1 \rightarrow P(1) \square c.2 \rightarrow P(2) \square c.3 \rightarrow P(3)$ (the dot is used in CSP to combine the channel name and the value to an event). With the notation of replicated choice this can be expressed as $\square a : \{1, 2, 3\} \bullet c.a \rightarrow P(a)$.

Following this tradition, the languages CSP_M and CSPz as defined in [Fis00] uses a special syntax known as multi-prefixing. For a channel c that communicates two integer values $c : \mathbb{Z} \times \mathbb{Z}$, Fischer allows writing

$$c!5?y : 1..3 \rightarrow P(y).$$

This denotes the process that writes 5 to the channel c , simultaneously receives a value from this channel in the range 1..3, and afterwards behaves like $P(y)$ where the received value is substituted for y . This process is equivalent to

$$c.5.1 \rightarrow P(1) \square c.5.2 \rightarrow P(2) \square c.5.3 \rightarrow P(3),$$

which can also be written as $\square y : 1..3 \bullet c.5.y \rightarrow P(y)$.

As mentioned earlier, our variant of CSPz does not allow channels to have non-structured types. Instead the channel c would be declared in CSPz as

$$\text{chan } c : [x : \mathbb{Z}; y : \mathbb{Z}]$$

and the names of the declared parameters x and y are part of the channel type. A syntax is needed that allows binding the output value, 5 in the example above, to the first channel parameter x . We use the following syntax, which follows the style of the Z standard:

$$(c!x == 5?y : 1..3) \rightarrow P(y)$$

This is an abbreviation for $\square x == 5 \bullet \square y : 1..3 \bullet c(\theta[x : \mathbb{Z}; y : \mathbb{Z}]) \rightarrow P(y)$. The θ operator creates the binding that maps the names x and y to the values of the variables with the same names. This way it relates the variables x and y that are bound by the operators \square and \square with the names x , y declared in the channel type. In general the syntax for multi-prefixing is as follows:

$$\text{Expression} ::= \text{NAME} [?\text{SchemaText} \dots !\text{SchemaText}] \rightarrow \text{Expression}$$

– multi event prefixing

The process $NAME!SchemaText_1?SchemaText_2 \cdots \rightarrow PROCESS$ is an abbreviation for

$$\begin{aligned} & \square SchemaText_1 \bullet \square SchemaText_2 \bullet \dots \\ & NAME(\theta \text{ typeof}(NAME)) \rightarrow PROCESS \end{aligned}$$

If a channel parameter does not occur in one of the *SchemaText* parts of a multi-prefix, it is implicitly taken from the current scope. Therefore it must be declared as process variables, class constants, class parameters, or global constants. If no variable with that name was declared this is a type error.

Parallel Composition

An important concept of CSP is parallel composition. There are several forms that differ in the way the processes synchronise. The CSPz syntax allows the most common constructs for parallel composition in binary and replicated form:

$$\begin{aligned} Expression ::= & Expr \parallel_{Expr} Expr && \text{-- generalised parallel} \\ & | Expr \parallel_{Expr} Expr && \text{-- alphabetised parallel} \\ & | Expr \parallel\!\!\! \parallel Expr && \text{-- interleaving} \\ & | Expr [Expr] Expr && \text{-- linked parallel} \\ & | \parallel [Expr] SchemaText \bullet Expr && \text{-- generalised parallel} \\ & | \parallel\!\!\! \parallel SchemaText \bullet [Expr] Expr && \text{-- alphabetised parallel} \\ & | \parallel\!\!\! \parallel\!\!\! \parallel SchemaText \bullet Expr && \text{-- interleaving} \\ & | [Expr] SchemaText \bullet Expr && \text{-- linked parallel} \end{aligned}$$

The generalised parallel operator $P \parallel_A Q$ denotes that processes P and Q run in parallel and synchronise on the events in A . Expressions P and Q must denote processes and A a set of events. The resulting process can engage in any communication that is not in A if either P or Q can engage in

that communication. The process can also engage in a communication in A provided both P and Q can engage in that communication synchronously. The alphabetised parallel operator $P_A \parallel_B Q$ is similar to $P \parallel_{A \cap B} Q$. Thus, P and Q synchronise on the intersection of the sets A and B . Additionally, P may only engage in communications in A and Q in communications in B . The interleaving operator $P \parallel\!\!\! \parallel Q$ is an abbreviation for $P \parallel_{\emptyset} Q$. Thus, P and Q do not synchronise at all.

The linked parallel operator $P [R] Q$ is a combination of renaming, generalised parallel composition and hiding. Here, R is a relation between events, i. e., $R : EVENTS \leftrightarrow EVENTS$. The process P can asynchronously engage in communication events that are not in $\text{dom } R$. Likewise, the process Q can engage in events that are not in $\text{ran } R$. Furthermore, if $(a, b) \in R$, process P can engage in a , and process Q can engage in b , they engage in these communication synchronously. The communication is hidden outside the parallel composition. Thus, linked parallel can be expressed as follows. First for each element $(a, b) \in R$ the event a of process P and event b in process Q are renamed to a fresh name. Then the renamed processes resulting from P and Q are synchronised on these fresh names by a generalised parallel operator. Finally, these fresh names are hidden from the resulting process.

Other Constructs

Furthermore, the following constructs are allowed in CSPz:

$Expression ::= Expression \ ; Expression$	– sequential composition
$\ ; SchemaText \bullet Expression$	– sequential composition
$Expression \ \Delta \ Expression$	– interrupt
$Expression \ \setminus \ Expression$	– hiding
$Expression [Renaming]$	– renaming
$Predicate \ \& \ Expression$	– guarding

The sequential composition $P \ ; Q$ behaves like the process P until P engages in a termination event \checkmark . This event is hidden, i. e., replaced by

a τ transition. Afterwards the process behaves like Q . There is also a replicated form: The process $\wp a : \langle 1, 2, 3 \rangle \bullet P(a)$ is equivalent to $P(1) \wp P(2) \wp P(3)$. In this case, the declaration following the symbol \wp must contain only a single variable of sequence type. The interrupt operator $P \triangle Q$ behaves like P until the first visible event of Q occurs. Afterwards it proceeds as Q . The hiding operator $P \setminus A$ hides a set of events A from a process P . Whenever P engages in an event $a \in A$ the process $P \setminus A$ engages in a τ event. Otherwise, $P \setminus A$ behaves in the same way as P . The renaming operator $P[R]$ renames all events according to the function R . The renaming function R must be a partial function $EVENTS \setminus \{\tau, \checkmark\} \mapsto EVENTS \setminus \{\checkmark\}$. Whenever P engages in an event $a \in \text{dom } R$, the process $P[R]$ engages in $R(a)$. Renaming an event to τ is the same as hiding the event. The process $g \& P$ is equivalent to the process

$$\text{if } g \text{ then } P \text{ else } Stop.$$

If the guard predicate g holds in the current context, the process behaves just like P . Otherwise, it behaves like $Stop$, i. e., it leads to a deadlock.

3.1.2. Semantics

In this chapter we give semantics for well-typed CSPz declarations. As a first step, we need to define what a well-typed declaration is. The Z standard uses type inference rules to define well-typed predicates and expressions and to compute the type of an expression. We extend these rules for process expressions and CSP declarations.

The next step is to define a Z semantics for process expressions and for process declarations. In the Z standard [ISO02] $\llbracket e \rrbracket^{\mathcal{E}}$ denotes the semantics of an expression e and $\llbracket D \rrbracket^{\mathcal{D}}$ the semantics of a Z paragraph D . For process expressions P , we define the Z semantics $\llbracket P \rrbracket^{\mathcal{E}}$ to represent the structure of the process. It does not describe the *behaviour* of a CSP process, though.

Finally, the behaviour is given as an operational semantics. It is given as a labelled transition system computed from the Z semantics of the process.

Extension to the Z Type Checker

The Z standard [ISO02] defines type inference rules to determine the type of expressions and to check if it is well-typed. In this section we extend these rules for process expressions. We will give new type inference rules for process expressions.

The basic notation for correctly typed expressions is $\Sigma \vdash^{\mathcal{E}} e \text{ ; } \tau$. This denotes that, in the context of Σ , expression e is of type τ . The operator $\vdash^{\mathcal{E}}$ is used for well-typed expressions; similar operators $\vdash^{\mathcal{P}}$ and $\vdash^{\mathcal{D}}$ are used for well typed predicates and paragraphs. The signature Σ comprises the table of symbols declared outside the expression e . In the meta-language it is a mapping from names to types. Amongst others, the Z language defines the following types (c.f. section 2.2.2):

- Given types are denoted by **GIVEN NAME**, where *NAME* is an arbitrary name.
- Power set types are denoted by $\mathbb{P}\tau$. This is the type comprising all sets containing elements of type τ .
- Cartesian product types are denoted by $\tau_1 \times \cdots \times \tau_n$.
- Schema types are denoted by $[\beta]$, where β is a signature, i.e., a mapping from names to types.

Process expressions are always of the type *PROCESS*. In the Z meta-language we represent it as **GIVEN PROCESS**. The definition of the type inference rules for CSP expressions is straightforward. For example, the rule for external choice is given here:

$$\frac{\begin{array}{l} \Sigma \vdash^{\mathcal{E}} e_1 \text{ ; } \tau_P \\ \Sigma \vdash^{\mathcal{E}} e_2 \text{ ; } \tau_P \end{array}}{\Sigma \vdash^{\mathcal{E}} e_1 \square e_2 \text{ ; } \tau_P} \left(\tau_P = \mathbf{GIVEN PROCESS} \right)$$

The part below the horizontal line is the conclusion of the rule. It states that $e_1 \square e_2$ is a well-typed expression of type τ_P in the context of Σ . The type τ_P is fixed to **GIVEN PROCESS** by the condition to the right of the rule. This rule can only be applied if the premises above the horizontal line hold. In this case, e_1 and e_2 must also be of type τ_P in the same context Σ .

The syntactic constructs for repeated application of operators may declare new variables:

$$\frac{\begin{array}{l} \Sigma \vdash^{\mathcal{E}} t \text{ ; } \tau \\ \Sigma \oplus \beta \vdash^{\mathcal{E}} e \text{ ; } \tau_P \end{array}}{\Sigma \vdash^{\mathcal{E}} \square t \bullet e \text{ ; } \tau_P} \left(\begin{array}{l} \tau = \mathbb{P}[\beta] \\ \tau_P = \mathbf{GIVEN PROCESS} \end{array} \right)$$

The premise of the rule demands that schema text t is well-typed with type τ . This type must be a schema type $\mathbb{P}[\beta]$. Here β is a signature that maps the variable names declared in t to their types. The signature β is added to the global signature Σ . The operator \oplus adds the mappings from β to the signature Σ returning a new signature. If any symbols are declared in both Σ and β the types declared in β are used. The combined signature is used to check if the process expression e is well-typed. Thus, e may use the symbols that are declared in t as well as symbols coming from the current context. Only if the type of e is **GIVEN PROCESS**, the expression $\square t \bullet e$ is well-typed and its type is **GIVEN PROCESS**, too.

The type-checking rule for a CSP paragraph is more complicated. The problem here is that CSP processes are recursive, which is not possible in \mathbb{Z} . When type-checking a \mathbb{Z} declaration like $v == e$, the type-checker first determines the type of e and then adds v to the symbol table. If expression e contains the symbol v this results in an error because the symbol v is not yet defined. In CSPz we need to write recursive declarations, though. Therefore, a special type-checking rule is necessary that adds the left-hand side to the symbol table before determining whether the right-hand side is a well-typed expression. This can be done since the type of the left-hand side is known before parsing the right-hand side: it is always **GIVEN PROCESS**.

Figure 3.1 depicts a rule can be used to check that a CSP declaration is well-typed: The conclusion of this rule states that the CSP paragraph

$$\begin{array}{l}
 p_1 \stackrel{c}{=} e_1 \\
 \vdots \\
 p_n \stackrel{c}{=} e_n \\
 q_1(t_1) \stackrel{c}{=} e'_1 \\
 \vdots \\
 q_m(t_m) \stackrel{c}{=} e'_m
 \end{array}$$

is well-typed in the context of the signature Σ and declares new symbols defined by σ . In this case, $\vdash^{\mathcal{D}}$ is used instead of $\vdash^{\mathcal{E}}$ because this is a declaration, not an expression. For the same reason σ is not a single type but a signature. This signature defines the types of the newly declared symbols, in this case $p_1, \dots, p_n, q_1, \dots, q_m$. These symbols are CSP process identifiers, t_1, \dots, t_m are the schema text expressions declaring the parameters of q_1, \dots, q_m , and $e_1, \dots, e_n, e'_1, \dots, e'_m$ are the process expressions that define the processes. The symbols p_1, \dots, p_n should be of type

$$\begin{array}{c}
\Sigma \vdash^{\mathcal{E}} t_1 \circ \tau'_1 \quad \dots \quad \Sigma \vdash^{\mathcal{E}} t_m \circ \tau'_m \\
\Sigma \oplus \beta_1 \vdash^{\mathcal{E}} \text{chartuple } t_1 \circ \tau_1 \\
\vdots \\
\Sigma \oplus \beta_m \vdash^{\mathcal{E}} \text{chartuple } t_m \circ \tau_m \\
\Sigma \oplus \sigma \vdash^{\mathcal{E}} e_1 \circ \tau_P \\
\vdots \\
\Sigma \oplus \sigma \vdash^{\mathcal{E}} e_n \circ \tau_P \\
\Sigma \oplus \sigma \oplus \beta_1 \vdash^{\mathcal{E}} e'_1 \circ \tau_P \\
\vdots \\
\Sigma \oplus \sigma \oplus \beta_m \vdash^{\mathcal{E}} e'_m \circ \tau_P \\
\hline
\Sigma \vdash^{\mathcal{D}} \begin{array}{c} p_1 \stackrel{c}{=} e_1 \\ \vdots \\ p_n \stackrel{c}{=} e_n \\ q_1(t_1) \stackrel{c}{=} e'_1 \quad \circ \sigma \\ \vdots \\ q_m(t_m) \stackrel{c}{=} e'_m \end{array} \quad \left(\begin{array}{l} \tau_P = \text{GIVEN PROCESS} \\ \tau'_1 = \mathbb{P}[\beta_1] \quad \dots \quad \tau'_m = \mathbb{P}[\beta_m] \\ \sigma = p_1 : \tau_P; \\ \vdots \\ p_n : \tau_P; \\ q_1 : \mathbb{P}(\tau_1 \times \tau_P); \\ \vdots \\ q_m : \mathbb{P}(\tau_m \times \tau_P) \end{array} \right)
\end{array}$$

Figure 3.1.: Type-checking rule for a CSPz process declaration

$\tau_P = \text{GIVEN PROCESS}$ and q_1, \dots, q_m should be functions mapping their parameters to **GIVEN PROCESS**.

The first step in computing the type τ_i of the parameters is to determine the type of the schema text t_i . Therefore, the rule contains the premise $\Sigma \vdash^{\mathcal{E}} t_i \circ \tau'_i$ for each t_i . The type τ'_i must be a schema type (set of bindings) with signature β_i . This signature is a function mapping the symbols declared in t_i to their types. With this signature the type of the characteristic tuple of t_i , denoted by chartuple t_i , can be determined to compute the parameter type τ_i . While τ'_i is a signature type where the order of the declared parameters is not relevant, τ_i is a Cartesian product of the types declared in t_i in the appropriate order. If t_i declares only one parameter then τ_i is the type of that parameter.

Furthermore, the rule requires that expressions $e_1 \dots e_n, e'_1, \dots, e'_m$ are well-typed expressions of type **GIVEN PROCESS**. They may use symbols

declared in the signature Σ that represents the current context. They may also use the newly defined process identifiers p_i , q_i from the signature σ recursively. Furthermore, the expression e'_i can use the symbols from β_i , which are the parameters declared in the schema text t_i .

Embedding CSP processes in Z meta-language

Since we allow mixing CSP processes with Z expressions and thus allow functions returning processes (parametrised processes) or external choice over a set of CSP processes, we need to define a Z semantics for CSP. Fischer [Fis00] defines the Z semantics of a CSP process as the operational semantics. However, for recursive process equations defining an operational semantics directly is cumbersome. Therefore, we use a different approach.

It is not necessary that the Z semantics defines the behaviour of the CSP process; it suffices if it defines the structure. The abstract semantics can be represented by a free type. Under this semantics, two CSP process expressions are equal if they are syntactically equivalent. A free type covering the concepts of CSPz as defined in the previous section is the following.

$$\begin{aligned}
 \text{PROCESS} ::= & \text{Stop} \mid \text{Skip} \mid \Omega \mid \text{Chaos} \langle\langle \mathbb{P} \text{EVENTS} \rangle\rangle \mid \text{Diver} \\
 & \mid \text{prefix} \langle\langle \text{EVENTS}, \text{PROCESS} \rangle\rangle \\
 & \mid \text{extchoice} \langle\langle \text{World} \rightarrow \text{PROCESS} \rangle\rangle \\
 & \mid \text{intchoice} \langle\langle \text{World} \rightarrow \text{PROCESS} \rangle\rangle \\
 & \mid \text{sequence} \langle\langle \text{seq } \text{PROCESS} \rangle\rangle \\
 & \mid \text{genparallel} \langle\langle \mathbb{P} \text{EVENTS}, \text{World} \rightarrow \text{PROCESS} \rangle\rangle \\
 & \mid \text{alphparallel} \langle\langle \text{World} \rightarrow \text{PROCESS} \times \mathbb{P} \text{EVENTS} \rangle\rangle \\
 & \mid \text{linkedparallel} \langle\langle \text{PROCESS}, (\text{EVENTS} \leftrightarrow \text{EVENTS}), \\
 & \hspace{15em} \text{PROCESS} \rangle\rangle \\
 & \mid \text{interrupt} \langle\langle \text{PROCESS}, \text{PROCESS} \rangle\rangle \\
 & \mid \text{hiding} \langle\langle \text{PROCESS}, \mathbb{P} \text{EVENTS} \rangle\rangle \\
 & \mid \text{renaming} \langle\langle \text{PROCESS}, (\text{EVENTS} \leftrightarrow \text{EVENTS}) \rangle\rangle \\
 & \mid \text{call} \langle\langle \text{NAME} \rangle\rangle \\
 & \mid \text{call} \langle\langle \text{NAME} \times \text{World} \rangle\rangle
 \end{aligned}$$

By $\mathbb{W}_P := \text{carrier GIVEN PROCESS}$ we denote the carrier set of the above type. Likewise we denote by $\mathbb{W}_E := \text{carrier GIVEN EVENTS}$ the set of all events. To represent replicated operators like $\square t \bullet P$ we use a function mapping from the type induced by t to PROCESS . As the type of t can be arbitrary we use World to represent the set of all elements. Such a set cannot be defined in Z because it would not be well-typed.

Because the meta-language of Z is untyped, this set can be defined in the meta-language, though. We assume that *Stop*, *Skip* are constants of type *PROCESS* and *prefix*, *extchoice*, etc. are functions returning objects of type *PROCESS*, just as if *PROCESS* had been declared as free type in some parent Z section. The functions are injective and pairwise disjoint.

A Z model M is a binding: a partial function mapping variable names to values. It provides values for all given sets and all constants and functions declared in axiomatic and horizontal definitions in the preceding specification. It also provides values for variables bound by λ operators or logical quantifier. The semantics of an expression $\llbracket e \rrbracket^{\mathcal{E}}$ is a function from models to values.

Using the data type *PROCESS* and the constants and functions defined above, it is straightforward to give the abstract semantics for all CSP operators. For example, the semantics for external choice is defined as follows.

$$\begin{aligned} \llbracket \square t \bullet P \rrbracket^{\mathcal{E}} M &= \text{extchoice}(\llbracket \lambda t \bullet P \rrbracket^{\mathcal{E}} M) \\ \llbracket P \square Q \rrbracket^{\mathcal{E}} M &= \text{extchoice}(\{1 \mapsto \llbracket P \rrbracket^{\mathcal{E}} M, 2 \mapsto \llbracket Q \rrbracket^{\mathcal{E}} M\}) \end{aligned}$$

The first line gives the semantics of the expression $\square t \bullet P$. It reuses the Z semantics for λ expressions to define the semantics of the replicated operator. The operator λ builds a function from the set induced from the schema text t to the set of processes. Then the *extchoice* function from the definition of *PROCESS* is applied to define the semantics. The second line gives the semantics of the expression $P \square Q$. It uses $\llbracket \cdot \rrbracket^{\mathcal{E}}$ and M recursively to compute the semantics of P and Q . Since *extchoice* takes a function as parameter, a function that maps 1 to the semantics of P and 2 to the semantics of Q is created.

The definition of *PROCESS* above provides no construct for guarding. The reason is that guarding is replaced by an if construct as follows.

$$\llbracket g \& P \rrbracket^{\mathcal{E}} M = \llbracket \text{if } g \text{ then } P \text{ else } \text{Stop} \rrbracket^{\mathcal{E}} M$$

The semantics for a *declaration* is a relation from models to models. This relation contains a tuple $M \mapsto M'$ if M' contains M and additionally the valuation for the newly declared symbols. The model M is a consistent model for all the preceding declarations and the model M' is a consistent extension of M including the new declaration. Note that there may be several consistent extension M' . If the declaration is inconsistent there is

no M' . Thus, the semantics of a declaration is a relation, not a function. The semantics for a recursive process equation is defined as follows.

$$\left[\begin{array}{l} p_1 \stackrel{c}{=} e_1 \\ \vdots \\ p_n \stackrel{c}{=} e_n \\ q_1(t_1) \stackrel{c}{=} e'_1 \\ \vdots \\ q_m(t_m) \stackrel{c}{=} e'_m \end{array} \right]^{\mathcal{D}} \{M, M', M'' : \mathbf{Model} \mid$$

$$= \begin{array}{l} M' = M \cup \{p_1 \mapsto \text{call}(p_1), \dots, p_n \mapsto \text{call}(p_n), \\ q_1 \mapsto \{t : \mathbb{W} \bullet t \mapsto \text{call}(q_1, t)\}, \dots, \\ q_m \mapsto \{t : \mathbb{W} \bullet t \mapsto \text{call}(q_m, t)\}\} \wedge \\ M'' = M \cup \{p_1 \mapsto \llbracket e_1 \rrbracket^{\mathcal{E}} M', \dots, p_n \mapsto \llbracket e_n \rrbracket^{\mathcal{E}} M', \\ q_1 \mapsto \llbracket \lambda t_1 \bullet e'_1 \rrbracket^{\mathcal{E}} M', \dots, \\ q_m \mapsto \llbracket \lambda t_m \bullet e'_m \rrbracket^{\mathcal{E}} M'\} \\ \bullet M \mapsto M'' \} \end{array}$$

The semantics is computed in two steps. In the first step the model M' is created. In this model the valuations of the symbols p_1, \dots, p_n are just *call* processes. The valuation for each of the symbols q_1, \dots, q_n is a function that assigns to the input parameters the corresponding parametrised *call* process. The intermediate model M' is used to assign a meaning to the right-hand sides of the process equations e_1, \dots, e_n and e'_1, \dots, e'_m . Thus, the semantics of the symbol p_1 occurring on the right-hand side of the equation is the process $\text{call}(p_1)$. The final model M'' assigns to each symbol p_1, \dots, q_m the semantics of the right-hand side. This model is used to define the semantics of the declaration.

Example 3.3. Let $a, b : [x : \mathbb{Z}]$ be channels with parameter $x : \mathbb{Z}$. Consider the following recursive equation:

$$P \stackrel{c}{=} a?x : \mathbb{Z} \rightarrow Q(x)$$

$$Q(y : \mathbb{Z}) \stackrel{c}{=} b!x == y + 1 \rightarrow P$$

When computing the semantics for this declaration, we start with some model M that already assigns to a and b an injective function $[x : \mathbb{Z}] \rightarrow \mathbf{EVENTS}$. In the first step M' is build. The model M' assigns to P the concrete process $\text{call}(P)$ and to Q a function $\mathbb{Z} \rightarrow \mathbf{PROCESS}$ with $M'(Q)(t) = \text{call}(Q, t)$. Using this model M' , the right-hand sides of the process equations are evaluated. The resulting values $\llbracket a?x : \mathbb{Z} \rightarrow Q(x) \rrbracket^{\mathcal{E}} M'$ and

$$\llbracket \lambda y : \mathbb{Z} \bullet b!x == y + 1 \rightarrow P \rrbracket^{\mathcal{E}} M'$$

are assigned to the symbols P and Q in M'' . When evaluating multi-prefixes the corresponding operators \square and \sqcap are created. Thus the final model M'' looks as follows.

$$\begin{aligned} M''(P) &= \text{extchoice}(\{t : \mathbb{Z} \bullet t \mapsto \text{prefix}(a \downarrow x == t \downarrow, \text{call}(Q, t))\}) \\ M''(Q) &= \{y : \mathbb{Z} \bullet \\ &\quad y \mapsto \text{intchoice}(\{y + 1 \mapsto \text{prefix}(b \downarrow x == y + 1 \downarrow, \text{call}(P))\})\} \end{aligned}$$

Operational Semantics

The operational semantics of CSPz is based on labelled transition systems.

The configurations are the semantic values from \mathbb{W}_P , the world of processes in CSPz. In \mathbb{Z} , the expression carrier τ denotes the set of all elements of type τ . The transitions are labelled by elements from \mathbb{W}_E , the semantic values of events. A labelled transition system is represented as a tuple $(\mathbb{W}_P, \mathbb{W}_E, p_0, \longrightarrow)$, where $p_0 \in \mathbb{W}_P$ is the initial configuration and $\longrightarrow \subseteq \mathbb{W}_P \times \mathbb{W}_E \times \mathbb{W}_P$ is the transition relation. The element $(p, \alpha, p') \in \longrightarrow$ means that there is a transition from process p to process p' under the event α . It is also denoted by $p \xrightarrow{\alpha} p'$.

The transition relation \longrightarrow is almost the same for all models; only the transitions for the processes $\text{call}(P)$ and $\text{call}(P, t)$ depend on the context. The context is given as a \mathbb{Z} model M and is computed using $\llbracket \cdot \rrbracket^D$ according to the semantics defined in the previous section. The transition relation is generated inductively by the following inference rules. We use the symbols P, Q to denote processes, α to denote arbitrary events, a for events except \checkmark and τ . When defining the operational semantics for replicated operators, we use the symbol \mathcal{P} to denote a function that maps an arbitrary type into the set of processes.

Basic Processes For the process *Stop* no outgoing transition is defined. The process *Skip* terminates immediately emanating a \checkmark event. The terminated process is conventionally represented as Ω and is equivalent to the *Stop* process, i. e., no outgoing transition is defined for this process.

$$\overline{\text{Skip} \xrightarrow{\checkmark} \Omega}$$

The process *Chaos* can either communicate some event or make an internal transition to the *Stop* process. In the second case the system deadlocks.

$$\frac{}{Chaos(A) \xrightarrow{a} Chaos(A)} [a \in A] \qquad \frac{}{Chaos(A) \xrightarrow{\tau} Stop}$$

The process *Diver* diverges by emanating τ events.

$$\frac{}{Diver \xrightarrow{\tau} Diver}$$

Prefixing As explained earlier, multi-prefixing is always translated to external and internal choice over event prefixing. Therefore, only a simple rule for event prefixing is needed.

$$\frac{}{prefix(a, P) \xrightarrow{a} P}$$

Internal and External Choice The binary internal choice is a special case of the parameterised internal choice. The latter has the following simple rule:

$$\frac{}{intchoice(\mathcal{P}) \xrightarrow{\tau} P} [P \in \text{ran } \mathcal{P}]$$

Here \mathcal{P} is a function mapping some arbitrary elements to processes. An element from the domain of \mathcal{P} contains the variables defined in the schema text t of the original CSPz expression $\square t \bullet P$. These variables are not relevant for the semantics, therefore the rule only uses $\text{ran } \mathcal{P}$. The process $intchoice(\mathcal{P})$ can enter any of these processes by an internal τ -transition.

In case of external choice $extchoice(\mathcal{P})$ there are two rules: A visible event a can be performed if one of the processes P in $\text{ran } \mathcal{P}$ can perform this event. In this case, the successor of the external choice is the successor of P .

$$\frac{P \xrightarrow{a} P'}{extchoice(\mathcal{P}) \xrightarrow{a} P'} [P \in \text{ran } \mathcal{P}]$$

The other possibility is that some processes in $\text{ran } \mathcal{P}$ can perform a τ -transition. We follow [Fis00] and demand that all processes that can perform τ -transitions take them simultaneously. This is necessary to prevent

infinite chains of τ -transitions as is explained in [Fis00]. We use the set T to denote the pre-image of all processes that can perform τ -transitions.

$$\frac{\forall t : T \bullet \mathcal{P}t \xrightarrow{\tau} \mathcal{P}'t}{extchoice(\mathcal{P}) \xrightarrow{\tau} extchoice(\mathcal{P}')} \left[\begin{array}{l} T = \{t : \text{dom } \mathcal{P} \mid \\ \quad \exists \mathcal{P}' \bullet \mathcal{P}t \xrightarrow{\tau} \mathcal{P}'\} \\ T \neq \emptyset \\ T \triangleleft \mathcal{P} = T \triangleleft \mathcal{P}' \end{array} \right]$$

Parallel Composition Three rules are needed for generalised parallel composition, one for asynchronous steps, one for synchronous steps, and one for termination. The following rule is for asynchronous steps:

$$\frac{P \xrightarrow{\alpha} P'}{genparallel(A, \mathcal{P}) \xrightarrow{\alpha} genparallel(A, \mathcal{P} \oplus \{t \mapsto P'\})} \left[\begin{array}{l} (t \mapsto P) \in \mathcal{P} \\ \alpha \notin A \cup \{\checkmark\} \end{array} \right]$$

Here P is one of the processes in the range of \mathcal{P} . The rule is applicable if P can evolve with the communication α that is not in the synchronisation alphabet and P' is its successor. Note that the internal operation $\alpha = \tau$ is allowed. The parallel composition can then evolve to $\mathcal{P} \oplus \{t \mapsto P'\}$, i. e., P is replaced by P' in \mathcal{P} . For events from the alphabet A all processes need to engage in a transition synchronously:

$$\frac{\forall t : \text{dom } \mathcal{P} \bullet \mathcal{P}t \xrightarrow{a} \mathcal{P}'t}{genparallel(A, \mathcal{P}) \xrightarrow{a} genparallel(A, \mathcal{P}')} [a \in A, \text{dom } \mathcal{P} = \text{dom } \mathcal{P}']$$

Termination needs a special rule that is similar to the asynchronous rule. The difference is that the \checkmark event is communicated only after the last process has terminated. The \checkmark events of the processes that terminated before are replaced by τ events.

$$\frac{P \xrightarrow{\checkmark} \Omega}{genparallel(A, \mathcal{P}) \xrightarrow{\tau} genparallel(A, \mathcal{P} \oplus \{t \mapsto \Omega\})} [(t \mapsto P) \in \mathcal{P}]$$

$$\frac{}{genparallel(A, \mathcal{P}) \xrightarrow{\checkmark} \Omega} [\text{ran } \mathcal{P} = \{\Omega\}]$$

With the alphabetised parallel operator every process has its own synchronisation alphabet. Instead of a function $\mathcal{P} : World \rightarrow PROCESS$ a function $\mathcal{PA} : World \rightarrow PROCESS \times \mathbb{P} EVENTS$ is used. A process synchronises with all processes that have the same event in their alphabet. The τ events are always asynchronous:

$$\frac{P \xrightarrow{\tau} P'}{\text{alphparallel}(\mathcal{PA}) \xrightarrow{\tau} \text{alphparallel}(\mathcal{PA} \oplus \{t \mapsto (P', A)\})} \quad [(t \mapsto (P, A)) \in \mathcal{PA}]$$

For visible events the following rule is used:

$$\frac{\forall t : \text{dom } \mathcal{PA}' \bullet (\mathcal{PA} t).1 \xrightarrow{a} (\mathcal{PA}' t).1}{\text{alphparallel}(\mathcal{PA}) \xrightarrow{a} \text{alphparallel}(\mathcal{PA} \oplus \mathcal{PA}')} \quad \left[\begin{array}{l} \text{dom } \mathcal{PA}' = \\ \{t \mid a \in \mathcal{PA} t.2\} \\ \mathcal{PA}' \neq \emptyset \\ \forall t : \text{dom } \mathcal{PA}' \bullet \\ \mathcal{PA} t.2 = \mathcal{PA}' t.2 \end{array} \right]$$

Note that $x.1$ and $x.2$ select the first and second element of a Cartesian tuple x , in this case the process and the synchronisation alphabet. The function $\mathcal{PA}' : World \rightarrow PROCESS \times \mathbb{P} EVENTS$ gives the successor state of exactly those processes that synchronise on the event a . The event a must be in the synchronisation alphabet of some process, so \mathcal{PA}' may not be empty. Again we need a special termination rule:

$$\frac{P \xrightarrow{\checkmark} \Omega}{\text{alphparallel}(\mathcal{PA}) \xrightarrow{\tau} \text{alphparallel}(\mathcal{PA} \oplus \{t \mapsto (\Omega, A)\})} \quad [(t \mapsto (P, A)) \in \mathcal{P}]$$

$$\frac{}{\text{alphparallel}(\mathcal{PA}) \xrightarrow{\checkmark} \Omega} \quad [\forall t : \text{dom } \mathcal{PA} \bullet \mathcal{PA} t.1 = \Omega]$$

Recursion If a process identifier P occurs recursively on the right-hand side, this is translated to a process $\text{call}(P, e)$ where e is the value of the parameter. If P takes more than one parameter, e is a tuple. To determine

the operational semantics of the process P the model M is needed. This is the only part where the transition relation \longrightarrow depends on the underlying Z model. The called process is entered with a τ -transition. If the process identifier P does not take any parameter, $\llbracket P \rrbracket^{\mathcal{E}} M$ is the associated process term. Otherwise, $\llbracket P \rrbracket^{\mathcal{E}} M$ is a function mapping values to process terms. The following two rules distinguish these cases:

$$\frac{}{call(P) \xrightarrow{\tau} \llbracket P \rrbracket^{\mathcal{E}} M}$$

$$\frac{}{call(P, e) \xrightarrow{\tau} \llbracket P \rrbracket^{\mathcal{E}} M(e)}$$

3.2. CSP-OZ-DC Classes

The language CSP-OZ-DC is an extension of Object-Z, which is an extension of Z . Thus, every Z specification is already a valid CSP-OZ-DC specification, but not the other way round. In Z the basic building blocks of a specification are paragraphs. A paragraph is either an axiomatic declaration, a schema definition, a free type declaration, a given type declaration, or a horizontal definition. In Object-Z [Smi00] a new kind of paragraph was introduced, namely Object-Z classes. Likewise, the main extension of the language CSP-OZ-DC is the CSP-OZ-DC class declaration. This section gives the syntax and semantics for CSP-OZ-DC classes.

3.2.1. Syntax

The syntax for a CSP-OZ-DC class is as follows.

$$COD_Class ::= \left[\begin{array}{l} NAME \ [[Formals]] \ [(SchemaText)] \\ Interface \\ ProcessDeclaration \\ [Z_Paragraph \dots Z_Paragraph] \\ State \\ [Init] \\ [Operation \dots Operation] \\ \hline DC \end{array} \right]$$

This declaration resembles Object-Z class declarations. The main syntactic difference is the interface part that declares the CSPz channels the class uses to communicate with its environment. In the remainder of the section the parts of CSP-OZ-DC classes are presented in details. The syntax is similar to CSP-OZ classes which are described in [Fis00]. Only the *DC* part is new in CSP-OZ-DC. We therefore only give a brief summary and provide the syntactic rules.

Interface Part

In the interface part the channels of a class are declared. A channel declaration introduces a new channel that allows communicating synchronous events. Each channel can have a type that declares the data values exchanged via this channel. In textbook CSP the channel type is a Cartesian tuple, however, in the Z world it makes more sense to use schema types for communications. The syntax of a channel declaration has already been explained in section 3.1. It is given here again for completeness.

$$\begin{aligned} \textit{Interface} & ::= \textit{ChannelDecl } NL \dots NL \textit{ ChannelDecl} \\ \textit{ChannelDecl} & ::= \textbf{chan } NAME, \dots, NAME [: \textit{Expression}] \\ & \quad | \textbf{local_chan } NAME, \dots, NAME [: \textit{Expression}] \end{aligned}$$

Process Declaration

The CSP process declaration is identical to the CSPz process declaration introduced in section 3.1:

$$\begin{aligned} \textit{ProcessDeclaration} & ::= \textit{ProcessEquation } NL \dots NL \textit{ ProcessEquation} \\ \textit{ProcessEquation} & ::= NAME [(\textit{SchemaText})] \stackrel{c}{=} \textit{Expression} \end{aligned}$$

In every CSP-OZ-DC class a process named **main** must be declared. This is the process that will be entered initially.

Z Paragraphs

A CSP-OZ-DC class may include arbitrary Z paragraphs declaring new symbols that are local in the class. These symbols have some values that do not change during the lifetime of the class. For example, axiomatic definitions may be used declaring constants, auxiliary functions, or internal parameters invisible to the environment.

State

The state schema is declared in the same way as in Object-Z. The state schema is an unnamed schema that declares the state variables of a CSP-OZ-DC class.

$$State ::= \left[\begin{array}{l} \overline{DeclPart} \\ \overline{Predicate} \end{array} \right]$$

The syntax for *DeclPart* is taken from the Z standard. It is a list of declarations. A declaration can be a simple variable declaration $v_1, v_2 : World$ declaring typed variables, a schema reference including the declaration and invariant of the referred schema, or a definition $v == e$ declaring v to be equal to expression e . The *Predicate* serves as a global invariant of the state schema. State variables may be changed by operation schemas, which are triggered by communication events.

Init Schema

The **Init** schema is a simple schema that gives the initial values of the state variables. Unlike all the other schemas in Z it only contains the predicate. The declaration part is always implicitly the state schema of the current class. If the **Init** schema is omitted the schema with predicate **true** is used by default, i.e., every value that satisfies the state invariant is allowed as initial value.

$$Init ::= \left[\begin{array}{l} \overline{Init} \\ \overline{Predicate} \end{array} \right]$$

Operations

Each communication event can be linked to an operation on the state variables (the variables declared in the state schema). To this end, an operation schema is used. The syntax is as follows:

$$Operation ::= \left[\begin{array}{l} \overline{com_NAME} \\ \overline{\Delta(NAME, \dots, NAME)} \\ \overline{DeclPart} \\ \overline{Predicate} \end{array} \right]$$

An operation is a schema with a special name `com_NAME`. Here *NAME* is the name of the channel that is associated with this operation. The first line of the schema gives the Δ -list, which lists the variables that are changed by the operation. The variables listed here must be declared in the state schema. The remaining part of the declaration can declare input and output variables. These must be those variables declared in the schema declaration of the channel. The variables decorated by ? are inputs of the operation. Those decorated by ! are outputs. The predicate can reference the variables in the state schema (pre-state), the same variables decorated with ' (post-state), and the input and output parameters declared in the *DeclPart*. For any state variable *v* in the state schema that is not in the Δ -list, the equation $v = v'$ is implicitly added to the predicate to denote that this variable does not change.

Duration Calculus Part

The DC part is a list of Duration Calculus formulae that restrict the timing behaviour of the CSP-OZ-DC class. Only the counterexample formulae introduced in section 2.3.6 are allowed. The formulae are separated by newline characters (*NL*).

$$\begin{aligned}
 DC &::= ce_formula \text{ NL} \dots \text{ NL } ce_formula \\
 ce_formula &::= \neg (phase \wedge (phase \mid events) \\
 &\quad \wedge \dots \wedge (phase \mid events) \wedge \mathbf{true}) \\
 phase &::= (\mathbf{true} \mid [Predicate])[\wedge \ell \sim t] \\
 &\quad [\wedge \boxplus NAME \dots \wedge \boxminus NAME] \\
 \sim &::= \leq \mid < \mid > \mid \geq \\
 events &::= \uparrow NAME \mid \downarrow NAME \\
 &\quad \mid events \vee events \mid events \wedge events
 \end{aligned}$$

3.2.2. Case Study

To illustrate the syntactic constructs of CSP-OZ-DC, we give as example the specification for a small part of an elevator. It is kept simple and only contains the core of the controller. This example is used as a case study to demonstrate the features of the combined language CSP-OZ-DC without bloating the specification.

Communication aspects are described with CSPz. For the controller of the elevator we use four signals (channels that carry no data values):

```
chan start, stop, passed
local_chan newgoal
```

The channel *newgoal* is an internal operation. It is used to determine the next floor that the elevator has to go to. The *start* operation is an external event used to start the engines. Likewise the *stop* event is used to stop the engines. The *passed* event is signalled by the environment whenever the elevator passed one floor. For the elevator controller the admissible sequences of events are defined by this CSP process:

$$\begin{aligned} \mathbf{main} &\stackrel{c}{=} \text{newgoal} \rightarrow \text{start} \rightarrow \text{Drive} \\ \text{Drive} &\stackrel{c}{=} (\text{passed} \rightarrow \text{Drive}) \square (\text{stop} \rightarrow \mathbf{main}) \end{aligned}$$

The elevator has a cyclic behaviour switching between the processes *main* and *Drive*. The elevator first chooses a new goal floor, then it starts the engines and switches to the *Drive* process. It can either pass a floor and keep on driving, or stop and return to the *main* process. We will see later that the external choice between the *passed* and *stop* event is determined by the Object-Z and Duration Calculus part of the specification.

Data aspects are described by Object-Z state and operation schemas. The floors are modelled by integers ranging from the constants *Min* to *Max*. No concrete values for the boundaries are given but the only requirement is $Min < Max$. In Z these constants are declared in an axiomatic definition:

$Min, Max : \mathbb{Z}$	
$Min < Max$	

The internal state of the elevator is given by the following state schema. It contains two variables for *current* and *goal* floor and a variable *dir*, which describes the direction the elevator is heading to (1 for upwards, -1 for downwards). The initial values for the variables are given by the *Init* schema. One could also add an invariant like $Min \leq current \leq Max$ to the state schema, however, we later want to show that the remaining specification will always keep the value of *current* in this range. If we added the invariant to the state schema, it would hold trivially because a communication that lead to a violation of the invariant would be rejected.

$current, goal : \mathbb{Z}$ $dir : \{-1, 0, 1\}$	Init $goal = current = Min$ $dir = 0$
--	--

The link between events and states is established by communication schemas. By naming conventions, the following schema describes the effect that the *passed* event induces:

com_passed $\Delta(current)$
$current' = current + dir$

The Δ list on the first line contains the variables that are changed by the operation. In this case, only *current* is changed by adding the value of *dir*, which increases or decreases the floor counter depending on the value of *dir*.

For simplicity we abstracted from the set of requested floors and the algorithm to choose the next goal floor. Instead the goal floor is chosen non-deterministically from the range of all floors except the current one.

$com_newgoal$ $\Delta(goal)$
$Min \leq goal' \leq Max$ $goal' \neq current$

When the elevator starts, the controller chooses the direction in accordance with the position of the new goal floor: If the number of the goal floor is larger than the current floor, the elevator goes up, denoted by $dir' = 1$, if the goal floor is smaller, the elevator goes down.

com_start $\Delta(dir)$
$goal > current \Rightarrow dir' = 1$ $goal < current \Rightarrow dir' = -1$

Finally, the elevator is not allowed to stop before reaching the goal floor. This can be stated by a communication schema with an empty delta list that restricts the pre-state.

<i>com_stop</i>
$\Delta()$
<i>goal = current</i>

Real-Time aspects are described with Duration Calculus (DC) by counterexample formulae. In the case study, real-time properties ensure that the elevator stops when it reaches the goal floor instead of passing to the next floor. To achieve this, a minimum time of three seconds between two adjacent *passed* events is demanded. This is expressed by a negated counterexample where two consecutive *passed* events occur, with a time span of at most three seconds between the events.

$$\neg \diamond (\downarrow \textit{passed} \wedge \ell \leq 3 \wedge \uparrow \textit{passed}).$$

Furthermore it is claimed that the elevator stops within two seconds. The following formula states the impossibility of the stop event not occurring if the goal has been reached for at least two seconds.

$$\neg \diamond (\lceil \textit{current} \neq \textit{goal} \rceil \wedge (\lceil \textit{current} = \textit{goal} \rceil \wedge \ell \geq 2 \wedge \exists \textit{stop}))$$

The complete specification of the elevator is shown in Fig. 3.2. For all reachable states of the elevator the invariant $Min \leq \textit{current} \leq Max$ holds. However, a manual proof would be very complex, since does not only depend on the operation schemas but also on the real-time aspects of the elevator: The elevator will stop before it passes its goal floor. In section 6.4.4 we will present a method to prove this invariant automatically.

3.2.3. Semantics

When evaluating the semantics of a specification according to the Z standard [ISO02], the schema

<i>Name</i>
<i>SchemaText</i>

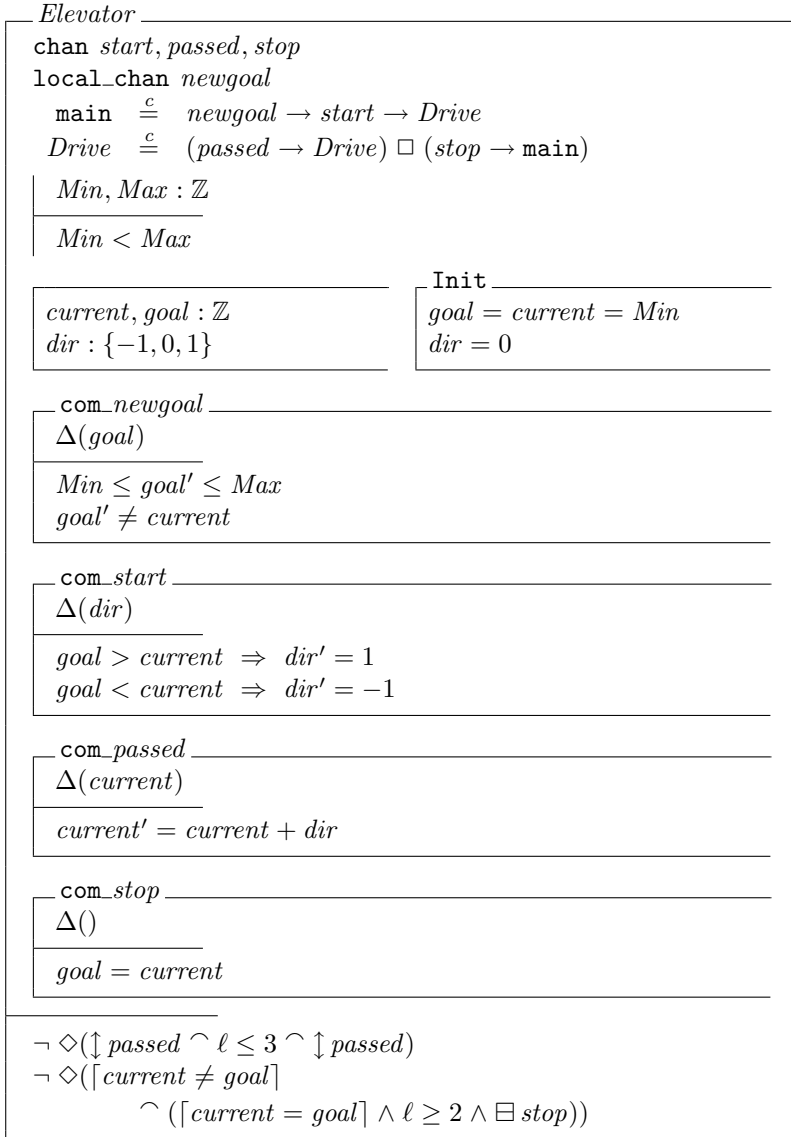


Figure 3.2.: Elevator specification

is rewritten to

$$Name == [SchemaText].$$

This defines a new constant $Name$ and sets its value to the expression $[SchemaText]$. For CSP-OZ-DC we proceed the same way. To this end, we extend Z expressions to include anonymous CSP-OZ-DC classes of the form $[COD_SchemaText]$. The syntax of $COD_SchemaText$ is the one for named CSP-OZ-DC classes:

$$\begin{aligned}
 COD_SchemaText ::= & \textit{Interface} \\
 & \textit{ProcessDeclaration} \\
 & [\textit{Paragraph} \dots \textit{Paragraph}] \\
 & \textit{State} \\
 & [\textit{Init}] \\
 & [\textit{Operation} \dots \textit{Operation}] \\
 & | \textit{DC}
 \end{aligned}$$

When determining the semantics, the following CSP-OZ-DC paragraph

$$\boxed{
 \begin{array}{l}
 Name[Formals](Params) \text{-----} \\
 Contents
 \end{array}
 }$$

it is rewritten to

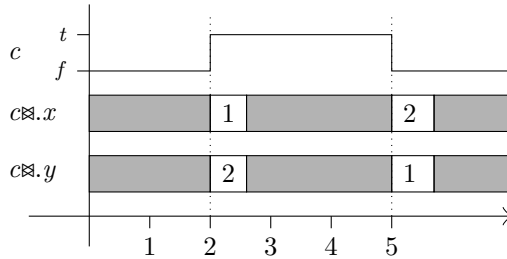
$$Name[Formals] == \lambda Params \bullet [Contents].$$

To provide semantics for a CSP-OZ-DC class, a semantic domain is necessary, in which the semantics for each language can be expressed. Duration Calculus extended by Z interpretations and CSP events is suitable for this. We give semantics for such a class in terms of the set of Duration Calculus interpretations that satisfy the specification. The semantics is based on the trace semantics for CSP, the history semantics for Object-Z, and the set of interpretations for Duration Calculus formulae.

As already mentioned in section 2.3, we use interpretations of the form $\mathcal{I} : \text{Time} \rightarrow \text{Model}$, where $\text{Time} == \mathbb{R}^+$ is the time domain and $\text{Model} == \text{NAME} \mapsto \mathbb{W}$ is the set of Z models. This is necessary to support complex data types that are used in the Object-Z part. The model must give values

for all variables used in the CSP-OZ-DC class. These include the variables declared in the state schema of the class and in axiomatic definitions in the CSP-OZ-DC class. Also this model includes a model of the environment, in which the class is declared. Furthermore, for each channel there is a Boolean *channel variable* with the same name. If a channel has parameters, e. g., if the channel was declared as $c : [x, y : \mathbb{Z}]$, there is another *parameter variable* that holds the parameters, in the example $c\bowtie : [x, y : \mathbb{Z}]$. The symbol \bowtie is just an arbitrary symbol to distinguish the parameter variable from the channel variable.

Whenever an event occurs, the channel variable corresponding to the channel changes and the parameter variable must be set to the value that is sent over the channel. The latter must be stable for some positive time. The value of the Boolean channel variable must not change until the next event occurs. The following diagram illustrates an interpretation in which the events $c \langle x == 1; y == 2 \rangle$ and $c \langle x == 2; y == 1 \rangle$ are sent at times 2 and 5.



Initially, the channel variable c is either **false** or **true**. In the diagram it starts with **false**. Its value changes whenever an event of channel c occurs. The value of the parameter variable $c\bowtie$ is only relevant immediately after an event on channel c occurs. At other times its value is arbitrary.

The variables from axiomatic definitions and from the environment must have the same value for all points in time $t : \text{Time}$. For the elevator example from the last section this means that *Min* and *Max* always keep their initial values. Also, the semantics of symbols declared by the (implicit) environment, e. g., \mathbb{Z} , $+$, $-$, etc. may not change during a single interpretation. There may be different interpretations with different values for *Min* and *Max* or different models of numbers, though.

An interpretation \mathcal{I} is in the set given by the semantics of a CSP-OZ-DC specification if it satisfies the CSP, Object-Z, and Duration calculus part. For the CSP and Z part only the untimed behaviour of \mathcal{I} is relevant. It

can be computed as follows: First the moments $0 = t_0 < t_1 < t_2 < \dots$ are determined when the interpretation $\mathcal{I}(t)$ changes. For almost all t in the interval $[t_i, t_{i+1}]$ the interpretation is constantly $\mathcal{I}(t) = M_i$ and it changes at t_{i+1} , so $M_i \neq M_{i+1}$. The interpretation may only change if an event occurs, i. e., the value of one of the channel variables c changes. In a CSP-OZ-DC class only one event can occur at a given time t_i , so there should be a unique c_i with $M_{i-1}(c_i) \neq M_i(c_i)$. If more than one or none of the channel variables change, the interpretation does not satisfy the specification. The parameters of the i -th event are given by $M_i(c_i\mathfrak{B})$. Thus, the i -th event is $c_i(M_i(c_i\mathfrak{B}))$. This produces a sequence

$$Untime(\mathcal{I}) = \langle M_0, c_1(M_1(c_1\mathfrak{B})), M_1, c_2(M_2(c_2\mathfrak{B})), M_2, \dots \rangle.$$

The sequence is finite if there is an n such that the interpretation is constant for almost all t in $[t_n, \infty]$ else it is infinite.

To determine whether an interpretation \mathcal{I} satisfies the CSP part of a CSP-OZ-DC part (denoted by $\mathcal{I} \models CSP_part$), we compute the operational semantics of the `main` process of the CSP part. This yields a labelled transition system $(\mathbb{W}_P, \mathbb{W}_E, p_0, \longrightarrow)$. Furthermore, the alphabet A of the CSP part is computed, which is the set of channels that syntactically occur in the CSP processes of the class. Of the sequence $Untime(\mathcal{I})$ only the events in the alphabet A are considered. The interpretation satisfies the CSP part if and only if there is a run of the labelled transition system corresponding to this sequence of events.

The interpretation satisfies the Object-Z part (denoted by $\mathcal{I} \models OZ_part$) if and only if the sequence $Untime(\mathcal{I})$ is in the history semantics [Smi92] of the Object-Z part of the CSP-OZ-DC specification. This means that the following must hold.

- $M_0 \in \llbracket Init \rrbracket^P$, i. e., the first element of the sequence is a model of the `Init` schema.
- $M_i \in \llbracket State \rrbracket^P$ for all i , i. e., all models M_i of the sequence are models of the `State` schema. Particularly, they satisfy the state invariant.
- $(M_{i-1} \cup M'_i \cup M_i(c_i\mathfrak{B})) \in \llbracket com_c_i \rrbracket^P$, i. e., the pre-state, post-state, and parameters fulfil the predicate given in the operation schema `com_c_i`. Here M'_i denotes the same model as M_i except that it provides values for primed variables, i. e., all names of the form v in the domain are replaced by v' .

In section 2.3, we already defined what it means that an interpretation \mathcal{I} satisfies a Duration Calculus formula F ($\mathcal{I} \models F$). The Duration Calculus part of a CSP-OZ-DC part is a list of formulae. The interpretation satisfies the complete DC part if it satisfies every formula. An equivalent formulation is that the interpretation satisfies the conjunction of all DC formulae of the class.

To integrate the semantics into the Z meta-language we extend the definition of $\llbracket \cdot \rrbracket^\varepsilon$ to CSP-OZ-DC classes as follows:

$$\left[\begin{array}{l} \textit{Interface} \\ \textit{CSP_part} \\ \textit{Paragraphs} \\ \textit{OZ_part} \\ | \textit{DC_part} \end{array} \right]^\varepsilon M = \begin{array}{l} \{M' : \textit{Model}; \mathcal{I}, \mathcal{I}' : \textit{Time} \rightarrow \textit{Model} \mid \\ (M, M') \in \llbracket \textit{Paragraphs} \rrbracket^\mathcal{D} \\ \wedge \forall t : \textit{Time} \bullet M' \subseteq \mathcal{I}(t) \\ \wedge \mathcal{I} \models \textit{CSP_part} \\ \wedge \mathcal{I} \models \textit{OZ_part} \\ \wedge \mathcal{I} \models \textit{DC_part} \\ \wedge \forall t : \textit{Time} \bullet \mathcal{I}'(t) = \textit{Interface} \triangleleft \mathcal{I}(t) \\ \bullet \mathcal{I}'\} \end{array}$$

The model M is the model of the environment. This model is extended according to the semantics of *Paragraphs* in the CSP-OZ-DC specification to a model M' that provides values for constants like *Min* and *Max* in the elevator specification. Each model is then extended to a interpretation \mathcal{I} . For each time t the model given by the interpretation $\mathcal{I}(t)$ must be an extension of the model M' . Furthermore, the interpretation \mathcal{I} must satisfy all conditions of the CSP, OZ, and DC part of the specification. Then the interpretation \mathcal{I}' is build from \mathcal{I} by restricting the models to include values for the variables occurring in the interface, only. Thus, the private state variables, local channels, and constants declared in the class are removed. The semantics of the class is the set of all interpretation \mathcal{I}' created according to the above procedure.

3.3. Parallel Composition of Systems

In CSP-OZ-DC large systems can be build from several classes. Unlike in CSP, we have only a single mechanism for parallel composition, namely conjunction. If c_1, c_2 are CSP-OZ-DC classes then we write $c_1 \wedge c_2$ to denote the parallel composition of the classes c_1 and c_2 . It is similar to schema conjunction: first the interpretations $\mathcal{I}_1, \mathcal{I}_2$ of c_1 and c_2 are computed. If these interpretation agree on their common domain, i. e., if

$\mathcal{I}_1(t) \cup \mathcal{I}_2(t)$ is a model, then the interpretation with $t \mapsto \mathcal{I}_1(t) \cup \mathcal{I}_2(t)$ is a valid interpretation of $c_1 \wedge c_2$. The exact definition is as follows:

$$\llbracket c_1 \wedge c_2 \rrbracket^{\mathcal{E}} M = \begin{array}{l} \{\mathcal{I}_1 : \llbracket c_1 \rrbracket^{\mathcal{E}} M; \mathcal{I}_2 : \llbracket c_2 \rrbracket^{\mathcal{E}} M \\ | \forall t : \mathbf{Time} \bullet \mathcal{I}_1(t) \cup \mathcal{I}_2(t) \in \mathbf{Model} \\ \bullet \lambda t : \mathbf{Time} \bullet \mathcal{I}_1(t) \cup \mathcal{I}_2(t)\} \end{array}$$

In CSP-OZ other parallel operators, e.g., linked parallel, are defined. These can be expressed by applying renaming before using conjunction. For renaming we use the same syntax as in Z. The expression $c_1[j_1/i_1, \dots, j_n/i_n]$ denotes the class c_1 where the communication channel i_k is renamed to j_k . The i_1, \dots, i_n must all be different. However, the case $c_1[j/i_1, \dots, j/i_n]$ is allowed, where multiple events i_1, \dots, i_n are renamed to the same event j . In that case, we require that at each point in time at most one event i_k occurs. Let \mathcal{I} be an interpretation of c_1 and let $t_1 < t_2 < \dots$ be the sequence of points of time when such an event occurs. Let k_1, k_2, \dots denote the indices of the events i_{k_j} that occur at time t_j . Then we define the interpretation \mathcal{I}' of the renamed class as

$$\mathcal{I}'(t)(j) = \begin{cases} \mathbf{false} & \text{if } t \in [0, t_1[, \\ \mathbf{false} & \text{if } t \in [t_{2^*j}, t_{2^*j+1}[, \\ \mathbf{true} & \text{if } t \in [t_{2^*j-1}, t_{2^*j}[, \text{ and} \end{cases}$$

$$\mathcal{I}'(t)(j\boxtimes) = \begin{cases} \text{arbitrary} & \text{if } t \in [0, t_1[, \\ \mathcal{I}(t)(i_{k_j\boxtimes}) & \text{if } t \in [t_j, t_{j+1}[. \end{cases}$$

Thus, the value of $\mathcal{I}'(t)(j)$ changes at each t_j , which is interpreted as the occurrence of the event j , and the value of $\mathcal{I}'(t)(j\boxtimes)$ equals the value $\mathcal{I}(t)(i_{k_j\boxtimes})$ of the last event that occurred before time t .

The last operator defined on CSP-OZ-DC classes is hiding, which just removes the corresponding channel variables and parameter variables from an interpretation:

$$\llbracket c \setminus (i_1, \dots, i_n) \rrbracket^{\mathcal{E}} M = \begin{array}{l} \{\mathcal{I} : \llbracket c \rrbracket^{\mathcal{E}} M \bullet \\ \lambda t : \mathbf{Time} \bullet \{i_1, \dots, i_n, i_1\boxtimes, \dots, i_n\boxtimes\} \triangleleft \mathcal{I}(t)\} \end{array}$$

With renaming, hiding, and conjunction we can express the remaining parallel operators of CSP: Linked parallel composition $c_1[in \leftrightarrow out]c_2$ can be expressed as $(c_1[x/in] \wedge c_2[x/out]) \setminus (x)$ where x is a fresh channel name. Interleaving $c_1 \parallel c_2$ is a special case of generalised parallel $c_1 \parallel_{\emptyset} c_2$ with an

empty synchronisation alphabet. Generalised parallel can be expressed by renaming all unsynchronised channels ch to unique names ch_1 in the first process and ch_2 in the second process. After conjunction, they are renamed back. For example, if c_1 and c_2 use only one channel ch , interleaving can be expressed as

$$c_1 \parallel_{\emptyset} c_2 = (c_1[ch_1/ch] \wedge c_2[ch_2/ch])[ch/ch_1, ch/ch_2].$$

3.4. Discussion and Related Work

3.4.1. CSPz

In this chapter we have presented a variant of CSPz that differs from Fischer's language presented in [Fis00]. The differences are only in some details and fixes some oddities of this language. The main syntactic change is the different channel declaration and the corresponding change to the multi-prefix operator. On the semantical side we additionally presented type-checking rules and provided a new Z semantics $\llbracket P \rrbracket^{\mathcal{E}}$ of a CSP process P that describes the structure of a process.

Channel Types

Fischer [Fis00] has two slightly different syntactic constructs for channel declaration for CSPz and CSP-OZ. For CSPz the declaration can be of an arbitrary type, while for CSP-OZ the type of the event must be a schema type. This has a simple reason: a Z schema references its parameters by name. Therefore it is convenient if every parameter that can be communicated over a channel gets a name that can be used for Object-Z. In textbook CSP, however, the parameters are traditionally identified by their position in a tuple.

Fischer therefore demanded that interface declarations in CSP-OZ have the form

$$\mathbf{chan} c : [p_1 : T_1; \dots; p_n : T_n]$$

When translating such a declaration into CSPz the names p_1, \dots, p_n were thrown away and the channel got a tuple type:

$$\mathbf{chan} c : T_1 \times \dots \times T_n$$

However, having a channel with two different types, one for CSP and another for Object-Z, is disturbing. It may also lead to subtle errors when the order of the parameters is accidentally swapped in two different classes. In [Fis00] the parameters are identified by their position instead of their name.

We presented in this chapter a way to work with channels having schema types. It only requires minimal syntactic changes to the multi-prefix operator to reference the parameters by name.

Semantics of CSPz

In [Fis00] Fischer defined the operational semantics for CSPz in parallel with the Z semantics. His definition has problems, though. For example, consider a simple recursive processes

$$P \stackrel{c}{=} a \rightarrow P \tag{3.1}$$

To determine the semantics of P we need to compute the semantics of the right-hand side $\llbracket a \rightarrow P \rrbracket^{\mathcal{E}} M_0$. The model M_0 cannot provide a value for P because its semantics has not been computed, yet. According to [Fis00] page 64, $\llbracket \cdot \rrbracket^{\mathcal{E}}$ is considered to be an extension of the operational semantics $\llbracket \cdot \rrbracket^{\mathcal{O}}$. Therefore, $\llbracket a \rightarrow P \rrbracket^{\mathcal{E}} M_0$ should denote the operational semantics of $a \rightarrow P$. However, the recursive equation of P is not accessible through M_0 , hence the operational semantics cannot be fully given.

We solve this by separating the Z semantics $\llbracket \cdot \rrbracket^{\mathcal{E}}$ from the operational semantics $\llbracket \cdot \rrbracket^{\mathcal{O}}$. The Z semantics of the above process $\llbracket a \rightarrow P \rrbracket^{\mathcal{E}} M_0$ is just an abstract syntax tree of the expression $prefix(\llbracket a \rrbracket^{\mathcal{E}} M_0, call(P))$. Only the value of a is further evaluated according to the underlying model M . The operational model is only computed for the complete paragraphs (3.1). The labelled transition system uses the model of the complete specification. In this model the value of P is the abstract syntax tree, which suffices to give the operational semantics of P and $a \rightarrow P$.

In [Fis00] the configurations of the labelled transition system are built from a process expression and a model M . The model M is used to evaluate the variables in the event part of the prefix operator or to evaluate the parameters when another process is called. However, sometimes the operational semantics fails to evaluate the expressions. For example, consider the configuration

$$(\square x : 0..1 \bullet \text{if } x == 0 \text{ then } b \rightarrow Stop \text{ else } Stop, M).$$

The process part is of the form $\square e \bullet p$, for which Fischer defined the following rule in [Fis00] page 74.

$$\frac{\forall t : \llbracket e \rrbracket^{\mathcal{E}} M \bullet C(t) = (p, M \oplus t)}{(\square e \bullet p, M) \xrightarrow{\tau} \square \llbracket e \rrbracket^{\mathcal{E}} M \bullet C} [\llbracket e \rrbracket^{\mathcal{E}} M \neq \emptyset]$$

In this case, e is $x : 0..1$ and $\llbracket x : 0..1 \rrbracket^{\mathcal{E}}$ is the set:

$$\{\langle x == 0 \rangle, \langle x == 1 \rangle\}$$

The process p is if $x == 0$ then $b \rightarrow Stop$ else $Stop$ and the function C can be explicitly written as

$$C = \{\langle x == 0 \rangle \mapsto (p, M \oplus \{x \mapsto 0\}), \\ \langle x == 1 \rangle \mapsto (p, M \oplus \{x \mapsto 1\})\}$$

The next rule that should be applied is

$$\frac{\exists t : F \bullet C(t) \xrightarrow{a} c'}{\square F \bullet C \xrightarrow{a} c'} [a \in \Sigma^{\vee}]$$

For $t = \langle x == 0 \rangle$ this rule is only applicable if $C(t) \xrightarrow{b} c'$ holds. However, the configuration $C(t)$ is

$$(\text{if } x == 0 \text{ then } b \rightarrow Stop \text{ else } Stop, M \oplus \{x \mapsto 0\})$$

and there is no rule for the if construct. Fischer assumes that this is somehow simplified to $b \rightarrow Stop$, since the value of x is 0 in the model. However, this is never made explicit.

With our approach the if construct is evaluated by the Z semantics as follows.

$$\begin{aligned} & \llbracket \square x : 0..1 \bullet \text{if } x == 0 \text{ then } b \rightarrow Stop \text{ else } Stop \rrbracket^{\mathcal{E}} M \\ &= \text{extchoice}(\llbracket \lambda x : 0..1 \bullet \text{if } x == 0 \text{ then } b \rightarrow Stop \text{ else } Stop \rrbracket^{\mathcal{E}} M \\ &= \text{extchoice}(\{0 \mapsto \llbracket \text{if } x == 0 \text{ then } b \rightarrow Stop \text{ else } Stop \rrbracket^{\mathcal{E}} M \oplus \{x \mapsto 0\}, \\ & \quad 1 \mapsto \llbracket \text{if } x == 0 \text{ then } b \rightarrow Stop \text{ else } Stop \rrbracket^{\mathcal{E}} M \oplus \{x \mapsto 1\}\}) \\ &= \text{extchoice}(\{0 \mapsto \text{prefix}(b, Stop), 1 \mapsto Stop\}). \end{aligned}$$

3.4.2. Semantics of CSP-OZ-DC

The first semantics definition of CSP-OZ-DC was given in [HO02a] and [HO02b] based on the failures-divergence semantics of CSP-OZ. It introduces a state variables tr that records all communicated events. This observable directly corresponds to the traces of the trace semantics $\mathcal{T}(P)$ of the underlying CSP process P . Our new approach uses a separate state variable for each event instead of the monolithic variable tr . Due to this alphabetised parallel $A \parallel_B$ can be expressed much simpler by using conjunction, provided A and B is the alphabet of the underlying processes. In [HO02a], the trace variables of each individual process had to be renamed. The renamed trace variables were then combined into a new trace and hidden to the outside:

$$\begin{aligned} \exists tr_1, tr_2 \bullet F_1[tr_1/tr] \wedge F_2[tr_2/tr] \\ \wedge [tr \in \text{seq}(A \cup B) \wedge tr \upharpoonright A = tr_1 \wedge tr \upharpoonright B = tr_2] \end{aligned}$$

Another difference of the semantics in [HO02a] is the inclusion of the acceptance set. Thus, it provides a richer semantics where deadlocks can be observed. Via the *enable* predicate the DC part could even reference the acceptance set. However, this extension has a problem because it is not considered whether the DC part allows an event. For example, in the DC formula $\neg \diamond([P] \wedge \ell > 1 \wedge \downarrow e)$, the event e may not occur if state P was observed for more than a second. Nonetheless this DC formula does not modify the acceptance set. Hence, a deadlock cannot always be detected. The acceptance semantics is only valid for the CSP-OZ part of a class. For future work, it may be useful to consider a failure or acceptance semantics using the operational semantics given in the next chapter, which also considers the DC part.

In [HO02a] the semantics of CSP-OZ is reused which translates the Z part into a CSP process, first. Unfortunately, this translation hides the state space from the other parts of the specification, i.e., the DC part cannot reference variables from the Z part. Our new semantics of the Z part is based on its history semantics [Smi92], which provides both, communicated events and state space. Since the CSP semantics of Object-Z was designed by Fischer to be compatible with the history semantics, our new semantics of CSP-OZ-DC is compatible to the one given in [HO02a].

3.4.3. Parallel Composition

The language CSP-OZ [Fis00] had a CSP semantics; this makes it easy to create larger systems from several processes. CSP supports several parallel operators: full interleaving, synchronisation on common alphabet, synchronisation on a given alphabet, linked parallel. Also other CSP operators like renaming, hiding, etc. are available. We have only provided a renaming and hiding operator for CSP-OZ-DC classes which corresponds to CSP renaming and hiding. Furthermore, we defined a single parallel operator \wedge (conjunction) which implements alphabetised parallel. However, we showed that all other parallel operators can be implemented with renaming, hiding and conjunction.

3.4.4. Related Work

CSP-OZ-DC is not the only language that combines CSP, Z, and a real-time formalism. A related language is Real-Time Object-Z (RTOZ) [SH99]. This language is based on Object-Z. It introduces a special variable τ , that can be used in operations. Unlike operations in CSP-OZ-DC, operations in RTOZ take time. The start time of an operation is given by τ , the end time by the post state τ' . Hence, the maximum execution time can be specified by adding the formula $\tau' - \tau < \text{max}$ to the schema.

Furthermore, formulae in an interval logic can be used to restrict the timing behaviour. For example, the formula

$$\langle \delta \geq 10 \rangle \subseteq \langle \text{true} \rangle; \langle \text{SendData} \rangle; \langle \text{true} \rangle$$

given in [Smi02] demands that in any interval of 10 time units the operation *SendData* needs to be executed at least once. It is equivalent to the DC formula

$$\ell \geq 10 \Rightarrow \mathbf{true} \wedge \text{SendData} \wedge \mathbf{true},$$

and thus to the counterexample formula $\neg \diamond(\ell \geq 10 \wedge \Box \text{SendData})$. In [Smi02] a CSP semantics for RTOZ is given. This allows Smith to define semantics for parallel classes using CSP parallel operators.

In RTOZ the timing behaviour is not strictly separated from the logical behaviour: it can be either given by restricting the values of τ and τ' in the state schema, or by using the interval logic. In our logic the timing behaviour is kept strictly separated. Moreover, the interval logic used in RTOZ is designed from the scratch instead of using the well-researched

language DC. No decidability results and no tool support exist for this language. In [Smi02], CSP is used only to build larger systems from multiple classes. It cannot be used to specify the order in which operations may be invoked. For this, the interval logic has to be used, which is not suited for this purpose.

A different approach is the combination of Timed CSP [DS95] and Z. The languages TCOZ [MD98] and RT-Z [Süh02] follow this approach. Timed CSP is similar to CSP but introduces a new operator *timeout*, denoted by $P \triangleright \{d\}Q$. This process behaves like P if P performs a visible event within d seconds. Otherwise, it waits for d time units and behaves as Q afterwards. Thus, $STOP \triangleright \{d\}Q$ waits for d time units and behaves as Q afterwards. In the process $a \rightarrow P \triangleright \{d\}STOP$ either the event a occurs within d time units or the system will end in a deadlock. It is not possible to state that the event a must occur after d time units. To denote the operational semantics of Timed CSP, there is a new kind of transition $\overset{d}{\rightsquigarrow}$ for the passage of time. Also the timing behaviour is closely integrated into the CSP specification. In CSP-OZ-DC the timing behaviour is kept in a separate part of the class.

The language RT-Z [Süh99, Süh02] is a combination of Timed CSP and Z. It uses a different syntax but the components are similar to CSP-OZ-DC: A specification unit has an interface declaring CSP channels of some type. It has a behaviour that corresponds to the CSP part in CSP-OZ-DC. The only difference is the usage of Timed CSP instead of CSPz. The state is defined as Z schema, there is an init schema and some operation schemas. A difference in RT-Z is that operation schemas can only be referenced by the CSP process in the behaviour part. They cannot be part of the exported interface as they are in CSP-OZ-DC.

TCOZ [MD98, MD99b, MD99a] is a combination of Timed CSP and Object-Z. In this language, communication, data and time are not separated. Instead the syntax for CSP is extended by Z operations. A Z operation is interpreted as a CSP process that changes the state space and terminates.

4. Phase Event Automata

Contents

4.1. Prerequisites	69
4.2. Syntax of Phase Event Automata	72
4.3. Operational Semantics	75
4.4. Automata and Formulae	81
4.5. Deterministic Automata	83
4.6. Case Study: Audio Control Protocol	86
4.7. Discussion and Related Work	90
4.7.1. Discussion	90
4.7.2. Other Timed Automata Models	92

Fischer gives the semantics for CSP-OZ in CSPz, which has an operational semantics in form of a labelled transition system. The advantage of this approach is the easy integration into the model checker FDR. There is a simple transformation for a subset of CSPz (with finite domains) into CSP_M, the language of FDR. For CSP_M the transition system semantics can be computed using FDR. However, when adding real-time behaviour a different model checker is needed, because FDR cannot handle real-time issues.

For checking real-time systems, timed automata are commonly used as modelling language. They were introduced by Alur and Dill [AD94]. A timed automaton is a Büchi automaton extended with clock variables. These variables can be used to measure time and force some action to occur within certain time bounds.

Our first approach for model checking CSP-OZ-DC was the following. First the CSP-OZ part of the specification is translated to CSP_M. This is used as input language for the FDR model checker. FDR can produce a finite automaton, provided that the CSP-OZ class only uses finite data types. This automaton is interpreted as timed automaton that allows arbitrary timing behaviour. In the last step the DC part is added to this automaton. For each formula certain annotations are added to the automaton to assure that the timing restrictions are met.

The approach has several disadvantages. In the resulting automaton there is no separation of the behavioural, data, and timing part, anymore. They are intermixed into a single large automaton. It is not possible to check several separated classes that run in parallel. This is because the synchronisation in off-the-shelf timed automata model checker, e.g., Uppaal, use a different type of synchronisation. Furthermore, in the finite automaton FDR produces, the data part is not visible anymore. Therefore it is not possible to use DC formulae that reference variables from the Z part.

To build systems from several small automata, a parallel composition is needed that matches the parallel operator of CSP-OZ-DC. In the model of Alur and Dill the parallel components are synchronised over common events. Each component has an alphabet and every event that occurs in both alphabets must occur synchronously. This behaviour matches the alphabetised parallel operator $A \parallel_B$ of CSP, which is used to synchronise the CSP and the Z part of a specification. However, these automata do not define variables that are necessary to model the Z part and to synchronise the Z part with the Duration Calculus part.

To overcome these difficulties, a new automaton model is helpful. The automata have to meet the following requirements. First, the synchronisation of events has to match the CSP synchronisation over a common alphabet. Second, the data variables should be visible. Last, the parallel composition should be semantically equivalent to conjunction in Duration Calculus. The first requirement is needed to build larger systems as separate automata running in parallel. It is also needed to synchronise the CSP and Z part with the semantics given in the previous chapter. The second requirement is necessary to let the DC part reference variables from the Z part. The last requirement makes it possible to translate the DC formulae into separate automata that run in parallel and supervise the timing behaviour.

In this chapter this new kind of automaton, called *phase event automaton*, is presented. Phase event automata have a built-in notion of state assertions and events. To support the real-time behaviour, the concepts of clocks from the timed automata model [AD94] are used. An operational semantics for the automaton model is given, which is a prerequisite to develop a model checking algorithm.

Figure 4.1 gives an example for a phase event automaton. It models the behaviour of a simple watchdog that observes a boolean variable *danger*. If this variable is **true**, the watchdog must issue an event *alarm* after at most

two seconds. If the variable *danger* is reset to false within two seconds, the alarm does not have to occur.

The automaton has three locations each labelled with a state invariant (in this case either *danger* or \neg *danger*). As long as the automaton is in a certain location, its state invariant has to hold. Transitions can be labelled with guards. If the guard is omitted, the edge is always enabled. The automaton has two initial locations p_0 and p_1 . If the initial value of the variable *danger* is **false**, it will start in p_0 otherwise in p_1 . When the variable *danger* changes from **false** to **true**, the automaton has to leave location p_0 and enter location p_1 . Taking this transition it resets the clock variable c to zero. This clock variable now measures the time the automaton stays in location p_1 . Due to the state invariant p_1 cannot be active for more than two seconds, so one of the outgoing transitions has to be taken. The first edge allows to go back to p_0 if the variable *danger* is reset to **false**. The second edge forces the event *alarm* to occur and switches to p_2 .

Each location needs a loop transition that can be taken if nothing relevant to the automaton changes. This loop edge allows an automaton to perform stuttering steps. Stuttering steps help to simplify the definition of the parallel composition. In our parallel composition only synchronous steps are allowed but automata that do not participate in this step can do a stuttering step. This loop edge can also be labelled by events that must not occur while the automaton is in a certain location. For example, the location p_1 must be left if the *alarm* event occurs because the loop edge forbids this event.

4.1. Prerequisites

As shown in Figure 4.1, the nodes and edges of the automaton can be labelled by formulae. For a set of variables V we denote with $\mathcal{L}(V)$ the language of predicates referencing only variables in V . We assume that the variables are declared as Z variables in a global context. The language used for these constraints is a subset of the language of Z predicates. This subset has to be chosen to meet two opposing constraints. On the one hand, it must be expressive enough so that specifications with complex data type can be translated. On the other hand, the subset should permit automatic verification and model checking.

Like the metalanguage of Z we use mappings from a subset of variable

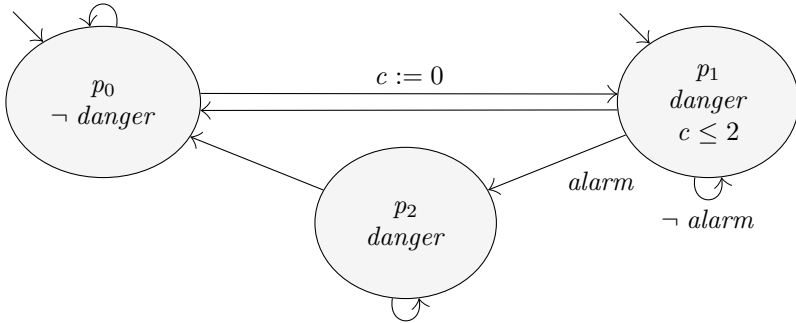


Figure 4.1.: Phase event automaton for a watchdog

names $NAME$ to the universe of data value \mathbb{W} . These mappings are called *valuations*, as they give values to variable names. For a valuation β and a formula g , we write $\beta \models g$ to denote that g holds for the valuation β . In terms of the Z metalanguage this is denoted by $\beta \in \llbracket g \rrbracket^{\mathcal{P}}$, which states that β is a model for the formula g . If we have multiple valuations β_1, β_2 for distinct sets of variables, we use the notation $\beta_1, \beta_2 \models g$ to denote that g holds for the combined valuation $\beta_1 \cup \beta_2$.

To denote the change of the state, we use unprimed and primed variables to denote the pre- and post-state of a transition, respectively. The same mechanism is used for operations in Object-Z. If we have a valuation β for a set of unprimed variables, we denote with β' the valuation for primed variables, where $\beta'(v') = \beta(v)$ for each variable v in the domain of β .

For specifying real-time behaviour, we use the notion of *clock variables*, *clock constraints* and *clock valuation* following Alur and Dill [AD94]: A clock variable stores values of type Time , in our case continuous time $\text{Time} == \mathbb{R}^+$. The value of clock variables automatically increases as time passes. They can be explicitly reset to zero by taking transitions and then are measuring the time passed since this transition.

The sublanguage $\mathcal{L}(C)$ for clock variables should allow the basic predicates known from timed automata. The atoms are comparisons of a clock variable against a rational number. These atoms may be joined by arbitrary boolean operators to build formulae.

For state invariants we require a different more restricted set of clock constraints, named *convex* clock constraints. These only allow conjunction

of atoms of the form $c \leq T$, where c is a clock variable and T a rational number. Convex clock constraints have the following useful property: if they hold at a certain moment they also hold for all earlier moments unless the clocks involved were reset.

Definition 4.1. For a set C of clock variables, the set $\mathcal{L}(C)$ of *clock constraints* δ is defined by the following grammar:

$$\delta ::= c \leq T \mid c < T \mid \neg \delta \mid \delta \wedge \delta$$

where $c \in C$ is a clock and $T \in \mathbb{Q}^+$ (the positive rational numbers) is a constant.

The set $\mathcal{L}^c(C)$ of *convex clock constraints* δ^c are defined by the following grammar

$$\delta^c ::= c \leq T \mid c < T \mid \delta^c \wedge \delta^c$$

A *clock valuation* γ for a set C of clocks is a function $\gamma : C \rightarrow \text{Time}$ that assigns to each clock a non-negative real value. For each $t \in \text{Time}$, $\gamma + t$ denotes the clock valuation where each clock is increased by t , i.e. $(\gamma + t)(c) = \gamma(c) + t$. For a set $X \subseteq C$, $\gamma[X := 0]$ denotes the clock valuation where each clock in X is reset to zero while the values of the other clocks remain unchanged.

If for the valuation γ the clock constraint δ holds, we say that γ *satisfies* δ , denoted by $\gamma \models \delta$.

When an automaton enters a location, we demand that it stays there for some positive time. If the invariant forces it to leave the location immediately, there is a contradiction. Therefore, we only allow entering locations if the invariant still holds after a short positive duration. To express this condition we define an operator *strict*. This operator builds for a convex clock constraints (the clock invariant) the corresponding strict constraint: every \leq -operator is replaced by an $<$ -operator. It can be defined inductively as follows:

$\begin{aligned} & \text{strict} : \mathcal{L}^c(C) \rightarrow \mathcal{L}^c(C) \\ \forall c : C; T : \mathbb{Q}^+; \delta_1, \delta_2 : \mathcal{L}^c(C) \bullet \\ & \text{strict}(c \leq T) = (c < T) \\ & \wedge \text{strict}(c < T) = (c < T) \\ & \wedge \text{strict}(\delta_1 \wedge \delta_2) = (\text{strict}(\delta_1) \wedge \text{strict}(\delta_2)) \end{aligned}$
--

The following lemma proves some useful properties of convex clock constraints.

Lemma 4.2. *If $\delta \in \mathcal{L}^c(C)$ is a convex clock constraint then for all clock valuations γ , all $t > 0$ and all $t' \geq 0$ the following holds:*

$$\gamma + t \models \delta \text{ implies } \gamma \models \text{strict}(\delta), \quad (4.1)$$

$$\gamma \models \text{strict}(\delta) \text{ implies } \gamma \models \delta, \quad (4.2)$$

$$t' < t \wedge \gamma + t \models \delta \text{ implies } \gamma + t' \models \text{strict}(\delta). \quad (4.3)$$

Proof. The first two formulae are easily shown by induction over the structure of δ . The last formula follows from (4.1):

$$\gamma + t' + (t - t') \models \delta \text{ implies } \gamma + t' \models \text{strict}(\delta).$$

□

4.2. Syntax of Phase Event Automata

In this section we give the definition and intuitive semantics for phase event automata. We also define the mentioned parallel composition operator.

Definition 4.3. A *phase event automaton* $\mathcal{A} = (P, V, A, C, E, s, I, E_0)$ consists of the following components:

- P : a set of *locations (phases)*.
- $V \subseteq \text{NAME}$: a set of typed *state variables*.
- $A \subseteq \text{NAME}$: a set of boolean *event variables*.
- C : a finite set of real-valued *clocks*.
- $E \subseteq P \times \mathcal{L}(V \cup A \cup C \cup V') \times \mathbb{P}(C) \times P$: a set of *edges*. An element (p, g, X, p') represents an edge from p to p' . The current valuation of state and clock variables and events must satisfy the guard g . All clocks in X are reset after the transition.
- $s : P \rightarrow \mathcal{L}(V)$: a labelling function that associates each location with a *state invariant* that must hold while this location is active.

- $I : P \rightarrow \mathcal{L}^c(\text{Clocks})$: a function assigning to each location a *clock invariant*.
- $E_0 \subseteq \mathcal{L}(V) \times P$: a set of *initial edges*. An element (g, p) allows the automaton to start in p if the state predicate g holds.

Intuitively, the automaton starts by taking an initial edge $(g, p) \in E_0$. Both formulae g and $s(p)$ hold for the initial values of the variables. All clocks are initially set to zero and are then accumulating the time that passed since they were last reset. Whenever a variable changes or an event occurs, the automaton has to take some edge $(p, g, X, p') \in E$ from the current location p . The guard g must hold for the current valuation of the clocks, the previous and the new values of the state variables, and for the events that occur. Furthermore, the state and clock variables satisfy the invariants of the new location $s(p')$ and $\text{strict}(I(p'))$. The automaton switches to the new location p' and resets all clocks in X to zero. Each edge is taken instantaneously; however, after taking an edge there must be some delay before taking the next edge. This is necessary because there is no super-step semantics in Duration Calculus. During this non-zero delay the new location is active and the clocks are increased accordingly. Of course, the delay has to ensure that the clock invariant $I(p)$ remains satisfied. Since the clock invariant holds strictly when entering a location, there is always a non-zero delay that keeps the invariant satisfied.

It is convenient to let the automaton do steps at any time without changing the current location if the valuations of the state variables do not change and no event that is relevant for this automaton occurs. Lamport calls this behaviour *stuttering* [Lam83], because nothing relevant for this automaton changes. These stuttering steps are useful when running several automata in parallel. Parallel automata should be able to do steps either independently or synchronously, which complicates the definition of parallel composition. However, if every automaton is stuttering invariant, i. e., it can make a stuttering step at any time, parallel composition can be defined with synchronous steps only.

Therefore all automata should be stuttering invariant in the following sense: Every location has a loop edge, whose guard is satisfied if no global variable in V changes, no event in A occurs, and the invariant of the location is satisfied. If the location has a clock invariant, the guard of the loop edge only needs to hold if the clock invariant is strictly satisfied.

Definition 4.4. An automaton \mathcal{A} is *stuttering invariant*, if for each location $p \in P$ there is an *stuttering edge* $(p, g, \emptyset, p) \in E$, with

$$(\forall x : V \bullet x = x') \wedge (\forall e : A \bullet \neg e) \wedge s(p) \wedge \text{strict}(I(p)) \Rightarrow g.$$

For parallel composition of automata we can assume that all automata run synchronously. All automata that are not interested in a computation step do a stuttering step. The automata synchronise over both events and global variables. A global variable may only be changed if all automata agree or do not care about that variable. Likewise an event may only occur if all automata allow it. For two parallel automata the product automaton can be constructed as follows.

Definition 4.5. The *parallel composition of two automata* \mathcal{A}_1 and \mathcal{A}_2 , $\mathcal{A}_i = (P_i, V_i, A_i, C_i, E_i, s_i, I_i, E_{0,i})$, is the product automaton

$$\mathcal{A} = (P, V, A, C, E, s, I, E_0)$$

defined as follows:

- $P := \{p_1 : P_1; p_2 : P_2 \mid s(p_1) \wedge s(p_2) \not\Leftarrow \text{false} \bullet (p_1, p_2)\}$. The set of locations P is the cross product of P_1, P_2 where the locations having a combined state invariant equivalent to false are removed.
- $V := V_1 \cup V_2$ and $A := A_1 \cup A_2$.
- $C := C_1 \dot{\cup} C_2$. The clock set is the disjoint union of C_1 and C_2 ; clocks that appear in both sets need to be renamed.
- The edges E are the tuples $((p_1, p_2), g_1 \wedge g_2, X_1 \cup X_2, (p'_1, p'_2))$ for each pair of edges $(p_i, g_i, X_i, p'_i) \in E_i, i = 1, 2$. Edges with $g_1 \wedge g_2$ equivalent to **false** can be removed.
- $s(p_1, p_2) = s_1(p_1) \wedge s_2(p_2)$ The state invariant for a location of \mathcal{A} is the conjunction of the state invariants of the corresponding locations of \mathcal{A}_1 and \mathcal{A}_2 .
- $I(p_1, p_2) = I_1(p_1) \wedge I_2(p_2)$ Likewise the clock invariant is the conjunction of the corresponding clock invariants of \mathcal{A}_1 and \mathcal{A}_2 .
- The set of initial edges E_0 consists of tuples $(g_1 \wedge g_2, (p_1, p_2))$ for each pair of initial edges $(g_i, p_i) \in E_{0,i}$.

When abstracting from the names of the locations P , the product automaton $\mathcal{A}_1 \parallel \mathcal{A}_2$ is equivalent to $\mathcal{A}_2 \parallel \mathcal{A}_1$ because the above definition is symmetric. Likewise the parallel composition is associative, i. e., $(\mathcal{A}_1 \parallel \mathcal{A}_2) \parallel \mathcal{A}_3$ is equivalent to $\mathcal{A}_1 \parallel (\mathcal{A}_2 \parallel \mathcal{A}_3)$. Moreover, stuttering invariance is preserved under parallel composition:

Lemma 4.6. *The parallel product $\mathcal{A}_1 \parallel \mathcal{A}_2$ of two stuttering invariant automata $\mathcal{A}_1, \mathcal{A}_2$ is also stuttering invariant.*

Proof. For each location $(p_1, p_2) \in P$ there are edges $(p_i, g_i, \emptyset, p_i) \in E_i$ for $i = 1, 2$ with

$$(\forall x : V_i \bullet x = x') \wedge (\forall e : A_i \bullet \neg e) \wedge s(p_i) \wedge \text{strict}(I(p_i)) \Rightarrow g_i$$

Therefore $\mathcal{A}_1 \parallel \mathcal{A}_2$ contains the stuttering edge $((p_1, p_2), g_1 \wedge g_2, \emptyset, (p_1, p_2))$ and

$$\begin{aligned} (\forall x : V \bullet x = x') \wedge (\forall e : A \bullet \neg e) \wedge s(p_1, p_2) \wedge \text{strict}(I(p_1, p_2)) \\ \Rightarrow g_1 \wedge g_2. \end{aligned}$$

This proves that $\mathcal{A}_1 \parallel \mathcal{A}_2$ is stuttering invariant. \square

4.3. Operational Semantics

The operational semantics is defined in terms of runs, which are sequences of configurations. A configuration describes the state of the automaton and its environment, i. e., values of clocks, variables, events, and the current location. The configurations of an automaton $\mathcal{A} = (P, V, A, C, E, s, I, E_0)$ are formally defined as

$$PEAConfig = P \times \mathbb{P}A \times \mathbf{Val}(V) \times \mathbf{Val}(C) \times \mathbf{Time}.$$

A configuration $(p, Y, \beta, \gamma, t) \in PEAConfig$ describes an interval of duration t where the automaton is in location p , Y is the set of events occurring at the beginning of the interval, the valuations of the variables in V are given by β and the valuations of the clocks at the beginning of the interval are given by γ . No events occur during the interval.

Definition 4.7. A run is a finite sequence of configurations

$$\langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots, (p_n, Y_n, \beta_n, \gamma_n, t_n) \rangle \in \text{seq } PEAConfig$$

that satisfies the following conditions:

1. $Y_1 = \emptyset$,
2. $\beta_1 \models g$ for some initial edge $(g, p_1) \in E_0$,
3. $\gamma_1(c) = 0$ for all $c \in C$,
4. $t_i > 0$ for all $i \in 1..n$,
5. $\beta_i \models s(p_i)$ for all $i \in 1..n$,
6. $\gamma_i + t_i \models I(p_i)$ for all $i \in 1..n$,
7. For all $i \in 1..(n-1)$ there is an edge (p_i, g, X, p_{i+1}) with

$$\beta_i, \beta'_{i+1}, \gamma_i + t_i, Y_{i+1} \models g \text{ and } \gamma_{i+1} = (\gamma_i + t_i)[X := 0].$$

We require $Y_1 = \emptyset$ because our definition of $\uparrow e$ in section 2.3.5 does not allow an event to occur at time zero. Furthermore, we require $t_i > 0$ for all i , i.e., the automaton stays in each location for some non-zero time. Because the invariant needs to hold at the end of the interval, it must hold strictly at the beginning of the interval by lemma 4.2, i.e., $\gamma_i \models \text{strict}(I(p_i))$.

The set of runs of a parallel product is the intersection of the set of runs of the component automata. This allows compositional reasoning. If the runs of one component satisfy a safety property, the runs of the complete system also do.

Lemma 4.8. *A run $= \langle \langle (p_{1,1}, p_{1,2}), Y_1, \beta_1, \gamma_1, t_1 \rangle, \dots \rangle$ is a run of $\mathcal{A}_1 \parallel \mathcal{A}_2$ if and only if each of its projections*

$$\text{run}_j = \langle \langle p_{1,j}, A_j \cap Y_1, V_j \triangleleft \beta_1, C_j \triangleleft \gamma_1, t_1 \rangle, \dots \rangle$$

is a run of \mathcal{A}_j for $j = 1$ and $j = 2$.

Proof. Let $\mathcal{A}_j = (P_j, V_j, A_j, C_j, E_j, s_j, E_{0,j})$ and

$$\mathcal{A}_1 \parallel \mathcal{A}_2 = (P, V, A, C, E, s, I, E_0).$$

First we show that run_j is a run of \mathcal{A}_j for $j = 1, 2$ if run is a run of $\mathcal{A}_1 \parallel \mathcal{A}_2$. We check the seven conditions of definition 4.7 for the projections run_j . From $Y_1 = \emptyset$, we get $A_j \cap Y_1 = \emptyset$ for $j = 1, 2$. In $\mathcal{A}_1 \parallel \mathcal{A}_2$, we have $\beta_0 \models g$ for some initial edge $(g, (p_{0,1}, p_{0,2})) \in E_0$. From the definition of parallel composition follows that $g = g_1 \wedge g_2$ with $(g_j, p_{0,j}) \in E_{0,j}$ for

$j = 1, 2$. Then, $\beta_0 \models g_j$ for $j = 1, 2$ and since g_j can only reference variables from V_j , also $(V_j \triangleleft \beta_0) \models g_j$. From $\gamma_1(c) = 0$ for all $c \in C$, we have $C_j \triangleleft \gamma_1(c) = \gamma_1(c) = 0$ for all $c \in C_j$. Also $t_i > 0$ for all $i \in 1..n$, since run was a valid run. The state invariants of the locations in $\mathcal{A}_1 \parallel \mathcal{A}_2$ hold, so we have $\beta_i \models s(p_{i,1}, p_{i,2}) = s(p_{i,1}) \wedge s(p_{i,2})$ and therefore $(V_j \triangleleft \beta_i) \models s(p_{i,j})$ for $j = 1, 2$. Similarly, for the clock invariants.

For all $i \in 1..(n-1)$ there is an edge $((p_{i,1}, p_{i,2}), g, X, (p_{i+1,1}, p_{i+1,2})) \in E$. By definition of E we have edges $(p_{i,j}, g_j, X_j, p_{i+1,j}) \in E_j$ for $j = 1, 2$ and $g = g_1 \wedge g_2$, $X = X_1 \cup X_2$. Because $X_j \subseteq C_j$ and C_1 and C_2 are disjoint we have $X_j = X \cap C_j$. From $\beta_i, \beta'_{i+1}, \gamma_i + t_i, Y_{i+1} \models g = g_1 \wedge g_2$ we have that $V_j \triangleleft \beta_i, (V_j \triangleleft \beta_{i+1})', C_j \triangleleft \gamma_i + t_i, A_j \cap Y_{i+1} \models g_j$. Also $C_j \triangleleft \gamma_{i+1} = C_j \triangleleft ((\gamma_i + t_i)[X := 0]) = ((C_j \triangleleft \gamma_i) + t_i)[X \cap C_j := 0] = ((C_j \triangleleft \gamma_i) + t_i)[X_j := 0]$. Therefore run_j is a run of \mathcal{A}_j for $j = 1, 2$.

Now suppose run_j is a run of \mathcal{A}_j for $j = 1, 2$. From $A_j \cap Y_1 = \emptyset$ for $j = 1, 2$, we have $Y_1 = \emptyset$. There are two initial edges $(g_j, p_{1,j}) \in E_0$ with $V_j \triangleleft \beta \models g_j$. Therefore $\beta \models g_1 \wedge g_2$ and $(g_1 \wedge g_2, (p_{1,1}, p_{1,2}))$ is an initial edge of $\mathcal{A}_1 \parallel \mathcal{A}_2$. From $C_j \triangleleft \gamma_1(c) = 0$ for all $c \in C_j, j = 1, 2$ we have $\gamma_1(c) = 0$ for all $c \in C = C_1 \cup C_2$. From $V_j \triangleleft \beta_i \models s(p_{i,j})$ we can deduce $\beta_i \models s(p_{i,1}) \wedge s(p_{i,2}) = s((p_{i,1}, p_{i,2}))$. Similarly, for the clock invariant $I((p_{i,1}, p_{i,2}))$. For all $i \in 1..n-1$ there are two edge $(p_{i,j}, g_j, X_j, p_{i+1,j}) \in E_j$ for $j = 1, 2$ with

$$\begin{aligned} V_j \triangleleft \beta_i, V_j \triangleleft \beta'_{i+1}, (C_j \triangleleft \gamma_i) + t_i, A_j \cap Y_i \models g_j \\ \text{and } C_j \triangleleft \gamma_{i+1} = ((C_j \triangleleft \gamma_i) + t_i)[X_j := 0]. \end{aligned}$$

So there is an edge $((p_{i,1}, p_{i,2}), g_1 \wedge g_2, X_1 \cup X_2, (p_{i+1,1}, p_{i+1,2})) \in E$. Since C_1 and C_2 are disjoint, we have $\gamma_i + t_i = ((C_1 \triangleleft \gamma_i) + t_i) \cup ((C_2 \triangleleft \gamma_i) + t_i)$ and $\gamma_i + t_i[X_1 \cup X_2 := 0] = ((C_1 \triangleleft \gamma_i) + t_i)[X_1 := 0] \cup ((C_2 \triangleleft \gamma_i) + t_i)[X_2 := 0]$. Hence,

$$\beta_i, \beta'_{i+1}, \gamma_i + t_i, Y_i \models g_1 \wedge g_2 \text{ and } \gamma_{i+1} = (\gamma_i + t_i)[X_1 \cup X_2 := 0].$$

□

If we have two runs run_1 and run_2 of different automata that are supposed to run in parallel, it is useful to insert “stuttering” steps into each run until the steps are synchronous. The following lemma shows that inserting stuttering steps is possible.

Definition 4.9. A run run' is a *stuttering extension* of run if it emerges from run by inserting a finite number of stuttering steps, i. e., replacing a

step $(p_i, Y_i, \beta_i, \gamma_i, t_i)$ by two steps $(p_i, Y_i, \beta_i, \gamma_i, t), (p_i, \emptyset, \beta_i, \gamma_i + t, t_i - t)$, for some t with $0 < t < t_i$.

Lemma 4.10. If \mathcal{A} is a stuttering invariant automaton and run' a stuttering extension of run , then run is a run of \mathcal{A} if and only if run' is a run of \mathcal{A} .

Proof. If run' is a run of \mathcal{A} , deleting a stuttering step from run' produces another run of \mathcal{A} . Deleting a stuttering step means replacing two configurations $(p, Y, \beta, \gamma, t_1), (p, \emptyset, \beta, \gamma + t_1, t_2)$ by a single configuration $(p, Y, \beta, \gamma, t_1 + t_2)$. The new run has only fewer conditions in definition 4.7 that have to hold. By deleting a finite number of stuttering steps we arrive at run , so this is also a run of \mathcal{A} .

For the other direction we need to show that inserting one stuttering step into run produces a new run of \mathcal{A} . So assume that run' is derived from run by inserting a single stuttering step:

$$run' = \langle \dots, (p_i, Y_i, \beta_i, \gamma_i, t), (p_i, \emptyset, \beta_i, \gamma_i + t, t_i - t), \dots \rangle \text{ with } 0 < t < t_i$$

Almost all conditions for run' of definition 4.7 are the same as those for run , because $\gamma_i + t + (t_i - t) = \gamma_i + t_i$. Only the following new conditions need to be shown:

$$t > 0 \text{ and } t_i - t > 0 \tag{4.4}$$

$$\gamma_i + t \models I(p) \tag{4.5}$$

$$\beta_i, \beta'_i, \gamma_i + t, \emptyset \models g \text{ and } \gamma_i + t = (\gamma_i + t)[X := 0] \tag{4.6}$$

for some edge $(p_i, g, X, p_i) \in E$

From $0 < t < t_i$, (4.4) follows immediately. Formula (4.5) follows from $\gamma_i + t_i \models I(p)$ and lemma 4.2. In fact the lemma gives us even $\gamma_i + t \models \text{strict}(I(p))$.

Finally, we need to show that there is some edge $(p_i, g, X, p_i) \in E$ satisfying (4.6). From $\beta_i \models s(p_i)$ and $\gamma_i + t \models \text{strict}(I(p_i))$, we can deduce that for the stuttering edge (p_i, g, \emptyset, p_i) the last condition $\beta_i, \beta'_i, \gamma_i + t, \emptyset \models g$ holds. \square

To compare phase event automaton with Duration Calculus formulae, we can relate runs with DC interpretations.

Definition 4.11. A run

$$\langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots, (p_n, Y_n, \beta_n, \gamma_n, t_n) \rangle$$

matches an interpretation \mathcal{I} , if and only if:

1. $\mathcal{I}(v)(t) = \beta_i(v)$ for $v \in V$, $i \in 1..n$, and almost all $t \in \text{Int}_i$,
2. $\mathcal{I}(e)(t) = \mathcal{I}(e)(t')$ for $e \in A$, $i \in 1..n$, and almost all $t, t' \in \text{Int}_i$
3. $\mathcal{I}(e)(t) = (\text{if } e \in Y_i \text{ then } \neg \mathcal{I}(e)(t') \text{ else } \mathcal{I}(e)(t'))$ for all $e \in A$, $i \in 2..n$, and almost all $t' \in \text{Int}_{i-1}$, $t \in \text{Int}_i$.

Here $\text{Int}_i := [\sum_{j=1}^{i-1} t_j, \sum_{j=1}^i t_j]$ denotes the interval corresponding to the i -th entry of the run.

The first condition asserts that \mathcal{I} and β_i coincide for all observables v and almost all points t in the time interval to which β_i belongs. The second condition asserts, that an observable associated with an event e does not change during some time interval; its valuation may only change between two intervals and it must do so according to the third condition, i. e., it may only change if and only if it is in the corresponding event set Y_{i+1} .

Lemma 4.12. For any run of an automaton \mathcal{A} there is an interpretation \mathcal{I} such that the run matches \mathcal{I} .

Proof. Given a run $run = \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots \rangle$ define $\mathcal{I}(v)(t) = \beta_i$ for all $t \in [\sum_{j=1}^{i-1} t_j, \sum_{j=1}^i t_j[$. Furthermore, define $\mathcal{I}(e)(t) = \mathbf{false}$ for $t \in [0, t_1[$ and $\mathcal{I}(e)(t) = \text{if } e \in Y_i \text{ then } \neg \mathcal{I}(e)(\sum_{j=1}^{i-1} t_j) \text{ else } \mathcal{I}(e)(\sum_{j=1}^{i-1} t_j)$ for $t \in [\sum_{j=1}^i t_j, \sum_{j=1}^{i+1} t_j[$. This defines \mathcal{I} for every $v \in V$ and $e \in A$ and every $t \in [0, \sum_{j=1}^{\#r} t_j[$. It is easy to see that all conditions of definition 4.11 are satisfied. The value of \mathcal{I} for $t > \sum_{j=1}^{\#r} t_j$ can be defined arbitrarily. \square

However, for some interpretation \mathcal{I} there is not always a matching run of \mathcal{A} . We can define the accepted language of an automaton as the set of interpretations that have a matching run.

Definition 4.13. An automaton \mathcal{A} accepts an interpretation \mathcal{I} for duration T ($\mathcal{I}, [0, T] \models \mathcal{A}$), if there is a run of \mathcal{A} ,

$$run = \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots, (p_n, Y_n, \beta_n, \gamma_n, t_n) \rangle,$$

with duration $T = \sum_{i=0}^n t_n$ and \mathcal{I} matches *run*. An automaton *accepts* \mathcal{I} *completely*, if there is an infinite sequence

$$run_\infty = \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots \rangle$$

that is non-Zeno, i. e., $\sum_{i=0}^\infty t_n = \infty$, such that each finite prefix of run_∞ matches \mathcal{I} and is a run of \mathcal{A} .

The following lemma relates parallel composition of phase event automata to conjunction in Duration Calculus. This is an important property of phase event automata. It allows to translate the DC formulae of a CSP-OZ-DC specification separately into phase event automata and combine the automata by parallel composition. The lemma states that the parallel product of two automata accepts an interpretation if and only if both automata accept the interpretation.

Lemma 4.14. *The product automaton $\mathcal{A}_1 \parallel \mathcal{A}_2$ of two stuttering invariant automata \mathcal{A}_1 and \mathcal{A}_2 accepts an interpretation \mathcal{I} for duration T if and only if each component \mathcal{A}_i accepts \mathcal{I} for duration T :*

$$\mathcal{I}, [0, T] \models \mathcal{A}_1 \parallel \mathcal{A}_2 \text{ iff } \mathcal{I}, [0, T] \models \mathcal{A}_1 \text{ and } \mathcal{I}, [0, T] \models \mathcal{A}_2$$

Proof. If $\mathcal{A}_1 \parallel \mathcal{A}_2$ accepts \mathcal{I} for duration T , there is a run

$$run = \langle ((p_{1,1}, p_{1,2}), Y_1, \beta_1, \gamma_1, t_1), \dots \rangle$$

of $\mathcal{A}_1 \parallel \mathcal{A}_2$ with duration T that matches \mathcal{I} . By lemma 4.8, the projection $\langle (p_{1,j}, A_j \cap Y_1, V_j \triangleleft \beta_1, C_j \triangleleft \gamma_1, t_1), \dots \rangle$ is a run of \mathcal{A}_j ($j = 1, 2$). Obviously, \mathcal{I} also matches these projections.

For the other direction assume \mathcal{I} matches run_1 of \mathcal{A}_1 and run_2 of \mathcal{A}_2 , both of duration T . Without loss of generality one can assume that the phase durations t_i in both runs are identical: Otherwise, one can insert stuttering steps into both runs until this is the case. So we have:

$$\begin{aligned} run_1 &= \langle (p_1^{(1)}, Y_1^{(1)}, \beta_1^{(1)}, \gamma_1^{(1)}, t_1), \dots \rangle \\ run_2 &= \langle (p_1^{(2)}, Y_1^{(2)}, \beta_1^{(2)}, \gamma_1^{(2)}, t_1), \dots \rangle \end{aligned}$$

Since \mathcal{I} matches both runs, the state valuations $\beta_i^{(j)}$ and the event sets $Y_i^{(j)}$ coincide on their common variables. So the valuation $\beta_i = \beta_i^{(1)} \cup \beta_i^{(2)}$ is well defined and $V_j \triangleleft \beta_i = \beta_i^{(j)}$. Likewise for $Y_i = Y_i^{(1)} \cup Y_i^{(2)}$ we

have $Y_i^{(j)} = A_j \cap Y_i$. Since the clock variables of \mathcal{A}_1 and \mathcal{A}_2 are disjoint, $\gamma_i = \gamma_i^{(1)} \cup \gamma_i^{(2)}$ is also well defined. By lemma 4.8

$$run = \langle \langle (p_1^{(1)}, p_1^{(2)}), Y_1, \beta_1, \gamma_1, t_1 \rangle, \dots \rangle$$

is a run of $\mathcal{A}_1 \parallel \mathcal{A}_2$ of duration T and \mathcal{I} matches run . So $\mathcal{I}, [0, T] \models \mathcal{A}_1 \parallel \mathcal{A}_2$
 \square

The last lemma has some important consequences. For example, if for some component of a large system all accepted interpretations satisfy a formula F , the interpretations of the complete system also satisfy the formula F . Thus, for a large system it suffices showing that some component ensures a property. This property will hold for the complete system.

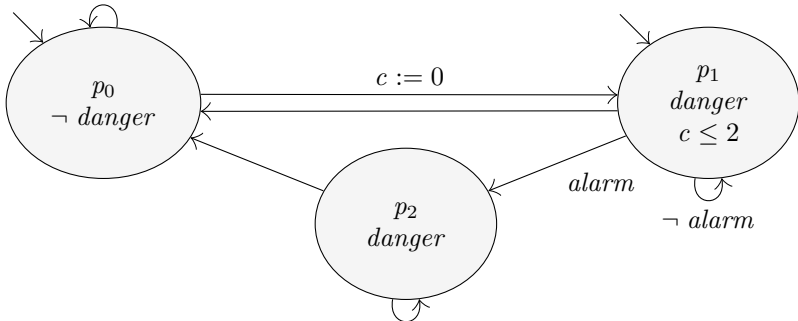
4.4. Automata and Formulae

A phase event automaton can be used to implement a Duration Calculus formula. The semantics of a formula is given by the set of interpretations for which it holds. Likewise an automaton \mathcal{A} is related to the set of interpretations \mathcal{I} which it accepts ($\mathcal{I} \models \mathcal{A}$). If these sets of interpretations coincide then \mathcal{A} implements the formula.

Definition 4.15. A phase event automaton \mathcal{A} *implements* a Duration Calculus formula F (denoted by $\mathcal{A} \hat{=} F$), if for all interpretations \mathcal{I} the following equivalence holds:

$$\mathcal{I} \models \mathcal{A} \Leftrightarrow \mathcal{I} \models F.$$

Example 4.16. The following watchdog automaton \mathcal{A}_w (c.f. figure 4.1)



implements the formula

$$F_w = \neg (((\mathbf{true} \wedge [\neg \text{danger}]) \vee \ell = 0) \wedge [\text{danger}] \wedge \exists \text{alarm} \wedge \ell > 2)$$

Proof. Assume that $\mathcal{I}, [0, t] \models \neg F_w$ for some t . Then there is a t_1 such that $t_1 = 0$ or $\mathcal{I}, [0, t_1] \models (\mathbf{true} \wedge [\neg \text{danger}])$ and $\mathcal{I}, [t_1, t] \models [\text{danger}] \wedge \exists \text{alarm} \wedge \ell > 2$. If $t_1 = 0$ then \mathcal{A}_w will start in p_1 (the only initial location with state invariant danger) with clock $c = 0$. If $\mathcal{I}, [0, t_1] \models \mathbf{true} \wedge [\neg \text{danger}]$ holds, the automaton will switch at time t_1 to location p_1 and reset c to zero. It cannot leave location p_1 as $[\text{danger}] \wedge \exists \text{alarm}$ holds. On the other hand, the clock invariant forbids to stay there for $t - t_1 > 2$. Hence, the automaton does not accept \mathcal{I} . Thus, $\mathcal{I} \not\models F_w$ implies $\mathcal{I} \not\models \mathcal{A}_w$.

Now, assume $\mathcal{I} \not\models \mathcal{A}_w$. This can only happen if location p_1 is active, $c = 2$, danger holds and alarm does not occur because in any other case there is a successor location that can be entered. Let t denote the current time and t_1 denote the time when location p_1 was entered. Since c is reset when entering p_1 , this location was active for exactly two time units. We therefore have $\mathcal{I}, [t_1, t] \models [\text{danger}] \wedge \exists \text{alarm} \wedge \ell = 2$. By assumption, p_1 cannot be left. Hence, there is some $\varepsilon > 0$ such that $\mathcal{I}, [t_1, t + \varepsilon] \models [\text{danger}] \wedge \exists \text{alarm} \wedge \ell > 2$. Location p_1 has only one incoming transition. Thus, either the automaton took the transition from location p_0 to p_1 at the moment t_1 , or it started in p_1 and $t_1 = 0$. Therefore, $\mathcal{I}, [0, t_1] \models (\mathbf{true} \wedge [\neg \text{danger}]) \vee \ell = 0$. Hence in both cases, $\mathcal{I}, [0, t + \varepsilon] \models \neg F_w$. Thus, $\mathcal{I} \not\models \mathcal{A}_w$ implies $\mathcal{I} \not\models F_w$.

Thus, the automaton \mathcal{A}_w implements F_w . \square

If automata $\mathcal{A}_1, \mathcal{A}_2$ implementing formulae F_1, F_2 are given, the automata implementing $F_1 \wedge F_2$ and $F_1 \vee F_2$ can be constructed. For conjunction the parallel composition of automata is used. For $F_1 \vee F_2$ the following construction is used:

Definition 4.17. The *disjunction* $\mathcal{A}_1 \cup \mathcal{A}_2$ of two automata \mathcal{A}_1 and \mathcal{A}_2 , $\mathcal{A}_i = (P_i, V_i, A_i, C_i, E_i, s_i, I_i, E_{0,i})$ is the automaton

$$\mathcal{A} = (P, V, A, C, E, s, I, E_0)$$

defined as follows:

- $P := P_1 \dot{\cup} P_2$. The locations are the disjoint union of P_1 and P_2 ; locations that appear in both automata need to be renamed.

- $V := V_1 \cup V_2$ and $A := A_1 \cup A_2$.
- $C := C_1 \dot{\cup} C_2$.
- $E := E_1 \cup E_2$ and $E_0 := E_{0,1} \cup E_{0,2}$.
- $s := s_1 \cup s_2$ and $I := I_1 \cup I_2$.

Theorem 4.18. *Let \mathcal{A}_1 and \mathcal{A}_2 be phase event automata and F_1, F_2 Duration Calculus formulae. If $\mathcal{A}_1 \hat{=} F_1$ and $\mathcal{A}_2 \hat{=} F_2$ then $\mathcal{A}_1 \cup \mathcal{A}_2 \hat{=} F_1 \vee F_2$*

Proof. Let \mathcal{I} be some interpretation with $\mathcal{I} \models F_1 \vee F_2$. Then either $\mathcal{I} \models F_1$ or $\mathcal{I} \models F_2$ hold. If $\mathcal{I} \models F_1$ then $\mathcal{A}_1 \models \mathcal{I}$. Hence, there is a run *run* of \mathcal{A}_1 that matches \mathcal{I} . Since \mathcal{A} contains all locations, edges and clocks of \mathcal{A}_1 , *run* is also a run of \mathcal{A} , hence $\mathcal{A} \models \mathcal{I}$. Analogously, if $\mathcal{I} \models F_2$.

Now assume that $\mathcal{I} \models \mathcal{A}$. Hence, there is a run *run* starting in some location p_0 . This location belongs to one of the automata \mathcal{A}_1 or \mathcal{A}_2 . Assume it belongs to \mathcal{A}_1 . Since the locations of the two automata are disjoint all edges that are taken and all locations in *run* belong to \mathcal{A}_1 . Hence, *run* is a run of \mathcal{A}_1 . Analogously, *run* is a run of \mathcal{A}_2 if p_0 belongs to \mathcal{A}_2 . Thus, either $\mathcal{I} \models \mathcal{A}_1$ or $\mathcal{I} \models \mathcal{A}_2$. From $\mathcal{A}_1 \hat{=} F_1$ and $\mathcal{A}_2 \hat{=} F_2$ we get $\mathcal{I} \models F_1$ or $\mathcal{I} \models F_2$ hence $\mathcal{I} \models F_1 \vee F_2$. \square

Theorem 4.19. *Let \mathcal{A}_1 and \mathcal{A}_2 be phase event automata and F_1, F_2 Duration Calculus formulae. If $\mathcal{A}_1 \hat{=} F_1$ and $\mathcal{A}_2 \hat{=} F_2$ then $\mathcal{A}_1 \parallel \mathcal{A}_2 \hat{=} F_1 \wedge F_2$*

Proof. Let \mathcal{I} be some interpretation with $\mathcal{I} \models F_1 \wedge F_2$. Then \mathcal{I} satisfies F_1 and F_2 . With $\mathcal{A}_1 \hat{=} F_1$ we get $\mathcal{I} \models \mathcal{A}_1$. Analogously, $\mathcal{I} \models \mathcal{A}_2$. With lemma 4.14 hence $\mathcal{I} \models \mathcal{A}_1 \parallel \mathcal{A}_2$.

Now assume $\mathcal{I} \models \mathcal{A}_1 \parallel \mathcal{A}_2$. With lemma 4.14 we get $\mathcal{I} \models \mathcal{A}_1$ and $\mathcal{I} \models \mathcal{A}_2$. Since $\mathcal{A}_1 \hat{=} F_1, \mathcal{A}_2 \hat{=} F_2, \mathcal{I} \models F_1$ and $\mathcal{I} \models F_2$. Hence, $\mathcal{I} \models F_1 \wedge F_2$. \square

4.5. Deterministic Automata

An automaton is deterministic if for every possible behaviour of the environment the automaton has exactly one run modulo insertion and deletion of stuttering steps. Such an automaton accepts every interpretation for any duration. Nonetheless it will be useful for the power set construction in section 5.3, which builds a deterministic automaton from a Duration Calculus formula. By removing certain states, it can be transformed into an automaton that accepts exactly those interpretations that satisfy the

formula. A deterministic automaton can be characterised by the following definition.

Definition 4.20. A stuttering invariant automaton

$$\mathcal{A} = (P, V, A, C, E, s, I, E_0)$$

is *deterministic* if and only if

1. for each valuation $\beta \in \mathbf{Val}(V)$ there is exactly one initial edge $(g, p) \in E_0$ such that $\beta \models s(p)$ and $\beta \models g$,
2. for each location $p : P$ and each pair of valuations $\beta_1, \beta_2 \in \mathbf{Val}(V)$, $Y \in \mathbb{P}A$, $\gamma \in \mathbf{Val}(C)$ there is exactly one edge $(p, g, X, p') \in E$ such that $\beta_1, \beta_2, Y, \gamma \models g$, $\beta_2 \models s(p')$ and $\gamma[X := 0] \models \mathit{strict}(I(p'))$,
3. each clock invariant $I(p)$ does not contain any strict inequality. Only conjunctions of $c \leq t$, where $c \in C$ and $t \in \mathbb{Q}$, are allowed.

The first requirement assures that there is only one initial location for a given valuation. The second requirement assures that for each location p and each pair of valuations there is exactly one transition that can be taken. A transition can only be taken if $\gamma[X := 0] \models \mathit{strict}(I(p'))$ because the new location needs to be active for a non-zero time. The third condition is necessary to assure that a location is left at a well-defined time if no variable changes and no event occurs. Suppose the clock invariant $I(p)$ of a location p contains a strict inequality $c < t$. As long as no event occurs, no variable changes, and the invariant $I(p)$ holds, the state cannot be left because the stuttering edge is the only outgoing edge. However, the clock invariant also forbids staying in this state forever. Hence, there is a deadlock and the automaton does not accept the interpretation. However, if the inequalities in the invariant have the form $c \leq t$, the location can be left when $\gamma(c) = t$. The stuttering edge is not enabled because $\gamma \not\models \mathit{strict}(I(p))$.

Lemma 4.21. *Let \mathcal{A} be a deterministic and stuttering invariant automaton. For any interpretation there is an infinite non-Zeno sequence, such that each prefix is a run of the automaton matching this interpretation. Thus, \mathcal{A} accepts each interpretation \mathcal{I} completely. Furthermore, all other sequences with this property can be derived by inserting or deleting stuttering steps.*

Proof. First, we show that the sequence is *unique*.

Assume that there are two different sequences run_1, run_2 of \mathcal{A} that both match \mathcal{I} . Furthermore, assume that in both runs all stuttering steps are removed. Let i be the number of the first configuration where run_1 and run_2 differ:

$$\begin{aligned} run_1 &= \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots, (p_i^{(1)}, Y_i^{(1)}, \beta_i^{(1)}, \gamma_i^{(1)}, t_i^{(1)}), \dots \rangle \\ run_2 &= \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots, (p_i^{(2)}, Y_i^{(2)}, \beta_i^{(2)}, \gamma_i^{(2)}, t_i^{(2)}), \dots \rangle \end{aligned}$$

Without loss of generality assume that $t_i^{(1)} \leq t_i^{(2)}$. Since both runs match \mathcal{I} , we have $\beta_i^{(1)}(v) = \mathcal{I}(v)(t) = \beta_i^{(2)}(v)$ for all $v \in V$ and almost all $t \in]\sum_{j=1}^{i-1} t_j, \sum_{j=1}^{i-1} t_j + t_i^{(1)}[$, so $\beta_i^{(1)} = \beta_i^{(2)} =: \beta_i$. Similarly, $Y_i^{(1)} = Y_i^{(2)} =: Y_i$. Because \mathcal{A} is deterministic there is only one edge $(p_{i-1}, g, X, p_i) \in E$ such that $\beta_{i-1}, \beta'_i, Y_i, \gamma_{i-1} + t_{i-1} \models g$, $\beta_i \models s(p_i)$ and $\gamma[X := 0] \models \text{strict}(I(p_i))$. Hence, $p_i^{(1)} = p_i = p_i^{(2)}$ and $\gamma_i^{(1)} = (\gamma_{i-1} + t_{i-1})[X := 0] = \gamma_i^{(2)} =: \gamma_i$. Thus, the difference between run_1 and run_2 is in the duration $t_i^{(1)} < t_i^{(2)}$. This leads to a contradiction, though:

Because the invariant $I(p_i)$ holds at $\gamma_i + t_i^{(2)}$ it must hold strictly at $\gamma_i + t_i^{(1)}$. Furthermore, since the interpretation \mathcal{I} does not change at $\sum t_j + t_i^{(1)}$ we have $\beta_{i+1}^{(1)} = \beta_i$ and $Y_{i+1}^{(1)} = \emptyset$. The stuttering edge of \mathcal{A} is enabled. Since \mathcal{A} is deterministic, the stuttering edge is the only enabled edge. Thus, $p_{i+1}^{(1)} = p_i$, although we assumed that all stuttering steps in run_1 were removed. This is a contradiction. Thus, run_1 and run_2 are equal.

Now, we create the infinite sequence for \mathcal{I} .

Due to finite variability there is some $\varepsilon > 0$ such that for all $v \in V$ the interpretation $\mathcal{I}(v)(t)$ is constant for all $t \in]0, \varepsilon[$. The constant value can be expressed as $\lim_{t \searrow 0} \mathcal{I}(v)(t)$. Choose $\beta_1(v) = \lim_{t \searrow 0} \mathcal{I}(v)(t)$ for all $v \in V$ and $Y_1 = \emptyset$. There is one initial edge $(g, p_1) \in E_0$ such that $\beta_1 \models s(p_1)$ and $\beta_1 \models g$. Set $\gamma_1(c) = 0$ for all $c \in C$. Furthermore, choose

$$t_1 = \sup\{t : \text{Time} \mid \gamma_1 + t \models I(p_1) \wedge \mathcal{I} \text{ almost constant on } [0, t]\}.$$

Because $I(p_1)$ contains no strict inequalities, $\gamma_1 + t_1 \models I(p_1)$ holds. Hence, $run_1 := \langle (p_1, \emptyset, \beta_1, \gamma_1, t_1) \rangle$ is a run of duration t_1 for \mathcal{I} .

If there is a run run_{n-1} of length $n - 1$ and duration $T_{n-1} := \sum_{j=1}^{n-1} t_j$ that matches \mathcal{I} , then a run of length n can be constructed as follows: Let $\beta_n(v) = \lim_{t \searrow T_{n-1}} \mathcal{I}(v)(t)$ the valuation of the state variables after

time T_{n-1} and $Y_n := \{e \mid \lim_{t \searrow T_{n-1}} \mathcal{I}(e)(t) \neq \lim_{t \nearrow T_{n-1}} \mathcal{I}(e)(t)\}$ the set of events occurring at T_{n-1} . For $p_{n-1}, \beta_{n-1}, \gamma_{n-1}, t_{n-1}, \beta_n, Y_n$ there is exactly one edge $(p_{n-1}, g, X, p_n) \in E$ such that $\beta_{n-1}, \beta'_n, Y_n, \gamma_{n-1} + t_{n-1} \models g, \beta'_n \models s(p_n)$ and $(\gamma_{n-1} + t_{n-1})[X := 0] \models \text{strict}(I(p_n))$. Set $\gamma_n = (\gamma_{n-1} + t_{n-1})[X := 0]$ and choose

$$t_n = \sup\{t : \text{Time} \mid \gamma_n + t \models I(p_n) \wedge \mathcal{I} \text{ almost constant on } [T_{n-1}, t]\}.$$

Since $\gamma_n \models \text{strict}(I(p_n))$ and due to the finite variability of \mathcal{I} we have $t_n > 0$. Also $\gamma_n + t_n \models I(p_n)$ because $I(p_n)$ contains no strict inequalities. Therefore, $\text{run}_n := \text{run}_{n-1} \hat{\ } \langle (p_n, Y_n, \beta_n, \gamma_n, t_n) \rangle$ is a run of length n that matches \mathcal{I} .

What remains to be shown is that this leads to a non-Zeno infinite sequence. We need to show that for each T we get a run of duration $t > T$ by the preceding algorithm. Assume, this is not the case and $\sum_{j=1}^n t_j < T$ for all n . Since all t_j are positive and the sum is bounded it has a limit $T' := \sum_{j=1}^{\infty} t_j$. Due to the finite variability of \mathcal{I} the interpretation is constant on some interval $]T' - \varepsilon, T'[\$ for some $\varepsilon > 0$. There is some n_0 such that $\sum_{j=1}^{n_0} t_j > T' - \varepsilon$. So for all $n > n_0$ we have $\beta_n = \beta_{n_0}, Y_n = \emptyset$ and $t_n = \sup\{t : \text{Time} \mid \gamma_n + t \models I(p_n)\}$. The latter means that $\gamma_n(c_i) + t_n = \delta_i$ for some constraint $c_i \leq \delta_i$ in $I(p_n)$. We call this constraint *responsible* for t_n . There are only finitely many constraints $c_i \leq \delta_i$ in the automaton. Each of these constraint can be responsible for at most ε/δ_i different t_n because the clock c_i needs to be reset and run for at least δ_i before the constraint is responsible again. However, there are infinitely many t_n . This is a contradiction, so the assumption $\sum_{j=1}^{\infty} t_j < T$ was wrong and for every T there is an n such that $\sum_i = 0t_n > T$. Thus, the infinite run is non-Zeno. \square

4.6. Case Study: Audio Control Protocol

Modern hi-fi systems consist of several components connected by a simple control bus. The purpose of this bus is to build a tiny local network. This way the customer can operate the system with a single remote control and can wake-up all components with a single button press.

This tiny network uses a simple protocol, so that almost no additional costs arise when building the hardware. Philips used a bus consisting of a single wire. On this wire a Manchester encoding is used to transmit data, cf. figure 4.2. Every bit is encoded in a bit slot of equal length. For a one

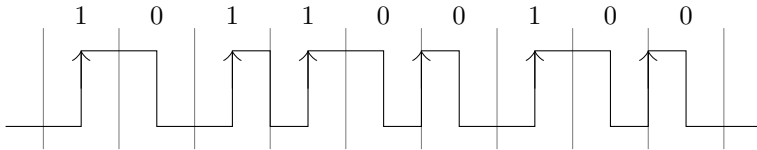


Figure 4.2.: Manchester coding of ‘101100100’

bit the wire is kept low on the first half and high on the second half and for a zero bit it is the other way round.

One problem in deciphering the encoding is that for hardware reasons the receiver can only sense the rising edges. These edges are marked with an arrow in the diagram. Another problem is that the timing of sender and receiver can vary by up to 5 % because of cheap hardware. It was shown in [BPV94] and [BGK⁺96] that despite these problems the protocol is correct.

Two different codes are separated by keeping the wire low for a certain time (the so called radio silence). It is $8ms$ in the original specification but we can shorten it to a single bit slot. Since it is impossible for the receiver to distinguish if a code starts with zero or one, the audio protocol specifies that it must always start with a one. Likewise it is not possible to detect a final zero that follows a one bit as there is no rising edge. The audio protocol specifies that every string must either end in two zeros, which can be detected, or have odd length. However, here we use a much simpler condition, stating that every packet must end with a zero bit. There is a simple bijective mapping between the original codes and the simplified ones.

An implementation of this protocol in the form of two parallel phase event automata is given in figure 4.3. There is one state variable *wire* representing the state of the wire. It can have the value *high* or *low*. The sender and receiver automata synchronise on this state variable. Each location has a state invariant restricting *wire* to a single value. In the figure the invariant $wire = high$ is encoded by a square node, and $wire = low$ by a round node. The stuttering edges are removed in this diagram for sake of readability. Each location of the sender automaton has a stuttering edge labelled with $\neg s.0 \wedge \neg s.1 \wedge \neg s.stop$. Thus, it forbids all send events. Likewise, the stuttering edge of each location of the receiver automaton forbids all receive events. All time constants are in multiples of Q , which

is a nominal duration of quarter of a bit slot. To represent the timing uncertainty we introduce the constants $Q_l := Q(1 - \delta)$ and $Q_h := Q(1 + \delta)$ where $\delta > 0$ is the maximum timing drift. So Q_l is the minimum duration and Q_h the maximum duration of a quarter of a bit slot if the sender's clock drift is bound by δ .

The sender automaton consists of six locations. Location 1 is the idle location, where it waits for the environment to send a message. The message is sent via the channel s that has three events: $s.1$ to send a one bit, $s.0$ to send a zero bit and $s.stop$ to mark the end of the message. Because the message must start with a one bit only $s.1$ is allowed from the idle location.

Locations 4 and 5 send a one bit on the wire. As depicted in figure 4.2 a one bit is sent by keeping the wire *low* for half a slot and *high* for the second half. The time is measured by the guard $2Q_l < x < 2Q_h$. In location 5 the sender checks the next bit to send. The events $s.0$ or $s.1$ have to be sent by the environment at the right time. The automaton will then change to 4 to send another one bit or to location 2 to send a zero bit.

Locations 2 and 3 are constructed analogously to the locations 4 and 5. The main difference is that after a zero bit the message can be terminated by a $s.stop$ event. In that case, location 6 is entered which will wait for the duration of a bit slot, so that the receiver can detect the end of the message, and then switch to the idle location again.

The receiver automaton is shown on the bottom of figure 4.3. It starts in location **a** which is also the idle location where no message is sent. This location is kept as long as the wire is *low*. When the first rising edge is detected the automaton must change to location **b**. The transition triggers an $r.1$ event that communicates to the environment that a one bit marking the start of the message was received. Also the clock y is reset. It measures the time between the rising edges.

On the falling edge the receiver changes to location **c** and no further action is taken. This is because the specification required that only the rising edges can be timed. Now there are four possibilities depicted in the first row in figure 4.4. The first case is that another one bit follows. In this case, the next rising edge follows after one bit slot ($4Q$) and the receiver changes from location **c** to location **b** because $y < 5Q$ holds. The timing uncertainties are again modelled using Q_l and Q_h .

In all other cases, there is a zero bit after the one bit which can be detected because there is no rising edge for at least $6Q$ time units. In this

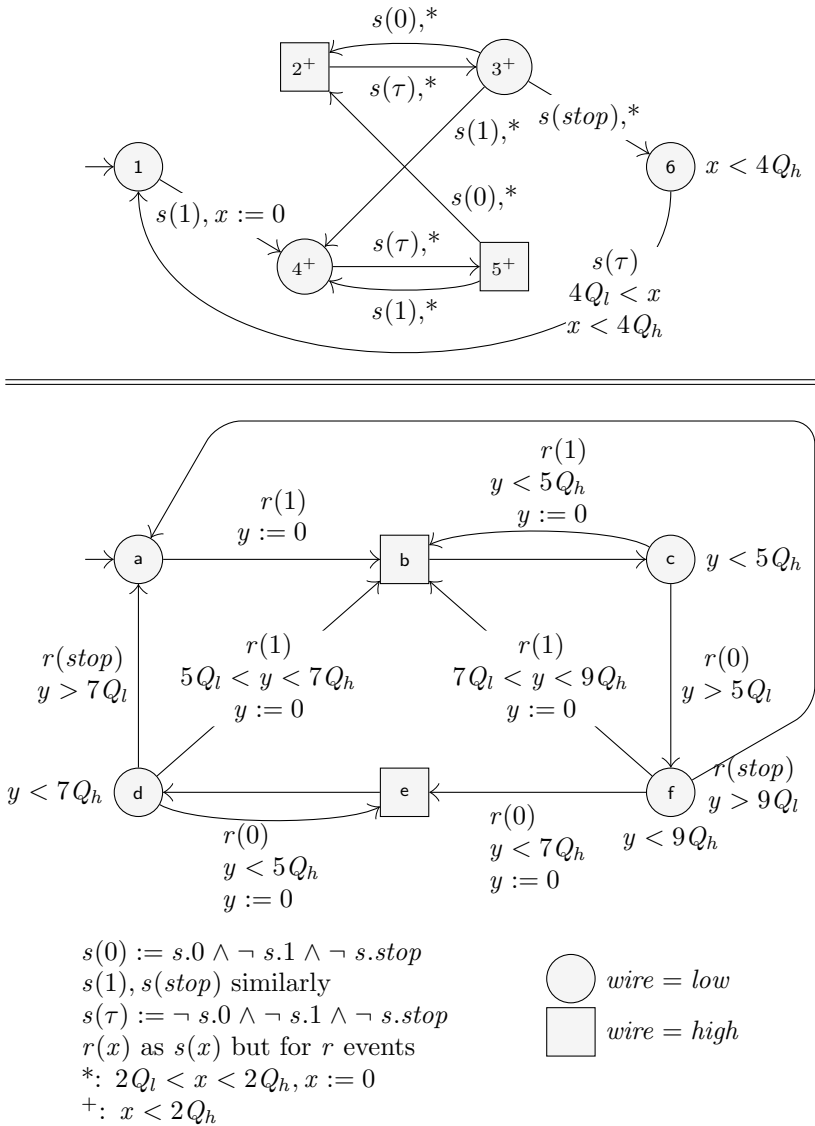


Figure 4.3.: The sender (top) and receiver (bottom) automata

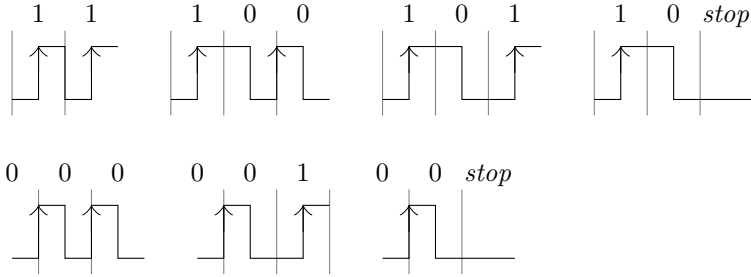


Figure 4.4.: Gaps between two rising edges in Manchester encoding

case $y > 5Q$ holds and the receiver sends an $r.0$ event and changes to location f . Now there are three possibilities. If a second zero bit follows, there is a rising edge between these bits after $6Q$ time units. In this case, the receiver changes to location e and sends another $r.0$ event to the environment. If a one bit follows the zero bit, the rising edge is detected after $8Q$ time units and the receiver changes back to location b . Finally, when no rising edge is received within $9Q$ time units the receiver detects the end of a message, sends an $r.stop$ event, and switches to the idle location a .

The location e is entered when the rising edge between two adjacent zero bits is detected. In this location, it switches to d when the wire becomes low. Then there are three possibilities, depicted in the second row in figure 4.4. If a third zero bit follows, the rising edge will occur after $4Q$ time units and location e is entered again. If a one bit follows, there is a gap of $6Q$ time units and location b is entered. If no rising edge is detected, the receiver changes to the idle location a after $7Q$ time units.

The correctness of the implementation is proven in section 6.3. We will also give the maximum timing uncertainty δ under which the protocol remains correct. We show that each event $s.x$ of the sender is eventually followed by the corresponding event $r.x$ of the receiver.

4.7. Discussion and Related Work

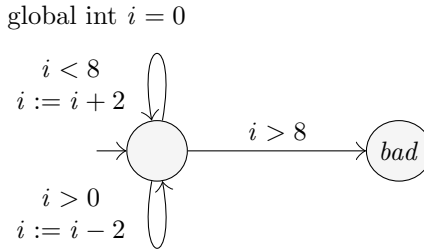
4.7.1. Discussion

The purpose of phase event automata is to define an operational semantics for CSP-OZ-DC. Its parallel composition allows us to describe the seman-

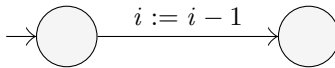
tics for multiple classes running in parallel, but it will also be used in the next chapter to combine the three parts CSP, Object-Z and Duration Calculus. An important property of the parallel composition is theorem 4.19, which shows that parallel composition corresponds to logical conjunction. This theorem shows that it is possible to express the Duration Calculus part as separate automata running in parallel.

The parallel composition also facilitates modular verification. If a property only depends on a small part of a complex system it is much easier to show this for this small part only. The behaviour of the complete system is always the intersection of the behaviour of each of its part as lemma 4.14 shows. So any property that is satisfied by all interpretations of some part, also holds for the interpretations of the complete system.

Not all automata models enable such a modular verification. For example, the timed automata model of Uppaal [YPD94] supports global variables. These global variables can be changed by any component. For example, consider the following automaton using a global variable i .



It can be easily verified with Uppaal that the location labelled *bad* is not reachable because i is never increased beyond 8. However, if we add the following automaton in parallel, the bad state of the first automaton suddenly becomes reachable, even though the second automaton only decreases i .



If the first automaton runs alone, the bad state is not reachable, because the value of i is always even and at most to 8. If the second automaton runs in parallel i is assigned an odd value. If $i = 7$, it can be increased by the first automaton to $i = 8$ and the bad state can be entered. Thus with assignment to global variables, modular reasoning becomes much more difficult. One has to consider interferences with global invariants that are not explicitly present in the automaton; the invariant is only in its

correctness proof.

4.7.2. Other Timed Automata Models

The work on timed automata goes back to Alur and Dill [AD94]. Their automaton model also uses the same synchronisation on events as our automaton model but only allows one event in each transition. A trace of an automaton only records the events and their time stamps. There is no notion of state as in our automaton model.

The model of Timed I/O Automata [LV93, KLSV06], which is a restricted version of Hybrid I/O Automata [LSVW96], has both events and state variables. The event synchronisation is as in CSP. The state variables are divided into input, output, and local variables. The output and local variables can only be changed by visible events; input variables can also be changed by (internal) τ events. On the other hand, the automaton may not make any restriction on the new values of the input variables. The parallel composition of Timed I/O Automata is similar to parallel composition of Phase Event Automata: the common state variables of both automata need to agree on their values, and common events have to occur simultaneously.

However, the distinction of input and output variables is too strict for our purpose. In a CSP-OZ-DC specification the DC part does not determine the values for the state variables (it is the Z part that does this) but it forbids those traces of the output variables that correspond to the forbidden behaviour. Thus it cannot be represented by an input enabled Timed I/O Automata that runs in parallel with the Z automaton.

In [CCO⁺04], Clarke proposes a model checker for state/event based systems. He uses labelled Kripke structures (LKS) where locations are labelled by state propositions and edges are labelled by a non-empty set of events. The model does not allow multiple events to occur at the same time; instead exactly one event from the set must occur. If no events occur the location of the automaton may not be changed. The parallel composition synchronises on common events; components that do not participate on a communication keep their current location. This model has no real-time and is thus not suitable for CSP-OZ-DC. Also the states are only modelled by atomic proposition. Complex data types are not supported.

A tool to analyse timed automata is Uppaal [BDL04, YPD94], a sophisticated model checker developed jointly by Aalborg University in Denmark and Uppsala University in Sweden. Unfortunately, their timed automata

model uses a different type of synchronisation similar to CCS. Each event occurs in two flavours, either it is decorated with ? (input) or with ! (output). Exactly two automata synchronise, when one automaton sends the input event and the other the output event. There is also the notion of broadcasting, where a sender automaton can reach multiple receiving automata. However, the semantics are not as in CSP where all processes have to agree. With broadcasting a sending process can send an event to some receiving process if other receiving processes are not ready to receive the event.

The model checker Kronos [Yov97] uses a similar synchronisation as in our model: both automata must agree on their common events. Like in our model, there may be more than one event in one transition. There is no support for data variables, though.

The model of Statecharts [Har87] is widely used in practise. A variation of these are *state diagrams*, which are included in the Unified Modelling Language (UML) [ISO05]. They are popular because they offer a lot of tool support including model checking and code generation. They also have notion of events, data variables and real-time. However, the semantics is complex using asynchronous communication, global variables, and a super-dense time model. Defining a compositional semantics that enables modular verification is difficult [Hel98].

The phase event automata are inspired by phase automata [Tap01]. Tapken used these automata to provide an operational semantics for Duration Calculus formulae. He also designed a model checker for these automata. Because Tapken uses DC without any event extension, his automata have no events. The edges in his model are not labelled, but the locations can be labelled by state invariants. He uses a simple model of clocks: locations can be part of a clock region, which specifies a minimum and maximum duration that this region may be active. Parallel composition of automata is the same as conjuncting the corresponding DC formulae.

Most automata models have a compositional parallel operators allowing modular reasoning for large specifications. However, they usually pay the price of restricting synchronisations to events only. For example in [CCO⁺04] the authors state:

In particular, we forbid the sharing of variables. This restriction facilitates the use of compositional reasoning in verifying specifications.

On the other hand, Statecharts and Uppaal's timed automata allow shar-

ing of global variables. This leads to a non-compositional parallel operator as we explained in section 4.7.1. However, Tapken demonstrated in [Tap01] that sharing state variables and a simple compositional parallel operator are not mutual exclusive properties. We extended his automata model by events to get the advantages of event- and state-based synchronisation together with a compositional parallel operator.

Figure 4.5 summarises the different automata models and compares them by four different properties: synchronisation on events, data, real-time and whether parallel composition has a simple compositional semantics. All automata models except for phase automata have a notion of instantaneous events on which parallel automata synchronise. In phase automata it is only possible to synchronise over state variables. Some other models also allow synchronising over state variables, but this usually results in a non-compositional parallel operator. In Timed I/O Automata this is solved by restricting the usage of state variables: only one automaton can change these variables.

Models	Events	Data	Real-Time	Compositional
Timed Automata [AD94]	✓	-	✓	✓
Timed I/O Automata [LV93]	✓	(✓)	✓	✓
LKS [CCO ⁺ 04]	✓	-	-	✓
Uppaal [YPD94]	✓	✓	✓	-
Kronos [Yov97]	✓	-	✓	✓
Statecharts [Har87]	✓	✓	✓	-
Phase Automata [Tap01]	-	✓	✓	✓
Phase Event Automata	✓	✓	✓	✓

Figure 4.5.: Comparison of timed automata models

5. From CSP-OZ-DC to Phase-Event-Automata

Contents

5.1. Translating CSP	95
5.2. Translating Object-Z	98
5.3. Translating DC	101
5.3.1. Power Set Construction for Counterexamples	103
5.3.2. Creating the Accepting Automaton	135
5.3.3. Case Study: Elevator	138
5.4. Discussion and Related Work	139

In this chapter, we present an operational semantics for CSP-OZ-DC classes by composition of phase event automata. This translation is compositional in the following sense: the CSP part, the Z part, and the DC part are each translated into separate phase event automata. The behaviour of the complete class is the parallel composition of these automata.

5.1. Translating CSP

For a CSP process P , we denote by $\alpha(P)$ the set of channels that this process P uses. This can be computed statically by including all channels that are referenced in the CSP process equations defining P . Then $\alpha(P)$ is also the set of event variables of the phase event automata implementing this process P .

For each channel $c \in \alpha(P)$, there is a corresponding event variable in the automata. If there is data associated with an event, there is a corresponding parameter variable $c\bowtie$ that is used to communicate the data values. For example, from the channel declaration

```
chan request : [x : Z]
```

the event variable $request$ and the state variable $request\bowtie : [x : Z]$ is derived.

The operational semantics of the CSP process is determined according to the previous chapter. This leads to a labelled transition system

$(\mathbb{W}_P, \mathbb{W}_E, Q_0, \longrightarrow)$. This transition system can be simplified by removing the unreachable states or applying reduction techniques such as [Ros94]. We assume further that P is a non-terminating process and thus there is no reachable state with an outgoing \checkmark -transition. The resulting transition system $(Q, \mathbb{W}_E, q_0, \longrightarrow)$ can be translated to a phase event automaton $\mathcal{A} = (Ph, V, A, C, E, s, I, E_0)$ as follows:

- $Ph = Q$. The states of the phase event automata are the same as in the labelled transition system.
- $V = \{c : \alpha(q_0) \bullet c\sharp\}$. The state variables are the parameter variables associated with the channels occurring in the alphabet of the main process.
- $A = \alpha(q_0)$. The alphabet of the phase event automaton is the alphabet of the main process, i. e., the set of channels occurring there.
- $C = \emptyset$. There are no clock variables.
- $E = \{q, q' : Q; \alpha : EVENTS \mid q \xrightarrow{\alpha} q' \bullet (q, \text{only}(\alpha), \emptyset, q')\} \cup E_{st}$ and $E_{st} = \{q : Q \bullet (q, \text{only}(\tau), \emptyset, q)\}$. For each transition in the LTS there is a corresponding edge in the phase event automaton. The formula $\text{only}(\alpha)$ demands that the event α is communicated and no other event. Formula $\text{only}(\tau)$ demands that no event is communicated. For an event $c(v)$ communicating the values v over channel c we demand that the event variable c is the only event variable that is true and that the post state of the parameter variable, namely $c\sharp'$, equals the communicated values v .

$$\begin{aligned} \text{only}(\tau) &= \forall e : A \bullet \neg e \\ \text{only}(c(v)) &= c \wedge c\sharp' = v \wedge \forall e : A \setminus \{c\} \bullet \neg e \end{aligned}$$

Furthermore, the stuttering edges E_{st} are added to E . For each $q \in Q$ there is a simple stuttering edge that allows no communication over the channels of the automaton.

- $s(q) = I(q) = \mathbf{true}$ for all $q \in Q$. The state and clock invariants are always true.
- $E_0 = \{(\mathbf{true}, q_0)\}$, i. e., q_0 is the single starting state.

Theorem 5.1 (Soundness of CSP Translation). *Let*

$$LTS(P) = (Q, \mathbb{W}_E, q_0, \longrightarrow)$$

the simplified labelled transition system of the process P and \mathcal{A} be the automaton with alphabet A constructed as described above. Then

$$\mathcal{I} \models \mathcal{A} \text{ if and only if } Untime(\mathcal{I}) \triangleright A \in \mathcal{T}(LTS(P))$$

Proof. If $\mathcal{I} \models \mathcal{A}$ holds there is a run

$$run = \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots, (p_j, Y_j, \beta_j, \gamma_j, t_j), \dots \rangle$$

that matches \mathcal{I} and is accepted by \mathcal{A} . If we remove all stuttering steps from this run we get another run that matches \mathcal{I} and accepts \mathcal{A} . Hence, we can assume that the run contains no stuttering steps. For each j there is an α_j corresponding to an edge

$$(p_j, \text{only}(\alpha_j), \emptyset, p_{j+1}) \in E \setminus E_{st}$$

such that $\beta_j, \beta'_{j+1}, Y_{j+1} \models \text{only}(\alpha_j)$. Hence,

$$Y_{j+1} = \begin{cases} \{c\} & \text{if } \alpha_j = c(v), \\ \emptyset & \text{if } \alpha_j = \tau \text{ and} \end{cases} \quad (5.1)$$

$$\beta_{j+1}(c\boxtimes) = v \text{ if } \alpha_j = c(v). \quad (5.2)$$

Therefore, the sequence of events of the alphabet A occurring in the untimed behaviour, $Untime(\mathcal{I}) \triangleright A$, is $\langle \alpha_1, \alpha_2, \dots, \alpha_{n-1} \rangle \setminus \{\tau\}$. The edge $(p_j, \text{only}(\alpha_j), \emptyset, p_{j+1})$ corresponds to the relation $p_j \xrightarrow{\alpha_j} p_{j+1}$. This yields the trace

$$p_1 \xrightarrow{\alpha_1} p_1 \xrightarrow{\alpha_2} p_2 \dots$$

For the initial state p_1 there is an edge in E_0 , which means $p_1 = q_0$. Hence, $Untime(\mathcal{I}) \triangleright A \in \mathcal{T}(LTS(P))$.

If $Untime(\mathcal{I}) \triangleright A \in \mathcal{T}(LTS(P))$, there is a transition sequence

$$q_0 = p_1 \xrightarrow{\alpha_1} p_1 \xrightarrow{\alpha_2} p_2 \dots$$

with $Untime(\mathcal{I}) \triangleright A = \langle \alpha_1, \alpha_2, \dots \rangle \setminus \{\tau\}$. This sequence corresponds to the run

$$run = \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots \rangle$$

of \mathcal{A} with Y_i and β_i defined according to (5.1) and (5.2) and $\gamma_i = \emptyset$. If the times t_i are set according to the point in time when α_i occurs, this run matches the interpretation \mathcal{I} . Hence, \mathcal{I} is accepted by the automaton. \square

Case Study

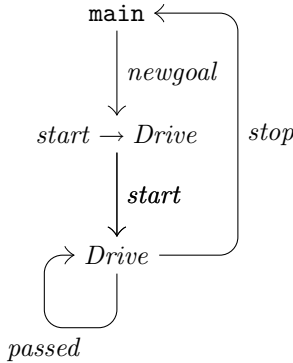
We apply the above algorithm on the case study for the elevator introduced in section 3.2.2. The CSP part is as follows

$$\begin{aligned} \mathbf{main} &\stackrel{c}{=} \mathit{newgoal} \rightarrow \mathit{start} \rightarrow \mathit{Drive} \\ \mathit{Drive} &\stackrel{c}{=} (\mathit{passed} \rightarrow \mathit{Drive}) \square (\mathit{stop} \rightarrow \mathbf{main}) \end{aligned}$$

The alphabet of the CSP process \mathbf{main} is

$$\alpha(\mathbf{main}) = \{\mathit{newgoal}, \mathit{start}, \mathit{passed}, \mathit{stop}\}.$$

These are all signal channels, i. e., there are no parameters associated with them, hence there is no need to include the parameter channels. Constructing the labelled transition system and reducing it with the technique of [Ros94] results in the following transition system.



The CSP process is translated to the phase event automata $\mathcal{A}(CSP) = (P, V, A, C, E, s, I, \mathit{Init})$, with $P = \{\mathbf{main}, \mathit{start} \rightarrow \mathit{Drive}, \mathit{Drive}\}$, $V = \emptyset$, $A = \{\mathit{newgoal}, \mathit{start}, \mathit{passed}, \mathit{stop}\}$, $C = \emptyset$, $s(p) = I(p) = \mathbf{true}$ and E, Init as depicted in figure 5.1.

5.2. Translating Object-Z

The Object-Z part is translated into an automaton with a single state. The variables of the automaton are the variables $\mathit{Var}(\mathit{State})$ declared in the state schema of the CSP-OZ-DC class. Additionally, it contains the

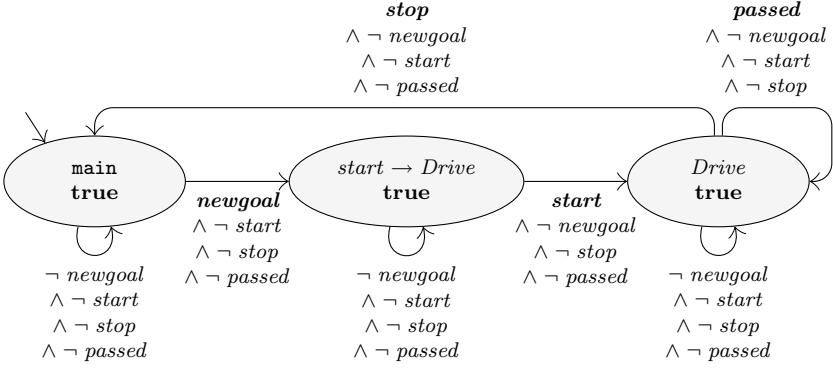


Figure 5.1.: Translation of CSP part of elevator

parameter variables $c\bowtie$ for the channels c for which it provides operation schemas. The communication alphabet A of this automaton consists of all communication channels c for which the class contains an operation schema com_c . For each communication event the automaton contains an edge looping from the single state into itself. The guard of this edge demands that only the corresponding event occurs and that the pre and post state relate according to the communication schema. If the schema has parameters $\text{in?}, \text{out!}$, they are assigned by a let construct to the values communicated by $c\bowtie$, e. g., $\text{let } \text{in?} == c\bowtie.in; \text{out!} == c\bowtie.out \bullet \text{com}_c$. Furthermore, there is the stuttering edge that demands that all state variables do not change and no communication in the alphabet occurs. Finally, there is one initial edge to the single state that has the *Init* schema of the CSP-OZ-DC class as guard. Thus, the resulting automaton is as follows:

$$\begin{aligned}
 \mathcal{A} &= (P, V, A, C, E, s, I, E_0) \\
 P &= \{p_0\} \\
 V &= \text{Var}(\text{State}) \cup \{c : A \bullet c\bowtie\} \\
 A &= \{\text{set of channels with communication schemata}\} \\
 C &= \emptyset \\
 E &= \{c : A \bullet (p_0, \text{only}(c) \wedge \text{let } \text{param} == c\bowtie.param \bullet \text{com}_c, \emptyset, p_0)\} \\
 &\quad \cup \{(p_0, \text{only}(\tau) \wedge \exists \text{State}, \emptyset, p_0)\}
 \end{aligned}$$

$s(p_0) = \textit{State}$ (invariant from the state schema)

$I(p_0) = \mathbf{true}$

$E_0 = \{(Init, p_0)\}$

Theorem 5.2 (Soundness of Z Translation). *Let \mathcal{A} be the automaton constructed as described above. Then $\mathcal{I} \models \mathcal{A}$ if and only if \mathcal{I} is accepted by the Z part, which means (c.f. page 58) that for*

$$Untime(\mathcal{I}) = \langle M_0, c_1(M_1(c_1\boxtimes)), M_1, \dots \rangle :$$

- $M_0 \in \llbracket Init \rrbracket^{\mathcal{P}}$,
- $M_i \in \llbracket State \rrbracket^{\mathcal{P}}$ for all i ,
- $(M_{i-1} \cup M'_i \cup M_i(c_i\boxtimes)) \in \llbracket com_c_i \rrbracket^{\mathcal{P}}$,

Proof. The formula $\mathcal{I} \models \mathcal{A}$ holds if and only if there is a run

$$run = \langle (p_0, Y_1, \beta_1, \emptyset, t_1), \dots \rangle$$

that matches \mathcal{I} and is accepted by \mathcal{A} . By definition 4.7, run is a run of \mathcal{A} if and only if

- for the initial edge $(Init, p_0)$, $\beta_1 \models Init$ holds,
- for each β_j the state invariant $State$ holds,
- for each $\beta_j, \beta_{j+1}, Y_{j+1}$ there is an edge

$$(p_0, \textit{only}(c) \wedge \textit{let param} == c\boxtimes'.param \bullet \textit{com_c}, \emptyset, p_0) \in E$$

such that

$$\beta_i, \beta'_{i+1}, Y_{i+1} \models \textit{only}(c) \wedge \textit{let param} == c\boxtimes'.param \bullet \textit{com_c}, \emptyset, p_0).$$

For the untimed run $Untime(\mathcal{I}) = \langle M_0, c_1(M_1(c_1\boxtimes)), M_1, \dots \rangle$ we have $\beta_{j+1} = M_j$ and $Y_{j+1} = \{c_j\}$ for all j . Hence the three conditions above hold if and only if

- $M_0 \in \llbracket Init \rrbracket^{\mathcal{P}}$,
- $M_i \in \llbracket State \rrbracket^{\mathcal{P}}$ for all i ,
- $(M_{i-1} \cup M'_i \cup M_i(c_i\boxtimes)) \in \llbracket com_c_i \rrbracket^{\mathcal{P}}$.

□

Case Study

We apply the translation of the Z part on the case study for the elevator introduced in section 3.2.2. There are four channels with operation schemas. Therefore, the alphabet is $A = \{newgoal, start, stop, passed\}$. As for the CSP part there are no parameter variables as all channels are signals. The state variables are $V = \{current, goal, start\}$, the variables occurring in the state schema. The automaton is given in figure 5.2. The main state has five loop edges. The top most is the stuttering edge, the other edges correspond to the four operation schemas.

5.3. Translating DC

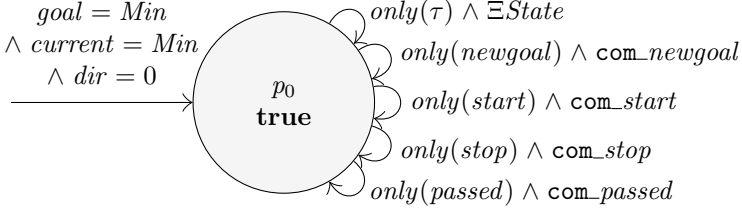
The formulae occurring in the DC part of a CSP-OZ-DC class are of the kind $\neg (phase_1 \wedge \dots \wedge phase_n)$. Each DC formula is translated into a separate phase event automaton. The basic idea is to build a deterministic automaton that reaches certain states if and only if it observed $phase_1 \wedge \dots \wedge phase_n$. If these states are then removed from the automaton it forbids exactly those runs that would violate the formula.

For the constructs occurring in a trace specification ($[P], \ell \sim k, F \wedge G, F \wedge G$) it is possible to build a *non-deterministic* phase event automaton that can reach a certain location if the interpretation for an observed run satisfies the formula. However, a deterministic automaton may not always exist. As an example consider the formula (2.2) on page 23, which is given here again:

$$\neg (\mathbf{true} \wedge \uparrow ev \wedge \ell = 1 \wedge \uparrow ev \wedge \mathbf{true}). \quad (5.3)$$

A deterministic automaton that should observe this behaviour has to start a clock every time it observes an *ev* event. It cannot know in advance if this is the *ev* event that corresponds to the first event in the trace above, or if it is just part of one of the **true** phases. Only after the clock reached 1, the clock variable can be used for another *ev* event. However, there may be an unbounded number of *ev* events in an interval of length 1, so a deterministic automaton would need an infinite number of clocks. Therefore no deterministic phase event automaton for the above trace exists. It is not possible to build a phase event automaton that implements formula (5.3).

In this section we will show that for the restricted set of counterexample traces it is possible to build a deterministic automaton. The basic idea is adopted from finite automata theory: to construct a deterministic finite



The loop edges are labelled with:

$$\text{only}(\tau) \wedge \exists \text{State}$$

$$= \neg \text{newgoal} \wedge \neg \text{start} \wedge \neg \text{stop} \wedge \neg \text{passed} \\ \wedge \text{current}' = \text{current} \wedge \text{goal}' = \text{goal} \wedge \text{dir}' = \text{dir}$$

$$\text{only}(\text{newgoal}) \wedge \text{com_newgoal}$$

$$= \text{newgoal} \wedge \neg \text{start} \wedge \neg \text{stop} \wedge \neg \text{passed} \\ \wedge \text{Min} \leq \text{goal}' \leq \text{Max} \wedge \text{goal}' \neq \text{current} \\ \wedge \text{current}' = \text{current} \wedge \text{dir}' = \text{dir}$$

$$\text{only}(\text{start}) \wedge \text{com_start}$$

$$= \text{start} \wedge \neg \text{newgoal} \wedge \neg \text{stop} \wedge \neg \text{passed} \\ \wedge (\text{goal} > \text{current} \Rightarrow \text{dir}' = 1) \\ \wedge (\text{goal} < \text{current} \Rightarrow \text{dir}' = -1) \\ \wedge \text{current}' = \text{current} \wedge \text{goal}' = \text{goal}$$

$$\text{only}(\text{stop}) \wedge \text{com_stop}$$

$$= \text{stop} \wedge \neg \text{newgoal} \wedge \neg \text{start} \wedge \neg \text{passed} \\ \wedge \text{goal} = \text{current} \\ \wedge \text{current}' = \text{current} \wedge \text{goal}' = \text{goal} \\ \wedge \text{dir}' = \text{dir}$$

$$\text{only}(\text{passed}) \wedge \text{com_passed}$$

$$= \text{passed} \wedge \neg \text{newgoal} \wedge \neg \text{start} \wedge \neg \text{stop} \\ \wedge \text{current}' = \text{current} + \text{dir} \\ \wedge \text{goal}' = \text{goal} \wedge \text{dir}' = \text{dir}$$

Figure 5.2.: Translation of Object-Z part of elevator

automaton (DFA) from a nondeterministic one (NFA) the locations of the DFA are labelled by subsets of the locations of the NFA. One problem here are the clocks. If one transition of the non-deterministic automaton resets a clock and another does not and both transitions are enabled there must be a decision whether the clock should be reset in the deterministic automaton. To solve this problem we allow only either an upper bound ($\ell \leq t$) or a lower bound ($\ell \geq t$) for each phase. For upper bounds a clock is reset whenever the corresponding phase can be entered (even if it was active before) to keep the value of the clock as small as possible; for lower bounds the clock is reset only when the phase cannot stay active.

The state variables V of the DC automata are the same as for the Z part: $V = \text{Vars}(\text{State})$. The set of event variables A contains all channels in the interface part of the specification. The clock variables C are defined later. We need a unique clock for each phase with a time restriction.

The guards used in the automaton are from the set $\mathcal{L}(V \cup A \cup C)$. We quote logical operations on formulae of the set $\mathcal{L}(V \cup A \cup C)$, e. g., ‘ \wedge ’, ‘ \vee ’, ‘ \neg ’, ‘ \Leftrightarrow ’ to distinguish them from the logical symbols in the meta-language, in which we specify our algorithm. The operator ‘ \bigwedge ’ repeatedly applies ‘ \wedge ’ on a set of predicates. It is logically equivalent to a universal quantifier \forall , but is only used for finite sets. The values that represent the truth and falsehood predicates are named $[\mathbf{true}]$, $[\mathbf{false}] : \mathcal{L}(V \cup A \cup C)$ to distinguish them from the **true** and **false** predicate of the meta-language. Furthermore, we use the expression $[predicate]$ for some Boolean predicate in the meta-language to denote $[\mathbf{true}]$ or $[\mathbf{false}]$ depending on the value of predicate. This expression is defined as $[predicate] == \text{if } predicate \text{ then } [\mathbf{true}] \text{ else } [\mathbf{false}]$. Note that the evaluation of $[predicate]$ is done in the meta language.

We assume that equivalent formulae are represented by the same element in $\mathcal{L}(V \cup A \cup C)$. All unsatisfiable formulae are equal to $[\mathbf{false}]$ and all tautologies are equal to $[\mathbf{true}]$. For example, $([\mathbf{true}] \vee F) = [\mathbf{true}]$ and $([\mathbf{false}] \Leftrightarrow F) = (\neg F)$ for all $F \in \mathcal{L}(V \cup A \cup C)$.

5.3.1. Power Set Construction for Counterexamples

Recall the syntax of counterexample formulae from page 23:

$$\begin{aligned} ce_formula \quad ::= & \neg (phase \wedge (phase \mid events)) \\ & \wedge \dots \wedge (phase \mid events) \wedge \mathbf{true} \end{aligned}$$

$$\begin{aligned}
\textit{phase} & ::= (\mathbf{true} \mid \lceil \textit{Predicate} \rceil) [\wedge \ell \sim t] \\
& \quad \lceil \wedge \boxplus \textit{NAME} \dots \wedge \boxplus \textit{NAME} \rceil \\
\sim & ::= \leq \mid < \mid > \mid \geq \\
\textit{events} & ::= \uparrow \textit{NAME} \mid \not\downarrow \textit{NAME} \\
& \quad \mid \textit{events} \vee \textit{events} \mid \textit{events} \wedge \textit{events}
\end{aligned}$$

A counterexample is a negated list of *phase* and *events* formulae that are combined by chop operators. It must begin with a *phase* formula and end with **true**. A *phase* is either **true** or an everywhere formula $\lceil \textit{Predicate} \rceil$ where *Predicate* is some formula from $\mathcal{L}(V)$. Its duration can be restricted with $\ell \sim t$ and events can be forbidden to occur during the phase using $\boxplus \textit{NAME}$. Here *NAME* must denote an element from the set *A*, the set of channels declared in the interface part. Note that there can be at most one time restriction and that equality $\ell = k$ is not allowed. This is necessary to forbid specifications like (5.3) that cannot be converted into a phase event automaton. An *events* formula always requires a zero-length interval on which some events must occur and others are forbidden.

We can represent a *phase* formula and all its preceding *events* formulae by the data structure *PhaseSpec*:

$$\textit{TimeOp} ::= \textit{none} \mid \textit{less} \mid \textit{lessequal} \mid \textit{greater} \mid \textit{greaterequal}$$

$ \begin{array}{l} \textit{PhaseSpec} \\ \textit{inv} : \mathcal{L}(V) \\ \textit{allowEmpty} : \mathbb{B} \\ \textit{timeop} : \textit{TimeOp} \\ \textit{bound} : \mathbf{Time} \\ \textit{forbidden} : \mathbb{F} A \\ \textit{entryEvents} : \mathcal{L}(A) \end{array} $
$ \begin{array}{l} \textit{allowEmpty} \Rightarrow (\textit{inv} = [\mathbf{true}] \\ \quad \wedge \textit{timeop} \notin \{\textit{greater}, \textit{greaterequal}\}) \\ \textit{timeop} = \textit{none} \Leftrightarrow \textit{bound} = 0 \end{array} $

The predicate *inv* is the state expression occurring in the *phase* if it is of the form $\lceil \textit{inv} \rceil$ and is **true** if *phase* is a **true**-phase. The Boolean flag *allowEmpty* is set for a **true**-phase to distinguish it from a phase $\lceil \mathbf{true} \rceil$; the latter requires a non-empty interval. The variable *timeop* denotes the

relation \sim and *bound* is the minimum or maximum time. If there is no restriction on the duration, *timeop* = *none* and *bound* = 0. The set *forbidden* contains all events *ev* occurring in a $\exists ev$ element of the *phase* formula. Finally, *entryEvents* is the conjunction of all *events* formulae preceding the *phase* formula. It is [**true**] if there is no preceding *events* formula.

A counterexample trace can be represented as a sequence of phase specifications.

$$Trace == \text{seq } PhaseSpec$$

For proving correctness of the construction it is convenient to define the *i*th prefix of a trace *tr*. The function *Prefix* returns a DC formula that describes the first *i* phases of a trace *tr*: The complete counterexample formula is the negation of *Prefix*(#*tr*, *tr*).

$$\left| \begin{array}{l} Prefix : \mathbb{N} \times Trace \rightarrow DCForm \\ \hline Prefix(0, tr) = \ell = 0 \\ \forall i : \mathbb{N} \mid i > 0 \bullet Prefix(i, tr) = Prefix(i - 1, tr) \\ \quad \wedge tr(i).entryEvents \\ \quad \wedge ([tr(i).inv] \vee tr(i).allowEmpty) \wedge \\ \quad \quad \ell \ tr(i).timeop \ tr(i).bound \wedge \\ \quad \quad \wedge ev : tr(i).forbiddenEvents \bullet \exists ev \end{array} \right.$$

The formula *Prefix* is defined inductively. With *Prefix*(0, *tr*) we denote the empty prefix, which is equivalent to $\ell = 0$, the neutral element of \wedge . When computing *Prefix*(*i*, *tr*) for $i > 0$ we add to *Prefix*(*i* - 1, *tr*) the DC formula that describes the *i*-th phase: At the beginning of the *i*th phase the entry event formula must be satisfied. If the phase is a **true** phase (denoted by *allowEmpty*) the part $[tr(i).inv] \vee tr(i).allowEmpty$ evaluates to **true**, otherwise the formula $[tr(i).inv]$ must hold. The length restriction $\ell \sim t$ is added (\sim is represented by *tr*(*i*).*timeop* and *t* by *tr*(*i*).*bound*). Finally, the forbidden events must not occur during this phase.

The algorithm needs to distinguish between phases that have a lower bound $\ell \geq k$, phases that have an upper bound $\ell \leq k$, and phases without any time bounds. Therefore we define auxiliary sets *LB*(*tr*) (for lower bounds) and *UB*(*tr*) (for upper bounds) that comprise these phases.

$$\begin{array}{|l}
LB, UB : Trace \rightarrow \mathbb{P}\mathbb{N} \\
\hline
\forall tr : Trace \bullet \\
\quad LB(tr) = \{i : \text{dom } tr \mid \\
\quad \quad tr(i).timeop \in \{greater, greaterequal\}\} \\
\quad \wedge UB(tr) = \{i : \text{dom } tr \mid \\
\quad \quad tr(i).timeop \in \{less, lessequal\}\}
\end{array}$$

For each phase i with an upper or lower bound we add a corresponding clock c_i that measures the duration of this phase.

The construction of the power set automaton works in a similar way as in finite automata theory. The locations of the automaton are labelled by set of trace indices. Whenever the automaton has detected a prefix of the counterexample trace up to a certain element, the corresponding index is in the set labelling the current location of the automaton. Additionally, the automaton also needs to remember discrete parts of the real-time behaviour, e.g., whether the lower bounds on the duration have passed. Instead of a single subset, we therefore have four subsets:

in An index is in this set if and only if the automaton has detected a prefix of the counterexample trace up to this element. We also call phase i *active*, if $i \in in$. Phases having a lower bound are even active if their bound has not yet been reached.

wait The index of a phase with a lower bound is in this set if it is active and the duration of the phase is less than the lower bound. In this case, the corresponding clock measures the duration of the phase and as soon as the lower bound is reached, a new location is entered, where the index is no longer in the *wait* set. We will also say that a phase is *waiting* if its index is in this set.

gteq To support phases with the bound $\ell \geq k$, we reset the corresponding clock c_i at the earliest moment, the phase gets active. As soon as $c_i \geq k$, the waiting state is left. Under certain circumstances, the clock is reset *before* the phase can be entered. Later on, we present an example where this happens. In this case, instead of checking for $c_i \geq k$, the automaton needs to check for $c_i > k$. To distinguish these cases, the *gteq* set is used. A phase i is in this set if the automaton needs to check for $c_i \geq k$. This set is always a subset of *wait*. We will also say that the *gteq* flag is set for the phase i if $i \in gteq$.

less Similarly, sometimes for phases with an upper bound $\ell \leq k$ the clock is reset too late. So instead of checking for $c_i \leq k$, the automaton needs to check for $c_i < k$. In this case, phase i is in the set *less*. If the bound of a phase is strict ($\ell < k$), this phase is always in the set when active. We will then also say that the *less* flag is set for this phase. Obviously, the *less* flag is only set for active phases with an upper bound. There is another side condition $i - 1 \in in \setminus wait \vee \emptyset \not\models tr(i).entryEvents$. This condition is explained in example 5.9.

The following schema *PowerSet* combines the four sets *in*, *wait*, *gteq*, and *less*. The definition depends on the counterexample trace $tr : Trace$.

$PowerSet(tr : Trace) \text{---}$ $in, wait, gteq, less : \text{dom } tr$
$in \subseteq \text{dom } tr$ $wait \subseteq in \cap LB(tr)$ $gteq \subseteq wait$ $less \subseteq in \cap UB(tr)$ $\forall i : less \bullet i - 1 \notin in \setminus wait \vee \emptyset \not\models tr(i).entryEvents$

Because of $gteq \subseteq wait \subseteq in$ and $less \subseteq in$ and $less \cap wait \subseteq UB \cap LB = \emptyset$, there are only five possibilities for a phase to belong to these sets. The four sets can be abbreviated as a single set, where the elements are flagged by symbols.

- If $i \in gteq \subseteq wait \subseteq in$ then $i \notin less$. We denote this case by i^{\geq} .
- If $i \notin gteq$ and $i \in wait \subseteq in$ then $i \notin less$. We denote this by $i^>$.
- If $i \in less \subseteq in$, then $i \notin gteq$ and $i \notin wait$. We denote this by $i^{<}$.
- If $i \notin gteq, wait, less$ and $i \in in$ then we denote this by plain i .
- Otherwise phase i belongs to none of the sets and is omitted from the single set.

Example 5.3. The location $p = \langle in \mapsto \{1, 2, 4\}, wait \mapsto \{2\}, gteq \mapsto \{2\}, less \mapsto \{4\} \rangle$ is abbreviated as $\{1, 2^{\geq}, 4^{<}\}$.

Example 5.4. Figure 5.3 depicts the automaton for the DC formula

$$\underbrace{\text{true}}_1 \wedge \underbrace{[A] \wedge \ell \geq 4}_2 \wedge \underbrace{[B] \wedge \ell < 6}_3 .$$

The unreachable locations with $1 \notin in$ were omitted. The figure shows that the power set automaton becomes complicated even for DC formulae with only few chops. In the following we will explain the algorithm by which this automaton is constructed.

Note that the automaton is deterministic in the sense of definition 4.20. For each location of the automaton and each clock, variable and event valuation there is exactly one successor location. For example, the top right location $\{1, 3^<\}$ has four outgoing edges to the top four locations. If for the next state, B does not hold or if $c_3 \geq 6$ holds the automaton changes to one of the left locations depending on the new value of A . Otherwise (if $c_3 < 6$ and B holds) it either stays in location $\{1, 3^<\}$ or changes to location $\{1, 2^{\geq}, 3^<\}$, also depending on the new value of A . It cannot change to one of these two locations if $c_3 = 6$ holds as it has to stay there for some non-zero time and that would violate the clock invariant.

To define when the automaton has observed the prefix of the trace up to and including the i th element, we need to know the current location of the automaton, the current value of clocks, and the pending events. For example, when the location labelled by $\{1, 3^<\}$ in figure 5.3 is active, the automaton has accepted the whole trace $Prefix(3, tr)$ only if $c_3 < 6$ holds. If $c_3 = 6$ holds, the bound of the last phase would be violated. If the current location is $\{1, 2^{\geq}\}$, the automaton accepted $Prefix(2, tr) = \mathbf{true} \wedge [A] \wedge \ell \geq 4$ if and only if $c_2 \geq 4$ holds.

Given a power set location p and a phase i , the function *complete* gives the condition under which the prefix $Prefix(i, tr)$ was observed. It also takes clocks and succeeding **true** phases with zero length into account. We will later use this function to prove the correctness of the construction.

$$complete : Trace \times \text{ran } PowerSet \times \mathbb{N} \rightarrow \mathcal{L}(A \cup C)$$

$$\forall tr : Trace; p : PowerSet(tr); i : \mathbb{N} \bullet$$

$$complete(tr, p, i) =$$

$$([i \in p.in] \wedge$$

$$\text{if } i \in p.wait \text{ then}$$

$$(\text{if } i \in p.gteq \text{ then } c_i \geq tr(i).bound \text{ else } [\mathbf{false}]))$$

$$\text{else } (\text{if } i \in p.less \text{ then } c_i < tr(i).bound \text{ else } [\mathbf{true}]))$$

$$\wedge ([i > 1 \wedge tr(i).allowEmpty] \wedge$$

$$complete(tr, p, i - 1) \wedge tr(i).entryEvents)$$

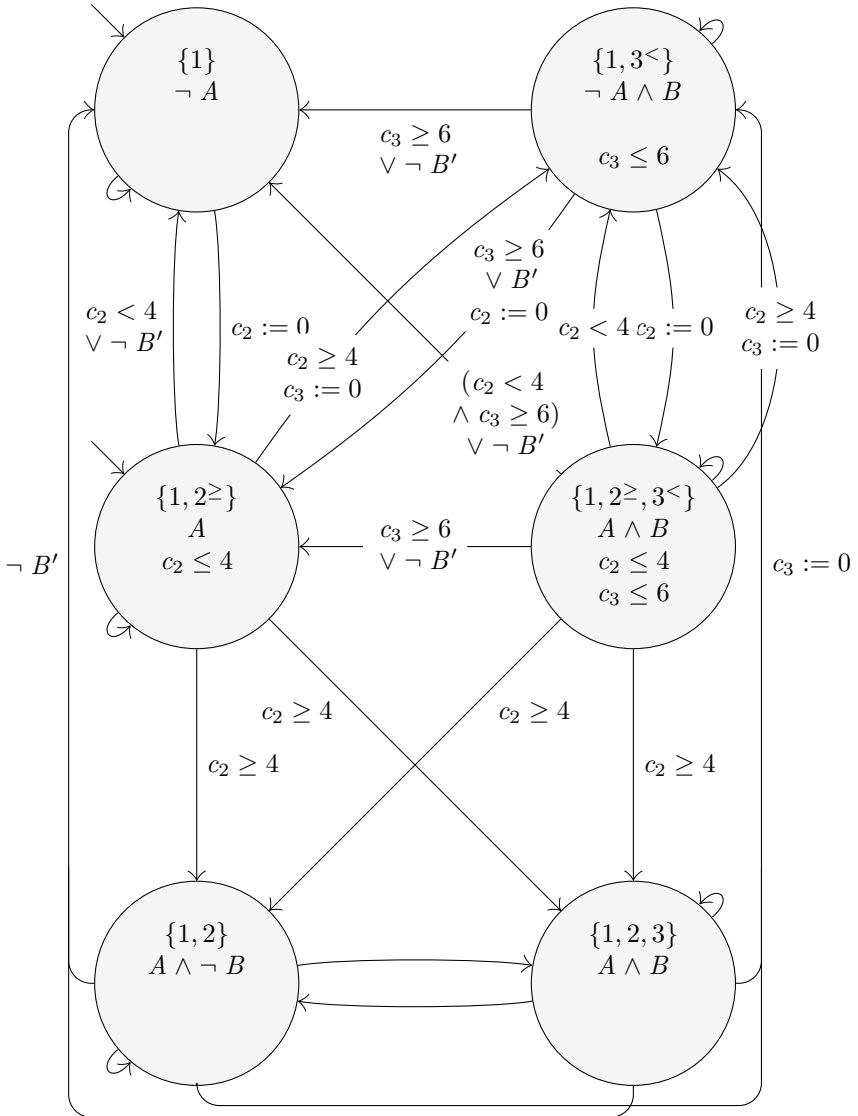


Figure 5.3.: Automaton for $\mathbf{true} \wedge [A] \wedge l \geq 4 \wedge [B] \wedge l < 6$

If phase i is active and has no time bounds it is always complete and the function *complete* will return **[true]**. If phase i has a lower time bound and is not waiting it is also complete. If it is waiting, it can only be complete if the phase has a greater-equal bound and the clock has just reached the bound. If the phase has a strict upper time bound ($i \in p.less$), it is only complete if the upper bound has not yet been reached.

A phase i is also complete if the previous phase is complete and phase i holds for an empty interval. In this case, the entry events of phase i must occur.

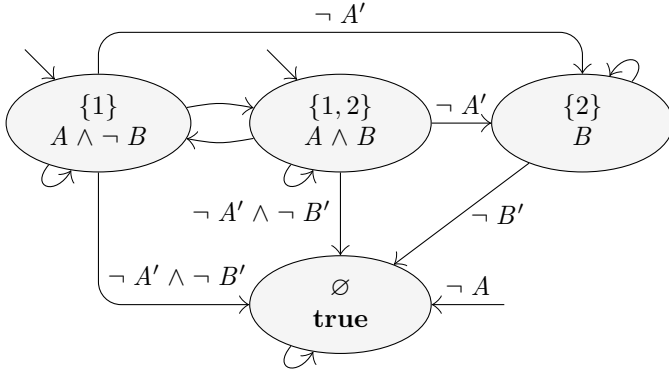
Example 5.5. For the top right location of the automaton in figure 5.3, the third phase is complete if and only if

$$\begin{aligned}
& complete(tr, \downarrow in \mapsto \{1, 3\}, wait \mapsto \emptyset, gteq \mapsto \emptyset, less \mapsto \{3\} \downarrow, 3) \\
&= [3 \in \{1, 3\}] \text{ '}\wedge\text{' } [3 \in \emptyset \text{ then } \dots \\
&\quad \text{else if } 3 \in \{3\} \text{ then } c_3 < 6 \text{ else } [\mathbf{true}]] \\
&\quad \text{'}\vee\text{' } ([\mathbf{false}] \text{ '}\wedge\text{' } complete(tr, p, 2) \text{ '}\wedge\text{' } [\mathbf{true}]) \\
&= ([\mathbf{true}] \text{ '}\wedge\text{' } c_3 < 6) \text{ '}\vee\text{' } ([\mathbf{false}]) \\
&= c_3 < 6
\end{aligned}$$

Consider two adjacent phases without any event required in between, e. g., $[A] \frown [B]$. If an interpretation satisfies $[A \wedge B]$ for some interval, it also satisfies the formula $[A] \frown [B]$. Therefore, for a valuation satisfying $A \wedge B$, the automaton should not only enter the first phase $[A]$ but should also immediately enter the second phase. We call this *seeping* because the control flow seeps through phase $[A]$ immediately to phase $[B]$. The function *canseep* determines if the automaton can seep into a phase i from the active phases given by the power set location p . This is not possible if an event is required for entering phase i . Also, the previous phase must be active and not waiting.

$$\left| \begin{array}{l}
canseep : Trace \times \text{ran } PowerSet \times \mathbb{N} \rightarrow \mathbb{B} \\
\hline
\forall tr : Trace; p : PowerSet(tr) \bullet \\
\quad canseep(tr, p, i) \Leftrightarrow i - 1 \in p.in \setminus p.wait \wedge \\
\quad \emptyset \models tr(i).entryEvents
\end{array} \right.$$

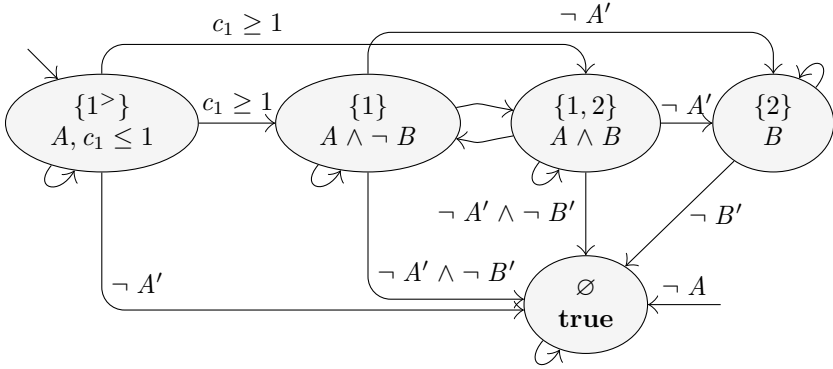
The mechanism of seeping is crucial to create a stuttering invariant automaton. However, it complicates the construction and is responsible for the *gteq* and *less* flags. The following examples illustrate the use of *canseep*:

Figure 5.4.: Automaton for $\lceil A \rceil \cap \lceil B \rceil$

Example 5.6. The automaton for the above example $\lceil A \rceil \cap \lceil B \rceil$ is given in figure 5.4. Note that there are three initial locations. If initially $\neg A$ holds, the location \emptyset is entered: the automaton observed $\lceil \neg A \rceil$, which is not a prefix of the formula $\lceil A \rceil \cap \lceil B \rceil$. If initially $A \wedge B$ holds, we enter location $\{1, 2\}$ because the interpretation satisfies both $\lceil A \rceil$ and $\lceil A \rceil \cap \lceil B \rceil$. If $A \wedge \neg B$ holds we only enter location $\{1\}$. The invariant of this location also requires $\neg B$ and the edge to $\{1, 2\}$ must be taken as soon as $A \wedge B$ holds. The location $\{2\}$ is entered if $B \wedge \neg A$ holds and the previous location was $\{1\}$ or $\{1, 2\}$. The condition B' was omitted from the corresponding edges as it is already implied by the invariant of the location $\{2\}$. However, the invariant does not require $\neg A$: We cannot reenter $\{1, 2\}$ as A did not hold continuously from the beginning.

The state invariant of a location p is the logical conjunction of the state expressions $tr(i).inv$ of all active phases $i \in p.in$ and the negation of the state expressions $tr(j).inv$ of all inactive phases j for which $canseep(tr, p, j)$ is true. The location $\{1\}$ has the additional invariant $\neg B$ because $canseep(tr, \{1\}, 2)$ is true. The location $\{2\}$ does not have the condition $\neg A$ because $canseep(tr, \{2\}, 1)$ is false.

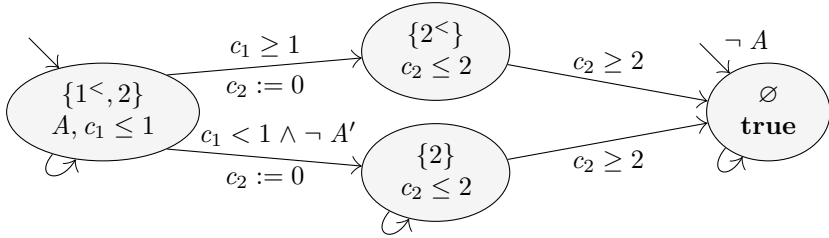
Example 5.7. If there is a lower bound for the first phase as in $\lceil A \rceil \wedge \ell > 1 \cap \lceil B \rceil$ seeping into B is only possible when the duration has expired. The corresponding automaton is depicted in figure 5.5. Note that the right four locations are identical to the four locations of the previous automaton.

Figure 5.5.: Automaton for $[A] \wedge \ell > 1 \wedge [B]$

The new location $\{1^>\}$ does not require $\neg B$, as the B -phase cannot be entered while $c_1 < 1$. The formula $\text{canseep}(tr, \{1^>\}, 2)$ evaluates to false because 1 is in the set *wait*. After one time unit, the automaton leaves the waiting location and switches to $\{1\}$ or $\{1, 2\}$ depending on the value of B . For these locations the seeping rule applies.

As seen in the previous examples, sometimes phases are entered too early. In example 5.6, the second phase can be entered at time zero, even though $[A]$ requires some positive duration. In example 5.7, the second phase can be entered at time 1, even though the first phase requires $\ell > 1$. Because of this, phases with a *greater* time bound need special treatment. The following example illustrates the construction using the *gteq* flag.

Example 5.8. The automaton in figure 5.6 belongs to the counterexample trace $[\text{true}] \wedge [B] \wedge \ell \geq 2 \wedge [\neg B]$. The first phase requires some non-zero duration, and thus if initially we start in phase 1 and phase 2 by seeping, then phase 2 is entered too early. Here location $\{1, 2^>\}$ is an initial location although phase 2 cannot start immediately. However, it can be initially entered because $\text{canseep}(tr, \{1\}, 2)$ holds. When the value of clock c_2 equals 2, the automaton has observed an interval where $[B] \wedge \ell = 2$ holds. However, the formula $[\text{true}] \wedge [B] \wedge \ell \geq 2$ does not hold for this interval as it requires a length of more than 2 time units. Therefore, the automaton cannot switch to phase 3 from $\{1, 2^>\}$ when $c_2 = 2$ and $\neg B$ holds. On the other hand, if a non-empty interval where $\neg B$ holds is

Figure 5.8.: Automaton for $\lceil A \rceil \wedge \ell < 1 \wedge \ell \leq 2$

The rule for a phase i with an upper bound is as follows. As long as the formula $\text{canseep}(tr, p, i)$ holds, the clock c_i is not used by the automaton. The transition from phase $i - 1$ to phase i can be taken at any time, so the upper bound of phase i can be fulfilled by taking the transition as late as possible. If seeping is not possible, for example, if phase $i - 1$ is no longer active, the clock c_i is reset to check the upper bound of phase i . Only in this case, the clock is compared against the bound of phase i . The last condition in $\text{PowerSet}(tr)$ schema makes sure that the *less* flag is only set when $\neg \text{canseep}(tr, p, i)$ holds, i.e., $i - 1 \notin p.in \setminus p.wait$ or $\emptyset \neq tr(i).entryEvents$.

Example 5.10. When applying the previous rule, the clock is reset too late, sometimes. For example, consider the formula $\lceil A \rceil \wedge \ell < 1 \wedge \ell \leq 2$. The corresponding automaton, c.f. figure 5.8, starts in $\{1<, 2\}$ if A holds, because if phase 1 is active, phase 2 is also active due to seeping. It stays in this location as long as A holds and $c_1 \leq 1$. Note that for a deterministic automaton, the clock invariant must not contain strict bounds. If A continues to hold, the location is left after one time unit. At this time, clock c_2 is reset. However, the DC formula requires phase 2 to start earlier. The clock was reset too late. Now, $\mathcal{I}, [0, t] \models \lceil A \rceil \wedge \ell < 1 \wedge \ell \leq 2$ holds only if $t < 3$. At time t , the clock c_2 has the value $t - 1$ so the DC formula is only accepted as long as $c_2 < 2$. Therefore, the *less* flag for phase 2 must be set in this case. On the other hand, if we have a different interpretation where at time $t < 1$ the predicate A no longer holds, then the automaton enters location $\{2\}$ at time t and the clock is reset at the right moment. In this case, the less flag is not set.

After a transition, a phase can be active for three different reasons: If it was active before, it *keeps* staying active if no forbidden events occur, and the upper bound is still satisfied. If the previous phase was complete, the phase can be *entered*. If the previous phase is active *after* the transition, a phase can be activated by *seeping*. For these three cases, the corresponding condition is computed by the following functions.

$$\begin{array}{|l}
 \hline
 \textit{keep}, \textit{enter}, \textit{seep} : \textit{Trace} \times \textit{ran PowerSet} \times \mathbb{N} \rightarrow \mathcal{L}(V' \cup A \cup C) \\
 \hline
 \forall tr : \textit{Trace}; p : \textit{PowerSet}(tr); i : \mathbb{N} \bullet \\
 \quad \textit{keep}(tr, p, i) = [i \in p.in] \textit{'}\wedge\textit{' } \\
 \quad \quad \textit{'}\bigwedge\{ev : tr(i).forbidden \bullet \neg ev\} \textit{'}\wedge\textit{' } (tr(i).inv) \textit{'}\wedge\textit{' } \\
 \quad \quad \textit{(if } i \in UB(tr) \wedge \neg \textit{canseep}(tr, p, i) \\
 \quad \quad \quad \textit{then } c_i < tr(i).bound \textit{ else } [\mathbf{true}]) \wedge} \\
 \quad \textit{enter}(tr, p, i) = \textit{complete}(tr, p, i - 1) \textit{'}\wedge\textit{' } \\
 \quad \quad tr(i).entryEvents \textit{'}\wedge\textit{' } (tr(i).inv) \textit{'}\wedge\textit{' } \\
 \quad \textit{seep}(tr, p, i) = [\textit{canseep}(tr, p, i)] \textit{'}\wedge\textit{' } (tr(i).inv) \textit{' }
 \end{array}$$

A phase stays active (*keep*) if and only if it was active before, no forbidden event occurs, the post state still satisfies the invariant, and for phases with an upper bound, that bound must not have been reached, yet. A new phase is activated (*enter*) if and only if the previous phase is complete, the post state satisfies the invariant of the new phase, and all entry events occur. The automaton can *seep* into a phase if the condition *canseep* and the phase invariant hold.

Given a counterexample trace tr , a location of the power set automaton $p : \textit{PowerSet}(tr)$ a set of clocks $X : \mathbb{P}C$ that are reset, and a successor location of the automaton $p' : \textit{PowerSet}(tr)$, the function *guard* (depicted in figure 5.9) calculates the corresponding guard g of the edge (p, g, X, p') in the power set automaton. A phase is active in the successor location $(p'.in)$ if and only if at least one of the following is true: It stays active (*keep*), it is activated (*enter*), or it can be entered together with the previous phase by the seeping rule (*seep*). Note that the function *seep* takes the successor location p' as parameter.

If for a phase with a lower bound the *keep* condition holds, the clock is not reset. This is because the clock should continue accumulating the time the phase was active. If *keep* does not hold, but the phase is activated due to *enter* or *seep*, the clock needs to be reset to start measuring the duration of the phase. In this case, the phase must also be marked as waiting. A phase is also waiting if it was waiting before and the corresponding clock

$$\begin{aligned}
& \text{guard} : \text{Trace} \times \text{ran PowerSet} \times \mathbb{P} C \times \text{ran PowerSet} \\
& \quad \rightarrow \mathcal{L}(V' \cup A \cup C) \\
\hline
& \forall tr : \text{Trace}; p : \text{PowerSet}(tr); X : \mathbb{P} C; p' : \text{PowerSet}(tr) \bullet \\
& \text{guard}(tr, p, X, p') = \wedge \{ i : \text{dom } tr \bullet \\
& \quad ([i \in p'.in] \Leftrightarrow \\
& \quad \quad \text{keep}(tr, p, i) \vee \text{enter}(tr, p, i) \vee \text{seep}(tr, p', i)) \wedge \\
& \quad (\text{if } i \in LB(tr) \cap p'.in \text{ then} \\
& \quad \quad ([c_i \in X] \Leftrightarrow \neg \text{keep}(tr, p, i)) \wedge \\
& \quad \quad ([i \in p'.wait] \Leftrightarrow \\
& \quad \quad \quad [c_i \in X] \vee \\
& \quad \quad \quad ([i \in p.wait] \wedge c_i < tr(i).bound)) \wedge \\
& \quad \quad ([i \in p'.gteq] \Leftrightarrow \\
& \quad \quad \quad \text{if } c_i \in X \text{ then } [tr(i).timeop = \text{greaterequal}] \wedge \\
& \quad \quad \quad \quad \text{enter}(tr, p, i) \\
& \quad \quad \quad \text{else } [i \in p.gteq \wedge i \in p'.wait]) \wedge \\
& \quad \quad [i \notin p'.less] \\
& \quad \text{else if } i \in UB(tr) \cap p'.in \wedge \neg \text{canseep}(tr, p', i) \text{ then} \\
& \quad \quad ([c_i \in X] \Leftrightarrow \text{enter}(tr, p, i) \vee [\text{canseep}(tr, p, i)]) \wedge \\
& \quad \quad [i \notin p'.wait] \wedge [i \notin p'.gteq] \wedge \\
& \quad \quad ([i \in p'.less] \Leftrightarrow \\
& \quad \quad \quad \text{if } c_i \in X \text{ then } [tr(i).timeop = \text{less}] \vee \\
& \quad \quad \quad \quad \neg \text{enter}(tr, p, i) \\
& \quad \quad \quad \text{else } [i \in p.less]) \\
& \quad \text{else } [c_i \notin X] \wedge \\
& \quad \quad [i \notin p'.wait] \wedge [i \notin p'.gteq] \wedge [i \notin p'.less] \}
\end{aligned}$$
Figure 5.9.: Definition of *guard*

has not reached its bound, yet. If the automaton enters a phase with a \geq bound directly (by the *enter* condition) this is remembered by the *gteq* flag. The flag is preserved if the automaton keeps staying in this phase as long as the waiting flag is also set.

When the automaton is in location p' containing a phase i with an upper bound and the clock is active, i.e., $\neg \text{canseep}(tr, p', i)$, the clock needs to be reset whenever the *enter* condition holds. It also needs to be reset if in the previous phase the clock was not yet active ($\text{canseep}(tr, p, i)$ holds, but the *enter* condition does not hold due to a violated *entryEvents* constraint). In the latter case the *less* flag is set even for phases with a \leq bound as the reset came too late: The phase i could have been entered at any time during the previous location p as $\text{canseep}(tr, p, i)$ holds but not at the moment the automaton takes the transition. To compensate this we have to treat this phase as if it has a strict bound. Of course, the *less* flag always has to be set when entering a phase with a strict bound. If the clock was not reset, the value of the *less* flag has to be kept the same.

Example 5.11. We continue the example of the automaton in figure 5.3 by calculating some guards for the trace

$$\underbrace{\mathbf{true}}_1 \wedge \underbrace{[A] \wedge \ell \geq 4}_2 \wedge \underbrace{[B] \wedge \ell < 6}_3 .$$

and the locations

$$p_1 = \{1\} = \langle \text{in} \mapsto \{1\}, \text{wait} \mapsto \emptyset, \text{gteq} \mapsto \emptyset, \text{less} \mapsto \emptyset \rangle$$

$$p_2 = \{1, 2^{\geq}, 3^{<}\} = \langle \text{in} \mapsto \{1, 2, 3\}, \text{wait} \mapsto \{2\}, \text{gteq} \mapsto \{2\}, \text{less} \mapsto \{3\} \rangle$$

$$p_3 = \{1, 3^{<}\} = \langle \text{in} \mapsto \{1, 3\}, \text{wait} \mapsto \emptyset, \text{gteq} \mapsto \emptyset, \text{less} \mapsto \{3\} \rangle$$

$$\text{guard}(tr, p_3, \emptyset, p_1) =$$

$$([1 \in p_1.\text{in}] \text{ '}\Leftrightarrow\text{'})$$

$$\text{keep}(tr, p_3, 1) \text{ '}\vee\text{' } \text{enter}(tr, p_3, 1) \text{ '}\vee\text{' } \text{seep}(tr, p_1, 1) \text{ '}\wedge\text{'}$$

$$[c_1 \notin \emptyset] \text{ '}\wedge\text{' } [1 \notin p_1.\text{wait}] \text{ '}\wedge\text{' } [1 \notin p_1.\text{gteq}] \text{ '}\wedge\text{' } [1 \notin p_1.\text{less}] \text{ '}\wedge\text{'}$$

$$([2 \in p_1.\text{in}] \text{ '}\Leftrightarrow\text{'})$$

$$\text{keep}(tr, p_3, 2) \text{ '}\vee\text{' } \text{enter}(tr, p_3, 2) \text{ '}\vee\text{' } \text{seep}(tr, p_1, 2) \text{ '}\wedge\text{'}$$

$$[c_2 \notin \emptyset] \text{ '}\wedge\text{' } [2 \notin p_1.\text{wait}] \text{ '}\wedge\text{' } [2 \notin p_1.\text{gteq}] \text{ '}\wedge\text{' } [2 \notin p_1.\text{less}] \text{ '}\wedge\text{'}$$

$$([3 \in p_1.\text{in}] \text{ '}\Leftrightarrow\text{'})$$

$$\begin{aligned}
& \text{keep}(tr, p_3, 3) \text{'}\vee\text{' } \text{enter}(tr, p_3, 3) \text{'}\vee\text{' } \text{seep}(tr, p_1, 3)) \text{'}\wedge\text{' } \\
& [c_3 \notin \emptyset] \text{'}\wedge\text{' } [3 \notin p_1.\text{wait}] \text{'}\wedge\text{' } [3 \notin p_1.\text{gteq}] \text{'}\wedge\text{' } [3 \notin p_1.\text{less}] \\
= & (\text{keep}(tr, p_3, 1) \text{'}\vee\text{' } \text{enter}(tr, p_3, 1) \text{'}\vee\text{' } \text{seep}(tr, p_1, 1)) \text{'}\wedge\text{' } [\mathbf{true}] \text{'}\wedge\text{' } \\
& \text{'}\neg\text{' } (\text{keep}(tr, p_3, 2) \text{'}\vee\text{' } \text{enter}(tr, p_3, 2) \text{'}\vee\text{' } \text{seep}(tr, p_1, 2)) \text{'}\wedge\text{' } [\mathbf{true}] \text{'}\wedge\text{' } \\
& \text{'}\neg\text{' } (\text{keep}(tr, p_3, 3) \text{'}\vee\text{' } \text{enter}(tr, p_3, 3) \text{'}\vee\text{' } \text{seep}(tr, p_1, 3)) \text{'}\wedge\text{' } [\mathbf{true}] \text{'}\wedge\text{' } \\
= & ([\mathbf{true}] \text{'}\vee\text{' } [\mathbf{false}] \text{'}\vee\text{' } [\mathbf{false}]) \text{'}\wedge\text{' } \\
& \text{'}\neg\text{' } ([\mathbf{false}] \text{'}\vee\text{' } A' \text{'}\vee\text{' } A') \text{'}\wedge\text{' } \\
& \text{'}\neg\text{' } ((c_3 < 6 \text{'}\wedge\text{' } B') \text{'}\vee\text{' } [\mathbf{false}] \text{'}\vee\text{' } [\mathbf{false}]) \text{'}\wedge\text{' } \\
= & [\mathbf{true}] \text{'}\wedge\text{' } \text{'}\neg\text{' } A' \text{'}\wedge\text{' } \text{'}\neg\text{' } (c_3 < 6 \text{'}\wedge\text{' } B') \\
= & \text{'}\neg\text{' } A' \text{'}\wedge\text{' } (c_3 \geq 6 \text{'}\vee\text{' } \text{'}\neg\text{' } B') \\
\text{guard}(tr, p_2, \emptyset, p_1) = & \\
& ([1 \in p_1.\text{in}] \text{'}\Leftrightarrow\text{' } \\
& \text{keep}(tr, p_2, 1) \text{'}\vee\text{' } \text{enter}(tr, p_2, 1) \text{'}\vee\text{' } \text{seep}(tr, p_1, 1)) \text{'}\wedge\text{' } \\
& [c_1 \notin \emptyset] \text{'}\wedge\text{' } [1 \notin p_1.\text{wait}] \text{'}\wedge\text{' } [1 \notin p_1.\text{gteq}] \text{'}\wedge\text{' } [1 \notin p_1.\text{less}] \text{'}\wedge\text{' } \\
& ([2 \in p_1.\text{in}] \text{'}\Leftrightarrow\text{' } \\
& \text{keep}(tr, p_2, 2) \text{'}\vee\text{' } \text{enter}(tr, p_2, 2) \text{'}\vee\text{' } \text{seep}(tr, p_1, 2)) \text{'}\wedge\text{' } \\
& [c_2 \notin \emptyset] \text{'}\wedge\text{' } [2 \notin p_1.\text{wait}] \text{'}\wedge\text{' } [2 \notin p_1.\text{gteq}] \text{'}\wedge\text{' } [2 \notin p_1.\text{less}] \text{'}\wedge\text{' } \\
& ([3 \in p_1.\text{in}] \text{'}\Leftrightarrow\text{' } \\
& \text{keep}(tr, p_2, 3) \text{'}\vee\text{' } \text{enter}(tr, p_2, 3) \text{'}\vee\text{' } \text{seep}(tr, p_1, 3)) \text{'}\wedge\text{' } \\
& [c_3 \notin \emptyset] \text{'}\wedge\text{' } [3 \notin p_1.\text{wait}] \text{'}\wedge\text{' } [3 \notin p_1.\text{gteq}] \text{'}\wedge\text{' } [3 \notin p_1.\text{less}] \\
= & (\text{keep}(tr, p_2, 1) \text{'}\vee\text{' } \text{enter}(tr, p_2, 1) \text{'}\vee\text{' } \text{seep}(tr, p_1, 1)) \text{'}\wedge\text{' } [\mathbf{true}] \text{'}\wedge\text{' } \\
& \text{'}\neg\text{' } (\text{keep}(tr, p_2, 2) \text{'}\vee\text{' } \text{enter}(tr, p_2, 2) \text{'}\vee\text{' } \text{seep}(tr, p_1, 2)) \text{'}\wedge\text{' } [\mathbf{true}] \text{'}\wedge\text{' } \\
& \text{'}\neg\text{' } (\text{keep}(tr, p_2, 3) \text{'}\vee\text{' } \text{enter}(tr, p_2, 3) \text{'}\vee\text{' } \text{seep}(tr, p_1, 3)) \text{'}\wedge\text{' } [\mathbf{true}] \text{'}\wedge\text{' } \\
= & ([\mathbf{true}] \text{'}\vee\text{' } [\mathbf{false}] \text{'}\vee\text{' } [\mathbf{false}]) \text{'}\wedge\text{' } \\
& \text{'}\neg\text{' } ((c_2 < 4 \text{'}\wedge\text{' } A') \text{'}\vee\text{' } A' \text{'}\vee\text{' } A') \text{'}\wedge\text{' } \\
& \text{'}\neg\text{' } ((c_3 < 6 \text{'}\wedge\text{' } B') \text{'}\vee\text{' } (c_2 \geq 4 \text{'}\wedge\text{' } B') \text{'}\vee\text{' } [\mathbf{false}]) \text{'}\wedge\text{' } \\
= & [\mathbf{true}] \text{'}\wedge\text{' } \text{'}\neg\text{' } A' \text{'}\wedge\text{' } \text{'}\neg\text{' } (B' \text{'}\wedge\text{' } (c_3 < 6 \text{'}\vee\text{' } c_2 \geq 4)) \\
= & \text{'}\neg\text{' } A' \text{'}\wedge\text{' } ((c_3 \geq 6 \text{'}\wedge\text{' } c_2 < 4) \text{'}\vee\text{' } \text{'}\neg\text{' } B')
\end{aligned}$$

The guards in figure 5.3 were further simplified by omitting $\text{'}\neg\text{' } A'$, which is already implied by the invariant of p_1 .

The following function computes the guard on the initial edge for each location of the automaton $p : \text{PowerSet}(tr)$.

$\text{init} : \text{Trace} \times \text{ran PowerSet} \rightarrow \mathcal{L}(V')$ <hr style="border: 0.5px solid black;"/> $\text{init}(tr, p) =$ $\text{if } p.\text{wait} = p.\text{in} \cap \text{LB}(tr)$ $\quad \wedge p.\text{less} = \text{if } tr(1).\text{timeop} = \text{less} \text{ then } \{1\} \text{ else } \emptyset$ $\quad \wedge p.\text{gteq} = \{i : p.\text{wait} \bullet tr(i).\text{timeop} = \text{greaterequal}$ $\quad \quad \wedge (\forall j : 1..(i-1) \bullet tr(j).\text{allowEmpty})\}$ $\text{then } \text{'}\bigwedge\text{' } i : \text{dom } tr \bullet$ $\quad ([i \in p.\text{in}] \text{'}\Leftrightarrow\text{' } tr(i).\text{inv} \text{'}\wedge\text{' } ([i = 1 \vee \text{canseep}(tr, p, i)]))$ $\text{else } [\mathbf{false}]$

In the initial location, all active phases with a lower bound must be in the waiting set. The *less* flag for the first phase is set if and only if the phase has a less bound. For all other phases with an upper bound, $\text{canseep}(tr, p, i)$ holds, so the clock is not yet started. The *gteq* flag must be set for all phases with a greaterequal bound that can be entered immediately. This is the case when all previous phases (if there are any) hold for the point interval. If any of these conditions is violated, $\text{init}(tr, p)$ is set to $[\mathbf{false}]$. Otherwise a phase is active in the initial location if and only if its invariant is satisfied and it is either the first phase or the seep condition holds.

The power set automaton $\mathcal{A}(tr)$ of a trace $tr \in \text{Trace}$ is defined as follows:

$\mathcal{A}(tr : \text{Trace})$ <hr style="border: 0.5px solid black;"/> $P == \text{PowerSet}(tr)$ $C == \{i : \text{UB}(tr) \cup \text{LB}(tr) \bullet c_i\}$ $E == \{p : P; X : \mathbb{P}(C); p' : P \mid \text{guard}(tr, p, X, p') \neq [\mathbf{false}]$ $\quad \bullet (p, \text{guard}(tr, p, X, p'), X, p')\}$ $s : P \rightarrow \mathcal{L}(V)$ $I : P \rightarrow \mathcal{L}(C)$ $E_0 == \{p : P \mid \text{init}(tr, p) \neq [\mathbf{false}] \bullet \text{init}(tr, p) \mapsto p\}$ <hr style="border: 0.5px solid black;"/> $\forall p : P \bullet s(p) = \text{'}\bigwedge\text{'}\{i : p.\text{in} \bullet tr(i).\text{inv}\} \text{'}\wedge\text{'}$ $\quad \text{'}\bigwedge\text{'}\{i : \text{dom } tr \setminus p.\text{in} \mid \text{canseep}(tr, p, i) \bullet \text{'}\neg\text{' } tr(i).\text{inv}\}$ $\forall p : P \bullet I(p) = \text{'}\bigwedge\text{'}\{i : p.\text{in} \mid i \in p.\text{wait} \vee$ $\quad (i \in \text{UB}(tr) \wedge \neg \text{canseep}(tr, p, i)) \bullet$ $\quad c_i \leq tr(i).\text{bound}\}$
--

The locations of the automaton are given by the schema $PowerSet(tr)$ as mentioned before. For each phase i with an upper or lower bound there is a clock $c_i \in C$. The edges are determined by the *guard* predicate we just defined: If $guard(tr, p, X, p')$ is not equal to **[false]**, the edge $(p, guard(tr, p, X, p'), X, p')$ is added. Likewise for each location p where $init(tr, p)$ is not equal to **[false]** the corresponding initial edge $(init(tr, p), p)$ is in E_0 . The state invariant $s(p)$ makes sure that the invariants of all active phases are satisfied and that all invariants of all inactive phases that can be entered by the seeping rule are not satisfied. The clock invariant $I(p)$ checks the clocks for all waiting phases and for all active phases with an upper bound. As mentioned earlier, the clock is only checked for phases with an upper bound if $\neg canseep(tr, p, i)$ holds.

Note that $guard(tr, p, X, p')$ (respectively $init(tr, p)$) can often be simplified by taking the invariant of the new location $s(p') \wedge I(p')$ (respectively $s(p)$) into account. In fact, there is only one case where $init(tr, p) \neq \mathbf{[false]}$ and $init(tr, p) \neq s(p)$: if $tr(1).inv \neq \mathbf{[true]}$ then $init(\emptyset) = \neg tr(1).inv$.

Our next goal is to prove that the automaton is stuttering invariant. Because of the seeping rule, no new locations can be entered by stuttering steps. Whenever a phase can be entered by a stuttering step, there is no entering event required and *canseep* holds. Hence, the phase was already active before due to seeping. This is shown by the following auxiliary lemma. It states that if a phase i is complete in location p , and the next phase $i + 1$ can be entered without any events occurring, and the clock invariant of location p holds strictly, the formula $canseep(tr, p, i + 1)$ holds.

Lemma 5.12. *Let p be a location of the power set automaton and $i \in dom\ tr$ some phase. Then*

$$(\forall e : A \bullet \neg e) \wedge s(p) \wedge strict(I(p)) \\ \wedge complete(tr, p, i) \wedge tr(i + 1).entryEvents \Rightarrow canseep(tr, p, i + 1)$$

Proof. Proof by induction over i : If $i \notin p.in$ then from $complete(tr, p, i)$, we have $i > 1$, $tr(i).allowEmpty$, $tr(i).entryEvents$, $complete(tr, p, i - 1)$. Thus $canseep(tr, p, i)$ holds by the induction hypothesis. Moreover, from $tr(i).allowEmpty$, we have $tr(i).inv = \mathbf{[true]}$ by definition of *PhaseSpec*. With $i \notin p.in$ and $canseep(tr, p, i)$, we have $s(p) = \mathbf{[false]}$ by definition of the power set automaton $\mathcal{A}(tr)$. This is a contradiction so $i \in p.in$.

If $i \in p.in$ and $i \in p.wait$ holds, then $i \in LB(tr)$ holds, i. e., $tr(i).timeop$ is *greater* or *greaterequal*. Hence, $\neg tr(i).allowEmpty$ holds by definition of *PhaseSpec*. Since $complete(tr, p, i)$ holds, $i \in p.gteq$ holds and

$complete(tr, p, i) = (c_i \geq tr(i).bound)$. However, $strict(I(p))$ requires $c_i < tr(i).bound$. Again, this is a contradiction. Therefore, $i \in p.in \setminus p.wait$.

From $(\forall e : A \bullet \neg e) \wedge tr(i+1).entryEvents$ follows

$$\emptyset \models tr(i+1).entryEvents.$$

With $i \in p.in \setminus p.wait$, this yields $canseep(tr, p, i+1)$. \square

Lemma 5.13. *The automaton is stuttering invariant: For each location $p \in P$ the following formula holds:*

$$\begin{aligned} (\forall x : V \bullet x = x') \wedge (\forall e : A \bullet \neg e) \wedge s(p) \wedge strict(I(p)) \\ \Rightarrow guard(p, g, X, p). \end{aligned}$$

Proof. We prove for all i that each conjunct of $guard(tr, p, \emptyset, p)$ holds. The first conjunct

$$[i \in p.in] \text{ '}\Leftrightarrow\text{' } keep(tr, p, i) \text{ '}\vee\text{' } enter(tr, p, i) \text{ '}\vee\text{' } seep(tr, p, i)$$

is implied by the following formulae that we show below:

$$keep(tr, p, i) \text{ '}\Leftrightarrow\text{' } [i \in p.in] \tag{5.4}$$

$$seep(tr, p, i) \text{ '}\Rightarrow\text{' } canseep(tr, p, i) \text{ '}\wedge\text{' } (tr(i).inv)' \tag{5.5}$$

$$enter(tr, p, i) \text{ '}\Rightarrow\text{' } canseep(tr, p, i) \text{ '}\wedge\text{' } (tr(i).inv)' \tag{5.6}$$

$$canseep(tr, p, i) \text{ '}\wedge\text{' } (tr(i).inv)' \text{ '}\Rightarrow\text{' } [i \in p.in] \tag{5.7}$$

If $i \notin p.in$ then

$$\begin{aligned} keep(tr, p, i) = & [i \in p.in] \text{ '}\wedge\text{' } \bigwedge \{ev : tr(i).forbidden \bullet \neg ev\} \text{ '}\wedge\text{' } \\ & (tr(i).inv)' \text{ '}\wedge\text{' } \\ & (\text{if } i \in UB(tr) \wedge \neg canseep(tr, p, i) \text{ then} \\ & \quad c_i < tr(i).bound \\ & \quad \text{else } [\mathbf{true}]) \end{aligned}$$

is false by definition. On the other hand, if $i \in p.in$ then $\bigwedge \{ev : tr(i).forbidden \bullet \neg ev\}$ follows from $(\forall e : A \bullet \neg e)$. The formula $(tr(i).inv)'$ follows from $s(p)$ and $\forall x : V \bullet x = x'$. If $i \in UB(tr) \wedge$

$\neg \text{canseep}(tr, p, i)$ then $c_i < tr(i).bound$ follows from $\text{strict}(I(p))$. This proves (5.4).

Formula (5.5) follows directly from the definition of $\text{seep}(tr, p, i)$. From the definition of $\text{enter}(tr, p, i)$ and lemma 5.12, we get (5.6):

$$\begin{aligned} \text{enter}(tr, p, i) &\Rightarrow \text{complete}(tr, p, i - 1) \wedge tr(i).entryEvents \wedge (tr(i).inv)' \\ &\Rightarrow \text{canseep}(tr, p, i) \wedge (tr(i).inv)' . \end{aligned}$$

Now assume that $(tr(i).inv)'$ and $\text{canseep}(tr, p, i)$ hold. From $s(p)$ and $\forall x : V \bullet x = x'$, it follows that $s(p)'$ holds. If $i \notin p.in$ holds, $s(p)'$ and $\text{canseep}(tr, p, i)$ would imply $\neg (tr(i).inv)'$. Therefore, $i \notin p.in$ leads to a contradiction. This proves formula (5.7).

If $i \in LB(tr) \cap p.in$ the other conjuncts of $\text{guard}(tr, p, \emptyset, p)$ are:

$$[c_i \in \emptyset] \Leftrightarrow \neg \text{keep}(tr, p, i) \quad (5.8)$$

$$[i \in p.wait] \Leftrightarrow [c_i \in \emptyset] \vee ([i \in p.wait] \wedge c_i < tr(i).bound), \quad (5.9)$$

$$[i \in p.gteq] \Leftrightarrow$$

$$\text{if } c_i \in \emptyset \text{ then } [tr(i).timeop = \text{greaterequal}] \wedge \text{enter}(tr, p, i) \quad (5.10)$$

$$\text{else } [i \in p.gteq \wedge i \in p.wait],$$

$$[i \notin p.less]. \quad (5.11)$$

From (5.4) follows $\text{keep}(tr, p, i)$ hence (5.8) holds because both sides are false. From $\text{strict}(I(p))$, we have that $i \in p.wait$ implies $c_i < tr(i).bound$. Thus, (5.9) holds. Because $c_i \notin \emptyset$ and $p.gteq \subseteq p.wait$ formula (5.10) simplifies to

$$[i \in p.gteq] \Leftrightarrow [i \in p.gteq].$$

Finally, (5.11) is true because $i \in LB(tr)$, and $p.less \subseteq UB(tr)$ by definition of $\text{PowerSet}(tr)$.

If $i \in UB(tr) \cap p.in$ and $\neg \text{canseep}(tr, p, i)$ holds, then we need to show:

$$[c_i \in \emptyset] \Leftrightarrow \text{enter}(tr, p, i) \vee [\text{canseep}(tr, p, i)], \quad (5.12)$$

$$[i \notin p.wait] \wedge [i \notin p.gteq], \quad (5.13)$$

$$[i \in p.less] \Leftrightarrow$$

$$\text{if } c_i \in \emptyset \text{ then } [tr(i).timeop = \text{less}] \vee \neg \text{enter}(tr, p, i) \quad (5.14)$$

$$\text{else } [i \in p.less],$$

From (5.6) and $\neg \text{canseep}(tr, p, i)$ follows that $\text{enter}(tr, p, i)$ does not hold. Hence (5.12) is true because both sides are false. From $i \in UB(tr)$ and

$p.gteq \subseteq p.wait \subseteq LB(tr)$, we have (5.13). Finally, formula (5.14) simplifies to

$$[i \in p.less] \text{ '}\Leftrightarrow\text{' } [i \in p.less],$$

which is trivially true.

If neither $i \in LB(tr) \cap p.in$ nor $i \in UB(tr) \cap p.in \wedge \neg canseep(tr, p, i)$ we have from the invariant of $PowerSet(tr)$, that $i \notin p.wait \wedge i \notin p.gteq \wedge i \notin p.less$. \square

Now, we prove that the power set automaton is also deterministic. This is very useful: It guarantees that there is a run for each interpretation and that all runs only differ in duration and the number of stuttering steps.

Lemma 5.14. *The automaton is deterministic.*

Proof. We prove the three parts of definition 4.20 separately:

1. For each valuation β , there is exactly one location $p \in PowerSet(tr)$ with $\beta \models s(p)$ and $\beta \models init(tr, p)$. This unique location corresponds to the unique initial edge $(init(tr, p), p) \in E_0$.

Location p can be constructed as follows: Start with $p.in = p.wait = p.less = p.gteq = \emptyset$ and $i = 1$. Then execute the following steps:

- (1) Add i to $p.in$ if (and only if)

$$\beta \models tr(i).inv \text{ '}\wedge\text{' } [i = 1 \vee canseep(tr, p, i)] \quad (5.15)$$

holds. Note that $canseep(tr, p, i)$ only depends on $i - 1 \in p.in \setminus p.wait$, which has been computed before.

- (2) Add i to $p.wait$ if $i \in p.in \cap LB(tr)$ holds.
- (3) Add i to $p.gteq$ if $i \in p.wait$ and $tr(i).timeop = greaterequal$ hold and $tr(j).allowEmpty$ holds for all $j < i$.
- (4) Add i to $p.less$ if $i = 1$ and $tr(1).timeop = less$ hold.
- (5) Increase i and go to step (1) if $i \leq \#tr$ holds.

This procedure gives us a state p such that $\beta \models init(tr, p)$ holds. Note that this state is unique. For example, if i is not added to $p.in$ in step (1) although (5.15) holds, $init(tr, p)$ cannot hold for valuation β , because it contains the conjunct

$$[i \in tr(i).inv] \Leftrightarrow tr(i).inv \text{ '}\wedge\text{' } [i = 1 \vee canseep(tr, p, i)].$$

Similarly, if i is added to $p.in$ although (5.15) does not hold. Step (1) also guarantees that $\beta \models s(p)$ holds.

If p is constructed as above, it is indeed a location that fulfils the invariant of the schema $PowerSet(tr)$. It was constructed such that $p.wait = p.in \cap LB(tr)$, $p.gteq \subseteq p.wait$, and if $i \in p.less$ then $i = 1$, $1 \in UB(tr)$ and $0 \notin p.wait \setminus p.less$.

2. For each location $p \in P$ and each pair of state valuations $\beta_1, \beta_2 \in \mathbf{Val}(V)$, $Y \in \mathbb{P}A$, and $\gamma \in \mathbf{Val}(C)$ there is exactly one $X \in \mathbb{P}C$ and one location $p' \in P$ such that $\beta_1, \beta'_2, Y, \gamma \models guard(tr, p, X, p')$ holds. This gives the unique edge $(p, guard(tr, p, X, p'), X, p') \in E$. The set X and location p' can be constructed as follows. Start with $X = p'.in = p'.wait = p'.less = p'.gteq = \emptyset$ and $i = 1$. Then execute the following steps:

- (1) Add i to $p'.in$ if and only if

$$\beta_1, \beta'_2, Y, \gamma \models keep(tr, p, i) \text{ '}\vee\text{' } enter(tr, p, i) \text{ '}\vee\text{' } seep(tr, p', i)$$

holds. Note that $seep(tr, p', i)$ only depends on tr, i and $i - 1 \in p'.in \setminus p'.wait$. The latter has already been computed.

- (2) If $i \in LB(tr) \cap p'.in$ does not hold continue with step (7).
- (3) Add c_i to X iff $\neg keep(tr, p, i)$ holds.
- (4) Add i to $p'.wait$ if and only if:

$$\beta_1, \beta'_2, Y, \gamma \models [c_i \in X] \text{ '}\vee\text{' } ([i \in p.wait] \wedge c_i < tr(i).bound).$$

- (5) Add i to $p'.gteq$ if and only if

$$\begin{aligned} \beta_1, \beta'_2, Y, \gamma \models \\ \text{if } c_i \in X \text{ then } [tr(i).timeop = greaterequal] \text{ '}\wedge\text{' } enter(tr, p, i) \\ \text{else } [i \in p.gteq \wedge i \in p'.wait]. \end{aligned}$$

- (6) Go to step (10).
- (7) If $i \in UB(tr) \cap p'.in \wedge \neg canseep(tr, p', i)$ does not hold continue with step (10).
- (8) Add c_i to X iff $enter(tr, p, i) \text{ '}\vee\text{' } [canseep(tr, p, i)]$ holds.

(9) Add i to $p'.less$ if and only if

$$\begin{aligned} \beta_1, \beta_2', Y, \gamma \models & \\ \text{if } c_i \in X \text{ then } [tr(i).timeop = less] \vee \neg \text{enter}(tr, p, i) & \\ \text{else } [i \in p.less]. & \end{aligned}$$

(10) Increase i and go to step (1) if $i \leq \#tr$.

To evaluate the conditions, the algorithm uses only the part of the set X and state p' that has been computed in the preceding steps. Each step of the algorithm corresponds to a conjunct of $guard(tr, p, X, p')$. This conjunct evaluates to true if p' and X were constructed as above. Furthermore, if at some step a different value for p' or X would be chosen, the corresponding conjunct evaluates to *false*. Hence this algorithm computes a unique solution for p' and X such that $guard(tr, p, X, p')$ is true.

The computed location p' is indeed a valid location satisfying the invariant of $PowerSet(tr)$. If $i \notin p'.in \cap LB(tr)$ the algorithm skips step (3–6), hence $i \notin p'.wait$. Thus, $p'.wait \subseteq p'.in \cap LB(tr)$. Similarly, i is only added to $p'.gteq$ in step (5) if $c_i \in X$ or $i \in p'.wait$. If $c_i \in X$, i was added to $p'.wait$ in step (4). Hence, $p'.gteq \subseteq p'.wait$. If $i \in p'.less$, then i was added in step (9). This step is only reachable if $i \in UB(tr) \cap (p'.in)$ and $\neg canseep(tr, p', i)$ in step (7). The latter implies $i - 1 \notin p'.in \setminus p'.wait \vee \emptyset \not\models tr(i).entryEvents$, as required by the invariant of $PowerSet(tr)$.

3. The clock invariant $I(p)$ is always a conjunction of $c_i \leq tr(i).bound$ by definition. □

The following lemma shows that whenever $complete(tr, p, i)$ holds, then for the corresponding DC interpretation and interval, $Prefix(i, tr)$ holds. Furthermore, when a phase i is active ($i \in p.in$) then formula $Prefix(i, tr)$ holds, except that the condition $\ell \sim tr(i).bound$ does not need to hold. If the clock is active, the duration of that interval roughly equals the value of the clock $\gamma_n(c_i) + t_n$. However, it may be smaller or larger if the clock was reset too early or too late.

Lemma 5.15. *Given an interpretation \mathcal{I} and a matching run*

$$run = \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots, (p_n, Y_n, \beta_n, \gamma_n, t_n), \dots \rangle$$

of the power set automaton. Then for all steps n of the run and all $i \in \text{dom } tr$:

1. *If $i \in p_n.in$ then for some $len > 0$:*

$$\mathcal{I}, \left[0, \sum_{j=0}^n t_j \right] \models Prefix(i-1, tr) \wedge tr(i).entryEvents \wedge \\ [tr(i).inv] \wedge \ell = len \wedge \bigwedge ev : tr(i).forbidden \bullet \boxminus ev \quad (5.16)$$

Furthermore, the variable len can be chosen such that the following side-conditions hold:

- (a) *If $i \in LB$ and $i \notin p_n.wait$ then (5.16) holds for some $len > tr(i).bound$.*
 - (b) *If $i \in p_n.wait$ then for all $\varepsilon > 0$, it holds for some $len > \gamma_n(c_i) + t_n - \varepsilon$.*
 - (c) *If $i \in p_n.gteq$ it holds for $len = \gamma_n(c_i) + t_n$.*
 - (d) *If $i \in UB \wedge \text{canseep}(tr, p_n, i)$ then for all $\varepsilon > 0$, (5.16) holds for some $len < \varepsilon$.*
 - (e) *If $i \in UB \wedge \neg \text{canseep}(tr, p_n, i)$ then it holds for some $len < \gamma_n(c_i) + t_n + \varepsilon$.*
 - (f) *If $i \in UB \wedge \neg \text{canseep}(tr, p_n, i) \wedge i \notin p_n.less$ then it holds for $len = \gamma_n(c_i) + t_n$.*
2. *If $\gamma_n + t_n, Y_{n+1} \models \text{complete}(tr, p_n, i)$ then*

$$\mathcal{I}, \left[0, \sum_{j=0}^n t_j \right] \models Prefix(i, tr).$$

Proof. We show both parts by simultaneous induction over n and i ; the well-ordering used for induction is the component-wise ordering of (n, i) . Firstly, we show part 1 under the assumption that the induction hypothesis holds for *both* parts of the lemma. Secondly, we show that part 2 is implied by part 1.

1. We assume $i \in p_n.in$.

Assume $n = 1$ and $i = 1$. The initial edge $(init(tr, p_1), p_1) \in E_0$ is taken by the automaton. Hence, $\beta_1 \models init(tr, p_1)$ holds. Because $i \in p_1.in$, this implies $\beta_1 \models s(p_1)$. Therefore, formula (5.16) holds for $len = t_1$. Since $\gamma_1(c_i) = 0$, we have $len = \gamma_1(c_i) + t_1$. Hence, (c) and (f) hold. Also $len > \gamma_1(c_i) + t_1 - \varepsilon$ and $len < \gamma_1(c_i) + t_1 + \varepsilon$ for all $\varepsilon > 0$. Hence, (b) and (e). If $i \in LB$ holds, we can deduce $i \in p_1.wait$ because otherwise $init(tr, p_1)$ would be false. Hence, (a) is vacuously true. Note that by definition, $canseep(tr, p_1, 1)$ never holds. Hence, (d) is vacuously true.

Assume $n = 1$ and $i > 1$. The definition of $init$ requires that $canseep(tr, p_1, i)$ holds. Therefore, we have $i - 1 \in p_1.in \setminus p_1.wait$. If $i - 1 \in p_1.less$ holds, the invariant $I(p_1)$ asserts that $\gamma_1(c_i) + t_1 \leq tr(i).bound$. Therefore, for each t with $0 < t < t_1$, $\gamma_1(c_i) + t, \emptyset \models complete(tr, p_1, i - 1)$ holds. The induction hypothesis for part 2. and $n = 1$ and $i - 1$ applied on the stuttering run

$$run' = \langle (p_1, Y_1, \beta_1, \gamma_1, t), (p_1, \emptyset, \beta_1, \gamma_1 + t, t_1 - t), \dots \rangle$$

yields $\mathcal{I}, [0, t] \models Prefix(i - 1, tr)$. From $canseep(tr, p_1, i)$, we have $\emptyset \models tr(i).entryEvents$. From $\beta_1 \models s(p_1)$ and $i \in p_1.in$, we have $\beta_1 \models tr(i).inv$. Therefore, (5.16) holds for $len = t_1 - t = \gamma_1(c_i) + t_1 - t$.

Consider the further conditions on len . Case (a) is vacuously true since $i \in LB$ implies $i \in p_1.wait$ by definition of $init$. For (b), choose for $\varepsilon > 0$ some $t < \varepsilon$ yielding $len > \gamma_1(c_i) + t_1 - \varepsilon$. To prove (d), choose some $t > t_1 - \varepsilon$ yielding $len < \varepsilon$. Since $canseep(tr, p_1, i)$ holds, (e) and (f) hold vacuously. If $i \in p_1.gteq$, then by definition of $init$ for all $j < i$ $tr(j).allowEmpty$ hold. Thus, $Prefix(i - 1, tr)$ holds on the interval $[0, 0]$ and (5.16) holds for $len = t_1 = \gamma_1(c_i) + t_1$. Hence, (c) holds.

Now, assume $n > 1$. There is an edge (p_{n-1}, g, X, p_n) of the automaton with

$$\beta_{n-1}, \beta'_n, \gamma_{n-1} + t_{n-1}, Y_n \models g.$$

For the power set automaton we have $g = guard(tr, p_{n-1}, X, p_n)$. If $i \in p_n.in$ then by definition of $guard$:

$$\beta_{n-1}, \beta'_n, \gamma_{n-1} + t_{n-1}, Y_n \models \\ keep(tr, p_{n-1}, i) \vee enter(tr, p_{n-1}, i) \vee seep(tr, p_n, i)$$

We distinguish the three cases:

keep If $keep(tr, p_{n-1}, i)$ holds then $i \in p_{n-1}$. By induction hypothesis

$$\begin{aligned} \mathcal{I}, \left[0, \sum_{j=0}^{n-1} t_j \right] \models & Prefix(i-1, tr) \frown tr(i).entryEvents \frown \\ & (\lceil tr(i).inv \rceil \vee tr(i).allowEmpty) \wedge \\ & \ell = len' \wedge \bigwedge ev : tr(i).forbidden \bullet \boxplus ev \end{aligned} \quad (5.17)$$

The formula $keep(tr, p_{n-1}, i)$ requires that no forbidden events occur in Y_n and that $tr(i).inv$ holds in β_n . Therefore, (5.16) holds for $len = len' + t_n$. It remains to be shown that len' can be chosen such that $len = len' + t_n$ fulfils the side-conditions.

If $i \in LB$ then $c_i \notin X$ by definition of *guard*. Thus, $\gamma_n(c_i) = \gamma_{n-1}(c_i) + t_{n-1}$. Since $c_i \notin X$, $guard(tr, p, X, p')$ implies

$$[i \in p'.wait] \text{ '}\Leftrightarrow\text{' } [i \in p.wait] \text{ '}\wedge\text{' } c_i < tr(i).bound .$$

If $i \notin p_n.wait$, then either $i \notin p_{n-1}.wait$, or $i \in p_{n-1}.wait$ and $\gamma_{n-1} + t_{n-1} \models c_i \geq tr(i).bound$. In the former case (5.17) holds for some $len' > tr(i).bound$ by (a). In the latter case, we can choose $\varepsilon = t_n$ and (5.17) holds for some

$$len' > \gamma_{n-1}(c_i) + t_{n-1} - t_n \geq tr(i).bound - t_n .$$

Therefore, in both case $len = len' + t_n > tr(i).bound$, which means that (a) holds. If $i \in p_n.wait$ then $i \in p_{n-1}.wait$ by definition of *guard*. For each $\varepsilon > 0$ there is some $len' > \gamma_{n-1}(c_i) + t_{n-1} - \varepsilon = \gamma_n(c_i) - \varepsilon$. Then $len > \gamma_n(c_i) + t_n - \varepsilon$. Therefore (b) holds. If $i \in p_n.gteq$ then $i \in p_{n-1}.gteq$ by definition of *guard*. Hence, the induction hypothesis yields $len' = \gamma_{n-1}(c_i) + t_{n-1} = \gamma_n(c_i)$. Therefore, $len = \gamma_n(c_i) + t_n$, as required by (c).

Now consider $i \in UB$: Assume $\neg canseep(tr, p_n, i)$ holds; the other case is proven in section **seep**. Therefore, only (e) and (f) have to be considered. The formula $guard(tr, p_{n-1}, i, p_n)$ implies

$$[c_i \in X] \text{ '}\Leftrightarrow\text{' } enter(tr, p_{n-1}, i) \text{ '}\vee\text{' } [canseep(tr, p_{n-1}, i)] . \quad (5.18)$$

We consider the case $c_i \in X$ and $c_i \notin X$ separately.

If $c_i \in X$, then $\gamma_n(c_i) = 0$. If $\text{enter}(tr, p_{n-1}, i)$ holds the property is proven in section **enter**. Otherwise, formula (5.18) implies $\text{canseep}(tr, p_{n-1}, i)$. By induction hypothesis for each $\varepsilon > 0$ there is a $len' < \varepsilon$ such that (5.17) holds. This yields $len < t_n + \varepsilon = \gamma_n(c_i) + t_n + \varepsilon$. Thus, (e) holds. From $c_i \in X$ and $\neg \text{enter}(tr, p_{n-1}, i)$ follows $i \in p_n.\text{less}$ by definition of *guard*. Thus, (f) holds vacuously.

If $c_i \notin X$ then $\neg \text{canseep}(tr, p_{n-1}, i)$ holds. Since c_i is not reset we have $\gamma_n(c_i) = \gamma_{n-1}(c_i) + t_{n-1}$. The induction hypothesis gives for each $\varepsilon > 0$ some $len' < \gamma_{n-1}(c_i) + t_{n-1} + \varepsilon = \gamma_n(c_i) + \varepsilon$ with (5.17). Thus, $len < \gamma_n(c_i) + t_n + \varepsilon$ holds, as required by the condition (e). If additionally $i \notin p_n.\text{less}$, then $i \notin p_{n-1}.\text{less}$ by definition of *guard*. Therefore, the induction hypothesis yields $len' = \gamma_{n-1}(c_i) + t_{n-1}$, which implies $len = \gamma_n(c_i) + t_n$. Thus, the condition (f) holds.

enter If $\text{enter}(tr, p_{n-1}, i)$ holds then

$$\gamma_{n-1} + t_{n-1}, Y_n \models \text{complete}(tr, p_{n-1}, i - 1).$$

By using the induction hypothesis for part 2. and $n-1$ and $i-1$, we get

$$\left[0, \sum_{j=0}^{n-1} t_j \right] \models \text{Prefix}(i - 1, tr).$$

Furthermore, $Y_n \models tr(i).\text{entryEvents}$ and $\beta_n \models tr(i).\text{inv}$ hence (5.16) holds for $len = t_n$.

If $i \in LB$ we can assume $\neg \text{keep}(tr, p_{n-1}, i)$, as we proved the other case above. By definition of *guard*, $c_i \in X$ and $i \in p_n.\text{wait}$ hold. Hence (a) holds vacuously. We have $len = t_n = \gamma_n(c_i) + t_n$ and $len > \gamma_n(c_i) + t_n - \varepsilon$ for all $\varepsilon > 0$. Thus, (b) and (c) hold.

If $i \in UB$ we assume $\neg \text{canseep}(tr, p_n, i)$. The other case is proved in section **seep** below. By definition of *guard*, $c_i \in X$ holds. Therefore, $len = t_n = \gamma_n(c_i) + t_n$. Thus, (f) holds. Furthermore, $len < \gamma_n(c_i) + t_n + \varepsilon$ for all $\varepsilon > 0$. Thus, (e) holds.

seep From $\text{canseep}(tr, p_n, i)$, we have $i - 1 \in p_n.\text{in} \setminus p_n.\text{wait}$. If $i - 1 \in p_n.\text{less}$ holds, the invariant $I(p_n)$ asserts that $\gamma_n(c_i) +$

$t_n \leq tr(i).bound$. Therefore for each t with $0 < t < t_n$: $\gamma_n(c_i) + t, \emptyset \models complete(tr, p_n, i - 1)$. The induction hypothesis for part 2. and n and $i - 1$ applied on the stuttering run

$$run' = \langle \dots, (p_{n-1}, Y_{n-1}, \beta_{n-1}, \gamma_{n-1}, t_{n-1}), \\ (p_n, Y_n, \beta_n, \gamma_n, t), (p_n, \emptyset, \beta_n, \gamma_n + t, t_n - t), \dots \rangle$$

yields $\mathcal{I}, [0, \sum_{j=0}^{n-1} t_j + t] \models Prefix(i - 1, tr)$. From $seep(tr, p_n, i)$ follows $\emptyset \models tr(i).entryEvents$ and $\beta_n \models tr(i).inv$. Therefore, (5.16) holds for $len = t_n - t$.

If $i \in LB$ holds, assume $keep(tr, p_{n-1}, i)$ and $enter(tr, p_{n-1}, i)$ both do not hold (these cases were examined before). By definition of *guard*, $c_i \in X$, $i \in p_n.wait$, and $i \notin p_n.gteq$ hold. Hence (a) and (c) hold vacuously. From $c_i \in X$ follows $\gamma_n(c_i) = 0$. For each $\varepsilon > 0$, t can be chosen such that $0 < t < \varepsilon$ and $t < t_n$. Then, $len = t_n - t > \gamma_n(c_i) + t_n - \varepsilon$. Hence, (b) holds.

If $i \in UB$, for each $\varepsilon > 0$, t can be chosen such that $t_n - \varepsilon < t < t_n$. This yields $len = t_n - t < \varepsilon$. Hence, (d) holds. Since $canseep(tr, p_n, i)$ holds, (e) and (f) hold vacuously.

2. We assume $\gamma_n + t_n, Y_{n+1} \models complete(tr, p_n, i)$. If $i \notin p_n$ holds, the definition of *complete* requires that $complete(tr, p_n, i - 1)$ holds, so by the induction hypothesis $\mathcal{I}, [0, \sum_{j=1}^n t_j] \models Prefix(i - 1, tr)$ holds. Also $Y_{n+1} \models tr(i).entryEvents$ and $tr(i).allowEmpty$ hold, which yields

$$\mathcal{I}, \left[0, \sum_{j=1}^n t_j \right] \models Prefix(i, tr).$$

Otherwise, $i \in p_n.in$ and by 1. (5.16) holds. We need to show that the duration of the last interval satisfies $len \geq tr(i).timeop \ tr(i).bound$.

If $i \in LB$ and $i \notin p_n.wait$, then $len > tr(i).bound$ by 1(a). If $i \in p_n.wait$ holds, $i \in p_n.gteq$ must hold as otherwise $complete(tr, i, p_n)$ would be *[false]*. Therefore $tr(i).timeop = greaterequal$ and by (c) $len = \gamma_n(c_i) + t_n$. From $\gamma_n + t_n, Y_n \models complete(tr, p_n, i)$ follows $len \geq tr(i).bound$.

If $i \in UB$ and $canseep(tr, p_n, i)$ holds, then by (d) (5.16) holds for some $len < \varepsilon := tr(i).bound$. If $\neg canseep(tr, p_n, i)$ and $i \in p_n.less$, then by definition of *complete* we have $(\gamma_n(c_i) + t_n) < tr(i).bound$.

For $\varepsilon := tr(i).bound - (\gamma_n(c_i) + t_n) > 0$ there is by (e) some $len < \gamma_n(c_i) + t_n + \varepsilon = tr(i).bound$ such that (5.16) holds. The remaining case is $\neg canseep(tr, p_n, i)$ and $i \notin p_n.less$. Then $tr(i).timeop = lessequal$. From the clock invariant $I(p_n)$ we have $\gamma_n(c_i) + t_n \leq tr(i).bound$. By (f) (5.16) holds for $len = \gamma_n(c_i) + t_n \leq tr(i).bound$.

□

Now, we show the reverse direction: If $\mathcal{I}, [0, T] \models Prefix(i, tr)$, the last configuration of the corresponding run satisfies $complete(tr, p_n, i)$. The following auxiliary lemma shows this for the last interval. If

$$\mathcal{I}, [b, e] \models [tr(i).inv] \wedge \bigwedge ev : tr(i).forbidden \bullet \boxminus ev$$

holds, and for the location p_m corresponding to b , $i \in p_m.in$ holds and some timing constraints are fulfilled, then $complete(tr, p_n, i)$ holds for the location p_n corresponding to e .

Lemma 5.16. *Let \mathcal{I} be an interpretation and let*

$$run = \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots, (p_n, Y_n, \beta_n, \gamma_n, t_n), \dots \rangle$$

be a matching run. Let $m \leq n$ be some positions in run and $i \in dom tr$ a trace element. Assume

$$\mathcal{I}, \left[\sum_{j=0}^{m-1} t_j, \sum_{j=0}^n t_j \right] \models [tr(i).inv] \wedge \bigwedge ev : tr(i).forbidden \bullet \boxminus ev \quad (5.19)$$

$$i \in p_m.in \quad (5.20)$$

$$i \in p_m.gteq \Rightarrow \gamma_m(c_i) + \sum_{j=m}^n t_j \geq tr(i).bound \quad (5.21)$$

$$i \in p_m.wait \setminus p_m.gteq \Rightarrow \gamma_m(c_i) + \sum_{j=m}^n t_j > tr(i).bound \quad (5.22)$$

$$i \in UB(tr) \Rightarrow \sum_{j=m}^n t_j \leq tr(i).bound \quad (5.23)$$

$$i \in UB(tr) \wedge \neg canseep(tr, p_m, i) \Rightarrow \gamma_m(c_i) + \sum_{j=m}^n t_j \leq tr(i).bound \quad (5.24)$$

$$i \in p_m.less \Rightarrow \gamma_m(c_i) + \sum_{j=m}^n t_j < tr(i).bound \quad (5.25)$$

Then $(\gamma_n + t_n), Y_{n+1} \models complete(tr, p_n, i)$.

Proof. If $m = n$ then we can show that $complete(tr, p_n, i)$ holds for $(\gamma_n + t_n), Y_{n+1}$: From the premise have $i \in p_n.in$. If $i \in p_n.wait \setminus p_n.gteq$ then $\gamma_n(c_i) + t_n > tr(i).bound$. However, the invariant of p_n states $c_i \leq tr(i).bound$, which is a contradiction. Thus $i \notin p_n.wait \setminus p_n.gteq$ holds. If $i \in p_n.gteq$, then $\gamma_n(c_i) + t_n \geq tr(i).bound$ as required by $complete(tr, p_n, i)$. If $i \in p_n.less$, then $\gamma_m(c_i) + t_n < tr(i).bound$ as demanded by $complete(tr, p_n, i)$. Otherwise $complete(tr, p_n, i) = [\mathbf{true}]$.

If $m < n$, we show that the premises also hold for $m + 1$, then the conclusion follows by induction over $n - m$. The premises (5.19) and (5.23) obviously still hold for $m + 1$.

From (5.19) we have $Y_m \models \wedge \{tr(i).forbidden \bullet \neg ev\}$ and $\beta_{m+1} \models tr(i).inv$. If $i \in UB(tr) \wedge \neg canseep(tr, p_m, i)$, then $\gamma_m(c_i) + t_m < \gamma_m(c_i) + \sum_{j=m}^n t_j \leq tr(i).bound$. So

$$\beta_m, \beta'_{m+1}, Y_m, \gamma_m(c_i) + t_m \models keep(tr, p_m, i). \quad (5.26)$$

With the definition of $guard(tr, p_m, X, p_{m+1})$ thus (5.20) for $m + 1$.

If $i \in LB(tr)$ then also from (5.26) and the definition of $guard$ $c_i \notin X$, so

$$\gamma_m(c_i) + \sum_{j=m}^n t_j = \gamma_{m+1}(c_i) + \sum_{j=m+1}^n t_j$$

Also $i \in p_{m+1}.wait \Rightarrow i \in p_m.wait$ and $i \in p_{m+1}.gteq \Leftrightarrow i \in p_m.gteq \wedge i \in p_{m+1}.wait$ hence (5.21) and (5.22) hold for $m + 1$. For $i \notin LB(tr)$ these implications hold trivially.

If $i \in UB(tr) \wedge \neg canseep(tr, p_{m+1}, i)$, there are two cases: If $c_i \in X$ then $\gamma_{m+1}(c_i) + \sum_{j=m+1}^k t_j = \sum_{j=m+1}^k t_j < \sum_{j=m}^n t_j \leq tr(i).bound$. If $c_i \notin X$, then $\neg canseep(tr, p_m, i)$ holds, thus (5.24). If $i \in p_{m+1}.less$, then $i \in p_m.less$. Thus (5.25) is also valid. \square

Lemma 5.17. *Given an interpretation \mathcal{I} and a matching run*

$$run = \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots \rangle$$

of the power set automaton. Let $i \in dom tr$ be some phase and n some step of the run. If

$$\mathcal{I}, \left[0, \sum_{j=0}^n t_j \right] \models Prefix(i, tr) \quad (5.27)$$

then

$$(\gamma_n + t_n), Y_{n+1} \models complete(tr, p_n, i) \quad (5.28)$$

Proof. This is proven by induction over i . For $i = 0$ we have $Prefix(0, tr) = (\ell = 0)$, so (5.27) never holds (not even for $n = 0$) thus the lemma is true.

Induction step: Assume $i > 0$ and the implication holds for $i-1$. Because of (5.27) and the inductive definition of $Prefix(i, tr)$ there is a time t with $0 \leq t \leq \sum_{j=0}^n t_j$ and

$$\mathcal{I}, [0, t] \models Prefix(i-1, tr) \quad (5.29)$$

$$\begin{aligned} \mathcal{I}, \left[t, \sum_{j=0}^n t_j \right] \models & tr(i).entryEvents \wedge ([tr(i).inv] \vee tr(i).allowEmpty) \wedge \\ & \ell tr(i).timeop tr(i).bound \wedge \\ & \bigwedge ev : tr(i).forbidden \bullet \boxplus ev \end{aligned} \quad (5.30)$$

We now distinguish three cases:

$t = 0$: Our goal is to show that the premises of lemma 5.16 are fulfilled for $m = 0$: From (5.30) follows the premise (5.19).

From equation (5.29) we can conclude $tr(j).allowEmpty$, hence the phase invariant $tr(j).inv$ is **true**. Also $\emptyset \models tr(j).entryEvents$ holds for all $j < i$. This implies $j \in p_0.in$ for all $j < i$ (by induction over j). Because of (5.30) we have $\beta_0 \models tr(i).inv$, so $i \in p_0.in$.

If $tr(i).timeop = greaterequal$ then $i \in p_0.gteq$ and the length of the interval in (5.30), $\sum_{j=0}^n t_j$, is greater or equal $tr(i).bound$. If $tr(i).timeop = greater$ then $i \in p_0.wait \setminus p_0.gteq$ and $\sum_{j=0}^n t_j > tr(i).bound$. With $\gamma_0(c_i) = 0$ we have the corresponding premises (5.21) and (5.22).

If $i \in UB(tr)$ then from (5.30) we have $\sum_{j=0}^n t_j \leq tr(i).bound$.

If $i \in UB(tr) \wedge \neg canseep(tr, p_0, i)$, then $\sum_{j=0}^n t_j \leq tr(i).bound$. Also $i = 1$ (otherwise $canseep(tr, p_0, i)$ holds). Thus $i \in p_0.less \Leftrightarrow tr(i).timeop = less$. Thus if $i \in p_0.less$, then $\sum_{j=0}^n t_j < tr(i).bound$. This gives the premises (5.24) and (5.25).

All premises of lemma 5.16 are fulfilled, which implies (5.28).

$t = \sum_{j=0}^n t_j$: From the induction hypothesis and (5.29) we can conclude

$$(\gamma_n + t_n), Y_{n+1} \models complete(tr, p_n, i - 1).$$

From (5.30) we get $tr(i).allowEmpty$ and $Y_{n+1} \models tr(i).entryEvents$, thus $(\gamma_n + t_n), Y_{n+1} \models complete(tr, p_n, i)$.

$0 < t < \sum_{j=0}^n t_j$: We can assume that there is step m at time t , with $\sum_{j=0}^{m-1} t_j = t$, as otherwise we can insert a stuttering step at exactly this time in the run, which will not affect the property we want to show. We now show that for this m the premises of lemma 5.16 are fulfilled.

From (5.30) we have $Y_{m-1} \models tr(i).entryEvents$ and $\beta_m \models tr(i).inv$, so

$$\beta_{m-1}, \beta'_m, Y_{m-1}, \gamma_{m-1}(c_i) + t_{m-1} \models enter(tr, p_{m-1}, i). \quad (5.31)$$

So from the definition of $guard(tr, p_{m-1}, X, p_m)$ we have $i \in p_m$.

If $i \in LB(tr)$ then obviously $\sum_{j=m}^n t_j \geq tr(i).bound$, so (5.21) holds. If $i \in p_m.wait \setminus p_m.gteq$, then either $c_i \notin X$ and thus $\gamma_m(c_i) + \sum_{j=m}^n t_j > \sum_{j=m}^n t_j \geq tr(i).bound$, or $c_i \in X$ and because of (5.31) also $tr(i).timeop \neq greaterequal$. Therefore $\sum_{j=m}^n t_j > tr(i).bound$. In any case (5.22) holds.

If $i \in UB(tr)$ from (5.30) $\sum_{j=m}^n t_j \leq tr(i).bound$. Thus premise (5.23) holds.

If $i \in UB(tr) \wedge \neg canseep(tr, p_m, i)$ then we have from (5.31) $c_i \in X$, thus $\gamma_m(c_i) = 0$. With (5.30) we have (5.24). Also from (5.31) if $i \in p_m.less$ then $tr(i).timeop = less$, so (5.25) holds.

All premises of lemma 5.16 are fulfilled, thus (5.28) holds.

□

The following corollary concludes this section by summarising the correlation of the DC formula $Prefix(i, tr)$ and the formula $complete(tr, p_n, i)$ applied on the current location p_n of the power set automaton. This result is used in the next chapter to prove the correctness of translation of the DC formulae.

Corollary 5.18. *Given an interpretation \mathcal{I} and a matching run*

$$run = \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots \rangle$$

of the power set automaton. Then for all phases $i \in dom tr$ and for all steps n of the run:

$$\mathcal{I}, \left[0, \sum_{j=0}^n t_j \right] \models Prefix(i, tr)$$

if and only if

$$(\gamma_n + t_n), Y_{n+1} \models complete(tr, p_n, i)$$

Proof. Follows from lemmas 5.15 and 5.17 □

5.3.2. Creating the Accepting Automaton

The automaton created in the last section is deterministic and thus accepts every interpretation. However, when translating the DC part of a CSP-OZ-DC class we need an automaton \mathcal{A} that implements F , i. e., it accepts exactly those interpretations \mathcal{I} that satisfy the DC formula F :

$$\mathcal{I} \models \mathcal{A} \Leftrightarrow \mathcal{I} \models F$$

The formula F is given as a counterexample trace $\neg (\phi_1 \wedge \dots \wedge \phi_n \wedge \mathbf{true})$. Although we require that each counterexample formula ends with a **true** phase, this does not affect the expressiveness of the counterexample formula class. If a formula does not already end with a **true**-phase, it can be added without changing the interpretations that satisfy F :

$$\begin{aligned} \mathcal{I} &\models \neg (\phi_1 \wedge \dots \wedge \phi_n) \\ \Leftrightarrow \forall t : \mathbf{Time} \bullet \mathcal{I}, [0, t] &\not\models (\phi_1 \wedge \dots \wedge \phi_n) \\ \Leftrightarrow \forall t' : \mathbf{Time} \bullet \forall t : \mathbf{Time} \mid t < t' \bullet \mathcal{I}, [0, t] &\not\models (\phi_1 \wedge \dots \wedge \phi_n) \end{aligned}$$

$$\begin{aligned} &\Leftrightarrow \forall t' : \text{Time} \bullet \mathcal{I}, [0, t'] \not\models (\phi_1 \wedge \dots \wedge \phi_n \wedge \mathbf{true}) \\ &\Leftrightarrow \mathcal{I} \models \neg (\phi_1 \wedge \dots \wedge \phi_n \wedge \mathbf{true}) \end{aligned}$$

We build the implementing automaton by constructing the power set automaton \mathcal{A}_{full} and omitting all locations for which the last **true**-phase is active. I. e., the locations p with $\#tr \in p.in$ are omitted from \mathcal{A}_{full} to create the implementing automaton \mathcal{A} . The following corollary proves that the construction is correct.

Lemma 5.19. *Let \mathcal{A}_{full} be the power set automaton and let \mathcal{A} the automaton \mathcal{A}_{full} where all phases with $\#tr \in p.in$ are removed. Then*

$$\mathcal{I}, [0, t] \models F \text{ implies } \mathcal{I}, [0, t] \models \mathcal{A}$$

and

$$\mathcal{I}, [0, t] \models \mathcal{A} \text{ implies } \forall t' : \text{Time} \mid t' < t \bullet \mathcal{I}, [0, t'] \models F$$

Proof. Since the last phase of tr is a **true** phase without any bounds, $\#tr \in p.in$ implies that $complete(tr, p, \#tr) = [\mathbf{true}]$ holds.

For the first implication, assume $\mathcal{I}, [0, t] \models F$. As \mathcal{A}_{full} is deterministic there is a run run of \mathcal{A}_{full} of duration t and length $\#run = n$ that matches \mathcal{I} . This run can also be extended by one step to a run run' of duration $t + t_{n+1}$. By corollary 5.18 we have for n :

$$(\gamma_n + t_n), Y_{n+1} \not\models complete(tr, p_n, \#tr) \quad (5.32)$$

Since F ends with a **true**-phase it also holds for all subintervals $[0, t']$ for all $t' < t$. So equation 5.32 holds for any $i \leq n$. Hence, $\#tr \notin p_i.in$ because otherwise formula $complete(tr, p_i, \#tr)$ is **true**. So all locations p_i that occur in run are also present in \mathcal{A} . Therefore run is a run of \mathcal{A} and thus $\mathcal{I}, [0, t] \models \mathcal{A}$ holds.

For the second implication, assume $\mathcal{I}, [0, t] \models \mathcal{A}$ and run is an accepting run that matches \mathcal{I} . For $t' < t$ insert a stuttering step into run at time t' if there is not already a step at this time. Then there is some $n < \#run$ with $\sum_{j=0}^n t_j = t'$. We now show that

$$\gamma_n + t_n, Y_{n+1} \not\models complete(tr, p_n, \#tr)$$

and thus $\mathcal{I}, [0, t'] \models F$ by corollary 5.18. We show this by contradiction:

Assume that $\gamma_n + t_n, Y_{n+1} \models complete(tr, p_n, \#tr)$ holds. Since p_n is a location of \mathcal{A} , we have $\#tr \notin p_n.in$, so by definition of $complete$ for the

above valuation $complete(tr, p_n, \#tr - 1)$ and $tr(\#tr).entryEvents$ hold. Since $tr(\#tr).inv$ is **true** this means that $enter(tr, p, \#tr)$ holds under the valuations $\beta_n, \gamma_n + t_n, Y_{n+1}, \beta'_{n+1}$. Hence for the next location p_{n+1} we have $\#tr \in p_{n+1}.in$. However, p_{n+1} is a location of \mathcal{A} , so $\#tr \notin p_{n+1}.in$. \square

The following corollary proves the soundness of the translation for Duration Calculus formulae.

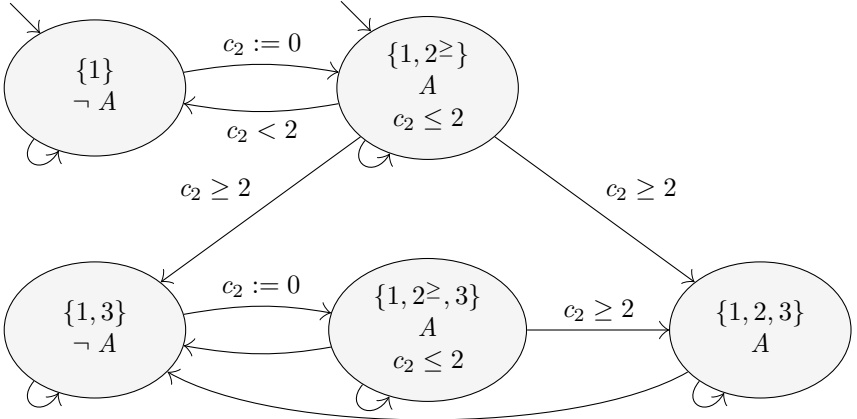
Corollary 5.20 (Soundness of DC Translation). *Let \mathcal{A} be defined as in the previous lemma. Then \mathcal{A} implements F ($\mathcal{A} \hat{=} F$).*

Proof. If $\mathcal{I} \models \mathcal{A}$, then $\mathcal{I}, [0, t] \models \mathcal{A}$ for all t . By the previous lemma, we have $\mathcal{I}, [0, t'] \models F$ for all $t' < t$. Since t is arbitrary, so is t' . Hence, $\mathcal{I} \models F$.

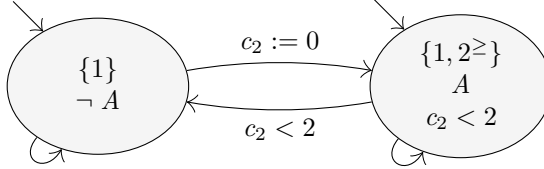
If $\mathcal{I} \models F$, then $\mathcal{I}, [0, t] \models F$ for all t . By the previous lemma $\mathcal{I}, [0, t] \models \mathcal{A}$ for all t . Hence, $\mathcal{I} \models \mathcal{A}$. \square

If the final **true**-phase is immediately preceded by a phase with \geq bound without any events specification in between, i. e., $tr(\#tr).entryEvents = \mathbf{true}$, then the clock invariant for all phases with $\#tr - 1 \in p.gteq$ is changed from $c \leq tr(\#tr - 1).bound$ to $c < tr(\#tr - 1).bound$. This change prevents the automaton from ending in a timelock.

Example 5.21. For the formula $\neg(\mathbf{true} \wedge [A] \wedge \ell \geq 2 \wedge \mathbf{true})$, the full power set automaton is:



If we just omit all locations with $3 \in p.in$ (the second row), the resulting automaton would have a deadlock. If the automaton is in location $\{1, 2^{\geq}\}$ and c_2 is equal to 2, the guard of the only remaining outgoing edge is not satisfied but the location must be left due to the invariant. We therefore have to change the clock invariant according to the algorithm above and the resulting automaton does not allow the clock c_2 reaching the value 2:



In the last example the resulting automaton was less than half the size of the power set automaton. This is typical for the power set construction. Therefore, one should not generate the full power set automaton but already omit the locations with $\#tr \in p.in$, as these are removed anyway.

5.3.3. Case Study: Elevator

The case study of the elevator contains the following two DC-formulae:

$$\neg \diamond (\downarrow passed \wedge \ell \leq 3 \wedge \uparrow passed)$$

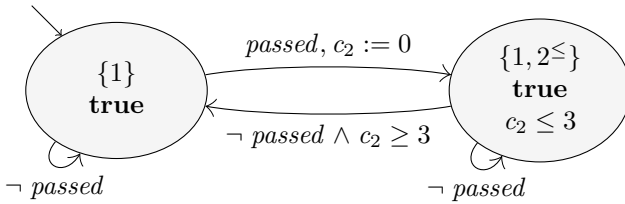
$$\neg \diamond ([current \neq goal] \wedge ([current = goal] \wedge \ell \geq 2 \wedge \boxplus stop))$$

Both formulae are in counterexample trace form (if the eventually operator is expanded) and end with a **true** phase. The expanded form is:

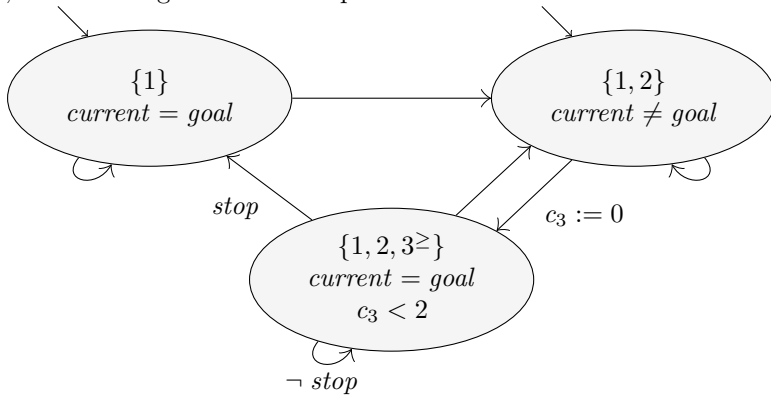
$$\neg (\underbrace{\text{true}}_1 \wedge \downarrow passed \wedge \underbrace{\ell \leq 3}_2 \wedge \uparrow passed \wedge \underbrace{\text{true}}_3)$$

$$\neg (\underbrace{\text{true}}_1 \wedge \underbrace{[current \neq goal]}_2 \wedge \underbrace{([current = goal] \wedge \ell \geq 2 \wedge \boxplus stop)}_3 \wedge \underbrace{\text{true}}_4)$$

The automaton for the first formula is as follows.



In the second DC formula the **true**-phase is preceded by a phase with \geq bound. Therefore, the invariant of the location containing 3^{\geq} is made strict. When applying the power set construction and simplifying the result, the following automaton is produced:



5.4. Discussion and Related Work

The main contribution of this chapter is the translation from a DC counterexample formula to a phase event automaton. The constructed automaton accepts exactly those traces that satisfy the counterexample formula. Thus, it provides an operational semantics for this formula. The automaton works like a *security automaton*, which was introduced by Schneider [Sch00]. A security automaton runs in parallel to the target system, restricting the permitted executions. In this case, an execution is permitted if it satisfies the counterexample formula.

The class of counterexample formulae was chosen, because they are expressive enough to cover all DC implementables (c.f. lemma 2.2). There is a fully developed theory on how implementables can be obtained from general DC specifications [Rav94]. The counterexample formulae are more

general than implementables. In [Tap01] the predicative semantics of phase automata is defined as a conjunction of counterexample formulae. Our extension of counterexample formula allows specification of timing behaviour for events, which is not directly possible with implementables. We do not allow exact bounds in counterexample formulae. However, in most cases where an exact bound $\ell = t$ occurs, it can be replaced by $\ell > t$. This was used to rewrite the synchronisation pattern as a counterexample trace.

In [HO02b] we extended the set of DC implementables by introducing six new patterns to specify properties involving events. For example, the leads-to pattern

$$\lceil Q \rceil \xrightarrow[Y]{t} \lceil R \rceil$$

states that, whenever Q holds for t seconds and no event Y occurs during this interval, Y must occur and the state expression R must be satisfied. All six patterns can be represented as counterexample formulae. For example, the leads-to pattern can be reformulated as

$$\begin{aligned} \lceil Q \rceil \xrightarrow[Y]{t} \lceil R \rceil &\equiv \neg \diamond (\lceil Q \rceil \wedge \ell \geq t \wedge \Box Y \wedge \not\exists Y) \\ &\wedge \neg \diamond (\lceil Q \rceil \wedge \ell \geq t \wedge \Box Y \wedge \lceil \neg R \rceil) \end{aligned}$$

Therefore, our new approach is more general. In the old approach the target automaton created from the CSP-OZ part was modified, to add the timing behaviour. For each pattern a different algorithm exists. This complicates correctness proofs for this construction. In [HO02b] we presented the algorithm for the leads-to pattern, but did not show its correctness.

Tapken [Tap01] gives a translation for each DC implementable formula pattern into phase automata. He provides a translation for each pattern, which he proves separately. His automata are non-deterministic. Thus, for each interpretation satisfying the implementable there is an accepting run, but not every run matching the interpretation can be extended to an accepting run. Tapken does not consider any events.

In [Die99] a translation from a set of DC implementables into PLC-Automata is presented. PLC-Automata [Die01] can be directly translated into programs running on hardware devices called *Programmable Logic Controllers* (PLC). The translation assumes that some observables are inputs, controlled by the environment, and some are outputs, fully controlled by the automaton. The resulting automata are deterministic except for their reaction time, which is bounded by a maximum delay ε . If the set of

implementables is inconsistent or requires an infinitely small reaction time, the algorithm flags an error. All runs of the translated automaton satisfy all implementables, but not all runs that satisfy the implementables can be performed by the automaton.

In [BLR95] the authors give a translation from a Duration Interval Logic (DIL) to linear hybrid automata. They translate formulae of the shape $\varphi = \Box(\Downarrow \pi \wedge (\ell > c) \Rightarrow \phi)$, which they call *Control Design Formulas* (CDF). The formula ϕ is a sequence of phases $\mu_1 \wedge \xi_1 \wedge \dots \wedge \mu_n \wedge \xi_n$, where μ_i is of the form $\Downarrow \pi_1 \wedge \Uparrow \pi_2 \wedge [\pi]$. The formulae ξ_i can restrict the length of the interval, e.g., $\ell \sim t$. Additionally, they may constrain the *duration* of state expressions to $\sum_j (k_j \cdot \int P_j) \sim c$. Note that a CDF φ is equivalent to

$$\neg \Diamond (\Downarrow \pi \wedge \neg (\mu_1 \wedge \xi_1 \wedge \dots \wedge \mu_n \wedge \xi_n) \wedge \ell > c).$$

To avoid the problems with non-deterministic automata the authors required that the formulae are *overlap-free*, which roughly means that adjacent phases cannot hold at the same time. They also require $\mu_1 \wedge \dots \wedge \mu_n$ to be *π -alternation-free*, which means that the event π cannot occur in this formula. These restrictions guarantee that the resulting automaton is deterministic. In CDF, equality in length constraints and constraints on durations $\int P$ are allowed. Thus, not all CDF formulae can be translated by our algorithm. On the other hand, due to the restricting to overlap-free and π -alternation-free formulae, Bouajjani, Lakhnech and Robbana cannot handle the counterexample formulae that occur in the specification of the elevator; these formulae do not have a unique event π that only occurs at the beginning of the formula.

6. Model Checking

Contents

6.1. Implementation of Phase Event Automata	144
6.1.1. Representation of Formulae	144
6.1.2. Computing the Power Set Automaton	146
6.2. Reachability and Phase Event Automata	149
6.3. Translation to Uppaal Automata	151
6.3.1. Case Study: Audio Protocol	154
6.4. A Constraint-based Semantics for PEA	159
6.4.1. Transition Constraint Systems	160
6.4.2. Translation of PEA to TCS	161
6.4.3. Bounded Model Checking	165
6.4.4. ARMC	166
6.4.5. Case Study: Elevator	167
6.5. Related Work	171
6.5.1. Audio Protocol	171
6.5.2. Model Checking Duration Calculus	172

In this chapter, we present different methods to automatically verify CSP-OZ-DC specifications. The methods are based on translation into phase event automata. In the first section, we present the implementation of the power set construction. This is the most difficult part of the translation process. Doing this manually is very error-prone and thus not desirable. Then, we show that the general reachability problem for phase event automata is not decidable. By using abstraction techniques, model checking is still possible in many cases: we present a translation to Uppaal by abstracting from the state variables. Finally, we give a translation into transition constraint systems. These are first order formulae defining the transitions and initial states. There is a model checker ARMC, which can handle infinite state systems given as transition constraint systems. We evaluate this model checker for our phase event automata.

6.1. Implementation of Phase Event Automata

In this section we sketch our implementation of the power set construction. Although complete formulae were given in the last chapter, some further work is necessary to implement an efficient tool that computes the automaton automatically. Furthermore, an efficient representation of the formulae used for guards and invariants in the automaton is necessary.

6.1.1. Representation of Formulae

To implement a tool to create and manipulate phase event automata we need an efficient representation of formulae over states, events and clocks. We use binary decision diagrams (BDD)[Bry86]. These decision diagrams are directed acyclic graphs consisting of terminal and non-terminal nodes. There are only two terminal nodes, namely **true** and **false** that represent themselves. A non-terminal node N is labelled with a boolean variable $N.var$ and has two successor nodes $N.false$ and $N.true$. The formula represented by node N can be calculated recursively as $F(N) = \text{if } N.var \text{ then } F(N.true) \text{ else } F(N.false)$. Since the graphs are acyclic, the recursion terminates.

More precisely we use shared reduced ordered BDD (ROBDD) that have some side conditions. There is a total variable ordering $<$ and for each non-terminal node N and each non-terminal child $C = N.false$ or $C = N.true$, the variables satisfy the ordering $N.var < C.var$. Furthermore, in a graph there are no two nodes representing the same formula. This also implies that the two children of a node are not identical, as otherwise the formula represented by the node is the same as the formula represented by its child.

Only formulae over boolean variables can be directly encoded into BDDs. To encode formulae over clocks an extension of BDDs such as interval decision diagrams (IDD) [ST98] is useful. In IDDs a non-terminal node is labelled with a integer variable and a list of boundaries $b_1 < \dots < b_n$. It has multiple children c_0, \dots, c_n . The child c_i is used if the value of the variable lies in the half-open interval $[b_i, b_{i+1})$, with $b_0 = -\infty$ and $b_{n+1} = +\infty$. These intervals build a disjoint partition of the variable domain. Although the [ST98] defines IDDs only for variables with integer domain, they can be extended in a straightforward manner for variables over reals to represent clock constraints. The boundaries of the intervals need to be represented exactly, so they should be integers or rationals. Furthermore, an additional bit is needed for every boundary to determine whether that value belongs

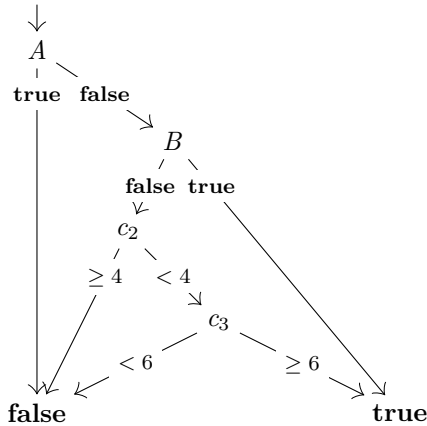


Figure 6.1.: Decision Diagram representation of $\neg A \wedge (B \vee (c_3 \geq 6 \wedge c_2 < 4))$

to the left or right interval.

Unfortunately, the standard BDD libraries are not easy to extend to IDD. The internal data structure allows only two children, the nodes are often only represented as an integer representing the index of the boolean variable. For this reason we needed to implement a new decision diagram package, called *Constraint Decision Diagram* (CDD), that allows nodes with general constraints and more than one successor. This package facilitates creating custom nodes as those necessary for IDD. Different types of nodes can be implemented by extending the `Decision` class using the object-oriented programming paradigm. Besides Boolean decisions, a decision can also be a real variable that is checked against interval bounds. Figure 6.1 gives an example for such a decision diagram. Of course, the CDD package cannot be as efficient as a specialised BDD package, nonetheless it is fast enough to calculate automata with several thousand guards in few seconds.

A common technique for representing decision diagrams is to share sub-diagrams between different formulae. This is extremely useful for the power set construction of section 5.3 where large sub-formulae, e. g., the *keep*, *enter* and *seep* formulae, are needed for many different edges of the automaton. Instead of storing many different copies of the same formula

```

1  private static UnifyHash<CDD> unifyHash;
2  public static CDD create(Decision decision, CDD[] children) {
3      int hashCode = calculateHashCode(decision, children);
4
5      for (cdd : unifyHash.iterateHashCode(hashCode)) {
6          if (cdd.equals(decision, children))
7              return cdd;
8      }
9      cdd = new CDD(decision, children);
10     unifyHash.put(hashCode, cdd);
11     return cdd;
12 }

```

Figure 6.2.: Java code to manage Constraint Decision Diagrams

the previously created formulae are stored and reused whenever they are needed. A prerequisite for sharing these structures is to make them read-only. Instead of manipulating the existing structures, every operation creates a new CDD and returns it. Whenever a new node is needed, it is created by a special function, c. f. figure 6.2 that checks, if the same node already exists. If this is the case the existing node is returned, otherwise a new node is created. This design principle is also known as the fly-weight design pattern [GHJV95].

The function that creates the objects first computes a hash code for the new node in line 3. Then it iterates all previously created nodes from *unifyHash* with the same hash code, c. f. lines 5–8. If a node with the same data is found it is returned in line 7. Otherwise a new node is created, added to *unifyHash*, and returned.

6.1.2. Computing the Power Set Automaton

In section 5.3, we defined the construction of a power set automaton for a given trace. Implementing it directly as stated there is not feasible for the following reasons. For a counterexample trace of length n , the power set automaton has at least 2^n locations in *PowerSet*(*tr*). It has at most 4^n locations if all phases have a greater-equal bound, as each phase can be waiting with or without *gteq* flag, it can be active and not waiting or it is not active at all. The number of edges grows quadratically with the

number of locations and exponentially with the length of the trace. Doing the construction naively would require up to $4^n \cdot 4^n \cdot 2^n = 2^{5n}$ computations of the *guard* function for each location, although most of the computation return an unsatisfiable guard. Even for short traces of length 4 this gives millions of computations.

However, most locations are not reachable and most of the guards are **false**. Thus, they can be omitted. Also when building the accepting automaton from section 5.3.2, further locations can be omitted. Therefore, we build the reachable locations on the fly, starting with the initial locations and only adding a new location if it is reachable from a previously built location. However, from a reachable location, we still need to consider outgoing edges to all possible locations. Hence, we need a way to identify those successor locations for which the guard is satisfiable.

The *guard* function is a conjunction ranging over the number of phases in the trace. The first i conjuncts only depend on the first part of the successor location, i. e., the value of $j \in p'.in, p'.wait, p'.gteq, p'.less$ for $j \leq i$. This suggests to compute the reachable successor location and the corresponding guard function in parallel. A recursive algorithm is sketched in Figure 6.3. The procedure is called with $i = 1$, an empty successor location $p' = \emptyset$, an empty reset set $X = \emptyset$, and *guard* = **true**. The procedure computes the partial guard up to phase i for the cases p' , $p' \cup \{i\}$, $p' \cup \{i \geq\}$, $p' \cup \{i >\}$, and $p' \cup \{i <\}$ and for X and $X \cup \{c[i]\}$. For each combination of successor location and reset set that has a satisfiable guard, the procedure calls itself with increased i . Thus, the *guard* is recursively computed for all possible values of p' and X . If the guard becomes unsatisfiable, e. g., because a phase cannot be entered, this is checked at the beginning of the method and the computation is aborted in lines 2–3. Thus, it only proceeds with values for p' and X that can be extended to reachable successor locations.

In lines 5–9 it is checked if the location has been completely built. In this case, the successor location p' is added, unless it contains the last phase of the trace. Thus, this function computes the accepting automaton instead of the full power set automaton. If the successor location was seen for the first time it is also added to the list of pending locations.

The functions *enter* and *keep* can be computed beforehand. Only *seep* needs to be computed as it depends on the partial p' . It only depends on the first part of the successor location. It is computed in line 13.

The function now recursively calls itself for the different possibilities of p' and X . If $i \notin p'.in$, then the guard function can be simplified to $\neg (keep \vee enter \vee seep) \wedge c_i \notin X$. Therefore, the reset set X does not change and

```

1  procedure buildSuccs(i, p, p', X, guard)
2  if (guard = false)          /* abort if guard is not satisfiable */
3  return;
4
5  if (i > tr.length)          /* check if p' and guard are complete */
6  if (i - 1  $\notin$  p'.in)        /* only add p' if last phase not included */
7  p.addSuccessor(guard, X, p');
8  if (p'  $\notin$  visited  $\cup$  pending)
9  pending := pending  $\cup$  {p'};
10 return;
11
12 /* seep depends on partial successor location */
13 seep[i] := canseep(tr, p', i)  $\wedge$  tr[i].inv
14
15 /* case 1: i  $\notin$  p'.in */
16 buildSuccs(i+1, p, p', X,
17 guard  $\wedge$   $\neg$ (enter[i]  $\vee$  keep[i]  $\vee$  seep[i]));
18
19 guard := guard  $\wedge$  (enter[i]  $\vee$  keep[i]  $\vee$  seep[i]);
20 /* case 2: i  $\in$  p'.in  $\cap$  LB */
21 if (i  $\in$  LB)
22 /* case 2a: c[i]  $\in$  X */
23 if (tr[i].timeop = greaterequal)
24 buildSuccs(i+1, p, p'  $\cup$  { i $\geq$  }, X  $\cup$  {c[i]},
25 guard  $\wedge$   $\neg$  keep[i]  $\wedge$  enter[i]);
26 buildSuccs(i+1, p, p'  $\cup$  { i $>$  }, X  $\cup$  {c[i]}),
27 guard  $\wedge$   $\neg$  keep[i]  $\wedge$   $\neg$  enter[i]);
28 else
29 buildSuccs(i+1, p, p'  $\cup$  { i $>$  }, X  $\cup$  {c[i]},
30 guard  $\wedge$   $\neg$  keep[i], inv);
31 /* case 2b: c[i]  $\notin$  X */
32 if (i  $\in$  p.wait)
33 buildSuccs(i+1, p, p'  $\cup$  (p  $\cap$  { i $>$ , i $\geq$  } ), X,
34 guard  $\wedge$  keep[i]  $\wedge$  c[i] < tr[i].bound);
35 buildSuccs(i+1, p, p'  $\cup$  { i }, X,
36 guard  $\wedge$  keep[i]  $\wedge$  c[i]  $\geq$  tr[i].bound);
37 else
38 buildSuccs(i+1, p, p'  $\cup$  { i }, X, guard  $\wedge$  keep[i]);
39 /* case 3: i  $\in$  p'.in  $\cap$  UB  $\wedge$   $\neg$  canseep(tr, p', i) */
40 else if (i  $\in$  UB  $\wedge$   $\neg$  canseep(tr, p', i))
41 ...
42 else /* i  $\in$  p' */
43 buildSuccs(i+1, p, p'  $\cup$  { i }, X, guard);

```

Figure 6.3.: Pseudo code to compute successors p' and corresponding $guard$

the guard needs to be modified as in line 17. Here is the first recursive call. In all other cases $keep \vee enter \vee seep$ needs to hold, hence $guard$ is modified in line 19. The cases $i \in LB$ and $i \in UB \wedge \neg canseep(tr, p', i)$ are handled separately. This case distinction is also present in the definition of the $guard$ function on page 116. If the phase has an upper bound the cases $c_i \in X$ and $c_i \notin X$ must be handled separately. In the first case we know from the definition of $guard$ that $\neg keep[i]$ must hold and that $i \in p'.wait$. We need to check for $tr[i].timeop = greaterequal$. In that case the successors $p' \cup \{i^{\geq}\}$ and $p \cup \{i^{>}\}$ are both reachable with a different guard. Otherwise only $p' \cup \{i^{>}\}$ is reachable. If $c_i \notin X$ then $keep[i]$ must hold. We need to check for $i \in p.wait$. If $i \notin p.wait$ the successor is never waiting, otherwise it depends on $c[i] < tr[i].bounds$, which is added to the guard. The case $i \in UB \wedge \neg canseep(tr, p', i)$ can be analogously implemented. It is omitted here. In the last case $guard$ requires that $c_i \notin X$ and $i \notin p.wait \cup p.gteq \cup p.gt.less$. Hence, we have only one recursive call.

Figure 6.4 shows the procedure that builds the automaton by repeatedly calling $buildSuccs$ on the reachable locations. Because the $init$ predicate can be seen as a special case of the $guard$ predicate with different values of $enter$ and $keep$ the procedure $buildSuccs$ can be used to determine the initial locations. This is used in lines 4–7 of the algorithm. The procedure also adds the initial locations to the $pending$ set.

In lines 11–20 a location from pending is chosen. It is removed from the $pending$ set and added to the $visited$ set. Its successors are calculated with $buildSuccs$. If new locations are found they are automatically added to the $pending$ set. This is repeated until no new locations are found.

6.2. Reachability and Phase Event Automata

Model checking [CE81, CGP99] is the process of checking whether a model satisfies a specification. A specification can be given as a formula in a temporal logic and the model as an automata. In our context the automata are phase event automata and the specifications are given as Duration Calculus formulae.

It is sometimes possible to reduce the model checking problem into a automata-theoretic problem such as reachability of states or emptiness of the language. For example, for LTL model checking [Var96] of a formula F , the negated formula $\neg F$ can be translated into a Büchi-Automaton. This automaton is put in parallel to the system model. If the parallel product

```

1  pending := ∅;
2
3  /* build initial locations and put them on pending list */
4  for (i : dom(tr))
5      if (i = 0)
6          enter[i] := true
7      else
8          enter[i] := enter[i-1] ∧ tr[i-1].allowEmpty ∧ tr[i].inv;
9          keep[i] := false;
10 buildSuccs(1, init, ∅, ∅, true);
11 while (pending ≠ ∅)
12     choose p ∈ pending
13     pending := pending \ {p}
14     visited := visited ∪ {p}
15
16 /* build successor locations and put them on pending list */
17 for (i : dom(tr))
18     enter[i] := enter(tr, p, i)
19     keep[i] := keep(tr, p, i)
20 buildSuccs(1, p, ∅, ∅, true);

```

Figure 6.4.: Pseudo code to build power set automaton

of these automata has a run, this is a run of the system model satisfying $\neg F$. Hence, the system does not satisfy the specification.

A similar approach can be used for Duration Calculus formulae [Mey05]. Meyer translates a large class of formulae to a variant of phase event automata with bad locations, called test automata. These bad locations are reachable if the negation of the formula is satisfied. The translation is based on the power set construction presented in section 5.3. He uses non-deterministic automata and can thus handle a much larger class of formulae. The idea of test automata is based on the work of Dierks and Lettrari [DL02], who used Uppaal timed automata to check formulae of a graphical specification language based on Duration Calculus.

Therefore, in the following sections we tackle only the reachability problem for phase event automata. Thus, we investigate under which constraints it is decidable, or at least semi-decidable, that a certain location of the automaton is reachable. Decidability of phase event automata depends on the constraint language $\mathcal{L}(V)$. If only finite data types are allowed it is possible to decide reachability with the region construction for timed automata. A practical approach using the model checker Uppaal is described in the next section.

For infinite data types it immediately becomes undecidable. The simplest infinite data type, natural numbers, together with operations increment, decrement, and check for zero leads to undecidability of the reachability problem:

Lemma 6.1. *The reachability problem of phase event automata, with two natural number variables $v_1, v_2 \in V$, where $\mathcal{L}(V)$ contains increment, decrement, and check for zero is undecidable.*

Proof. A phase event automaton with two natural number variables and without any clocks or events is equivalent to a two-counter machine. The reachability problem for two-counter machines is not decidable. \square

The reachability problem is still semi-decidable as long as the satisfiability problem for $\mathcal{L}(V)$ is semi-decidable. A proof for this is given in section 6.4.3.

6.3. Translation to Uppaal Automata

It is possible to translate a phase event automaton into a simple timed automaton by abstracting from states and events. By this an off-the-shelf

model checker for timed automata can be used, such as Uppaal or Kronos.

The automata are abstracted as follows. First the edges are normalised such that each guard is a conjunct of literals. This can be achieved by converting the guards to disjunctive normal form and splitting the edges. Then for all edges (p, g, X, p') of the automata, it is checked whether $s(p) \wedge g \wedge (s(p'))'$ is satisfiable. If it is unsatisfiable the edge is removed without changing the behaviour of the automaton. Likewise initial edges (g, p) are removed if $g \wedge s(p)$ is not satisfiable. Then all unreachable locations are removed. The resulting automaton is then abstracted by removing all literals from the guard except clock constraints and by setting all state invariants to **true**.

The abstraction throws away any synchronisation requirements on state variables and events. Therefore, it should be used on a single automaton. For a net of automata, the full parallel product is computed in advance to preserve synchronisation.

For checking reachability of locations in the automaton, this abstraction is exact under the following condition. The stuttering edge (p, g, \emptyset, p) must satisfy the following formula:

$$(\forall e : A \bullet \neg e) \wedge s(p) \wedge \text{strict}(I(p)) \Rightarrow g \quad (6.1)$$

In this formula the pre-condition $\forall v : V \bullet x = x'$, which is present in definition 4.4 on page 73 (stuttering invariant), is dropped.

Lemma 6.2. *Let $\mathcal{A} = (P, V, A, C, E, s, I, E_0)$ be a phase event automaton and \mathcal{A}' the automaton abstracted according to the algorithm above. If for each location $p : P$ there is a stuttering edge $(p, g, \emptyset, p) \in E$ satisfying (6.1) then a location is reachable in \mathcal{A} if and only if it is reachable in \mathcal{A}' .*

Proof. If a location p_n is reachable in \mathcal{A} , there is a run

$$\text{run} = \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots, (p_n, Y_n, \beta_n, \gamma_n, t_n) \rangle$$

in \mathcal{A} . By definition 4.7, there is an edge $(p_i, g_i, X_i, p_{i+1}) \in E$ for each step i of the run such that $\beta_i, \beta'_{i+1}, \gamma_i + t_i, Y_i \models g_i$. Because also $\beta_i \models s(p_i)$ and $\beta'_{i+1} \models (s(p_{i+1}))'$ hold, the formula $s(p_i) \wedge g \wedge (s(p_{i+1}))'$ is satisfiable. Hence, the edge is still present in \mathcal{A}' (with a weaker guard). Therefore, run is also a run of \mathcal{A}' and p_n is reachable in \mathcal{A}' .

If location p_n is reachable in \mathcal{A}' , then there is a run

$$\text{run}' = \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots, (p_n, Y_n, \beta_n, \gamma_n, t_n) \rangle$$

in \mathcal{A}' . However, this run is not always a run of \mathcal{A} because the guards in \mathcal{A}' are weaker. However, a run of \mathcal{A} can be created as follows: For each step i , there is an edge $(p_i, \bar{g}_i, X_i, p_{i+1})$ in \mathcal{A}' that corresponds to an edge (p_i, g_i, X_i, p_{i+1}) in \mathcal{A} . In \mathcal{A} the formula $s(p_i) \wedge g_i \wedge (s(p_{i+1}))'$ is satisfiable. Hence, there is a satisfying valuation $\beta_i^{(2)} \cup \beta_{i+1}'^{(1)} \cup \gamma_i^1 \cup Y_{i+1}^1$. Because g_i is a conjunct of literals and each literal involving clocks is also in \bar{g}_i , the formula is also satisfied with γ_i instead of γ_i^1 . Analogously, there is an initial edge (g, p_1) in \mathcal{A} and $g \wedge s(p_1)$ is satisfied by some valuation $\beta_1^{(1)}$. Then

$$\begin{aligned} \text{run} = \langle & (p_1, \emptyset, \beta_1^{(1)}, \gamma_1, t_1/2), (p_1, \emptyset, \beta_1^{(2)}, \gamma_1 + t_1/2, t_1/2), \\ & (p_2, Y_2^{(1)}, \beta_2^{(1)}, \gamma_2, t_2/2), (p_2, \emptyset, \beta_2^{(2)}, \gamma_2 + t_2/2, t_2/2), \\ & \dots, (p_n, Y_n^{(1)}, \beta_n^{(1)}, \gamma_n, t_n/2) \rangle \end{aligned}$$

is a run of the automaton \mathcal{A} : Each $\beta_i^{(1)}$ and $\beta_i^{(2)}$ satisfies $s(p_i)$ by construction. For the steps from

$$(p_i, Y_i^{(1)}, \beta_i^{(1)}, \gamma_i, t_i/2) \text{ to } (p_i, \emptyset, \beta_i^{(2)}, \gamma_i + t_i/2, t_i/2),$$

the special stuttering edge satisfying (6.1) is used. Its guard is satisfied because

$$\beta_i^{(1)} \cup (\beta_i^{(2)})' \cup \gamma_i + t_i/2 \cup \emptyset \models (\forall e : A \bullet \neg e) \wedge s(p) \wedge \text{strict}(I(p))$$

holds. For the other steps, the edge (p_i, g_i, X_i, p_{i+1}) that corresponds to the edge $(p_i, \bar{g}_i, X_i, p_{i+1})$ in \mathcal{A}' is used. Thus, the location p_n is reachable in \mathcal{A} . \square

If the condition (6.1) does not hold, i.e., the stuttering edge contains conjuncts such as $v = v'$, the above abstraction is a safe abstraction but not exact. If the state space is finite the automaton can be transformed such that (6.1) holds. For each location l and each possible state β of the state variables a new location (l, β) is created. The state invariant of each location (l, β) fixes the value of the state according to β . Then from the stuttering edge the literals $v = v'$ can be removed as the state invariant already assures that the state variables do not change. Of course, this method is only practical if the state space is small.

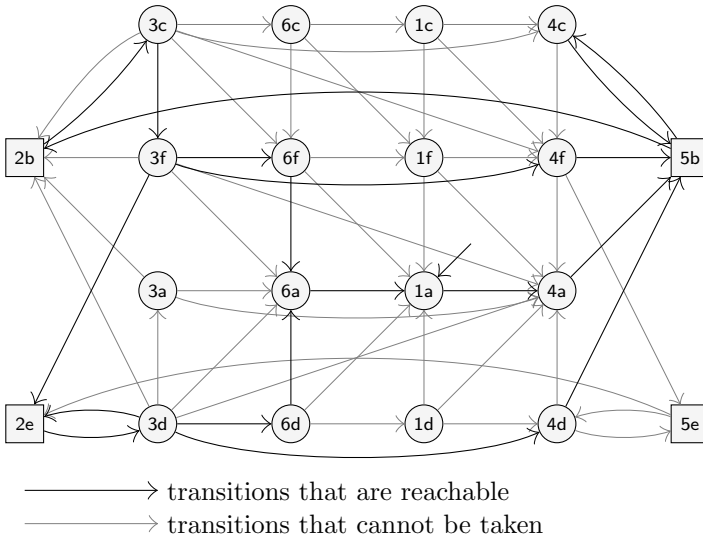


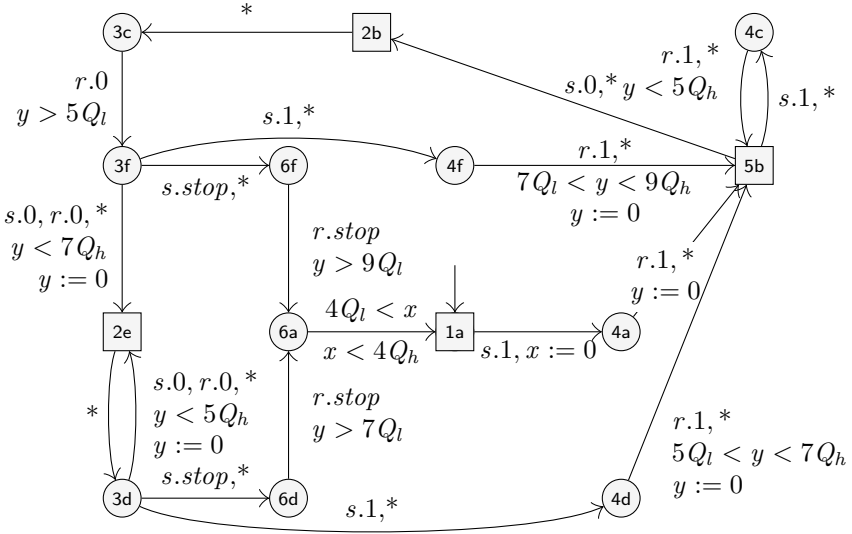
Figure 6.5.: Product automaton for the Audio Protocol

6.3.1. Case Study: Audio Protocol

We apply the abstraction approach to the case study of the Philips Audio Protocol introduced in section 4.6. Our goal is to prove that if the timing uncertainty δ is small enough the protocol is correct, i.e. every event $s.x$ communicated by the sender is eventually followed by an event $r.x$ by the receiver. The sender and receiver have no events in common. They only synchronise on the common state space, namely the *wire* variable. This models the hardware constraints because the Philips hardware components are only connected by this wire.

The first step is to compute the parallel product of sender and receiver automaton. It is depicted in figure 6.5. All locations with an unsatisfiable invariant are already removed. These are those locations where one component demands $wire = \text{false}$ and the other $wire = \text{true}$. In figure 6.5 the labelling of the edges and the stuttering edges were omitted to keep that diagram readable.

In the resulting automaton the variable *wire* can be abstracted. It is an internal variable shared only between sender and receiver. By omitting the



*: $2Q_l < x < 2Q_h, x := 0$

Figure 6.6.: Product automaton without unreachable edges and locations

locations in the product automaton where sender and receiver do not agree on the value of *wire*, these automata synchronise as expected.

For $\delta = 1/20$, which means five percent tolerance of clock drift, many edges and some locations are not reachable. This is because of conflicting conditions for the x and y clock. **Uppaal** can be used to determine the reachable locations. It allows simple LTL queries like $\diamond audio.1f$. To determine the reachable edges one can introduce an auxiliary variable *edge*. This variable is assigned a different value for each edge. The query $\diamond edge = value$ reveals whether the edge is reachable. In the diagram the unreachable edges are depicted by thin lines. Figure 6.6 shows the product automaton where these edges and the unreachable locations are removed. We omitted labelling the edge with forbidden events for simplicity. All events are forbidden unless they are explicitly required. The stuttering edges forbid all events.

The resulting automaton has the nice property that every edge with a send event $s.x$ is followed eventually or immediately by an edge with the

corresponding receive event $r.x$, and there is no other event in between. This can be checked manually on figure 6.6.

We cannot specify the property that the event $r.x$ eventually occurs as counterexample trace, since this is an unbounded liveness property. However, it is possible to formulate it as bounded liveness property. The following formula states that it is not possible that a send event is not followed by a corresponding receive event in an interval larger than $4Q$. Thus, the receive event must occur within that time.

$$\forall x \bullet \neg \diamond (\downarrow s.x \wedge \not\downarrow r.x \wedge \Box r.x \wedge \ell > 4Q)$$

The property that there is no unexpected receive event between a send event and the corresponding receive event can be specified as follows:

$$\forall x \bullet \neg \diamond (\downarrow s.x \wedge \not\downarrow r.x \wedge \Box r.x \wedge \exists y \mid y \neq x \bullet \downarrow r.y)$$

where x and y range over the symbols from $\{0, 1, stop\}$. This formula forbids the event $r.y$ with $y \neq x$ to occur if previously an $s.x$ event occurred and it was not yet received. Furthermore, there should be no send event before the previously sent symbol was properly received, hence:

$$\forall x \bullet \neg \diamond (\downarrow s.x \wedge \not\downarrow r.x \wedge \Box r.x \wedge \ell > 0 \wedge \exists y \bullet \downarrow s.y)$$

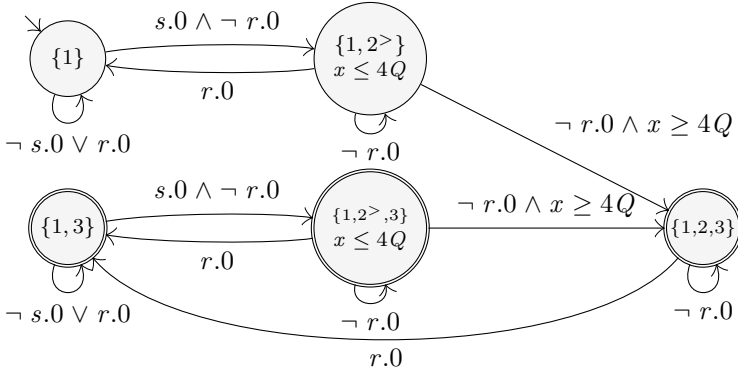
This formula is very similar to the last one. The extra condition $\ell > 0$ is necessary as otherwise the formula would be violated by a simple send event $s.x$. A point interval where $s.x$ occurs satisfies $\downarrow s.x \wedge \Box r.x \wedge \downarrow s.x$.

Because we do not allow quantifiers in counterexample traces we have to write out the possible values for x and y explicitly, which leads to the following nine formulae:

$$\begin{aligned} & \neg \diamond (\downarrow s.0 \wedge \not\downarrow r.0 \wedge \Box r.0 \wedge \ell > 4Q) \\ & \neg \diamond (\downarrow s.1 \wedge \not\downarrow r.1 \wedge \Box r.1 \wedge \ell > 4Q) \\ & \neg \diamond (\downarrow s.stop \wedge \not\downarrow r.stop \wedge \Box r.stop \wedge \ell > 4Q) \\ & \neg \diamond (\downarrow s.0 \wedge \not\downarrow r.0 \wedge \Box r.0 \wedge \uparrow r.1 \vee \downarrow r.stop) \\ & \neg \diamond (\downarrow s.1 \wedge \not\downarrow r.1 \wedge \Box r.1 \wedge \uparrow r.0 \vee \downarrow r.stop) \\ & \neg \diamond (\downarrow s.stop \wedge \not\downarrow r.stop \wedge \Box r.stop \wedge \uparrow r.0 \vee \uparrow r.1) \end{aligned}$$

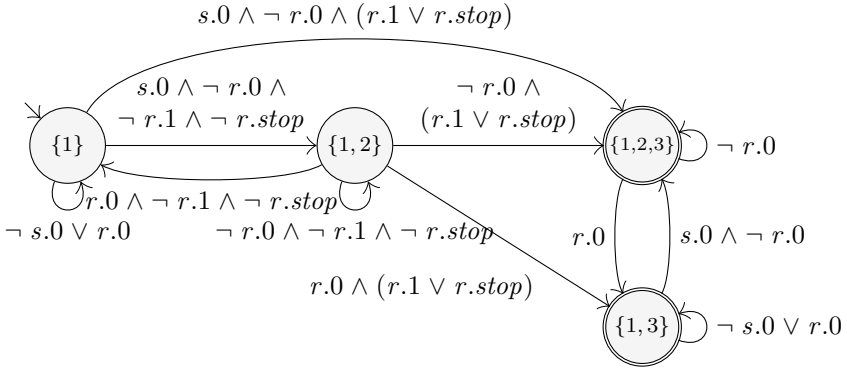
- $$\neg \diamond (\downarrow s.0 \wedge \nexists r.0 \wedge \exists r.0 \wedge \ell > 0 \wedge \downarrow s.0 \vee \downarrow s.1 \vee \downarrow s.stop)$$
- $$\neg \diamond (\downarrow s.1 \wedge \nexists r.1 \wedge \exists r.1 \wedge \ell > 0 \wedge \downarrow s.0 \vee \downarrow s.1 \vee \downarrow s.stop)$$
- $$\neg \diamond (\downarrow s.stop \wedge \nexists r.stop \wedge \exists r.stop \wedge \ell > 0 \wedge \downarrow s.0 \vee \downarrow s.1 \vee \downarrow s.stop)$$

To check against these properties we can build the power set automaton for each of these counterexample traces. The power set automaton for the first formula looks like this:



The automaton starts in location $\{1\}$. If a $s.0$ event occurs the automaton enters $\{1, 2>\}$ uses the clock x to measure the delay $4Q$ for the second phase. Once this time has passed and no $r.0$ event occurred in between, the automaton can enter location $\{1, 2, 3\}$. This location is only reachable if the first DC formula is not valid. It is therefore called a *bad location* and marked with a double border. In the systematic power set construction there are several bad locations and there is no outgoing edge to a good location. This is because phase 3 is a **true**-phase and thus never left.

The automaton for the forth formula is a bit simpler and does not need any clocks.



Again, the locations containing phase 3 are bad locations and marked with a doubled border. Similarly, the other formulae are translated to test automata. The automata for the first three formula has five locations, the other have four locations. However, the cross product of these nine automata already has more than 10000 locations, even if edges with unsatisfiable guards are omitted.

Building the full product automaton and converting it to Uppaal syntax takes 33 seconds on an Athlon 2800+ Processor. The automaton has more than 13729 locations and 76366 transitions. The test automaton has three clocks corresponding to the lower bounds in the first three DC formulae. The sender and receiver each has an additional clock and one additional clock is needed to guarantee that every locations is visited for a non-zero time. This totals to six clocks. After 107 minutes Uppaal proved that the property is satisfied.

Most of the locations of the product automaton are bad locations. Because we are only interested whether *some* bad location is reachable, not *which* bad location is reachable, we can combine all bad locations into a single location, thus drastically reducing the size of the automaton. If we do it after each computation step we can very quickly compute the cross product.

With this optimisation the full cross product of sender, receiver, and all test automata has only 20 locations and 73 transitions. All other locations were combined into a single bad location. Building the automaton takes 0.25 seconds and converting to Uppaal syntax takes 0.15 seconds. The model checker Uppaal needs only 30 milli seconds to prove that the bad location is not reachable.

This proves that our modified protocol is correct for a timing uncertainty

of $1/20$ which is better than the 5% that we had to prove. The value of δ can now be modified to determine the maximum allowed timing uncertainty. For $\delta > 1/19$ the protocol breaks. Analysing the counterexample trace that Uppaal produces shows that the problem is in the short radio silence between two adjacent messages. When the sender sends two message directly after each other and its clock is going very fast, it can send the first bit of the second message before the receiver has detected the stop bit. When extending the gap to $5Q$ (the original specification demanded a gap of $36Q$) we can even increase the uncertainty to $1/17$. The same uncertainty was proven in [BPV94] and [BGK⁺96]. Note that the protocol is still correct for minimum gap $4Q$ and uncertainty $1/17$. However, in that case the sender can issue two send events, before they are received by the receiver. This results in a trace *s.stop, s.1, r.stop, r.1*, which is not allowed by the simple DC formulae we presented above.

For $\delta > 1/17$ the protocol is broken. The model checker gives an error trace showing the locations of the sender and receiver and the test automata. In this trace the sender sends the bits 1,0,1 with the minimal duration of $2Q(1 - \delta)$ for each bit slot. The interval between the rising edges sums up to $8Q(1 - \delta)$, which is less than $7Q(1 + \delta)$ for $\delta > 1/17$. So the receiver can decode this sequence as 1,0,0. This error trace is the same as in [BPV94] and [BGK⁺96].

This case study showed that the Uppaal method is feasible. But it also shows that calculating the cross product in advance can lead to large automaton description. In this case study, only few locations are not bad locations, so it makes sense to combine all bad locations into a single one. Checking against multiple duration calculus formulae at the same time, increases the number of bad locations, but if the bad locations are combined this does not result in any performance penalties.

6.4. A Constraint-based Semantics for PEA

A completely different approach to model check phase event automata utilises discrete transition system. Although this only allows discrete steps, it is still possible to represent dense time using an “old-fashioned recipe for real-time” [AL92, Lam94]. The runs are described by sequences of states, where each state gives the values of all variables for a given time interval. One variable *len* denotes the duration of the time interval. Events are represented by changes of Boolean variables as in section 2.3.5. We

translate the automata into discrete transition systems (with constraints) in such a way that the transition system generates as runs exactly the above sequences of interval states.

This section proceeds as follows: first we introduce transition constraint systems and their runs. We show how parallel composition is defined for them. Then we give the translation from phase event automata into transition constraint systems and show the correctness of the translation by giving a direct relation between runs of phase event automata and runs of the transition constraint system. Then we apply this translation on some examples. We will close this sections with some results of the model checker ARMC for transition constraint systems.

6.4.1. Transition Constraint Systems

A transition constraint system is described by two formulae. One defines the initial state and the other defines the transition relation:

Definition 6.3. A *transition constraint system* (TCS)

$$\mathcal{T} = (Var, Init, Trans)$$

consists of

- $Var \subseteq \mathcal{V}$ is a finite set of unprimed (state) variables,
- $Init : \mathcal{L}(Var)$ a initial (state) constraint, and
- $Trans : \mathcal{L}(Var \cup Var')$ a (transition) constraint.

Informally, the transition constraint system starts in a state satisfying $Init$. It can do transitions according to the transition constraint $Trans$, which is a relation between pre-state (valuations of the unprimed variables) and post-states (valuations of the primed variables).

Let $\mathcal{T} = (Var, Init, Trans)$ be a TCS. A *state* of \mathcal{T} is a Var -valuation α . Taking states as vertices, the TCS \mathcal{T} can be viewed as a (potentially infinite) directed graph (where two states are connected by an edge if they satisfy the respective transition constraint). This graph gives rise to the usual notions of run and reachable state. Formally, a *run* of \mathcal{T} is a sequence of states $\langle \alpha_1, \dots, \alpha_n \rangle$ such that

1. $\alpha_1 \models Init$, and

2. for all $i \geq 0$, $\alpha_i, \alpha'_{i+1} \models Trans$.

We call a state α *reachable* if there is a run $\langle \alpha_1, \dots, \alpha_n \rangle$ of \mathcal{T} such that $\alpha = \alpha_n$. By $Reach(\mathcal{T})$, we denote the set of reachable states of \mathcal{T} .

6.4.2. Translation of PEA to TCS

The basic idea in translating PEA to TCS is to introduce a global variable len of type **Time** representing the amount of time that the automaton spent in the current state. Furthermore for any state variable, event, and clock variable a corresponding variable of the TCS is introduced. Finally, there is a variable $p_{\mathcal{A}}$ that denotes the currently active location of the automaton. Each edge is translated separately into a transition constraint and these are then combined by disjunction to produce the complete transition system. Furthermore we need to represent the time elapse transitions that are implicit in the phase event automata model.

Definition 6.4. The translation of a phase event automaton

$$\mathcal{A} = (P, V, A, C, E, s, I, E_0)$$

into a transition constraint system $\mathcal{T}(\mathcal{A}) = (Var, Init, Trans)$ is given by:

$$Loc = P \tag{6.2}$$

$$Var = V \cup A \cup C \cup \{len, p_{\mathcal{A}}\} \tag{6.3}$$

$$Init = \bigvee (g, p_0) \in E_0, \bullet len > 0 \wedge g \wedge \tag{6.4}$$

$$\left(\bigwedge c \in C \bullet c = len \right) \wedge p_{\mathcal{A}} = p_0 \wedge s(p_0) \wedge I(p_0)$$

$$Trans = \bigvee (q_1, g, X, q_2) \in E \bullet len' > 0 \wedge p_{\mathcal{A}} = q_1 \wedge p'_{\mathcal{A}} = q_2 \wedge g \wedge \tag{6.5}$$

$$\bigwedge_{c \in X} c' = len' \wedge \bigwedge_{c \in C \setminus X} c' = c + len' \wedge s'(q_2) \wedge I'(q_2)$$

The formula *Init* demands that there is an initial edge $(g, p_0) \in E_0$ such that g , the state and clock invariant of p_0 , and $p_{\mathcal{A}} = p_0$ hold. Furthermore, len , the duration the automaton stays in the first location, is positive and all clocks are initialised to this value. Thus the passing of the time len in the initial location is already modelled by the *Init* formula.

The formula *Trans* demands that there is an edge $(q_1, g, X, q_2) \in E$ by which location q_1 is left. The state and clock invariant of q_1 were

already checked when entering the location, therefore the formula only checks the guard g and the invariants for q_2 . Again, len' denotes the duration the automaton stays in q_2 . Therefore all clocks that are not reset are incremented by len' and the other clocks are set to len' .

We show that the translation $\mathcal{T}(\mathcal{A})$ of a PEA \mathcal{A} preserves the semantics in the sense that there is a correspondence between the runs of \mathcal{A} and $\mathcal{T}(\mathcal{A})$.

Definition 6.5. Given a run $r = \langle \alpha_1, \dots, \alpha_n \rangle$ of the TCS $\mathcal{T}(\mathcal{A})$, we define the sequence $r_{\mathcal{A}} = \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots, (p_n, Y_n, \beta_n, \gamma_n, t_n) \rangle$ such that for all $i \geq 1$,

- $p_i = \alpha_i(\mathbf{p}_{\mathcal{A}})$,
- $\beta_i = V \triangleleft \alpha_i$,
- $\gamma_i(c) = \alpha_i(c) - \alpha_i(\text{len})$ for all $c \in C$,
- $t_i = \alpha_i(\text{len})$, and
- $Y_1 = \emptyset$ and $Y_i = \{e \in A \mid \alpha_{i-1}(e)\}$ for $i > 1$

As the following theorem shows, this translation maps runs of the TCS $\mathcal{T}(\mathcal{A})$ to runs of the PEA \mathcal{A} . Furthermore, the translation is surjective, so for every run of \mathcal{A} there is a corresponding run of $\mathcal{T}(\mathcal{A})$.

Theorem 6.6. *Let \mathcal{A} be a PEA and $\mathcal{T}(\mathcal{A})$ its TCS translation.*

1. *For all runs r of $\mathcal{T}(\mathcal{A})$, $r_{\mathcal{A}}$ is a run of \mathcal{A} .*
2. *For every run \hat{r} of \mathcal{A} there is a run r of $\mathcal{T}(\mathcal{A})$ such that $r_{\mathcal{A}} = \hat{r}$.*

Proof. 1. Let $r = \langle \alpha_1, \dots, \alpha_n \rangle$ be a run of $\mathcal{T}(\mathcal{A})$ and $r_{\mathcal{A}}$ the corresponding translation of that run. We need to show that this is indeed a run of \mathcal{A} by checking definition 4.7.

From 6.5 we have immediately $Y_1 = \emptyset$.

Because $\alpha_1 \models \text{Init}$ holds and from (6.4), there is an initial edge $(g, p_0) \in E_0$, such that

$$\alpha_1 \models \text{len} > 0 \wedge g \wedge \left(\bigwedge c \in C \bullet c = \text{len} \right) \wedge \mathbf{p}_{\mathcal{A}} = p_0 \wedge s(p_0) \wedge I(p_0) \quad (6.6)$$

holds. From definition 6.5 of $r_{\mathcal{A}}$ we have $p_1 = \alpha_1(\mathbf{p}_{\mathcal{A}}) = p_0$ and $\beta_1 = (V \triangleleft \alpha_1) \models g$, which is the second condition in 4.7

From $\alpha_1 \models c = \text{len}$ we have $\gamma_1(c) = \alpha_1(c) - \alpha_1(\text{len}) = 0$ for all $c \in C$.

For $1 \leq i < n$ we have $\alpha_i, \alpha'_{i+1} \models \text{Trans}$. Because of (6.5) there is an edge $(q_1, g, X, q_2) \in E$ such that

$$\begin{aligned} \alpha_i, \alpha'_{i+1} \models \text{len}' > 0 \wedge \mathbf{p}_{\mathcal{A}} = q_1 \wedge \mathbf{p}'_{\mathcal{A}} = q_2 \wedge g \wedge \\ \bigwedge_{c \in X} c' = \text{len}' \wedge \bigwedge_{c \in C \setminus X} c' = c + \text{len}' \wedge s'(p_2) \wedge I'(p_2) \end{aligned} \quad (6.7)$$

From definition 6.5 we have $p_i = \alpha_i(\mathbf{p}_{\mathcal{A}}) = q_1$ and $p_{i+1} = \alpha_{i+1}(\mathbf{p}_{\mathcal{A}}) = q_2$.

Looking just at the first, the third and the last two conjuncts and omitting the primes we get $\alpha_{i+1} \models \text{len} > 0 \wedge s(\mathbf{p}_{\mathcal{A}}) \wedge I(\mathbf{p}_{\mathcal{A}})$. For α_1 the same follows from equation (6.6). This implies $t_i = \alpha_i(\text{len}) > 0$ for all $i \in 1..n$. From definition 6.5 we have $p_i = \alpha_i(\mathbf{p}_{\mathcal{A}})$ hence $\beta_i = V \triangleleft \alpha_i \models s(p_i)$. Also from definition 6.5 we have $(\gamma_i + t_i)(c) = \gamma_i(c) + \alpha_i(\text{len}) = \alpha_i(c)$, hence $\gamma_i + t_i = C \triangleleft \alpha_i \models I(p_i)$.

From $\alpha_i, \alpha'_{i+1} \models g$ we have $\beta_i, \beta'_{i+1} \gamma_i + t_i, Y_{i+1} \models g$, because $\beta_i = V \triangleleft \alpha_i$, $\beta'_{i+1} = V \triangleleft \alpha'_{i+1}$ and $\gamma_i + t_i = C \triangleleft \alpha_i$ and $\chi_{Y_{i+1}} = E \triangleleft \alpha_i$. For $c \in X$ we have $\gamma_{i+1}(c) = \alpha_{i+1}(c) - \alpha_{i+1}(\text{len}) = 0$. For $c \notin X$ we have $\gamma_{i+1}(c) = \alpha_{i+1}(c) - \alpha_{i+1}(\text{len}) = \alpha_i(c) = \gamma_i(c) + t_i$. Hence, $\gamma_{i+1} = (\gamma_i + t_i)[X := 0]$.

2. Let $\hat{r} = \langle (p_1, Y_1, \beta_1, \gamma_1, t_1), \dots, (p_n, Y_n, \beta_n, \gamma_n, t_n) \rangle$ be a run of \mathcal{A} . Then define α_i for $i \in 1..n$ as follows:

$$\begin{aligned} \alpha_i(\mathbf{p}_{\mathcal{A}}) &= p_i \\ \alpha_i(\text{len}) &= t_i \\ \alpha_i(v) &= \beta_i(v) && \text{for all } v \in V \\ \alpha_i(c) &= \gamma_i(c) + t_i && \text{for all } c \in C \\ \alpha_i(e) &= \mathbf{true} && \text{for all } e \in Y_{i+1} \\ \alpha_i(e) &= \mathbf{false} && \text{for all } e \in A \setminus Y_{i+1} \end{aligned}$$

Set $r = \langle \alpha_1, \dots, \alpha_n \rangle$. It is straightforward to check that $r_{\mathcal{A}} = \hat{r}$. We show that r is indeed a run of $T(\mathcal{A})$. Since \hat{r} is a run of \mathcal{A}

there is by definition 6.5 an initial edge $(g, p_1) \in E_0$, such $\beta_1 \models g$. Furthermore, $\beta_1 \models s(p_1)$, $\gamma_1 + t_1 \models I(p_1)$ by definition 6.5. Also $\gamma_1(c) = 0$ for all $c \in C$, hence $\alpha_1(c) = t_1 = \alpha_1(\text{len})$. Therefore, $\alpha_1 \models (\bigwedge c \in C \bullet c = \text{len})$. Also $t_1 > 0$ hence

$$\alpha \models \text{len} > 0 \wedge g \wedge (\bigwedge c \in C \bullet c = \text{len}) \wedge \mathbf{p}_{\mathcal{A}} = p_1 \wedge s(p_1) \wedge I(p_1)$$

This implies $\alpha \models \text{Init}$.

For each step i there is by definition 6.5 an edge $(p_i, g, X, p_{i+1}) \in E$ with $\beta_i, \beta'_{i+1}, \gamma_i + t_i, Y_{i+1} \models g$ and $\gamma_{i+1} = (\gamma_i + t_i)[X := 0]$. Hence, $\alpha_i, \alpha'_{i+1} \models g$ and

$$\alpha_{i+1}(c) = \begin{cases} t_{i+1} & c \in X \\ \alpha_i(c) + t_{i+1} & c \notin X \end{cases}$$

From definition 6.5 we further have $t_{i+1} > 0$, $\beta_{i+1} \models s(p_{i+1})$ and $\gamma_{i+1} + t_{i+1} \models I(p_{i+1})$ hence

$$\begin{aligned} \alpha_i, \alpha'_{i+1} \models \text{len} > 0 \wedge \mathbf{p}_{\mathcal{A}} = p_i \wedge \mathbf{p}'_{\mathcal{A}} = p_{i+1} \wedge g \wedge \\ \bigwedge_{c \in X} c' = \text{len}' \wedge \bigwedge_{c \in C \setminus X} c' = c + \text{len}' \wedge s'(p_{i+1}) \wedge I'(p_{i+1}) \end{aligned}$$

Hence, $\alpha_i, \alpha'_{i+1} \models \text{Trans}$. Therefore, r is indeed a run of $\mathcal{T}(\mathcal{A})$. \square

Parallel Product of Transition Constraint Systems

The parallel product of phase event automata has a simple corresponding operation on TCS: conjunction of formulae. The exact definition also needs to take the set of variables into account:

Definition 6.7. The *parallel composition* $\mathcal{T}_1 \parallel \mathcal{T}_2$ of two transition constraint systems \mathcal{T}_1 and \mathcal{T}_2 (where $\mathcal{T}_i = (\text{Var}_i, \text{Init}_i, \text{Trans}_i)$, $i = 1, 2$) is the TCS

$$\mathcal{T} = (\text{Var}_1 \cup \text{Var}_2, \text{Init}_1 \wedge \text{Init}_2, \text{Trans}_1 \wedge \text{Trans}_2)$$

The following result similar to lemma 4.8 holds for the parallel composition of transition constraint systems:

Lemma 6.8. *For TCS \mathcal{T}_1 and \mathcal{T}_2 , $\langle \alpha_1, \dots, \alpha_n \rangle$ is a run of $\mathcal{T}_1 \parallel \mathcal{T}_2$ if and only if $\langle \text{Var}_1 \triangleleft \alpha_1, \dots, \text{Var}_1 \triangleleft \alpha_n \rangle$ and $\langle \text{Var}_2 \triangleleft \alpha_1, \dots, \text{Var}_2 \triangleleft \alpha_n \rangle$ are runs of \mathcal{T}_1 and \mathcal{T}_2 , respectively.*

Proof. Let $\langle \alpha_1, \dots, \alpha_n \rangle$ be a run of TCS \mathcal{T} . Since $\alpha_1 \models \text{Init}_1 \wedge \text{Init}_2$ and Init_1 only references variables from Var_1 , we have $(\text{Var}_1 \triangleleft \alpha_1) \models \text{Init}_1$. Likewise from $\alpha_i, \alpha'_{i+1} \models \text{Trans}_1 \wedge \text{Trans}_2$ we have $(\text{Var}_1 \triangleleft \alpha_i), (\text{Var}_1 \triangleleft \alpha'_i) \models \text{Trans}_1$. So $\langle \text{Var}_1 \triangleleft \alpha_1, \dots, \text{Var}_1 \triangleleft \alpha_n \rangle$ is indeed a run of Trans_1 . Analogously for Trans_2 .

Now let $\langle \text{Var}_1 \triangleleft \alpha_1, \dots, \text{Var}_1 \triangleleft \alpha_n \rangle$ and $\langle \text{Var}_2 \triangleleft \alpha_1, \dots, \text{Var}_2 \triangleleft \alpha_n \rangle$ be runs of Trans_1 and Trans_2 , respectively. From $\text{Var}_1 \triangleleft \alpha_1 \models \text{Init}_1$ and $\text{Var}_2 \triangleleft \alpha_1 \models \text{Init}_2$ we have $\alpha_1 \models \text{Init}_1 \wedge \text{Init}_2$. Similarly, for the transition relation $\text{Trans}_1 \wedge \text{Trans}_2$. \square

6.4.3. Bounded Model Checking

If the satisfiability problem for the language $\mathcal{L}(\text{Var})$ is semi-decidable, the reachability problem for the transition constraint system is also semi-decidable. The simplest algorithm works by *unrolling* the TCS upto a pre-determined depth. This approach, called *bounded model checking* [CBRZ01, BCCZ03], is useful to quickly find short counterexamples.

The function $\text{unroll}(\mathcal{T}, k)$ computes the set of states reachable in k steps of the transition system. It can be computed by the following recursive equation:

$$\text{unroll}(\mathcal{T}, 0) = \text{Init}[v_{(0)}/v] \quad (6.8)$$

$$\text{unroll}(\mathcal{T}, k+1) = \text{unroll}(\mathcal{T}, k) \wedge \text{Trans}[v_{(k)}/v][v_{(k+1)}/v'] \quad (6.9)$$

Here the renaming $[v_{(k)}/v]$ should denote that *all* variables in Var are renamed by decorating them with the index (k). The variables $v_{(i)}$ in the above formula represent the state after exactly i steps. The states reachable in zero steps are the initial states, which is stated by (6.8). The next equation computes the states reachable after exactly $k+1$ steps: First, the reachable states after k steps are computed. Then the transition relation Trans is applied on these states. The variables of the pre-states are renamed the $v_{(k)}$ in the transition relation and the variables of the post-state are renamed to $v_{(k+1)}$. A non-deterministic algorithm that semi-decides reachability for a transition constraint system is given in figure 6.7.

Input: TCS \mathcal{T} , state predicate ϕ .

Output: **true**, if a state satisfying ϕ is reachable.

1. Guess unrolling depth k . This corresponds to the number of steps needed to reach a state satisfying ϕ .
2. Compute the unrolled system $F := \text{unroll}(\mathcal{T}, k)$.
3. Return **true** if $F \wedge \phi[v_{(k)}/v]$ is satisfiable.

Figure 6.7.: Algorithm to semi-decide reachability for a TCS

6.4.4. ARMC

The bounded model checking technique cannot be used to prove that the system is safe. It can only be shown that no counterexample of a length upto k exists. However, there is no guarantee that no larger counterexamples exists.

A different approach to prove safety properties of infinite state systems is abstraction by over-approximation. If the abstraction is too coarse, it will produce *spurious* counterexamples in the abstract system that cannot be refined for the concrete system. On the other hand, if no counterexample is found in the abstract system, the concrete system is safe, too. When a spurious counterexample is found it gives a hint to which parts of the abstraction are too coarse. This hint can be used to produce a finer abstraction that will not produce the same spurious counterexample. This technique is called *abstraction refinement model checking*.

We use the abstraction refinement model checker ARMC [Ryb02]. This model checker uses predicate abstraction and interpolants on the counterexample to generate new predicates in the refinement cycle. The current implementation supports infinite state systems involving linear real or integer arithmetic. As the reachability problem for transition constraint system is only semi-decidable, it is impossible to guarantee that ARMC will terminate. The following three cases are possible.

- ARMC finds an abstract counterexample that can be transferred to the concrete system. In this case, the property is not satisfied and ARMC outputs the counterexample.
- A safe abstraction of the system is found, that contains no counterexample for the property. In this case, ARMC assures that the property

is satisfied. It also outputs the predicates used for abstracting the system.

- ARMC enters an infinite abstraction refinement loop. All abstractions contain spurious counterexamples. New abstraction predicates are added to exclude this particular counterexample. However, another larger spurious counterexample is still present in the abstract system. ARMC will never terminate.

The transition constraint systems used in ARMC differ from definition 6.3. They have an explicit notion of program counters; thus there is no need to represent them as state variables. Instead of a single initial predicate there is a set of tuples $(p_0, Init)$, where p_0 is an initial location and $Init$ a predicate describing the state. A system state is represented by a tuple (p, α) , where p is a location and α a state valuation.

Unfortunately, ARMC only works on disjunctive normal form formulae. When working on a single automaton the translation given in section 6.4.2 almost produces a formula of this form. Only edges with guards that contain disjunctions need to be transformed to disjunctive normal form and split into multiple edges, one for each disjunct. For disjunction free guards the formulae $Init$ and $Trans$ are already in disjunctive normal form.

When checking several phase event automata that run in parallel this restriction of ARMC poses a problem, though. The parallel composition for TCS combines the transition guards with conjunction. When using the distributive law to convert this to disjunctive normal form we get an exponential blow up. This gives the same result as computing the parallel product automaton directly and translating this automaton to a TCS.

6.4.5. Case Study: Elevator

We apply the above algorithm to the case study for the elevator introduced in section 3.2.2. The goal is to prove the following safety property: all reachable states of the system satisfy the invariant

$$Min \leq current \leq Max . \tag{6.10}$$

This invariant states that the current floor is always between minimum and maximum floor. It holds because the elevator must stop when it reaches the goal floor due to the real-time requirements. The invariant depends also on the Z and CSP part. Thus, we need to translate the full specification into

a transition constraint system. The phase event automata of the elevator specification were already given in sections 5.1, 5.2, and 5.3.3.

For the CSP automaton we use the variable p_C to store the current location. For simplicity we enumerate the locations in figure 5.1 on page 99 from left to right as 0, 1, 2. The resulting transition constraint system is as follows:

$$\begin{aligned}
 \text{Init}_{CSP} &= p_C = 0 \\
 \text{Trans}_{CSP} &= p_C = 0 \wedge \neg \text{newgoal} \wedge \neg \text{start} \wedge \neg \text{stop} \wedge \neg \text{passed} \wedge p'_C = 0 \\
 &\quad \vee p_C = 0 \wedge \text{newgoal} \wedge \neg \text{start} \wedge \neg \text{stop} \wedge \neg \text{passed} \wedge p'_C = 1 \\
 &\quad \vee p_C = 1 \wedge \neg \text{newgoal} \wedge \neg \text{start} \wedge \neg \text{stop} \wedge \neg \text{passed} \wedge p'_C = 1 \\
 &\quad \vee p_C = 1 \wedge \neg \text{newgoal} \wedge \text{start} \wedge \neg \text{stop} \wedge \neg \text{passed} \wedge p'_C = 2 \\
 &\quad \vee p_C = 2 \wedge \neg \text{newgoal} \wedge \neg \text{start} \wedge \neg \text{stop} \wedge \neg \text{passed} \wedge p'_C = 2 \\
 &\quad \vee p_C = 2 \wedge \neg \text{newgoal} \wedge \neg \text{start} \wedge \neg \text{stop} \wedge \text{passed} \wedge p'_C = 2 \\
 &\quad \vee p_C = 2 \wedge \neg \text{newgoal} \wedge \neg \text{start} \wedge \text{stop} \wedge \neg \text{passed} \wedge p'_C = 0
 \end{aligned}$$

The Z automaton on page 102 has only a single location. Therefore, we do not need a variable to store the current location. The resulting transition constraint system is as follows:

$$\begin{aligned}
 \text{Init}_Z &= \text{current} = \text{goal} = \text{Min} \\
 \text{Trans}_Z &= \neg \text{newgoal} \wedge \neg \text{start} \wedge \neg \text{stop} \wedge \neg \text{passed} \\
 &\quad \wedge \text{current}' = \text{current} \wedge \text{goal}' = \text{goal} \wedge \text{dir}' = \text{dir} \\
 &\quad \vee \text{newgoal} \wedge \neg \text{start} \wedge \neg \text{stop} \wedge \neg \text{passed} \\
 &\quad \wedge \text{Min} \leq \text{goal}' \leq \text{Max} \wedge \text{goal}' \neq \text{current} \\
 &\quad \wedge \text{current}' = \text{current} \wedge \text{dir}' = \text{dir} \\
 &\quad \vee \text{start} \wedge \neg \text{newgoal} \wedge \neg \text{stop} \wedge \neg \text{passed} \\
 &\quad \wedge (\text{goal} > \text{current} \Rightarrow \text{dir}' = 1) \\
 &\quad \wedge (\text{goal} < \text{current} \Rightarrow \text{dir}' = -1) \\
 &\quad \wedge \text{current}' = \text{current} \wedge \text{goal}' = \text{goal} \\
 &\quad \vee \text{stop} \wedge \neg \text{newgoal} \wedge \neg \text{start} \wedge \neg \text{passed} \\
 &\quad \wedge \text{goal} = \text{current} \\
 &\quad \wedge \text{current}' = \text{current} \wedge \text{goal}' = \text{goal} \wedge \text{dir}' = \text{dir} \\
 &\quad \vee \text{passed} \wedge \neg \text{newgoal} \wedge \neg \text{start} \wedge \neg \text{stop} \\
 &\quad \wedge \text{current}' = \text{current} + \text{dir}
 \end{aligned}$$

$$\wedge \text{goal}' = \text{goal} \wedge \text{dir}' = \text{dir}$$

The phase event automata for the two DC formulae of the specification are depicted on page 139. The first has two locations $\{1\}$, $\{1, 2^{\leq}\}$ that we enumerate as 0 and 1. Initially, the automaton starts in location 0 and resets clock c_2 . By convention, the variable c_2 of the transition constraint is set to the value of clock c_2 after the first delay len has passed, therefore it is initialised to len . Also when the clock is reset by some transition it is set to len . Otherwise it is incremented by len . The resulting transition constraint system comprises these formulae:

$$\begin{aligned} \text{Init}_{DC1} &= \mathbf{p}_{D1} = 0 \wedge c_2 = \text{len} \\ \text{Trans}_{DC1} &= \mathbf{p}_{D1} = 0 \wedge \neg \text{passed} \wedge c'_2 = c_2 + \text{len} \wedge \mathbf{p}'_{D1} = 0 \\ &\vee \mathbf{p}_{D1} = 0 \wedge \text{passed} \wedge c'_2 = \text{len} \wedge c'_2 \leq 3 \wedge \mathbf{p}'_{D1} = 1 \\ &\vee \mathbf{p}_{D1} = 1 \wedge \neg \text{passed} \wedge c'_2 = c_2 + \text{len} \wedge c'_2 \leq 3 \wedge \mathbf{p}'_{D1} = 1 \\ &\vee \mathbf{p}_{D1} = 1 \wedge \neg \text{passed} \wedge c_2 \geq 3 \wedge c'_2 = c_2 + \text{len} \wedge \mathbf{p}'_{D1} = 0 \end{aligned}$$

The translation of the second DC automaton is very similar. The locations $\{1\}$, $\{1, 2\}$ and $\{1, 2, 3^{\geq}\}$ are encoded as 0, 1, and 2, respectively. In this automaton some locations have invariants. This invariant is checked whenever the variable \mathbf{p}_{D2} is set to a new value. This automaton has two initial edges, so the initial constraint is a disjunction of two constraints, one for each initial edge.

$$\begin{aligned} \text{Init}_{DC2} &= \mathbf{p}_{D2} = 0 \wedge c_3 = \text{len} \wedge \text{current} = \text{goal} \\ &\vee \mathbf{p}_{D2} = 1 \wedge c_3 = \text{len} \wedge \text{current} \neq \text{goal} \\ \text{Trans}_{DC2} &= \mathbf{p}_{D2} = 0 \wedge c'_3 = c_3 + \text{len} \wedge \text{current} = \text{goal} \wedge \mathbf{p}'_{D2} = 0 \\ &\vee \mathbf{p}_{D2} = 0 \wedge c'_3 = c_3 + \text{len} \wedge \text{current} \neq \text{goal} \wedge \mathbf{p}'_{D2} = 1 \\ &\vee \mathbf{p}_{D2} = 1 \wedge c'_3 = c_3 + \text{len} \wedge \text{current} \neq \text{goal} \wedge \mathbf{p}'_{D2} = 1 \\ &\vee \mathbf{p}_{D2} = 1 \wedge c'_3 = \text{len} \wedge \text{current} = \text{goal} \wedge c'_3 < 2 \wedge \mathbf{p}'_{D2} = 2 \\ &\vee \mathbf{p}_{D2} = 2 \wedge \neg \text{stop} \wedge c'_3 = c_3 + \text{len} \wedge \text{current} = \text{goal} \wedge \\ &\quad c'_3 < 2 \wedge \mathbf{p}'_{D2} = 2 \\ &\vee \mathbf{p}_{D2} = 2 \wedge \text{stop} \wedge c'_3 = c_3 + \text{len} \wedge \text{current} = \text{goal} \wedge \mathbf{p}'_{D2} = 0 \\ &\vee \mathbf{p}_{D2} = 2 \wedge c'_3 = c_3 + \text{len} \wedge \text{current} \neq \text{goal} \wedge \mathbf{p}'_{D2} = 1 \end{aligned}$$

Before this transition constraint system can be fed into the model checker ARMC some transformations are necessary. ARMC can only handle transition constraint systems in disjunctive normal form (DNF). Each of the

above transition constraint is in this form already. However, the TCS for the parallel product of the four automaton is the conjunction of the formulae above and therefore not in DNF anymore. Therefore, this formula needs to be converted back into DNF, which gives the same formulae as computing the parallel product automaton and converting it to TCS.

The model checker ARMC does not support boolean variables. It can only handle either real or integer variables. We only need boolean variables to model events. These can be abstracted away after converting the TCS to DNF: conjuncts that contain event variables in positive and negative form are equivalent to **false** and can thus be omitted. In the other conjuncts the event variables can be omitted. The particular events occurring is only important for synchronisation and that is resolved when computing the DNF of the full TCS.

For these experiments we used our tool for phase event automata to compute the parallel product as a single automaton. This automaton is then converted into ARMC syntax automatically. The conversion also uses the explicit locations that are provided by ARMC.

Since ARMC does not handle transition constraint systems with both integer and real variables, we have to abstract the integer variables by real variables. Using real variables for *current*, *goal* and *dir* is a safe abstraction. This is because it allows all the runs that are allowed with integer variables and additional runs with non-integer values for these variables. However, this abstraction is too coarse for our purposes. If the difference between *current* floor and *goal* floor is not integer the elevator will never reach the *goal* floor exactly. To prevent this, an additional axiom is inserted that holds for the original case study, where *current* and *goal* are integers:

$$current \neq goal \Rightarrow current \leq goal - 1 \vee current \geq goal + 1$$

This axiom just encodes a property of integer numbers that is needed to prove the safety of the elevator. Because it holds for all runs of the original specification, adding the axiom yields a safe abstraction.

In [HM05] we successfully used ARMC to prove the invariant 6.10 using this algorithm. When abstracting from the event variables, as described above, ARMC proves the property within 96 seconds on a standard Linux PC (2.6 GHz Pentium 4).

However, the general reachability problem for transition constraint systems is only semi-decidable. Therefore, ARMC does not always give an answer. Its termination depends on the heuristic it uses to generate new

abstraction predicates. Unfortunately, the latest version of ARMC cannot find the correct abstraction predicates needed for the elevator case study. The results given above are generated with a previous release of ARMC.

6.5. Related Work

Using binary decision diagrams (BDD) to represent boolean formulae is proposed in [Bry86]. They introduce the notion of reduced ordered BDD to get a small and canonic representation. The theory of ROBDD is well-researched and there are several libraries such as CUDD [Som98], BuDDy [LN99] and JBDD [Vah04]. These libraries use a very efficient representation and highly optimised algorithm. Unfortunately, these optimisations make it very hard to extend the libraries for different kind of decision diagrams.

Our implementation of a decision diagram library can handle nodes with an arbitrary number of children, which are needed for interval decision diagrams [ST98]. New types of nodes can be created in an object-oriented way. For each formula only one instance is created. A reference to this shared instance it is used at all places where the formula is needed. This makes the implementation space efficient.

6.5.1. Audio Protocol

The audio protocol was first analysed in [BPV94] using Timed I/O automata to show that it is correct for $t > 1/17$. The authors establish a weak timed forward simulation from the parallel composition of sender and receiver to a specification. The specification is a simple buffering automaton that receives a message and outputs it after a time. The simulation relation and the proof of correctness were created manually. They were checked with the Coq proof assistant [BC04].

In [LPY95] an automatic proof using Uppaal was given. The synchronisation between sender and a receiver was achieved by an event *up* representing the rising edge on the wire. The correctness was shown by constructing a test automaton by hand. If this automaton expects a zero bit to be received next, it can communicate *output_0*, otherwise, *output_neq_0*. The latter causes the receiver to enter an *error* state if it received a zero. Similarly events exists for one bits. At any state of the test automaton either *output_0* or *output_neq_0* is enabled but this has to be established manually. If one of the conditions under which *output_neq_0* must be communicated

is missing, it is possible that the receiver cannot enter the *error* state even if the protocol is incorrect.

We automatically create the test automaton from Duration Calculus formulae. Thus, provided that the formula describes the desired behaviour the automaton is guaranteed to be correct. It is much easier to create a correct formula than a complex test automaton. Also the behaviour can easily be specified by several small formulae, each testing for one undesired behaviour.

Our model is also simpler than the one given in [LPY95], which is because the change of the wire variable and the receiving and sending of events can be done synchronously with phase event automata. In Uppaal the sender automata need intermediate locations to perform an *up* event at the same time it receives a *input_1* event.

6.5.2. Model Checking Duration Calculus

Although the general Duration Calculus is not decidable [ZHS93], there have been several approaches to prove decidability for subsets of the language or different time models. For example, DCVALID [Pan01] checks a large subset of discrete time duration calculus formulae for satisfiability. In [Frä98], Fränzle gives decidability results for different subsets of DC and underlying time models. The problem he decides is the model property for timed automata: given a timed automata and a DC formula, do all runs of the automata satisfy the formula. For dense-time duration calculus with finitely variable models the largest subset of DC is $\{[P], \ell < k, \ell = k, \ell > k\}$ combined with the operators $\wedge, \vee, \hat{\wedge}$ and exactly one outermost negation.

Using our power set construction, Meyer [Mey05] improves the result of Fränzle: With his test automata he can decide the model property for the formula built from arbitrary positive or negative counterexample formulae using $\wedge, \vee, \hat{\wedge}$ and exactly one outermost negation. Since $[P], \ell < k, \ell > k$ are positive counterexample formulae and $\ell = k$ can be represented by $\ell \geq k \wedge \ell \leq k$, this class is a superset of the set given by Fränzle. With the abstraction given in section 6.3, reachability for Meyer's test automata, which satisfy the strong stuttering invariant (6.1), can be decided with Uppaal.

In [DL02] Dierks and Lettrari present another translation from requirements to test automata. The requirements are given as Constraint Diagrams [Kle00] which have a Duration Calculus semantics with an assump-

tion and a commitment part. Dierks and Lettrari demand that adjacent commitments exclude each other, so that they can be checked in a deterministic way. The class of formulae that can be handled is neither a subclass nor a super class of the formulae that can be handled by Meyer using the power set construction.

7. Conclusion

Contents

7.1. Summary	175
7.2. Future Work	176

In this chapter, we summarise the results of this thesis and discuss topics for future work.

7.1. Summary

In this thesis, we presented CSP-OZ-DC, a new combination of CSP, Object-Z, and Duration Calculus. This language is based on the existing combination CSP-OZ by Fischer. We also redefined the CSP dialect CSPz that is used in the CSP part of CSP-OZ-DC specification to cope with schema types as channel values, and gave new type-checking rules and a more accurately defined semantics. For CSP-OZ-DC, we defined its semantics by a set of interpretations, which are mappings from the time domain into Z models.

To give an operational semantics of CSP-OZ-DC, we defined the model of phase event automata. These automata synchronise on state variables and events, and have the notion of real-time. Although the automata can share global state variables, their parallel composition is fully compositional, which facilitates modular reasoning. The parallel composition corresponds to logical conjunction in Duration Calculus and to alphabetic parallel composition of CSP processes. Therefore, it can be used to define the interaction between the CSP, Object-Z, and Duration Calculus part of a class. It can also be used to build larger systems from multiple classes.

We defined the translation of CSP-OZ-DC to phase event automata. Here, the main contribution is the translation of DC counterexample formulae. We defined an automatic translation for a large class of formulae that is a proper superclass of the DC implementables. In theorem 5.1, theorem 5.2, and corollary 5.20, we proved the soundness of the translation.

Finally, we presented model checking techniques for phase event automata. Instead of creating a new model checker, we provided translations into timed automata and transition constraint systems to reuse existing model checkers for these domains, Uppaal and ARMC. The feasibility of the approach was illustrated with case studies.

7.2. Future Work

In this thesis, we defined a trace semantics for CSP-OZ-DC. For phase event automata a trace semantics was given in form of a set of runs. However, it is well known that a trace semantics is too weak to represent non-deterministic choice and deadlocks. For CSP, the failures divergence semantics gives a more complete view of the behaviour of a process. The semantics is defined as a set of *failures* that describe a behaviour by a trace and the set of communication the process can refuse after the trace. In [Dav93], Davies develops a theory for timed failures, which he uses to define semantics for Timed CSP. A similar semantics can be given for phase event automata, and thus for CSP-OZ-DC classes.

In section 6.3, we used *bad locations* to check properties specified by DC formulae. These bad locations correspond to final states in automata theory. The system violates the formula if there is an accepting run of the automaton that ends in a final state. For liveness and fairness properties, we need to consider infinite runs and Büchi acceptance [Büc60]: a run is accepted if one of the final states is visited infinitely often. Alur and Dill [AD94] analyse timed Büchi automata and the more general timed Muller automata. Their results should be transferable to phase event automata.

The latest version of ARMC can check for termination, i. e., whether a group of locations must eventually be left. A non-terminating run corresponds to an accepting run of a Büchi automaton. Therefore, it should be possible to use Büchi test automata to check general liveness properties. Meyer [Mey05] considers DC formulae with liveness. Although he does not give a translation to phase event automata (which would require Büchi automata), he provides simplifications for liveness formulae. With these simplifications, it should be easy to build the Büchi phase event automaton that can be used with ARMC.

Another interesting task is to combine phase event automata with stop watch automata. These automata have clocks that can be stopped in

certain states without resetting them to zero. If the clock is stopped if and only $\neg P$ holds, the clock measures the duration that P hold, which is expressed as $\int P$ in Duration Calculus. The counterexample traces can be enriched with phases demanding that $\int P \sim k$, where $\sim \in \{<, \leq, \geq, >\}$. Multiple formulae of the form $\int P \sim k$ can be conjuncted, provided that \sim is always the same operator. Because formulae containing $\ell = k$ cannot always be translated into a deterministic automaton, and because it can be expressed as $\int 1 \leq k \wedge \int 1 \geq k$, mixing of comparison operators cannot be allowed. Although the reachability problem for stop watch automata is not decidable, the translation of stop watch phase event automata to transition constraint systems is still possible. There is no guarantee that a model checker like ARMC will produce a result though.

A. Syntax of CSP-OZ-DC

The basic syntax of CSP-OZ-DC is based on the syntax of Z. All constructs allowed in Z are also valid CSP-OZ-DC specifications. The syntax for *Expression* and *Paragraph* is extended to allow the new constructs for CSPz and CSP-OZ-DC. Also some new non-terminals are introduced.

To denote the syntax, we use BNF grammar with a few extensions to make the syntax description more readable. The notation X, \dots, X stands for one or more X separated by the commas. Furthermore, slanted brackets are used as in $[X]$ denote that X is optional.

A.1. New constructs in CSPz

$$\textit{Interface} ::= \textit{ChannelDecl} \textit{NL} \dots \textit{NL} \textit{ChannelDecl}$$

$$\begin{aligned} \textit{ChannelDecl} ::= & \text{chan } \textit{NAME}, \dots, \textit{NAME} [: \textit{Expression}] \\ & | \text{local_chan } \textit{NAME}, \dots, \textit{NAME} [: \textit{Expression}] \end{aligned}$$

$$\textit{ProcessDeclaration} ::= \textit{ProcessEquation} \textit{NL} \dots \textit{NL} \textit{ProcessEquation}$$

$$\textit{ProcessEquation} ::= \textit{NAME} [(\textit{SchemaText})] \stackrel{c}{=} \textit{Expression}$$

$$\textit{Expression} ::= \textit{Stop} \mid \textit{Skip} \mid \textit{Chaos}(\textit{Expression}) \mid \textit{Diver}$$

$$| \textit{Expression} \rightarrow \textit{Expression}$$

$$| \textit{Expression} \square \textit{Expression}$$

$$| \textit{Expression} \sqcap \textit{Expression}$$

$$| \textit{Expression} \textit{;} \textit{Expression}$$

$$| \textit{Expression} \underset{\textit{Expression}}{\parallel} \textit{Expression}$$

$$| \textit{Expression} \underset{\textit{Expression}}{\parallel} \underset{\textit{Expression}}{\parallel} \textit{Expression}$$

$| \textit{Expression} \parallel \textit{Expression}$
 $| \textit{Expression} [\textit{Expression}] \textit{Expression}$
 $| \textit{Expression} \triangle \textit{Expression}$
 $| \textit{Expression} \setminus \textit{Expression}$
 $| \textit{Expression}[\textit{Renaming}]$
 $| \textit{Predicate} \& \textit{Expression}$
 $| \textit{Expression}$
 $| \square \textit{SchemaText} \bullet \textit{Expression}$
 $| \sqcap \textit{SchemaText} \bullet \textit{Expression}$
 $| \wp \textit{SchemaText} \bullet \textit{Expression}$
 $| \parallel [\textit{Expression}] \textit{SchemaText} \bullet \textit{Expression}$
 $| \parallel \textit{SchemaText} \bullet [\textit{Expression}] \textit{Expression}$
 $| \parallel \parallel \textit{SchemaText} \bullet \textit{Expression}$
 $| [\textit{Expression}] \textit{SchemaText} \bullet \textit{Expression}$
 $| \dots (\text{standard rules for Expression}) \dots$

A.2. New constructs in CSP-OZ-DC

$\textit{Paragraph} ::= \textit{COD_Class}$
 $| \dots (\text{standard rules for Paragraph}) \dots$
 $\textit{COD_Class} ::= \frac{\textit{NAME} [[\textit{Formals}]] [(\textit{SchemaText})]}{\textit{COD_SchemaText}}$
 $\textit{COD_SchemaText} ::= \textit{Interface}$
 $\quad \textit{ProcessDeclaration}$
 $\quad [\textit{Paragraph} \dots \textit{Paragraph}]$
 $\quad \textit{State}$
 $\quad [\textit{Init}]$

$$\begin{aligned}
& [\textit{Operation} \dots \textit{Operation}] \\
& | \textit{DC} \\
\textit{State} ::= & \frac{}{[\textit{DeclPart}]} \\
& \frac{}{[\textit{Predicate}]} \\
\textit{Init} ::= & \frac{\textit{Init}}{\textit{Predicate}} \\
\textit{Operation} ::= & \frac{\textit{com_NAME} \quad \textit{DeclPart}}{\Delta(\textit{NAME}, \dots, \textit{NAME})} \\
& \frac{}{[\textit{Predicate}]} \\
\textit{Expression} ::= & [\textit{COD_SchemaText}] \\
& | \dots (\text{standard rules for Expression}) \dots
\end{aligned}$$

A.3. DC formulae

In the DC part of a CSP-OZ-DC part only counterexample formulae are allowed:

$$\begin{aligned}
\textit{DC} ::= & \textit{ce_formula} \textit{NL} \dots \textit{NL} \textit{cd_formula} \\
\textit{ce_formula} ::= & \neg (\textit{phase} \wedge (\textit{phase} | \textit{events}) \\
& \quad \wedge \dots \wedge (\textit{phase} | \textit{events}) \wedge \mathbf{true}) \\
\textit{phase} ::= & (\mathbf{true} | [\textit{Predicate}])[\wedge \ell \sim t] \\
& [\wedge \boxminus \textit{NAME} \dots \wedge \boxminus \textit{NAME}] \\
\sim ::= & \leq | < | > | \geq \\
\textit{events} ::= & \downarrow \textit{NAME} | \uparrow \textit{NAME} \\
& | \textit{events} \vee \textit{events} | \textit{events} \wedge \textit{events}
\end{aligned}$$

Bibliography

- [Abr96] J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.
- [AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
- [AH89] R. Alur and T.A. Henzinger. A really temporal logic. In *IEEE Symposium on Foundations of Computer Science*, pages 164–169, 1989.
- [AL92] M. Abadi and L. 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*, volume 600 of *Lecture Notes in Computer Science*, pages 1–27. Springer-Verlag, 1992.
- [BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.
- [BCCZ03] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Bounded model checking. In *Advances in Computers*, volume 58. Academic Press, 2003.
- [BCM⁺90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

- [BDL04] G. Behrmann, A. David, and K.G. Larsen. A tutorial on UP-PAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in Lecture Notes in Computer Science, pages 200–236. Springer-Verlag, September 2004.
- [BGK⁺96] J. Bengtsson, D. Griffioen, K. Kristoffersen, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using uppaal. In R. Alur and T. Henzinger, editors, *Computer Aided Verification CAV'96*, volume 1102 of *Lecture Notes in Computer Science*, pages 244–256. Springer-Verlag, July 1996.
- [BLR95] A. Bouajjani, Y. Lakhnech, and R. Robbana. From Duration Calculus to linear hybrid automata. In P. Wolper, editor, *Computer Aided Verification CAV'95*, volume 939 of *Lecture Notes in Computer Science*, pages 196–210, Liege, Belgium, 1995. Springer Verlag.
- [BPV94] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an audio control protocol. In H. Langmaack, W.-P. de Roever, and J. Vytupil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863, pages 170–192, Lübeck, Germany, 1994. Springer-Verlag.
- [Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
- [Büc60] J.R. Büchi. On a decision method in restricted second order arithmetic. In E. Nagel et al., editor, *Proc. Internat. Congr. on Logic, Methodology and Philosophy of Science*. Stanford Univ. Press, 1960.
- [CBRZ01] E.M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19, 2001.
- [CCO⁺04] S. Chaki, E.M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In E.A.

- Boiten, J. Derrick, and G. Smith, editors, *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM '04)*, volume 2999 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag, April 2004.
- [CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, number 131 in *Lecture Notes in Computer Science*, pages 52–71, London, UK, 1981. Springer-Verlag.
- [CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
- [CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.
- [Col75] G. E. Collins. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. In *2nd GI Conference on Automata Theory and Formal Languages*, *Lecture Notes in Computer Science*, pages 134–183. Springer-Verlag, 1975.
- [Dav93] J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge Univ. Press, 1993.
- [Die99] H. Dierks. Synthesizing controllers from real-time specifications. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 18:33–43, 1999.
- [Die01] H. Dierks. PLC-automata: A new class of implementable real-time automata. *Theoretical Computer Science*, 253(1):61–93, 2001.
- [DKRS91] R. Duke, P. King, G. Rose, and G. Smith. The object-z specification language. In *In Technology of Object-Oriented Languages and Systems (TOOLS 5)*. Prentice-Hall, 1991.
- [DL02] H. Dierks and M. Lettrari. Constructing test automata from graphical real-time requirements. In *FTRTFT '02: Formal*

- Techniques in Real-Time and Fault-Tolerant Systems*, number 2469 in Lecture Notes in Computer Science, pages 433–454. Springer-Verlag, 2002.
- [DS95] J. Davies and S. Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138:243–271, 1995.
- [Fis00] C. Fischer. *Combination and Implementation of Processes and Data: From CSP-OZ to Java*. PhD thesis, Bericht Nr. 2/2000, University of Oldenburg, April 2000.
- [For05] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, June 2005.
- [Frä98] M. Fränzle. Model-checking dense-time duration calculus. In M.R. Hansen, editor, *Duration Calculus: A Logical Approach to Real-Time Systems*, Workshop proceedings of the 10th European Summer School in Logic, Language and Information, pages 31–40. DFKI Saarbrücken, Germany, August 1998.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [Gri98] A. Griffiths. *A Formal Semantics to Support Modular Reasoning in Object-Z*. PhD thesis, University of Queensland, 1998.
- [Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science Of Computer Programming*, 8(3):231–274, 1987.
- [Hei99] S. Heilmann. *Proof Support for Duration Calculus*. PhD thesis, Dept. Inform. Technology, Tech. Univ. Denmark, June 1999. Tech. Report IT-TR: 1999-030.
- [Hel98] J. Hellbig. *Linking Visual Formalisms: A Compositional Proof System for Statecharts Based on Symbolic Timing Diagrams*. PhD thesis, University of Oldenburg, 1998.
- [HM05] J. Hoenicke and P. Maier. Model-checking of specifications integrating processes, data and time. In J.S. Fitzgerald, I.J. Hayes, and A. Tarlecki, editors, *FM 2005*, volume 3582 of *LNCS*, pages 465–480. Springer, 2005.

-
- [HO02a] J. Hoenicke and E.-R. Olderog. Combining specification techniques for processes data and time. In M. Butler, L. Petre, and K. Sere, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 245–266. Springer-Verlag, May 2002.
- [HO02b] J. Hoenicke and E.-R. Olderog. CSP-OZ-DC: A combination of specification techniques for processes, data and time. *Nordic Journal of Computing*, 9(4):301–334, 2002.
- [Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [HZ97] M.R. Hansen and C. Zhou. Duration calculus: Logical foundations. *Formal Aspects of Computing*, 9:283–330, 1997.
- [ISO02] ISO. *Information technology – Z formal specification notation – Syntax, type system and semantics*. Number 13568. ISO/IEC, 2002.
- [ISO05] ISO. *Information technology – Open Distributed Processing – Unified Modelling Language (UML)*. Number 19501. ISO/IEC, 2005.
- [Joh96] W. Johnston. A type checker for object-z. Technical Report 96-24, University of Queensland, 1996.
- [Kle00] C. Kleuker. *Constraint Diagrams*. PhD thesis, University of Oldenburg, 2000.
- [KLSV06] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures in Computer Science. Morgan & Claypool Publishers, 2006.
- [Kol97] Kolyang. *HOL-Z – An Integrated Formal Support Environment for Z in Isabelle/HOL*. PhD thesis, Univ. Bremen, 1997. Shaker Verlag, Aachen, 1999.

- [Lam83] L. Lamport. What good is temporal logic? In R.E.A. Mason, editor, *Proceedings of the IFIP 9th World Congress*, number 83 in Information Processing, pages 657–668. IFIP, 1983.
- [Lam94] L. Lamport. The temporal logic of actions. *ACM TOPLAS*, 16:872–973, 1994.
- [LN99] J. Lind-Nielsen. BuDDy – a binary decision diagram package. Technical report, Department of Information Technology, Technical University of Denmark, 1999.
- [LPY95] K.G. Larsen, P. Pettersson, and W. Yi. Diagnostic model-checking for real-time systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 575–586. Springer-Verlag, October 1995.
- [LSVW96] N. Lynch, R. Segala, F. Vaandrager, and H. Weinberg. Hybrid I/O automata. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 496–510. Springer-Verlag, 1996.
- [LV93] N. Lynch and F. Vaandrager. Forward and backward simulations – part II: timing-based systems. Technical Report MIT/LCS/TM-487, Massachusetts Institute of Technology, 1993.
- [McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.
- [MD98] B.P. Mahony and J.S. Dong. Blending Object-Z and Timed CSP: an introduction to TCOZ. In K. Futatsugi, R. Kemmerer, and K. Torii, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104. IEEE Computer Society Press, 1998.
- [MD99a] B.P. Mahony and J.S. Dong. Overview of the semantics of TCOZ. In A. Galloway K. Araki and K. Taguchi, editors, *Integrated Formal Methods (IFM'99)*, pages 66–85. Springer-Verlag, 1999.

-
- [MD99b] B.P. Mahony and J.S. Dong. Sensors and actuators in TCOZ. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1166–1185. Springer-Verlag, 1999.
- [Mey05] R. Meyer. Model-Checking von Phasen-Event Automaten bezüglich Duration Calculus Formeln mittels Testautomaten. Master's thesis, Universität Oldenburg, 2005.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [Pan01] P.K. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using DCVALID. In *Real-Time Tools (RTTOOLS 2001)*, Aalborg, 2001.
- [Plo81] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
- [Rav94] A.P. Ravn. Design of embedded real-time computing systems. Dr. tech. dissertation, Technical University of Denmark, September 1994.
- [Ros94] A.W. Roscoe. Model-checking CSP. In A.W. Roscoe, editor, *A Classical Mind – Essays in Honour of C.A.R. Hoare*, pages 353–378. Prentice-Hall, 1994.
- [Ros97] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
- [Ryb02] A. Rybalchenko. A model checker based on abstraction refinement. Master's thesis, Universität des Saarlandes, 2002.
- [Saa97] M. Saaltink. The Z/EVES system. In J. Bowen, M. Hinchey, and D. Till, editors, *ZUM'97*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–88. Springer-Verlag, 1997.
- [Sca94] B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, University of Oxford, 1994.

- [Sch00] F.B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
- [SH99] G. Smith and I. Hayes. Towards real-time Object-Z. In K. Araki, A. Galloway, and K. Taguchi, editors, *Integrated Formal Methods*, pages 49–65. Springer-Verlag, 1999.
- [Ska94] J.U. Skakkebæk. *A Verification Assistant for a Real-Time Logic*. PhD thesis, Dept. Comp. Sci., Tech. Univ. Denmark, Nov. 1994. Tech. Report ID-TR: 1994-150.
- [SKS02] G. Smith, F. Kammüller, and T. Santen. Encoding Object-Z in Isabelle/HOL. In D. Bert, J.P. Bowen, M.C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *LNCS*, pages 82–99. Springer, 2002.
- [Smi92] G. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, University of Queensland, 1992.
- [Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.
- [Smi02] G. Smith. An integration of Real-Time Object-Z and CSP for specifying concurrent real-time systems. In M. Butler, L. Petre, and K. Sere, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 245–266. Springer-Verlag, May 2002.
- [SO99] M. Schenke and E.-R. Olderog. Transformational design of real-time systems – Part 1: from requirements to program specifications. *Acta Inform.*, 36:1–65, 1999.
- [Som98] F. Somenzi. Cudd: Cu decision diagram package, 1998.
- [Spi88] J.M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge tracts in theoretical computer science. Cambridge University Press, 1988.
- [SS63] J.C. Shepherdson and H.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10(2):217–255, 1963.

-
- [ST98] K. Strehl and L. Thiele. Symbolic model checking of process networks using interval diagram techniques. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, pages 686–692, San Jose, California, 8–12, 1998.
- [Süh99] C. Sühl. RT-Z: An integration of Z and Timed CSP. In A. Galloway K. Araki and K. Taguchi, editors, *Integrated Formal Methods (IFM'99)*, pages 66–85. Springer-Verlag, 1999.
- [Süh02] C. Sühl. *An Integration of Z and Timed CSP for Specifying Real-Time Embedded Systems*. PhD thesis, Technische Universität Berlin, 2002.
- [Tap01] J. Tapken. *Model-Checking of Duration Calculus Specifications*. PhD thesis, University of Oldenburg, June 2001.
- [Vah04] A. Vahidi. JBDD, a Java interface to CUDD and BuDDY, 2004. <http://javaddlib.sourceforge.net/jbdd>.
- [Var96] M. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency*, pages 238–266. Springer-Verlag, 1996.
- [Yov97] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1:123–133, 1997.
- [YPD94] W. Yi, P. Pettersson, and M. Daniels. Automatic verification of real-time communicating systems by constraint-solving. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the 7th Int. Conf. on Formal Description Techniques*, pages 223–238. North-Holland, 1994.
- [ZH96] C. Zhou and M.R. Hansen. Chopping a point. In *Proc. BCS FACS 7th Refinement Workshop: Theory and Practice of System Design*, Electronic Workshops in Computing. Springer-Verlag, 1996.
- [ZH04] C. Zhou and M.R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. Springer-Verlag, 2004.

- [ZHS93] C. Zhou, M.R. Hansen, and P. Sestoft. Decidability and undecidability results for Duration Calculus. In P. Enjalbert, A. Finkel, and K.W. Wagner, editors, *Symposium on Theoretical Aspects of Computer Science (STACS'93)*, LNCS 665, pages 58–68. Springer, 1993.

Index

- Σ , 12
- \square (ext. choice), 6, 33, 45
- \sqcap (int. choice), 6, 33, 45
- \wedge , 17
- ℓ , 19
- \updownarrow , 21
- \mathcal{L} , 21
- \boxplus , 21
- f , 17
- $[\cdot]$ (everywhere), 19
- \diamond (eventually), 19
- \square (always), 19
- \longrightarrow , 22
- \xrightarrow{t} , 22
- $\xrightarrow{\leq t}$, 22
- $\&$, 36
- \triangle , 36
- \boxtimes , 57
- \circ , 12
- \parallel , 6, 35, 46
- $\|$, 35
- $[R]$, 35
- \triangleleft , 7
- \trianglelefteq , 7
- \triangleright , 7
- \trianglerighteq , 7
- \circledast , 6, 36
- $[\cdot]^{\mathcal{D}}$, 13
- $[\cdot]^{\mathcal{E}}$, 13
- $[\cdot]^{\mathcal{P}}$, 13
- $[\cdot]^T$, 13
- $\vdash^{\mathcal{D}}$, 12
- $\vdash^{\mathcal{E}}$, 12
- $\vdash^{\mathcal{P}}$, 12
- \mathbb{A} , 11
- accepting, 79
- accepting automaton, 136
- allowEmpty*, 104
- alphaparallel*, 41
- ARMC, 166
- \mathbb{B} , 13
- binding, 8
- bound*, 104
- bounded model checking, 165
- canseep*, 110
- channel, 5, 30
- channel variable, 57
- ChannelDecl*, 30, 49, 179
- Chaos*, 32
- choice
 - external, 6, 33, 42, 45
 - internal, 6, 33, 45
- clock, 70, 72
 - constraints, 71
 - convex constraints, 71
 - invariant, 73
- COD-Class, 48
- COD-SchemaText, 56
- complete*, 108

- convex, 71
 counterexample formula, 23, 51, 103
 CSP
 semantics, 6
 syntax, 6

DeclPart, 8
 deterministic automaton, 84
 disjunction, 82
Diver, 32
 Duration Calculus
 counterexample formula, 23, 51, 103
 CSP-OZ-DC, 51
 decidability, 24
 implementables, 22
 semantics, 18
 syntax, 17

Elevator, 51–55, 138–139, 167–171
enter, 115
entryEvents, 104
 event, 5, 31
Expression, 30, 181
extchoice, 41, 42, 45

 followed-by, 22
forbidden, 104
 formula
 counterexample, 23, 51, 103
 DC, 17
 implementables, 22
 free type, 10
 function symbols, 17

 generic type, 11
genparallel, 41
 GENTYPE, 12

 GIVEN, 11
 given type, 9
 global variables, 17
gteq, 106
guard, 116

 Implementables, 22
 implementing, 81
in, 106
Init, 50, 181
init, 119
intchoice, 41, 45
Interface, 30, 49, 179
 interval, 18
inv, 104

keep, 115
 Kronos, 93

LB (lower bound phases), 105
 len, 159
 less, 107
linkedparallel, 41
 location, 72

 matching, 78
 Model, 13
 multi-prefixing, 33

 NAME, 12
NL (newline), 30

 Object-Z, 14
 semantics, 15
 observables, 17
Operation, 50, 181

 \mathbb{P} (power set), 7
 p_A (active location), 161
Paragraph, 9, 30, 180

- parallel
 - alphabetised (CSP), 35, 47
 - automata, 74
 - generic (CSP), 35, 46
 - interleaved (CSP), 35
- parameter variable, 57
- phase, 23, 51, 103
- Phase Event Automaton
 - accepting, 79
 - accepting automaton, 136
 - definition, 72
 - deterministic, 84
 - disjunction, 82
 - implementing, 81
 - parallel, 74
 - power set, 119
- PhaseSpec*, 104
- PolarCoord*, 8, 14
- power set automaton, 119
- PowerSet*, 107
- Prefix* (of a trace), 105
- prefix* (CSP), 41
- prefixing, 33, 45
- Process*, 179
- ProcessDeclaration*, 32, 49
- ProcessEquation*, 49
- relation symbols, 17
- run, 75
 - matching, 78
 - stuttering extension, 77
- schema, 8
- schema type, 11
- SchemaText*, 8
- seep*, 115
- semantics
 - CSP, 6
 - CSP-OZ-DC, 56, 59
 - CSPz, 44–48
 - DC, 18
 - Object-Z, 15, 58
 - prefixing, 45
 - Z, 12
- sequence*, 41
- signal, 31
- Signature, 11
- Skip*, 6, 32
- State*, 50, 181
- state invariant, 72
- Stop*, 6, 32
- strict*, 71
- stuttering
 - extension, 77
 - invariant, 73
- syntax
 - channel declaration, 30, 49
 - CSP, 6
 - CSP-OZ-DC, 48
 - CSPz, 30–37
 - DC, 17
 - init schema, 50
 - prefixing, 33
 - process declaration, 32, 49
 - state schema, 50
 - Z, 9
- TCS, *see* transition constraint system
- Time, 18
- timed automata, 67, 92
- TimeOp*, 104
- Trace*, 105
- transition constraint system, 160
- type
 - free type, 10
 - generic type, 11
 - given type, 9

- schema type, 11
- type checking, 11, 37–41
- Z types, 11

- U (universe), 12
- UB (upper bound phases), 105
- unroll*, 165
- Uptime*, 58
- Uppaal, 92, 151–159

- W (world), 12
 - W_E (events), 41
 - W_P (processes), 41
- wait*, 106

- xlat*, 25

- Z
 - semantics, 12
 - syntax, 9
 - type checking, 11
 - type system, 11

Curriculum Vitae

- August 21, 1975 born in Nordenham, Germany
- 1981 – 1994 school in Rodenkirchen and Nordenham
- 1994 – 1999 study of computer science at the Carl von Ossietzky
University of Oldenburg, title of the Master’s thesis:
“Graphische Spezifikationsprachen: Der Zusammenhang
zwischen Constraint Diagrams und Real-Time Symbolic
Timing Diagrams” (“Graphical Specification Techniques:
The connection between Constraint Diagrams and Real-Time
Symbolic Timing Diagrams”)
- 1999 – 2006 in the working group “Correct System Design” of
Prof. Dr. Ernst-Rüdiger Olderog at the University
of Oldenburg
- July 12, 2006 defense of the dissertation

Technical Reports

Fakultät II, Department für Informatik, Universität Oldenburg,
Postfach 2503, 26111 Oldenburg, Germany

- 1/87 A. Viereck: „Klassifikationen, Konzepte und Modelle für den Mensch-Rechner-Dialog“ (Dissertation)
- 2/87 A. Schwill: „Forbidden subgraphs and reduction systems: A comparison“
- 3/87 J. Kämper: „Non-uniform proof systems: A new framework to describe non-uniform and probabilistic complexity classes“
- 1/88 K. Ambos-Spies, H. Fleischhack, H. Huwig: „Diagonalizing over deterministic polynomial time“
- 2/88 A. Schwill: „Shortest edge-disjoint paths in geodetically connected graphs“
- 3/88 V. Claus, U. Lichtblau (Hrsg.): „1. Tagung zur Küsten-Informatik“
- 1/89 U. van der Valk: „Einige Entscheidbarkeits- und Unentscheidbarkeitsresultate für Klasse von S/T-Netzen unter Maximum Firing Strategie und unter Prioritätenstrategien“
- 2/89 J. Kämper: „Strukturelle Untersuchungen im Umfeld der Komplexitätsklassen P und NP unter besonderer Berücksichtigung nichtuniformer, probabilistischer und disjunktiv selbstreduzierender Algorithmen“ (Dissertation)
- 3/89 J. Kämper: „Nondeterministic oracle Turing machines with maximal computation paths“
- 1/90 A. Schwill: „Shortest edge-disjoint paths in graphs“ (Dissertation)
- 2/90 K.R. Apt, E.-R. Olderog: „Using transformations to verify parallel programs“
- 3/90 U. Lichtblau: „Flußgraphgrammatiken“ (Dissertation)
- 4/90 K.R. Apt, E.-R. Olderog: „Introduction to program verification“
- 5/90 H. Jasper: „Datenbankunterstützung für Prolog-Programmierungsumgebungen“ (Dissertation)
- 1/91 F. Korf: „Net-based efficient simulation of AADL specifications“
- 2/91 S.V. Krishnan, C. Pandu Rangan, A. Schwill, S. Seshadri: „Two disjoint paths in chordal graphs“
- 3/91 H. Eirund: „Modellierung und Manipulation multimedialer Dokumente“ (Dissertation)
- 4/91 G. Schreiber: „Ein funktionaler Äquivalenzbegriff für den hierarchischen Entwurf von Netzen“

- 1/92 A. Viereck (Hrsg.): „Ergebnisse der 11. Arbeitstagung, Mensch-Maschine Kommunikation“
- 2/92 P. Gorny, U. Daldrup, H. Schwab: „Zwischenbilanz: Menschengerechte Gestaltung von Software“
- 3/92 E.-R. Olderog, St. Rössig, J. Sander, M. Schenke: „ProCoS at Oldenburg: The Interface between Specification Language and occam-like Programming Language“
- 4/92 F. Korf: „Synthesis of VHDL Test Environments form Temporal Logic Specifications“
- 5/92 W. Kowalk: „Konstruktorentchnik: Neue Methoden zur Mengenrechnung, Logikrechnung und Intervallrechnung“
- 1/93 Ch. Dietz, G. Schreiber: „Eine Termdarstellung für S/T-Netze“
- 2/93 J. Sauer: „Wissensbasiertes Lösen von Ablaufplanungsproblemen durch explizite Heuristiken“
- 3/93 M. Sonnenschein, U. Lichtblau (Hrsg.): „6. Kolloquium der Arbeitsgruppe Informatik-Systeme“
- 4/93 H. Fleischhack, U. Lichtblau, M. Sonnenschein, R. Wieting: „Generische Definition hierarchischer zeitbeschrifteter höherer Petrinetze“
- 5/93 F. Köster, L. Twele, R. Wieting, W. Ziegler: „Fallbeispiele zur Modellierung mit THORNetzen“
- 1/94 R. Götze: „Dialogmodellierung für multimediale Benutzerschnittstellen“
- 2/94 B. Müller: „PPO – Eine objektorientierte Prolog-Erweiterung zur Entwicklung wissensbasierter Anwendungssysteme“
- 3/94 W. Damm, A. Mikschl: „Projekt Entwurf und Implementierung eines Multithreaded RISC-Prozessors“
- 4/94 S. Rössig: „A Transformational Approach to the Design of Communicating Systems“ (Dissertation)
- 5/94 G. Schreiber: „Funktionale Äquivalenz von Petri-Netzen“ (Dissertation)
- 1/95 A. Gronewold, H. Fleischhack: „Language Preserving Reductions of Safe Petri-Nets“
- 2/95 H. Reineke: „Struktur und Verhalten von verteilten endlichen Automaten“ (Dissertation)
- 3/95 H. Behrends: „Beschreibung ereignisgesteuerter Aktivitäten in datenbankgestützten Informationssystemen“ (Dissertation)
- 4/95 U. M. Levens: „Computerunterstütztes Modellieren von Musikstücken mit Petri-Netzen: Das Mailänder Konzept“
- 1/96 M. Burke: „FDDI und ATM in multimedialen Anwendungsumgebungen“ (Dissertation)
- 2/96 I. Pitschke: „Interaktive Rekonstruktion geometrischer Modelle aus digitalen Bildern“ (Dissertation)
- 1/97 L. Bölke: „Ein akustischer Interaktionsraum für blinde Rechnerbenutzer“ (Dissertation)

-
- 2/97 S. Schöf: „Verteilte Simulation höherer Petrinetze“ (Dissertation)
- 1/98 S. Kleuker: „Inkrementelle Entwicklung von verifizierten Spezifikationen für verteilte Systeme“ (Dissertation)
- 2/98 J. Bohn: „Mechanical Support and Validation of a Design Calculus for Communicating Systems by a Logic-Based Proof System“ (Dissertation)
- 3/98 L. Köhler: „Fuzzy Geometrie und Anwendungen in der medizinischen Bildverarbeitung“ (Dissertation)
- 4/98 J. Helbig: „Linking Visual Formalisms: A Compositional Proof System for Statecharts Based on Symbolic Timing Diagrams“ (Dissertation)
- 5/98 G. Stiege: „Edge Partitions in Undirected Graphs“
- 6/98 A. Gerns: „Entwicklung und Bewertung von Objektmigrationsstrategien für verteilte Umgebungen“
- 7/98 M. Stadler: „Abstrakte Rechnernetzmodelle als Grundlage einer umfassenden Automatisierung des Netzmanagements – Konzepte und Sprachen zu ihrer Umsetzung“ (Dissertation)
- 8/98 M.-S. Steiner: „Lastverteilung in heterogenen Systemen“
- 9/98 Clemens Otte: „Fuzzy-Prototyp-Klassifikatoren und deren Anwendung zur automatischen Merkmalsselektion“
- 1/99 Juliane Vorndamme: „Die Auswirkungen rechtlicher Verpflichtungen auf die Software-entwicklung“
- 2/99 E. Best/K.M. Richter: „Relational Semantics Revisited“
- 3/99 J. S. Lie: „Einsatz von Objektmigrationssystemen zur Leistungssteigerung in verteilten Systemen“
- 4/99 Zweijahresbericht des Fachbereichs Informatik
- 5/99 Ingo Stierand, Olaf Maibaum, Björn Briel, Günther Stiege: „Cassandra – Generierung, Analyse und Simulation von eingebetteten Multiprozessor-Echtzeitsystemen“
- 6/99 Gunnar Wittich: „Ein problemorientierter Ansatz zum Nachweis von Realzeiteigenschaften eingebetteter Systeme“
- 7/99 Annegret Habel, Jürgen Müller, Detlef Plump: „Double-Pushout Graph Transformation Revisited“
- 8/99 Ingo Stierand: „Eine Konfigurationssprache zur Erstellung von Ambrosia/MP-Systemen“
- 9/99 Igor V. Tarasyuk: „Equivalences for Concurrent and Distributed Systems“
- 10/99 Eike Best, Alexander Lavrov: „Generalised Composition Operations for High-Level Petri-Nets“
- 11/99 Alexander Lavrov: „Enhancing Mixed Nonlinear Optimization: A Hybrid Approach“
- 12/99 Alexander Lavrov: „Hybrid Techniques in Discrete-Event System Modelling and Control: some Examples“

- 13/99 Eike Best, Raymond Devillers, Maciej Koutny: „Recursion and Petri Nets“
- 14/99 Eike Best, Raymond Devillers, Maciej Koutny: „The Box Algebra = Petri Nets + Process Expressions“
- 15/99 Eike Best, Harro Wimmel: „Reducing k -safe Petri Nets to Pomset-equivalent 1-safe Petri Nets“
- 16/99 Udo Brockmeyer: „Verifikation von STATEMATE Designs“ (Dissertation)
- 1/00 Henning Dierks: „Specification and Verification of Polling Real-Time Systems“ (Dissertation)
- 2/00 Clemens Fischer: „Combination and Implementation of Processes and Data: from CSP-OZ to Java“ (Dissertation)
- 3/00 Cheryl Kleuker: „Constraint Diagrams“ (Dissertation)
- 4/00 Thomas Thielke: „Linear-algebraische Methoden zur Beschreibung, Verfeinerung und Analyse gefärbter Petrinetze“ (Dissertation)
- 1/01 Günther Stiege: „Higher Decomposition in Undirected Graphs“ (Bericht)
- 2/01 Ute Vogel: Zweijahresbericht
- 3/01 Josef Tapken: „Model-Checking of Duration Calculus Specifications“ (Dissertation)
- 4/01 Björn Briel: „Analyse eingebetteter Systeme mittels verteilter Simulation“ (Dissertation)
- 5/01 Günther Stiege: „Standard Decomposition and Periodicity of Digraphs“ (Bericht)
- 6/01 Ingo Stierand: „Ambrosia/MP – Ein Echtzeitbetriebssystem für eingebettete Mehrprozessorsysteme“ (Dissertation)
- 1/02 Giorgio Busatto, Annegret Habel: „Improving the Quality of Hypertexts Using Graph Transformation“ (Bericht)
- 2/02 Giorgio Busatto: „Modeling Hyperweb Dynamics through Hierarchical Graph Transformation“ (Bericht)
- 3/02 Giorgio Busatto: „An Abstract Model of Hierarchical Graphs and Hierarchical Graph Transformation“ (Dissertation)
- 4/02 Laila Kabous: „An Object Oriented Design methodology for hard real Time Systems: The OOHARTS approach“ (Dissertation)
- 1/03 Ute Vogel: „Zweijahresbericht“
- 2/03 Olaf Maibaum: „Bestimmung symbolischer Laufzeiten in eingebetteten Echtzeitsystemen“ (Dissertation)
- 3/03 Günther Stiege, Ingo Stierand: „Connectedness-Based Hierarchical Decomposition of Undirected Graphs“ (Bericht)
- 4/03 Willi Hasselbring, Susanne Petersen: „Standards für die medizinische Kommunikation und Dokumentation“ (Bericht)
- 5/03 Andreas Möller: „Eine virtuelle Maschine für Graphprogramme“ (Bericht)
- 6/03 Tom Bienmüller: „Reducing Complexity for the Verification of StateMATE Designs“ (Bericht)

-
- 7/03 Sandra Steinert: „Graph Programs for Graph Algorithms“ (Bericht)
- 8/03 Jochen Klose: „Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior“ (Dissertation)
- 1/04 Jens Oehlerking: „Transformation of Edmonds’ Maximum Matching Algorithm into a Graph Program“ (Bericht)
- 2/04 Sergej Alekseev: „Dienste Intelligenter Netze Graphentheoretische Methoden in der Kontrollflussanalyse“ (Bericht)
- 3/04 Giorgio Busatto: „GraJ: A System for Executing Graph Programs in Java“ (Bericht)
- 1/05 Sergej Alekseev: „Ablaufanalyse objektorientierter Echtzeitanwendungen mit graphentheoretischen Methoden“ (Dissertation)
- 2/05 Ute Vogel: „Zweijahresbericht“
- 3/05 Igor Tarasyuk: „Discrete time stochastic Petri box calculus“ (Bericht)
- 1/06 Henning Dierks: „Time, Abstraction and Heuristics“ (Habilitation)
- 2/06 Li Sek Su: „Full-Output Siphons and Deadlock-Freeness for Free Choice Petri Nets“ (Bericht)
- 3/06 Timo Warns: „Solving Consensus Using Structural Failure Models“ (Bericht)
- 4/06 Sergej Alekseev: „Graphentheoretische Methoden in der Ablaufanalyse objektorientierter Anwendungen“ (Dissertation)
- 5/06 Li Sek Su: „Some Considerations on the Foundation of NP-Completeness Theory“ (Bericht)
- 6/06 Li Sek Su: „Semitraps and Deadlock-Freeness for Reduced Asymmetric Choice Nets“ (Bericht)
- 7/06 Li Sek Su: „Algorithms of computing the Deadlock Markings Sets for Petri Nets“ (Bericht)
- 8/06 Annegret Habel, Karl-Heinz Pennemann, Arend Rensink: „Weakest Preconditions for High-Level Programs (Long Version)“ (Bericht)
- 9/06 Jochen Hoenicke: „Combination of Processes, Data, and Time“ (Dissertation)