Model-Checking of Specifications Integrating Processes, Data and Time*

Jochen Hoenicke[1] and Patrick Maier[2]

[1] Universität Oldenburg, Department für Informatik, 26111 Oldenburg, Germany
hoenicke@informatik.uni-oldenburg.de

[2] MPI für Informatik, Programming Logics Group, 66123 Saarbrücken, Germany
maier@mpi-sb.mpg.de

Abstract. We present a new model-checking technique for CSP-OZ-DC, a combination of CSP, Object-Z and Duration Calculus, that allows reasoning about systems exhibiting communication, data and real-time aspects. As intermediate layer we will use a new kind of timed automata that preserve events and data variables of the specification. These automata have a simple operational semantics that is amenable to verification by a constraint-based abstraction-refinement model checker. By means of a case study, a simple elevator parameterised by the number of floors, we show that this approach admits model-checking parameterised and infinite state real-time systems.

1 Introduction

Complex computing systems exhibit various behavioural aspects such as communication between components, state transformation within components, and real-time constraints on the communications and state changes. This observation has led research to combine and semantically integrate specification techniques. In [13] and [14] we introduced CSP-OZ-DC, the combination of three well-investigated specification techniques: CSP [11], Object-Z [21, 22] and Duration Calculus [26, 25]. Due to its expressiveness, however, CSP-OZ-DC is not suited for automated verification.

In this paper, we present an approach to automatically verify CSP-OZ-DC specifications by model-checking. To this end, the specifications are translated to *transition constraint systems* (transition systems whose transitions are labelled by constraints expressed in first-order logic), which are model-checked using constraint-based symbolic techniques [5] plus predicate abstraction [9] with counterexample-driven abstraction refinement [3, 10, 4].

The translation from CSP-OZ-DC to transition constraint systems is via a novel class of timed automata, called *phase event automata*, providing an essential prerequisite for model-checking: an operational semantics for CSP-OZ-DC specifications. These automata describe the behaviour of instantaneous events that stem from the CSP

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See www.avacs.org for more information.

J.S. Fitzgerald, I.J. Hayes, and A. Tarlecki (Eds.): FM 2005, LNCS 3582, pp. 465–480, 2005.
© Springer-Verlag Berlin Heidelberg 2005

world, states with durations that model the Object-Z state variables, and clocks used for real-time constraints defined by Duration Calculus. The translation to phase event automata is compositional, i. e., the translation of a CSP-OZ-DC specification is a parallel product of several automata, each corresponding to one part of the specification. Thus, our phase event automata provide the first compositional operational semantics for (a subclass of) CSP-OZ-DC.

The translation from phase event automata to transition constraint systems follows an "old-fashioned recipe for real-time" [1, 15] by splitting continuous runs into discrete sequences of intervals. Moreover, the translation is compositional with respect to parallel products. All in all, the process of translating CSP-OZ-DC specifications via phase event automata into transition constraint systems and model-checking these systems can be automated completely. In all steps the structure of the original specification is preserved, so that counterexamples found by the model-checker can easily be translated back to the CSP-OZ-DC world.

For being able to model-check, we have to pay a price. We have to restrict the CSP-OZ-DC specifications such that the CSP part is finite state, the constraints in the OZ part fall into a decidable class, and the DC part consists of so-called *counterexample formulae* only. Nevertheless, this subclass of CSP-OZ-DC still admits non-trivial specifications, as we show in a small case study.

The paper is organised as follows. Section 2 introduces the main constructs of CSP-OZ-DC via a case study. Section 3 describes phase event automata. Section 4 sketches the translation from CSP-OZ-DC to phase event automata. In section 5 we will introduce transition constraint systems and give the translation from automata to transition constraints. Section 6 presents the results of applying our approach to the case study and verifying an invariant. Finally, we conclude with section 7.

2 Case Study

In this section we introduce the combined formalism CSP-OZ-DC [13] and the case study of a controller for an elevator, see Fig. 1. The case study is kept very simple and only contains the core of the controller. It is separated into three aspects, each of which is specified in one of the three languages. The control and communication aspects are specified with CSP and encompass the interaction with the environment abstracting from concrete values transmitted. Data aspects specified with Object-Z involve the calculation of current and goal floor. The real-time behaviour is specified with Duration Calculus.

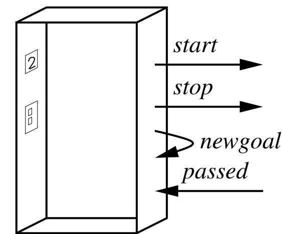

Fig. 1. Elevator

Communication Aspects. are described with CSP [11], a language for communicating sequential processes. It is used to define the admissible sequences of events:

$$\mathtt{main} \overset{c}{=} \mathit{newgoal} \rightarrow \mathit{start} \rightarrow \mathit{Drive}$$
$$\mathit{Drive} \overset{c}{=} (\mathit{passed} \rightarrow \mathit{Drive}) \mathbin{\Box} (\mathit{stop} \rightarrow \mathtt{main})$$

The elevator has a cyclic behaviour switching between the processes `main` and *Drive*. The keyword `main` names the process that will be entered initially. The elevator first chooses a new goal floor, then it starts the engine and switches to the *Drive* process. It can then either pass a floor and keep on driving, or stop and return to the `main` process. The symbol $\Box$ denotes an external choice, which means that the environment determines which of these event will be taken. In this case it is determined by the interaction with the Object-Z and Duration Calculus part of the specification.

Data Aspects. The representation of data state and the algorithmic part of the elevator is described with Object-Z. 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 $Min < Max$. These bounds can be seen as parameters of the elevator. In Z these constants are declared in a so called axiomatic definition. 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 a schema with the special name `Init`.

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

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

`Init`
$goal = current = Min$
$dir = 0$

In CSP-OZ-DC the link between events and states is established by communication schemas. By naming convenience, the following schema describes the change that the *passed* event induces:

com_*passed*
$\Delta(current)$
$current' = current + dir$

The Δ list on the first line mentions 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 the set of requested floors and the algorithm to choose the next goal floor is abstracted from. Instead the goal floor is chosen non-deterministically from the range of all floors except the current one. When the elevator starts, it will choose the direction in accordance with the position of the new goal floor. 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. These schemas are given Fig. 2.

Real-Time Aspects. are described with Duration Calculus (DC). This is a logic that allows specifying real-time behaviours. Unfortunately the full logic of Duration Calculus is too powerful to be checked automatically. Therefore only a restricted class of formulae, called *counterexample formulae*, may be used in CSP-OZ-DC specifications[1].

[1] In [13] we used implementables for Duration Calculus but that are just abbreviations for certain counterexample formulae.

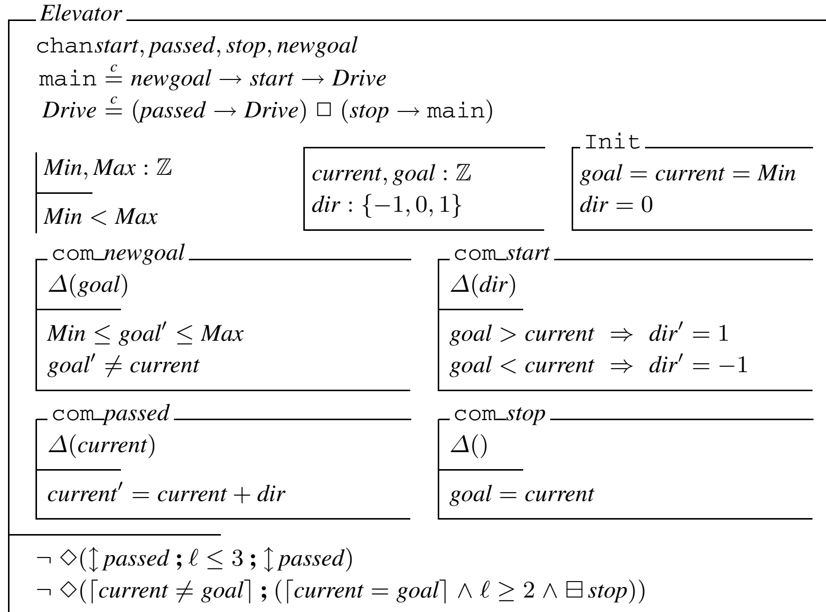

Fig. 2. Elevator specification

A counterexample formula describes a specific undesired behaviour in form of a linear trace. This formula is negated as it is a forbidden behaviour. Restricting ourself to these types of formulae makes over-specification less likely: It is easy to see that a certain behaviour should not occur and this is all the formula states.

The general shape of a counterexample formula is as follows:

$$\neg \Diamond(phase_1 \,;\, \ldots \,;\, phase_n)$$

Here the formula $\Diamond F$ states that there is a subinterval (DC formulae describe the shape of trajectories in a given time interval) where F holds. This interval is then chopped up into n subintervals (this is denoted by **;**) each satisfying $phase_i$, which must be a simple formula restricting the current state of the system, the events that may or may not occur during this interval, and either the minimum or the maximum length of this interval. The whole formula is negated as it is a counterexample.

To restrict the state of a variable the standard Duration Calculus notation is used: For example, $\lceil dir = 1 \rceil$ holds for intervals satisfying $dir = 1$. For each event a new Boolean variable is introduced that changes every time the event occurs. The formula $\updownarrow ev$ holds for a point interval, at which the Boolean variable ev changes[2]. The formula $\boxminus ev$ states that an event does not occur during a non-empty interval[3].

In the case study real-time properties are used to ensure that the elevator stops when it reaches the goal floor before passing the next floor. To achieve this, a minimum time

[2] It is defined as $\updownarrow ev = \uparrow ev \vee \downarrow ev$ with the operators $\uparrow, \downarrow$ as defined in [25]

[3] It is defined as $\boxminus ev := \lceil ev \rceil \vee \lceil \neg\, ev \rceil$

of three seconds between two adjacent *passed* events is demanded. This is expressed by a negated counterexample where two *passed* events occur after each other, with an interval in between that has a duration (denoted by ℓ) of at most three seconds.

$$\neg \Diamond(\updownarrow passed \,;\, \ell \leq 3 \,;\, \updownarrow passed)$$

Furthermore it is claimed that the elevator stops within two seconds. The following formula states the impossibility of the stop event not occurring even after the goal has been reached for more than two seconds.

$$\neg \Diamond(\lceil current \neq goal \rceil \,;\, (\lceil current = goal \rceil \wedge \ell \geq 2 \wedge \boxminus stop))$$

The complete specification of the elevator is shown in Fig. 2. The specification is framed and given a name. It starts with the interface specification that lists the names of the communication events. The interface is followed by the CSP and Object-Z part. Then follows the DC part, which is separated by a short horizontal line.

The property to verify for this specification is $Min \leq current \leq Max$. Note that it is not obvious that this property holds at all, as there is no such check in com_*passed*. It only holds because of the interaction between the CSP process, the data transformation and the real-time properties of the specification. As a matter of fact, every single line of the above specification contributes to this property. In the remainder of this paper this combined specification is translated into a certain kind of timed automata and the invariant property above is proven.

3 Phase Event Automata

In this section a new type of timed automata is introduced, so called *phase event automata*, that can characterise the behaviour of state- and event-based systems. These automata serve as a bridge between CSP-OZ-DC described in section 2 and transition constraint systems that will be described in the next section. They possess the notion of events, variables and clocks.

Fig. 3 shows an example of a phase event automaton. This automaton corresponds to the second Duration Calculus formula of the case study specifying that the automaton should stop when the destination floor has been reached. Initially it can be either in

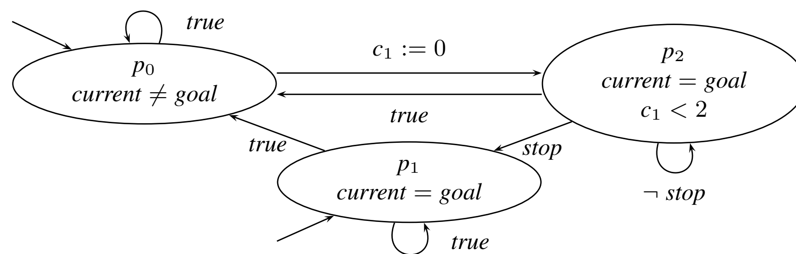

Fig. 3. A phase event automaton

phase p_0 (if $current \neq goal$ holds) or in phase p_1 (otherwise). There are no restrictions of what may happen next. As soon as a change from $current \neq goal$ to $current = goal$ occurs, the automaton switches to phase p_2, resets the clock c_1 to zero and makes sure that the elevator will stop within two seconds. Due to the invariant $c_1 < 2$, phase p_2 must be left in time. One possibility is to back to p_1, which can only be done if $current \neq goal$ holds. The other possibility is by a *stop* event.

3.1 Notation

The states of the systems are described by first-order formulae. We work in many-sorted first-order logic with equality denoted by $\approx$. The set of variables is denoted by $\hat{\mathcal{V}}$. With each variable $x \in \hat{\mathcal{V}}$ a sort $type(x)$ is associated, which restricts the possible values for x. The logic uses typed functions and predicate symbols. From this terms and formulae are defined inductively. By $\mathcal{L}$, we denote the class of first-order formulae that are allowed in the specification. $\mathcal{L}(V)$ denotes the set of those formulae in $\mathcal{L}$ that only refer to variables in $V \subseteq \hat{\mathcal{V}}$. To be able to formulate the translations in Section 5, we demand that $\mathcal{L}$ contains at least the class of quantifier-free formulae involving only Booleans variables and linear arithmetic expressions over the reals. For the case study, $\mathcal{L}$ should moreover contain linear arithmetic expressions over the integers.

The set of variables $\hat{\mathcal{V}}$ is partitioned into two disjoint sets $\mathcal{V}$ and $\mathcal{V}'$ such that $\mathcal{V}'$ is a copy of $\mathcal{V}$. We call the variables in $\mathcal{V}'$ *primed*, those in $\mathcal{V}$ *unprimed*. The unprimed variables refer to the state before a transition while the primed variables refer to the post state.

Semantically, variables are interpreted by valuations and all syntactic symbols except variables by a fixed algebra. Given a subset $V \subseteq \hat{\mathcal{V}}$, a V-valuation α is a mapping that assigns a value in $U_{type(x)}$ to each variable $x \in V$, the domain of that type. Sometimes, we denote a V-valuation α by the expression $\{x \mapsto \alpha(x) \mid x \in V\}$. The set of all V-valuations is denoted by $Val(V)$. Given two subsets $V_1, V_2 \subseteq \hat{\mathcal{V}}$ and a V_1-valuation α, we denote the restriction of α to a $(V_1 \cap V_2)$-valuation by $\alpha|_{V_2}$. Given two subsets $V_1, V_2 \subseteq \hat{\mathcal{V}}$, a V_1-valuation α_1 and a V_2-valuation α_2 with $\alpha_1|_{V_2} = \alpha_2|_{V_1}$, we write $\alpha_1 \cup \alpha_2$ to denote the $(V_1 \cup V_2)$-valuation α with $\alpha|_{V_1} = \alpha_1$ and $\alpha|_{V_2} = \alpha_2$. Given a subset $V \subseteq \mathcal{V}$ and a V-valuation α, we write α' to denote the V'-valuation with $\alpha'(x') = \alpha(x)$ for all $x \in V$. Given a V-valuation α and a formula φ with $free(\varphi) \subseteq V$, we write $\alpha \models \varphi$ to denote that α satisfies φ. We write $\models \varphi$ to denote that φ is valid.

To introduce the timed automata notion of clocks, we distinguish a sort Time, interpreted by the (non-negative) real numbers. Let **Clocks** $\subseteq \mathcal{V}$ be a set of time variables, i. e., $type(c) = \mathsf{Time}$ for all $c \in$ **Clocks**, which we call *clocks*. Let $C \subseteq$ **Clocks** be a set of clocks. Given two C-valuations α and β, a non-negative real number $t \geq 0$ and a subset of clocks $X \subseteq C$, we define $\alpha + \beta$, $\alpha + t$, $t\alpha$ as the C-valuations that are obtained by addition resp. multiplication of the clock values and $\alpha[X := 0]$ as the C-valuation that assigns all clocks in X the value zero and leaves all other unchanged. We call a formula $\varphi \in \mathcal{L}(C)$ *convex* if $(1 - t)\alpha + t\beta \models \varphi$ for all real numbers $0 \leq t \leq 1$ and all C-valuations α and β with $\alpha \models \varphi$ and $\beta \models \varphi$.

The *events* are modelled by a set **Events** $\subseteq \mathcal{V}$ of boolean variables, i. e., $type(e) = \mathsf{Bool}$ for all $e \in$ **Events**. However, here events are not modelled by changes of this

variable, but the variable is true if the event occurs, false otherwise. Let $E \subseteq$ **Events** be a set of events. By χ_E, we denote the characteristic function of E, i. e., the mapping from **Events** to $U_{\mathsf{Bool}} = \mathbb{B}$ such that for all $e \in$ **Events**, $\chi_E(e) = true$ iff $e \in E$. Note that χ_E is an **Events**-valuation.

3.2 Formal Definition

A *phase event automaton (PEA)* is defined as a tuple $\mathcal{A} = (P, V, A, C, E, s, I, P_0)$ of the following components:

- P is a set of states (phases).
- $V \subseteq \mathcal{V} \setminus$ (**Events** $\cup$ **Clocks**) is a finite set of (state) variables.
- $A \subseteq$ **Events** is a finite set of events.
- $C \subseteq$ **Clocks** is a finite set of clocks.
- $E \subseteq P \times \mathcal{L}(V \cup V' \cup A \cup C) \times \mathbb{P}(C) \times P$ is a set of edges. An edge $(p_1, g, X, p_2) \in E$ represents a transition from phase p_1 to phase p_2 under guard g. All clocks in X are reset when this transition is taken.
- $s : P \rightarrow \mathcal{L}(V)$ is a labelling function that associates each phase with a predicate that must hold during this phase.
- $I : P \rightarrow \mathcal{L}(C)$ is a function assigning to each phase a clock invariant that has to hold while the automaton is in this phase.
- $P_0 \subseteq P$ is a set of possible initial phases.

We impose the extra requirements that

- for all $p \in P$, the clock invariant $I(p)$ is convex, and
- for all $p \in P$, E contains a stuttering edge $(p, \neg e_1 \wedge \ldots \wedge \neg e_k \wedge v_1 = v_1' \wedge \ldots \wedge v_j = v_j', \varnothing, p)$ for some particular $\{e_1, \ldots, e_k\} \subseteq A$, $\{v_1, \ldots, v_j\} \subseteq V$.

To make the intuitive meaning of phase event automata precise we define the traces of an automaton as sequences of variable and clock evaluations, time delays and communicated events. Let $\mathcal{A} = (P, V, A, C, E, s, I, P_0)$ be a PEA. A *state* of $\mathcal{A}$ is a triple (p, β, γ) of a phase $p \in P$, a V-valuation β and a C-valuation γ. A *duration* is a positive real number. A *run* of $\mathcal{A}$ is an infinite sequence

$$\langle (p_0, \beta_0, \gamma_0), t_0, Y_0,\ (p_1, \beta_1, \gamma_1), t_1, Y_1, \ldots \rangle$$

alternating states (p_i, β_i, γ_i), durations t_i and sets of events $Y_i \subseteq A$ such that the following holds:

1. $p_0 \in P_0$.
2. For all $c \in C$, $\gamma_0(c) = 0$.
3. For all $i \geq 0$, $\beta_i \models s(p_i)$.
4. For all $i \geq 0$ and all $0 \leq \delta \leq t_i$, $\gamma_i + \delta \models I(p_i)$.
5. For all $i \geq 0$ there is an edge $(p_i, g, X, p_{i+1}) \in E$ such that
   (a) $\beta_i \cup \beta_{i+1}' \cup (\gamma_i + t_i) \cup \chi_{Y_i} \models g$ and
   (b) $\gamma_{i+1} = (\gamma_i + t_i)[X := 0]$.

We denote the set of runs by $Run(\mathcal{A})$. We call a state (p, β, γ) *reachable* if there is a run $\langle (p_0, \beta_0, \gamma_0), t_0, Y_0,\ (p_1, \beta_1, \gamma_1), t_1, Y_1, \ldots \rangle$ of $\mathcal{A}$ such that $(p, \beta, \gamma) = (p_i, \beta_i, \gamma_i + \delta)$ for some $i \geq 0$ and $0 \leq \delta \leq t_i$. By $Reach(\mathcal{A})$, we denote the set of reachable states of $\mathcal{A}$.

The stuttering edge $(p_i, \neg e_1 \wedge \ldots \wedge \neg e_k \wedge v_1 = v_1' \wedge \ldots \wedge v_j = v_j', \varnothing, p_i)$ is required to make the definition invariant against stuttering. This simplifies the definition of parallel composition, because automata can step synchronously.

Lemma 1. *Let $\mathcal{A}$ be a PEA and $r = \langle (p_0, \beta_0, \gamma_0), t_0, Y_0,\ (p_1, \beta_1, \gamma_1), t_1, Y_1, \ldots \rangle$ a run of $\mathcal{A}$. Then for all $i \geq 0$, stuttering the i-th state in r yields another run of $\mathcal{A}$; more precisely for all $0 < \delta < t_i$, replacing the subsequence $\langle (p_i, \beta_i, \gamma_i), t_i, Y_i \rangle$ in r by $\langle (p_i, \beta_i, \gamma_i), \delta, \varnothing,\ (p_i, \beta_i, \gamma_i + \delta), t_i - \delta, Y_i \rangle$ yields a run of $\mathcal{A}$.*

Given a run $\langle (p_0, \beta_0, \gamma_0), t_0, Y_0,\ (p_1, \beta_1, \gamma_1), t_1, Y_1, \ldots \rangle$ of $\mathcal{A}$, we call the infinite sequence $(\beta_0, t_0, Y_0,\ \beta_1, t_1, Y_1, \ldots)$ a *trace* of $\mathcal{A}$, i. e., a trace is a sequence alternating V-valuations, durations and sets of events. By $Trace(\mathcal{A})$, we denote the *trace language* (i. e., set of traces) of $\mathcal{A}$.

3.3 Parallel Composition

To build a larger system from multiple automata a parallel composition operator has to be defined. Here, it also plays an important role in defining semantics for CSP-OZ-DC. Each part is translated separately into an automaton and they are put in parallel. In [13] the CSP and Object-Z part are joined by the CSP synchronised parallel operator and the Duration Calculus part is joined with logical conjunction. To define equivalent semantics with phase event automata the parallel composition is required to have the same property. To achieve this, the automata are synchronised on both events and states: An event that is in the alphabet of both automata may only be taken if both automata agree, which is the same as CSP synchronisation. Likewise a variable of both automata may only be changed if both automata allow it, which corresponds to logical conjunction. The clocks need to be disjoint, so they do not interfere with each other. The *parallel composition* $\mathcal{A}_1 \parallel \mathcal{A}_2$ *of two automata* $\mathcal{A}_1$ *and* $\mathcal{A}_2$, $\mathcal{A}_i = (P_i, V_i, A_i, C_i, E_i, s_i, I_i, P_{0i})$, is the PEA $\mathcal{A} = (P, V, A, C, E, s, I, P_0)$ defined as follows:

- $P := P_1 \times P_2$. This is a standard product automata construction.
- $V := V_1 \cup V_2$.
- $A := A_1 \cup A_2$. The new alphabet is the union of the two alphabets.
- $C := C_1 \cup C_2$ and $C_1 \cap C_2 = \varnothing$. The clock set is the disjoint union of C_1 and C_2, that is clocks that appear in both sets need to be renamed.
- $s((p_1, p_2)) = s(p_1) \wedge s(p_2)$. The states are labelled with the conjunction of the corresponding state predicates in $\mathcal{A}_1$ and $\mathcal{A}_2$.
- $I((p_1, p_2)) = I(p_1) \wedge I(p_2)$. Likewise the clock invariant is the conjunction of the clock invariants in $\mathcal{A}_1$ and $\mathcal{A}_2$.
- $P_0 := P_{01} \times P_{02}$.
- The set of edges E contains $((p_1, p_2), g_1 \wedge g_2, X_1 \cup X_2, (p_1', p_2'))$ for each two edges $(p_i, g_i, X_i, p_i') \in E_i, i = 1, 2$ in the corresponding automata $\mathcal{A}_i$. Note that the stuttering edges of one automaton allow the other automaton to do a step independently from the first automaton. This is the reason why stuttering edges are required.

This is a product automaton construction. Both automata must agree on the state space and events must occur synchronously, therefore the state predicates and transition guards are the conjunction of the predicates for the two automata. It is obvious from this definition that parallel composition is commutative (modulo renaming of phases) and that it preserves the extra requirements of convexity and stuttering edges. The traces of the parallel automaton are exactly those that are allowed by both automata:

Lemma 2. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be PEA. Then $\langle \beta_0, t_0, Y_0, \ldots \rangle \in Trace(\mathcal{A}_1 \parallel \mathcal{A}_2)$ if and only if $\langle \beta_0|_{V_1}, t_0, Y_0 \cap A_1, \ldots \rangle \in \mathcal{A}_1$ and $\langle \beta_0|_{V_2}, t_0, Y_0 \cap A_2, \ldots \rangle \in \mathcal{A}_2$.*

This can be easily seen by comparing the runs of the three automata. This lemma suggests the following verification method for properties that are satisfied if they hold for every trace. To prove such a property for a system of automata $\mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$, one can choose some automata that seem to be related to the property. The hope is that for this small subsystem it is much easier to prove than for the full system. If the smaller subsystem satisfies the property, the complete system does also, because it has only fewer traces. Otherwise the model-checker gives a counterexample that can be examined. If it is prevented by one of the remaining automata the automaton is added to the parallel product and the model checking is repeated.

4 PEA Semantics for CSP-OZ-DC

In this section we will give semantics for CSP-OZ-DC based on phase event automata. They are equivalent to the semantics given in [13]. The semantics is compositional: The CSP, Object-Z and Duration Calculus part are translated separately into phase event automata and then run in parallel. The semantics of the complete elevator specification is

$$\mathcal{A}(Elevator) = \mathcal{A}(CSP_{Elevator}) \| \mathcal{A}(OZ_{Elevator}) \| \mathcal{A}(DC_{Elevator})$$

Translation of CSP. The translation of the CSP part to a phase event automaton is straightforward. The operational semantics of CSP [17] is used to construct an equivalent phase event automaton. The phases are labelled by CSP processes, the alphabet A is the alphabet of `main`. There are no state variables V and no clocks C. For each transition $p \xrightarrow{a} p'$ of the operational semantics there is an edge $(p, a \wedge \bigwedge_{e \in A \setminus \{a\}} \neg e, \varnothing, p') \in E$, which allows only event a and forbids all other events in the alphabet. For a τ transition $p \xrightarrow{\tau} p'$ the corresponding edge is $(p, \bigwedge_{e \in A} \neg e, \varnothing, p')$ communicating no events. And finally there is the stuttering edge $(p, \bigwedge_{e \in A} \neg e, \varnothing, p)$ for every $p \in P$. The initial phase is the phase corresponding to the `main`-process. Fig. 4 shows the phase event automaton for the CSP process given in section 2.

Translation of Object-Z. The Object-Z part is translated into a two-phase automaton. The initial phase restricts the state with the predicates in `Init`. This phase is connected with the main phase by a single edge allowing no events or variable changes. The main phase has one edge for each event that allows exactly this event, keeps all variables not in the Δ-list constant, and restricts the variables in accordance with the communication schema. Every phase further has the stuttering edge, disallowing all events and variable changes.

Translation of Duration Calculus. Despite their expressiveness it is possible to translate each DC counterexample formula to a phase event automaton. The basic algorithm is the same that is used for negating a finite automaton, namely the power set construction. As defined in section 2, a counterexample formula consists of several phases $phase_1$ **;** $\ldots$ **;** $phase_n$. The idea is to remember for each of these phases, whether the time interval from the start of the system to the current time satisfies the formula

$$\text{true} \mathbin{;} phase_1 \mathbin{;} \ldots \mathbin{;} phase_i, \quad 1 \leq i \leq n$$

A phase of the PEA is labelled by a set of those phases of the counterexample, for which the above formula holds. For a phase with a lower bound on its duration there is an additional flag that signals if the above formula would only hold without the lower bound. Each phase $phase_i$ with a time bound needs a clock c_i that measures the duration of the phase. Because only either an upper or a lower bound on the duration is allowed it is obvious, when to reset those clocks (as often as possible for upper bounds; only when we have to reenter the phase for lower bounds).

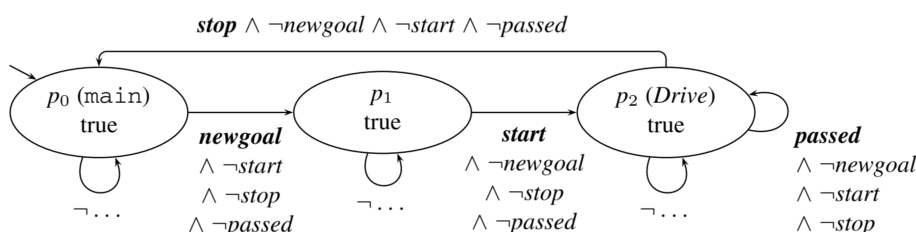

Fig. 4. Translation of CSP part

We implemented a tool that converts a counterexample formula into a phase event automaton. Due to space limitations the algorithm cannot be explained in full detail here. One of the resulting automata was already shown in Fig. 3. The automaton for the other formula is given in [12].

5 A Constraint-Based Semantics for PEA

To give semantics for CSP-OZ-DC (and phase event automata) in a domain where model-checking is possible, we use an "old-fashioned recipe for real-time" [1, 15]. The runs are described by sequences of states, where each state gives the values of all variables for a given time interval. Lamport adds one variable to denote the time since the start of the system. As we are not interested in absolute time, we have a variable len instead, denoting the length of the time interval. Events are represented by changes of Boolean variables as in section 2. Since we want to verify safety properties of phase event automata using a discrete time model checker, we translate the automata into discrete transition systems (with constraints) in such a way that the transition system generate as runs exactly the above sequences of interval states.

5.1 Transition Constraint Systems

A *transition constraint system* (TCS) $\mathcal{T} = (Loc, Var, Init, Trans)$ is a 4-tuple such that

- *Loc* is a set (of locations),
- $Var \subseteq \mathcal{V}$ is a finite set of unprimed (state) variables,
- $Init : Loc \to \mathcal{L}(Var)$ assigns a (state) constraint to every location, and
- $Trans : Loc \times Loc \to \mathcal{L}(Var \cup Var')$ assigns a (transition) constraint to every pair of locations.

We can view *Init*, which is a vector of state constraints, as vector of sets of initial states of a transition system. Likewise, *Trans* is matrix of transition constraints, which can be viewed as a matrix of relations between pre-states (valuations of the unprimed variables) and post-states (valuations of the primed variables) of a transition system. See [12] for examples of transition constraint systems.

We define the *parallel composition* $\mathcal{T}_1 \parallel \mathcal{T}_2$ of two transition constraint systems $\mathcal{T}_1$ and $\mathcal{T}_2$ (where $\mathcal{T}_i = (Loc_i, Var_i, Init_i, Trans_i)$, $i = 1, 2$) as the TCS $\mathcal{T} = (Loc_1 \times Loc_2, Var_1 \cup Var_2, Init, Trans)$ such that for all locations $(\ell_1, \ell_2), (\ell_1', \ell_2') \in Loc_1 \times Loc_2$,

- $Init((\ell_1, \ell_2)) = Init_1(\ell_1) \wedge Init(\ell_2)$, and
- $Trans((\ell_1, \ell_2), (\ell_1', \ell_2')) = Trans_1(\ell_1, \ell_1') \wedge Trans(\ell_2, \ell_2')$.

Let $\mathcal{T} = (Loc, Var, Init, Trans)$ be a TCS. A *state* of $\mathcal{T}$ is a pair (ℓ, α) of a location $\ell \in Loc$ and 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 an infinite sequence of states $\langle(\ell_0, \alpha_0), (\ell_1, \alpha_1), \ldots\rangle$ such that

1. $\alpha_0 \models Init(\ell_0)$, and
2. for all $i \geq 0$, $\alpha_i \cup \alpha_{i+1}' \models Trans(\ell_i, \ell_{i+1})$.

We call a state (ℓ, α) *reachable* if there is a run $\langle(\ell_0, \alpha_0), (\ell_1, \alpha_1), \ldots\rangle$ of $\mathcal{T}$ such that $(\ell, \alpha) = (\ell_i, \alpha_i)$ for some $i \geq 0$. By $Reach(\mathcal{T})$, we denote the set of reachable states of $\mathcal{T}$. As is easily seen, the notion of run is compatible with parallel composition.

Lemma 3. *For TCS $\mathcal{T}^1$ and $\mathcal{T}^2$, $\langle((\ell_0^1, \ell_0^2), \alpha_0), ((\ell_1^1, \ell_1^2), \alpha_1), \ldots\rangle$ is a run of $\mathcal{T}^1 \parallel \mathcal{T}^2$ if and only if $\langle(\ell_0^1, \alpha_0|_{Var^1}), (\ell_1^1, \alpha_1|_{Var^1}), \ldots\rangle$ and $\langle(\ell_0^2, \alpha_0|_{Var^2}), (\ell_1^2, \alpha_1|_{Var^2}), \ldots\rangle$ are runs of $\mathcal{T}^1$ and $\mathcal{T}^2$, respectively.*

5.2 Translation of PEA to TCS

We now present a translation of a phase event automaton $\mathcal{A} = (P, V, A, C, E, s, I, P_0)$ into a transition constraint system $\mathcal{T}(\mathcal{A}) = (Loc, Var, Init, Trans)$. There are two key features of this translation. First, continuous transitions of the automaton (which are implicit in the timed automata model) are translated into explicit discrete transitions. Second, the distinction between state and event variables is given up in favour of state variables; events are modelled by state change. To this end, we transform formulas $\varphi \in \mathcal{L}(\mathcal{V})$ into formulas $\varphi[e \not\approx e'/e]_{e \in \mathbf{Events}} \in \mathcal{L}(\mathcal{V} \cup \mathbf{Events}')$ by replacing each event variable $e \in \mathbf{Events}$ with a disequation $e \not\approx e'$. Furthermore, we introduce two auxiliary

variables, disc of type Bool (indicating whether the next transition is a discrete one) and len of type Time (recording the length of the time interval of a continuous transition). These auxiliary variables are reserved specially for translating PEA to TCS, therefore they may not be used by any PEA. Formally, the translation $\mathcal{T}(\mathcal{A})$ is given by:

- $Loc = P$.
- $Var = V \cup A \cup C \cup \{\mathsf{len}, \mathsf{disc}\}$.
- For all $p \in P$,

$$Init(p) = \begin{cases} \neg\mathsf{disc} \wedge \bigwedge_{c \in C} c \approx 0 \wedge s(p) \wedge I(p) \wedge \mathsf{len} > 0 & \text{if } p \in P_0, \\ \mathit{false} & \text{otherwise.} \end{cases}$$

- For all $p_1, p_2 \in P$,

$$Trans(p_1, p_2) = \begin{cases} Inv(p_2)' \wedge \left(Cont \vee \bigvee_{(p_1,g,X,p_2) \in E} Disc(g, X) \right) & \text{if } p_1 = p_2, \\ Inv(p_2)' \wedge \left(\bigvee_{(p_1,g,X,p_2) \in E} Disc(g, X) \right) & \text{if } p_1 \neq p_2, \end{cases}$$

  where the formulas $Inv(p_2)$, $Cont$ and $Disc(g, X)$ are given by:

$$Inv(p_2) = \mathsf{len} > 0 \wedge s(p_2) \wedge I(p_2)$$

$$Cont = \neg\mathsf{disc} \wedge \mathsf{disc}' \wedge \bigwedge_{c \in C} c' \approx c + \mathsf{len} \wedge \bigwedge_{x \in V \cup A} x' \approx x$$

$$Disc(g, X) = \mathsf{disc} \wedge \neg\mathsf{disc}' \wedge g[e \not\approx e'/e]_{e \in \mathbf{Events}} \wedge \bigwedge_{c \in X} c' \approx 0 \wedge \bigwedge_{c \in C \setminus X} c' \approx c$$

Here, $Inv(p)$ expresses the invariant constraints (state and clock) associated with phase p, $Cont$ relates pre- and post-states in a continuous transition, and $Disc(g, X)$ relates pre- and post-states of a discrete transition (with guard g and resetting the clocks in X). See [12] for samples of PEA translated to TCS.

5.3 Semantical Correctness of the Translation

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 $\mathcal{T}(\mathcal{A})$. Given a run $r = \langle (\ell_0, \alpha_0), (\ell_1, \alpha_1), \ldots \rangle$ of the TCS $\mathcal{T}(\mathcal{A})$, we define an infinite sequence $r_{\mathcal{A}} = \langle (p_0, \beta_0, \gamma_0), t_0, Y_0, (p_1, \beta_1, \gamma_1), t_1, Y_1, \ldots \rangle$ such that for all $i \geq 0$, $p_i = \ell_{2i}$, $\beta_i = \alpha_{2i}|_V$, $\gamma_i = \alpha_{2i}|_C$, $t_i = \alpha_{2i}(\mathsf{len})$ and $Y_i = \{e \in A \mid \alpha_{2i+1}(e) \neq \alpha_{2i+2}(e)\}$. 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 $\mathcal{T}(\mathcal{A})$. See [12] for a proof.

Theorem 4. *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 r of $\mathcal{A}$ there is a run $\hat{r}$ of $\mathcal{T}(\mathcal{A})$ such that $\hat{r}_{\mathcal{A}} = r$.*

Note that the proof of the first half of the theorem requires convexity of the clock invariants of the PEA. In fact, without convexity, $\mathcal{T}(\mathcal{A})$ might show runs that are artefacts of the translation and do not correspond to runs of $\mathcal{A}$.

As a corollary, we obtain a correspondence between the reachable states of $\mathcal{A}$ and $\mathcal{T}(\mathcal{A})$, which justifies doing reachability analysis on the discrete system $\mathcal{T}(\mathcal{A})$ instead of the timed automaton $\mathcal{A}$. To state the correspondence formally, we translate a state (ℓ, α) of $\mathcal{T}(\mathcal{A})$ into a state $(\ell, \alpha)_{\mathcal{A}} = (\ell, \alpha|_V, \alpha|_C)$ of $\mathcal{A}$. The corollary claims that this translation is a surjective mapping from the reachable states of $\mathcal{T}(\mathcal{A})$ to the reachable states of $\mathcal{A}$; see [12] for a proof.

Corollary 5. *Let $\mathcal{A}$ be a PEA and $\mathcal{T}(\mathcal{A})$ its TCS translation.*

1. *For all states (ℓ, α) of $\mathcal{T}(\mathcal{A})$, if $(\ell, \alpha) \in Reach(\mathcal{T}(\mathcal{A}))$ then $(\ell, \alpha)_{\mathcal{A}} \in Reach(\mathcal{A})$.*
2. *For all states (p, β, γ) of $\mathcal{A}$, if $(p, \beta, \gamma) \in Reach(\mathcal{A})$ then there is state $(\ell, \alpha) \in Reach(\mathcal{T}(\mathcal{A}))$ such that $(\ell, \alpha)_{\mathcal{A}} = (p, \beta, \gamma)$.*

Note that the translation of the reachable states of the TCS $\mathcal{T}(\mathcal{A})$ ignores variables that are not state variables of the PEA $\mathcal{A}$, i. e., the event variables in *A* and the auxiliary variables disc and len. However, the reachable states of $\mathcal{T}(\mathcal{A})$ are not more informative than the reachable states of $\mathcal{A}$, because the values of the event variables are irrelevant for reachability in $\mathcal{T}(\mathcal{A})$.

6 Model Checking TCS

We verify temporal properties of CSP-OZ-DC specifications by translating them to transition constraint systems, which we can model check. In this paper, we confine ourselves to the verification of state invariants, i. e., to checking whether a set of unsafe states (violating the invariant) is reachable from the initial states. It is well known that this implies the ability to verify arbitrary safety properties by augmenting the system with suitable monitors or test-automata [6].

For verification, we decided to use the constraint-based model checker ARMC [18], because its constraint solver can handle linear arithmetic over the reals, which is crucial for our approach to real-time. The model checker takes as input a transition constraint system and a set of unsafe states (given as a vector of constraints, like the initial states). Going backwards from the unsafe states, it tries to determine whether the initial states are reachable by alternating the following two steps.

1. Over-approximating the reachable states using predicate abstraction (w. r. t. a current set of abstraction predicates) in order to disprove reachability, i. e., to prove the invariant.
2. Under-approximating the reachable states using a bounded (yet precise) symbolic backwards reachability analysis in order to prove reachability, i. e., to detect real counterexamples (and to refine the set of abstraction predicates to exclude spurious counterexamples).

In general, this abstraction-refinement loop may not terminate. However, in practice it does terminate on numerous examples after a small number of iterations.

We would like to stress that the effectiveness and the performance of the model checker crucially depend on the constraints in the input. Both steps in the abstraction-refinement loop, computing a predicate abstraction and doing a symbolic reachability analysis, require to decide satisfiability of formulae in $\mathcal{L}$. Therefore, $\mathcal{L}$ should be a decidable class of constraints, e. g., linear arithmetic over the integers and reals as in our case study. Moreover, the solver for $\mathcal{L}$ should be performant in practice, since one run of the model checker may trigger thousands of calls to the solver.

6.1 Verification of the Case Study

To demonstrate our approach, we verified that our parameterised elevator never drives below the lowest or above the highest floor, i. e., we verified the invariant

$$Min \leq current \leq Max \ . \tag{1}$$

In order to model check, we translated the CSP-OZ-DC specification according to section 4 into a parallel product of four PEA, one for the CSP part, one for the OZ part and one for each DC formula. As described in section 5, each PEA was translated to a TCS; see [12] for the details. The parallel composition of these TCS together with the negation of the invariant were fed into the model checker ARMC, which proved the invariant in about 2 minutes[4] with two iterations of the abstraction-refinement loop. Recall that the CSP-OZ-DC specification as well as the invariant were parameterised by the symbolic constants *Min* and *Max*. Thus, we have verified the invariant for all elevators that are instances of the specification, independent of the actual size the state space of those instances.

Note that even the simple invariant (1) is a real-time property, despite it does not contain timing constraints. However, the invariant does depend on the timing constraints enforced by the DC formulas; in fact, erasing any of the two DC formulas from the CSP-OZ-DC specification causes (1) to be violated, which ARMC can demonstrate with counterexample traces in less than 20 seconds.

7 Conclusion

We presented a technique to model-check a combined specification written in CSP-OZ-DC by translating it into phase event automata. The semantics of CSP-OZ-DC used here is equivalent to the original one given in [13], however, it is defined in a different way. The three parts of the specification are separately translated into phase event automata, which are then joined by parallel composition. These automata have the notion of events, data variables and clocks, which allows to represent these concepts without encoding. Their special parallel composition is equivalent to CSP synchronised parallel composition and logical conjunction in Object-Z and Duration Calculus. These automata are further translated into transition constraint systems that are then checked by a constraint-based model-checker using the abstraction-refinement paradigm. The

[4] Measured on a standard Linux PC (2.6 GHz Pentium 4, 512 MB RAM).

model-checker can work with symbolic values, thus admits checking parameterised specifications.

7.1 Related Work

In [13] we already presented a model-checking algorithm using the model-checker Uppaal for timed automata. However, it could only handle a very restricted set of Duration Calculus that could not refer to state variables. Also it could only handle finite system.

In [7] a translation from TCOZ, a combination of Timed-CSP and Object-Z, to Timed Automata is presented. In TCOZ timing behaviour is not separated but mixed with the CSP part and the translation closely follows the structure of the Timed-CSP part. This approach lacks support for infinite data.

A bounded model-checking (BMC) approach for checking validity of dense-time Duration Calculus was first presented in [8] and is the basis for the tool IDLVALID [19]. However, BMC can only find counter-examples upto a given length and also does not support infinite data.

Closest to our model of phase event automata are the timed automata of Kronos [24], which use the same model of clocks and the same synchronisation on events but lack the data part, and phase automata [23], where the idea of synchronisation over states is taken from. In many other automata models, e.g., state charts, there is a shared data space in the form of global variables, that can be read from and written to by any component. This leads to unexpected side-effects though, for example, if a component that writes to the variable is added later.

HyTech [2] can also check parameterised systems. However the approach used there is complementary: HyTech finds the parameter values for which the system is safe, while in our approach safety is checked for all possible parameters values. Also HyTech can only have parameters in timing constraints.

There exist a number of other abstraction-refinement model checkers, for example BLAST [10], MAGIC [4] and SLAM [3]. These model checkers are tailored to check properties of sequential or multi-threaded imperative programs, often operating systems code, and they generally deal well with arrays and linear arithmetic over the integers. However, to our knowledge, none of the above model checkers supports reals, which are essential for model checking real-time systems.

7.2 Future Work

Currently the model-checker can only check for reachability. We would like to use the technique of test-automata [6] to reduce model-checking of DC-formulae to reachability. In this approach a parallel automaton checks the formula and reaches a certain state if the formula is violated. We are currently researching the class of Duration Calculus formulae that can be checked by this approach. It is even larger than the set of counterexample formulae.

The above approach only allows safety properties. However, there exists an extension of ARMC, the model-checker used here, that allows to check liveness properties [16]. It can only check for fair termination, but with the idea of test automata it is possible to check for liveness properties given in Duration Calculus extended by liveness [20].

References

1. M. Abadi and L. Lamport. An old-fashioned recipe for real time. In *Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 1–27. Springer, 1992.
2. R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Trans. Software Engineering*, 22:181–201, 1996.
3. T. Ball and S. K. Rajamani. The SLAM toolkit. In *CAV'01*, pages 260–264. Springer, 2001.
4. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE'03*, pages 385–395, 2003.
5. G. Delzanno and A. Podelski. Model checking in CLP. In *TACAS'99*, pages 223–239, 1999.
6. H. Dierks and M. Lettrari. Constructing test automata from graphical real-time requirements. In *FTRTFT'02*, volume 2469 of *LNCS*, pages 433–454, 2002.
7. J.S. Dong, P. Hao, S.C. Qin, J. Sun, and W. Yi. Timed patterns: TCOZ to timed automata. In *ICFEM'04*, volume 3308 of *LNCS*, pages 483–498. Springer, 2004.
8. M. Fränzle. Take it NP-easy: Bounded model construction for duration calculus. In *FTRTFT'02*, volume 2469 of *LNCS*, pages 234–264. Springer, 2002.
9. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV'97*, pages 72–83, 1997.
10. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL'02*, pages 58–70. ACM Press, 2002.
11. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
12. J. Hoenicke and P. Maier. Model-checking of specifications integrating processes, data and time. Technical Report 5, SFB/TR 14 AVACS, http://www.avacs.org/, 2005.
13. J. Hoenicke and E.-R. Olderog. Combining specification techniques for processes data and time. In *IFM'02*, volume 2335 of *LNCS*. Springer, May 2002.
14. 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), 2002.
15. L. Lamport. The temporal logic of actions. *ACM TOPLAS*, 16:872–973, 1994.
16. A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL'05*, pages 132–144. ACM Press, 2005.
17. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
18. A. Rybalchenko. A model checker based on abstraction refinement. Master's thesis, Universität des Saarlandes, Saarbrücken, Saarland, September 2002.
19. B. Sharma, P.K. Pandya, and S. Chakraborty. Bounded validity checking of interval duration logic. In *TACAS'05*, volume 3440 of *LNCS*, pages 301–316. Springer, 2005.
20. J. U. Skakkebæk. Liveness and fairness in duration calculus. In *CONCUR'94*, pages 283–298, 1994.
21. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.
22. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International Series in Computer Science, 2nd edition, 1992.
23. J. Tapken. *Model-Checking of Duration Calculus Specifications*. PhD thesis, University of Oldenburg, June 2001.
24. S. Yovine. Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1(1+2), October 1997.
25. C. Zhou and M.R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. EATCS: Monographs in Theoretical Computer Science. Springer, 2004.
26. C. Zhou, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.