Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Master's Thesis

Verifying Networks of Phase Event Automata

submitted by

Walid Haddad

on 9. March 2009

Supervisor
Prof. Bernd Finkbeiner, Ph.D.

Advisor
Klaus Dräger

Reviewers
Prof. Bernd Finkbeiner, Ph.D.
Prof. Dr. Reinhard Wilhelm

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, 9. March, 2009

Walid Haddad

Contents

1 Introduction **2**
- 1.1 Overview and Related Work . . . 2
- 1.2 Motivation and Organization . . . 3

2 CSP-OZ-DC **6**
- 2.1 Overview . . . 6
- 2.2 Communicating Sequential Processes (CSP) . . . 6
  - 2.2.1 Example . . . 8
- 2.3 Object-Z (OZ) . . . 8
  - 2.3.1 Example . . . 8
- 2.4 Duration Calculus (DC) . . . 9
  - 2.4.1 Example . . . 9
- 2.5 Combination of CSP, OZ, and DC . . . 10

3 From CSP-OZ-DC to Phase Event Automata **11**
- 3.1 Overview and Related Work . . . 11
- 3.2 Phase Event Automata (PEAs) . . . 12
  - 3.2.1 Basic Notations . . . 12
  - 3.2.2 Formal Definition . . . 13
  - 3.2.3 Product Construction of Phase Event Automata . . . 14
- 3.3 Translating CSP-OZ-DC . . . 15
  - 3.3.1 Translating CSP . . . 15
  - 3.3.2 Translating Object-Z . . . 15
  - 3.3.3 Translating Duration Calculus . . . 16

4 From Phase Event Automata to Transition Constraint Systems **18**
- 4.1 Transition Constraint Systems (TCS) . . . 18
- 4.2 Translating PEAs into transition constraint systems . . . 20

5 An Improved Translation Method to Verify PEA-Networks of CSP-OZ-DC Specifications **22**
5.1 Overview . . . 23
5.2 Analysis . . . 23
5.3 Generating Event-locks . . . 24
5.3.1 Motivation . . . 24
5.3.2 Method . . . 25
5.3.3 Example . . . 26
5.4 Transition Constraint Systems for PEA Networks . . . 26
5.4.1 Overview and Definition . . . 26
5.4.2 Example . . . 32
5.4.3 The Init Part . . . 37
5.4.4 Tick-negotiate Sub-steps . . . 38
5.4.5 The COMMIT Sub-step . . . 39
5.4.6 The Refresh Sub-step . . . 40
5.4.7 CSP Sub-steps with Lock-variable Preconditions . . . 40
5.4.8 OZ Sub-steps . . . 44
5.4.9 DC Sub-steps . . . 46
5.4.10 TF sub-step . . . 47
5.4.11 CommitVar Sub-step . . . 47

6 Example and Analysis **52**
6.1 Example: Elevator . . . 52
6.2 Analysis . . . 56

7 Conclusion **60**
7.1 Summary and Final Thoughts . . . 60
7.2 Future Work . . . 61

List of Figures

1.1 From specification in CSP-OZ-DC to model checking in ARM-C/SLAB – with a given test formula. 4
1.2 From specification in CSP-OZ-DC to model checking in ARM-C/SLAB – with a given test formula – (Improved) 4
1.3 Three phase event automata – first example 5

2.1 Elevator. 7
2.2 OZ specification schemas of the elevator example 9
2.3 CSP-OZ-DC class 10

3.1 Phase event automaton for a watchdog. 12
3.2 Phase event automaton for the CSP specification of the elevator. 17
3.3 Phase event automaton for the OZ specification of the elevator. . 17
3.4 Phase event automaton for the second DC formula of the elevator. 17

5.1 Example: Phase event automata 27
5.2 Example: Phase event automata and event-locks 27
5.3 Procedures for identifying locked phases and adding locks . . . 28
5.4 The order of sub-steps 29
5.5 CSP automata of the network $\mathcal{N}$ 33
5.6 Lock-events for the CSP automata of the network $\mathcal{N}$ 34
5.7 OZ automata of the network $\mathcal{N}$ 35
5.8 DC automata of the network $\mathcal{N}$ 36
5.9 TF automaton 36
5.10 Without adding lock-variable preconditions 43
5.11 After adding lock-variable preconditions 43
5.12 TF sub-step – Error phase reachability 48

6.1 Phase event automata of $\mathcal{N}_{Elevator}$ 54

6.2 Phase event automata of $\mathcal{N}_{Elevator} - (Door)$ 55
6.3 Test automaton of $\mathcal{N}_{Elevator}$ 56

List of Tables

5.1 Analyzing the components of a network of phase event automata of a CSP-OZ-DC specification 23
5.2 The components of the automata of network $\mathcal{N}$ 32
5.3 The `Tick-negotiate` part of the transition constraint system of $\mathcal{N}$ 39
5.4 The `COMMIT` part of the transition constraint system of $\mathcal{N}$ 40
5.5 The `Refresh` part of the transition constraint system of $\mathcal{N}$ . . . 40
5.6 CSP-automata part of the transition constraint system of $\mathcal{N}$ . . 49
5.7 OZ part of the transition constraint system of $\mathcal{N}$ 50
5.8 DC part of the transition constraint system of $\mathcal{N}$ 51
5.9 TF and the CommitVAR parts of the transition constraint system of $\mathcal{N}$ 51

6.1 The *Trans* part of $\mathcal{T}(\mathcal{N}_{Elevator})$ 58
6.2 The *Trans* part of $\mathcal{T}(\mathcal{N}_{Elevator})$ (2) 59

Abstract

There is a growing interest in research related to presenting correct and efficient model checking approaches for complex computing systems. Abstraction model checking is one example which proved its success as a model checking approach for infinite-state systems.

In this thesis, we present an approach for translating *CSP-OZ-DC* specifications into *transition constraint systems* (as a part of a model checking approach for infinite-state systems). In previous works, the CSP-OZ-DC specifications are translated into phase event automata, then, the product of the phase event automata is translated into a transition constraint system. For large specifications, the obtained transition constraint system can result in a large number of transitions. In our approach, we provide an improved representation of transition constraint systems consisting of a notably lower number of transitions in the transition constraint system with the aim of achieving faster model checking for available model-checkers.

1
Introduction

1.1 Overview and Related Work **2**
1.2 Motivation and Organization **3**

1.1 Overview and Related Work

Complex computing systems find increasing deployment in safety critical systems like nuclear power plants, aeroplanes, and modern cars where failures can have catastrophic consequences. Different techniques for studying and analyzing such systems were presented in research.

Formal verification is one technique which is described as a formal mathematical proof of the correctness of algorithms underlying a system with respect to a certain formal specification or property; it can be used for various systems. To verify a system, a rigorous behavioral and performance model of the system is needed.

Many specification languages used to specify a system model can only deal with restricted parts of a specification (*e.g.* processes or data). However, other specification languages try to cover multiple aspects of specification, such as *CSP-OZ*[Fis00], *TCOZ*[MD98], *RT-Z*[Süh99] and *CSP-OZ-DC*[Hoe06].

CSP-OZ-DC combines three specification languages: CSP, Object-Z (OZ), and Duration Calculus (DC). Highly expressive specification languages, such as CSP-OZ-DC, are not usually suited for automated verification. In [Hoe06] and [HM05], an approach to automatically verify CSP-OZ-DC specifications by model-checking is presented. In more details, the specifications are translated to *transition constraint systems* (TCS) which are model-checked using constraint-based symbolic techniques [DP99] and predicate abstraction [GS97] with counterexample-driven abstraction refinement [BR01] [SCV03] [THS02]. Example model-checkers are:

- SLAB [IBW07], and
- ARMC [Ryb02]

To construct the TCS of a system specified in CSP-OZ-DC (in terms of *CSP-OZ-DC classes*), as done in [Hoe06] and [HM05], the specifications which are defined by the constituent languages of CSP-OZ-DC (CSP, Object-Z, and Duration calculus) are individually translated into *Phase Event Automata* (PEAs). Phase Event Automata are a new class of timed automata which preserve events and data variables of the specification; [Hoe06][HM05] provide a compositional semantics for CSP-OZ-DC based on them. The product of the resulting PEAs is constructed and translated into a TCS; see Fig.1.1 as an example which includes also a test formula.

1.2 Motivation and Organization

The aim of this thesis is to present an alternative approach for translating *CSP-OZ-DC* specifications into *transition constraint systems* as a part of a model checking approach for infinite-state systems. For large specifications, the obtained transition constraint system can result in a large number of transitions when using the approaches in [Hoe06] and [HM05].

Our new approach is depicted in Fig.1.2. It mainly includes the following steps:

1. The CSP, Object-Z, and Duration Calculus specifications and the test formula (TF) are translated to phase event automata which are then distinguished and labeled by their types: CSP, OZ, DC, or TF.
2. The phase event automata are explored in order to extract some information needed for optimizing the new approach.
3. The transition constraint system of a given network is constructed with a new method which avoids the construction the product automaton.

The result is a an improved transition constraint system with notably fewer transitions than those which result using previous approaches.

As an example, we present a simple idea shown in Fig.1.3 which shows a transition step of a network $\mathcal{N}$ of three phase event automata. As $A1$ and $A2$ synchronize on an event a, the third automaton $A3$ can either stay in its phase by taking a self-loop transition (stutters) or it can change its current phase. In the transition constraint system representation of $\mathcal{N}$ a set of transitions appear for the same synchronization step as $A3$ stutters:

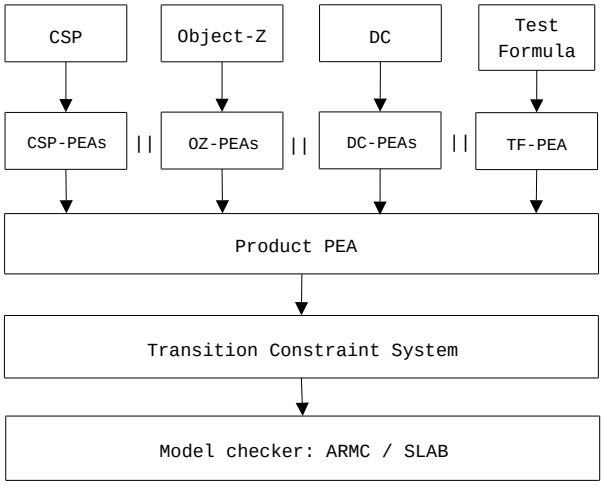


Figure 1.1: From specification in CSP-OZ-DC to model checking in ARMC/SLAB – with a given test formula.

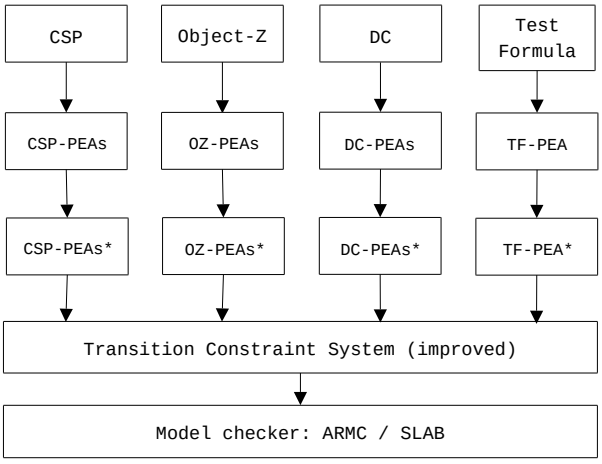


Figure 1.2: From specification in CSP-OZ-DC to model checking in ARMC/SLAB – with a given test formula – (Improved)

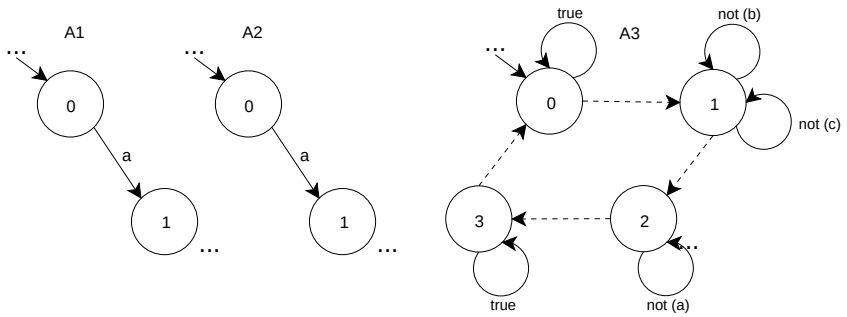


Figure 1.3: Three phase event automata with events a, b, and c. For all events e, not$(e) = \neg e$. A1, A2 and A3 have the id's 1, 2, and 3, respectively.

$$
\begin{gathered}
\ldots \\
\mathrm{pc}_1 = 0 \wedge \mathrm{pc}_2 = 0 \wedge \mathrm{pc}_3 = 0 \wedge a \wedge \mathrm{pc}_1' = 1 \wedge \mathrm{pc}_2' = 1 \wedge \mathrm{pc}_3' = 0 \\
\mathrm{pc}_1 = 0 \wedge \mathrm{pc}_2 = 0 \wedge \mathrm{pc}_3 = 1 \wedge a \wedge \neg b \wedge \mathrm{pc}_1' = 1 \wedge \mathrm{pc}_2' = 1 \wedge \mathrm{pc}_3' = 1 \\
\mathrm{pc}_1 = 0 \wedge \mathrm{pc}_2 = 0 \wedge \mathrm{pc}_3 = 1 \wedge a \wedge \neg c \wedge \mathrm{pc}_1' = 1 \wedge \mathrm{pc}_2' = 1 \wedge \mathrm{pc}_3' = 1 \\
\mathrm{pc}_1 = 0 \wedge \mathrm{pc}_2 = 0 \wedge \mathrm{pc}_3 = 2 \wedge a \wedge \neg a \wedge \mathrm{pc}_1' = 1 \wedge \mathrm{pc}_2' = 1 \wedge \mathrm{pc}_3' = 2 \\
\mathrm{pc}_1 = 0 \wedge \mathrm{pc}_2 = 0 \wedge \mathrm{pc}_3 = 3 \wedge a \wedge \mathrm{pc}_1' = 1 \wedge \mathrm{pc}_2' = 1 \wedge \mathrm{pc}_3' = 3 \\
\ldots
\end{gathered}
$$

In our approach, we avoid constructing the product automaton of the automata of the network which consists of one or more CSP-OZ-DC classes. The general idea is to translate each component of a network of phase event automata independently applying restrictions for each step which is done by the other components of the network. For example, if a chosen CSP-automaton performs a step which requires the synchronization on an event *e*, then this should be allowed by all the other components which must take part. The automata are distinguished by their types as they have different structures; this will also be taken into consideration.

In the next chapter, we start by defining CSP-OZ-DC and provide some examples. In chapter 3, we define the phase event automaton model and present a known translation method from CSP-OZ-DC into phase event automata. The translation method from phase event automata into transition constraint systems, as defined in [Hoe06][HM05], is introduced in chapter 4. In chapter 5, we introduce our new approach for constructing transition constraint systems. Chapter 6 shows an example. Chapter 7 is a conclusion.

2

CSP-OZ-DC

2.1 Overview **6**
2.2 Communicating Sequential Processes (CSP) **6**
2.2.1 Example 8
2.3 Object-Z (OZ) **8**
2.3.1 Example 8
2.4 Duration Calculus (DC) **9**
2.4.1 Example 9
2.5 Combination of CSP, OZ, and DC **10**

2.1 Overview

In the following, we provide a short definition of CSP-OZ-DC. We start by introducing three specification languages which are well-known in research: CSP, OZ, and DC. While CSP is known to describe behavioral aspects, such as sequential and concurrent behavior and synchronous communication, Object-Z describes complex data operations, and on the other hand, Duration Calculus is used to describe real-time requirements. A combination of these languages results in a combined language denoted by CSP-OZ-DC.

As an example, we describe the core of a controller of an elevator (see Fig.2.1) with each of the three specification languages; a step before we reach its combined specification in CSP-OZ-DC. The example was already introduced in [Hoe06][HM05].

2.2 Communicating Sequential Processes (CSP)

CSP is a specification language for communicating sequential processes which was introduced by Hoare [Hoa78] [Hoa85]. In its first version it

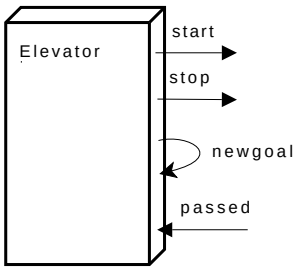


Figure 2.1: Elevator.

was presented in [Hoa78] essentially as a concurrent programming language rather than a process calculus; its theory was later developed and refined into its process algebraic form [Hoa85].

In CSP, processes communicate through instantaneous events which can be structured as (c, v) where c is a communication channel and v the communicated value. The alphabet of a process describes the set of events a process can communicate.

The grammar of the basic syntax of CSP defines processes as follows:

$$\mathcal{P} ::= \mathit{Stop} \mid \mathit{Skip} \mid a \rightarrow \mathcal{P} \mid \mathcal{P}_1; \mathcal{P}_2 \mid \mathcal{P}_1 \Box \mathcal{P}_2 \mid \mathcal{P}_1 \sqcap \mathcal{P}_2 \mid \mathcal{P}_1 \parallel_A \mathcal{P}_2 \mid \mathcal{P} \setminus \mathcal{A} \mid \mathcal{X}$$

Stop represents a deadlock process while *Skip* is a process which terminates immediately.

$a \rightarrow \mathcal{P}$ is a process which communicates a and starts $\mathcal{P}$. $\mathcal{P}_1; \mathcal{P}_2$ represents sequential composition of two processes $\mathcal{P}_1$ and $\mathcal{P}_2$ (*i.e.* $\mathcal{P}_1$ activates $\mathcal{P}_2$ with a termination event).

$\mathcal{P}_1 \Box \mathcal{P}_2$ represents a process where external choice decides whether $\mathcal{P}_1$ or $\mathcal{P}_2$ gets active. $\mathcal{P}_1 \sqcap \mathcal{P}_2$ represents a process where internal choice picks $\mathcal{P}_1$ or $\mathcal{P}_2$ to run.

$\mathcal{P}_1 \parallel_A \mathcal{P}_2$ represents parallel composition of two processes $\mathcal{P}_1$ and $\mathcal{P}_2$ with synchronization alphabet $\mathcal{A}$. $\mathcal{P} \setminus \mathcal{A}$ behaves like $\mathcal{P}$ except that the events from $\mathcal{A}$ have been hidden (or internalized). Finally, $\mathcal{X}$ represents the timeout process.

For the operational semantics of CSP we refer to [Plo83] which introduces a structural based semantics using labeled transition systems.

2.2.1 Example

With CSP we define admissible sequences of events. For the elevator example in Fig.2.1 we define the two processes:

$$\text{main} \stackrel{c}{=} newgoal \rightarrow \textit{start} \rightarrow \text{Drive}$$
$$\text{Drive} \stackrel{c}{=} (passed \rightarrow \text{Drive}) \,\square\, (\textit{stop} \rightarrow \text{main})$$

The initial process is `main`. The elevator chooses a new goal floor (*newgoal*), starts the engine (*start*) and switches to process `Drive`. In process `Drive` the elevator is moving and external choice (*passed* or *stop*) decides whether to stop and switch back to `main` or pass a floor and stay in `Drive`.

2.3 Object-Z (OZ)

Object-Z[Smi00] can be used to represent data state and the algorithmic part of a system. It was developed at the Software Verification Research Center in the University of Queensland as an object-oriented extension of the specification language Z[Spi88][ISO00]. OZ adds to Z language constructs that resemble the object-oriented paradigm, most notably, classes. It also supports polymorphism and inheritance.

OZ uses schemas to define constants, variables, the initial state and operations which can manipulate state variables. Schemas have different forms; they can be identified by names and they can include variable and constant definitions, predicates and operations such as assignments. For convenience, we will explain some of these in the following example.

2.3.1 Example

Fig.2.2 shows different schemas describing data aspects of the elevator. Schemas that describe constants are called *axiomatic definitions*. *Min* and *Max* are two integers defining the lowest and highest floor, respectively. The predicate *Min* < *Max* ensures that the lowest floor number is less than the highest one.

Variables (*current*, *goal* and *dir*) are defined in their own schemas. *Init* is a schema which defines the initial state of a system; initially, the direction is neutral (*dir = 0*).

The other schemas in Fig.2.2 define operations which can manipulate variables of the system. Those schemas are identified by the Δ sign in the upper row, which changes the values of variables in its parameter according

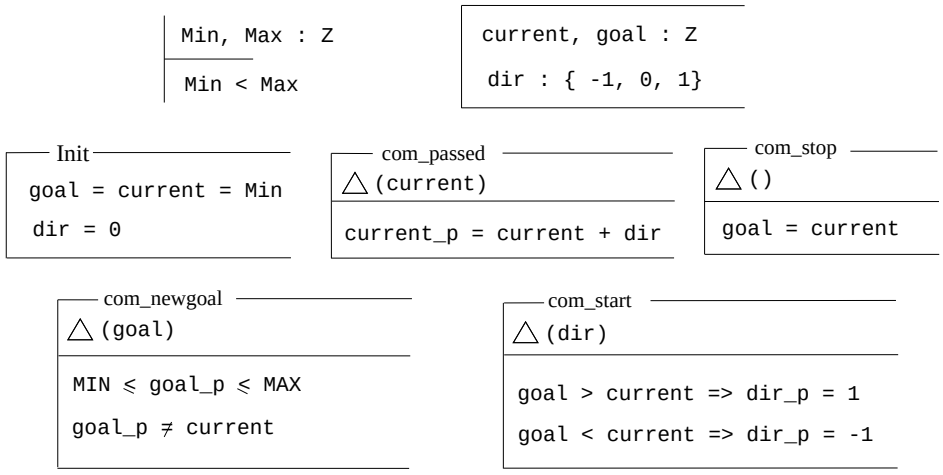


Figure 2.2: OZ specification schemas of the elevator example

to defined predicates and prevents all others from changing their values. Constraints in operations schemas can serve also as preconditions which restrict the behavior of an operation.

2.4 Duration Calculus (DC)

Duration Calculus[ZH04] is an interval logic for specifying real-time behaviors. In CSP-OZ-DC specifications only a restricted class of DC may be used since the full logic of DC is too powerful and undecidable[ZH04]; this restricted class is defined in [Rav94][SO99] as the class of *implementables* which can be reformulated to equivalent *counterexample formulae* according to [Tap01]. Counterexample formulae have the following general form:

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

2.4.1 Example

The controller of the elevator needs to fulfill some real-time properties which are specified using negated counterexample DC-formulae. We need to ensure that the elevator stops after reaching the goal floor and before it reaches

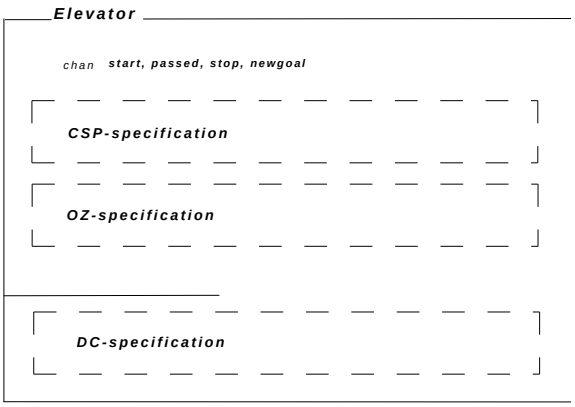


Figure 2.3: CSP-OZ-DC class

the next one. The following formula requires to have a minimum time of three seconds between two adjacent passed events:

$$\neg\Diamond\ (\updownarrow passed;\ l \leq 3;\ \updownarrow passed)$$

Additionally, we require that the elevator stops within two seconds; this can be expressed with the following DC-formula which states that it is impossible that the stop event does not occur even after the goal has been reached for more than two seconds:

$$\neg\Diamond\ (\lceil \text{current} \neq \text{goal} \rceil\ ;\ (\lceil \text{current} = \text{goal} \rceil \wedge\ l \geq 2 \wedge \boxminus \text{stop}))$$

2.5 Combination of CSP, OZ, and DC

Combining the three specification languages results in a higher-level language called CSP-OZ-DC. CSP-OZ-DC can be also seen as an extension to OZ since we define the combined language in terms of CSP-OZ-DC classes. The structure of the CSP-OZ-DC class of the elevator example is shown in figure 2.3. A semantics for CSP-OZ-DC based on phase event automata is introduced in the next chapter. For more details about CSP-OZ-DC classes, we refer to [Hoe06].

3

From CSP-OZ-DC to Phase Event Automata

3.1 Overview and Related Work 11
3.2 Phase Event Automata (PEAs) 12
3.2.1 Basic Notations 12
3.2.2 Formal Definition 13
3.2.3 Product Construction of Phase Event Automata 14
3.3 Translating CSP-OZ-DC 15
3.3.1 Translating CSP 15
3.3.2 Translating Object-Z 15
3.3.3 Translating Duration Calculus 16

3.1 Overview and Related Work

CSP-OZ-DC needs a semantics that is able to cover the basic entities of the constituent languages (events for CSP, data spaces and operations for OZ and time dependent observables for DC). There are different known timed automata models which are used for specifying real-time systems. Hoenicke discussed, in his work, why the phase event automaton model is suitable for describing CSP-OZ-DC classes.

Phase Event Automata are a new class of timed automata which were inspired by *phase automata* [Tap01], which were used to provide an operational semantics for Duration Calculus formulae. The phase automaton model, as it is, is not suitable for describing CSP-OZ-DC (for example it has no events) but it was extended to obtain the phase event automaton model.

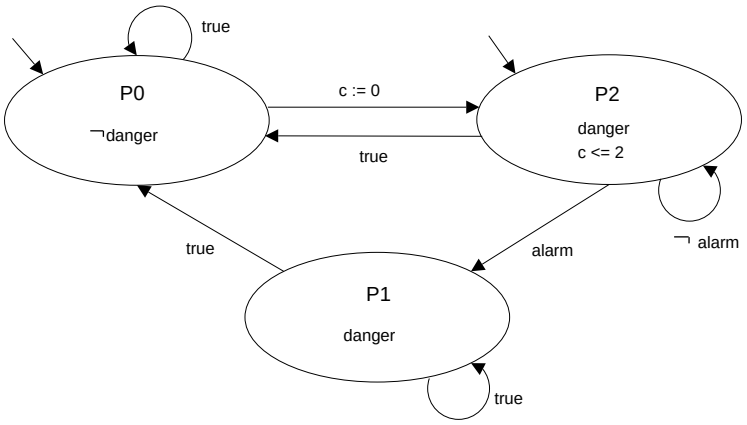


Figure 3.1: Phase event automaton for a watchdog.

3.2 Phase Event Automata (PEAs)

Phase Event Automata were designed to provide a compositional semantics for CSP-OZ-DC classes and we will see later that they serve as a bridge between CSP-OZ-DC and transition constraint systems (next chapter).

The phase event automaton model has a built-in notion of state assertions and events and it supports real-time using the same concept of clocks as in the timed automaton model[AD94].

As an example, Fig.3.1 shows a phase event automaton of a simple watch dog which uses a boolean variable *danger* and an event *alarm*. Additionally, the automaton is extended by a clock c which is used to measure the time the automaton stays in phase P2. The phases (states) are labeled by their names (P0, P1, and P2) and invariants (danger, ¬danger, and $c \leq 2$) which have to hold whenever the automaton is in the corresponding phase. Transitions are labeled with guards which can block the automaton from taking a step if a certain condition does not hold.

3.2.1 Basic Notations

Phases of a phase event automaton represent states of a systems and are described by formulae. The logic of these formulae is first order logic with equality. We denote the set of variables by $\widehat{\mathcal{V}}$. $\mathcal{L}$ denotes the class of first order formulae that are allowed in the specification. $\mathcal{L}(\mathcal{V})$ defines the formulae which correspond to the variables in $\mathcal{V} \subseteq \widehat{\mathcal{V}}$; similarly, $\mathcal{L}(\mathcal{C})$ corresponds to clock invariants. We require that $\mathcal{L}$ contains at least the class of quanti-

fier free formulae with boolean variables and linear arithmetic expressions over the reals. By $\mathcal{V}'$, we denote the primed version of the variables in $\mathcal{V}$; primed variables denote the state after a transition has been taken while the unprimed ones in $\mathcal{V}$ refer to the state before a transition is taken.

We call a set of clock constraints *convex* if they only allow conjunction of atoms of the form $c \leq T$ and $c < T$, where c is a clock variable and T a rational number.

Events are modeled as boolean variables and are a subset of $\mathcal{V}$. An event e is true if e occurs, otherwise false. Clocks are positive reals and are denoted by the set C $\subseteq \mathcal{V}$.

3.2.2 Formal Definition

Formally, a *phase event automaton* is defined as an 8-tuple $\mathcal{M}$ = $(\mathcal{P}, \mathcal{V}, \mathcal{A}, \mathcal{C}, \mathcal{E}, \int, \mathcal{I}, \mathcal{P}_0)$ where:

- $\mathcal{P}$ is a set of locations (phases)
- $\mathcal{V} \subseteq$ NAME is a set of typed *state variables*
- $\mathcal{A} \subseteq$ NAME is a set of boolean *event variables*
- $\mathcal{C}$ is a finite set of real-valued *clocks*
- $\mathcal{E} \subseteq \mathcal{P} \times \mathcal{L}(\mathcal{V} \cup \mathcal{V}' \cup \mathcal{A} \cup \mathcal{C}) \times P(\mathcal{C}) \times \mathcal{P}$ is a set of edges
- $\int : \mathcal{P} \rightarrow \mathcal{L}(\mathcal{V})$ is a labeling function which associates each phase with a predicate that must hold during this phase
- $\mathcal{I} : \mathcal{P} \rightarrow \mathcal{L}(\mathcal{C})$ is a function which assigns to each phase a clock invariant that must hold while the automaton is in this phase
- $\mathcal{P}_0 \subseteq \mathcal{P}$ is a set of possible initial phases

We require that for all $p \in \mathcal{P}$, $\mathcal{I}(p)$ is convex and that $\mathcal{E}$ contains a stuttering edge $(p, \neg e_1 \wedge \ldots \wedge \neg e_k \wedge v_1 = v_1' \wedge \ldots \wedge v_j = v_j', \emptyset, p)$ for some $\{e_1 \ldots e_k\} \subseteq \mathcal{A}$ and $\{v_1 \ldots v_j\} \subseteq \mathcal{V}$. The requirement of the stuttering transitions is needed for simplifying the definition of parallel composition, *i.e.* they are used to allow an automaton to make a step independently from others.

A *trace* of a phase event automaton $\mathcal{M}$ is a sequence of variables and clock evaluations, time delays and communicated events. A *state* of $\mathcal{M}$ is a triple (p, β, γ), where $p \in \mathcal{P}$, β is a $\mathcal{V}$-valuation, and γ is a $\mathcal{C}$-valuation. A *duration* is a positive real number.

A *run* of $\mathcal{M}$ is an infinite sequence $((p_0, \beta_0, \gamma_0), t_0, Y_0, (p_1, \beta_1, \gamma_1), t_1, Y_1, ...)$ of alternating states (p_i, β_i, γ_i), durations t_i and sets of events Y_i such that:

- $p_0 \in \mathcal{P}_0$
- For all $c \in \mathcal{C}$, $\gamma_0(c) = 0$.
- For all $i \geq 0$, $\beta_i \models \int(p_i)$.
- For all $i \geq 0$, $\gamma_i \models \mathcal{I}(p_i)$, $\gamma_i + t_i$, $t_i > 0 \models \mathcal{I}(p_i)$.
- For all $i \geq 0$ there is an edge $(p_i, g, X, p_{i+1}) \in \mathcal{E}$ such that $\beta_i \cup \beta'_{i+1} \cup (\gamma_i + t_i) \cup \mathcal{X}_{Yi} \models g$ and $\gamma_i + 1 = (\gamma_i + t_i)[X := 0]$

3.2.3 Product Construction of Phase Event Automata

Phase event automata synchronize on both events and states, as defined in [HO02]. Synchronization on events is done as in CSP: An event which is in the alphabet of both automta is only taken if both automata agree on it. Similarly, state synchronization ensures that a variable of both automata is changed only in a way which both automata allow.

For simplicity, we define the parallel composition of two automata $\mathcal{M}_1$ and $\mathcal{M}_2$, $\mathcal{M}_i = (\mathcal{P}_i, \mathcal{V}_i, \mathcal{A}_i, \mathcal{C}_i, \mathcal{E}_i, \int_i, \mathcal{I}_i, \mathcal{P}_{0,i})$. Let $\mathcal{M} = (\mathcal{P}, \mathcal{V}, \mathcal{A}, \mathcal{C}, \mathcal{E}, \int, \mathcal{I}, \mathcal{P}_0)$ be the product construction of the parallel composition of $\mathcal{M}_1$ and $\mathcal{M}_2$. Then $\mathcal{M}$ is defined as follows:

- $\mathcal{P} := \mathcal{P}_1 \times \mathcal{P}_2$,
- $\mathcal{V} := \mathcal{V}_1 \cup \mathcal{V}_2$,
- $\mathcal{A} := \mathcal{A}_1 \cup \mathcal{A}_2$,
- $\mathcal{C} := \mathcal{C}_1 \cup \mathcal{C}_2$ ($\mathcal{C}_1$ and $\mathcal{C}_2$ are disjoint sets, otherwise, clocks need to be renamed),
- $\int(p_1,p_2) = \int(p_1) \wedge \int(p_2)$,
- $\mathcal{I}(p_1,p_2) = \mathcal{I}(p_1) \wedge \mathcal{I}(p_2)$,
- $\mathcal{P}_0 := \mathcal{P}_{0,1} \times \mathcal{P}_{0,2}$,
- for each two edges $(p_i, g_i, X_i, p'_i) \in \mathcal{E}_i$, i = 1,2 in $\mathcal{M}_1$ and $\mathcal{M}_2$, $\mathcal{E}$ contains the edge $((p_1,p_2), g_1 \wedge g_2, X_1 \cup X_2, (p'_1,p'_2))$.

3.3 Translating CSP-OZ-DC

Each part of the CSP-OZ-DC specification is translated into an automaton and they are put in parallel. In the following, we introduce the translation methods for the different specification parts as proposed in [Hoe06]:

3.3.1 Translating CSP

To construct the equivalent phase event automaton $\mathcal{M}_{CSP}$, the operational semantics of CSP is used:

- Phases of $\mathcal{M}_{CSP}$ are labeled by processes.
- Each transition $p \xrightarrow{a} p'$ is an edge

  $$(p,\ a \wedge \bigwedge_{e \in A \setminus \{a\}} \neg e,\ \emptyset,\ p') \text{ of } \mathcal{M}_{CSP},$$

  where A is the alphabet of process `main`.
- A τ transition ($p \xrightarrow{\tau} p'$) which communicates no events is an edge

  $$(p,\ \bigwedge_{e \in A} \neg e,\ \emptyset,\ p') \text{ of } \mathcal{M}_{CSP}.$$

- For every $p \in \mathcal{P}_{CSP}$ there is a stuttering edge $(p,\ \bigwedge_{e \in A} \neg e,\ \emptyset,\ p)$.

Fig.3.2 shows the automaton which corresponds to the CSP part of the elevator example.

3.3.2 Translating Object-Z

The Object-Z part is translated into a phase (event) automaton $\mathcal{M}_{OZ}$ with a single phase p_{main}. $\mathcal{M}_{OZ}$ has the following characteristics:

- The variables of the automaton are the variables $Var(State)$ declared in the state schema of the CSP-OZ-DC class.
- p_{main} has one edge for each event which does not allow any other event and a stuttering edge disallowing all events and variable changes.
- The communication alphabet A of the automaton consists of all channels c for which the class contains an operation schema com_c.

- For each communication event the automaton contains a self-loop transition t; the guard of t demands that only the corresponding event occurs and that the pre- and post-state relate according to the communication schema.
- There is one initial edge to p_{main} that has the Init schema of the CSP-OZ-DC class as a guard.

The automaton in Fig.3.3 corresponds to the Object-Z part of the elevator example.

3.3.3 Translating Duration Calculus

The counterexample Duration Calculus formulae are translated separately into corresponding phase event automata.Each phase of the automaton is labeled by a set of phases of the counterexample such that the following formula holds:

$$\text{true};\mathit{phase}_1;\ldots;\mathit{phase}_i,\ 1 \leq i \leq \text{n}$$

Each phase phase_i with a time bound needs a clock c_i that measures the duration of the phase and is reset appropriately.

Fig.3.4 shows the automaton which corresponds to the following counterexample formula:

$$\neg\Diamond\ (\lceil \text{current} \neq \text{goal} \rceil\ ;\ (\lceil \text{current} = \text{goal} \rceil \wedge\ l \geq 2 \wedge \boxminus\ \text{stop})).$$

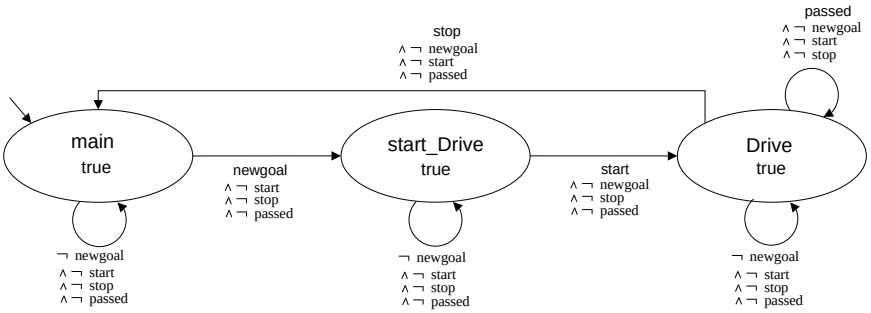
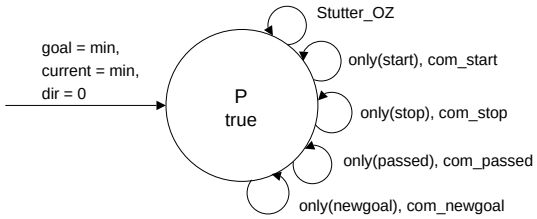


Figure 3.2: Phase event automaton for the CSP specification of the elevator.



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

Figure 3.3: Phase event automaton for the OZ specification of the elevator.

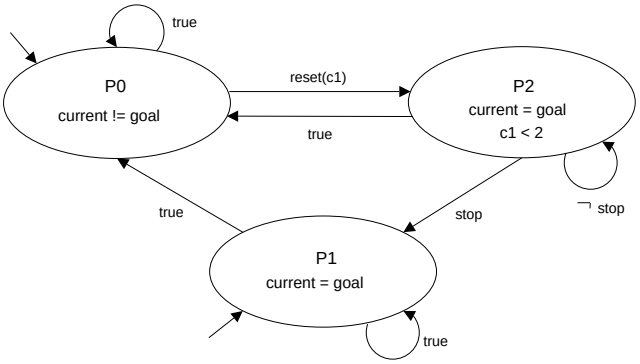


Figure 3.4: Phase event automaton for the second DC formula of the elevator.

4

From Phase Event Automata to Transition Constraint Systems

4.1 Transition Constraint Systems (TCS) 18
4.2 Translating PEAs into transition constraint systems . . . 20

In the first chapter, we have mainly introduced CSP-OZ-DC, its translation into phase event automata and a definition of a transition constraint system. In this chapter, we introduce a translation method which translates phase event automata into transition constraint systems as proposed in [HM05]; in the next chapter, we present our approach for constructing improved transition constraint systems.

4.1 Transition Constraint Systems (TCS)

[HM05] provides a domain for model-checking for CSP-OZ-DC using a constraint-based semantics for PEAs. Transition constraint systems represent a transition relation by a first-order formula where events are encoded as boolean variables and clocks are encoded as real-valued variables. A system is described by two formulae: One defines the initial state and the other defines the transition relation.

Formally, a TCS is a 4-tuple $T = (Var, Init, Trans)$, such that:

- $Var \subseteq \mathcal{V}$ is a finite set of (unprimed) state variables

- *Init* : $\mathcal{L}(Var)$ assigns a (state) constraint to every location
- *Trans* : $\mathcal{L}(Var \cup Var')$ assigns a (transition) constraint to every pair of locations

Init can be viewed as a vector of sets of initial states of a transition system. Likewise, *Trans* 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.

Let T = (*Loc*, *Var*, *Init*, *Trans*) be a transition constraint system. A state of T is a pair (l, α) of a location l and a Var-valuation α. A *run* of T is defined as a sequence of states $\langle s_1, ..., s_n \rangle$ such that

1. $s_1 \models$ *Init*, and
2. for all $i \geq 0$, s_n, $s'_{n+1} \models$ *Trans*.

A state s of T is *reachable* if there is a run $\langle s_1, ..., s_m \rangle$ such that $s_m = s$.

As an example, the following is a transition constraint system of the automaton in Fig.3.2 (PC is a variable used to store the current location; PC′ refers to the next current location):

- $Init_{CSP} = \text{PC} = 0$
- $Trans_{CSP} = \text{PC} = 0 \wedge \neg\, newgoal \wedge \neg\, start \wedge \neg\, stop \wedge \neg\, passed \wedge \text{PC}' = 0$
  $\vee\ \text{PC} = 0 \wedge newgoal \wedge \neg\, start \wedge \neg\, stop \wedge \neg\, passed \wedge \text{PC}' = 1$
  $\vee\ \text{PC} = 1 \wedge \neg\, newgoal \wedge \neg\, start \wedge \neg\, stop \wedge \neg\, passed \wedge \text{PC}' = 1$
  $\vee\ \text{PC} = 1 \wedge \neg\, newgoal \wedge start \wedge \neg\, stop \wedge \neg\, passed \wedge \text{PC}' = 2$
  $\vee\ \text{PC} = 2 \wedge \neg\, newgoal \wedge \neg\, start \wedge \neg\, stop \wedge \neg\, passed \wedge \text{PC}' = 2$
  $\vee\ \text{PC} = 2 \wedge \neg\, newgoal \wedge \neg\, start \wedge \neg\, stop \wedge passed \wedge \text{PC}' = 2$
  $\vee\ \text{PC} = 2 \wedge \neg\, newgoal \wedge \neg\, start \wedge stop \wedge \neg\, passed \wedge \text{PC}' = 0$

Definition 4.1. *Let T_1 and T_2 be two transition constraint systems, where T_i = (Loc$_i$, Var$_i$, Init$_i$, Trans$_i$), for i = 1,2. The parallel composition $T_1 \parallel T_2$ is defined as*

$$T = (Loc_1 \times Loc_2,\ Var_1 \cup Var_2,\ Init_1 \wedge Init_2,\ Trans),$$

such that for all locations (l_1, l_2),$(l'_1, l'_2) \in Loc_1 \times Loc_2$,

$$Trans((l_1, l_2),(l'_1, l'_2)) = Trans(l_1, l'_1) \wedge Trans(l_2, l'_2).$$

4.2 Translating PEAs into transition constraint systems

Let $\mathcal{M} = (\mathcal{P}, \mathcal{V}, \mathcal{A}, \mathcal{C}, \mathcal{E}, \int, \mathcal{I}, \mathcal{P}_0)$ be a phase event automaton. The translation in [HM05] makes sure that continuous transitions, which are implicit in the timed automaton model, are translated into explicit discrete transitions. Events are modeled by state change.

Two new variables are introduced $(disc : bool, len : TIME)$; $disc$ indicates whether the next transition represents a discrete one and len records the length of a time interval of a continuous transition.

Formally, the translation of $\mathcal{M}$ into $T(\mathcal{M}) = (Loc, Var, Init, Trans)$ is given by:

- $Loc = \mathcal{P}$
- $Var = \mathcal{V} \cup \mathcal{A} \cup \mathcal{C} \cup \{len, disc\}$
- For all $p \in \mathcal{P}$, $Init(p) = false$ if $p \notin \mathcal{P}_0$, otherwise:

$$Init(p) = \neg disc \wedge \bigwedge_{c \in \mathcal{C}} c \approx 0 \wedge \int(p) \wedge \mathcal{I}(p) \wedge len > 0$$

- For all p_1, $p_2 \in \mathcal{P}$,

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

where:

- $Inv(p_2) = \int(p_2) \wedge \mathcal{I}(p_2) \wedge len > 0$
  ($Inv(p)$ expresses the state and clock invariants of a phase p)
- $Cont = \neg disc \ \wedge \ disc' \wedge \ \bigwedge_{c \in \mathcal{C}} c' \approx c + len \ \wedge \ \bigwedge_{x \in \mathcal{V} \cup \mathcal{A}} x' \approx x$
  (*Cont* describes pre- and post-states in continuous transitions)
- $Disc(g, X) = disc \wedge \neg disc' \wedge g[e \not\approx e'/e]_{e \in Events} \wedge \ \bigwedge_{c \in X} c' \approx 0 \wedge \ \bigwedge_{c \in \mathcal{C} \setminus X} c' \approx c$
  (*Disc*(g,X) relates pre- and post-states of a discrete transition − X describes the set of clocks which are to be reset)

[Hoe06] proved the semantical correctness of this translation.

The example below is a translation of the DC-automaton in Fig.3.4 into a transition constraint system which also shows how clock invariants are handled:

- $\mathit{Init}_{DC} = \mathrm{PC} = 1 \wedge c_1 = \mathrm{len} \wedge \mathit{current} = \mathit{goal} \wedge \neg \mathit{disc}$
  $\vee\ \mathrm{PC} = 0 \wedge c_1 = \mathrm{len} \wedge \mathit{current} \neq \mathit{goal} \wedge \neg \mathit{disc}$

- $\mathit{Trans}_{DC} = \mathrm{PC} = 1 \wedge c_1' = c_1 \wedge \mathit{current}' = \mathit{goal}' \wedge \mathrm{PC}' = 1$
  $\wedge\ \mathit{disc} \wedge \neg \mathit{disc}'$
  $\vee\ \mathrm{PC} = 1 \wedge c_1' = c_1 + \mathrm{len} \wedge \mathit{current} = \mathit{current}' \wedge \mathit{goal} = \mathit{goal}'$
  $\wedge\ \mathrm{PC}' = 1 \wedge \mathit{disc}' \wedge \neg \mathit{disc}$
  $\vee\ \mathrm{PC} = 1 \wedge c_1' = c_1 \wedge \mathit{current}' \neq \mathit{goal}' \wedge \mathrm{PC}' = 0$
  $\wedge\ \mathit{disc} \wedge \neg \mathit{disc}'$
  $\vee\ \mathrm{PC} = 0 \wedge c_1' = c_1 \wedge \mathit{current}' \neq \mathit{goal}' \wedge \mathrm{PC}' = 0$
  $\wedge\ \mathit{disc} \wedge \neg \mathit{disc}'$
  $\vee\ \mathrm{PC} = 0 \wedge c_1' = c_1 + \mathrm{len} \wedge \mathit{current} = \mathit{current}' \wedge \mathit{goal} = \mathit{goal}'$
  $\wedge\ \mathrm{PC}' = 0 \wedge \mathit{disc}' \wedge \neg \mathit{disc}$
  $\vee\ \mathrm{PC} = 0 \wedge c_1' = 0 \wedge \mathit{current}' = \mathit{goal}' \wedge c_1' < 2 \wedge \mathrm{PC}' = 2$
  $\wedge\ \mathit{disc} \wedge \neg \mathit{disc}'$
  $\vee\ \mathrm{PC} = 2 \wedge \neg\ \mathit{stop} \wedge c_1' = c_1 \wedge \mathit{current}' = \mathit{goal}'$
  $\wedge\ c_1' < 2 \wedge \mathrm{PC}' = 2 \wedge \mathit{disc} \wedge \neg \mathit{disc}'$
  $\vee\ \mathrm{PC} = 2 \wedge c_1' = c_1 + \mathrm{len} \wedge \mathit{current} = \mathit{current}' \wedge \mathit{goal} = \mathit{goal}'$
  $\wedge\ \mathrm{PC}' = 2 \wedge \mathit{disc}' \wedge \neg \mathit{disc}$
  $\vee\ \mathrm{PC} = 2 \wedge \mathit{stop} \wedge c_1' = c_1 \wedge \mathit{current}' = \mathit{goal}' \wedge \mathrm{PC}' = 1$
  $\wedge\ \mathit{disc} \wedge \neg \mathit{disc}'$
  $\vee\ \mathrm{PC} = 2 \wedge c_1' = c_1 \wedge \mathit{current}' \neq \mathit{goal}' \wedge \mathrm{PC}' = 0$
  $\wedge\ \mathit{disc} \wedge \neg \mathit{disc}'$

5

An Improved Translation Method to Verify PEA-Networks of CSP-OZ-DC Specifications

5.1 Overview 23
5.2 Analysis 23
5.3 Generating Event-locks 24
5.3.1 Motivation 24
5.3.2 Method 25
5.3.3 Example 26
5.4 Transition Constraint Systems for PEA Networks 26
5.4.1 Overview and Definition 26
5.4.2 Example 32
5.4.3 The Init Part 37
5.4.4 Tick-negotiate Sub-steps 38
5.4.5 The COMMIT Sub-step 39
5.4.6 The Refresh Sub-step 40
5.4.7 CSP Sub-steps with Lock-variable Preconditions 40
5.4.8 OZ Sub-steps 44
5.4.9 DC Sub-steps 46
5.4.10 TF sub-step 47
5.4.11 CommitVar Sub-step 47

5.1 Overview

A method for constructing the transition constraint system of a network of phase event automata is to construct the product of the automata and then translate it into a transition constraint system as shown in Fig.1.1. Nevertheless, the exponential growth of the product, is a problem for model checking. To tackle this, we introduce an approach which leads to representations with notably fewer transitions.

In more details, the transitions of the new representation do not include complete transition-steps of a given PEA-network, but instead they represent what we call *sub-steps* (for example, a step done by a component of the network is a sub-step). A sequence of sub-steps can (under conditions which can be checked by the model checker) form a complete transition-step of a given PEA-network. Hence, there is no need to construct the whole state space of the PEA-network as done in previous works.

Note that, our approach is applicable for PEA-networks that correspond to CSP-OZ-DC specifications (as defined in [Hoe06][HM05] - see chapter 3). We also note that, for simplicity, we assume that events do not communicate values.

Characteristics	**CSP**	**OZ**	**DC**
Clocks	No	No	Yes
Nr. of phases	Not fixed	1	Not fixed
Invariants	Always true	Not fixed	Not fixed
Stuttering transitions	Yes (all phases)	Yes	Possible

Table 5.1: Analyzing the components of a network of phase event automata of a CSP-OZ-DC specification

5.2 Analysis

When observing the behavior a network of phase event automata, we notice that the components often perform stuttering steps. Also, we notice that those of CSP-OZ-DC specifications (which are constructed with the method

in chapter 3) have similar structures based on their types (CSP, ObjectZ, or DC).

For example, each phase of a CSP specification (CSP-automaton) has a stuttering transition, while clocks are only defined for DC-automata. Also, an OZ-automaton has only one phase - see Table 5.1.

Now, let $\mathcal{M}$ be a phase event automaton of a CSP-automaton. We notice that according to the structure of CSP-automata (see section 3.3.1):

1. If an event *e* is in the alphabet of $\mathcal{M}$, then *e* communicates only in transitions which prevent the communication of the remainder alphabets and is not allowed to communicate, otherwise. Hence, CSP-automata have to always agree on whether to allow the communication of one of its defined events or not.
2. If *e* is not in the alphabet of $\mathcal{M}$, then *e* is enabled in all transitions of $\mathcal{M}$.

As we will see later, these facts play an important role in our approach for constructing the transition constraint system.

In the following, we present our approach. First, we illustrate how we explore the automata of a given network $\mathcal{N}$ to extract some required information, then we discuss the construction method of the transition constraint system of $\mathcal{N}$.

5.3 Generating Event-locks

5.3.1 Motivation

We denote a phase of an automaton $\mathcal{M}$ of a PEA-network $\mathcal{N}$ as:

- A *non-locked* phase of e, if e is enabled in at least one of its outgoing transitions,
- otherwise, we denote it as a *locked* phase of e.

We associate predicates (*event-locks*) with transitions of automata of $\mathcal{N}$ which correspond to events and that play a role in indicating whether for an event e the current active phases of $\mathcal{N}$ are:

- Non-locked phases (this would mean that e may be able to communicate in the next network step) or

- if there exists at least one locked phase which would mean e cannot communicate in the next network step.

For instance, if the system enters a new state and if we already know which set of active phases S of the components are locked phases of e, then if $S = \emptyset$, we know that the next step of $\mathcal{N}$ may involve e, otherwise we know that e is not admissible in its next step (which means that at least one of the active phases of the current state is a locked phase of e); based on this, we will construct predicates which will be added to the transitions of the transition constraint system. These predicates either change the values of these variables or serve as preconditions for chosen event-enabling transitions. The advantage of this is discussed in later sections.

In the following, we present the method which obtains (if needed) a set of event-locks for transitions of each automaton of $\mathcal{N}$.

5.3.2 Method

In general, for each automaton $\mathcal{M}$ of a network $\mathcal{N}$, we associate[1] event-locks with their transitions as such:

Let $\mathcal{M}_i = (\mathcal{P}_i, \mathcal{V}_i, \mathcal{A}_i, \mathcal{C}_i, \mathcal{E}_i, \int_i, \mathcal{I}_i, \mathcal{P}_{0,i})$ be an automaton of $\mathcal{N}$, $1 < i < n$, where n is the number of the automata in $\mathcal{N}$. For each $e \in \mathcal{E}_i$ which is shared between two or more CSP-OZ-DC classes:

1. Identify the locked phases,
2. Introduce a new variable e_lock (the prefix e is the name of the event); we denote such variables as *lock-variables*,
3. Add the following predicates to each $\mathcal{M}_i$ of $\mathcal{N}$:
   - For each locked phase *w.r.t.* e, associate the predicate
     $$(i)\ e\_lock' = e\_lock - 1$$
     with all outgoing transitions with a non-locked destination phase.
   - For each non-locked phase *w.r.t.* e, associate the predicate
     $$(ii)\ e\_lock' = e\_lock + 1$$

[1] Note that, event-locks are not added to transition-guards of the automata but are generated and added later to the translated transitions of the transition constraint system. This does not violate the behavior of the network since the component transitions are only taken sequentially in the transition constraint system as we will define it later.

with all outgoing transitions with a locked destination phase;

- initially, e_lock is set to a value υ, where υ is the number of the active initial locked phases (in other words, υ is possibly set to different values depending on the different combinations which correspond to the possible initial states of the system). This is to be defined in the `Init` part of the transition constraint system of a network (as we will see later).

Obviously, e_lock cannot have a negative value. Furthermore, the condition $e\_lock = 0$ is true whenever the system is active in only non-locked phases of its components. By construction, the initial value of e_lock is determined by the number of the active locked initial phases of a network. The algorithm for solving this problem ensures that $e\_lock > 0$, if there is at least one active locked phase. The pseudo-code is shown in Fig.5.3.

The new generated predicates are used later to block transitions of CSP-automata when we know in advance that they shouldn't be taken. This idea and advantage of this will be clear after we discuss how the transition constraint system is to be constructed.

5.3.3 Example

Consider the example in Fig.5.1 which shows two automata of a given PEA-network $\mathcal{N}$. Suppose that the only defined event for both automata is a. The locked phases of PEA_1 are the phases 1 and 2, while PEA_2 has one locked phase for a which is phase 0. Initially, the variable a_lock is set to the value 1, since the only possible initial state of $\mathcal{N}$ is phase 0 in PEA_1 and also phase 0 in PEA2 and only one of those is a locked phase for a. The predicates $a\_lock' = a\_lock + 1$ and $a\_lock' = a\_lock - 1$ are associated appropriately with transitions of both automata as required and defined.

5.4 Transition Constraint Systems for PEA Networks

5.4.1 Overview and Definition

Let us first define some terms:

- A *complete step* refers to:

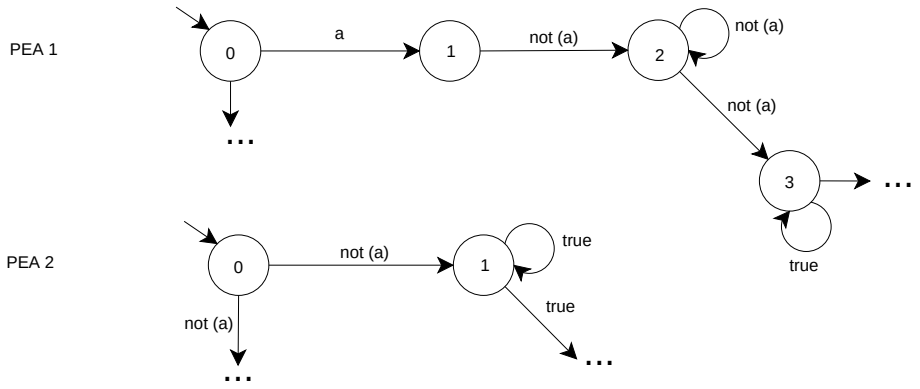


Figure 5.1: Example: Phase event automata

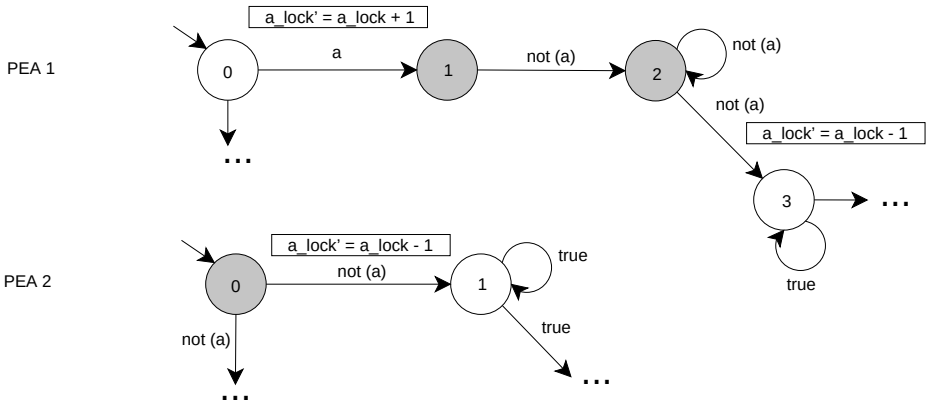


Figure 5.2: Example: Phase event automata and event-locks

$addLocks(pea)$:

for all $e \in pea.CSP\_shared\_events$ **do**
  $locked\_phases \leftarrow identifyLockedPhases(e, pea.phases)$;
  for all $p \in pea.phases$ **do**
    for all $t \in p.OutgoingTransitions$ **do**
      if $t.source \in locked\_phases$ *and* $t.destination \notin locked\_phases$ **then**
        $t.locks.add(e\_lock' = e\_lock - 1)$;
      end if
      if $t.source \notin locked\_phases$ *and* $t.destination \in locked\_phases$ **then**
        $t.locks.add(e\_lock' = e\_lock + 1)$;
      end if
    end for
  end for
end for

$identifyLockedPhases(event, pea)$:

$locked\_phases \leftarrow \emptyset$;
for all $p \in pea.phases$ **do**
  for all $t \in p.OutgoingTransitions$ **do**
    if $t.isNotEnabled(event)$ *and* $!(non\_locked\_phases.contains(p))$
    and $!(locked\_phases.contains(p))$ **then**
      $locked\_phases.add(p)$;
    end if
    if $t.isEnabled(event)$ **then**
      if $locked\_phases.contains(p)$ **then**
        $locked\_phases.remove(p)$;
      end if
      $non\_locked\_phases.add(p)$;
    end if
  end for
  if $pea.nr\_of\_outgoing\_transitions = 0$ **then**
    $locked\_phases.add(p)$;
  end if
end for
return $locked\_phases$;

Figure 5.3: Procedures for identifying locked phases and adding locks

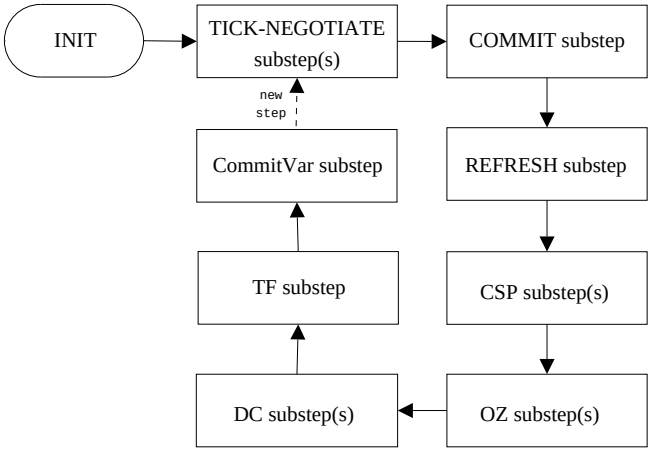


Figure 5.4: The order of sub-steps which is controlled by the variable $substep$; every round which starts in TICK-NEGOTIATE is a complete step

  - A continuous transition T_c of a network $\mathcal{N}$, and
  - individual transition-steps of components in a defined order which form together what represents a discrete transition T_d which follows T_c in $\mathcal{N}$.

  We require that, the conjunction of the individual transition-step-constraints is satisfiable.

- *sub-step*: A sub-step is a part of a complete step of $\mathcal{N}$; for example, transition-steps of an automaton of $\mathcal{N}$ correspond to a sub-step.

With the new translation method, the transitions of the transition constraint system correspond to sub-steps. We define a new variable $substep$ which plays a role in defining a complete step of a given PEA-network.

The new translation method of a network $\mathcal{N}$ constructs a transition constraint system. Its transitions can be categorized as follows:

- TICK-NEGOTIATE: A set of transitions represents the negotiation between the DC-automata *w.r.t* the time aspect.
- A COMMIT transition used to update clock variables.
- A REFRESH transition which resets the event variables.

- A set of transitions for each defined transition of each automaton of the given network. They are constructed under rules which differ for each type of automata (CSP, OZ, DC, or TF).
- A `CommitVar` transition used to update state variables.

Fig. 5.4 shows the required order of sub-steps. Every round starting in `TICK-NEGOTIATE` is a complete step.

Now, let $\mathcal{N}$ be a network of n phase event automata $\mathcal{M}_i = (\mathcal{P}_i, \mathcal{V}_i, \mathcal{A}_i, \mathcal{C}_i, \mathcal{E}_i, \int_i, \mathcal{I}_i, \mathcal{P}_{0,i})$, $1 \leq i \leq n$). Formally, the translation of $\mathcal{N}$ into a transition constraint system $\mathcal{T}(\mathcal{N}) = (\mathit{Loc}, \mathit{Var}, \mathit{Init}, \mathit{Trans})$ is defined as follows:

- *Loc* is a set of locations, such that, each sub-step (or $substep$ value) has a location, where the number of locations is in $O(n)$, and n is the number of automata of $\mathcal{N}$,
- $\mathit{Var} = \mathcal{V} \cup \mathcal{A} \cup \mathcal{C} \cup \{len, substep\} \cup Locks \cup LocVars \cup NextV$, where:
  - $\mathcal{V} = \mathcal{V}_1 \cup ... \cup \mathcal{V}_n$, $\mathcal{A} = \mathcal{A}_1 \cup ... \cup \mathcal{A}_n$, $\mathcal{C} = \mathcal{C}_1 \cup ... \cup \mathcal{C}_n$. Note that, events are defined as boolean variables.
  - $Locks$ is the set of lock-variables (which we introduced in the previous section).
  - $LocVars$ is the set of location variables $(pc_1, ..., pc_n)$ whose values identify the active locations of automata of $\mathcal{N}$.
  - $NextV$ is a set of variables as in $\mathcal{V}$ with the difference that the variables are prefixed by $next\_$. These variables are used to carry the values of corresponding variables in $\mathcal{V}$ during the sub-steps of OZ, such that, a variables $x \in \mathcal{V}$ is updated by the value of $next\_x \in NextV$ during the `CommitVar` sub-step.

  In the following, some abbreviations are defined:

  - $only\_change(S)$: refers to not allowing the change of variables $x \notin S$, where $S \subseteq \mathit{Var} \backslash Q$, where $Q = \{substep\} \cup Locks \cup LocVars$; formally:

    $$only\_change(S) \doteq \bigwedge_{x \in Var \backslash Q, x \notin S} x' = x,$$

  - $isZero(v_1, ..., v_n) \doteq \bigwedge_{1<i<n} v_i = 0$.
  - $lock(\{e_1, ..., e_n\}, \{a_1, ..., a_m\}) \doteq$

    $$e_1\_lock' = e_1\_lock + 1 \wedge ... \wedge e_n\_lock' = e_n\_lock + 1$$
    $$\wedge\ a_1\_lock' = a_1\_lock - 1 \wedge ... \wedge a_m\_lock' = a_m\_lock - 1$$

    defines lock-variable predicates of two given sets of events.

- $lock\_tcs(\{e_1, ..., e_n\}, \{a_1, ..., a_m\}) \doteq$
  $$lock(\{e_1, ..., e_n\}, \{a_1, ..., a_m\})$$
  $$\wedge \bigwedge_{g \in Locks \setminus \{e_1\_lock, ..., e_n\_lock, a_1\_lock, ..., a_m\_lock\}} g' = g$$
  defines lock-variable predicates of two given sets of events and prevents other lock-variables from changing their value.
- $at(p, X) \doteq pc_k = u$, where k is a number which identifies the automaton X of phase p, and u is a number which identifies p,
- $at\_P(p, X) \doteq pc'_k = u \bigwedge_{i \neq k, 1 \leq i \leq n} pc'_i = pc_i$, where k is a number which identifies the automaton X of phase p, u is a number which identifies p, and n is the number of automata of a network $\mathcal{N}$.
- $maintain\_locations \doteq \bigwedge_{1 \leq i \leq n} pc'_i = pc_i$, where n is the number of automata of a network $\mathcal{N}$.

- For all $p_1 \in \mathcal{P}_1, ..., p_n \in \mathcal{P}_n$,
  - ***Init***$(p_1, ..., p_n) = false$ if $p_1 \notin \mathcal{P}_{0,1}, ..., p_n \notin \mathcal{P}_{0,n}$, respectively:
  - ***Init***$(p_1, ..., p_n) \doteq substep = 1 \wedge \bigwedge_{c \in \mathcal{C}} c = 0 \wedge \int(\boldsymbol{p_1}) \wedge ... \wedge \int(\boldsymbol{p_n})$ $\wedge\, \mathcal{I}(\boldsymbol{p_1}) \wedge ... \wedge \mathcal{I}(\boldsymbol{p_n}) \wedge LocksConditions \wedge set(NextV, \mathcal{V}) \wedge$ $InitialConditions(p_1, ..., p_n) \wedge len > 0 \wedge substep' = 1$, otherwise,

  where:

  - $set(NextV, \mathcal{V})$ is a conjunction of predicates which requires that any variable $next\_x$ in $NextV$ have the same value as its corresponding variable x in $\mathcal{V}$.
  - $LockConditions$ is a conjunction of preconditions on lock-variables $z \in Locks$.
  - $InitialConditions(p_1, ..., p_n)$ is a conjunction of initial conditions of $p_1, ..., p_n$ including those of the location variables.

  Illustrations about the *Init* part are provided in section 5.4.3.

- ***Trans*** $= TickNegotiateTrans \cup CommitTrans \cup RefreshTrans \cup CSPTrans$ $\cup\ OZTrans \cup DCTrans \cup TFTrans \cup CommitVarTrans$ is the set of transitions whose subsets have a specific structure, such that, each element of ***Trans*** assigns a (transition) constraint to every pair of locations (sub-steps). Note that, for simplicity, we represent ***Trans*** here as a set of transitions. According to the definition of a transition constraint system, ***Trans*** is a disjunction of the elements of this set.

Before we explain how the subsets of ***Trans*** are constructed, we present an example which is used for illustration.

5.4.2 Example

Let the automata in Fig.5.5, Fig.5.7, Fig.5.8, and Fig.5.9 be the defined components of a network $\mathcal{N}$. Fig.5.6 shows the CSP-automata and the computed event-locks which correspond to shared events between CSP_X and CSP_Y; all other automata have no locked phases for any of the considered events and thus, they contain no event-locks.

Those correspond to two CSP-OZ-DC classes and a test formula TF:

- First class : CSP_X, OZ_X, and DC_X
- Second class : CSP_Y, OZ_Y, and DC_Y

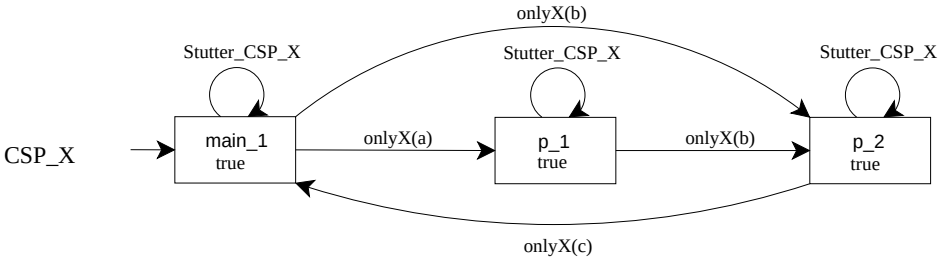
Table5.2 shows the different components of the automata of network $\mathcal{N}$. The two CSP-OZ-DC classes share the events a and b. For each of these events we introduce the lock-variables a_lock and b_lock. The two classes also have the variable x is in common. Phase $Error$ of TF is an example of a phase which represents an error phase whose reachability is tested by an abstraction refinement model checker.

Automata	*Phases*	*Initial Phases*	*Events*	*Variables*	*Clocks*
CSP_X	$main\_1, p\_1, p\_2$	$main\_1$	a, b, c	$-$	$-$
OZ_X	$mainOZ\_X$	$mainOZ\_X$	a, b, c	x	$-$
CSP_Y	$main\_2, q\_1, q\_2$	$main\_2$	a, b, d, e	$-$	$-$
OZ_Y	$mainOZ\_Y$	$mainOZ\_Y$	a, b, d, e	x, y	$-$
DC_X	$P1, P2$	$P1$	a	x	t_2
DC_Y	$P1, P2, P3$	$P1, P2$	d, e	x	t_1
TF	$P1, P2, Error$	$P1$	a	x	t_3

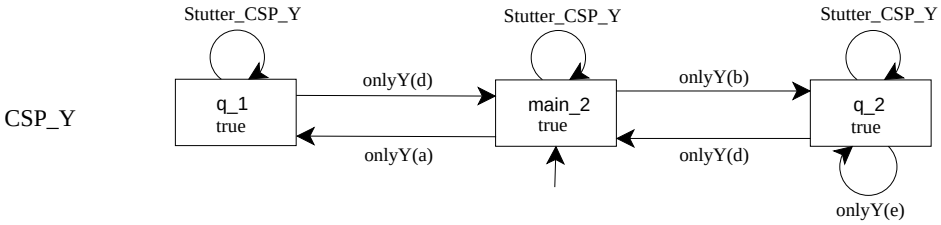
Table 5.2: The components of the automata of network $\mathcal{N}$

For $\mathcal{T}(\mathcal{N})$ we define:

- *Loc* $= \{l_1, ..., l_s\}$ is a set of locations, where, s is the number of sub-steps; the value of s is obtained later in the example.



$$Stutter\_CSP\_X = \neg a \wedge \neg b \wedge \neg c$$
$$onlyX(a) = a \wedge \neg b \wedge \neg c$$
$$onlyX(b) = b \wedge \neg a \wedge \neg c$$
$$onlyX(c) = c \wedge \neg b \wedge \neg a$$



$$Stutter\_CSP\_Y = \neg a \wedge \neg b \wedge \neg e \wedge \neg d$$
$$onlyY(a) = a \wedge \neg b \wedge \neg e \wedge \neg d$$
$$onlyY(b) = b \wedge \neg a \wedge \neg e \wedge \neg d$$
$$onlyY(e) = e \wedge \neg b \wedge \neg a \wedge \neg d$$
$$onlyY(d) = d \wedge \neg b \wedge \neg e \wedge \neg a$$

Figure 5.5: CSP automata of the network $\mathcal{N}$

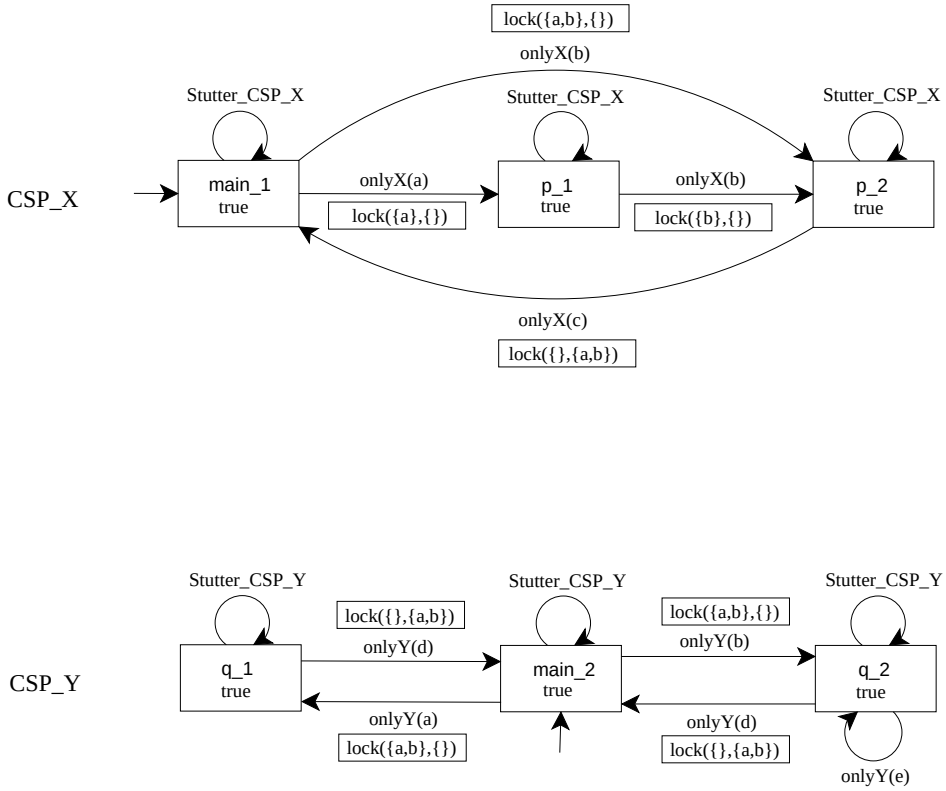
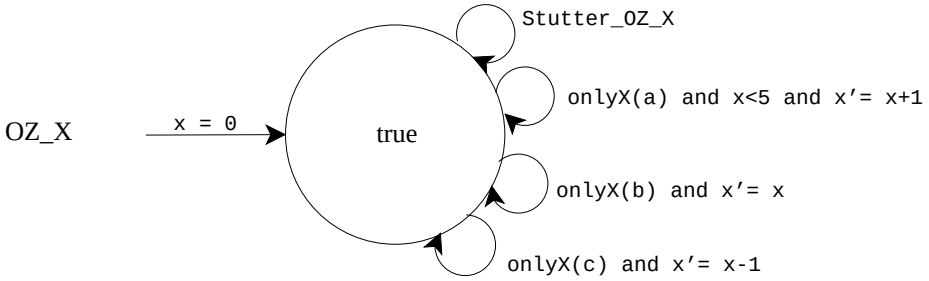
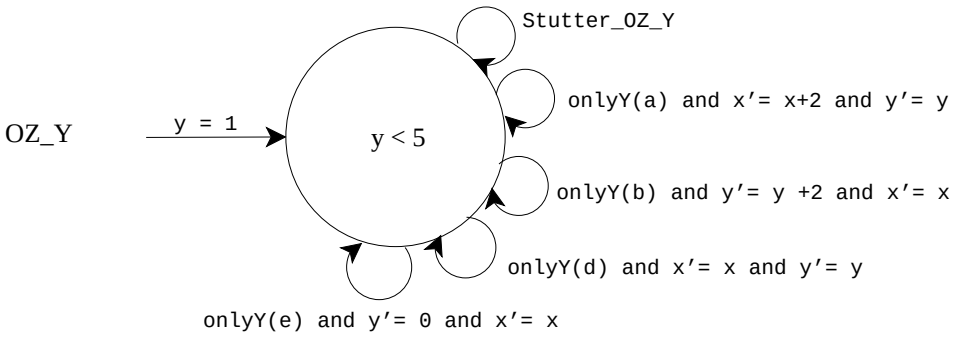


Figure 5.6: Lock-events for the CSP automata of the network $\mathcal{N}$



$$Stutter\_OZ\_X = \neg a \wedge \neg b \wedge \neg c \wedge x' = x$$
$$onlyX(a) = a \wedge \neg b \wedge \neg c$$
$$onlyX(b) = b \wedge \neg a \wedge \neg c$$
$$onlyX(c) = c \wedge \neg b \wedge \neg a$$



$$Stutter\_OZ\_Y = \neg a \wedge \neg b \wedge \neg e \wedge \neg d \wedge y' = y$$
$$onlyY(a) = a \wedge \neg b \wedge \neg e \wedge \neg d$$
$$onlyY(b) = b \wedge \neg a \wedge \neg e \wedge \neg d$$
$$onlyY(e) = e \wedge \neg b \wedge \neg a \wedge \neg d$$
$$onlyY(d) = d \wedge \neg b \wedge \neg e \wedge \neg a$$

Figure 5.7: OZ automata of the network $\mathcal{N}$

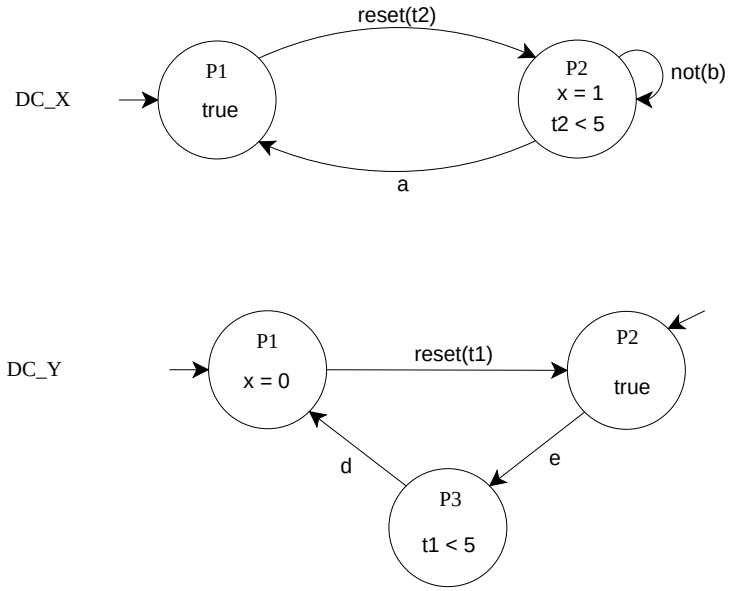


Figure 5.8: DC automata of the network $\mathcal{N}$

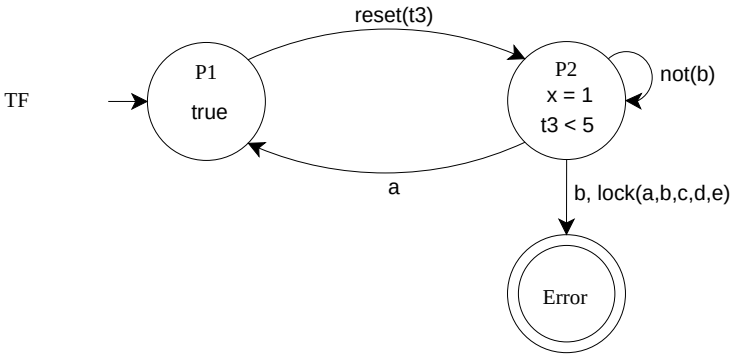


Figure 5.9: TF automaton

- The subsets of *Var* are:
  - $\{len, substep\}$,
  - $V = \{x, y\}$,
  - $A = \{a, b, c, d, e\}$,
  - $C = \{t_1, t_2, t_3\}$,
  - $NextV = \{next\_x, next\_y\}$,
  - $Locks = \{a\_lock, b\_lock\}$,
  - $LocVars = \{pc_1, pc_2, pc_3, pc_4, pc_5, pc_6, pc_7\ \}$.

In the following, the definition and illustration of the *Init* part and the different parts of *Trans* is presented:

5.4.3 The Init Part

Initially, the value of $substep$ is set to 1 which enables the first sub-step of `Tick-negotiate`. Different initial conditions *w.r.t.* the possible combinations of initial phases of the components exist. Also, the values of lock-variables of each event e is determined by the number of locked initial phases for e depending on the corresponding combination of initial phases.

In the running example, DC_Y has two initial phases while other components have only one initial phase, thus, the network has two initial conditions. Initially, non of the CSP-shared events (a and b) are blocked in any possible initial combination. Therefore, a_lock, b_lock must have the initial value 0. Note that, the requirement that the clocks have the value 0 in the *Init* sub-steps cannot violate clock invariants of initial phases since clock invariants are convex in all phases of a phase event automaton (see the definition of phase event automata in chapter 3). The following initial conditions correspond to *Init*:

- Initial condition 1:

$$\begin{gathered} substep = 1 \wedge isZero(a, b, c, d, e, t_1, t_2, t_3, a\_lock, b\_lock) \\ \wedge\ at(main\_1, CSP\_X) \wedge at(main\_2, CSP\_Y) \wedge at(P1, DC\_X) \\ \wedge\ at(P1, DC\_Y) \wedge at(P1, TF) \wedge x = 0 \wedge y = 1 \wedge\ next\_x = x \wedge next\_y = y \wedge len > 0 \end{gathered}$$

- Initial condition 2:

$$\begin{gathered} substep = 1 \wedge isZero(a, b, c, d, e, t_1, t_2, t_3, a\_lock, b\_lock) \\ \wedge\ at(main\_1, CSP\_X) \wedge at(main\_2, CSP\_Y) \wedge at(P1, DC\_X) \\ \wedge\ at(P2, DC\_Y) \wedge at(P1, TF) \wedge x = 0 \wedge y = 1 \wedge\ next\_x = x \wedge next\_y = y \wedge len > 0 \end{gathered}$$

5.4.4 Tick-negotiate Sub-steps

Transition constraint systems represent continuous transitions, which are implicit in the automaton model, as explicit discrete transitions. This needs to be represented in $\mathcal{T}(\mathcal{N})$. So, each DC-automaton (or each automaton with defined clocks) must perform a sub-step depending on its current location. The automata must agree (if possible) on values of their clocks which do not violate the current clock invariants. $TickNegotiateTrans$ is defined as follows ($substep$ has the value 1 in the first sub-step):

$TickNegotiateTrans \doteq \{$

$substep = 1 \wedge pred(1,1) \wedge substep' = 2, ..., substep = 1 \wedge pred(1,d_1) \wedge substep' = 2,$
$\vdots$
sub-steps of phases of DC-automata i, $2 \leq i \leq n_0 - 1$
$\vdots$
$substep = n_0 \wedge pred(n_0,1) \wedge substep' = n_0 + 1, ..., substep = n_0 \wedge pred(n_0,d_{n_0}) \wedge substep' = n_0 + 1$ $\}$.

where $pred(u_0, v_0)$ is a conjunction of predicates of corresponding to a phase v_0 of an automaton which corresponds to sub-step u_0; n_0 is the number of DC-automata and d_i the number of phases of the i-th DC-automaton. $pred(u_0, v_0)$ is defined as:

$$pred(u_0, v_0) \doteq len' \leq len \wedge len' > 0 \wedge only\_change(\{len\}) \\ \wedge maintain\_locations \wedge locks\_tcs(\{\}, \{\}) \wedge clockInv(v_1)$$

where:

- $clockInv(v_1)$ is a predicate which refers to the clock invariant which must be satisfied; if the clock invariant has the form $clock(v_1) < h$, then $clockInvPredicate(v_1) = clock(v_1) + len' < h$, where $clock(v_1)$ is the clock of the clock invariant and h is a given positive value of the invariant.

With $len' \leq len$, $len' > 0$, and $clockInv(v_1)$ each sub-step can find a value which does not violate the clock invariant of the corresponding phase which corresponds to its phase. This works since clock invariants are required to be convex.

Table5.3 shows the Tick-negotiation sub-step of the running example.

Tick_negotiate	
(1.1)	$substep = 1 \wedge at(P1, DC\_X) \wedge locks\_tcs(\{\}, \{\}) \wedge only\_change(\{len\})$ $\wedge maintain\_locations \wedge\ len' \leq len \wedge len' > 0 \wedge substep' = 2$
(1.2)	$substep = 1 \wedge at(P2, DC\_X) \wedge locks\_tcs(\{\}, \{\}) \wedge only\_change(\{len\})$ $\wedge t2 + len' < 5 \wedge maintain\_locations \wedge\ len' \leq len \wedge len' > 0 \wedge substep' = 2$
(2.1)	$substep = 2 \wedge at(P1, DC\_Y) \wedge locks\_tcs(\{\}, \{\}) \wedge only\_change(\{len\})$ $\wedge maintain\_locations \wedge\ len' \leq len \wedge len' > 0 \wedge substep' = 3$
(2.2)	$substep = 2 \wedge at(P2, DC\_Y) \wedge locks\_tcs(\{\}, \{\}) \wedge only\_change(\{len\})$ $\wedge maintain\_locations \wedge\ len' \leq len \wedge len' > 0 \wedge substep' = 3$
(2.3)	$substep = 2 \wedge at(P3, DC\_Y) \wedge locks\_tcs(\{\}, \{\}) \wedge only\_change(\{len\})$ $\wedge t1 + len' < 5 \wedge maintain\_locations \wedge\ len' \leq len \wedge len' > 0 \wedge substep' = 3$
(3.1)	$substep = 3 \wedge at(P1, TF) \wedge locks\_tcs(\{\}, \{\}) \wedge only\_change(\{len\})$ $\wedge maintain\_locations \wedge\ len' \leq len \wedge len' > 0 \wedge substep' = 4$
(3.2)	$substep = 3 \wedge at(P2, TF) \wedge only\_change(\{len\}) \wedge maintain\_locations$ $\wedge locks\_tcs(\{\}, \{\}) \wedge\ len' \leq len \wedge len' > 0 \wedge t3 + len' < 5 \wedge substep' = 4$
(3.3)	$substep = 3 \wedge at(P3, TF) \wedge locks\_tcs(\{\}, \{\}) \wedge only\_change(\{len\})$ $\wedge maintain\_locations \wedge\ len' \leq len \wedge len' > 0 \wedge substep' = 4$

Table 5.3: The `Tick-negotiate` part of the transition constraint system of $\mathcal{N}$

5.4.5 The COMMIT Sub-step

The `COMMIT` sub-step comes after the `Tick-negotiate` sub-step(s).

The value n_0 of *substep* enables the `COMMIT` sub-step, where n_0 is the value set to *substep* after taking the last sub-step of `Tick-negotiate`. In this single sub-step, the clock variables are updated by values which were obtained in the `Tick-negotiate` part.

$$CommitTrans \doteq \{substep = n_0 + 1 \wedge \textstyle\bigwedge_{t \in \mathcal{C}} t' = t + len \wedge only\_change(\mathcal{C}) \\ \wedge maintain\_locations \wedge locks\_tcs(\{\}, \{\}) \wedge substep' = n_0 + 2\},$$

Table5.4 shows the COMMIT sub-step of the running example.

COMMIT (4)	$substep = 4 \wedge t'_1 = t_1 + len \wedge t'_2 = t_2 + len \wedge t'_3 = t_3 + len$ $\wedge\ only\_change(\mathcal{C}) \wedge maintain\_locations$ $\wedge locks\_tcs(\{\}, \{\}) \wedge substep' = 5$

Table 5.4: The COMMIT part of the transition constraint system of $\mathcal{N}$

5.4.6 The Refresh Sub-step

The Refresh sub-step is a single sub-step which resets event variables to the value 0 – the reason for this will be clear in the next section. All other variables must remain unchanged in this sub-step. Table5.5 shows the *Refresh* sub-step of the running example.

Refresh (5)	$substep = 5 \wedge \bigwedge_{e \in \mathcal{A}} e' = 0 \wedge only\_change(\mathcal{A}) \wedge maintain\_locations$ $\wedge\ locks\_tcs(\{\}, \{\}) \wedge substep' = 6$

Table 5.5: The Refresh part of the transition constraint system of $\mathcal{N}$

Formally, $RefreshTrans \subset$ *Trans* is given by:

$$RefreshTrans \doteq \{ substep = n_0 + 2 \wedge \textstyle\bigwedge_{e \in \mathcal{A}} e' = 0$$
$$\wedge only\_change(\mathcal{A}) \wedge maintain\_locations \wedge locks\_tcs(\{\}, \{\}) \wedge substep' = n_0 + 3 \}.$$

5.4.7 CSP Sub-steps with Lock-variable Preconditions

After the REFRESH sub-step, the CSP-automata are the first automata of the network to perform a corresponding sub-step in the transition constraint system. As we know, whenever there is an event-communication by an event e between the components of a PEA-network, at least one CSP-automaton must take part with an e-communicating transition. Thus, CSP-automata sub-steps are chosen to (appropriately) enable event-variable preconditions in sub-steps which may follow the CSP's sub-step. This is one good reason why we distinguished automata by their types; since, for example, DC-automata may or may not take part in an event-communication. So, they can't be handled like CSP-automata; they get their event-communicating sub-steps enabled with the help of a previously taken CSP-sub-step. Also

the OZ-automata's sub-steps are handled the same as for DC-automata regarding event-variable preconditions.

In details, the CSP-automata of a given PEA-network are individually translated into a set of transitions. The automata are initially picked in a random order; consequently, translation rules are defined depending on this order – they differ on how events occurrences are represented, such that, the previously seen events enable corresponding event-variable preconditions in sub-steps which come next in order in the same complete step.

CSP-automata, as defined in [Hoe06][HM05], have two forms of transitions:

- A stuttering transition which prevents the communication of all the defined events of the corresponding automaton.
- A transition which only allows the communication of one defined event e to occur (we denoted them as $only(e)$-guarded-transitions).

Guards of stuttering transitions of CSP-automata have the form:

$$\bigwedge_{e \in \mathcal{A}_i} \neg e,$$

where $\mathcal{A}_i$ is the set of events of the $i-$th automaton. In the transition constraint system, the events of a CSP-stuttering transition are described as such:

$$\bigwedge_{e \in \mathcal{A}_i} e = 0$$

On the other hand, $only(e)$-guarded-transitions are represented as such:

1. If the current CSP-automaton $\mathcal{A}_i$ is the first one which has e in its alphabet (we denote it as its *main* CSP-automaton), then:

   $$only(e) = e \wedge \bigwedge_{d \in \mathcal{A}_i, d \neq e} \neg d,$$

   is translated into:

   $$\underline{e\_lock = 0 \wedge e' = 1} \wedge \bigwedge_{d \in \mathcal{A}_i, d \neq e} d = 0$$

2. If e was detected in a previously translated CSP-automaton, then $only(e)$ is translated into:

$$\underline{e = 1} \wedge \bigwedge_{d \in \mathcal{A}_i, d \neq e} d = 0$$

Rule (1) requires that the enabled event e is represented by the postcondition $e' = 1$ which enabled the $e = 1$ precondition in all translated $only(e)$ transitions of subsequent CSP-automata (following rule (2)).

Rule (1) also requires to have a precondition for e's lock-variable which make the transition `false` if $e\_lock \neq 0$, which means that event e is blocked in the current complete step; this is known from the value of lock-variables of the previous complete step. This explains why event-locks were computed in previous sections. Without the lock-variable precondition, the model-checker might continue constructing complete steps which are `false` without noticing this at the CSP-automata level.

Fig.5.10 shows what might happen if there were no lock-preconditions; on the other hand, Fig.5.11 shows what happens after adding lock-variables preconditions.

Lock-variables may have their values changed in a CSP sub-step. Also location variables change their value based on the goal phases of transitions. Other state variables may not change their value during a CSP sub-step.

Furthermore, each CSP-automaton is assigned a unique sub-step value which enables the corresponding set of transitions. Again, each set of transitions which correspond to a CSP-automaton has a precondition $(substep = k)$ and a postcondition $(substep' = k + 1)$, where k must have (at the beginning) the value which is reached right after the `REFRESH` sub-step. $k + 1$ enables the next component which might be another CSP-automaton or (when all CSP-automata are translated) an OZ-automataton (next section).

Suppose there are n CSP-automata in a given network, then $CSPTrans$ is defined as follows ($k = n_0 + 3$ is the sub-step value reached after the `Refresh` sub-step):

$CSPTrans \doteq \{$

$substep = k \wedge pred(1,1) \wedge substep' = k+1, ..., substep = k \wedge pred(1, m_k) \wedge substep' = k+1,$

$\vdots$

CSP sub-steps of automata i, $k + 2 \leq i \leq k + (n - 1)$

$\vdots$

$substep = k+n \wedge pred(n,1) \wedge substep' = k+n+1, ..., substep = k+n \wedge pred(n, m_{k+n}) \wedge substep' = k + n + 1$ $\}$.

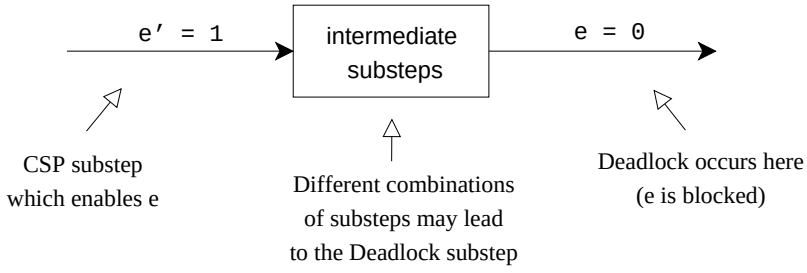


Figure 5.10: Without adding lock-variable preconditions

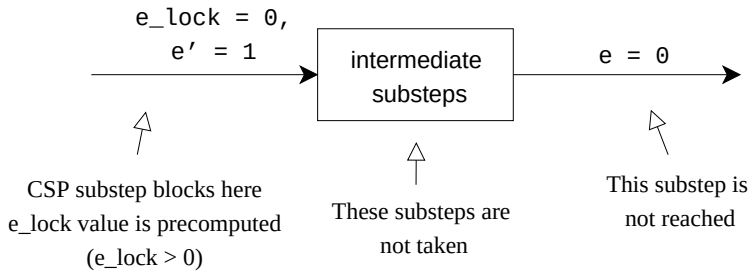


Figure 5.11: After adding lock-variable preconditions

where $pred(u, v)$ is a conjunction of predicates of a transition v of an automaton which correspond to sub-step u. It is defined as:

$$pred(u, v) \doteq predicate(E, v) \wedge only\_change(Q) \\ \wedge lock\_tcs(L, N) \wedge locationPredicates(u, v)$$

where:

- L and N are sets which corresponds to the precomputed lock-predicates of v.
- E is a set of enabled/disabled events e of transition v and $predicate(E, v)$ is a conjunction of $predicate(e)$'s for each $e \in E$, such that:

$$predicate(e) = \begin{cases} e' = 1 & \text{if an event } e \text{ communicates in its main CSP} \\ e = 1 & \text{if an event } e \text{ communicates not in its main CSP} \\ e = 0 & \text{if an event } e \text{ is disabled} \end{cases},$$

- Q is an empty set if the guard of v does not communicate an event. Otherwise, if v communicates an event e in its main CSP, then $Q = \{e\}$.

- $locationPredicates(u, v)$ = $at(p, X) \wedge at\_P(q, X)$ defines the pre- and post-conditions of the locations according to the transition of the corresponding automaton, such that, X is the automaton of sub-step u, p is the source phase of v, and q is v's target phase.

As an example, Table5.6 shows the translations of the CSP-automata in Fig.5.6.

5.4.8 OZ Sub-steps

Right after the sub-steps of CSP-automata, the next sub-steps are assigned to OZ-automata. OZ-automata can manipulate state variables. Also, variable constraints are defined for their automata-transitions. When constructing the OZ sub-steps, two important issues are noticed:

1. When an OZ-automaton changes a variable in a sub-step-transition, then a sub-step-transition of another automaton (which belongs to a sub-step that follows it) may be blocked, if a corresponding constraint is violated due to changing a variable by the previous OZ sub-step; this should'nt happen. So, we have to make sure that these variables don't change before the `CommitVar` sub-step (which is the final sub-step of a complete step of a given PEA-network).

2. When two or more OZ-automata change the same state variable, then a possible conflict may not be detected, since sub-steps are constructed individually. For example, suppose that an OZ-automaton OZ_1 has a transition with the guard $x > 0 \wedge x' = 1$ and that another automaton OZ_2 has a transition with the guard $x > 1 \wedge x' = 2$, then if $x > 1$ is true and the two transitions are taken, then x will have the value 2, or the value 1 depending on the order of sub-steps; this must not happen.

Case (1) may take place between an OZ-automaton, on one hand, and OZ- or DC-automata on the other hand.

This is solved by introducing a variable $next\_x$ for every variable $x \in \mathcal{V}$. Instead of changing a variable x in OZ sub-steps, we prevent changing x with the predicate $x' = x$ and allow changing the $next\_x$ variable which updates the value of x only during the `CommitVar` sub-step. As a result, constraints are not affected anymore since variables may only change their value at the end of each complete step.

Case (2) may occur between two or more OZ-automata. The problem is solved by adding the constraint $next\_x' = next\_x$ to all sub-steps of OZ-automata which are translated after those which belong to the OZ-automaton which has (potentially) modified x first. For the above stated example, the result we get is:

- $x > 0 \wedge x' = x \wedge next\_x' = 1$ is a part of the sub-step of the first picked OZ-automaton with a variable x.
- $x > 1 \wedge x' = x \wedge next\_x' = 2 \wedge next\_x' = next\_x$ is a part of the sub-step which belongs to a different OZ-automaton with a variable x.
- $x' = next\_x$ is a part of the `CommitVar` sub-step.

As an example for this problem consider the automata transitions which change the variable x in OZ_X and OZ_Y and their TCS-transitions `(8.2)` and `(9.2)` in Fig.5.7, respectively.

Notice that OZ-automata cannot have lock-variable predicates since they only have one phase each (see the sections about lock-variables). Also, since an OZ-automaton only has one phase, the sub-steps don't have to include location variables.

$OZTrans$ is similarly defined as $CSPTrans$ with the difference that value of its first *substep* starts at a value which is set by the last sub-step taken by CSP-automata. Also, $pred(u, v)$ has a different definition:

$$pred_{oz}(u, v) \doteq predicateOZ(E, v) \wedge VarPredicates \wedge only\_change(L) \wedge lock\_tcs(\{\}, \{\}) \wedge maintain\_locations$$

where E is a set of enabled/disabled events e in of transition v and $predicateOZ(E, v)$ is a conjunction of $predicateOZ(e)$'s for each $e \in E$, such that:

$$predicateOZ(e) = \begin{cases} e = 1 & \text{if an event } e \text{ communicates} \\ e = 0 & \text{if an event } e \text{ is disabled} \end{cases},$$

- $VarPredicates$ defines the pre- and postconditions of the variables $z \in \mathcal{V} \cup NextV$ according to the transition of the corresponding automaton. Those original predicates are modified appropriately as defined above.
- $L \subset NextV$ contains variables which correspond to variables $z \in \mathcal{V}$ which change their values when taking the tranistion v.

Furthermore, since OZ-automata have only one phase which can have state invariants, a sub-step follows the substeps of all the OZ-automata which checks the invariants which have to hold in every network step. Since the value of state variables is not updated until the final sub-step of a complete step, the value of the corresponding variables in are checked instead.

Table5.7 shows the translations of the OZ-automata in Fig.5.7.

5.4.9 DC Sub-steps

After the OZ sub-steps are taken, the DC-automata can perform their substeps (here we exclude the TF-automaton). Similar to CSP and OZ's construction, the sub-steps are constructed for DC-automata. Phases of DC-automata can have different clock/state invariants. So, $DCTrans$ is defined as for $OZTrans$; the difference again appears in $pred(u, v)$:

$$pred_{dc}(u, v) \doteq predicateDC(E, v) \wedge only\_change(X) \\ \wedge lock\_tcs(L, N) \wedge locationPredicates(u, v) \wedge locationInvariants(sink(v)) \\ \wedge VarPreconditions$$

where E is a set of enabled/disabled events e in of transition v and $predicateDC(E, v)$ is a conjunction of $predicateDC(e)$'s for each $e \in E$, such that:

$$predicateDC(e) = \begin{cases} e = 1 & \text{if an event } e \text{ communicates} \\ e = 0 & \text{if an event } e \text{ is disabled} \end{cases},$$

- $X \subset C$ includes clocks of clock resets of v and those of the clock invariants of $sink(v)$, where $sink(v)$ denotes the location which is reached after taking transition v,

- $locationPredictes(u, v)$ defines the pre- and postconditions of the locations according to the transition of the corresponding automaton.
- $locationInvariants(sink(v))$ are the primed predicates ($\int(sink(v))'$ and $\mathcal{I}(sink(v))'$) of the state/clock invariants which correspond to $\int(sink(v))$ and $\mathcal{I}(sink(v))$ of the current automaton. However, if variables of state invariants are non-local, then we require that $locationInvariants(sink(v))$ includes $next(\int(sink(v)))$ instead of $\int(sink(v))'$), where the function $next$ replaces each occurrence of a shared variable $x \in \mathcal{V}$ by $next\_x \in NextV$. This is because the value of variables $x \in \mathcal{V}$ can only be changed during the OZ sub-steps and thus the state invariants of the next phase of a DC-automaton hold if they hold for the next value of the variables which is assigned to their corresponding variables in $NextV$ (knowing that the variables of OZ sub-steps were not allowed to change until the next `CommitVar` sub-step).
- $VarPreconditions$ are preconditions on variables $z \in \mathcal{V}$ according to the transition of the corresponding automaton.

Table5.8 shows the translations of the DC-automata in Fig.5.8.

5.4.10 TF sub-step

After the DC sub-steps, which means also after that all sub-steps of other automata of the defined PEA-network have been taken, the TF automaton performs its sub-step. $TFTrans$ is defined as $DCTrans$ (the value of $substep$ is chosen appropriately).

There is a reason why TF sub-steps are taken only after all other automata have performed their sub-steps. Fig6.3 shows what happens if the TF sub-step was taken in the same complete step before other sub-steps which correspond to other automata in the given network. Thus, the TF sub-step must be the last sub-step to be taken. This insures that the complete step exists and is admissible in the network, if the error phase of TF is reached. Otherwise, the error phase might be reached even if the potential complete step is an existing deadlock of the system.

Table5.9 shows the translations of the DC-automata in Fig.5.9.

5.4.11 CommitVar Sub-step

At this level, variables x which correspond to variables $next\_x \in NextV$ are updated by the values of $next\_x$: $(x' = next\_x)$. Also, the variable len can change its value at this level. The next sub-steps are that of `Tick-negotiate`.

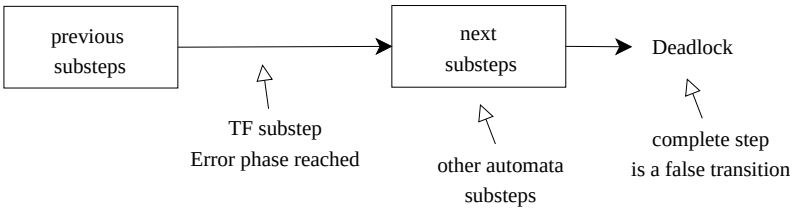


Figure 5.12: TF sub-step − Error phase reachability

$$CommitVarTrans \doteq \{substep = n_{dc} \wedge UpdateVars \wedge only\_change(\mathcal{V}) \\ \wedge maintain\_locations \wedge locks\_tcs(\{\}, \{\}) \wedge substep' = 1\},$$

where n_{dc} is the sub-step value reached after the DC sub-steps and $UpdateVars$ refers to updating the variables of $\mathcal{V}$ by the values of corresponding variables in $NextV$. Table5.9 shows the `CommitVar` sub-step of the running example.

CSP_X	
(6.1)	$substep = 6 \wedge at(main\_1, CSP\_X) \wedge a\_lock = 0 \wedge a' = 1 \wedge b = 0 \wedge c = 0$ $\wedge\ at_P(p\_1, CSP\_X) \wedge lock\_tcs(\{a\}, \{\}) \wedge only\_change(\{a\}) \wedge substep' = 7$
(6.2)	$substep = 6 \wedge at(main\_1, CSP\_X) \wedge b\_lock = 0 \wedge b' = 1 \wedge a = 0 \wedge c = 0$ $\wedge\ at_P(p\_2, CSP\_X) \wedge lock\_tcs(\{a, b\}, \{\}) \wedge only\_change(\{b\}) \wedge substep' = 7$
(6.3)	$substep = 6 \wedge at(main\_1, CSP\_X) \wedge a = 0 \wedge b = 0 \wedge c = 0$ $\wedge\ at\_P(main\_1, CSP\_X) \wedge only\_change(\{\}) \wedge lock\_tcs(\{\}, \{\}) \wedge substep' = 7$
(6.4)	$substep = 6 \wedge at(p\_1, CSP\_X) \wedge b\_lock = 0 \wedge b' = 1 \wedge a = 0 \wedge c = 0$ $\wedge\ at\_P(p\_2, CSP\_X) \wedge lock\_tcs(\{b\}, \{\}) \wedge only\_change(\{b\}) \wedge substep' = 7$
(6.5)	$substep = 6 \wedge at(p\_1, CSP\_X) \wedge a = 0 \wedge b = 0 \wedge c = 0$ $\wedge\ at\_P(p\_1, CSP\_X) \wedge lock\_tcs(\{\}, \{\}) \wedge only\_change(\{\}) \wedge substep' = 7$
(6.6)	$substep = 6 \wedge at(p\_2, CSP\_X) \wedge c\_lock = 0 \wedge c' = 1 \wedge a = 0 \wedge b = 0$ $\wedge\ at\_P(main\_1, CSP\_X) \wedge lock\_tcs(\{\}, \{a, b\}) \wedge only\_change(\{c\}) \wedge substep' = 7$
(6.7)	$substep = 6 \wedge at(p\_2, CSP\_X) \wedge a = 0 \wedge b = 0 \wedge c = 0$ $\wedge\ at\_P(p\_2, CSP\_X) \wedge lock\_tcs(\{\}, \{\}) \wedge only\_change(\{\}) \wedge substep' = 7$
CSP_Y	
(7.1)	$substep = 7 \wedge at(main\_2, CSP\_Y) \wedge a = 1 \wedge b = 0 \wedge d = 0 \wedge e = 0$ $\wedge at\_P(q\_1, CSP\_Y) \wedge lock\_tcs(\{a, b\}, \{\}) \wedge only\_change(\{\}) \wedge substep' = 8$
(7.2)	$substep = 7 \wedge at(main\_2, CSP\_Y) \wedge a = 0 \wedge b = 1 \wedge d = 0 \wedge e = 0$ $\wedge at\_P(q\_2, CSP\_Y) \wedge lock\_tcs(\{a, b\}, \{\}) \wedge only\_change(\{\}) \wedge substep' = 8$
(7.3)	$substep = 7 \wedge at(main\_2, CSP\_Y) \wedge a = 0 \wedge b = 0 \wedge d = 0 \wedge e = 0$ $\wedge\ at\_P(main\_2, CSP\_Y) \wedge lock\_tcs(\{\}, \{\}) \wedge only\_change(\{\}) \wedge substep' = 8$
(7.4)	$substep = 7 \wedge at(q\_1, CSP\_Y) \wedge a = 0 \wedge b = 0 \wedge e = 0 \wedge d' = 1$ $\wedge\ at\_P(main\_2, CSP\_Y) \wedge lock\_tcs(\{\}, \{a, b\}) \wedge only\_change(\{d\}) \wedge substep' = 8$
(7.5)	$substep = 7 \wedge at(q\_1, CSP\_Y) \wedge a = 0 \wedge b = 0 \wedge d = 0 \wedge e = 0$ $\wedge\ at\_P(q\_1, CSP\_Y) \wedge only\_change(\{\}) \wedge lock\_tcs(\{\}, \{\}) \wedge substep' = 8$
(7.6)	$substep = 7 \wedge at(q\_2, CSP\_Y) \wedge a = 0 \wedge b = 0 \wedge e = 0 \wedge d' = 1$ $\wedge\ at\_P(main\_2, CSP\_Y) \wedge lock\_tcs(\{\}, \{a, b\}) \wedge only\_change(\{d\}) \wedge substep' = 8$
(7.7)	$substep = 7 \wedge at(q\_2, CSP\_Y) \wedge a = 0 \wedge b = 0 \wedge d = 0 \wedge e = 0$ $\wedge\ at\_P(q\_2, CSP\_Y) \wedge lock\_tcs(\{\}, \{\}) \wedge only\_change(\{\}) \wedge substep' = 8$
(7.8)	$substep = 7 \wedge at(q\_2, CSP\_Y) \wedge a = 0 \wedge b = 0 \wedge d = 0$ $\wedge e' = 1 \wedge\ at\_P(q\_2, CSP\_Y) \wedge lock\_tcs(\{\}, \{\}) \wedge only\_change(\{e\}) \wedge substep' = 8$

Table 5.6: CSP-automata part of the transition constraint system of $\mathcal{N}$

OZ_X	
(8.1)	$substep = 8 \wedge a = 0 \wedge b = 0 \wedge c = 0$ $\wedge\ only\_change(\{\}) \wedge lock\_tcs(\{\},\{\}) \wedge maintain\_locations \wedge substep' = 9$
(8.2)	$substep = 8 \wedge a = 1 \wedge b = 0 \wedge c = 0$ $\wedge next\_x' = x + 1 \wedge x < 5 \wedge only\_change(\{next\_x\}) \wedge lock\_tcs(\{\},\{\})$ $\wedge maintain\_locations \wedge substep' = 9$
(8.3)	$substep = 8 \wedge a = 0 \wedge b = 1 \wedge c = 0$ $\wedge\ only\_change(\{\}) \wedge lock\_tcs(\{\},\{\}) \wedge maintain\_locations \wedge substep' = 9$
(8.4)	$substep = 8 \wedge a = 0 \wedge b = 0 \wedge c = 1$ $\wedge\ only\_change(\{next\_x\}) \wedge lock\_tcs(\{\},\{\})$ $\wedge maintain\_locations \wedge next\_x' = x - 1 \wedge substep' = 9$
OZ_Y	
(9.1)	$substep = 9 \wedge a = 0 \wedge b = 0 \wedge d = 0 \wedge e = 0$ $\wedge\ only\_change(\{\}) \wedge lock\_tcs(\{\},\{\})$ $\wedge maintain\_locations \wedge substep' = 10$
(9.2)	$substep = 9 \wedge a = 1 \wedge b = 0 \wedge d = 0 \wedge e = 0$ $\wedge\ only\_change(\{next\_x\}) \wedge lock\_tcs(\{\},\{\}) \wedge maintain\_locations$ $\wedge next\_x' = x + 2 \wedge next\_x' = next\_x \wedge substep' = 10$
(9.3)	$substep = 9 \wedge a = 0 \wedge b = 1 \wedge d = 0 \wedge e = 0$ $\wedge\ \ only\_change(\{next\_y\}) \wedge lock\_tcs(\{\},\{\})$ $\wedge maintain\_locations \wedge next\_y' = y + 2 \wedge substep' = 10$
(9.4)	$substep = 9 \wedge a = 0 \wedge b = 0 \wedge d = 1 \wedge e = 0$ $\wedge\ \ only\_change(\{\}) \wedge lock\_tcs(\{\},\{\})$ $\wedge maintain\_locations \wedge substep' = 10$
(9.5)	$substep = 9 \wedge a = 0 \wedge b = 0 \wedge d = 0 \wedge e = 1$ $\wedge\ \ only\_change(\{next\_y\}) \wedge lock\_tcs(\{\},\{\})$ $\wedge maintain\_locations \wedge next\_y' = 0 \wedge substep' = 10$
(10)	$substep = 10 \wedge next\_y < 5 \wedge substep' = 11$

Table 5.7: OZ part of the transition constraint system of $\mathcal{N}$

DC_X	
(11.1)	$substep = 11 \wedge at(P1, DC\_X) \wedge t_2' = 0$ $\wedge only\_change(\{t_2\}) \wedge lock\_tcs(\{\}, \{\}) \wedge at\_P(P2, DC\_X) \wedge substep' = 12$
(11.2)	$substep = 11 \wedge at(P2, DC\_X) \wedge a = 1 \wedge at\_p2\_DC2' = 0$ $\wedge\ only\_change(\{\}) \wedge lock\_tcs(\{\}, \{\}) \wedge at\_P(P1, DC\_X) \wedge substep' = 12$
(11.3)	$substep = 11 \wedge at(P2, DC\_X) \wedge b = 0 \wedge t_2' < 5$ $\wedge\ only\_change(\{t_2\}) \wedge lock\_tcs(\{\}, \{\}) \wedge at\_P(P2, DC\_X) \wedge substep' = 12$
DC_Y	
(12.1)	$substep = 12 \wedge at(P1, DC\_Y) \wedge t_1' = 0$ $\wedge\ only\_change(\{t_1\}) \wedge lock\_tcs(\{\}, \{\}) \wedge at\_P(P2, DC\_Y) \wedge substep' = 13$
(12.2)	$substep = 12 \wedge at(P2, DC\_Y) \wedge e = 1 \wedge t_1' < 5$ $\wedge\ only\_change(\{t_1\}) \wedge lock\_tcs(\{\}, \{\}) \wedge at\_P(P3, DC\_Y) \wedge substep' = 13$
(12.3)	$substep = 12 \wedge at(P3, DC\_Y) \wedge d = 1 \wedge next\_x = 0$ $\wedge\ only\_change(\{\}) \wedge lock\_tcs(\{\}, \{\}) \wedge at\_P(P1, DC\_Y) \wedge substep' = 13$

Table 5.8: DC part of the transition constraint system of $\mathcal{N}$

TF	
(13.1)	$substep = 13 \wedge at(P1, TF) \wedge t_3' = 0$ $\wedge\ only\_change(\{t_3\}) \wedge lock\_tcs(\{\}, \{\}) \wedge at\_P(P2, TF) \wedge substep' = 14$
(13.2)	$substep = 13 \wedge at(P2, TF) \wedge a = 1$ $\wedge\ only\_change(\{\}) \wedge lock\_tcs(\{\}, \{\}) \wedge at\_P(P1, TF) \wedge substep' = 14$
(13.3)	$substep = 13 \wedge at(P2, TF) \wedge b = 0 \wedge t_3' < 5$ $\wedge\ only\_change(\{t_3\}) \wedge lock\_tcs(\{\}, \{\}) \wedge at\_P(P2, TF)$
(13.4)	$substep = 13 \wedge at(P2, TF) \wedge b = 1$ $\wedge\ only\_change(\{\}) \wedge lock\_tcs(\{\}, \{\}) \wedge at\_P(Error, TF) \wedge substep' = 14$
CommitVAR	
(14)	$substep = 14 \wedge x' = next\_x \wedge y' = next\_y \wedge only\_change(x, y)$ $\wedge\ lock\_tcs(\{\}, \{\}) \wedge maintain\_locations \wedge len' > 0 \wedge substep' = 1$

Table 5.9: TF and the CommitVAR parts of the transition constraint system of $\mathcal{N}$

6
Example and Analysis

6.1 Example: Elevator **52**
6.2 Analysis **56**

In this chapter, we provide an example of our approach and some analysis. The network ($\mathcal{N}_{Elevator}$) models two CSP-OZ-DC classes and an automaton which is used to test an invariant.

6.1 Example: Elevator

The core of a controller of an elevator example was already presented in chapters 3 and 4. We modify it and provide another set of phase event automata; these correspond to a different CSP-OZ-DC class.

Fig.6.1 and Fig.6.2 show the automata of $\mathcal{N}_{Elevator}$. $\mathcal{N}_{Elevator}$ consists of: the CSP-automata $\mathcal{M}_{cspE}$ and $\mathcal{M}_{cspD}$, the OZ-automata $\mathcal{M}_{ozE}$ and $\mathcal{M}_{ozD}$, the DC-automata $\mathcal{M}_{dcE}$ and $\mathcal{M}_{dcD}$, and a Test-automaton $\mathcal{M}_{inv}$, where:

- $\mathcal{M}_{cspE} = (\{main, start\_Drive, Drive\}, \emptyset, \{start, stop, passed, newgoal\}, \emptyset, \mathcal{E}_{cspE}, \int_{cspE}, \mathcal{I}_{cspE}, \{main\})$, where $\mathcal{E}_{cspE}$ is the set of edges of $\mathcal{M}_{cspE}$ shown in Fig.6.1, and both $\int_{cspE}$ and $\mathcal{I}_{cspE}$ assign the predicate $true$ to all phases of $\mathcal{M}_{cspE}$,
- $\mathcal{M}_{cspD} = (\{P1, P2\}, \emptyset, \{start, open, close, wait\}, \emptyset, \mathcal{E}_{cspD}, \int_{cspD}, \mathcal{I}_{cspD}, \{P1\})$, where $\mathcal{E}_{cspD}$ is the set of edges of $\mathcal{M}_{cspD}$ shown in Fig.6.2, and both $\int_{cspD}$ and $\mathcal{I}_{cspD}$ assign the predicate $true$ to all phases of $\mathcal{M}_{cspD}$,
- $\mathcal{M}_{ozE} = (\{P0\}, \{goal, current, dir\}, \{start, stop, passed, newgoal\}, \emptyset, \mathcal{E}_{ozE}, \int_{ozE}, \mathcal{I}_{ozE}, \{P0\})$, where $\mathcal{E}_{ozE}$ is the set of edges shown in Fig.6.1, and both $\int_{ozE}$ and $\mathcal{I}_{ozE}$ assign the predicate $true$ to $P0$ and the initial condition $goal = min \wedge current = min \wedge dir = 0$ (min and max are constants),

- $\mathcal{M}_{ozD} = (\{Q0\}, \{start, open, close, wait\}, \{closed\}, \emptyset, \mathcal{E}_{ozD}, \int_{ozD}, \mathcal{I}_{ozD}, \{Q0\})$, where $\mathcal{E}_{ozD}$ is the set of edges shown in Fig.6.2, and both $\int_{ozD}$ and $\mathcal{I}_{ozD}$ assign the predicate $true$ to $Q0$ and the initial condition $closed = 1$,

- $\mathcal{M}_{dcE} = (\{P0, P1, P2\}, \{stop\}, \{current, goal\}, \{c_1\}, \mathcal{E}_{dcE}, \int_{dcE}, \mathcal{I}_{dcE}, \{P0, P1\})$, where
  $\mathcal{E}_{dcE}$ is the set of edges shown in Fig.6.1, where $\int_{dcE}$ and $\mathcal{I}_{dcE}$ assign predicates shown in Fig.6.1,

- $\mathcal{M}_{dcD} = (\{D0, D1\}, \{wait, close\}, \{closed\}, \{c_2\}, \mathcal{E}_{dcD}, \int_{dcD}, \mathcal{I}_{dcD}, \{D0\})$, where $\mathcal{E}_{dcD}$ is the set of edges shown in Fig.6.2, where $\int_{dcD}$ and $\mathcal{I}_{dcD}$ assign predicates shown in Fig.6.2,

- $\mathcal{M}_{inv} = (\{No\_Error, Error\}, \{\}, \{current\}, \{\}, \mathcal{E}_{inv}, \int_{inv}, \mathcal{I}_{inv}, \{No\_Error\})$, where $\mathcal{E}_{inv}$ is the set of edges shown in Fig.6.2, where $\int_{inv}$ and $\mathcal{I}_{inv}$ assign predicates shown in Fig.6.3.

The computed event-lock predicates are also shown in the figures Fig.6.2 and Fig.6.1. Note that, we use the same abbreviations as in the previous chapter.

$\mathcal{T}(\mathcal{N}_{Elevator})$ is defined as follows:

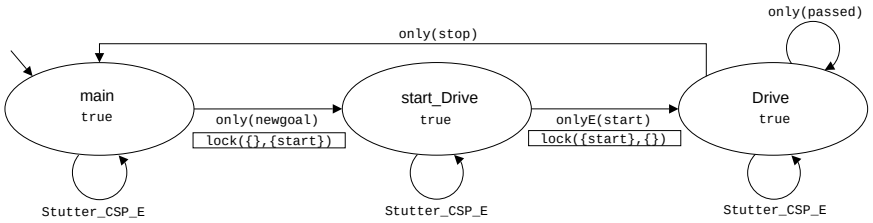
- $Loc = \{l_1, ..., l_{12}\}$, such that, for each sub-step there is a location.

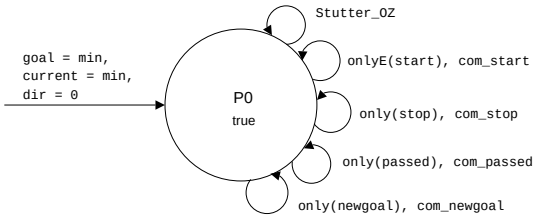
- $Var = \{current, goal, dir, closed, start, stop, newgoal, passed, open, close, wait, c_1, c_2, start\_lock, pc_1, ..., pc_7, next\_current, next\_goal, next\_dir, next\_closed, len, substep\}$

- $Init \doteq at(main, \mathcal{M}_{cspE}) \wedge at(P1, \mathcal{M}_{cspD}) \wedge at(P0, \mathcal{M}_{ozE}) \wedge at(Q0, \mathcal{M}_{ozD}) \wedge at(P0, \mathcal{M}_{dcE}) \wedge at(D0, \mathcal{M}_{cspD}) \wedge at(No\_Error, \mathcal{M}_{inv}) \wedge goal = min \wedge current = min \wedge dir = 0 \wedge closed = 1 \wedge start\_lock = 1 \wedge len > 0 \wedge substep = 1 \wedge isZero(c_1, c_2, start, stop, newgoal, passed, open, close, wait) \wedge next\_current = current \wedge next\_goal = goal \wedge next\_dir = dir \wedge next\_closed = closed$

- Transitions of each subset of *Trans* are shown in Table 6.1 (not all transitions of each subset are depicted).

- $reset\_events = start' = 0 \wedge stop' = 0 \wedge newgoal' = 0 \wedge passed' = 0 \wedge open' = 0 \wedge close' = 0 \wedge wait' = 0$





$$\text{onlyE(start), com\_start} = \neg\, newgoal \wedge start \wedge \neg\, stop \wedge \neg\, passed \wedge goal > current \wedge dir' = 1 \wedge current' = current \wedge goal' = goal$$

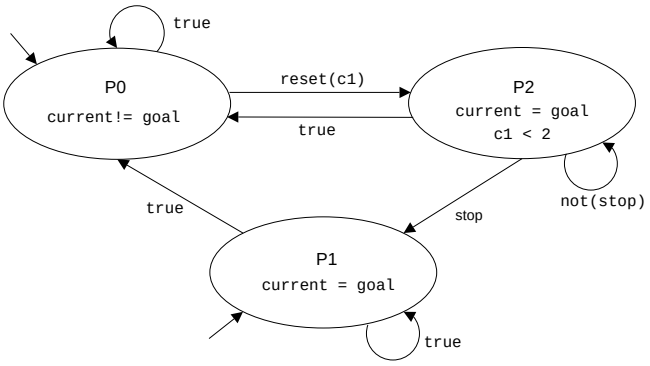


Figure 6.1: Phase event automata of $\mathcal{N}_{Elevator}$

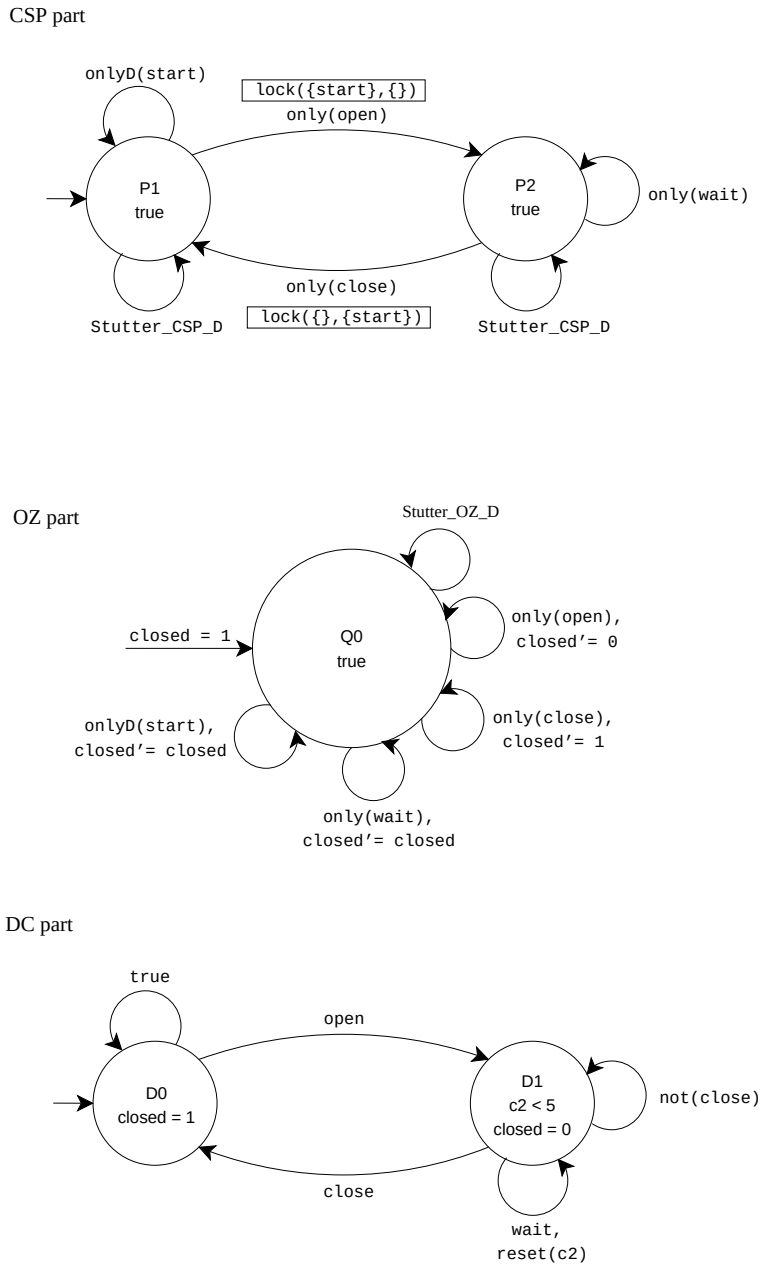


Figure 6.2: Phase event automata of $\mathcal{N}_{Elevator}$ – (2) not(x) = ¬ x

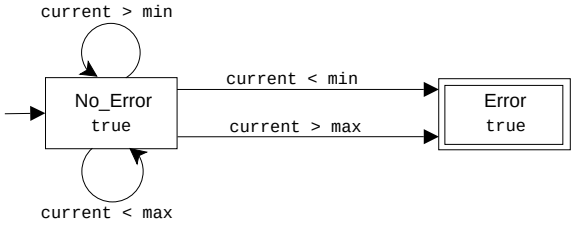


Figure 6.3: Test automaton of $\mathcal{N}_{Elevator}$

6.2 Analysis

The number of transitions of the *Trans* part of the our translation method of networks of phase event automata into transition constraint systems is:

$|nr\_of\_\textit{Tick-negotiate}\_substeps| + |nr\_of\_\textit{Commit}\_substeps|$
$+ |nr\_of\_\textit{Refresh}\_substeps| + |nr\_of\_\textit{CSP}\_substeps| + |nr\_of\_\textit{OZ}\_substeps|$
$+ |nr\_of\_\textit{DC}\_substeps| + |nr\_of\_\textit{TF}\_substeps| + |nr\_of\_\textit{VarCommit}\_substeps|$

$= (n_{dc,1}+...+n_{dc,r}) + 1 + 1 + (m_{csp,1}+...+m_{csp,u}) + (m_{oz,1}+...+m_{oz,v}) +$
$(m_{dc,1}+...+m_{dc,w}) + 1 + 1$, where:

- $m_{csp,i}$, $1 \leq i \leq u$ is the number of transitions of CSP-automaton i and u is the number of CSP-automata,
- $m_{oz,i}$, $1 \leq i \leq v$ is the number of transitions of OZ-automaton i and v is the number of OZ-automata,
- $m_{dc,i}$, $1 \leq i \leq w$ is the number of transitions of DC-automaton i and w is the number of DC-automata,
- $n_{dc,i}$, $1 \leq i \leq r$ is the number of phases of DC-automaton i and r is the number of DC-automata.

With previous approaches, the number of transitions of (product) transition constraint systems is in $\mathcal{O}(m_{csp,1} \cdot ... \cdot m_{csp,u} \cdot m_{oz,1} \cdot ... \cdot m_{oz,v} \cdot m_{dc,1} \cdot ... \cdot m_{dc,u})$, knowing that there were some slight optimizations to obtain simplified products of automata and also for products of DC-automata [Hoe06].

The new presented approach in this thesis had more success in representing fewer transitions since it avoids the construction of the product but leaves some work for the model checker; more details are discussed in chapter conclusion.

Tick_negotiate	
(1.1)	$substep = 1 \wedge at(P0, \mathcal{M}_{dcE}) \wedge locks\_tcs(\{\}, \{\}) \wedge only\_change(\{len\})$ $\wedge maintain\_locations \wedge\ len' \leq len \wedge len' > 0 \wedge substep' = 2$
(1.2)	$substep = 1 \wedge at(P1, \mathcal{M}_{dcE}) \wedge locks\_tcs(\{\}, \{\}) \wedge only\_change(\{len\})$ $\wedge maintain\_locations \wedge\ len' \leq len \wedge len' > 0 \wedge substep' = 2$
(1.3)	$substep = 1 \wedge at(P2, \mathcal{M}_{dcE}) \wedge locks\_tcs(\{\}, \{\}) \wedge only\_change(\{len\})$ $\wedge c_1 + len' < 2 \wedge maintain\_locations \wedge\ len' \leq len \wedge len' > 0 \wedge substep' = 2$
(2.1)	$substep = 2 \wedge at(D0, \mathcal{M}_{dcD}) \wedge locks\_tcs(\{\}, \{\}) \wedge only\_change(\{len\})$ $\wedge maintain\_locations \wedge\ len' \leq len \wedge len' > 0 \wedge substep' = 3$
(2.2)	$substep = 2 \wedge at(D1, \mathcal{M}_{dcD}) \wedge locks\_tcs(\{\}, \{\}) \wedge only\_change(\{len\})$ $\wedge c_2 + len' < 5 \wedge maintain\_locations \wedge\ len' \leq len \wedge len' > 0 \wedge substep' = 3$
COMMIT	
(3)	$substep = 3 \wedge c_1' = c_1 + len \wedge c_2' = c_2 + len$ $\wedge\ only\_change(\{c_1, c_2\}) \wedge maintain\_locations$ $\wedge locks\_tcs(\{\}, \{\}) \wedge substep' = 4$
Refresh	
(4)	$substep = 4 \wedge reset\_events \wedge maintain\_locations$ $\wedge only\_change(\{start, stop, newgoal, passed, open, close, wait\})$ $\wedge\ locks\_tcs(\{\}, \{\}) \wedge substep' = 5$
CSP sub-steps	
(5.1)	$substep = 5 \wedge at(main, \mathcal{M}_{cspE}) \wedge newgoal' = 1 \wedge isZero(start, stop, passed)$ $\wedge\ at\_P(start\_Drive, \mathcal{M}_{cspE}) \wedge lock\_tcs(\{\}, \{start\})$ $\wedge only\_change(\{newgoal\}) \wedge substep' = 6$
(5.2)	$substep = 5 \wedge at(start\_Drive, \mathcal{M}_{cspE}) start\_lock = 0 \wedge start' = 1$ $\wedge\ \wedge isZero(newgoal, stop, passed) \wedge at\_P(Drive, \mathcal{M}_{cspE})$ $\wedge lock\_tcs(\{start\}, \{\}) \wedge only\_change(\{start\}) \wedge substep' = 6$
⋮	⋮
(6.1)	$substep = 6 \wedge at(P1, \mathcal{M}_{cspD}) \wedge start = 1 \wedge isZero(newgoal, stop, passed)$ $\wedge\ at\_P(P1, \mathcal{M}_{cspD}) \wedge lock\_tcs(\{\}, \{\}) \wedge only\_change(\{\}) \wedge substep' = 7$
⋮	⋮

Table 6.1: The *Trans* part of $\mathcal{T}(\mathcal{N}_{Elevator})$

OZ sub-steps	
(7.1)	$substep = 7 \wedge isZero(newgoal, stop, passed, start)$ $\wedge\ only\_change(\{\}) \wedge lock\_tcs(\{\},\{\})$ $\wedge maintain\_locations \wedge substep' = 8$
(7.2)	$substep = 7 \wedge isZero(newgoal, stop, passed) \wedge start = 1$ $\wedge goal > current \wedge next\_dir' = 0 \wedge only\_change(\{next\_dir\}) \wedge lock\_tcs(\{\},\{\})$ $\wedge maintain\_locations \wedge substep' = 8$
⋮	⋮
(8.1)	$substep = 8 \wedge isZero(open, close, start)$ $\wedge wait' = 1 \wedge only\_change(\{\}) \wedge lock\_tcs(\{\},\{\})$ $\wedge maintain\_locations \wedge substep' = 9$
⋮	⋮
DC sub-steps	
(9.1)	$substep = 9 \wedge at(P0, \mathcal{M}_{dcE})) \wedge c_1' = 0 \wedge c_1' < 2$ $\wedge only\_change(\{c_1\}) \wedge lock\_tcs(\{\},\{\}) \wedge at\_P(P2, \mathcal{M}_{dcE})) \wedge substep' = 10$
(9.2)	$substep = 9 \wedge at(P2, \mathcal{M}_{dcE})) \wedge stop = 0 \wedge next\_current = next\_goal$ $\wedge only\_change(\{c_1\}) \wedge lock\_tcs(\{\},\{\})$ $\wedge\ at\_P(P1, \mathcal{M}_{dcE})) \wedge substep' = 10$
⋮	⋮
(10.1)	$substep = 10 \wedge at(D0, \mathcal{M}_{dcD})) \wedge open = 1 \wedge next\_closed = 0 \wedge c_2' < 5$ $\wedge only\_change(\{c_2\}) \wedge lock\_tcs(\{\},\{\}) \wedge at\_P(D1, \mathcal{M}_{dcD})) \wedge substep' = 11$
⋮	⋮
TF sub-step	
(11.1)	$substep = 11 \wedge at(No\_Error, \mathcal{M}_{inv}) \wedge current < min$ $\wedge\ only\_change(\{\}) \wedge lock\_tcs(\{\},\{\}) \wedge at\_P(Error, \mathcal{M}_{inv}) \wedge substep' = 12$
(11.2)	$substep = 11 \wedge at(No\_Error, \mathcal{M}_{inv}) \wedge current > min$ $\wedge\ only\_change(\{\}) \wedge lock\_tcs(\{\},\{\}) \wedge at\_P(No\_Error, \mathcal{M}_{inv}) \wedge substep' = 12$
⋮	⋮
CommitVar	
(12)	$substep = 12 \wedge current' = next\_current \wedge goal' = next\_goal \wedge dir' = next\_dir$ $\wedge closed' = next\_closed \wedge only\_change(\{current, goal, dir, closed, len\})$ $\wedge\ lock\_tcs(\{\},\{\}) \wedge maintain\_locations \wedge len' > 0 \wedge substep' = 1$

Table 6.2: The *Trans* part of $\mathcal{T}(\mathcal{N}_{Elevator})$ (2)

7
Conclusion

7.1 Summary and Final Thoughts 60
7.2 Future Work 61

7.1 Summary and Final Thoughts

CSP-OZ-DC is a highly expressive specification language which combines three specification languages: CSP, Object-Z (OZ), and Duration Calculus (DC). An approach to verify CSP-OZ-DC specifications of infinite systems is to translate classes of these specifications into phase event automata; those provide a compositional semantics for CSP-OZ-DC. Phase event automata are then translated into transition constraint systems. In this thesis, a new approach for translating networks of phase event automata is presented. Previous works required that the product of the automata is generated, which is then translated into a transition constraint system.

The new approach avoids constructing the product and builds a more compact and efficient transition constraint system whose transitions describes individual transition-steps (which we called substeps) instead of describing complete network transitions which are products of the individual automata transitions.

This new approach has advantages but also a disadvantage. Describing transition constraint systems this way, results in having new deadlock runs which may have not existed in the corresponding original specifications. Those appear for example in a simulation which tries to construct a complete step but fails due to a decision taken for a previous substep since the choice of next transitions is done at the component level and not at the network level; although we note here that, we solved this problem partially by using what we called event-locks.

Avoiding the construction of the product automaton resulted in notably fewer transitions. Instead of products of transitions, a collection of transitions under rules represents a network step. We conclude that the state space explosion problem, which is one of the most serious problems with model checking in practice, is solved at the level of building transition constraint systems. Thus, the approach of this thesis can prove its effectiveness, especially for large systems.

7.2 Future Work

The approach which is presented in this thesis can be applied for phase event automata of CSP-OZ-DC specifications. Possible different specification languages could use similar approaches to avoid the usual construction of products which is a common problem when it comes to the ability of exploring the entire state space with limited resources of time and memory.

References

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science, 126:183-235*, 1994. 12

[BR01] T. Ball and S. K. Rajamani. The SLAM toolkit. *In CAV'01, pages 260-264*, 2001. 2

[DP99] G. Delzanno and A. PodelskiModel. Model checking in CLP. *In TACAS'99, pages 223-239*, 1999. 2

[Fis00] C. Fischer. Combination and Implementation of Processes and Data: From CSP-OZ to Java. PhD thesis. *University of Oldenburg*, 2000. 2

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. *In CAV'97, pages 72-83*, 1997. 2

[HM05] Jochen Hoenicke and Patrick Maier. Model-Checking of Specifications Integrating Processes, Data and Time. *Sonderforschungsbereich/Transregio 14 AVACS (Automatic Verification and Analysis of Complex Systems).*, 2005. 2, 3, 5, 6, 18, 20, 23, 41

[HO02] J. Hoenicke and E.-R. Olderog. Combining specification teqniques for processes data and time. *In IFM'02, volume 2335 of LNCS - Spriger*, 2002. 14

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM, 21:666-677*, 1978. 6, 7

[Hoa85] C.A.R. Hoare. Communicating Sequential Processes. *Prentice Hall*, 1985. 6, 7

[Hoe06] Jochen Hoenicke. Combination of Processes, Data, and Time. *PhD thesis, University of Oldenburg*, 2006. 2, 3, 5, 6, 10, 15, 21, 23, 41, 56

[IBW07] Bernd Finkbeiner Ingo Brückner, Klaus Dräger and Heike Wehrheim. Slicing Abstractions. *FSEN*, 2007. 3

[ISO00] ISO. Z formal specification notation - Syntax, type system and semantics. *Information technology. Number 13568*, 2000. 8

[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. 2

[Plo83] G.D. Plotkin. An Operational Semantics for CSP. *Formal Description of Programming Concepts II, North-Holland, Amsterdam, pages 199-223*, 1983. 7

[Rav94] A.P. Ravn. Design of embedded real-time computing systems. *Dr. tech. dissertation, Technical University of Denmark*, 1994. 9

[Ryb02] A. Rybalchenko. A model checker based on abstraction refinement. *Master's thesis, University of Saarland, Germany*, 2002. 3

[SCV03] A. Groce S. Jha S. Chaki, E. Clarke and H. Veith. Modular verification of software components in C. *In ICSE'03, pages 385-395*, 2003. 2

[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*, 1999. 2

[Smi00] G. Smith. The Object-Z Specification Language. *Kluwer Academic Publisher*, 2000. 8

[SO99] M. Schenke and E.-R. Olderog. Transformational design of realtime systems. *Part 1: from requirements to program specifications*, 1999. 9

[Spi88] J.M. Spivey. Understanding Z: a specification language and its formal semantics. *Cambridge tracts in theoretical computer science*, 1988. 8

[Tap01] J. Tapken. Model-Checking of Duration Calculus Specifications. *PhD thesis, University of Oldenburg*, 2001. 9, 11

[THS02] R. Majumdar T.A. Henzinger, R. Jhala and G. Sutre. Lazy abstraction. *In POPL'02, pages58-70. ACM Press*, 2002. 2

[ZH04] C. Zhou and M.R. Hansen. Duration Calculus: A Formal Approach to Real-Time Systems. *Springer-Verlag*, 2004. 9