Deductive Model Checking with Transition Constraint Systems

Diploma Thesis

submitted by
Dominik Brill

Supervisor
Prof. Dr. Bernd Finkbeiner

Advisor
Dipl. Inf. Klaus Dräger

Reviewers
Prof. Dr. Bernd Finkbeiner
Prof. Dr. Reinhard Wilhelm

Saarland University
Faculty of Natural Sciences and Technology
Department of Computer Science
Reactive Systems Group

August 5, 2007

Abstract

This thesis presents an extension of deductive model checking that uses phase event automata to verify, if a given reactive system satisfies a specification, typically a temporal logic formula. Both the reactive system and the specification are encoded in an initial phase event automaton that is repeatedly refined and transformed until a counterexample computation or a correctness proof is found. Additional heuristics concerning the refinement and transformation rules are used to further improve the model checking algorithm. These heuristics lead to large state-space savings. To preserve full automatism, the specification language is restricted to state safety properties. The use of phase event automata allows the integration of CSP, Object-Z and Duration Calculus into the framework.

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

Saarbrücken, August 5, 2007

Dominik Brill

Contents

1 Introduction **11**

2 Preliminaries **13**
2.1 Reactive systems . . . 13
2.2 Real-time systems . . . 13
2.3 Model Checking . . . 14
2.4 Fair Transition Systems . . . 14
2.5 Example - Fischers' Mutual Exclusion Algorithm . . . 15
2.6 Linear-time Temporal Logic . . . 17
2.7 The Formula Tableau . . . 18
2.8 CSP-OZ-DC . . . 20
2.8.1 Communicating Sequential Processes . . . 20
2.8.2 Object-Z . . . 20
2.8.3 Duration Calculus . . . 21

3 Phase Event Automata **23**
3.1 Timed Automata . . . 23
3.2 Phase event automata . . . 24
3.3 Formal Definition . . . 25
3.4 Parallel Composition of Phase Event Automata . . . 26

4 The DMC Procedure - Safety **27**
4.1 Deductive Model Checking . . . 27
4.2 The Restricted DMC Procedure . . . 28

5 DMC with Phase Event Automata **32**
5.1 Configuration Files . . . 34
5.1.1 Boolean Constraints - Init and Invariants . . . 35
5.1.2 Transitions . . . 35
5.1.3 Nodes and Nodelabels . . . 36
5.1.4 Variables . . . 36
5.1.5 Edges . . . 36
5.1.6 Example: Elevator Configuration Script . . . 37
5.2 Phase Event Automata Creation . . . 38
5.2.1 The PEA Object . . . 38
5.2.2 The Node Object . . . 38
5.2.3 The Edge Object . . . 39
5.2.4 The Transition Object . . . 40
5.3 Initial Falsification Diagram . . . 41
5.3.1 Initial Falsification Diagram Construction . . . 41
5.3.2 Construct New Initial Nodes . . . 41
5.3.3 Change Transition Labels . . . 42

5.3.4 *Basic Transformations* . . . 43
5.3.5 Change Failure Nodes . . . 43
5.3.6 Change Remaining Nodes . . . 43
5.3.7 Split With Invariants . . . 43
5.3.8 Set Failure Distances . . . 44
5.3.9 Examples . . . 44
5.4 Basic Transformations . . . 47
5.4.1 Unreachable Node . . . 47
5.4.2 Unsatisfiable Node . . . 48
5.4.3 Remove Edge Label . . . 48
5.4.4 Empty Edge . . . 50
5.5 Node Splitting . . . 51
5.5.1 Create New Nodes . . . 51
5.5.2 Create Edges . . . 52
5.6 Basic Refinement Transformations . . . 53
5.6.1 Construction of the *enabled* formula . . . 54
5.6.2 Construction of the strongest postcondition formula . . . 55
5.6.3 Construction of the weakest precondition formula . . . 56
5.6.4 Structural Decisions . . . 59
5.6.5 Precondition Split . . . 64
5.6.5.1 Definition: Precondition Split . . . 64
5.6.5.2 Algorithm for Precondition Split . . . 64
5.6.5.3 Example: Fischer's mutual exclusion problem . . . 66
5.6.6 Postcondition Split . . . 69
5.6.6.1 Algorithm for Postcondition Split . . . 69
5.6.6.2 Example: Fischers Mutual Exclusion . . . 71

6 Analysis **73**
6.1 Example: Deque . . . 73
6.2 Example: Board4 . . . 75
6.3 Example: Elevator . . . 76
6.4 Example: Bakery . . . 79
6.5 Example: Fischer's Problem . . . 82

7 Conclusion **84**
7.1 Future Work . . . 84

List of Figures

2.1 **Timed automaton for process i of Fischer's algorithm** 16
2.2 **Formula tableau for φ** 19
2.3 **Simplified formula tableau for φ** 20
2.4 **Data aspects of an elevator specification in Object - Z** 21

3.1 **A simple timed automaton** 23
3.2 **A phase event automaton** 24

4.1 **Initial falsification diagram for Fischer's mutual exclusion** . 28
4.2 **Precondition split on edge $\langle N_1, N_2 \rangle$** 31

5.1 **Initial Falsification diagram without invariant splitting** . . . 45
5.2 **Initial Falsification diagram with invariant splitting** 46
5.3 **Initial Falsification diagram for the elevator example** 46
5.4 **Unreachable graph for the elevator example** 48
5.5 **Sample error trace** 59
5.6 **Target Enlargement in Precondition Split** 60
5.7 **General target enlargement** 61
5.8 **Source enlargement** 62
5.9 **Removing of redundant nodes** 63
5.10 **After the first precondition split and basic transformations** 68
5.11 **After the first postcondition split** 72

6.1 **Resulting graph of the Deque problem** 74
6.2 **Initial state of the board4 problem** 75
6.3 **Board configuration after trace $\tau_1, \tau_4, \tau_7, \tau_8, \tau_9, \tau_5, \tau_6, \tau_2, \tau_3$** . . . 76
6.4 **Resulting graph of the board4 problem** 77
6.5 **Resulting graph of the Elevator example** 78
6.6 **Program Bakery for mutual exclusion** 79
6.7 **Resulting graph of the Bakery problem** 81
6.8 **Resulting graph of the Fisher problem** 83

List of Tables

2.1 **Semantics of modal operators** 18

List of Algorithms

1 Initial falsification diagram construction 41
2 Construction of new initial nodes 42
3 Changing the labels of failure nodes 43
4 All nodes are split with the invariants 44
5 Remove edge label transformation rule 49
6 $create\_split\_nodes(l, initflag, failflag, newnodes)$ 52
7 Complete algorithm for deductive model checking 53
8 Calculation of $\neg wpc(\tau, \phi)$ 58
9 Complete algorithm for precondition split 64
10 Algorithm that realizes the procedure *pre_check* 65
11 Complete algorithm for postcondition split 69
12 Algorithm that realizes the procedure *post_check* 70

Chapter 1

Introduction

Reliability is the most important property of many of todays software and hardware systems. But by the inevitable increase of software and hardware complexity, the likelihood of small and subtle errors is really great and often results in a disastrous loss of money, time or even human life. Therefore, it is the major goal of software engineering to enable developers to construct realiable systems. The use of *formal methods*, i.e. mathematically-based techniques, tools or languages, is one way to achieve this goal. With these formal methods complex systems can be both specified and verified. Althought they do not guarantee correctness, they can help ruling out inconsistencies, redundancies and incompletenesses.[CWA+96]

One method that is used to verify formal systems is called *model checking*, that is, a model is derived from a formal system $\mathcal{S}$ to verify if it satisfies a specification φ, typically a temporal logic formula. First, the *behaviour graph* $(\mathcal{S}, \neg\varphi)$ is build. This graph is the product of the temporal tableau for $\neg\varphi$ and the *state transition graph* for $\mathcal{S}$. Then, the model checking procedure tests if $(\mathcal{S}, \neg\varphi)$ admits any counterexample computations. Unfortunately, this procedure is applicable to finite-state systems only.

In this thesis, *deductive model checking with transition constraint systems*, an extension of *deductive model checking* [HTZ96], is presented. Deductive model checking is an extension of classical tableau-based model checking procedures, that verifies *linear-time temporal logic* specifications for reactive systems that are described by *fair transition systems*. Deductive model checking incrementally constructs the behaviour graph by starting with a temporal tableau for $\neg\varphi$ and repeatedly refining and transforming that graph until a counterexample computation is found or it is showed that such a counterexample computation is not possible. The result is that we can eliminate large portions of the behaviour graph before they are fully expanded to the state level, which can lead to considerable savings, even for finite-state systems. Unfortunately, this savings may be at cost of full automatism.

Deductive model checking with transition constraint systems also incrementally constructs the behaviour graph $(\mathcal{S}, \neg\varphi)$, but instead of starting with a temporal tableau for $\neg\varphi$, it starts with a *falsification diagram*, i. e. a labeled transition system, that encodes both $\mathcal{S}$ and the behaviour of the temporal tableau. Similary to deductive model checking, this procedure also repeatedly refines and transforms the initial falsification diagram until, if the procedure does not diverge, a counterexample computation is found or a correctness proof shows that a counterexample cannot exist. The resulting graph is also an overapproximation of the concrete system. To preserve fully automatism, deductive model checking with transition constraint systems is restricted to the verification of temporal formulas that state safety properties. It basically uses the same *transformation* and *refinement* rules as

the original deductive model checking procedure. Additionally, some modifications of those rules and heuristics are used, to further improve the procedure. Just as the original procedure, deductive model checking with transition constraint systems is not guaranteed to terminate for infinite-state systems, in general. But partial results may be very useful too. They give possible counterexample computations that can be used for further verification or testing.

This thesis is organized as follows: Chapter 2 gives the theoretical background for phase event automata, which are presented in chapter 3, and for deductive model checking, which is presented in chapter 4. In chapter 5 deductive model checking with transition constraint systems is introduced. An analysis of the procedure is given in chapter 6. Finally, I conclude with chapter 7.

Chapter 2

Preliminaries

2.1 Reactive systems

A *Reactive system* is ideally a non terminating system permanantly interacting with its environment, whereas the environment is superior to the system. That is, the system must react to every stimuli from its environment. The processing of such an input has to take place within a limited period of time.
Applications of reactive systems are for example embedded systems, real-time systems, communication protocols and human-machine interfaces.
The most important characteristics of a reactive system are:

- Correctness
- Concurrency

Whereas reactive systems typically are structured as concurrent systems, correctness is a property that has to be proved for such a system.

2.2 Real-time systems

A Real-Time System responds in a (timely) predictable way to unpredictable external stimuli. In short, a Real-Time System has to fulfil under extreme load conditions:

- timeliness: meet deadlines, it is required that the application has to finish certain tasks within the time boundaries it has to respect
- simultaneity or simultaneous processing: more than one event may happen simultaneously, all deadlines should be met
- predictability: the real-time system has to react to all possible events in a predictable way
- dependability or trustworthiness: it is necessary that the real-time system environment can rely on it

Generally, two classes of real-time systems are distinguished: *Hard* and *soft* real time sytems. An example of a hard real time system is a control system of an aircraft. The time boundaries must not be violated under any circumstances, otherwise the aircraft is not controllable. Live audio-video systems are an example of a soft real-time system; the violation of constraints results in degraded quality, but the system can continue to operate.

2.3 Model Checking

> Model checking is an automatic technique for verifying correctness properties of safety-critical reactive systems. [CS01]

Model checking is besides *theorem proving* a very advanced technique for verifying such reactive systems. It checks if a model of a reactive system, often derived from a software or hardware design, satisfies a formal specification. This specification is often expressed in a *propositional temporal logic*. Formally, the model checking problem can be stated in the following way: given a desired property, i. e. a temporal logical formula p, and a model M with initial state s. Does $M, s \vDash p$ hold?

Model checking, in contrast to some *deductive* methods like for example *interactive theorem proving*, is fully automatic, i. e. there is no need for user interaction. The model checker's input is the finite description of the system and the specification. Note that the state space of the system does not have to be finite. The output is 'true' if the model satisfies the specification. Otherwise, if the method does not diverge, a counterexample is produced. Another advantage of model checking is the possibility to check also partial specifications, i. e. it also provides some useful information, although the system's specification is incomplete.

Today, there are two main types of model checking that are differentiated:

- In *temporal model checking*, specifications are formalized in temporal logic and systems are modeled as *finite state transition systems*. Efficient search procedures then check if the transition system is a model for the specification.

- In another approach, both the system and the specification are modeled as automata. These automata then are compared to determine if their behaviour is conform.

The main disadvantage of model checking is the so called *state explosion problem*, i. e. the combinatorial grow of the state space, which requires an exhaustive exploration of the search space. Different approaches, like the efficient representation of state transition systems with ordered binary decision diagrams [Bry86], has been made to work against this problem.

2.4 Fair Transition Systems

A *fair transition system* (FTS) is a triple $\langle \mathcal{V}, \Theta, \mathcal{T} \rangle$, where $\mathcal{V}$ is a set of variables, Θ is the initial condition, and $\mathcal{T}$ is a finite set of transitions. A finite set of *system variables* $\mathrm{V} \subset \mathcal{V}$ determines the possible states of the system. The *state-space*, Σ, is the set of all possible valuations of the system variables.

A first-order assertion language $\mathcal{A}$ is used to describe Θ and the transitions in $\mathcal{T}$. Θ is an assertion over the system variables V. A transition τ is desribed by a *transition relation* $p_\tau(\vec{x}, \vec{x}')$, an assertion over the set of system variables $\vec{x}$ and a set of *primed* variables $\vec{x}'$ indicating their values at the next state. $\mathcal{T}$ includes an *idling transition*, *Idle*, whose transition relation is $\vec{x} = \vec{x}'$.

A *run* is an infinite sequence of states $s_0, s_1, \ldots$ such that s_0 satisfies Θ, and for each $i \geq 0$, there is some transition $\tau \in \mathcal{T}$ such that $p_\tau\ (s_i, s_{i+1})$ evaluates to true. We then say that τ is *taken* at s_i, and that state s_{i+1} is a τ-successor of s. A transition is *enabled* if it can be taken at a given state. Such states are characterized by the formula

$$\mathrm{enabled}(\tau) \stackrel{\mathrm{def}}{=} \exists \vec{x}'.p_\tau(\vec{x}, \vec{x}')\ .$$

As usual, the *strongest postcondition* $post(\tau,\phi)$ and the *weakest precondition* $wpc(\tau,\phi)$ of a formula ϕ relative to a transition τ are defined as follows:

$$post(\tau,\phi) \stackrel{\text{def}}{=} \exists \vec{x_0}.(p_\tau(\vec{x_0},\vec{x}) \wedge \phi(\vec{x_0}))$$

$$wpc(\tau,\phi) \stackrel{\text{def}}{=} \forall \vec{x}'.(p_\tau(\vec{x},\vec{x}') \rightarrow \phi(\vec{x}')) \ .$$

Fairness: The transitions in $\mathcal{T}$ can be optionally marked as *just* or *compassionate*. A just (or *weakly fair*) transition cannot be continually enabled without ever being taken; a compassionate (or *strongly fair*) transition cannot be enabled infinitely often but taken only finitely many times. A *computation* is a run that satisfies these fairness requirements.

2.5 Example - Fischers' Mutual Exclusion Algorithm

Fischer's algorithm is an example of a timed mutual exclusion algorithm. It allows n timed processes to access a shared resource in mutual exclusion. Every process holds an identifier i and a clock c_i. Additionally, all processes have two constants T_1, T_2 and a shared variable X. The algorithm uses the shared variable X, ranging from 0 to n, to indicate which process wants to access the shared resource. $X = 0$ means that no process requests the source. Once it is requested by a process, the shared variable X has to be tested. If $X = 0$, the process has to reset its clock to 0 and has to set X to its identifier i in the next T_1 time units, i. e. T_1 has to be larger than the process own clock c_i. Then the process has to wait at least T_2 time units, i. e. it tests for $T_2 < c_i$. If its identifier i is not equal to X, it will retry later because another process has requested the resource. Otherwise the process can enter the critical section. When leaving the critical section, X is reset to 0.

Figure 2.1 shows the automaton for process i. The process can be at one of the four locations $pc_0, \ldots, pc_3$, where pc_3 is the critical section. T_1 and T_2 are constants, that are previously defined. To obtain mutual exclusion we additionally have the invariant constraint $T_1 < T_2$. The corresponding transition system for n processes is the triple $\langle \mathcal{V}, \Theta, \mathcal{T} \rangle$, with:

- $\mathcal{V} = pc_1, \ldots, pc_n, x, c_1, \ldots, c_n$
- $\Theta = x = 0 \wedge \forall i.pc_i = 0 \wedge c_i = 0$
- $\mathcal{T} = \forall i \in \{1, \ldots, n\}$ there exist transitions $\tau_{1,i}, \ldots, \tau_{6,i}$, with:

  $\tau_{1,i} : x = 0, pc_i = 0, c_i' = 0, pc_i' = 1$ [1] [2]
  $\tau_{2,i} : pc_i = 1, c_i \geq T_1, pc_i' = 0$
  $\tau_{3,i} : pc_i = 1, c_i < T_1, x' = i, c_i' = 0, pc_i' = 2$
  $\tau_{4,i} : pc_i = 2, c_i > T_2, x \neq i, pc_i' = 0$
  $\tau_{5,i} : pc_i = 2, c_i > T_2, x = i, pc_i' = 3$
  $\tau_{6,i} : pc_i = 3, x' = 0, pc_i' = 0$

  and additionally we have a transition *tick* that updates the clock values

[1] Note, that commas stand for conjunctions

[2] Note, that all primed variables, that are not explicitly mentioned in the constraint, remain unchanged, i. e. $x = x' \wedge (\forall j \neq i : c_j' = c_j \wedge pc_j = pc_j')$.

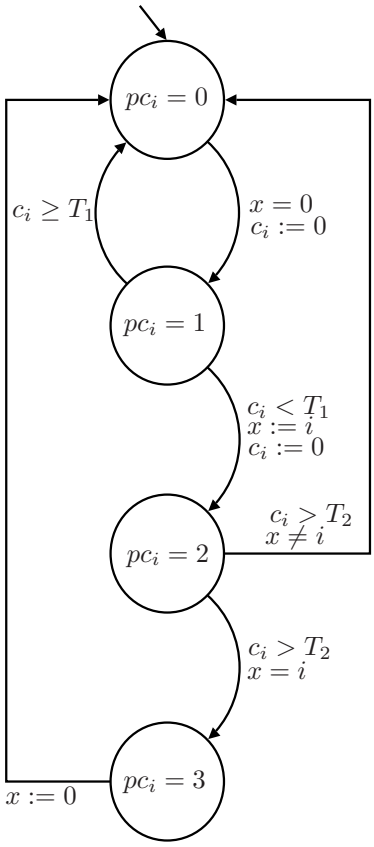

Figure 2.1: **Timed automaton for process i of Fischer's algorithm**

$$tick : c_1' > c_1, c_2' - c_2 = c_1' - c_1, \ldots, c_n' - c_n = c_1' - c_1$$

and the *idle* transition $\vec{x} = \vec{x}'$.

2.6 Linear-time Temporal Logic

> *Linear-time temporal logic* is a logic for talking about infinite sequences, where each element in the sequence corresponds to a propositional world.[Muk97]

Linear time temporal logic (LTL) is a subset of the *computation tree logic* (CTL*). In LTL time is discrete, implicit, has an initial moment with no predecessors and is infinite in the future. The elements of LTL are proposition variables $p_1, p_2, \ldots$, the usual logical connectives $\wedge, \vee, \rightarrow, \neg$ and the following *temporal modal operators*:

- **N** for next
- **G** for always (**g**lobally)
- **F** for eventually (in the **f**uture)
- **U** for until
- **R** for release

An LTL formula can be evaluated over a sequence of truth evaluations and a position on that path. It is satisfied by a path, if and only if it is satisfied for position 0 on that path. A system satisfies an LTL formula ϕ if each path through the system satisfies ϕ. In tabular 2.1 the semanitics of the modal operators are given.

Some useful equivalences:

- $\mathbf{F}\ \varphi \equiv \text{true}\ \mathbf{U}\ \varphi$
- $\mathbf{G}\ \varphi \equiv \neg\ \mathbf{F}\ \neg\ \varphi$
- $\psi\ \mathbf{R}\ \varphi \equiv \neg(\neg\ \psi\ \mathbf{U}\ \neg\ \varphi)$
- $\mathbf{F}\ \varphi \equiv \neg\ \mathbf{G}\ \neg\ \varphi$

In this thesis, a restricted linear-time temporal logic is used. It is a specification language over the assertion language $\mathcal{A}$, where no temporal operator is allowed to appear within the scope of a quantifier. These temporal formulas are called *state-quantified*[Pnu97] and use the usual temporal operators. A formula with no temporal operator is called a *state-formula* or an *assertion*. There are two main types of properties that can be stated with these formulas:

- **Safety**:

  Safety properties usally state that something we do not want never happens (e.g. $\Box p$ for a past formula p). Is p a state-formula, the property is called an *invariance*.

- **Liveness**:

  With *Liveness properties* we want to state that something we want eventually happens ($\Diamond p$).

Textual	Symbolic	Explanation	Diagram
N(or **X**)φ	$\circ\phi$	φ has to hold at the next state.	$\bullet \rightarrow \bullet \rightarrow$ N $\varphi \rightarrow \varphi \rightarrow \bullet$
G φ	$\Box\phi$	φ has to hold on the entire subsequent path.	$\varphi \rightarrow \varphi \rightarrow \varphi \rightarrow \varphi \rightarrow \varphi$
F φ	$\Diamond\phi$	φ eventually has to hold (somewhere on the subsequent path).	$\bullet \rightarrow \varphi \rightarrow \bullet \rightarrow \varphi \rightarrow \bullet$
ψ **U** φ	$\psi \mathcal{U} \phi$	φ holds at the current or a future position, and ψ has to hold until that position. At that position ψ does not have to hold any more.	$\psi \rightarrow \psi \rightarrow \psi \rightarrow \varphi \rightarrow \bullet$
ψ **R** φ	$\psi \mathcal{R} \phi$	ψ releases φ if φ is true until the first position in which ψ is true (or forever if such a position does not exist).	$\varphi \rightarrow \varphi \rightarrow \psi \rightarrow \bullet \rightarrow \bullet$

Table 2.1: **Semantics of modal operators**

Example:

In Fischer's mutual exclusion algorithm 2.5, we want to define an error state. Assume we have 2 processes. Consequently, we have the following 'error condition':

$$pc_1 = 3 \wedge pc_2 = 3$$

The resulting temporal logic formula that states that we never want to reach that state is:

$$\Box \neg (pc_1 = 3 \wedge pc_2 = 3)$$

The negation of this formula that we will need in the next section we just get by using the equivalences given above:

$$\neg (\Box \neg (pc_1 = 3 \wedge pc_2 = 3))$$

$$\equiv \quad \Diamond (pc_1 = 3 \wedge pc_2 = 3)$$

2.7 The Formula Tableau

As mentioned in section 2.3, the model checking problem is to check if some model satisfies a given property φ. We know that a formula φ is valid if and only if $\neg\varphi$ is unsatisfiable. The classic model checking algorithm for finite systems constructs a *behaviour graph*, that is the product of the state transition graph of a system S and the formula tableau $\Phi_{\neg\varphi}$ and checks if this behaviour graph contains a strongly connected subgraph (SCS) that is *fulfilling* and also satisfies the with S accociated fairness requirements [HTZ96]. Is there a path from an initial node to such an SCS, a counterexample can be produced, otherwise φ is valid. From [MP95] we know that, if there is given an LTL formula φ, it is possible to construct its tableau Φ_φ, that is a finite graph that describes all of its models and whose size normally is exponential in the size of the formula.

In the following, I show how the formula tableau for a formula φ is constructed. Hence, I use an incremental tableau algorithm, that constructs a graph G whose nodes (atoms) are labeled with sets of formulas derived from φ. Every model

of φ is represented as an infinite path in G. The advantage of an incremental algorithm is that only atoms reachable from an initial atom are present, that it is more efficient and that incremental tableaux are better for implementation [BAMP81, MW84]. A detailed describtion of the incremental tableau algorithm I use, is given in [KMMP93].

Given the formula $\varphi \equiv \mathbf{F}\ (pc_1 = 3 \wedge pc_2 = 3)$. Because in the given algorithm the formula φ is non-basic, it is first rewritten as $\mathit{true}\ \mathbf{U}\ (pc_1 = 3 \wedge pc_2 = 3)$ (see 2.6).

First, the preconditions of φ and $\neg\,\varphi$ are constructed:

- precondition of φ: $\{\underbrace{pc_1 = 3 \wedge pc_2 = 3}_{=:\psi}\}, \{\mathit{true}, \underbrace{\mathbf{N}(\mathit{true}\mathbf{U}(pc_1 = 3 \wedge pc_2 = 3))}_{=:\mathbf{N}\varphi}\}$
- precondition of $\neg\varphi$: $\{\mathit{false}, pc_1 \neq 3 \vee pc_2 \neq 3\}, \{pc_1 \neq 3 \vee pc_2 \neq 3, \neg\mathbf{N}\varphi\}$

The *closure* of φ, $CL(\varphi)$ is:

- $\{\mathit{first}, \neg\mathit{first}, \varphi, \neg\varphi, \psi, \neg\psi, \mathit{true}, \mathit{false}, \mathbf{N}\varphi, \neg\mathbf{N}\varphi\}$

Because the model checking algorithm presented in this paper only uses LTL formulas that describe *future* properties, we can throw away the predicates *first* and $\neg$ *first*.

The set *Cover*(φ) that is used in the algorithm *construct-initial* is constructed by the algorithm *incremental-cover* and is given below:

$$\mathit{Cover}(\varphi) = \{\{pc_1 = 3, pc_2 = 3, \varphi\}, \{\mathit{true}, \mathbf{N}\varphi, \varphi\}\}$$

The algorithm *construct-initial* first defines following sets:

$F = \{\}$

$\mathcal{V} = \{\ \mathrm{F}\}$

$\mathcal{E} = \{\langle\ \mathrm{F},\ \mathrm{F}\ \rangle\}$

$\overline{\mathcal{E}} = \{\}$

Now, for each atom A of *Cover*(φ), A is added to $\mathcal{V}$ and an edge $\langle$ A, F $\rangle$ is added to $\mathcal{E}$. Applying the method *correct-graph* results in figure 2.2.

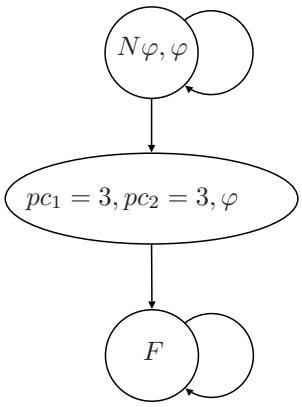

Figure 2.2: **Formula tableau for** φ

Because the model checking algorithm presented in this paper only cares about *Safety*, all formulas that represent *liveness* properties can also be thrown away. Additionally, the node with label F and the node with label $pc_1 = 3, pc_2 = 3$ can be merged into one node. Figure 2.3 shows the final tableau for φ that is also the input data for the model checking algorithm.

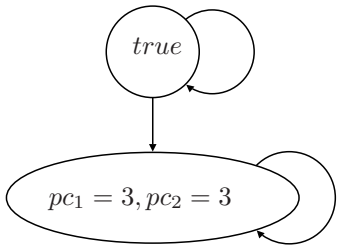

Figure 2.3: **Simplified formula tableau for** φ

2.8 CSP-OZ-DC

In this section, the language *CSP-OZ-DC* is presented, a combination of the formal languages *Communicating Sequential Processes* (CSP), *Object-Z* (OZ) and *Duration Calculus* (DC). In the following, I shortly introduce the three formalisms. For a detailed description of the languages I refer to [Hoe06].

2.8.1 Communicating Sequential Processes

CSP is a formal language for communicating sequential processes that was introduced by Hoare [Hoa78]. CSP can describe the behaviour of sequential and parallel processes. These processes communicate via instantaneous events. The real-time a communication needs is abstracted into a single moment. Processes that run in parallel communicate synchronous.

Example:

$\text{MAIN} \stackrel{c}{=} newgoal \rightarrow start \rightarrow Drive$

$Drive \stackrel{c}{=} (passed \rightarrow Drive) \,\Box\, (stop \rightarrow \text{MAIN})$

This example shows the control and communication aspects of an elevator. The elevator switches (cyclic) between the two processes MAIN and *Drive*. MAIN is the initial entered process. The elevator chooses a floor (*newgoal*), then it starts the engine and switches to the process *Drive*. In the *Drive* process, the elevator can either pass a floor and resume driving or it can stop and return to the MAIN process. The $\Box$ - symbol stands for an external choice which is determined by the environment.

2.8.2 Object-Z

Object-Z is an object-oriented extension of the formal specification language Z that is used for describing and modelling computing systems. Z defines notations for logical operations, quantifiers, sets and functions. It also provides a way to represent large state spaces and operations. It has been developed by Smith [Smi00] at the Software Verification Research Center in the University of Queensland.

Example:

In figure 2.4 we see the algorithmic part and and the data state of the elevator example in Object - Z. The floors are presented as integers ranging from constants *Min* to *Max* and *Min* is always smaller than *Max*. These boundaries are parameters of the elevator. The internal state of the elevator is given by the two schemas top right. The internal state consists of two variables *current* and *goal* and a variable

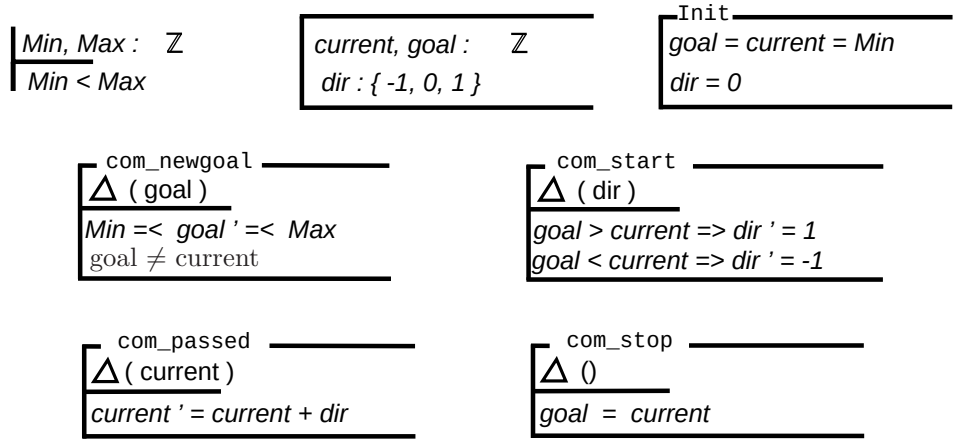

Figure 2.4: **Data aspects of an elevator specification in Object - Z**

dir which stands for the direction the elevator is driving (-1 for downwards, 1 for upwards). The initial values are given by the special schema *Init*.

The connection between states and events is presented by so called *communication schemas*. The schema *com_passed* for example describes the change of the state variable *current* that is caused by the event *passed*. The Δ list determines the variables that are changed by the operation. Here, *current* is increased or decreased by the value of *dir*. The event *newgoal* chooses non-deterministically a new *goal* floor that is different from *current*. *start* changes the value of *dir* depending on the relation between *current* and *goal*. The elevator can only stop if the *goal* floor and the *current* floor are equal (no variable is changed).

2.8.3 Duration Calculus

The third part of CSP-OZ-DC, Duration Calculus (DC), is an interval logic for real-time systems. It was originally developed by Zhou Chaochen, Anders P. Ravn and C. A. R. Hoare on the European ESPRIT Basic Research Action (BRA) ProCoS project on Provably Correct Systems [ZHR91].

In CSP-OZ-DC only a restricted class of DC is used because the full language is too powerful to be checked automatically. This class is called *counterexample formulae* and describes a behaviour we do not want as a linear trace. A counterexample formula generally looks like follows:

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

A formula $\Diamond\varphi$ states that there is a subinterval where φ holds. φ itself is split into n subintervals, each satisfiying $phase_i$. $Phase_i$ has to be a simple formula that restricts the current state of the system.

Example:

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

$$\neg \Diamond (\ \lceil current \neq goal \rceil;\ (\ \lceil current = goal \rceil \wedge l \geq 2 \wedge \boxminus stop))\ ^{4}$$

The real-time aspects of the elevator are presented by the two DC formulas above. They ensure that the elevator stops as soon as it reaches the goal floor and before passing the next floor. The first counterexample formula states that there always has to be a minimum time of three seconds between two adjacent *passed* events. The second formula states that the elevator has to stop within two seconds when reaching the goal floor.

[3] $\updownarrow$ *passed* holds for a point interval, at which the variable *passed* changes (see [ZH04])

[4] The formula $\boxminus$ *stop* states that the event stop does not occur during a non-empty interval (see [ZH04]).

Chapter 3

Phase Event Automata

3.1 Timed Automata

Most traditional model checking techniques have been unsuitable for the analysis of real time systems because they are not able to explicitly model time. Therefore, *timed automata* have been introduced to model the time behaviour of real-time systems [AD94].

In [Hen06] a timed automaton is defined as a tuple $(L, l_0, \Sigma, X, I, E)$, where L is a finite set of locations, $l_0 \in$ L is the initial location, Σ is a finite set of synchronization labels, X is a finite set of clocks, I : L $\rightarrow \Phi(X)$ labels each location with some clock constraint, and E $\subseteq$ L $\times \Sigma \times \Phi(X) \times 2^X \times$ L is a finite set of edges.

Example:

In figure 3.1 an example of a simple timed automaton is presented. The initial location is s and there is only a single clock x. Because the initial location has no invariant constraint, the system can spend an arbitrary amount of time in s. When the system takes an a-transition to location s', the clock is reset to 0. The value of x at location s' shows the amount of time elapsed since the last switch. The invariant $x \leq 2$ ensures that the system can only stay for two time units at this location. A switch back to location s is only possible if $x \geq 1$.

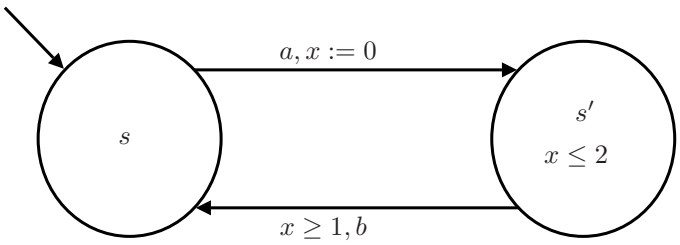

Figure 3.1: **A simple timed automaton**

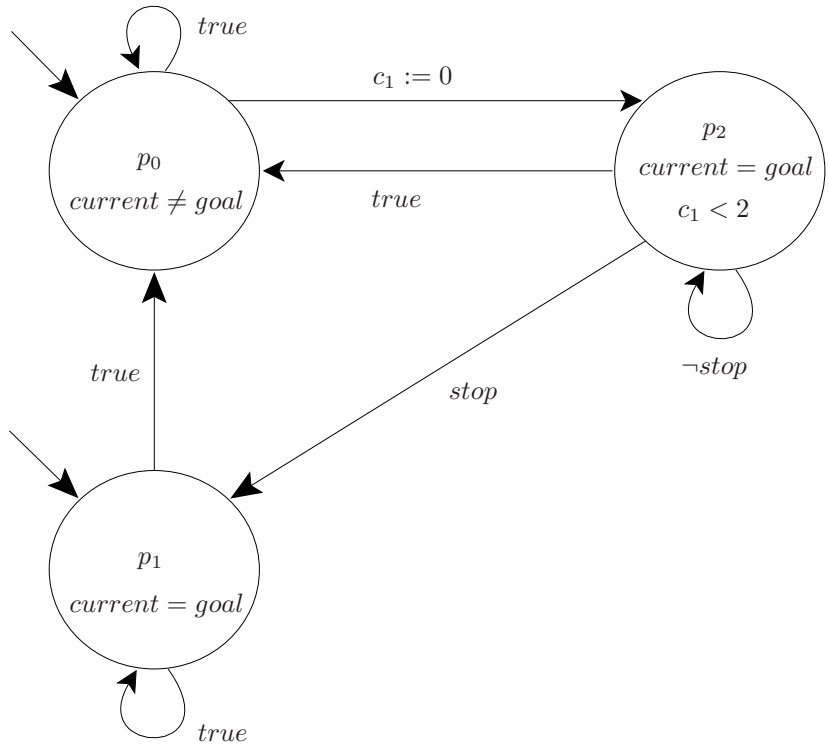

Figure 3.2: **A phase event automaton**

3.2 Phase event automata

Phase event automata (PEA) are a new class of timed automata that can characterize the behaviour of state- and event-based systems and provide an essential prerequisite for model checking [HM05]. PEA provide a connection between CSP-OZ-DC (2.8) and transition constraint systems. Due to their notion of events, variables and clocks and their ability to preserve events and data variables of the CSP-OZ-DC specification, they can describe both immediate events from the CSP part (2.8.1), states with durations - they model the Object-Z state variables (2.8.2) - and clocks for real-time constraints, which are predefined by Duration Calculus (2.8.3).

The conversion of a CSP-OZ-DC specification into phase event automata is a parallel product of automata, one for each part of the specification. Thus, PEA provide a compositional operational semantics for CSP-OZ-DC [HM05].

Phase event automata are converted to transition constraint systems by translating the *continuous transitions* of the PEA into explicite *discrete transitions*, modelling *events* by state changes and giving up the distinction between *state* and *event variables*. The result is a full automation of the translation process and the final model checking.

In figure 3.2 an example of a phase event automaton is given. This PEA represents the second formula of the elevator example in Duration Calculus (2.8.3):

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

The formula states that the elvator has to stop within two time units when it reaches the goal floor. Initially the automaton can be in two phases. It is in phase p_0, if the actual floor (current) is not the destination floor (goal) or in phase p_1, if the actual floor is the goal floor. If there is a change from $current \neq goal$ to

current = *goal*, the automaton has to switch to phase p_2 and the clock c_1 is reset to 0. The predicate $c1 < 2$ ensures that the elevator stops within the next two time units, i. e. phase p_2 has to be left in time. If *current* $\neq$ *goal* holds, it can go back to phase p_0, otherwise the *stop* event is requested.

In this thesis (see chapter 5), phase event automata are the underlying structure the model checking procedure works on. In the next section (3.3) I will give a formal definition of phase event automata. In section 3.4 a parallel composition operator is defined, because the different phase event automata, resulting from a separated translation of each part of the CSP-OZ-DC formalism, have to be conjoined.

3.3 Formal Definition

Let $\mathcal{L}$ be the class of first-order formulae that are allowed by the specification. A *phase event automaton* (Pea) is a tuple $\mathcal{A} = (\mathrm{P}, \mathrm{V}, \mathrm{A}, \mathrm{C}, \mathrm{E}, \mathrm{s}, \mathrm{I}, P_0)$ of the following components:

- P is a set of states (phases).
- V $\subseteq \mathcal{V} \setminus$ (**Events** $\cup$ **Clocks**) is a finite set of (state) variables.
- A $\subseteq$ **Events** is a finite set of events.
- C $\subseteq$ **Clocks** is a finite set of clocks.
- E $\subseteq P \times \mathcal{L}(\mathcal{V} \cup \mathcal{V}' \cup \mathcal{A} \cup \mathcal{C}) \times \mathcal{P} \times \mathcal{P}$ is a set of edges. An edge $(p_1, g, X, p_2) \in$ E represents a transition from phase p_1 to phase p_2 under guard g. All clocks in X are reset when this transition is taken [1].
- s: P $\rightarrow \mathcal{L}$(V) is a labeling function that associates each phase with a predicate that must hold during this phase.
- I: P $\rightarrow \mathcal{L}$(C) is a function assigning to each phase a clock invariant that has to hold while the automaton is in that phase.
- $P_0 \subseteq P$ is a set of possible initial phases.

Additionally we have:

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

Let $\mathcal{A} = (\mathrm{P}, \mathrm{V}, \mathrm{A}, \mathrm{C}, \mathrm{E}, \mathrm{s}, \mathrm{I}, P_0)$ be a PEA. A *state* of A is a triple (p, β, γ) of a phase p $\in$ P, a V-valuation β and a C-valuation γ. A *duration* is a positive real number. A *run* of $\mathcal{A}$ is an infinite sequence

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

with alternating states (p_i, β_i, γ_i), durations t_i and sets of events $Y_i \subseteq A$. The following always has to hold:

1. $p_0 \in P_0$.
2. For all $c \in C, \gamma_0(c) = 0$.
3. For all $i \geq 0, \beta_i \models s(p_i)$

[1] $\mathcal{L}$(V) denotes the set of those formulae in $\mathcal{L}$ that only refer to variables V $\subseteq \mathcal{V}$.

4. For all $i \geq 0$ and all $0 \leq \delta \leq t_i, \gamma_i + \delta \models I(p_i)$
5. For all $i \geq 0$ there is an edge $(p_i, g, X, P_{i+1}) \in E$ such that
   (a) $\beta_i \cup \beta'_{i+1} \cup (\gamma_i + t_i) \cup \mathcal{X}_{Y_i}$[2] $\models g$ and
   (b) $\gamma_{i+1} = (\gamma_i + t_i)[X := 0]$[3].

The stuttering edge $(p_i, \neg e_1 \wedge \ldots \wedge \neg e_k \wedge v_1 = v'_1 \wedge \ldots \wedge v_j = v'_j, \emptyset, p_i)$ is required to make the definition invariant against stuttering and simplifies the definition of parallel composition, because automata can step synchronously.

3.4 Parallel Composition of Phase Event Automata

In [HM05], for every part of the CSP-PZ-DC specification a special phase event automaton is build. Therefore we need an operator that composes these automata in parallel. The phase event automata are synchronised on both states and events: A variable that occurs in two automata can only be changed if both agree. Also, an event that is in the alphabet of two automata can only be taken, if both allow it. Clocks have to be disjoint, because the must not interfere with each other. Next, a formal definition of the parallel composition $A_1 \| A_2$ of two automata A_1 and A_2 with $A_i = (P_i, V_i, A_i, C_i, E_i, s_i, I_i, P_{0i})$ is given. The result is the PEA $\mathcal{A}= (P, V, A, C, E, s, I, P_0)$ with:

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

[2] $\mathcal{X}_E$ is the characteristic function of E, that is, the mapping from **Events** to $U_{Bool} = \mathbb{B}$ such that for all $e \in$ **Events**, $\mathcal{X}_E(e) = true$ iff $e \in E$. $\mathcal{X}_E$ is an **Events** - valuation.

[3] All clocks in X is assigned the value 0.

Chapter 4

The DMC Procedure - Safety

In this chapter, the concept of *deductive model checking (DMC)* is presented. It is based on [HTZ96] and represents an extension of the classical tableau-based model checking procedures for infinite-state systems. The deductive model checking procedure given therein is used to verify *linear-time temporal logic* (2.6) specifications of *reactive systems* (2.1). These systems are described by *fair transition systems* (2.4).

To preserve the property of full automatism, the model checking procedure presented in chapter 5 is restricted to verify *safety properties* ofreactive systems. Therefore, the description of the model checking procedure in this chapter will be restricted to those operations that are sufficient for the analysis of safety properties. Below, I will first give an short informal description of the *DMC procedure* (4.1). Then, I present the 'restricted' DMC procedure itself (4.2).

4.1 Deductive Model Checking

Classical tableau-based model checking procedures check the *behaviour graph* to verify that a system satisfies a temporal specification. The behaviour graph $(S, \neg\varphi)$ of a system S and a temporal specification φ is the product of the temporal tableau for $\neg\,\varphi$ and the state transition graph for S. S satisfies φ if the behaviour graph does not admit any counterexample computation. This procedure unfortunately is applicable to finite-state systems only.

Instead of explicitely building the behaviour graph $(S, \neg\,\varphi)$ of a system S and a specification φ, the deductive model checking procedure starts with a general representation of the behaviour graph and progressivly refines it. The graph contains all possible computations that violate φ and is called the *initial falsification diagram* for S and φ.

> **Definition:** A *falsification diagram* for a finite transition system S and a temporal property φ is a directed graph $\mathcal{G}$ whose nodes are labeled with pairs (A, f), where A is a temporal tableau atom for $\neg\,\varphi$ and f is a state formula. Edges of $\mathcal{G}$ are labeled with subsets of $\mathcal{T}$, the set of transitions of S. For nodes M and N, we write $\tau \in \langle M, N\rangle$ if transition τ is in the label of the edge from M to N. A subset of the nodes in $\mathcal{G}$ is marked *initial*.

4.2 The Restricted DMC Procedure

We start with the tableau graph $\Phi_{\neg\varphi}$ whose construction is described in 2.7. From this graph, the initial falsification diagram $\mathcal{G}_0$ is constructed as follows:

1. Replace each node label A by (A, f_A), where f_A is the conjunction of the state formulas in the tableau atom A.

2. For each node N: (A, f) such that A is initial in the tableau $\Phi_{\neg\varphi}$, add a new node N_0: $(A, f \wedge \Theta)$, with no incoming edges, whose outgoing edges go to exactly the same nodes as those of N. A self-loop $\langle N, N\rangle$ becomes an edge $\langle N_0, N\rangle$ in the new graph. These new nodes are the *initial nodes* in the falsification diagram.

3. Label each edge in $\mathcal{G}_0$ with the entire set of transitions $\mathcal{T}$.

Example:

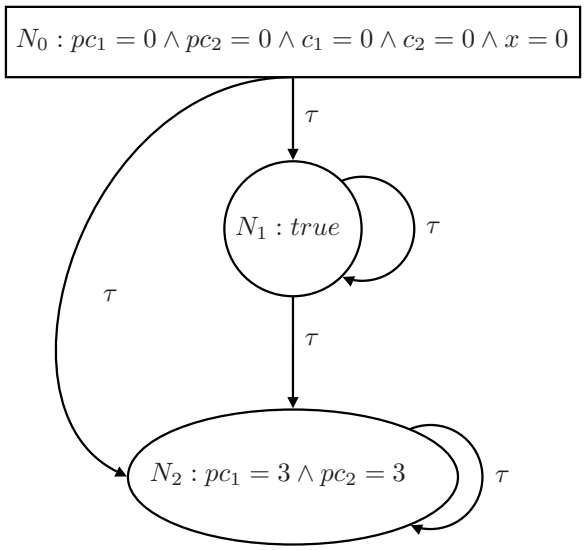

Figure 4.1: **Initial falsification diagram for Fischer's mutual exclusion**

In chapter 2 Fischer's mutual exclusion algorithm (2.5) and its representation as finite transition system $\mathcal{S}$ is given. In figure 2.3, the tableau graph for the negated specification $\neg\,\varphi$ is represented. In figure 4.1, we see the initial falsification diagram for $\neg\,\varphi$ (i. e. $\Diamond\,(pc_1 = 3 \wedge pc_2 = 3)$). It is constructed as follows: The two nodes N_1, N_2 of the tableau graph for $\neg\,\varphi$ are replaced by $(A_1, f_{A_1}) = (1, true)$ and $(A_2, f_{A_2}) = (2, pc_1 = 3 \wedge pc_2 = 3)$. Both nodes are marked *initial*, because the system could initially be in the *error-state* too.[1] Now, for both nodes new initial nodes are introduced:

$N_{01} : (01, true \wedge \Theta)$

$N_{02} : (02, (pc_1 = 3 \wedge pc_2 = 3) \wedge \Theta)$,

where Θ is the initial condition. Additionally, new edges $\langle N_{01}, N_1\rangle$, $\langle N_{01}, N_2\rangle$, $\langle N_{02}, N_2\rangle$ are constructed. All edge labels now are marked with $\mathcal{T}$, the entire set

[1] Error-states are those states for which, as long as they can be reached from an initial state, a counter-example computation exists.

of transitions in $\mathcal{S}$. Note, that node N_{02} is immediately pruned since its label is unsatisfiable. After having constructed the Initial falsification diagram, one of the following transformations have to be repeatedly applied. The deductive model checking procedure stops if there is no path from an initial node to an error-node, in which case we have a correctness proof. Otherwise, the procedure diverges or if no *basic refinement transformation* can be applied anymore, it stops and we get a counter-example computation.

Basic Transformations:

- **1 (remove edge label).** If an edge $\langle (A_1, f_1), (A_2, f_2) \rangle$ is labeled with a transition τ and the assertion

  $$f_1(\vec{x}) \wedge f_2(\vec{x}') \wedge p_\tau(\vec{x}, \vec{x}')$$

  is unsatisfiable, remove τ from the edgelabel.

- **2 (empty edge).** If an edge is labeled with the empty set, remove the edge.

- **3 (unsatisfiable node).** If f is unsatisfiable for a node (A, f), or if a node has no successors, remove the node.

- **4 (unreachable node).** Remove a node unreachable from an initial node.

Example:

As noted above, Figure 4.1 shows the initial falsification diagram for the temporal formula $\varphi := \Diamond(pc_1 = 3 \wedge pc_2 = 3)$. In this graph, the transformation rule **unsatisfiable node** has been already applied since the node N_{02} is pruned.

Next, the **remove edge label** transformation will remove for example all transitions in the label of the edge $\langle N_{01}, N_2 \rangle$, because there is no transition τ_i that can satisfy the constraint $f_1(\vec{x}) \wedge f_2(\vec{x}') \wedge p_{\tau_i}(\vec{x}, \vec{x}')$. Hence, the **empty edge** transformation rule will remove this edge from the diagram.

Node Splitting: In transformations 5 and 6, a node N is replaced by new nodes N_1 and N_2. Any incoming edge $\langle M, N \rangle$ is replaced by edges $\langle M, N_1 \rangle$ and $\langle M, N_2 \rangle$ with the same label, for $M \neq N$. Similary, any outgoing edge $\langle N, M \rangle$ is replaced by edges $\langle N_1, M \rangle$ and $\langle N_2, M \rangle$ with the same label as the original edge.

If a self-loop $\langle N, N \rangle$ was present, new edges $\langle N_1, N_1 \rangle$, $\langle N_2, N_2 \rangle$, $\langle N_1, N_2 \rangle$, $\langle N_2, N_1 \rangle$, all with the same label as $\langle N, N \rangle$. If an initial node is split, the two nodes are also labeled initial.

Basic Refinement Transformations:

- **5 (postcondition split).** Consider an edge from node N_1: (A_1, f_1) to N_2: (A_2, f_2) whose label includes transition τ. If $f_2 \wedge \neg\, post(\tau, f_1)$ is satisfiable (that is, f_2 does not imply $post(\tau, f_1)$), then replace (A_2, f_2) by the two nodes:

  $$N_{2,1} = (A_2, f_2 \wedge post(\tau, f_1))$$
  $$N_{2,2} = (A_2, f_2 \wedge \neg\, post(\tau, f_1)).$$

  We can immediately apply the **remove edge label** transformation to the edge between N_1 and $N_{2,2}$, removing τ from its label.

  Nodes N_1 and N_2 need not be distinct. If $N_1 = N_2$ then we split the node into two nodes as above, but now the self-loop for $N_{2,2}$ and the edge from $N_{2,1}$ to $N_{2,2}$ will not contain the transition τ.

- **5 (precondition split).** Consider an edge

  $$\langle N_1, N_2\rangle = \langle (A_1, f_1), (A_2, f_2)\rangle$$

  labeled with transition τ. If $f_1 \wedge \neg\,(enabled(\tau) \wedge wpc(\tau, f_2))$ is satisfiable, then replace (A_1, f_1) by the two nodes:

  $$N_{1,1} = (A_1, f_1 \wedge enabled(\tau) \wedge wpc(\tau, f_2))$$
  $$N_{1,2} = (A_1, f_1 \wedge \neg\,(enabled(\tau) \wedge wpc(\tau, f_2))).$$

The original deductive model checking procedure presented in [HTZ96] uses some more transformation rules. They are for example used to extend the model checking procedure to prove *fairness requirements*.

Example:

In figure 4.2 we see the result of applying the first precondition split transformation on edge $\langle N_1, N_2\rangle$ and the transition $\tau_7 : pc_1 = 2 \wedge c_1 > 3 \wedge x = 1 \wedge pc_1' = 3$ with $T_1 = 2, T_2 = 3$ and $T_1 < T_2$. Note, that all unsatisfiable edge-labels and edges have been already removed by the basic transformation rules. Because the precondition test $true \wedge \neg\,(enabled(\tau_7) \wedge wpc(\tau_7, pc_1 = 3 \wedge pc_2 = 3))$ is satisfiable, node N_1 is replaced by the following two new nodes:

$N_{1,1}$ with label: $pc_1 = 2 \wedge c_1 > 3 \wedge x = 1 \wedge pc_2 = 3$

$N_{1,2}$ with label: $(pc_1 \neq 2) \vee (c_1 \leq 3) \vee (x \neq 1) \vee$
$(pc_1 = 2 \wedge c_1 > 3 \wedge x = 1 \wedge pc_2 \neq 3)$

Next, all incoming-, outgoing- and self-edges are replaced according to the node splitting rules (4.2). Notice that the basic transformations have been already applied to the new edges (for instance, all transitions except τ_7 have been removed from edge $\langle N_{1,1}, N_{1,2}\rangle$). [2]

[2] For simplicity reasons, the *idle transition* has been omitted here.

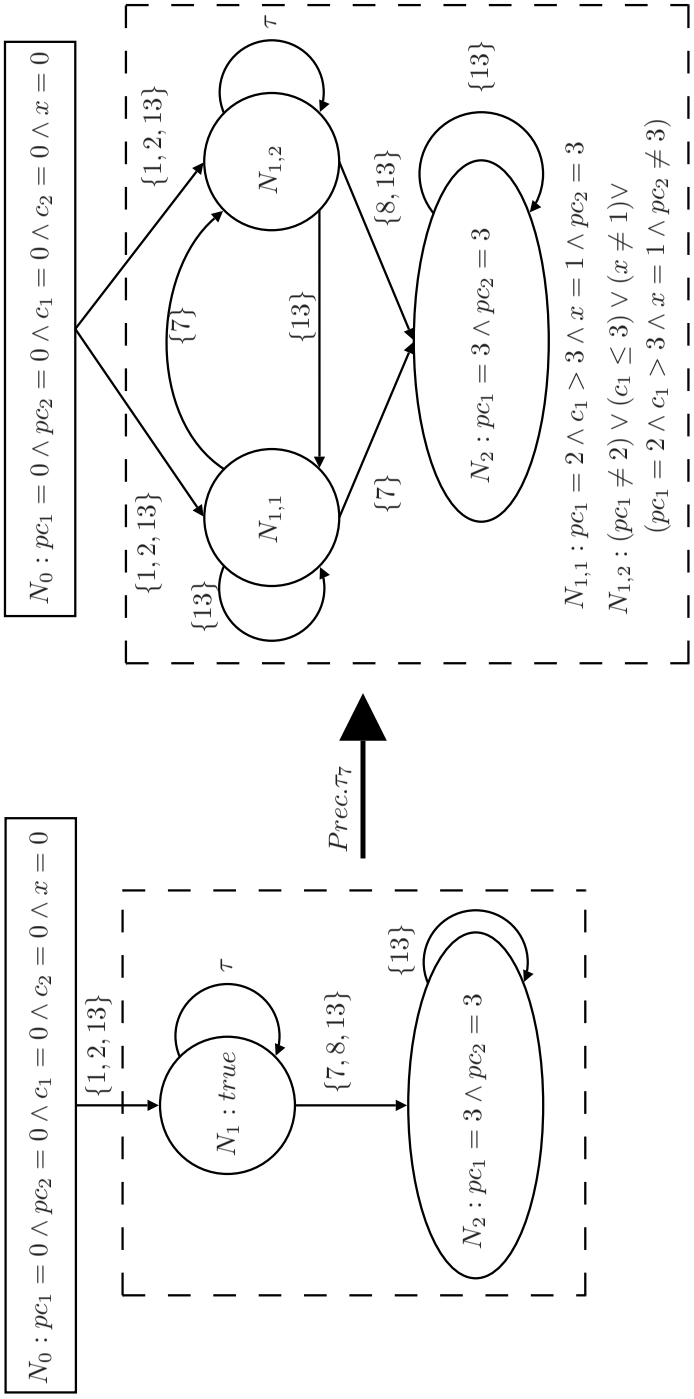

Figure 4.2: **Precondition split on edge** $\langle N_1, N_2 \rangle$

Chapter 5

DMC with Phase Event Automata

In this chapter, I present a new approach to verify temporal *safety* [1] properties of reactive systems, *deductive model checking with transition constraint systems*.

In chapter 3, phase event automata, a class of timed automata, that can characterize the behaviour of state- and event-based systems got already introduced. In chapter 4, deductive model checking, as an extension of the classical tableau based model checking procedure to verify temporal properties, was presented. Deductive model checking with transition constraint systems now combines these two formalisms. Instead of building a tableau graph for the negated specification formula ($\Phi_{\neg\varphi}$) for a given transition system $\mathcal{S}$, we want to verify a phase event automaton that comprises both the behaviour of the transition system $\mathcal{S}$ and the structure of $\Phi_{\neg\varphi}$.

In [Hoe06] is showed how CSP-OZ-DC specifications [2] are first translated into phase event automata before being converted to transition systems and finally model checked. At present, the underlying phase event automaton, that is repeatedly refined during the procedure, is build from a configuration file the user has to specify in advance. The deductive model checking procedure presented in this thesis works with labeled transition systems that represent phase event automata. This is done because of structural reasons, i. e. to make them applicable for the deductive model checking procedure given in [HTZ96]. Nevertheless, a nice feature is, that in the future, CSP-OZ-DC specifications could also be translated to transition constraint systems and with only little changes serve as underlying structure to this model checking procedure.

The deductive model checking algorithm itself is implemented in SICStus Prolog (version 3.12.12). This commercial Prolog version extends the original language for logic programming applications with an object oriented component and dedicated constraint solvers for linear equalities. The transition systems that are worked on here, are encoded as SICStus objects. The constraint logic library for rationals and reals is used to check the satisfiability of the constraints that incur during the basic and refinement transformations.

DMC with transition constraint systems first converts the configuration file mentionend above into a labeled transition system. After having been read in, the input file is converted into tokens. The resulting token list is additionally parsed

[1] This approach is not really restricted to formulas that state safety properties. This decision is made to preserve the fully automatism of the model checking procedure.

[2] CSP-OZ-DC specifications are not only well-investigated specification techniques but also quite intuitive and comprehensible.

to check the correctness of the configuration data, especially the given constraints. Finally, an 'object-creater' module translates the token list into a 'phase event automaton'(transition system). In the second step, the *initial falsification diagram* is constructed. This algorithm differs from the original construction algorithm given in [HTZ96]. For efficiency reasons, some heuristics are included in the procedure. When the initial falsification diagram is constructed, the **basic and refinement transformations** repeatedly refine the system until a counter-example computation or a correctness proof is found. Unfortunately, if the problem is undecidable in general, the algorithm also can diverge. The transformations given in this thesis also differ from the original transformations. These differences also are explained in detail later.

Some problems the algorithm has to deal with at present are the restriction of the specification language to safety properties. Fortunately, this problem is not unsolvable. It should not be a big problem to extend the algorithm with modules that implement those transformations in [HTZ96], that extend the specification language to *fairness* properties for example. Althought the incremental character of the model checking and the restriction to check local conditions, leads to significant savings concerning the state-space (compared to classic tableau-based model ckecking procedures for example), the state space in this approach is still quite large. Additional heuristics are used to deal with this problem.

This chapter is organized as follows: The first section (5.1) defines the grammar of a configuration script. Section 5.2 describes how the given input file is read in, tokenized, checked for correctness and finally translated into a 'phase event automaton'. In section 5.3, we see the construction of the *initial falsification diagram*. Section 5.4 presents the **basic transformations**. Finally, section 5.6 both describes the *precondition split* and the *postcondition split* rules of the **basic refinement transformations** and the heuristics used to improve the model checking algorithm.

5.1 Configuration Files

The preliminaries for the original deductive model checking procedure is a transition system $\mathcal{S}$ and the tableau graph $\Phi_{\neg\varphi}$ for the specification φ we want to check. The model checking procedure presented in this thesis needs a phase event automaton that comprises both the behaviour of the transition system $\mathcal{S}$ and the structure of $\Phi_{\neg\varphi}$ as input. Althought representing a phase event automaton, the configuration script specifies a kind of transition system. The difference to the transition constraint systems that has been constructed from PEAs in [Hoe06] is, that they combine the transition systems for different processes. In the following, the syntax of a configuration script is given:

The configuration file has to start with the keyword *pea*, followed by a colon. When the file has been read in, the tokenizer only accepts the following structure:

keyword: *arguments*

Every new keyword has to follow a line-break. The configuration script consists of nine keywords, appearing in the order given below: *pea:*, *init:*, *transitions:*, *nodes:*, *intvars:*, *realvars:*, *nlabels:*, *invariants:*, *edges:*. The configuration file also has to end with a line-break.[3] In listing 5.1 the configuration script for the Fischer's mutual exclusion problem is given.

Listing 5.1: Configuration file for Fischer's mutual exclusion

```
pea:
init:and{Pc1=0,and{Pc2=0,and{C1=0,and{C2=0,X=0}}}}
transitions:transitionarguments
nodes:(p0,true,false),(p1,true,true)
intvars:Pc1,Pc2,X,C1,C2
realvars:
nlabels:(p0,true),(p1,and{Pc1=3,Pc2=3})
invariants:invariantarguments
edges:(p0$true$p1),(p0$true$p0),(p1$true$p1)
```

Because the lines of the *transitionarguments* and *invariantarguments* are too long for the listing, they are given below. In listing 5.3, we see the replacement for *transitionarguments*. In listing 5.2, the formula *invariantarguments* is replaced with. (Note, that in the original configuration file, there is no newline between transitions.)

[3] Note, that the tokenizer gets a list of characters. Therefore every *keyword: arguments* list ends with a 'newline'.

5.1.1 Boolean Constraints - Init and Invariants

The arguments of *init* and *invariants* are always boolean constraints. Listing 5.2 shows an invariant for Fischer's mutual exclusion problem.

In the configuration file, boolean constraints F, G, H have the following structure:

```
F,G,H  ::= true        (*verum*)
         | false       (*falsum*)
         |  P          (*atomic formula*)
         | not{G}      (*negation*)
         | and{G,H}    (*conjunction*)
         | or{G,H}     (*disjunction*)
         | imp{G,H}    (*implication*)
```

An atomic formula P always has the following structure:

```
P  ::=   Exp1 = Exp2
       | Exp1 =/= Exp2
       | Exp1 >= Exp2
       | Exp1 > Exp2
       | Exp1 =< Exp2
       | Exp1 < Exp2

Exp1,Exp2 ::=  Var           (*variable*)
             | Const         (*constant*)
             | Exp1 + Exp2   (*add*)
             | Exp1 - Exp2   (*subtract*)
           ( | Exp1 * Exp2   (*multiply*) )
```

Multiplication binds stronger than addition and subtraction. The initial condition and the invariants always are restricted to be conjunctions or a single atomic formula.

Listing 5.2: Invariants in the configuration file

```
and{and{and{and{Pc1>=0,Pc1=<3},and{Pc2>=0,Pc2=<3}},
      and{X>=0,X=<2}},and{C1>=0,C2>=0}}
```

5.1.2 Transitions

In the configuration file, the argument of *transitions* has the structure given below:

$$trans_1, \ldots, trans_n,$$

where each $trans_i$ is a triple $(Num, Chvarlist, Constraint)$. *Num* denotes a positive integer starting with 1. *Chvarlist* is the list of variables that are changed by the transition and *Constraint* is a boolean constraint. Listing 5.3 shows the transitions of Fischer's algorithm.

Listing 5.3: Transitions in the configuration file

```
(1,[Pc1,C1],and{and{X=0,Pc1=0},and{Pc1'=1,C1'=0}}),
(2,[Pc2,C2],and{and{X=0,Pc2=0},and{Pc2'=1,C2'=0}}),
(3,[X,C1,Pc1],and{C1<2,and{Pc1=1,and{X'=1,and{C1'=0,Pc1'=2}}}}),
(4,[X,C2,Pc2],and{C2<2,and{Pc2=1,and{X'=2,and{C2'=0,Pc2'=2}}}}),
(5,[Pc1],and{Pc1=1,and{C1>=2,Pc1'=0}}),
(6,[Pc2],and{Pc2=1,and{C2>=2,Pc2'=0}}),
(7,[Pc1],and{Pc1=2,and{C1>3,and{X=1,Pc1'=3}}}),
(8,[Pc2],and{Pc2=2,and{C2>3,and{X=2,Pc2'=3}}}),
(9,[Pc1],and{Pc1=2,and{and{C1>3,X=/=1},Pc1'=0}}),
(10,[Pc2],and{Pc2=2,and{and{C2>3,X=/=2},Pc2'=0}}),
(11,[Pc1],and{Pc1=3,and{X'=0,Pc1'=0}}),
(12,[Pc2],and{Pc2=3,and{X'=0,Pc2'=0}}),
(13,[C1,C2],and{C1'=C1+1,C2'=C2+1})
```

5.1.3 Nodes and Nodelabels

Nodes are specified as triples $(Id, Initflag, Failflag)$, that are also separated by commas. Every node has an unique identifier Id that marks the different states of the phase event automaton. The $Initflag$ value marks, if a node is an initial node and the $Failflag$ value specifies, that a node is an error node (i. e. this node labels the failure condition; a counter-example computation is possible). Nodelabels are tuples $(Id, Constraint)$. For a node p_{Id}, the Id values of both the node and the nodelabel have to coincide. $Constraint$ is a boolean constraint. As already mentioned before, nodelabels always are conjunctions of predicates or a single atomic formula.

5.1.4 Variables

The variables in the configuration file specify the states, the data and clock values of a phase event automaton. Here, variables are strings that begin with a capital letter, followed by capital letters, lowercase letters or numbers. Variables also are devided into integer and real-valued variables. In listing 5.1 all variables are declared as integers. This separation into real and integer variables strengthens the constraint propagation. The constraint $x > 2 \wedge x < 3$ for example is satisfiable, if x is a real valued variable. This is not the case, if x is an integer. Therefore, whenever there is a nodel label that includes a constraint of the form $exp_1 < exp_2$($exp_1 > exp_2$) and all exp_i only contain integer variables, the constraint is replaced by $exp_1 =< exp_2 - 1$ ($exp_1 >= exp_2 + 1$).

5.1.5 Edges

The edges in the configuration script have a special form:
They are triples $(Id_1\$Edgelabel\$Id_2)$. For parsing reasons, the different members of the triple are separated by a \$ sign. Id_1 and Id_2 specify the origin and the goal states of an edge. $Edgelabel$ always is a list of transitions. Because the *initial falsification diagram* labels each edge with the entire set of transitions, all edgelabels are marked with *true* in the configuration file. We could also remove this placeholder and only work with tuples which specify the origin and the goal of an edge.

5.1.6 Example: Elevator Configuration Script

Below, the configuration script for the elevator example is presented.

Listing 5.4: Configuration file for the elevator example

```
pea:
init:and{Goal=Curr=Min,and{Min<Max,and{Dir=0,Pc=0}}}
transitions: transitionarguments
nodes:(p0,true,false),(p1,true,true),(p2,true,true)
intvars:Min,Curr,Max,Goal,Dir,Pc
realvars:
nlabels:(p0,Min=<Curr=<Max),(p1,Curr<Min),(p2,Curr>Max)
invariants:and{Min<Max,
                  and{Pc>=0,and{Pc=<2,and{Dir+1>=0,Dir=<1}}}}
edges:(p0$true$p0),(p1$true$p1),(p2$true$p2),(p0$true$p1),
   (p0$true$p2)
```

Listing 5.5: Transitions in the elevator configuration file

```
(1,[Goal,Pc],and{Pc=0,and{Curr<Goal'=<Max,Pc'=1}}),
(2,[Goal,Pc],and{Pc=0,and{Min=<Goal'<Curr,Pc'=1}}),
(3,[Dir,Pc],and{Pc=1,and{Pc'=2,and{Goal>Curr,Dir'=1}}}),
(4,[Dir,Pc],and{Pc=1,and{Pc'=2,and{Goal<Curr,Dir'+1=0}}}),
(5,[Curr],and{Pc=2,and{Curr=/=Goal,Curr'=Curr+Dir}}),
(6,[Pc],and{Pc=2,and{Pc'=0,Goal=Curr}})
```

5.2 Phase Event Automata Creation

This section briefly presents the creation of an initial phase event automaton from the configuration file. This is done in four steps. First the configuration file is read in by the module *object_creater*. The resulting data structure is a character list that is given next to the module *tokenizer*. The procedure *tokenize(Charlist, Tokenlist)* analyses the *Charlist* and converts it into a list of tokens. The third step exists more or less for security reasons. The procedure *parse* checks if the tokenlist is in right order. This is inevitable, because the order in which the single components of the PEA object are constructed is important. Additionally, the parser checks if all arguments for the fourth step are syntactically correct. Finally, the module *converter* constructs the object that represents the initial phase event automaton from the given tokenlist. Note, that this initial PEA object is different from the *initial falsification diagram*. Below, some more information about the initial PEA object is given.

5.2.1 The PEA Object

The procedure *convert_to_object* in the module *converter* first creates an object *pea*.(Remember, that this object rather describes a transition constraint system.) A PEA object has the following attributes:

- *nodelist*: A list of all actually existing states of the automaton.
- *edgelist*: A list of all actually existing edges of the automaton.
- *initv*: A boolean constraint that describes the initial condition of the phase event automaton.
- *vars*: A list of all actually existing variables of the automaton.
- *intvars*: A list of all actually existing integer variables of the automaton.
- *transitions*: The list of transitions.
- *initnodes*: A list of all actually existing states that are marked initial.
- *new_edges*: A list of all actually existing edges that have been recently created.
- *invariants*: A boolean constraint that describes some additional information to restrict node labels.

For efficiency reasons, the deductive model checking procedure presented in this thesis only refines this PEA object, i. e. no other PEA objects are created. The representation of an phase event automaton as an object also makes it possible to simply extend the PEA's features. Nodes, edges, variables and transitions are objects themselfs. Whereas variable objects are quite simply structured, i. e. variables only have procedures to set and get their values; node, edge and transition objects have some additional information. Their structure will be showed in the following.

5.2.2 The Node Object

A node object has the following attributes:

- *state*: Every node in the graph represents a state the automaton is in. Every node has a unique atom $id = p_i$.

- *label*: A node label always is a list with one element. This element is a boolean constraint. Nodelabels are only allowed to be single predicates or conjunctions. Predicates do not contain terms of the form $Exp1 \neq Exp2$ [4]. They are replaced by two new nodes with labels

  $Exp1 < Exp2$ and
  $Exp1 > Exp2$.

  All nodes also are disjoint concerning their labels.

- *init*: Is a boolean flag that specifies, if a node is marked initial. Additionally, the flag marks definitely reachable states.

- *visited*: Is a boolean flag that specifies, if a node already has been visited. This flag is used in a procedure that traverses the graph to remove nodes, that are unreachable from an initial node.

- *visited2*: Is a boolean flag that specifies, if a node already has been visited. This flag is used in a procedure that traverses the graph to calculate the minimal distance of the actual node to the next failure node.

- *failflag*: Is a boolean flag that specifies, if a node represents a failure state. These nodes become unreachable during the deductive model checking procedure.

- *selfs*: Is a boolean flag that specifies, if a node contains an selfedge.

- *tos*: Is a list of all actually existing nodes j, $j \neq i$, the node i contains edges to.

- *froms*: Is a list of all actually existing nodes j, $j \neq i$, the node i contains edges from.

- *fdist*: Is an integer that shows the actual distance of that node to the next failure node.

5.2.3 The Edge Object

An edge object has the following attributes:

- *from*: Is an atom p_i that specifies the origin state of the edge.

- *edgelabel*: Is a list of transition objects.

- *to*: Is an atom p_i that specifies the goal state of the edge.

- *mem*: Is a boolean flag that is initially true. If an edge becomes an edge whose goal node is marked initial, i. e. after source enlargement (5.6.4), the flag is set to *false* and and this edge will not be used for a split again.

- *presat*: Is a tuple list with elements $(trans_i, flag)$, where $trans_i$ specifies a transition the edge is labeled with and $flag$ specifies, if the test for a *precondition split* was successful for that edge concerning $trans_i$. Initially all these flags are true.

- *postsat*: Is a tuple list with elements $(trans_i, flag)$, where $trans_i$ specifies a transition the edge is labeled with and $flag$ specifies, if the test for a *postcondition split* was successful for that edge concerning $trans_i$. Initially all these flags are true.

[4] Node labels have to be convex

5.2.4 The Transition Object

A transition object has the following attributes:

- *id*: Is an atom to uniquely identify a transition.
- *label*: Is a list that always consists of two elements. The first element *chvar(Chlist)* is an atom with *Chlist* as the list of variables that are changed by the transition. The second element is the the transition constraint itself.[5]
- *enabled*: Is a boolean formula that results from applying *enabled* to the transition. Because the enabled formula for a transition never changes, it is calculated once to avoid unneccessary computation.

[5]Note that a transition constraint must not contain constraints of the form $y' > x \wedge y' < z$ because the implicit guard $x < z$ is missing

5.3 Initial Falsification Diagram

In this section, the construction of the *initial falsification diagram* is presented. In 4.2 we have already seen how it is constructed in the original deductive model checking procedure. Both methods ideally are equal. However, the algorithm given next contains some refinements concerning the node labels, i. e. all future and existing node labels are made convex and disjoint among each other. This may increase the size of the initial diagram relative to the falsification diagram of the original DMC procedure, but it also removes redundant nodes in the graph and therefore considerably reduces the state-space.

In the original procedure, for every inital node a new initial node is constructed. This is also improved now: Nodes which represent failure states can be marked initial too, and as soon as their label is conjoined with the initial condition, the new label should be '*false*'. Therefore, nodes with label *false* will not be created at all, instead of pruning them later during the satisfiability check. Additionally, we know that every failure-path contains a minimal failure-path. Therefore, for all nodes that are marked initial no incoming edges and for all nodes that are marked as failure states no outgoing edges are created because the removal of non-minimal error-paths preserves the correctness of the system.

An additional change pertains to the transitions. In the initial phase event automaton, a transition may be labeled with a boolean constraint that contains an implication or a disjunction. For structural reasons transition labels are only allowed to be conjunctions. Therefore, the are changed concerning this matter. In the following, I will first give an algorithmic description (pseudocode) of the initial falsification diagram construction. Then, every procedure that is used in the algorithm will be explained in detail. All procedures to construct the diagram belong to the module *init_false_diagram*.

5.3.1 Initial Falsification Diagram Construction

Algorithm 1 Initial falsification diagram construction

Input: Pea /* this is the initial PEA */
1: construct new initial nodes
2: change transition labels
3: *basic transformations 1, 3, 4*
4: change failure nodes
5: change remaining nodes
6: split with invariants
7: set failure distances
Output: modified PEA /* represents the initial falsification diagram */

5.3.2 Construct New Initial Nodes

In line 1 of the initial falsification diagram construction a new set of initial nodes is constructed. The procedure traverses the list of all actually existing nodes and for every node, that is marked initial, a set of new nodes is created.

In 2, first a new label $l = init \wedge label \wedge \neg error$ is created for each node N_i. *init* is the initial condition of the PEA, *label* is the actual label of a node and *error* is the disjunction of all existing failure states. Because we only allow node labes to be conjunctions, the new label has to be brought to disjunctive normal form. Now, for every disjunct a new node N_{i_j} is created. Additionally, for every new initial node N_{i_j}, new outgoing edges are created, that go to exactly the same nodes as those of

Algorithm 2 Construction of new initial nodes

Input: nodelist, error

```
1: for all nodes in nodelist do
2:   if node is marked initial then
3:     create new node label /* init ∧ label ∧ ¬ error */
4:     bring new label in disjunctive normal form
5:     create new initial nodes
6:     create new edges
7:     origin node is not marked initial anymore
8:   end if
9: end for
```

N_i. A self-loop $\langle N_i, N_i \rangle$ becomes an edge $\langle N_{i_j}, N_i \rangle$ in the new graph. All N_{i_j} have no incoming edges. Finally, the *init* flag of every N_i is set to *false*, i. e. they are not initial anymore.

The construction of new nodes, node labels and edges comprises some interesting details. Except for the edges in this procedure, all other edges and nodes will be constructed in the module *node_splitting*. The procedures included therein will be explained in 5.5.

5.3.3 Change Transition Labels

As already mentioned before, transition labels are only allowed to be conjunctions. But the transition labels of the initial pea also can contain implications, disjunctions or predicates with inequalities. Therefore, at this point of the model checking procedure, for every transition t_i a disjunctive normal form $t_{i_1} \vee \ldots \vee t_{i_k}$ is calculated. Also every transition that contains an inequality predicate $exp_i \neq exp_j$ is replaced by two transitions $exp_i \leq exp_j$ and $exp_i \geq exp_j$. If all variables in exp_i and exp_j are integers, $exp_i \neq exp_j$ is replaced by two transitions, one containing $exp_i \leq exp_j - 1$, the other $exp_i \geq exp_j + 1$. In the new set of transitions every t_i is replaced by all t_{i_j}. Finally, all edges of the initial falsification diagram are labeled with the entire set of transitions.

Example:

Below, the resulting transitions of the Fischer's mutual exclusion problem given in 5.3 are presented. Note, that the old transitions 9 and 10 are replaced by two new transitions each.

- $\tau_1 : x = 0 \wedge pc_1 = 0 \wedge pc_1' = 1 \wedge c_1' = 0$
- $\tau_2 : x = 0 \wedge pc_2 = 0 \wedge pc_2' = 1 \wedge c_2' = 0$
- $\tau_3 : c_1 < 2 \wedge pc_1 = 1 \wedge x' = 1 \wedge c_1' = 0 \wedge pc_1' = 2$
- $\tau_4 : c_2 < 2 \wedge pc_2 = 1 \wedge x' = 2 \wedge c_2' = 0 \wedge pc_2' = 2$
- $\tau_5 : pc_1 = 1 \wedge c_1 \geq 2 \wedge pc_1' = 0$
- $\tau_6 : pc_2 = 1 \wedge c_2 \geq 2 \wedge pc_2' = 0$
- $\tau_7 : pc_1 = 2 \wedge c_1 > 3 \wedge x = 1 \wedge pc_1' = 3$
- $\tau_8 : pc_2 = 2 \wedge c_2 > 3 \wedge x = 2 \wedge pc_2' = 3$

- $\tau_9 : pc_1 = 2 \wedge c_1 > 3 \wedge x < 1 \wedge pc_1' = 0$
- $\tau_{10} : pc_1 = 2 \wedge c_1 > 3 \wedge x > 1 \wedge pc_1' = 0$
- $\tau_{11} : pc_2 = 2 \wedge c_2 > 3 \wedge x < 2 \wedge pc_2' = 0$
- $\tau_{12} : pc_2 = 2 \wedge c_2 > 3 \wedge x > 2 \wedge pc_2' = 0$
- $\tau_{13} : pc_1 = 3 \wedge x' = 0 \wedge pc_1' = 0$
- $\tau_{14} : pc_2 = 3 \wedge x' = 0 \wedge pc_2' = 0$
- $\tau_{15} : c_1' = c_1 + 1 \wedge c_2' = c_2 + 1$

5.3.4 *Basic Transformations*

In line 3 of the initial falsification diagram construction some basic transformation rules are inserted. This is useful, because in the next procedures a lot of nodes are split and for every node a lot of edges are created. Therefore, all unnecessary information, i. e. all unsatisfiable transitions, nodes that are unreachable and edges with empty labels, are removed. The **Basic transformation rules** will be presented in 5.4.

5.3.5 Change Failure Nodes

Algorithm 3 describes how all nodes that mark a failure state are made disjoint from all remaining nodes of the diagram.

Algorithm 3 Changing the labels of failure nodes

Input: nodelist, init /* init is the initial condition of the PEA */
1: **repeat**
2: **if** node is marked as failure node, node is not an initial node **then**
3: create new node label /* $\neg\, init \wedge label$ */
4: create new nodes for the new label
5: delete the origin node
6: create new edges for the new nodes
7: **end if**
8: **until** nodelist is empty

5.3.6 Change Remaining Nodes

This procedure changes all nodes that are not marked initial or 'fail' (i. e. represent a failure state). The algorithm is the same as the one for changing failure nodes except, that the new label for a node with label l is $\neg\, init \wedge \neg\, error \wedge l$.

5.3.7 Split With Invariants

In line 6 of the algorithm, that constructs the initial falsification diagram, all nodes are split with the *invariant constraint* given in the configuration script. This is very useful, because the initial falsification diagram may contain nodes, that should never be satisfiable, but can not be directly removed. Only with some additional information, these nodes can be removed before beeing split and consequently adding a lot of useless information to the diagram. Note, that the splitting of a node (N, l) with

an invariant i [6] first removes unsatisfiable nodes and second gives a proof for the correctness of the additionally used information. Invariant constraints are propositions that will be proved. They are not taken for granted. Below, the algorithmic description of the procedure is presented.

Algorithm 4 All nodes are split with the invariants

Input: nodelist, invariant /* invariant is a conjunction of invariant predicates */
1: **for all** *atomic predicates* $p \in$ *invariant* **do**
2: **for all** *nodes with label* $l \in$ *nodelist* **do**
3: create new node with label: $l \wedge p$
4: create new nodes with label: $l \wedge \neg p$ /* may be several nodes */
5: create new edges for every new created node
6: *basic transformations 1, 3, 4*
7: **end for**
8: **end for**

5.3.8 Set Failure Distances

In line 7, the *fdist* value for every node in the initial falsification diagram is set. The *fdist* value of a node N gives the actual distance of N to the next node that is marked as failure state. The algorithm that calculates those distances is given in the module *functionlibrary*. It traverses the initial falsification diagram by a breadth first search, starting from the set of all nodes that are marked as failure state.

5.3.9 Examples

Figure 5.1 shows the initial falsification diagram of Fischer's mutual exclusion algorithm without invariant splitting. This phase event automaton has 13 nodes, where state p_4 is the error state and p_3 the initial state. The problem is, that the node labels of $p_9, p_{10}, p_{12}, p_{14}$ and p_{16} never should be satisfied, since they restrict one of their variables to be negative. By contrast, Figure 5.2 shows the result of splitting all nodes with the invariant constraint

$$pc_1 \geq 0 \wedge pc_1 \leq 3 \wedge pc_2 \geq 0 \wedge pc_2 \leq 3 \wedge x \geq 0 \wedge x \leq 2 \wedge c_1 \geq 0 \wedge c_2 \geq 0.$$

Figure 5.3 shows the initial falsification diagram of the elevator example.

[6] I. e. two new nodes $(N_1, l \wedge i)$ and $(N_2, l \wedge \neg i)$ may be constructed.

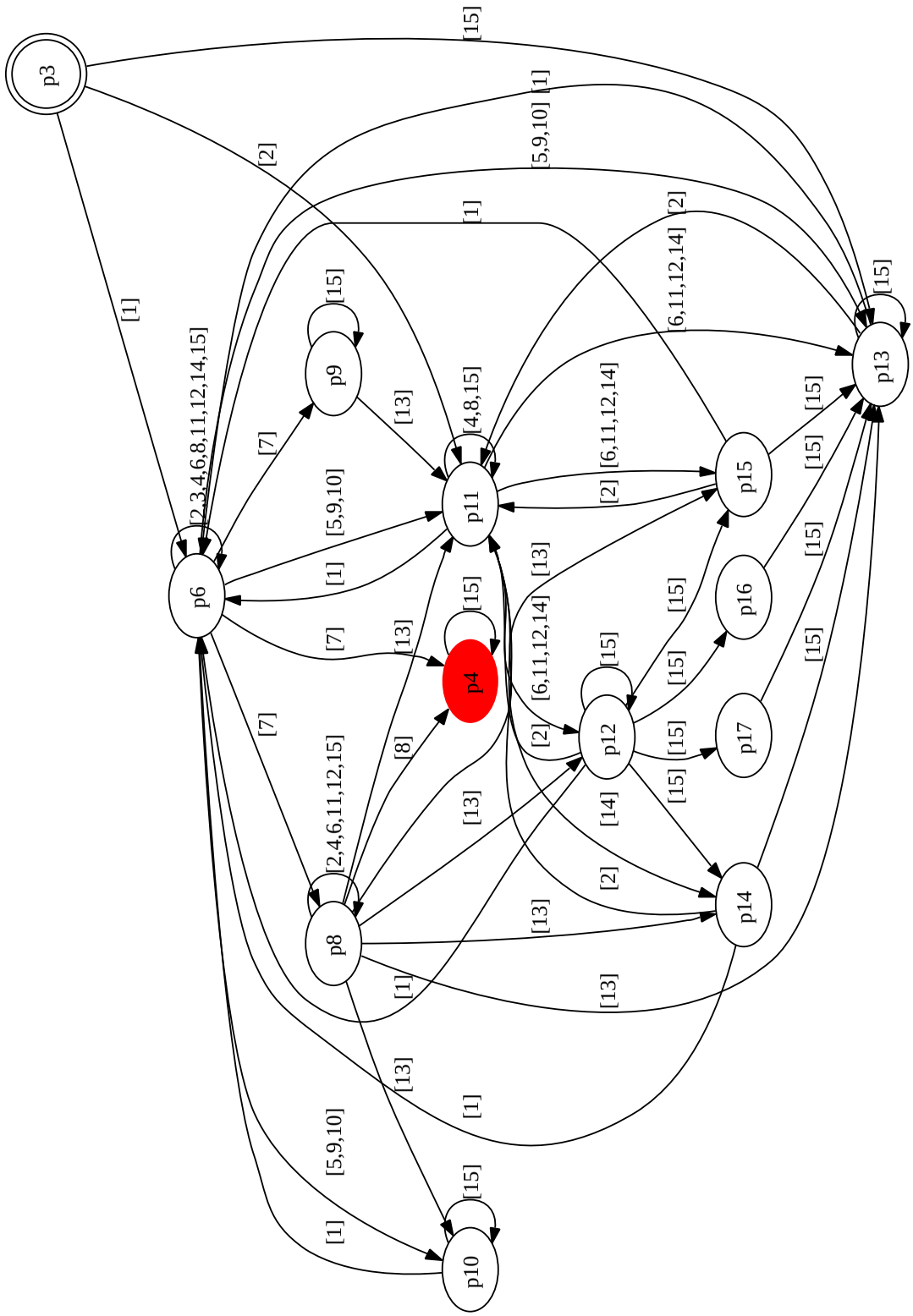

Figure 5.1: **Initial Falsification diagram without invariant splitting**

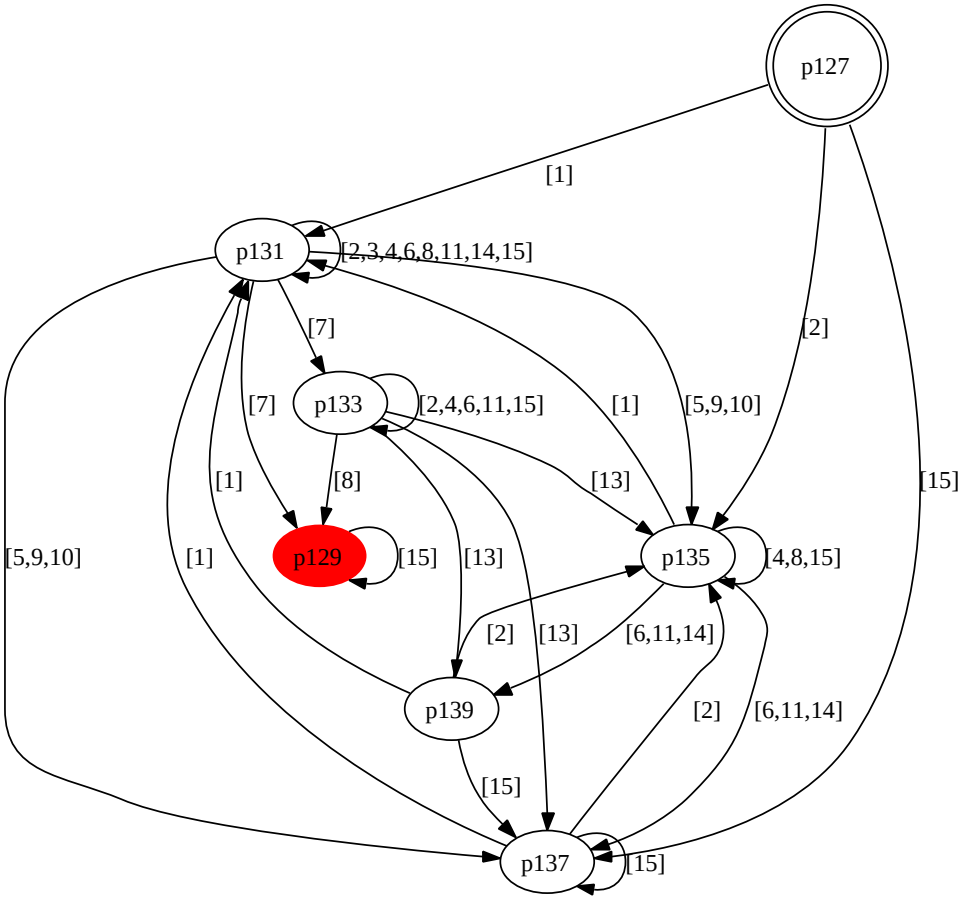

Figure 5.2: **Initial Falsification diagram with invariant splitting**

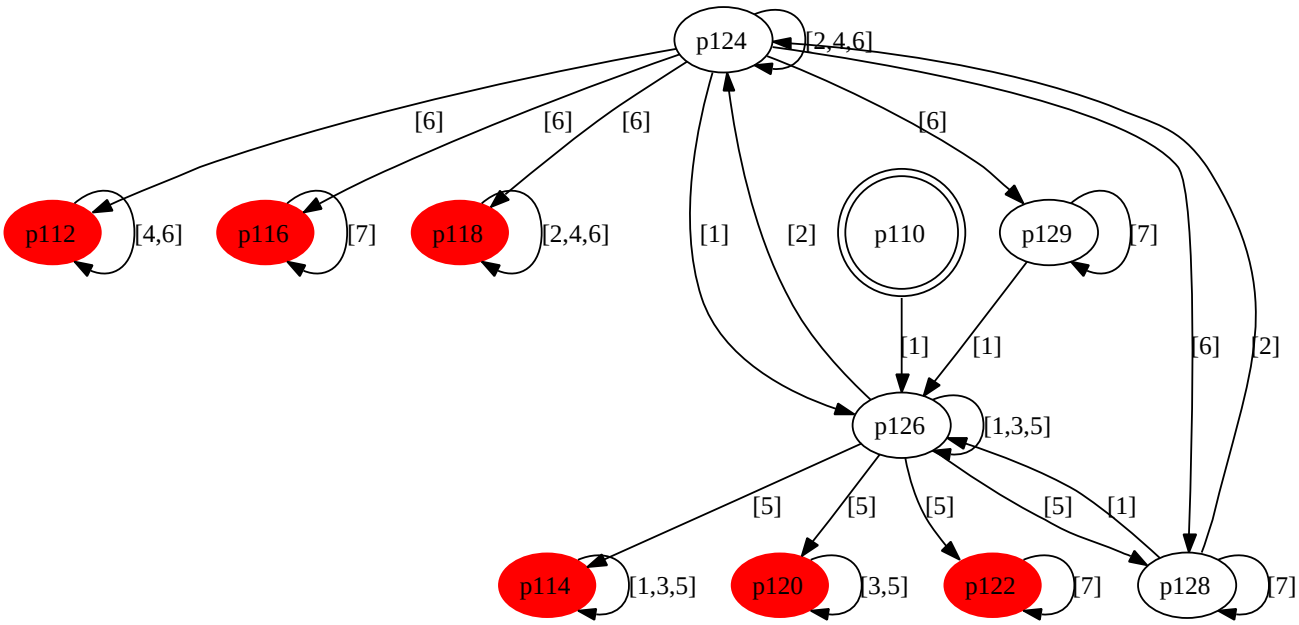

Figure 5.3: **Initial Falsification diagram for the elevator example**

5.4 Basic Transformations

This section presents the *basic transformations* of deductive model checking with transition constraint systems. Basic transformations are used to prune the phase event automaton, i. e. they are used to throw away unsatisfiable nodes, nodes, that are unreachable from an initial node, or nodes that have no successor. They are also used to remove transitions that are not applicable between two states or to throw edges with empty labels away.

In contrary to the basic transformations given in [HTZ96], the procedure presented in this thesis only uses those transformation rules that are sufficient to check the satisfiablity of temporal logic formulas. I. e., the transformation rules 5 and 6 of the original procedure have been dropped. In the original procedure it is also not really described, how to apply these transformations. Therefore, the transformation rules have a fixed order. First, all nodes, that are unreachable from an initial node, are removed. Then, all nodes, that have no successor, are thrown away. Next, all unsatisfiable edge labels between to nodes are removed. Now, every edge with an empty edge label is removed. Finally, the first transformation rule is applied again to remove every node (and its incoming and outgoing edges), that is not reachable anymore. The satisfiablity check is not necessary, because nodes whose label is not satisfiable are not created at all. This is a quite efficient change, because besides the creation of such a node also a lot of edges are created. Both the node and the edges would have to be removed during the next application of the basic transformation rules. In the following, all basic transformation rules are explained in detail.

5.4.1 Unreachable Node

The basic transformation rule *unreachable node* removes every state from the phase event automaton that is not reachable from an initial state. The underlying algorithm first computes the set of all initial nodes. From every initial node, a breadth first search traverses the graph. If there is an edge from an initial node N_0 to a child N_1, the childs attribute *visited* is set to *true*. This search is repeated until every traversed child is marked *visited*. Now, the list of all actually existing nodes is searched and every node that has not been visited yet is removed from the phase event automaton. As soon as an unreachable node is deleted, also all its incoming and outgoing edges have to be removed.

The easier this rule is to realize, the more crucial is its effect. Without removing unreachable nodes, the deductive model checking algorithm could refine a subgraph that never would belong to neither a correctness proof or a counterexample. This is not only a waste of computational power or time. The algorithm also could diverge in this subgraph, althought there may exist a correctness proof.

Example:

Figure 5.4 shows an unreachable subgraph of the elevator example after the first *precondition* and *postcondition split*. This is the result, if the *unreachable node* transformation rule is omitted. For simplicity reasons, all self-loops have been removed. The model checking procedure could never produce a correctness proof, since the graph contains failure states. If the transformation rule is not omitted, the model checking procedure will find a correctness proof after four *precondition splits*.

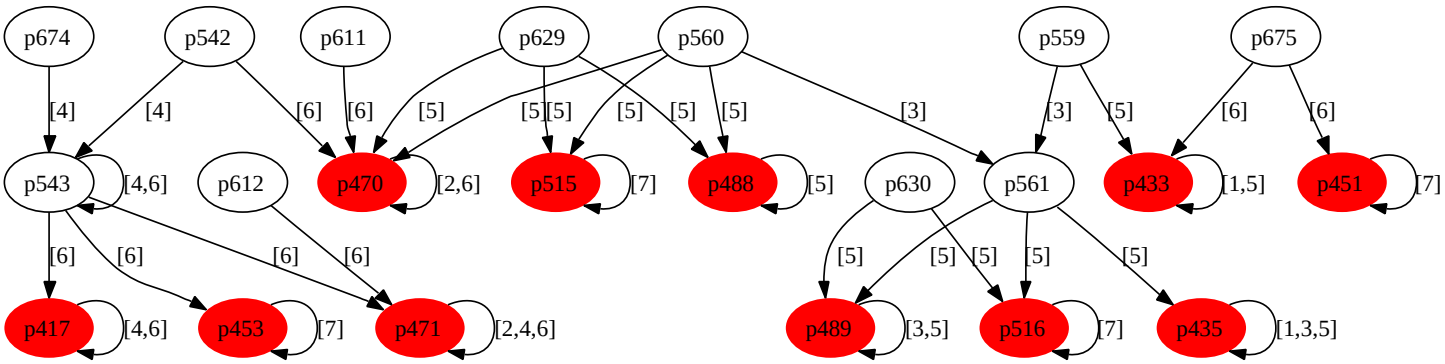

Figure 5.4: **Unreachable graph for the elevator example**

5.4.2 Unsatisfiable Node

The transformation rule *unsatisfiable node* is used to remove nodes that have no successors. As mentioned before, the satisfiabiliy check for node labels is omitted here.

In the algorithm presented here, the satisfiability check for node labels is advanced to the timepoint a node is created. If the label can not be satisfied, this node will not be created at all. This decision is really time and space saving. Assume a node is split into n new nodes in the precondition split for example, whereof the labels of the nodes $n_2, \ldots, n_n$ can not be satisfied. If the satisfiability check would have be postponed to the basic transformations part, the algorithm would have unnecessarily had to create $O(n^2)$ edges between those nodes, all incoming and outgoing edges to the nodes, all self-loops for the nodes and the nodes $n_2, \ldots, n_n$ themselves in the worst case. But it is not only the unnecessary creation and following deletion of the nodes, but also the unnecessary updating of distances and node specific information (a lot of lists have to be updated) that has to be done if unsatisfiable nodes are created.

The realisation of the *successor check* is quite simple. The input of this algorithm is the list of all actually existing nodes. For every node N is checked, if the attribute list *tos* of N is empty and the attribute flag *selfs* of N is set to *false*. If so, the node is removed, otherwise the input list is searched recursively. Again, if a node is removed all its incoming and outgoing edges have to be removed.

One design decision, that will be explained in detail later, is that, when it comes to a node split, failure nodes should not have any outgoing edges. The point here is, that this decision had to be reduced to outgoing edges that are not self-loops. If a failure state would not have a self-loop, the algorithm would incorrectly throw away all failure nodes in the first successor check.

5.4.3 Remove Edge Label

This section shows the functionality of the *remove edge label* transformation rule. If there is an edge $\langle (N_1, f_1), (N_2, f_2) \rangle$ that is labeled with an transition τ, the following constraint is checked:

$$f_1(\vec{x}) \wedge f_2(\vec{x}') \wedge p_\tau(\vec{x}, \vec{x}')$$

If this constraint is unsatisfiable, the transition τ has to be removed from the edge label.

The remove edge label rule is used to successively remove transitions from an edge label. After every *precondition* and *postcondition split*, transitions can be

removed from the edge labels of the new created edges. The goal is to remove all transitions until the label of an edge, that cannot belong to the set of possible edges for a counterexample, is empty. Then this edge can be removed by the *empty edge* transformation rule. Below, an algorithmic (pseudocode) description of the remove edgelabel rule is presented.

Algorithm 5 Remove edge label transformation rule

Input: new edges

1: **repeat**
2: prepare triple list TL for every edge
3: **for all** $(From, To, \tau) \in TL$ **do**
4: build *idle* constraint from τ
5: $p_\tau(\vec{x}, \vec{x}') := idle \wedge \tau$
6: **if** $f_{From}(\vec{x}) \wedge f_{To}(\vec{x}') \wedge p_\tau(\vec{x}, \vec{x}')$ is unsatisfiable **then**
7: remove τ from edge label
8: **end if**
9: **end for**
10: **until** edge list is empty

The algorithm's input is the set of *new edges*, i. e. the list of all edges created since the last *precondition* or *postcondition split*. Initially, all edges of the initial falsification diagram belong to that set. Note, that all other edges of the phase event automaton already have been checked for label removement. Next, for every edge in *new edges* a list of triples $(From, To, \tau)$ is constructed. Assume, an edge $\langle (N_1, f_1), (N_2, f_2) \rangle$ is labeled with transitions $\tau_1, \ldots, \tau_k$. The resulting triple list TL is $[(N_1, N_2, \tau_1), \ldots, (N_1, N_2, \tau_k)]$. Now, for every element of the list TL the assertion, that has to be checked for satisfiablility is built. Because all transitions τ_i, that are defined in the transition system, only assert the variables we want to change, we also have to build an *idle* constraint. The *idle* constraint is an assertion over unchanged variables $(\forall x_j \notin \mathit{changed\ vars} : x_j = x'_j)$. Finally the node labels f_1, f_2, the transition constraint $p_\tau(\vec{x}, \vec{x}')$ and the *idle* constraint for τ are conjoined and checked for satisfiability. If the constraint is satisfiable, the next element of TL is checked. Otherwise, the transition is removed from the edge label before checking the next element of TL.

The satisfiability check is done by the procedure *label_to_constr* in the module *own_constrain_function_library*. This procedure uses SICstus Prolog's built-in constraint library for rational and real numbers. Here, the first problem emerged. During the model checking algorithm really a lot of constraints have to be checked for satisfiability. Because most of these formulas are not constrained globally, we have to use for every local satisfiability test a new set of variable representatives. If one constraint restricts a variable a for example, the satisfiability test for another constraint that restricts a too, can not use the same variable, since a's restriction would have been already stored in the *constraint store* [7]. Therefore, every time a constraint is checked, a completely new set of variables is used. The result is, that the model checking algorithm filled Prolog's constraint store with millions of constraints and finally causes a memory error. The problem has been solved by using Prologs backtracking mechanism. Assume a procedure *constrain_f(Formula)* returns the value *true* iff the boolean formula *Formula* is satisfiable. The procedure *unsat* in 5.6 ensures, that every constraint is checked for satisfiability, but after having been checked, all constraints are removed from the constraint store again. If the constraint that has to be checked is true, the output *unsat* is *false* because the control flow passes a cut followed by *fail*. This causes the procedure to back-

[7] The constraint store stores the actual domains of variables.

track and throw away all collected information. The *cut* symbol ensures that the procedure does not try the second rule. So the result is *false*. Otherwise, the result is *true*.

Listing 5.6: Procedure that checks constraints for satisfiability

```
unsat(Formula, Vlist):-
        constrain_f(Formula, Vlist),!, fail.
unsat(_,_):-!.
```

Example:

Given for example two nodes (N_1, f_1) and (N_2, f_2) with labels

$f_1(\vec{x}) := c_1 = 0 \wedge pc_2 = 0 \wedge c_1 = 0 \wedge c_2 = 0 \wedge x = 0,$

$f_2(\vec{x}) := pc_1 = 3 \wedge pc_2 = 3 \wedge c_1 \geq 0 \wedge c_2 \geq 0 \wedge x \geq 0 \wedge x \leq 2$ and

transition τ_1 of Fischer's algorithm, i. e.

$\tau_1 := x = 0 \wedge pc_1 = 0 \wedge pc_1' = 1 \wedge c_1' = 0.$

First, the label of N_2 has to changed to a primed version. Next, the *idle* constraint for τ_1 is calculated. Because only pc_1 and c_1 are changed in the transition label, the *idle* constraint is:

$idle := pc_2 = pc_2' \wedge c_2 = c_2' \wedge x = x'.$

Finally, the constraint

$f_1(\vec{x}) \wedge f_2(\vec{x}') \wedge \tau_1 \wedge idle$

is checked for satisfiability. Because the conjunction $pc_2 = 0 \wedge pc_2' = 3 \wedge pc_2 = pc_2'$ is equivatent to *false*, this transition is removed from the edge label.

5.4.4 Empty Edge

The empty edge transformation rule removes all edges with empty labels. The underlying algorithm checks the set of new created edges, *new edges*, that also has been checked to remove transitions before. It suffices to check these edges, because all other edges already are removed or have not been changed since the last application of this rule. For every edge in *new edges*, the algorithm only has to test, if the label of that edge equals the empty list. If so, the edge is removed. When all edges with empty labels have been removed, the set *new edges* is set to the empty list.

5.5 Node Splitting

As already known, the basic idea of deductive model checking is to progressively refine the behaviour graph $(\mathcal{S}, \neg\varphi)$ of a given model $\mathcal{S}$ and a specification φ. When having constructed the initial falsification diagram, first, all unsatisfiable edge labels, i. e. transitions between two states, all nodes without successor states, all edges with empty labels and all nodes that are not reachable from an initial state are removed. If the resulting diagram does not contain any failure state, the deductive model checking procedure finishes. Additionally, we have obtained a correctness proof for $\mathcal{S} \models \varphi$.

But, if the resulting diagram still contains at least one failure state, we have to refine the diagram. The refinement is done by both *precondition* and *postcondition splits*. They will be explained later. The result of both kind of splits is that a node is replaced by new nodes with more restricted labels then the original one. This of course helps to remove impossible transitions and hopefully all paths to failure states.

This section describes how a node is split. In [HTZ96], a node N is always replaced by two new nodes N_1 and N_2. Here, the splitting is extended to nodes $N_1, \ldots, N_{k+1}$, because N_2 may have the form $N_{2_1} \vee N_{2_2} \vee \ldots \vee N_{2_k}$ and we do not allow node labels to be disjunctions. This has been done, because elimination of quantifiers for example needs a formula in disjunctive normal form. [8] The calculation of a disjunctive normal form is exponential and therefore unsuitable. In the following, we first see how new nodes are created. Then we see, how corresponding edges are updated. The underlying procedures are *create_split_nodes* to create new nodes and *split_node* to remove and create corresponding edges. Both procedures are located in the module *node_splitting*.

5.5.1 Create New Nodes

Before we see an algorithmic description of the *create_split_nodes* procedure, properties of node labels are listed:

1. A node label formula is always a conjunction of atomic predicates $p_1 \wedge p_2 \wedge \ldots \wedge p_n$
2. A node label is always satisfiable, i. e. unsatisfiable states are not created at all.
3. A node label never contains redundant predicates, i. e. a node could never contain both $x \geq 4$ and $x \geq 5$ as subformulas.
4. A node label never contains a negated predicate. Negations are put inward before creating a node.
5. A node label never contains predicates of the form $x \neq y$. Inequalities $x \neq y$ are replaced by $x < y \vee x > y$. Note that this change produces two new nodes.
6. A node label never contains predicates $x < y$ ($x > y$ resp.) if all variables in the predicate are integer variables. These predicates are replaced by $x \leq y - 1$ ($x \geq y + 1$ resp.).

Besides the *failflag* and *initflag* of the node that has to be replaced, the procedure *create_split_nodes* also needs a boolean formula in disjunctive normal form as argument. If this formula is a disjunction, the procedure is called recursively for every disjunct, i. e. every disjunct produces a new node. If the formula is a

[8] This will be explained when it comes to the calculation of the *strongest postcondition* in 5.6.3.

Algorithm 6 *create_split_nodes*$(l, initflag, failflag, newnodes)$

Input: node label l, *initflag*, *failflag*
Output: list of new created nodes *newnodes*

```
1: if l == a ∨ b then
2:    create_split_nodes(a, initflag, failflag, newnodes1)
3:    create_split_nodes(b, initflag, failflag, newnodes2)
4:    append(newnodes1, newnodes2, newnodes)
5: else if l = (l_0 ∧ p(x ≠ y)) then
6:    create_split_nodes(l_0 ∧ p(x < y), initflag, failflag, newnodes1)
7:    create_split_nodes(l_0 ∧ p(x > y), initflag, failflag, newnodes1)
8:    append(newnodes1, newnodes2, newnodes)
9: else
10:   remove redundant constraints from l
11:   replace all strict inequalities for predicates that only contain integers
12:   if l ≠ false then
13:      create new node N_i with label l
14:      N_i has the same failflag as original node /* except target enlargement */
15:      N_i has the same initflag as original node /* except source enlargement */
16:      newnodes = [N_i]
17:   end if
18: end if
```

single atomic predicate or a conjunction of atomic predicates, the procedure checks, if it contains a predicate with an inequality constraint $x \neq y$. Then, the formula is split again into two formulas containing $x < y$ and $x > y$. This split would also result in two new nodes. If the formula does not contain any predicate $x \neq y$, all redundant predicates are removed and strict inequalities replaced. Now, the formula is checked for satisfiability again. If it is satisfiable, a new node is created. The new node's *failflag* is the same as the original one, except in the case of *target enlargement* (5.6.4). The new node's *initflag* is the same as the original one, except in the case of *source enlargement* (5.6.4). The procedure output is the list of all from the formula created nodes. This list will be used by the procedure *split_node* to create new edges for the new nodes.

5.5.2 Create Edges

A soon as new nodes $N_1, \ldots, N_k$ have been created, the edges of the phase event automaton have to be updated. This is done by the procedure *split_node*. The procedure's input is the list of new created nodes and the original node that has to be removed. First, the procedure handles self-loops, i. e. if the origin node N_{old} had a self-loop $\langle N_{old}, N_{old}\rangle$, this edge is replaced by self-loops $\langle N_1, N_1\rangle, \ldots, \langle N_k, N_k\rangle$. Additionally, the procedure creates edges between all new created nodes, i. e. for all $i, j \in \{1, \ldots, k\}$ with $i \neq j$, edges $\langle N_i, N_j\rangle$ are created. All these edges have the same edge label es the original self-loop.

Then all incoming and outgoing edges (except the self-loop) are updated. Any incoming edge $\langle N_0, N_{old}\rangle$ is replaced by new edges $\langle N_0, N_1\rangle, \ldots, \langle N_0, N_k\rangle$ with the same label as $\langle N_0, N_{old}\rangle$. Then all outgoing edges $\langle N_{old}, N_{out}\rangle$ are replaced by edges $\langle N_1, N_{out}\rangle, \ldots, \langle N_k, N_{out}\rangle$. Again, they have the same label as $\langle N_{old}, N_{out}\rangle$.

Note that edges from failure nodes will not be created (except self-loops). Target enlargement is responsible for the existence of edges from failure nodes. These edges will be removed after the precondition split.

5.6 Basic Refinement Transformations

This section presents the *refinement transformations* of deductive model checking with transition constraint systems, **precondition split** and **postcondition split**.

As soon as the initial falsification diagram has been constructed, the basic transformations given in 5.4 are used to remove unsatisfiable edge labels, empty edges, unreachable states or states with no successors. Now, the procedure will test, if there is still a possible path from an initial state to a failure state. This test is really simple, because all unreachable nodes already have been removed. We only have to check, if the automaton contains a failure state. If there is no '*error trace*' [9], the procedure stops. Additionally, it has produced a correctness proof. Otherwise, one of the basic refinement transformations is applied.

Basic refinement transformations are used to strengthen the domains of variables at a given state and maybe rule out impossible paths in the diagram. I. e., whenever a node is split into new nodes, the domain of their variables is strengthened concering a given transition. Additionally, the labels of the updated edges are strengthened too. The goal is to reach an automaton configuration where all state domains are restricted that much, that no transition can be applied to reach a failure state.

Below, the complete algorithm for deductive model checking with transition constraint systems is given:

Algorithm 7 Complete algorithm for deductive model checking

Input: initial PEA representing $\Phi_{\neg\varphi}$

```
1: repeat
2:    test break_condition /* break_condition is a flag that is true if no failure state
      is reachable from an initial state */
3:    if break_condition ≠ true then
4:       precondition split
5:       basic transformations
6:       test break_condition
7:       if break_condition ≠ true then
8:          postcondition split
9:          basic transformations
10:         set break_condition to false
11:      else
12:         set break_condition to true
13:      end if
14:   else
15:      set break_condition to true
16:   end if
17: until break_condition = true or no new nodes have been created
```

The algorithm stops if one of the following two conditions is fullfilled:

1. There is no path from an initial state to a failure state.

2. Neither the precondition nor the postcondition split could have been applied.

In the second case, we obtain a counterexample computation, because these break conditions are exclusive and therefore we will have an error trace. Note, that if the underlying transition systems $\mathcal{S}$ is not a model for φ, the model checking

[9] An error trace is finite run of a phase event automaton, that starts with an initial state and ends with an failure state.

algorithm may diverge, because there may always be an edge that can be split. In the following, the procedures that create the *enabled*, *weakest precondition* and *strongest postcondition* formulas will be presented first. Then, the precondition and postcondition split are explained in detail.

5.6.1 Construction of the *enabled* formula

A transition is called *enabled* if it can be taken at a given state. The following formula characterizes such states:

$$enabled(\tau) \stackrel{\text{def}}{=} \exists \vec{x}'.p_\tau(\vec{x}, \vec{x}').$$

Informally, the *enabled* formula of a transition τ is always the *guard* of that transition τ, because the conjunction of the unprimed part of a transition and the label of a given state have to be satisfiable. Atomic predicates, that contain primed variables can be thrown away. They are always satisfiable if the transition does not contain any contradictions. Below, we first see a simple example. Then, the description of the algorithm is given.

Example:

Given for example the following transition:

$$\varphi_\tau \equiv (x > 1 \wedge y' = z)$$

The *enabled* formula of τ, is

$$\exists y' \varphi_\tau \equiv x > 1.$$

The formula *enabled*(τ) is calculated in the module *enabled*. Because transitions are always conjunctions or single atomic predicates, the calculation of *enabled*(τ) is straight forward. Traverse the boolean formula given in the transition label and throw away all predicates that contain primed variables. If the transition label has no *guard*, the resulting formula is equivalent to *true*.

To avoid code duplication, the procedure, that computes the *enabled* formula, is also used to compute the *strongest postcondition* and the *weakest precondition*. Therefore, the procedure *enabled* also does unification and variable replacement.

The arguments of the procedure enabled are a transition τ and the *idle* constraint constructed from τ. Because the transition argument can also be the conjunction of a transition and a node label, the procedure first simplifies the conjunction of the arguments, i. e. throws for example redundant information away. Finally, the existential quantifiers are eliminated. For more information see the following two sections.

Example:

Below, the *enabled* formulas for the transitions of the Fischer's mutual exclusion algorithm given in 5.3.3 are presented.

- $enabled(\tau_1) \equiv x = 0 \wedge pc_1 = 0$
- $enabled(\tau_2) \equiv x = 0 \wedge pc_2 = 0$
- $enabled(\tau_3) \equiv c_1 < 2 \wedge pc_1 = 1$
- $enabled(\tau_4) \equiv c_2 < 2 \wedge pc_2 = 1$
- $enabled(\tau_5) \equiv pc_1 = 1 \wedge c_1 \geq 2$
- $enabled(\tau_6) \equiv pc_2 = 1 \wedge c_2 \geq 2$
- $enabled(\tau_7) \equiv pc_1 = 2 \wedge c_1 > 3 \wedge x = 1$
- $enabled(\tau_8) \equiv pc_2 = 2 \wedge c_2 > 3 \wedge x = 2$
- $enabled(\tau_9) \equiv pc_1 = 2 \wedge c_1 > 3 \wedge x < 1$
- $enabled(\tau_{10}) \equiv pc_1 = 2 \wedge c_1 > 3 \wedge x > 1$
- $enabled(\tau_{11}) \equiv pc_2 = 2 \wedge c_2 > 3 \wedge x < 2$
- $enabled(\tau_{12}) \equiv pc_2 = 2 \wedge c_2 > 3 \wedge x > 2$
- $enabled(\tau_{13}) \equiv pc_1 = 3$
- $enabled(\tau_{14}) \equiv pc_2 = 3$
- $enabled(\tau_{15}) \equiv true$

5.6.2 Construction of the strongest postcondition formula

The *strongest postcondition* $post(\tau, \phi)$ of a formula ϕ relative to a transition τ is defined as follows:

$$post(\tau,\phi) \stackrel{\text{def}}{=} \exists \vec{x_0}.(p_\tau(\vec{x_0}, \vec{x}) \wedge \phi(\vec{x_0}))$$

The formulas $post(\tau,\phi)$ and $\neg post(\tau,\phi)$, that are used for the *postcondition split*, are computed by the procedures *post* and *npost* in the module *strongestpostcond*. Because the strongest postcondition formula is existentially quantified, the procedure *post* just uses the procedure *enabled*. Instead of just passing the transition and its *idle* constraint to the procedure, we additionally conjunct the transition argument with a node label formula ϕ. The whole formula now is first simplified and then, all quantifiers are eliminated (see 5.6.1).

Example:

Next, we see the resulting strongest postcondition formula in the first postcondition split of Fischer's mutual exclusion problem.[10] Given node label

$$\phi \equiv c_2 \geq 0 \wedge c_1 \geq 0 \wedge x \leq 2 \wedge x \geq 0 \wedge pc_2 \geq 0 \wedge pc_1 = 3 \wedge pc_2 \leq 2,$$

transition

$$\tau_8 \equiv pc_2 = 2 \wedge c_2 > 3 \wedge x = 2 \wedge pc_2' = 3$$

and the *idle* constraint constructed from τ_8

$$idle \equiv pc_1' = pc_1 \wedge c_1' = c_1 \wedge c_2' = c_2 \wedge x' = x.$$

The strongest postcondition $post(\tau_8,\phi)$ is

$$c_2' > 3 \wedge pc_2' = 3 \wedge c_1' \geq 0 \wedge pc_1' = 3 \wedge x' = 2$$

First, we know from the transition that $pc_2' = 3$. Now, the predicates $pc_1' = pc_1$ and $pc_1 = 3$ result in $pc_1' = 3$. Similary it is done with variable c_1. The predicates $x \leq 2 \wedge x \geq 0$ and $x' = 2 \wedge x' = x$ result in $x' = 2$ because redundant information is dropped. Similary it is done with variable c_2 again.

The construction of the negated strongest postcondition formula by the procedure *npost* not only computes $\neg\ post(\tau,\phi)$. It already conjuncts the result with the primed node label f_2. This will be explained in detail in section 5.6.6.

As already mentioned in 5.5, we only allow node labels to be conjunctions. This decision has been made, because the arguments of the procedure enabled, that are needed to compute the strongest postcondition formula for example, are a transition and a node label. Before the procedure can eliminate all quantifiers, the conjunction of these arguments has to be in disjunctive normal form. If the formula is not in disjunctive normal form, the procedure does maybe not know how variables have to be replaced. Given for example the following transition

$$\tau \equiv x = 0 \wedge x' = 1 \wedge y' = y \wedge z' = z$$

and a node label

$$(y = 0 \vee y = 3) \wedge z = 3.$$

The procedure does not know with what value it has to replace y for the calculation of the strongest postcondition formula for example. The result has to be a disjunction. But how can the procedure know that, without having build a disjunctive normal form before? To avoid this, node labels are always conjunctions.

5.6.3 Construction of the weakest precondition formula

The *weakest precondition* $wpc(\tau, \phi)$ of a formula ϕ relative to a transition τ is defined as follows:

$$wpc(\tau, \phi) \stackrel{\text{def}}{=} \forall \vec{x}'.(p_\tau(\vec{x}, \vec{x}') \rightarrow \phi(\vec{x}')).$$

[10] Note, that depending on the order edges are checked, the first split edge may have another label.

Both the formula $wpc(\tau, \phi)$ and $\neg\, wpc(\tau, \phi)$ are constructed in the module *weakestprecond*. The $wpc(\tau, \phi)$ is constructed by the procedure *extended_wpc*. The '*extended*' in the procedure name means that it not only computes the weakest precondition but also conjuncts the $wpc(\tau, \phi)$ with $enabled(\tau)$. This is because of structural reasons:

Given a transition relation $p_\tau \equiv \tau_1 \wedge \ldots \wedge \tau_k$ and a node label φ, the weakest precondition of φ relative to τ is

$$\begin{aligned} wpc(\tau, \phi) &\equiv \forall \vec{x}'.(p_\tau(\vec{x}, \vec{x}') \rightarrow \varphi(\vec{x}')) \\ &\equiv \forall \vec{x}'.\neg p_\tau(\vec{x}, \vec{x}') \vee \varphi(\vec{x}') \\ &\equiv \forall \vec{x}'.\neg(\tau_1) \vee \ldots \vee \neg(\tau_k) \vee \varphi(\vec{x}') \end{aligned}$$

If the formula above is conjoined with $enabled(\tau)$, all disjunctions except $\varphi(\vec{x}')$ can be dropped. The result is the conjunction of $enabled(\tau)$ and $\varphi(\vec{x}')$. Note, that all variables in $\varphi(\vec{x}')$ are primed and in the scope of an universal quantifier. This quantifier now is removed by unifying all primed variables in φ by the unprimed ones and replacing their values. Therefore, $extended\_wpc(\tau, \varphi)$ only computes $enabled(\tau)$ and conjuncts the result with the quantifier eliminated φ.

Example:

Given the node label

$$\varphi \equiv c_2 - c_1 \geq 1$$

and the transition (already conjuncted with *idle*)

$$\tau \equiv x = 0 \wedge pc_1 = 0 \wedge pc_1' = 1 \wedge c_1' = 0 \wedge pc_2' = pc_2 \wedge c_2' = c_2 \wedge x' = x.$$

We now get

$$enabled(\tau) \equiv x = 0 \wedge pc_1 = 0$$

and the quantifier eliminated

$$\varphi \equiv c_2 - 0 \geq 1.$$

Finally, the result of $extended\_wpc(\tau, \varphi)$ is

$$x = 0 \wedge pc_1 = 0 \wedge c_2 - 0 \geq 1.$$

The construction of the formula $\neg wpc(\tau, \phi)$ is done by the procedure *nwpc*. Because we want to avoid the computation of a disjunctive normal form, some structural details of $\neg wpc(\tau, \phi)$ are presented first. We know, that a node label $\varphi(\vec{x})$ always has the form $\varphi_1(\vec{x}) \wedge \ldots \wedge \varphi_k(\vec{x})$. Then,

$$\begin{aligned} & wpc(\tau, \phi) \equiv \forall \vec{x}'.(p_\tau(\vec{x}, \vec{x}') \rightarrow \varphi(\vec{x}')) \\ \Longrightarrow\ & \neg wpc(\tau, \phi) \equiv \exists \vec{x}'.(p_\tau(\vec{x}, \vec{x}') \wedge \neg\varphi(\vec{x}')) \\ \Longleftrightarrow\ & \neg wpc(\tau, \phi) \equiv \exists \vec{x}'.(p_\tau(\vec{x}, \vec{x}') \wedge (\neg\varphi_1(\vec{x}') \vee \ldots \vee \neg\varphi_k(\vec{x}')) \\ \Longleftrightarrow\ & \neg wpc(\tau, \phi) \equiv \underbrace{(\exists \vec{x}'.p_\tau(\vec{x}, \vec{x}') \wedge \neg\varphi_1(\vec{x}'))}_{L_1} \vee \ldots \vee \underbrace{(\exists \vec{x}'.p_\tau(\vec{x}, \vec{x}') \wedge \neg\varphi_k(\vec{x}'))}_{L_k} \end{aligned}$$

Algorithm 8 Calculation of $\neg wpc(\tau, \phi)$

Input: node label $\varphi = \varphi_1 \wedge \ldots \varphi_n$, transition $p_\tau(\vec{x}, \vec{x}')$
1: negate φ /* $\neg\varphi = L_1 \vee \ldots \vee L_k$ */
2: **for all** L_i **do**
3: apply *enabled* to $p_\tau(\vec{x}, \vec{x}') \wedge L_i$
4: **end for**

Because every $\varphi_i(\vec{x}')$ is an atomic predicate and every $p_\tau(\vec{x}, \vec{x}')$ is a conjunction, all L_i are in disjunctive normal form. So, the disjunction of all L_i is in disjunctive normal form, too. Note, that every L_i is an existential quantified formula. Therefore, we can use our procedure *enabled* again.

In line 1 of the algorithm, the node label φ is negated. The result is a disjunction $L_1 \vee \ldots \vee L_k$, where every $L_i \equiv \neg\varphi_i$. Because each of these L_i may be a potential new node, if it is satisfiable, we want the L_i to be disjoint too. Therefore, the negation of φ is strengthened to

$$\neg\varphi \equiv \underbrace{(\neg\varphi_1)}_{L_1} \vee \underbrace{(\varphi_1 \wedge \neg\varphi_2)}_{L_2} \vee \ldots \vee \underbrace{(\varphi_1 \wedge \ldots \wedge \varphi_{k-1} \wedge \neg\varphi_k)}_{L_k}.$$

Example:

Given the node label

$$\varphi \equiv c_2' \geq 0 \wedge c_1' \geq 0 \wedge x' \leq 2 \wedge x' \geq 0 \wedge pc_1' = 3 \wedge pc_2' = 3$$

and the transition

$$\tau \equiv pc_1 = 2 \wedge c_1 > 3 \wedge x = 1 \wedge pc_1' = 3 \wedge pc_2' = pc_2 \wedge x' = x \wedge c_1' = c_1 \wedge c_2' = c_2.$$

First, φ is negated, i. e., it is split into 8 exclusive disjuncts $\neg\varphi_1 \vee \ldots \vee \neg\varphi_8$.

- $\neg\varphi_1 \equiv pc_2' < 0$
- $\neg\varphi_2 \equiv c_2' \geq 0 \wedge c_1' < 0$
- $\neg\varphi_3 \equiv c_2' \geq 0 \wedge c_1' \geq 0 \wedge x' > 2$
- $\neg\varphi_4 \equiv c_2' \geq 0 \wedge c_1' \geq 0 \wedge x' \leq 2 \wedge x' < 0$
- $\neg\varphi_5 \equiv c_2' \geq 0 \wedge c_1' \geq 0 \wedge x' \leq 2 \wedge x' \geq 0 \wedge pc_1' < 3$
- $\neg\varphi_6 \equiv c_2' \geq 0 \wedge c_1' \geq 0 \wedge x' \leq 2 \wedge x' \geq 0 \wedge pc_1' > 3$
- $\neg\varphi_7 \equiv c_2' \geq 0 \wedge c_1' \geq 0 \wedge x' \leq 2 \wedge x' \geq 0 \wedge pc_1' = 3 \wedge pc_2' < 3$
- $\neg\varphi_8 \equiv c_2' \geq 0 \wedge c_1' \geq 0 \wedge x' \leq 2 \wedge x' \geq 0 \wedge pc_1' = 3 \wedge pc_2' > 3$

Now, all $\neg\varphi_i$ are conjoined with the transition label $p_\tau(\vec{x}, \vec{x}')$. The result are 8 existentially quantified formulas $L_1 \vee \ldots \vee L_8$. Finally, the procedure *enabled* is applied to every L_i to eliminate the quantifiers. The resulting weakest precondition formula is

$$(pc_1 = 2 \wedge c_1 > 3 \wedge x = 1 \wedge c_2 < 0) \vee$$

$$(pc_1 = 2 \wedge c_1 > 3 \wedge x = 1 \wedge c_2 \geq 0 \wedge c_1 \geq 0 \wedge x \leq 2 \wedge x \geq 0 \wedge pc_2 < 3) \vee$$

$$(pc_1 = 2 \wedge c_1 > 3 \wedge x = 1 \wedge c_2 \geq 0 \wedge c_1 \geq 0 \wedge x \leq 2 \wedge x \geq 0 \wedge pc_2 > 3)$$

All other L_i have been removed because they could not be satisfied.

5.6.4 Structural Decisions

In the following some structural decisions, that have been made to improve the model checking procedure are presented:

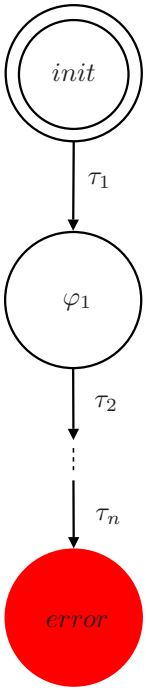

Figure 5.5: **Sample error trace**

- We never have to split self-loops. To point this intuitively out, assume the situation given in figure 5.5. For every loop-free path $\varphi_0 \xrightarrow{\tau_1} \varphi_1 \ \ldots \ \xrightarrow{\tau_n} \varphi_n$ (with φ_0 *initial*, φ_n *error*) we can have two situations:

  1. $\forall i : \varphi_{i-1} \equiv pre(\tau_i, \varphi_i) \wedge \varphi_i \equiv post(\tau_i, \varphi_{i-1})$
     $\rightsquigarrow$ We have a concrete failure-path. The failure state will be found with *target enlargement* (5.6.4).
  2. $\exists i : \varphi_{i-1} \not\equiv pre(\tau_i, \varphi_i) \vee \varphi_i \not\equiv post(\tau_i, \varphi_{i-1})$
     $\rightsquigarrow$ We can do a pre- or postcondition split.

  Thus, we do not need to split self-loops.

- Given an edge $e = \langle (N_1, f_1), (N_2, f_2) \rangle$ that is labeled with transition τ. It may happen, that the same edge will be checked several times. To avoid unneccessary computation, every edge has arguments *presat* and *postsat*. Before we test for example, if the precondition check $f_1 \wedge \neg(enabled(\tau) \wedge wpc(\tau, f_2))$ is satisfiable, we test, if the *presat* flag of e relative to τ is *true*. If so, the procedure continues. If not, e is tested with another transition it is labeled with, or, if e has no further transition in its label list, another edge is checked. During the precondition split procedure, the *presat* flag is set as follows: Assume, the precondition check $f_1 \wedge \neg(enabled(\tau) \wedge wpc(\tau, f_2))$ could not be satisfied. Then we set the *presat* value of this edge, relative to τ, to *false*. The same we do in the postcondition split with *postsat*.

- We never split outgoing edges from failure nodes. As soon as we have reached a failure state, all following states have to be failure states too. These states are not interesting, because we have already reached a failure state and thus, a counterexample computation still exists. The split of edges from failure nodes is already avoided, because we never split self-loops and failure states have no outgoing edges except a self-loop.

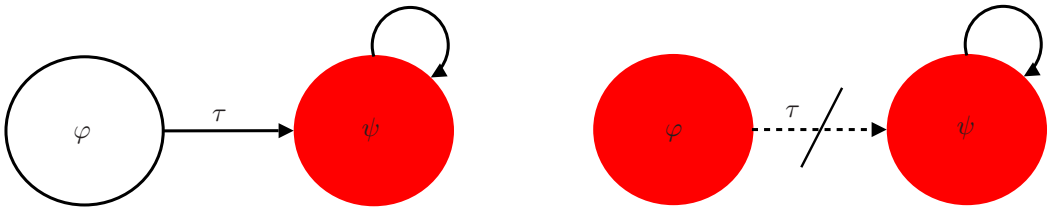

Figure 5.6: **Target Enlargement in Precondition Split**

- Assume, the situation on the left in figure 5.6. Given an edge $\langle (N_1, \varphi), (N_2, \psi) \rangle$ that is labeled with transition τ. Now, assume the precondition check $\varphi \wedge \neg (enabled(\tau) \wedge wpc(\tau, \psi))$ is not satisfiable. If so, we first know, that $\varphi \wedge \neg enabled(\tau)$ is unsatisfiable. Additionally, we know that all nodes that follow (N_1, φ) have to be failure states, because the precondition of a failure state relative to a transition models N_1. We know that $\varphi \wedge \neg wpc(\tau, \psi)$ is unsatisfiable [11]. Thus, $wpc(\tau, \psi)$ is a model for φ because:

  $$wpc(\tau, \psi) \models \varphi \Leftrightarrow \varphi \wedge \neg wpc(\tau, \psi) \text{ is unsatisfiable.}$$

  Therefore, we can throw edge $\langle (N_1, \varphi), (N_2, \psi) \rangle$ away. Finally, node (N_1, φ) is marked as a failure state. This method is called *target enlargement*[BKA02]. A special case of target enlargement is realized on the right in figure 5.6.

- *General target enlargement*: Assume the situation given on the top in figure 5.7. Additionally, we assume, that $\psi := wpc(error, \tau_1) \equiv \psi_1 \wedge \psi_2$. If φ will be split, we get three new nodes with labels $\varphi \wedge \psi$, $\varphi \wedge \neg\psi_1$ and $\varphi \wedge \psi_1 \wedge \neg\psi_2$. Because it will always be possible to reach the error node from the node with label $\varphi \wedge \psi$, this node's failure flag can be set to *true*. We can immediatly remove all outgoing edges of the node too. The result of the split we see at the bottom of figure 5.7.

[11] A disjunction is only unsatisfiable iff all its elements are unsatisfiable.

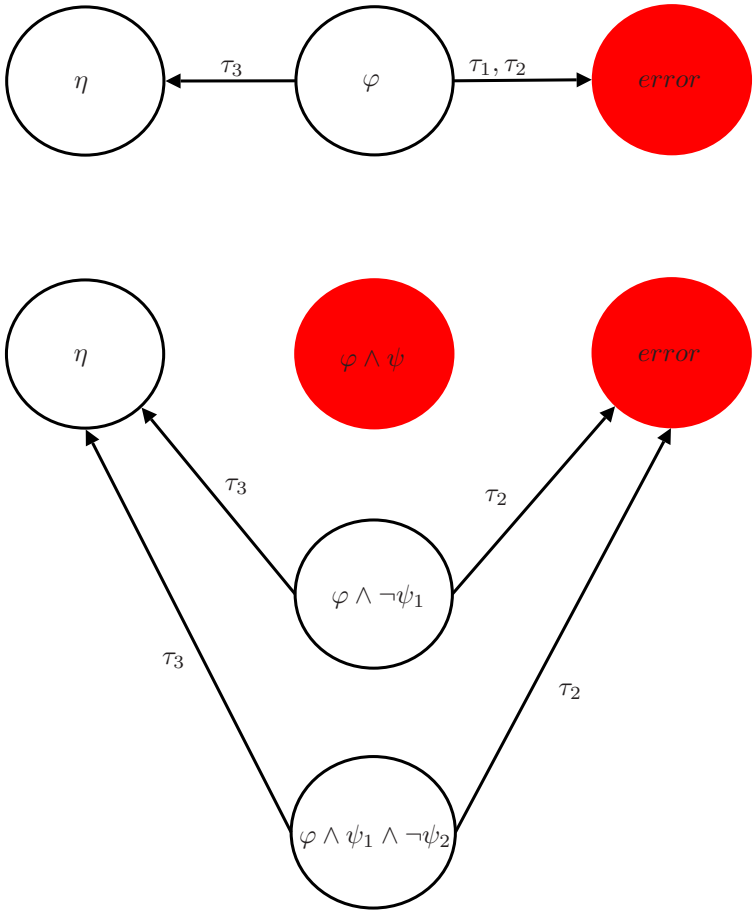

Figure 5.7: **General target enlargement**

- *Source enlargement*: Assume the situation given on the top in figure 5.8. Additionally, we assume, that $\psi := post(init, \tau_1) \equiv \psi_1 \wedge \psi_2$. If φ will be split, we get three new nodes with labels $\varphi \wedge \psi$, $\varphi \wedge \neg\psi_1$ and $\varphi \wedge \psi_1 \wedge \neg\psi_2$.

  Because it will always be possible to reach a concrete state that fullfills the constraint $\varphi \wedge \psi$, this node's init flag can be set to *true*. We can immediately remove all incoming edges of the node, too. The result of the split we see at the bottom of figure 5.8.

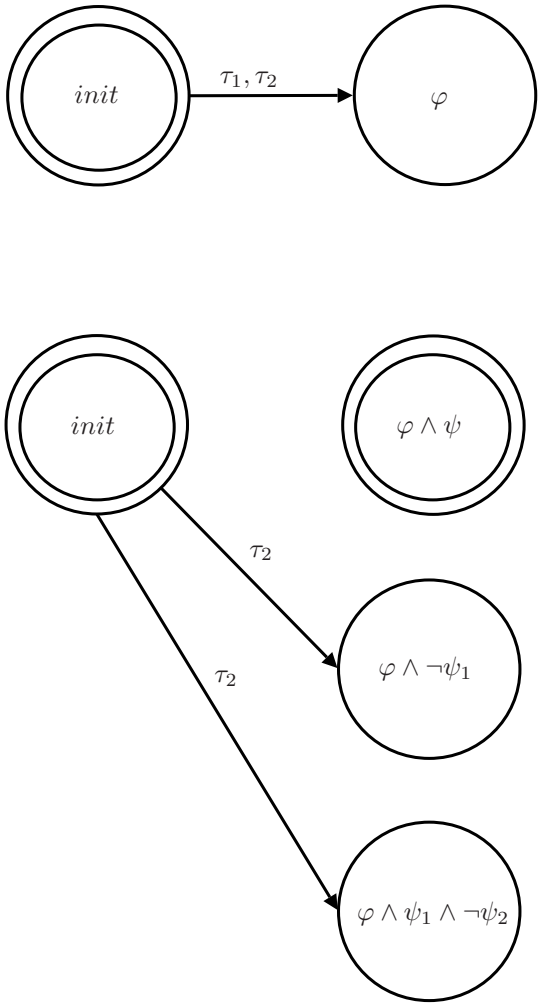

Figure 5.8: **Source enlargement**

- Assume, the situation on the left in figure 5.9. Given an edge $\langle (N_1, \varphi), (N_2, \psi) \rangle$ that is labeled with transition τ. We define $\eta \equiv \mathit{enabled}(\tau) \wedge \mathit{wpc}(\tau, \psi)$. Now, assume the precondition check $\varphi \wedge \neg(\mathit{enabled}(\tau) \wedge \mathit{wpc}(\tau, \psi))$ has been satisfiable. Therefore, we can split N_1 into two new nodes $(N_{1,1}, \varphi \wedge \eta)$ and $(N_{1,2}, \varphi \wedge \neg\eta)$. As already mentioned before, we never create unsatisfiable nodes. Assume, the node label $\varphi \wedge \eta$ is not satisfiable. Then, node $N_{1,1}$ will not be created. Additionally, we know that $\neg\,\eta$ is redundant, because:

  $$\varphi \wedge \eta \text{ is unsatisfiable} \Leftrightarrow \varphi \models \neg\,\eta \Leftrightarrow \varphi \Rightarrow \neg\,\eta \text{ is valid.}$$

  Therefore, we also do not create $N_{1,2}$, because all redundant information is dropped, before a node is created, and we do not copy nodes, because we want all nodes to be disjunctive. Since the result of a precondition split generally is the creation of $k + 1$ nodes, we have to test if at least two of those nodes are satisfiable.

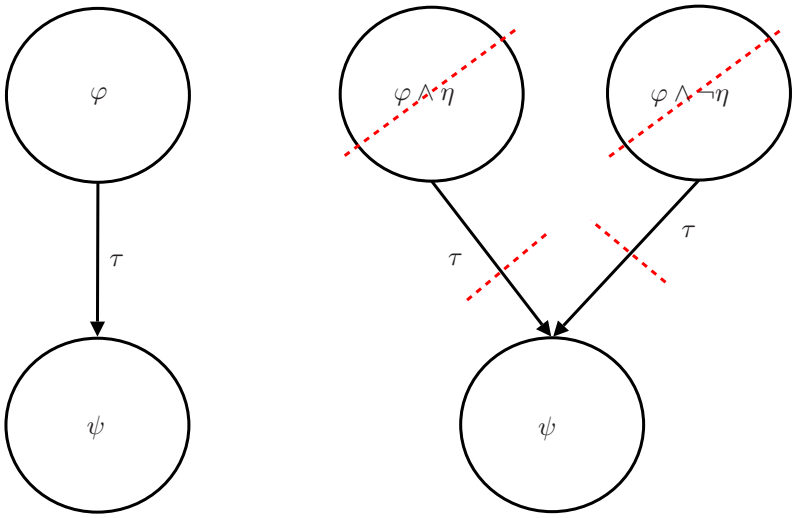

Figure 5.9: **Removing of redundant nodes**

- In the original deductive model checking algorithm we only know, that we have to split nodes if the basic transformations do not change the graph anymore. But there is no statement in which order edges have to be checked for a split. Since we want to throw away all failure states as fast as possible, four heuristics may be very helpful:
  - We prefer edges which have a failure state as goal node. A split on such an edge may throw away transitions and finally, if all transitions have been deleted from an edge label, the edge is removed. Additionally, we can do target enlargement.
  - We prefer edges $\langle N_1, N_2 \rangle$, if the shortest distance to a failure state of N_1 is greater then the shortest distance to a failure state of N_2 because it suffices to concentrate on minimal error-paths.
  - Self-edges are not considered at all. When edges are sorted, self-edges are thrown away.
  - We do not want to split with edges that go to an initial state. These edges are the result of source enlargement. Still, we do not throw them away, because they are interesting for the correctness proof. So, their *mem* value is set to *false* after a postcondition split.

5.6.5 Precondition Split

In section 5.6.3, 5.6.1, 4.2 we have already seen, how the *weakest precondition* and the *enabled* formulas are constructed and a node is split by the precondition split rule. Whereas in the original deductive model checking alogorithm, a node is always split into two new nodes, the model checking algorithm presented in this thesis splits a node into $k+1$ new nodes.

5.6.5.1 Definition: Precondition Split

Consider an edge $\langle (N_1, f_1), (N_2, f_2) \rangle$ with node labels f_1, f_2, that is labeled with transition τ. First, the following constraint is checked for satisfiability:

$$f_1 \wedge \neg(enabled(\tau) \wedge wpc(\tau, f_2)).$$

If this constraint is satisfiable, two formulas ϕ and ψ are constructed:

- $\phi \equiv f_1 \wedge enabled(\tau) \wedge wpc(\tau, f_2))$
- $\psi \equiv f_1 \wedge \neg(enabled(\tau) \wedge wpc(\tau, f_2))$

We already know, that $\neg(enabled(\tau) \wedge wpc(\tau, f_2))$ is a formula in disjunctive normal form. After having conjuncted every disjunction in $\neg(enabled(\tau) \wedge wpc(\tau, f_2))$ with f_1, we know that $\psi \equiv \psi_1 \vee \ldots \vee \psi_k$. Finally, node (N_1, f_1) is replaced by the new nodes $(N_{1,1}, \phi), (N_{1,2}, \psi_1), \ldots, (N_{1,k+1}, \psi_k)$.

5.6.5.2 Algorithm for Precondition Split

Below, the algorithm that realizes the precondition split and all structural decisions mentioned in 5.6.4 is presented:

Algorithm 9 Complete algorithm for precondition split

Input: *edge list* /* the list of all actually existing edges */
1: *sorted_edge_list* $\Leftarrow$ sort *edge list* relative to structural decisions (5.6.4)
2: **repeat**
3: prepare *triple_list* /* list of $(from, to, (transid, translabel))$ elements */
4: *break_condition* $\Leftarrow$ *pre_check*(*triple_list*)
5: **until** *break_condition* = *true* or *sorted_edge_list* is empty
6: remove outgoing edges of failure nodes /* constructed by target enlargement */

Algorithm 9 first sorts the list of all actually existing edges relative to the structural decisions mentioned in 5.6.4. A triple list *triple_list* with elements

$$(from, to, (transid, translabel))$$

is constructed from the first edge of the sorted edge list. This triple list contains as many elements as the edge label contains transitions. $from$ is the origin and to is the goal node of that edge. $transid$ and $translabel$ are the identifier and the label of an actual transition. Next, the procedure *pre_check*(*triple_list*) is called. The procedure's output is the flag *break_condition*. If this flag is set to *true*, the main algorithm stops, because the node $from$ has been split. Otherwise, the algorithm repeats with the next edge in the sorted edge list. If the sorted edge list is empty, the algorithm stops too. Target enlargement may set nodes which already have outgoing edges to failure nodes. All these edges are removed.

Algorithm 10 Algorithm that realizes the procedure *pre_check*

Input: *triple_list* /* list of $(from, to, (transid, translabel))$ elements */

```
1: repeat
2:   if triple_list ≠ [] then
3:     if presat(from, to, transid) = true then
4:       if σ ≡ from ∧ ¬(enabled(translabel) ∧ wpc(translabel, to)) is satisfiable
         then
5:         simplify σ /* σ my have redundant parts */
6:         if from ∧ enabled(translabel) ∧ wpc(translabel, to) is satisfiable then
7:           satnum ⇐ count satisfiable constraints of simplified σ
8:           if to is a failure node then
9:             general target enlargement
10:          end if
11:          if satnum ≥ 1 then
12:            split node from
13:            set break_condition to true
14:          else
15:            set break_condition to false
16:          end if
17:        else
18:          satnum ⇐ count satisfiable constraints of simplified σ
19:          if satnum ≥ 2 then
20:            split node from
21:            set break_condition to true
22:          else
23:            set break_condition to false
24:          end if
25:        end if
26:      else
27:        set presat(from, to, transid) = false
28:        if from is marked as failure node then
29:          set enlarge_flag to true /* generally, enlarge_flag is false */
30:        end if
31:        if enlarge_flag = true then
32:          delete edge ⟨from, to⟩
33:          set triple_list to empty list
34:          set break_condition to false
35:        end if
36:      end if
37:    end if
38:  else
39:    set break_condition to false
40:  end if
41: until triple_list is empty
```

Output: *break_condition*

Algorithm 10 realizes the procedure *pre_check*(*triple_list*). This procedure first tests, if the *triple list* is empty. If so, the procedure stops. The output *break_condition* is set to *false*.

Otherwise, the *presat* value of the edge relative to the transition with identifier *transid* is checked. If the value equals *false*, the procedure continues with the next triple.

If the value equals *true*, the precondition check

$$\sigma \equiv from \wedge \neg(enabled(translabel) \wedge wpc(translabel, to))$$

is tested for satisfiablility.

- If σ could have been satisfied, first, all redundant disjuncts of σ relative to the constraint

  $$\eta \equiv from \wedge enabled(translabel) \wedge wpc(translabel, to)$$

  are removed. σ may contain redundant constraints because the construction of the weakest precondition can eliminate disjunctive parts of constraints. Given for example the following disjunctive constraint:

  $$(x = 1 \wedge y' > 0) \vee (x = 1 \wedge y' < 0).$$

  The construction of the weakest precondition throws away the predicates $y' > 0$ and $y' < 0$ because they will be satisfiable. Therefore, the remaining disjunction is not disjoint anymore.
  Next, η is checked for satisfiability.

  - If it is satisfiable, we count all satisfiable disjuncts of σ. If the goal node of the edge is a failure node we do *general target enlargement* (5.6.4).
    - If at least one disjunct is satisfiable, the origin node $from$ is split and the *break_condition* flag is set to *true*.
    - If the number of satisfiable disjuncts, *satnum*, is smaller than 1, we do not split that node because the new created node and the old node to split are redundant. In this case, the *break_condition* flag is set to *false*.
  - If η is unsatisfiable, we check, if *satnum* is at least 2.
    - If so, $from$ is split and the *break_condition* flag is set to *true*.
    - Otherwise, the *break_condition* flag is set to *false*.

- If the precondition check σ is not satisfiable, the procedure checks, if the goal node is a failure node. If so, the origin node is marked as failure node and the *enlarge_flag* is set to *true*. This realizes the structural decision of *target enlargement* (5.6.4).

  Finally, the procedure checks, if the *enlarge_flag* equals *true*.

  - If not, the procedure continues with the next triple element.
  - Otherwise, edge $\langle from, to \rangle$ is deleted and the *break_condition* flag is set to *false*.

5.6.5.3 Example: Fischer's mutual exclusion problem

In figure 5.2 we have already seen the initial falsification diagram of Fischer's mutual exclusion problem. Figure 5.10 shows the phase event automaton after the first precondition split. The first edge of the sorted edge list is $\langle p131, p129 \rangle$. This edge is only labeled with transition τ_7. The label of node $p131$ is

$$\varphi \equiv\ c_2 \geq 0 \wedge c_1 \geq 0 \wedge x \leq 2 \wedge x \geq 0 \wedge pc_2 \leq 3 \wedge pc_2 \geq 0 \wedge pc_1 \geq 1 \wedge pc_1 \leq 2.$$

The label of node $p129$ is

$$\psi \equiv\ c_2' \geq 0 \wedge c_1' \geq 0 \wedge x' \leq 2 \wedge x' \geq 0 \wedge pc_1' = 3 \wedge pc_2' = 3.$$

Next, $\varphi \wedge (\neg \mathit{enabled}(\tau_7)) \vee (\neg \mathit{wpc}(\tau_7, \psi))$ is calculated. The result is the disjunction

$$\Psi = \Psi_1 \vee \Psi_2 \vee \Psi_3 \vee \Psi_4 \vee \Psi_5 \vee \Psi_6$$

with

$$\Psi_1 \equiv\ c_2 \geq 0 \wedge c_1 \geq 0 \wedge x \leq 2 \wedge x \geq 0 \wedge pc_2 \leq 3 \wedge pc_2 \geq 0 \wedge pc_1 \geq 1 \wedge pc_1 \leq 1 \wedge pc_1 \geq 3$$

$$\Psi_2 \equiv\ c_2 \geq 0 \wedge c_1 \geq 0 \wedge x \leq 2 \wedge x \geq 0 \wedge pc_2 \leq 3 \wedge pc_2 \geq 0 \wedge pc_1 \geq 1 \wedge pc_1 \leq 1 \wedge pc_1 \leq 1$$

$$\Psi_3 \equiv\ c_2 \geq 0 \wedge c_1 \geq 0 \wedge x \leq 2 \wedge x \geq 0 \wedge pc_2 \leq 3 \wedge pc_2 \geq 0 \wedge pc_1 \geq 1 \wedge pc_1 \leq 2 \wedge c_1 \leq 3$$

$$\Psi_4 \equiv\ c_2 \geq 0 \wedge c_1 \geq 0 \wedge x \leq 2 \wedge x \geq 0 \wedge pc_2 \leq 3 \wedge pc_2 \geq 0 \wedge pc_1 \geq 1 \wedge pc_1 \leq 2 \wedge x \geq 2$$

$$\Psi_5 \equiv\ c_2 \geq 0 \wedge c_1 \geq 0 \wedge x \leq 2 \wedge x \geq 0 \wedge pc_2 \leq 3 \wedge pc_2 \geq 0 \wedge pc_1 \geq 1 \wedge pc_1 \leq 2 \wedge x \leq 0$$

$$\Psi_6 \equiv\ c_2 \geq 0 \wedge pc_2 \geq 0 \wedge pc_1 = 2 \wedge c_1 \geq 4 \wedge x = 1 \wedge pc_2 \leq 2.$$

Because Ψ is satisfiable, $\varphi \wedge \mathit{enabled}(\tau_7) \wedge \mathit{wpc}(\tau_7, \psi)$ is calculated. The result is:

$$\Phi \equiv\ pc_1 = 2 \wedge c_1 \geq 4 \wedge x = 1 \wedge c_2 \geq 0 \wedge pc_2 = 3.$$

Because Φ is satisfiable too, for every Ψ_i and Φ, except Ψ_1, which is not satisfiable, a new node is created, i. e. we get new nodes

$$(p140, \Phi), (p141, \Psi_2), (p142, \Psi_3), (p143, \Psi_4), (p144, \Psi_5), (p145, \Psi_6).$$

Note, that node $(p140, \Phi)$ is marked as failure node because of target enlargement.

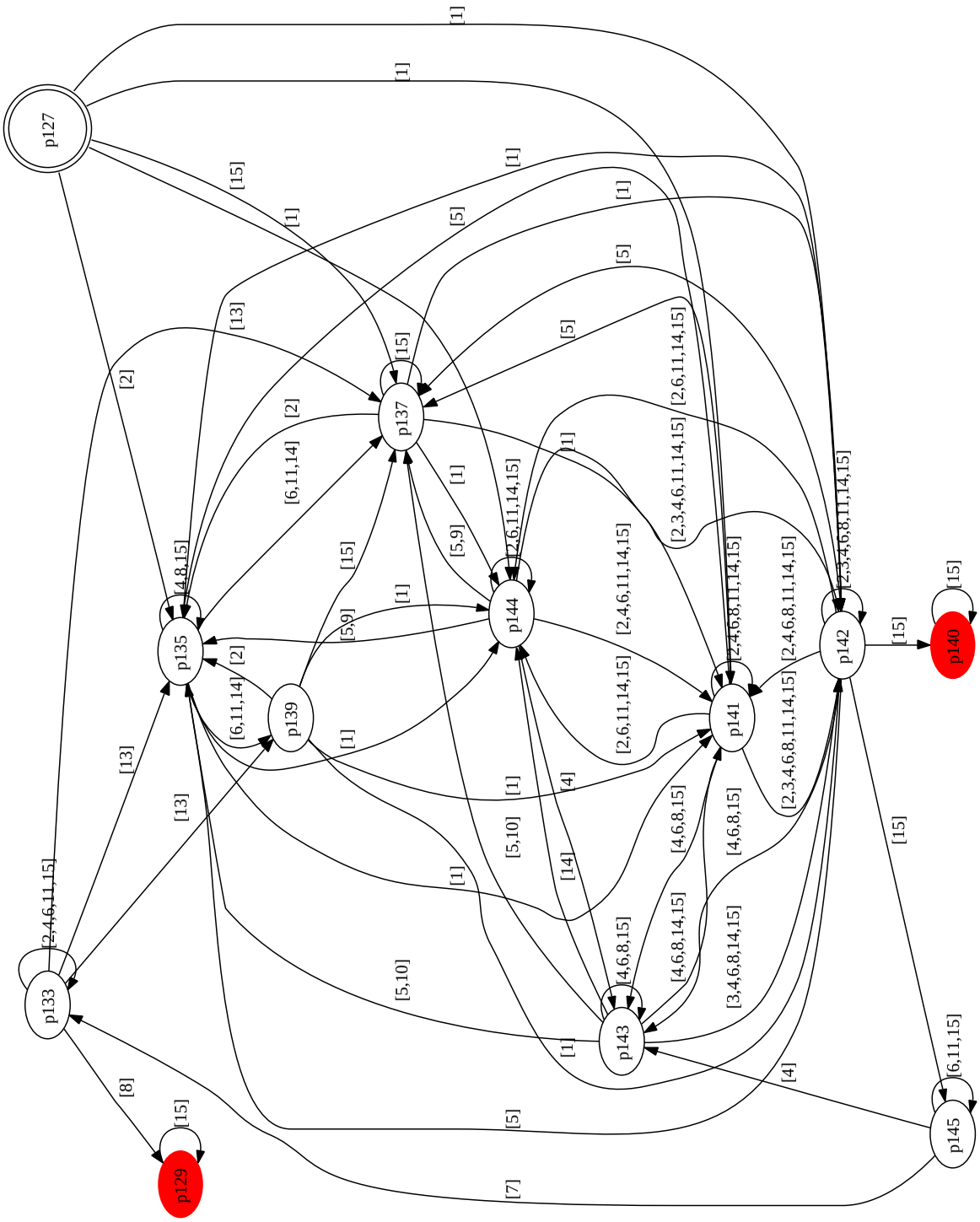

Figure 5.10: **After the first precondition split and basic transformations**

5.6.6 Postcondition Split

Consider an edge $\langle (N_1, f_1), (N_2, f_2) \rangle$ with node labels f_1, f_2, that is labeled with transition τ. First, the following constraint is checked for satisfiability:

$$f_2 \wedge \neg\, post(\tau, f_1).$$

If this constraint is satisfiable, two formulas ϕ and ψ are constructed:

- $\phi \equiv f_2 \wedge post(\tau, f_1)$
- $\psi \equiv f_2 \wedge \neg\, post(\tau, f_1)$

Again, ψ may have the form $\psi_1 \vee \ldots \vee \psi_k$. Therefore, node (N_1, f_1) is replaced by the new nodes $(N_{1,1}, \phi), (N_{1,2}, \psi_1), \ldots, (N_{1,k+1}, \psi_k)$.

5.6.6.1 Algorithm for Postcondition Split

Below, the algorithm that realizes the postcondition split and all structural decisions mentioned in 5.6.4 is presented:

Algorithm 11 Complete algorithm for postcondition split

Input: *edge list* /* the list of all actually existing edges */
1: *sorted_edge_list* ⇐ sort *edge list* relative to structural decisions (5.6.4)
2: **repeat**
3: prepare *triple_list* /* list of $(from, to, (transid, translabel))$ elements */
4: *break_condition* ⇐ *post_check*(*triple_list*)
5: **until** *break_condition* = *true* or *sorted_edge_list* is empty
6: change *mem* values of new created edges /* after source enlargement */

Algorithm 11 first sorts the list of all actually existing edges relative to the structural decisions mentioned in 5.6.4. Again, a triple list is constructed from the first edge of the sorted edge list. Next, the procedure *post_check*(*triple_list*) is called. The procedure's output is the flag *break_condition*. If this flag is *true*, the main algorithm stops, because the node *to* has been split. Otherwise, the algorithm repeats with the next edge in the sorted edge list. If the sorted edge list is empty, the algorithm stops too. Finally, the *mem* flag of all edges, that end in an initial node, are set to *false*.

Algorithm 12 realizes the procedure *post_check*(*triple_list*). This procedure first tests, if the *triple_list* is empty. If so, the procedure stops. The output *break_condition* is set to *false*. Next, the *postsat* value of the edge relative to the transition is checked. If the value equals *false*, the procedure continues with the next triple. If the value equals *true*, the postcondition check

$$\sigma \equiv to \wedge \neg post(translabel, from))$$

is tested for satisfiablility.

- If σ could have been satisfied, first, all redundant disjuncts of σ relative to the constraint $\eta \equiv to \wedge post(translabel, from)$ are removed. σ may contain redundant constraints because the construction of the strongest postcondition can eliminate disjunctive parts of constraints. Next, η is checked for satisfiability.
  - If it is satisfiable, we count all satisfiable disjuncts of σ. If at least one disjunct is satisfiable, we check, if the origin node of the edge is an initial node. If so, we do *source enlargement* 5.6.4. Then the goal node *to* is split

and the *break_condition* flag is set to *true*. If the number of satisfiable disjuncts, *satnum*, is smaller than 1, we do not split that node because the new created node and the old node to split are redundant. In this case, the *break condition* flag is set to *false*.

- If η is unsatisfiable, we check, if *satnum* is at least 2. If so, *to* is split and the *break_condition* flag is set to *true*. Otherwise, the *break condition* flag is set to *false*.

- If the postcondition check σ is not satisfiable, the postsat value is set to false and the procedure continues with the next triple element, because the *break_condition* flag is set to *false*.

Algorithm 12 Algorithm that realizes the procedure *post_check*

Input: *triple_list* /* list of $(from, to, (transid, translabel))$ elements */

repeat
  if *triple_list* $\neq$ [] **then**
    if $postsat(from, to, transid) = true$ **then**
      if $\sigma \equiv to \wedge \neg post(translabel, from)$ is satisfiable **then**
        simplify σ /* σ my have redundant parts */
        if $to \wedge post(translabel, from)$ is satisfiable **then**
          satnum $\Leftarrow$ count satisfiable constraints of simplified σ
          if $from$ is an initial node **then**
            source enlargement
          end if
          if *satnum* ≥ 1 **then**
            split node *to*
            set *break_condition* to *true*
          else
            set *break_condition* to *false*
          end if
        else
          satnum $\Leftarrow$ count satisfiable constraints of simplified σ
          if *satnum* ≥ 2 **then**
            split node *to*
            set *break_condition* to *true*
          else
            set *break_condition* to *false*
          end if
        end if
      else
        set $postsat(from, to, transid) = false$
        set *break_condition* to *false*
      end if
    end if
  else
    set *break_condition* to *false*
  end if
until *triple_list* is empty

Output: *break_condition*

5.6.6.2 Example: Fischers Mutual Exclusion

Figure 5.10 shows the phase event automaton after the first precondition split. The basic transformation rules have been applied too. Figure 5.11 shows the result of applying the first postcondition split. The first edge of the sorted edge list is $\langle p133, p129\rangle$. This edge is only labeled with transition τ_8. The label of node $p133$ is

$$\varphi \equiv c_2 \geq 0 \wedge c_1 \geq 0 \wedge x \leq 2 \wedge x \geq 0 \wedge pc_2 \leq 2 \wedge pc_2 \geq 0 \wedge pc_1 = 3.$$

The label of node $p129$ is

$$\psi \equiv c_2' \geq 0 \wedge c_1' \geq 0 \wedge x' \leq 2 \wedge x' \geq 0 \wedge pc_1' = 3 \wedge pc_2' = 3.$$

Next, $\psi \wedge \neg\, post(\tau_8, \varphi)$ is calculated. The result is the disjunction $\Psi = \Psi_1 \vee \Psi_2$ with

$$\Psi_1 \equiv c_2 \geq 0 \wedge c_1 \geq 0 \wedge x \leq 2 \wedge x \geq 0 \wedge pc_2 = 3 \wedge pc_1 = 3 \wedge c_2 \leq 3$$

$$\Psi_2 \equiv c_2 \geq 4 \wedge c_1 \geq 0 \wedge x \geq 0 \wedge pc_1 = 3 \wedge pc_2 = 3 \wedge 2 \geq x + 1$$

Because Ψ is satisfiable, $\psi \wedge post(\tau_8, \varphi)$ is calculated. The result is:

$$\Phi \equiv c_2 \geq 4 \wedge pc_2 = 3 \wedge c_1 \geq 0 \wedge pc_1 = 3 \wedge x = 2.$$

Finally, for every Ψ_i and Φ, a new node is created, i. e. we get new nodes

$$(p146, \Phi), (p147, \Psi_1), (p148, \Psi_2).$$

Note, that all new created nodes are marked as failure nodes. In the following basic transformations, transition τ_8 will be removed from both the edgelabel of edge $\langle p133, p148\rangle$ and $\langle p133, p147\rangle$. Thus, both node $(p148, \Psi_2)$ and node $(p147, \Psi_1)$ will be deleted because they are not reachable from an initial state anymore.

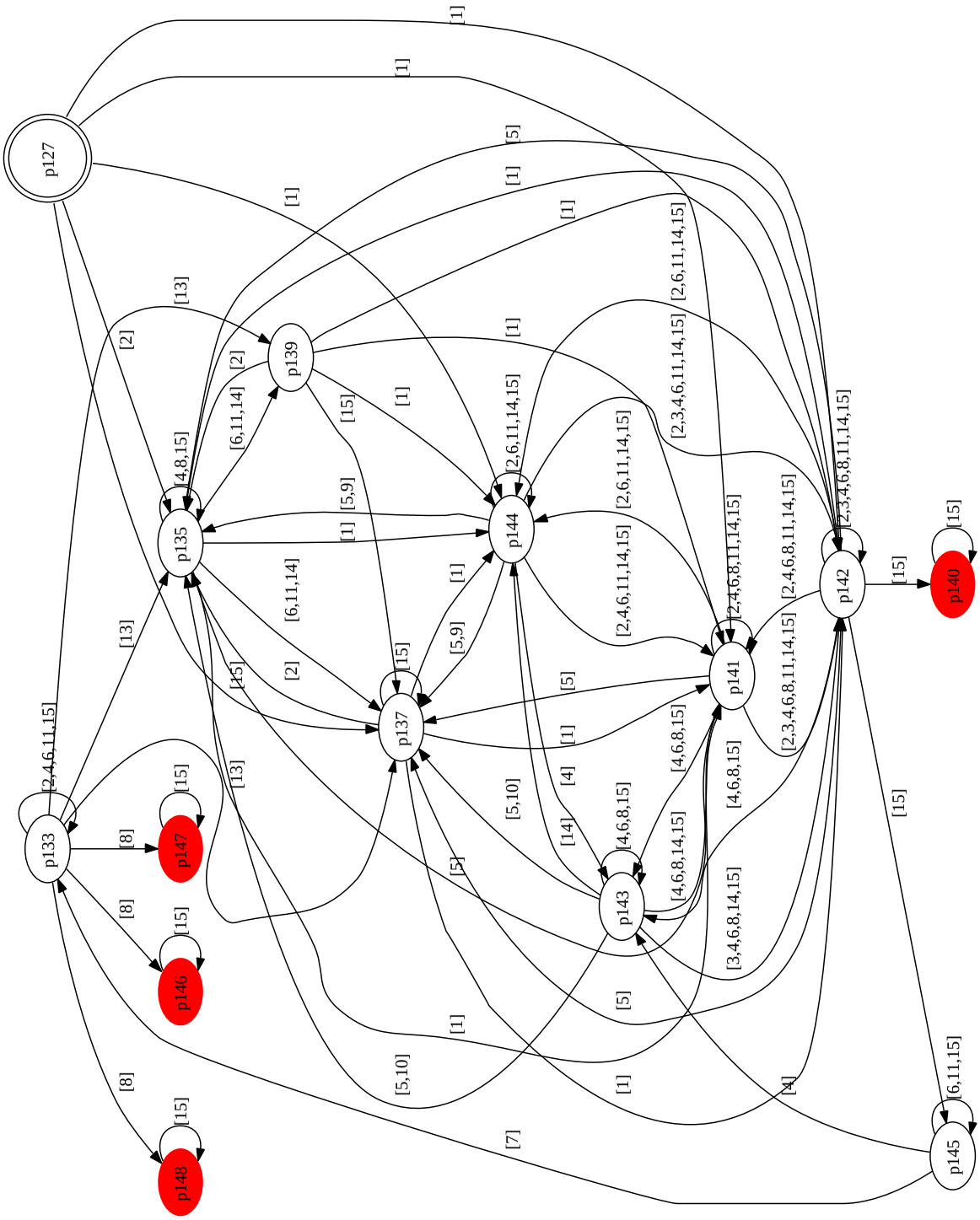

Figure 5.11: **After the first postcondition split**

Chapter 6

Analysis

In this chapter, first, the results of the two finite-state verification problems *Deque* and *Board4* are presented. Then, we see the results of the infinite-state *Bakery*, *Elevator* and *Fisher* problem.

6.1 Example: Deque

A *deque*, also called double-ended queue, is an abstract data structure where elements can be added to or removed from both the beginning and the end. The deque problem is a finite-state verification problem. Given the initial state $\Phi \equiv x_1 = 1 \wedge x_2 = 0 \wedge x_3 = 0 \wedge x_4 = 0$ and the following transitions:

- $\tau_1 \equiv x_4 + x_2 = 1 \wedge x_1' + x_1 = 1$
- $\tau_2 \equiv x_1 + x_3 = 1 \wedge x_2' + x_2 = 1$
- $\tau_3 \equiv x_2 + x_4 = 1 \wedge x_3' + x_3 = 1$
- $\tau_4 \equiv x_3 + x_1 = 1 \wedge x_4' + x_4 = 1$

The goal is to show, that starting from the initial state, we can never reach a state where all variables are set to 1. The transition system has an additional variable E that signalizes if we are at a failure state ($E = 1$) or not. Figure 6.1 shows the result of applying the model checking procedure to the deque problem. The resulting graph consists of 11 nodes, whereof some nodes can represent several configurations. State p_{167} for example restricts x_1 to be zero but all other variables are unrestricted. The resulting graph is a correctness proof because there is no path in the graph in which a failure state can be reached.

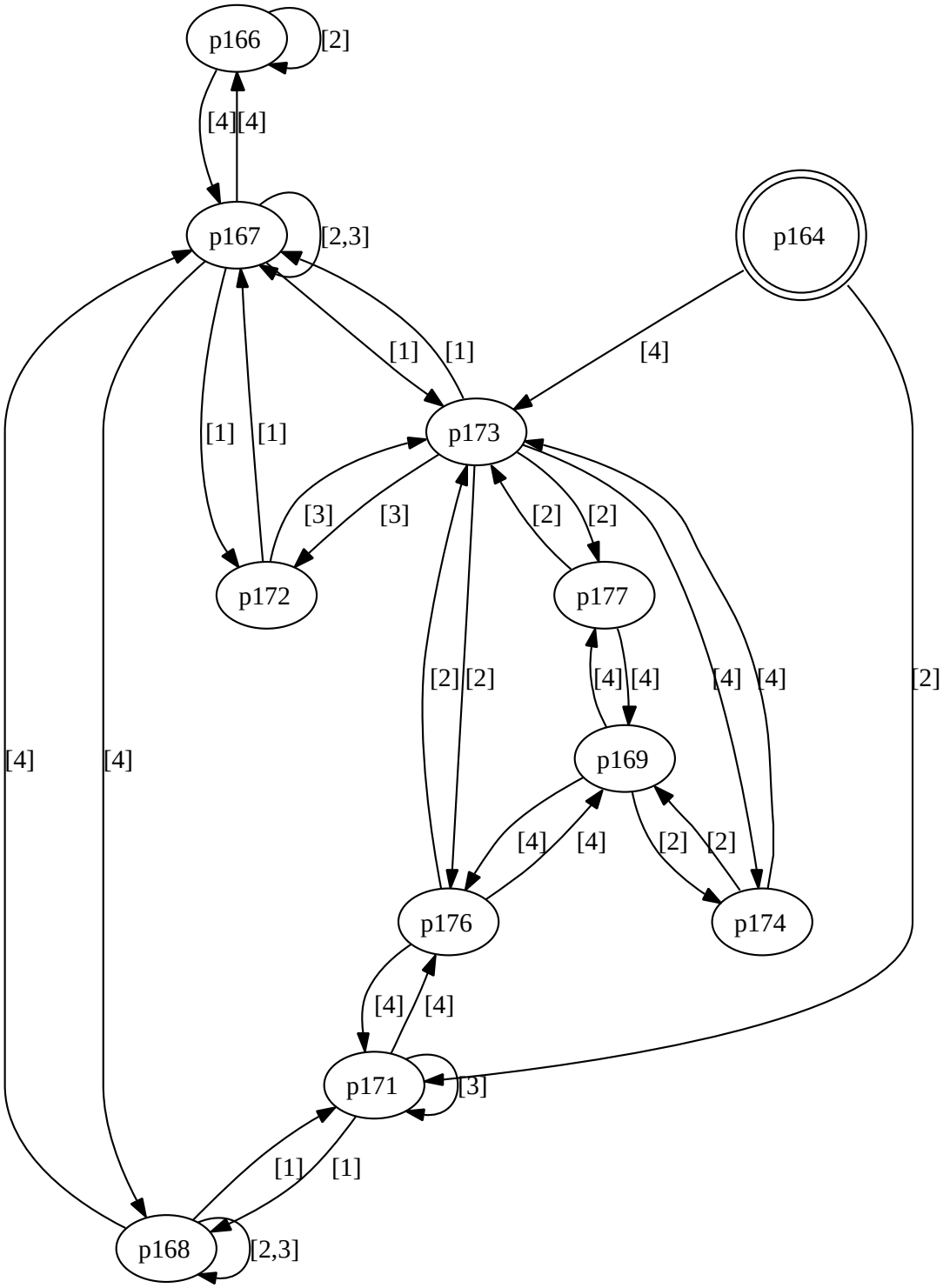

Figure 6.1: **Resulting graph of the Deque problem**

6.2 Example: Board4

The *board4* problem is a finite-state verification problem, too. Given a 4×4 board, where only the field in the bottom left corner is occupied with a piece, i. e., given a matrix with variables $x_{11}, x_{12}, \ldots, x_{21}, \ldots, x_{44}$, in the initial state, all $x_{i,j}$ are set to 0 except x_{11}, which is set to 1 (6.2). Additionally, we have the following set of transitions:

- Whenever a variable $x_{i,j}$ is set to 1 and $x_{i+1,j}$, $x_{i,j+1}$ are set to 0, we set $x_{i,j}$ to 0 and $x_{i+1,j}$, $x_{i,j+1}$ to 1.

The goal is to show, that we never can reach a state, where the variables x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{31} are set to 0 [1]. Below, we see the 9 possible transitions of the board4 problem:

- $\tau_1 \equiv x_{11} = 1 \wedge x_{12} = 0 \wedge x_{21} = 0 \wedge x'_{11} = 0 \wedge x'_{12} = 1 \wedge x'_{21} = 1$
- $\tau_2 \equiv x_{12} = 1 \wedge x_{13} = 0 \wedge x_{22} = 0 \wedge x'_{12} = 0 \wedge x'_{13} = 1 \wedge x'_{22} = 1$
- $\tau_3 \equiv x_{13} = 1 \wedge x_{14} = 0 \wedge x_{23} = 0 \wedge x'_{13} = 0 \wedge x'_{14} = 1 \wedge x'_{23} = 1$
- $\tau_4 \equiv x_{21} = 1 \wedge x_{22} = 0 \wedge x_{31} = 0 \wedge x'_{21} = 0 \wedge x'_{22} = 1 \wedge x'_{31} = 1$
- $\tau_5 \equiv x_{22} = 1 \wedge x_{23} = 0 \wedge x_{32} = 0 \wedge x'_{22} = 0 \wedge x'_{23} = 1 \wedge x'_{32} = 1$
- $\tau_6 \equiv x_{23} = 1 \wedge x_{24} = 0 \wedge x_{33} = 0 \wedge x'_{23} = 0 \wedge x'_{24} = 1 \wedge x'_{33} = 1$
- $\tau_7 \equiv x_{31} = 1 \wedge x_{32} = 0 \wedge x_{41} = 0 \wedge x'_{31} = 0 \wedge x'_{32} = 1 \wedge x'_{41} = 1$
- $\tau_8 \equiv x_{32} = 1 \wedge x_{33} = 0 \wedge x_{42} = 0 \wedge x'_{32} = 0 \wedge x'_{33} = 1 \wedge x'_{42} = 1$
- $\tau_9 \equiv x_{33} = 1 \wedge x_{34} = 0 \wedge x_{43} = 0 \wedge x'_{33} = 0 \wedge x'_{34} = 1 \wedge x'_{43} = 1$

Figure 6.4 shows the resulting graph of the board4 problem. A possible trace is for example $\tau_1, \tau_4, \tau_7, \tau_8, \tau_9, \tau_5, \tau_6, \tau_2, \tau_3$ [2]. The resulting board configuration is given in figure 6.3.

0	0	0	0
0	0	0	0
0	0	0	0
1	0	0	0

Figure 6.2: **Initial state of the board4 problem**

[1] This is even impossible for a n $\times$ n board with $n \to \infty$

[2] Note, that a path to the graph does not have to correspond to a programm execution. The resulting graph is an approximation of the system, i. e., to every program execution corresponds a path in the graph, but not to every path corresponds a program execution.

1	1	1	0
0	1	1	1
0	1	1	1
0	0	0	1

Figure 6.3: **Board configuration after trace** $\tau_1, \tau_4, \tau_7, \tau_8, \tau_9, \tau_5, \tau_6, \tau_2, \tau_3$

6.3 Example: Elevator

In 5.1.6 we have been already presented the specification of the *Elevator* example. Figure 6.5 shows the result of applying the model checking procedure to this problem. The resulting graph is also a correctness proof. So, the automaton can never reach a state, where $Curr < Min$ or $Curr > Max$.

From the initial state, i. e. $Goal = Curr = Min$ and $Dir = 0$, only the first transition, which sets the variable *Goal* to the desired floor, is possible. Given a new *Goal*, the floor counter of the elvator is increased from state p_{154} to p_{140} (by transition 3) because the value of *Goal* is greater than the value of $Curr$. Now, the floor counter is increased by transition 5 until the desired floor and the actual agree (i. e. state p_{128}). At this state, three actions are possible. Either the elevator gets a new goal that is below the actual floor or it gets a goal that is above the actual floor or it stops with transition 7. Note, that state p_{128} and p_{129} are different. If the automaton is at state p_{129}, the elevator has reached the minimal floor.

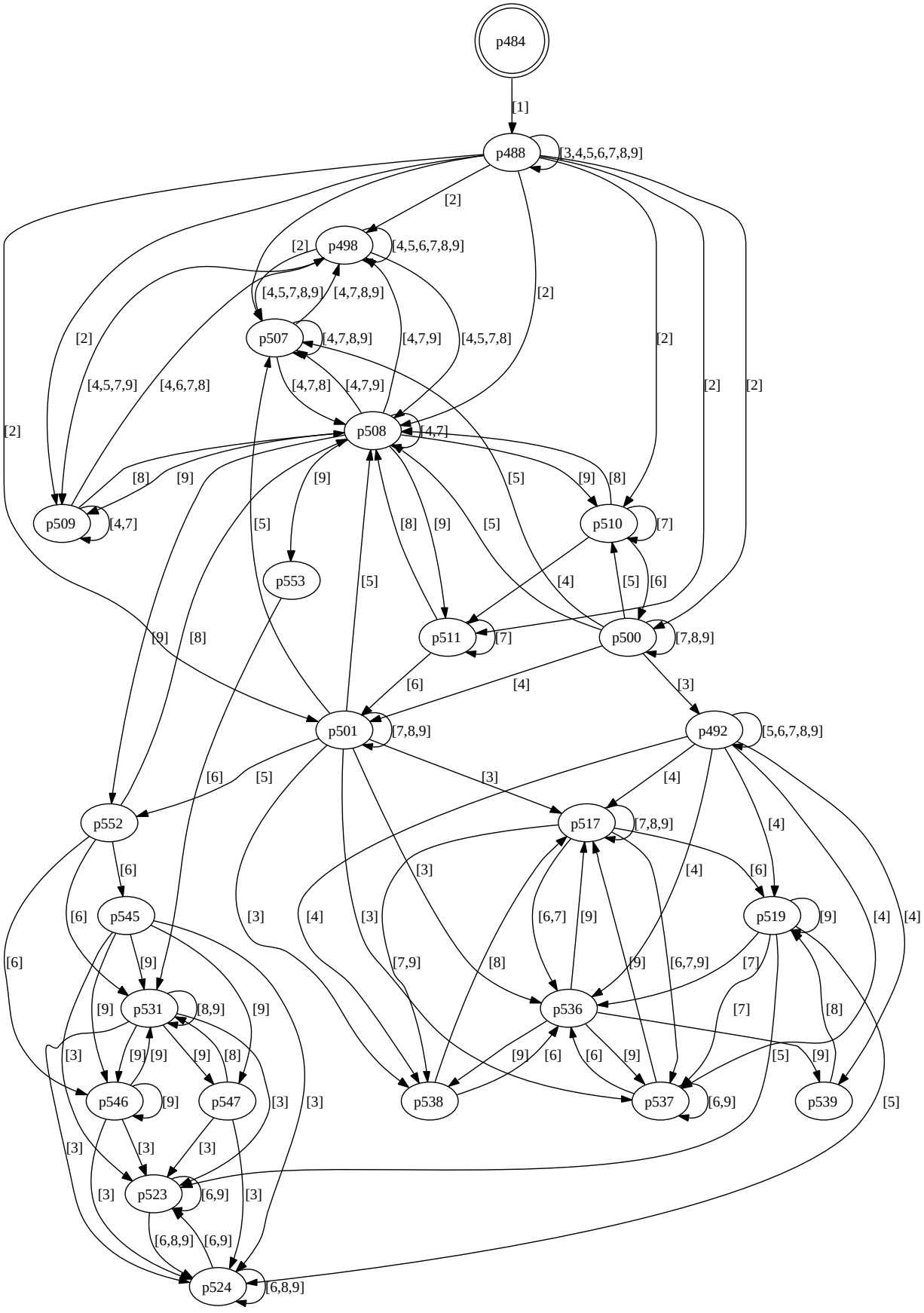

Figure 6.4: **Resulting graph of the board4 problem**

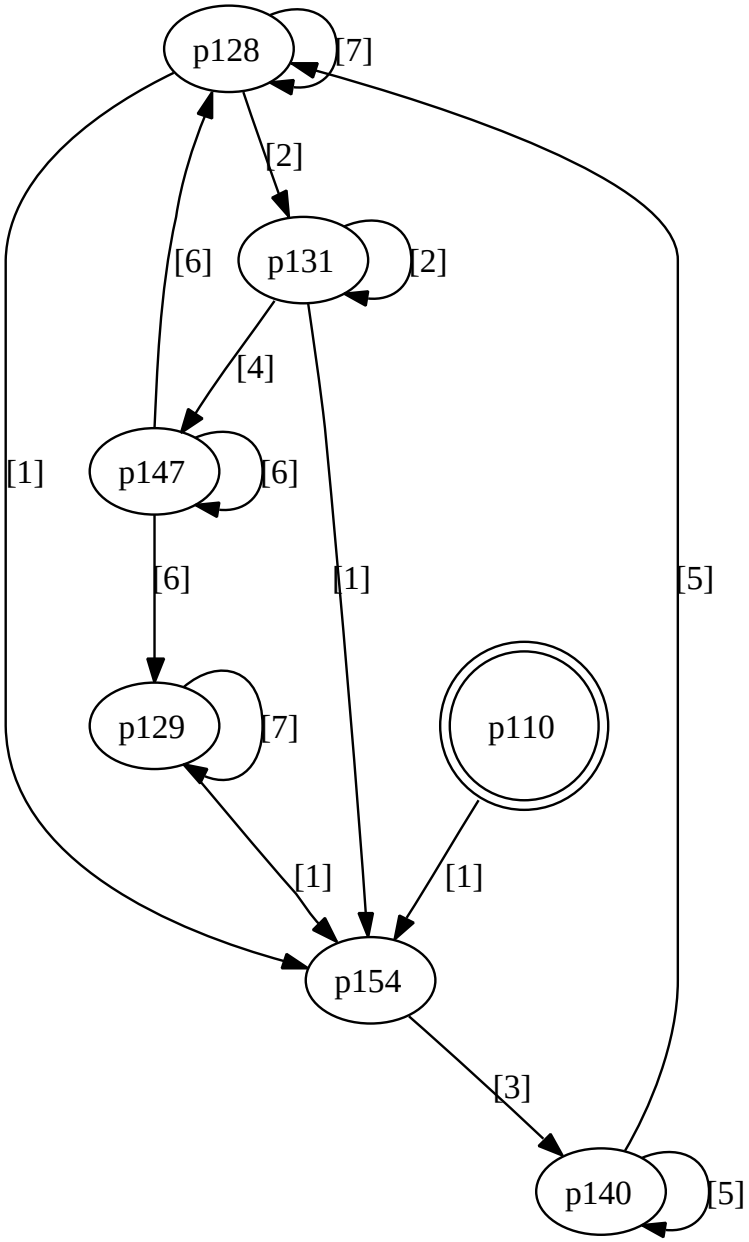

Figure 6.5: **Resulting graph of the Elevator example**

6.4 Example: Bakery

The bakery algorithm, introduced by Leslie Lamport [Lam74], is an algorithm to ensure mutual exclusion for multiple threads that try to access a shared resource.

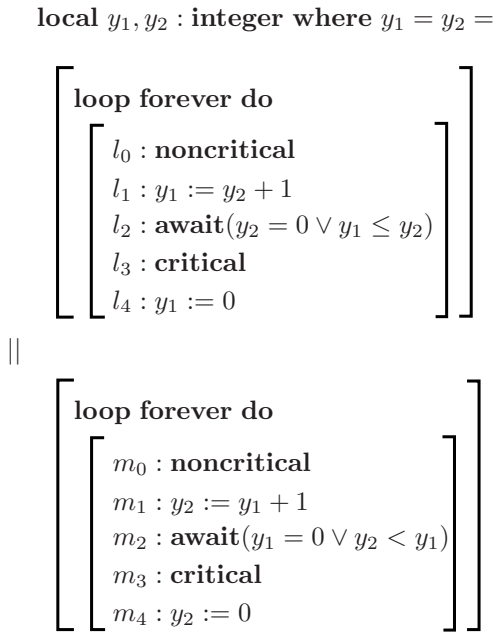

Figure 6.6: **Program Bakery for mutual exclusion**

Figure 6.6 shows a program that implements Lamport's bakery algorithm for mutual exclusion. Given the according transitions we want to ensure that we never reach a state in which both processes are in their critial section. In 6.1, 6.2 we see the configuration files for the phase event automaton. A failure state is reached, if both process counters are set to 3. Figure 6.7 shows the resulting graph of the model checking algorithm for the Bakery problem.

Listing 6.1: Configuration file for Bakery algorithm

```
pea:
init:and{and{Y1=0,Y2=0},and{Pc1=0,Pc2=0}}
transitions:transitionarguments
nodes:(p0,true,false),(p1,true,true)
intvars:Y1,Y2,Pc1,Pc2
realvars:
nlabels:(p0,true),(p1,and{Pc1=3,Pc2=3})
invariants:and{and{and{Pc1>=0,Pc1=<4},and{Pc2>=0,Pc2=<4}},
               and{Y1>=0,Y2>=0}}
edges:(p0$true$p0),(p1$true$p1),(p0$true$p1)
```

Listing 6.2: Transitions in the Bakery configuration file

```
(1,[Pc1],and{Pc1=0,Pc1'=1}),
(2,[Pc2],and{Pc2=0,Pc2'=1}),
(3,[Pc1,Y1],and{Pc1=1,and{Pc1'=2,Y1'=Y2+1}}),
(4,[Pc2,Y2],and{Pc2=1,and{Pc2'=2,Y2'=Y1+1}}),
(5,[Pc1],and{Pc1=2,and{Y2=0,Pc1'=3}}),
(6,[Pc2],and{Pc2=2,and{Y1=0,Pc2'=3}}),
(7,[Pc1],and{Pc1=2,and{Y1=<Y2,Pc1'=3}}),
(8,[Pc2],and{Pc2=2,and{Y2<Y1,Pc1'=3}}),
(9,[Pc1],and{Pc1=2,and{and{Y2=/=0,Y1>Y2},Pc1'=2}}),
(10,[Pc2],and{Pc2=2,and{and{Y1=/=0,Y2>=Y1},Pc2'=2}}),
(11,[Pc1,Y1],and{Pc1=3,and{Pc1'=4,Y1'=0}}),
(12,[Pc1],and{Pc1=3,Pc1'=3}),
(13,[Pc2],and{Pc2=3,Pc2'=3}),
(14,[Pc2,Y2],and{Pc2=3,and{Pc2'=4,Y2'=0}}),
(15,[Pc1],and{Pc1=4,Pc1'=0}),
(16,[Pc2],and{Pc2=4,Pc2'=0})
```

Note, that the resulting graph contains two nodes, that are marked initial. Whereas node $p222$ is the 'real' initial state of the diagram, node $p337$ is marked initial by source enlargement. Again, there are a lot of states in which the domains of the variables are not exactly determined. Node $p340$ for example does not restrict its variable pc_2 to be 1 or 0. Therefore, it is still possible to split this node. But, because the are no nodes left, that are marked as failure states, we do not need to further split that node. Consequently, a trace τ_1, τ_3 to node $p344$ and τ_4 to node $p312$ is possible, without having ever used transition τ_2. Again, i. e., because the graph is an approximation of the system.

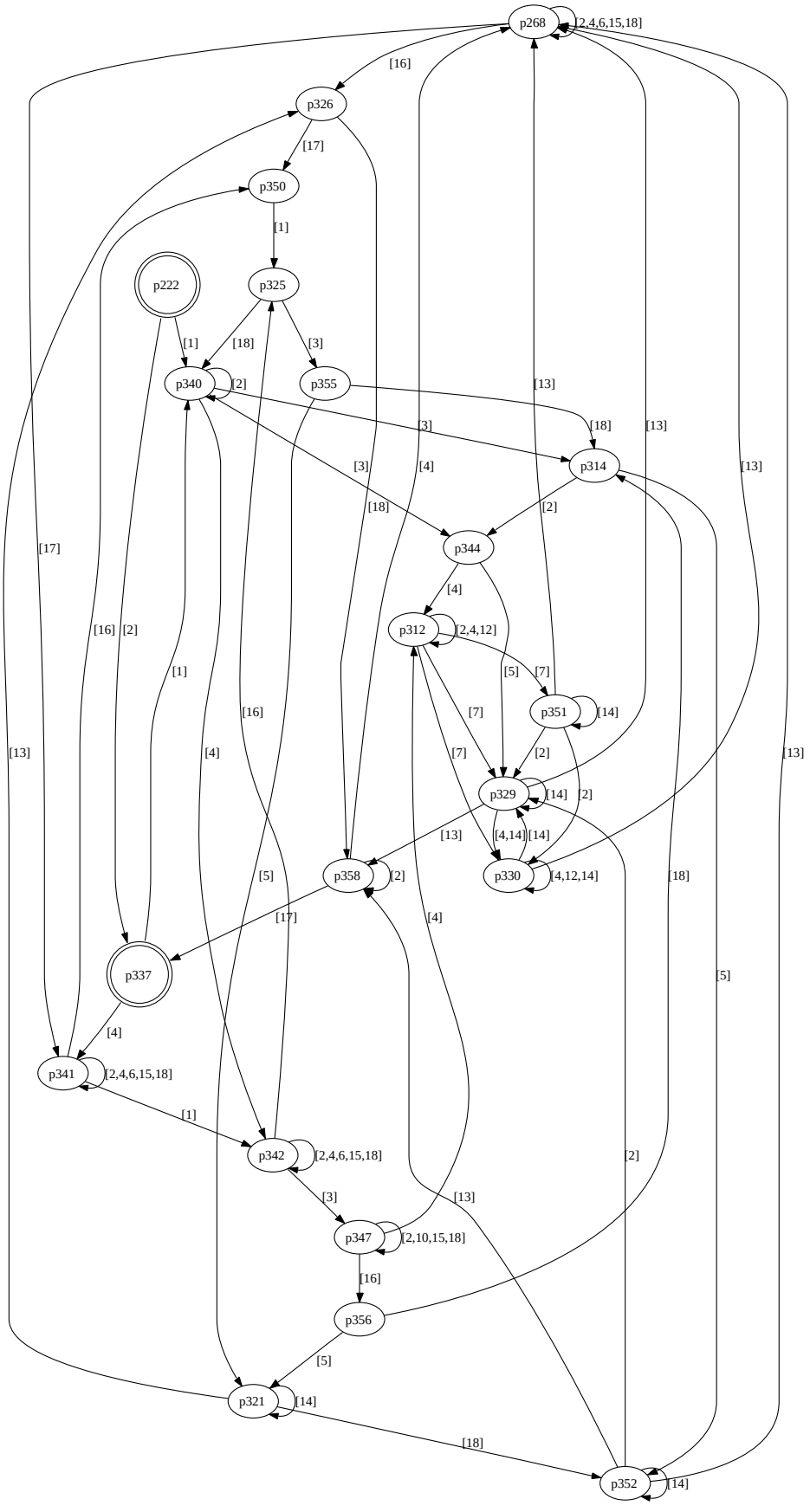

Figure 6.7: **Resulting graph of the Bakery problem**

6.5 Example: Fischer's Problem

In 2.5 we have already been presented the Fischer's problem for mutual exclusion. All examples of Fischer's problem that are used so far have been a special version where the clocks are always incremented by 1. In this section, the resulting graph of another version where clocks are incremented by a real valued number is presented. Additionally, the number of consecutive clock ticks is restricted. I. e., we can simulate the effects of several consecutive ticks in one tick and therefore a clock tick is only possible if the previous transition was not the tick transition. Below, the changed transitions and initial state in the configuration file will be presented first. In figure 6.8 we will finally see the resulting graph of Fischer's mutual exclusion problem with real valued clocks.

Listing 6.3: Initial state in the Fischer configuration file

```
and{Pc1=0,and{Pc2=0,and{D>0,and{C1>=0,and{C2>=0,and{T>=0,
                                        and{T=<1,and{X>=0,X=<2}}}}}}}}
```

Listing 6.4: Transitions in the Fischer configuration file

```
(1,[Pc1,C1,T],and{and{and{X=0,Pc1=0},and{Pc1'=1,C1'=0}},T'=0}),
(2,[Pc2,C2,T],and{and{and{X=0,Pc2=0},and{Pc2'=1,C2'=0}},T'=0}),
(3,[X,C1,Pc1,T],and{and{C1<2,and{Pc1=1,and{X'=1,
                                        and{C1'=0,Pc1'=2}}}},T'=0}),
(4,[X,C2,Pc2,T],and{and{C2<2,and{Pc2=1,and{X'=2,
                                        and{C2'=0,Pc2'=2}}}},T'=0}),
(5,[Pc1,T],and{and{Pc1=1,and{C1>=2,Pc1'=0}},T'=0}),
(6,[Pc2,T],and{and{Pc2=1,and{C2>=2,Pc2'=0}},T'=0}),
(7,[Pc1,T],and{and{Pc1=2,and{C1>3,and{X=1,Pc1'=3}}}},T'=0}),
(8,[Pc2,T],and{and{Pc2=2,and{C2>3,and{X=2,Pc2'=3}}}},T'=0}),
(9,[Pc1,T],and{and{Pc1=2,and{and{C1>3,X=/=1},Pc1'=0}},T'=0}),
(10,[Pc2,T],and{and{Pc2=2,and{and{C2>3,X=/=2},Pc2'=0}},T'=0}),
(11,[X,Pc1,T],and{and{Pc1=3,and{X'=0,Pc1'=0}},T'=0}),
(12,[X,Pc2,T],and{and{Pc2=3,and{X'=0,Pc2'=0}},T'=0}),
(13,[C1,C2,D,T],and{T=0,and{C1'=C1+D,and{C2'=C2+D,
                                        and{D'>0,T'=1}}}})
```

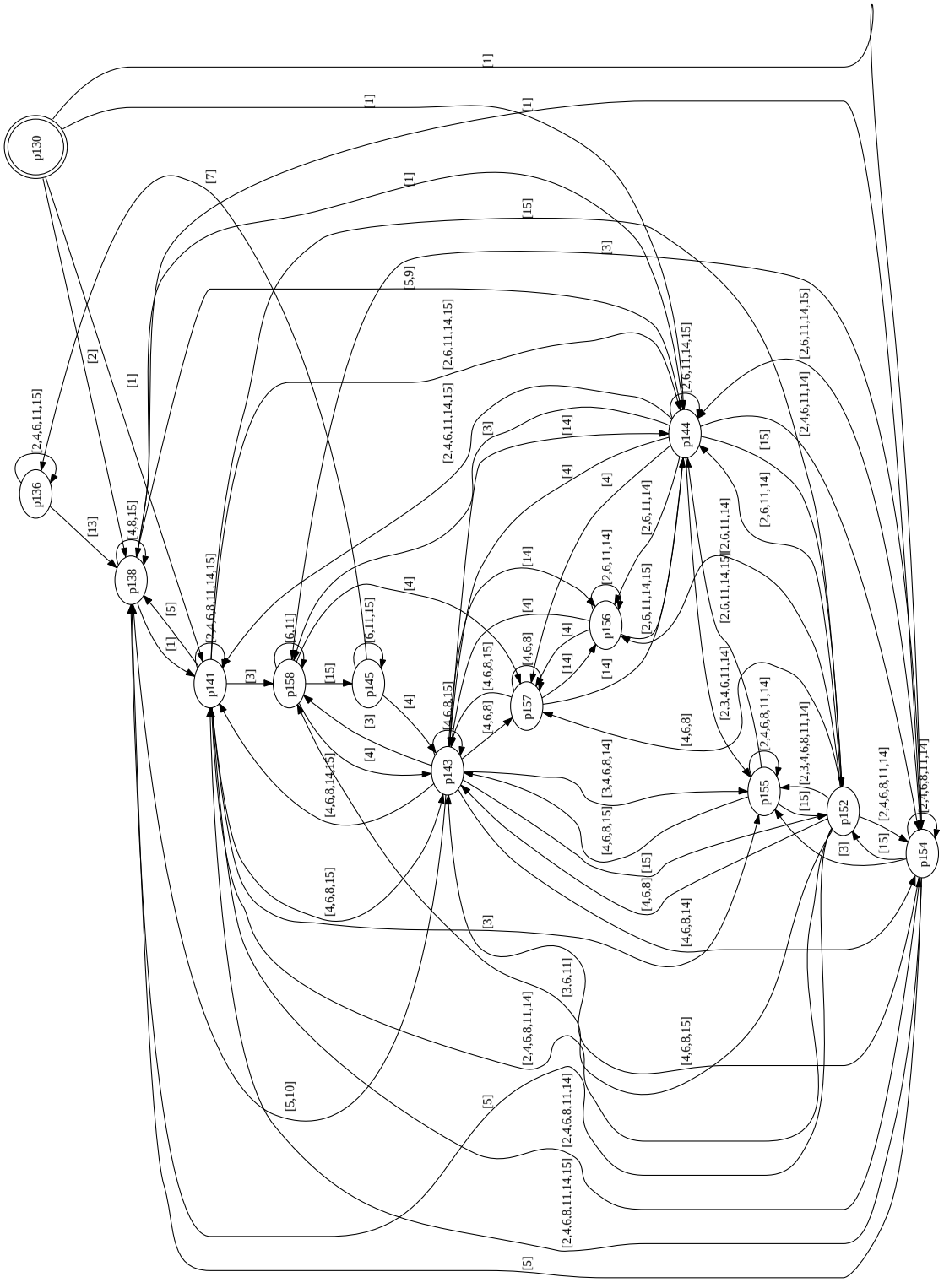

Figure 6.8: **Resulting graph of the Fisher problem**

Chapter 7

Conclusion

In this thesis, I introduced an extension of deductive model checking using phase event automata. The resulting model checking procedure does not need user guidance to efficiently apply the splitting rules. Auxiliary formulas, like invariants that restrict variable domains, may be provided by the user to speed up the verification process and reduce the size of the falsification diagram.

Several heuristics, like *target enlargement*, *source enlargement* and a rating function have been added to the original procedure, both to decide which edges have to be split and to decrease the search space. Structural changes, like the restriction of node labels to be conjunctions, failure nodes to have no consecutives and disjunctiveness of new created nodes, additionally speed up the process and reduce the size of the diagram.

7.1 Future Work

Currently, the model checking algorithm can only manage linear constraints over the reals. The extension of the procedure to reason about complex data types like array, lists and sets could be a further goal (dependent on the availability of suitable decision procedures).

Additionally, the size of many infinite-state verification problems can be significantly reduced by augmenting the actual set of heuristicts with new rules.

Currently, the model checking algorithm is restricted to prove safety properties. Hence, we could extend the procedure by the missing transformation rules given in [HTZ96] to allow also for the verification of *liveness* properties.

Acknowledgements

I want to thank Klaus Dräger for his inspirative and patient assistance.

Bibliography

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[BAMP81] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 164–176, New York, NY, USA, 1981. ACM Press.

[BKA02] J. Baumgartner, A. Kuehlmann, and J. Abraham. Property checking via structural analysis, 2002.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CS01] E.M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 24, pages 1635–1790. Elsevier Science, 2001.

[CWA+96] Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, Rance Cleaveland, David Dill, Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas Henzinger, Gerard Holzmann, Cliff Jones, Robert Kurshan, Nancy Leveson, Kenneth McMillan, J. Moore, Doron Peled, Amir Pnueli, John Rushby, Natarajan Shankar, Joseph Sifakis, Prasad Sistla, Bernhard Steffen, Pierre Wolper, Jim Woodcock, and Pamela Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[Hen06] M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. PhD thesis, ICIS, Radboud University Nijmegen, April 2006.

[HM05] Jochen Hoenicke and Patrick Maier. Model-checking of specifications integrating processes, data and time. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods; International Symposium of Formal Methods Europe*, volume 3582 of *Lecture Notes in Computer Science*, pages 465–480, Newcastle, UK, July 2005. Formal Methods Europe (FME), Springer.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Hoe06] Jochen Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, July 2006.

[HTZ96] H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of*

the Eighth International Conference on Computer Aided Verification CAV, volume 1102, pages 208–219, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[KMMP93] Yonit Kesten, Zohar Manna, Hugh McGuire, and Amir Pnueli. A decision algorithm for full propositional temporal logic. In *Computer Aided Verification*, pages 97–109, 1993.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[Muk97] Madhavan Mukund. Linear-time temporal logic and Büchi automata. 1997.

[MW84] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, 1984.

[Pnu97] A. Pnueli. The temporal logic of programs. Technical report, Jerusalem, Israel, Israel, 1997.

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

[ZH04] Chaochen Zhou and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems (Monographs in Theoretical Computer Science. an Eatcs Seris)*. SpringerVerlag, 2004.

[ZHR91] Chaochen Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Proc. Letters*, 40(5), Dec. 1991.