Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Generalizing lazy abstraction refinement algorithms with partial orders

Bachelor's Thesis

Author
Dóra Cziborová

Advisor
Kristóf Marussy
dr. András Vörös

December 10, 2021

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Cziborová Dóra*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2021. december 10.

Cziborová Dóra
hallgató

Kivonat

Kritikus valósidejű rendszerek fejlesztésében egy kulcsfontosságú kihívás a szoftverrendszerek biztonságának verifikációja. Az automatizált modellellenőrzésben a rendszernek egy formális leírásán kerülnek ellenőrzésre a kívánt biztonsági követelmények, egy automatizált eszköz által.

A modellellenőrzés sajnos akár már közepes méretű rendszereken végezve is algoritmikusan nehéz feladat lehet. A modellellenőrzés során bejárnandó állapotok száma véges számú lehetséges viselkedéssel rendelkező rendszerek esetén is gyakran a rendszer méretében exponenciálisan nő. Ezen kívül az időzítések figyelembe vételéhez az állapotoknak egy megszámlálhatatlanul végtelen halmazával szükséges dolgozni.

Az állapottér kompakt reprezentációjára számos absztrakció alapú módszer létezik az irodalomban. Ezen módszerek között a lusta absztrakció és változatai egy hatékony megoldást nyújtanak az időzített modellek analízisére. A vizsgált programok változatos tulajdonságai szükségessé teszik a különböző fajta lusta absztrakciókat. Ezek között nem létezik egyetlen legjobb algoritmus, így szükség van egy konfigurálható lusta absztrakció alapú keretrendszerre a lehetséges megközelítések általánosításaként.

Munkám során a részbenrendezések és Galois kapcsolatok elméletét alkalmazva egy konfigurálható lusta absztrakció alapú keretrendszer kidolgozását céloztam meg. Ezen belül is (i) definiálok egy általános lusta absztrakciófinomítási (elméleti) keretrendszert, (ii) integrálom a létező lusta absztrakciós módszereket a keretrendszerbe, (iii) új kombinációit és konfigurációit nyújtom a lusta absztrakciós módszereknek, valamint (iv) implementálok egy lusta absztrakció alapú modellellenőrző megközelítést a nyílt forráskódú Theta eszközben. Az algoritmusok alkalmazhatóságát benchmark modelleken értékeljük ki.

Abstract

Safety verification of software systems is a key challenge in the development of critical real-time embedded systems. In automated software model checking, a formal description of the system is checked against the desired safety properties by an automated tool.

However, model checking systems of even a moderate size can be difficult. For systems with a finite number of possible behavior, the number of states that has to be traversed during model checking often grows exponentially in the size of the system. Moreover, taking timing into account requires reasoning with an uncountably infinite set of states.

To provide a compact representation of the state space, various abstraction-based techniques were introduced in the literature. Among these techniques, lazy abstraction and its variants provide an efficient solution for the analysis of timed models with data. The various characteristics of the programs being analyzed necessitate different kinds of lazy abstractions. There is no single best algorithm; therefore, there is a need for a configurable lazy abstraction framework as a generalization of the available approaches.

Our work aims to provide a configurable lazy abstraction framework by adapting the theory of partial orders and Galois connections to lazy abstractions. In particular, we (i) define a general (theoretical) framework of lazy abstraction refinement; (ii) integrate the existing lazy abstraction techniques into the framework; (iii) provide new combinations and configurations of the lazy abstraction techniques; and (iv) implement a lazy abstraction based model checker in the open source Theta tool. We evaluate the applicability of the algorithms on benchmark models.

Chapter 1

Introduction

Critical real-time systems are expected to always maintain correct behaviour, therefore safety verification is a crucial part in the development of these systems. Verification is done by analysing the software with an automated model checker tool, the system is checked against the desired safety properties. The result of the analysis is either a proof of correctness or an error trace serving as a counterexample for correctness.

The main idea of model checking is to define error states based on the desired safety properties, then traverse all possible states of the system and check for the presence of an error state. However, the number of states of a system often grows exponentially in the size of the system. This phenomenon is known as the state space explosion problem. Because of this, model checking can be difficult even for systems of moderate size.

There are numerous abstraction-based approaches in the literature. They provide a solution for the state space explosion problem by representing the state space in a more compact form, i.e. by abstract states that comprise multiple states of the system. In real-time system timing also has to be taken into account. This creates another challenge in model checking, as the elapse of time introduces continuity and hence an uncountably infinite state space. To tackle this, special abstractions have to be used for the timed components of the system.

Among abstraction-based approaches, lazy abstraction, which is the focus of this work, and its variants provide an efficient solution for the analysis of timed systems with both timing and data variables.

A universally best algorithm for model checking cannot be given, as different models have different characteristics that determine the optimal technique for the given model. Moreover, for some models the set of abstractions that can be used is quite limited. This means that the use of different abstractions is necessary and it creates the need for a configurable abstraction framework.

The goal of our work is to provide this configurable framework by generalising the existing lazy abstraction techniques. The theoretical results then can be put into practice by being implemented in the Theta model checking framework.

Structure of this work Chapter 2 provides the necessary preliminaries including the basics of model checking and timed automata as a modelling formalism for timed systems. Chapter 3 presents different abstractions that can be used with lazy abstraction. The general lazy abstraction framework itself and the corresponding algorithms are also described here. Chapter 4 presents the interpolation techniques that are an integral part of

our lazy abstraction algorithm. Chapter 5 gives a brief overview of the implementation of the approach in Theta. Chapter 6 analyses the results of the implementation in the form of experiments on benchmark models. Chapter 7 discusses other related works. Chapter 8 summarizes our work and describes the possibilities for improvements of the presented approach.

Chapter 2

Preliminaries

2.1 Model checking

Model checking is a formal method built on a mathematical basis, which is capable of proving that certain specified properties hold in a model. The desired properties are typically safety properties (an unsafe state of the system is never reached) and liveness properties (the desired state is always reached eventually).

The inputs of a model checking algorithm are the model and the requirements, both formally specified. The algorithm then outputs whether the given model satisfies the given requirements or, in case of the model not satisfying a requirement, it may also output a corresponding counterexample, where the requirement is not met.

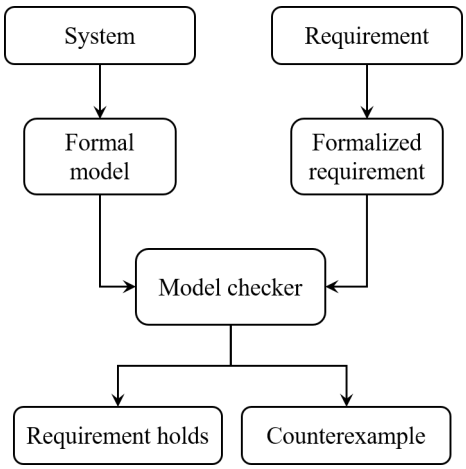


Figure 2.1: Model checking

As safety requirements usually define an error state that must not be reached, checking whether the model satisfies the requirement is reduced to a reachability problem.

When checking whether a state is reachable in a model, explicitly enumerating all states on all paths would not be an efficient or viable approach at all. This is because of the state space explosion problem, which means that the state space can become unmanageably large even in a relatively small model. Therefore, one of the main challenges in model checking is to find more efficient solutions to the reachability problem.

Many reachability analysis techniques are based on *abstraction*. The typical approach is applying *over-approximation* to the model. The over-approximated model preserves

all behaviour of the original model but also enables behaviour that is not present in the original one. It follows, that if a state is unreachable in the abstract model, then it is not reachable in the original model as well. However, it also follows that a model checking algorithm may find counterexamples in the abstract model that are not present in the original one, and thus false positives may occur when the abstraction is too coarse.

2.2 Modelling formalisms for software systems

In order to perform model checking on a system, it has to be represented as a formalized model. These modelling formalisms are most often graph-based, with nodes representing states and directed edges representing transitions of the system.

2.2.1 Control flow automata

When modelling programs, *control flow automata* provide a straightforward formalism.

Definition 1 (Control flow automaton). A control flow automaton is a tuple $CFA = \langle L, l_0, V, E \rangle$, where

- L is a finite set of locations,
- $l_0 \in L$ is the initial location,
- V is a finite set of variables,
- $E \subseteq L \times OP \times L$ is a set of edges. ▪

In a CFA, locations model the program counter. Edges are directed edges between locations, modelling the transition from a location to another while executing an operation $op \in OP$.

2.2.2 Timed automata

For the verification of timed systems such a model is required where the modelling of time is possible. Timed automata provide a suitable formalism, where time is modelled using clock variables.

Clock variables are real-valued variables. All clock variables are initialized at zero and are always incremented equally within the model. $\mathcal{C}$ denotes the set of clock variables in a timed automaton.

A *clock valuation* v is a mapping that maps each clock variable to a non-negative real number, $v(c)$ denotes the value of a clock variable c within the valuation v.

Applying a delay $\delta \in \mathbb{R}_{\geq 0}$ to a valuation v is denoted by $v + \delta$ and maps each $c \in \mathcal{C}$ to $v(c) + \delta$.

Clock variables can be reset at transitions, and that to any value, although the term resetting usually refers to setting a clock to zero. The new valuation after resetting is denoted by $[\lambda]v$ where $\lambda \subseteq \mathcal{C}$ and maps each $c \in \lambda$ to 0 and each $c \in \mathcal{C} \backslash \lambda$ to $v(c)$.

Clock constraints are formulas of the form $c_1 \sim n$ or $c_1 - c_2 \sim n$, where $c_1, c_2 \in \mathcal{C}$, $n \in \mathbb{N}$ and $\sim \in \{<, \leq, =, \geq, >\}$. $\mathcal{B}(\mathcal{C})$ denotes the set of clock constraints.

Clock constraints can appear either at locations or on edges of the automaton. At locations they are called *invariants*. Constraints on edges are *guards* that control which transitions are enabled.

A *timed automaton* is a finite automaton extended with clock variables.

Definition 2 (Timed automaton). A timed automaton [1] is a tuple $\mathcal{A} = \langle L, l_0, \mathcal{C}, X, \Sigma, \mathcal{G}, I, E \rangle$, where

- L is a finite set of locations,
- $l_0 \in L$ is the initial location,
- $\mathcal{C}$ is a finite set of clock variables,
- X is a finite set of data variables,
- Σ is a finite set of actions,
- $\mathcal{G}$ is a finite set of guards,
- $I \subseteq L \to 2^{\mathcal{G}}$ is a mapping that maps locations to invariants,
- $E \subseteq L \times \mathcal{G} \times \Sigma \times 2^{\mathcal{C}} \times L$ is a finite set of edges .

An edge $\langle l, g, a, \lambda, l' \rangle$ represents a transition from location l to l' with guard $g \in \mathcal{G}$, action $a \in \Sigma$ and a set $\lambda \subseteq \mathcal{C}$ of clocks to reset.

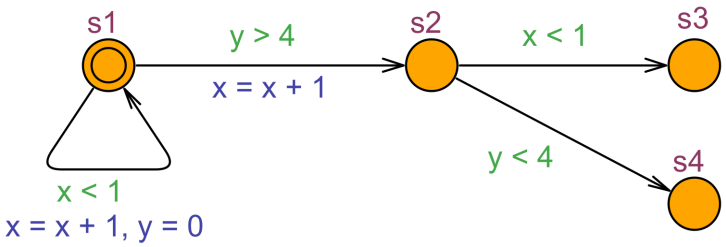


Figure 2.2: Example of a timed automaton with locations s1, s2, s3, s4, initial location s1 and variables x and y. Guards are marked by green colour, actions are marked by blue.

The semantics of timed automata is defined by a transition system, where a state is a tuple $\langle l, v, x \rangle$ consisting of a location l, a clock valuation v, and a valuation x on data variables, such that v and x satisfy the invariant $I(l)$. The initial state of the automaton is $\langle l_0, v_0, x_0 \rangle$, where $v_0(c) = 0$ for all $c \in \mathcal{C}$.

Transitions are either delays or actions:

- A delay transition $\langle l, v, x \rangle \xrightarrow{\delta} \langle l', v', x' \rangle$ is enabled if $v + \delta'$ satisfies $I(l)$ for all $0 \leq \delta' \leq \delta$. A delay means the elapse of time while the automaton stays at the same location, resulting in $l' = l$, $x' = x$ and $v' = v + \delta$.
- An action $\langle l, v, x \rangle \xrightarrow{e} \langle l', v', x' \rangle$ is enabled if an edge $e = \langle l, g, a, \lambda, l' \rangle \in E$ exists, v and x satisfy the guard g, and v' and x' satisfy the invariant $I(l')$ of the target location. At the firing of this transition, action a is executed and each clock $x \in \lambda$ is reset, resulting in $v' = [\lambda]v$.

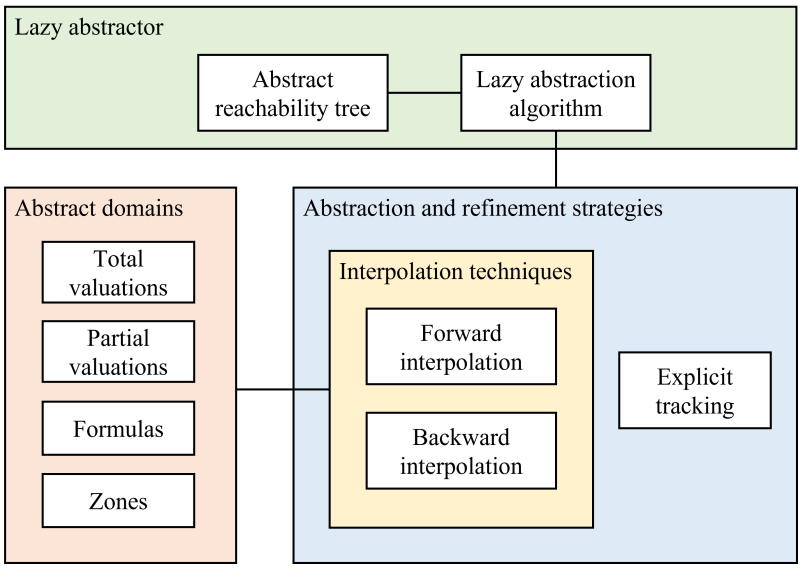
The next state for a state and a transition (the effect of a transition on a state) is computed by using the concrete *post-image* operator. A post-image $post_t(s)$ represents the postcondition of state s with respect to transition t.

A run of a timed automaton is a sequence of states from the initial state $\langle l_0, v_0, x_0 \rangle$ along enabled transitions $t_i \in \mathbb{R}_{\geq 0} \cup E$: $\langle l_0, v_0, x_0 \rangle \xrightarrow{t_1} \langle l_1, v_1, x_1 \rangle \xrightarrow{t_2} \ldots \xrightarrow{t_n} \langle l_n, v_n, x_n \rangle$. A location l is *reachable* if and only if a run exists where $l_n = l$.

Chapter 3

Framework for lazy abstractions

3.1 Overview of the approach



Our approach is a lazy algorithm that is based on abstraction and *abstraction refinement*. It builds an abstract reachability graph of the reachable states and occasionally refines the abstraction in the meantime. The abstraction and refinement strategies for the algorithm are configurable, we can choose to track all variables explicitly or we can choose an interpolation technique that suits our model and the abstract domain we use. The presented framework is general, it can work with various abstract domains.

3.2 Abstract domains

The state space of a system can be of infinite size. To handle an infinite state space, abstractions are used, which means that we use abstract labels instead of concrete states. Abstract labels are capable of comprising multiple states. With the proper use of abstractions, the state space of a system can be represented by a significantly smaller set of abstract labels.

Definition 3 (Abstract domain). An abstract domain is a tuple $\mathbf{D} = \langle \mathcal{S}, \mathcal{D}, \sqsubseteq, \gamma \rangle$, where

- $\mathcal{S}$ is the set of *states*,
- $\mathcal{D}$ is the set of *abstract labels*,
- $\sqsubseteq \subseteq \mathcal{D} \times \mathcal{D}$ is a *preorder*, i.e., it is a reflexive ($d \sqsubseteq d$ for all $d \in \mathcal{D}$) and transitive ($d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_3$ implies $d_1 \sqsubseteq d_3$ for all $d_1, d_2, d_3 \in \mathcal{D}$) binary relation,
- $\gamma \colon \mathcal{D} \to 2^{\mathcal{S}}$ is the *concretization* function that maps abstract labels to the sets of states they represent, such that $d_1 \sqsubseteq d_2$ implies $\gamma(d_1) \subseteq \gamma(d_2)$ for all $d_1, d_2 \in \mathcal{D}$. ▪

If there exists a function $\alpha \colon 2^{\mathcal{S}} \to \mathcal{D}$ such that $\gamma(\alpha(A)) \subseteq A$ and $A \subseteq \gamma(d)$ implies $\alpha(A) \sqsubseteq d$ for all $d \in \mathcal{D}$, then α is the *abstraction* function corresponding to γ and we say that $2^{\mathcal{S}} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}$ form a *Galois connection*.

3.2.1 Explicit value abstraction

In *explicit value abstraction*, we abstract over the possible *valuations* $\mathcal{S} = \{val \colon X \to V\}$ of the data variables X. Each abstract label $d \in \mathcal{D}$ explicitly tracks the values of some subset $\mathrm{supp}(d) \subseteq X$ of data variables, while the rest of the variables may take any value.

Definition 4 (Explicit value abstraction). Explicit value abstraction over the data variables X taking values from the set V uses the domain $\mathbf{Expl}(X, V) = \langle \mathcal{S}, \mathcal{D}, \sqsubseteq, \gamma \rangle$, where

- $\mathcal{S}$ is the set of all *total valuations* $\{val \colon X \to V\}$,
- $\mathcal{D}$ is the set of all *partial valuations* $\{pval \colon \mathrm{supp}(pval) \to V \mid \mathrm{supp}(pval) \subseteq X\}$ and the *inconsistent valuation* $\bot$,
- $\bot \sqsubseteq d$ for all $d \in \mathcal{D}$, while $pval_1 \sqsubseteq pval_2$ iff $\mathrm{supp}(pval_1) \supseteq \mathrm{supp}(pval_2)$ and $pval_1(x) = pval_2(x)$ for all $x \in \mathrm{supp}(pval_2)$,
- $\gamma(\bot) = \emptyset$ and $\gamma(pval) = \{val \in \mathcal{S} \mid val(x) = pval(x) \text{ for all } x \in \mathrm{supp}(pval)\}$. ▪

We will denote the *empty* partial valuation as $\top \in \mathcal{D}$, i.e., $\mathrm{supp}(\top) = \emptyset$ and $\gamma(\top) = \mathcal{S}$.

The corresponding abstraction function α forms the least general abstraction $\alpha(A)$ of a set of states $A \subseteq \mathcal{S}$ by collecting variables that have only a single value. In other words, $x \in \mathrm{supp}(\alpha(A))$ and $\alpha(A)(x) = v$ if $val(x) = v$ for all $val \in A$. Otherwise, if there are some $val_1, val_2 \in A$ such that $val_1(x) \neq val_2(x)$, then $x \notin \mathrm{supp}(\alpha(A))$. As a special case, we have $\alpha(\emptyset) = \bot$.

3.2.2 Identity abstraction

We may handle problems where we do not wish to introduce any abstraction, i.e., we set $\mathcal{D} = \mathcal{S}$. This allows extending our lazy abstraction based model checker to cases where we do not want to apply abstraction to parts of the system, while we still remain to be able to use abstraction for other parts where it delivers a benefit for the scalability of the algorithm.

Definition 5 (Identity abstraction). The *identity abstraction* over a set of states $\mathcal{S}$ is $\mathbf{Id}(\mathcal{S}) = \langle \mathcal{S}, \mathcal{S}, \sqsubseteq, \gamma \rangle$, where

- $s_1 \sqsubseteq s_2$ iff $s_1 = s_2 \in \mathcal{S}$,
- $\gamma(s) = \{s\}$. ▪

In particular, we will use identity abstraction over the set of total valuations $\mathbf{Total}(X, V) = \mathbf{Id}(\{val\colon X \to V\})$ to consider the set of data variable valuations of a system without applying any abstraction.

3.2.3 Formula abstraction

Definition 6 (First-order signature). A first-order logic (FOL) *signature* is a pair $\langle \Sigma, arity \rangle$, where

- $\Sigma = \{f_1, \ldots, f_m, R_1, \ldots, R_k\}$ is a finite set of *function symbols* $f_1, \ldots, f_m$ and *relation symbols* $R_1, \ldots, R_k$,
- $arity\colon \Sigma \to \mathbb{N}$ is the *arity* function. ▪

Definition 7 (First-order interpretation). An *interpretation* of the FOL signature $\langle \Sigma, arity \rangle$ over the set of values V is the function $\mathcal{I}$ that assigns the interpretation to all symbols $\sigma \in \Sigma$, i.e.,

- $\mathcal{I}(f_i)\colon V^{arity(f_i)} \to V$ is the functional interpretation of the function symbol f_i,
- $\mathcal{I}(R_j) \subseteq V^{arity(R_j)}$ is the relational interpretation of the relation symbol R_j. ▪

Definition 8 (First-order formula). The set of FOL formulas is inductively defined from *variable references* x_i, *function applications* $f_i(\varphi_1, \ldots, \varphi_{arity(f_i)})$, *relation applications* $R_j(\varphi_1, \ldots, \varphi_{arity(R_j)})$, *quantifiers* $\exists x_i\colon \varphi$ and $\forall x_i\colon \varphi$, and *logical connectives* $\neg, \vee, \wedge, \rightarrow$. ▪

Let $\Phi_{\langle \Sigma, arity \rangle}$ denote the set of all FOL formulas over $\langle \Sigma, arity \rangle$. For brevity, we will omit the signature from the subscript if it is clear from context.

Definition 9 (Semantics of formulas). The semantics $[\![\varphi]\!]_{val}$ of the FOL formulas $\varphi \in \Phi$ at the valuation val according to the interpretation $\mathcal{I}$ are defined as

- $[\![x_i]\!]_{val} = val(x_i)$, where val is a valuation with $x_i \in \mathrm{supp}(val)$,
- $[\![f_i(\varphi_1, \ldots, \varphi_{arity(f_i)})]\!]_{val} = \mathcal{I}(f_i)([\![\varphi_i]\!]_{val}, \ldots, [\![\varphi_{arity(f_i)}]\!]_{val})$,
- $[\![R_j(\varphi_1, \ldots, \varphi_{arity(f_i)})]\!]_{val} = \texttt{true}$ if $\langle [\![\varphi_i]\!]_{val}, \ldots, [\![\varphi_{arity(R_j)}]\!]_{val} \rangle \in \mathcal{I}(R_j)$, otherwise `false`,
- $[\![\exists x_i\colon \varphi]\!]_{val} = \texttt{true}$ if there exists some $v \in V$ such that $[\![\varphi]\!]_{val, x_i \mapsto v} = \texttt{true}$, otherwise `false`,
- $[\![\forall x_i\colon \varphi]\!]_{val} = \texttt{true}$ if $[\![\varphi]\!]_{val, x_i \mapsto v} = \texttt{true}$ for all $v \in V$, otherwise `false`,
- $[\![\neg\varphi]\!]_{val} = \neg[\![\varphi]\!]_{val}$, $[\![\varphi_1 \vee \varphi_r]\!]_{val} = [\![\varphi_1]\!]_{val} \vee [\![\varphi_2]\!]_{val}$, $[\![\varphi_1 \wedge \varphi_r]\!]_{val} = [\![\varphi_1]\!]_{val} \wedge [\![\varphi_2]\!]_{val}$, $[\![\varphi_1 \Rightarrow \varphi_2]\!]_{val} = [\![\neg\varphi_1 \vee \varphi_2]\!]_{val}$. ▪

Note that if a variable $x_i \notin \text{supp}(val)$ appears in the formula φ, then $[\![\varphi]\!]_{val}$ is undefined.

For a set of data variables X, let $[\![\varphi]\!] = \{val\colon X \to V \mid [\![\varphi]\!]_{val} = \texttt{true}\}$ denote the set of valuation where the formula φ holds. This now lets us define *formula abstraction.*

Definition 10 (Formula abstraction). Formula abstraction over the data variables X, the signature $\langle \Sigma, arity \rangle$, and the interpretation $\mathcal{I}$ is $\mathbf{Form}(X, \Sigma, arity, \mathcal{I}) = \langle \mathcal{S}, \Phi_{\langle \Sigma, arity \rangle}, \Rightarrow, \gamma \rangle$, where

- $\mathcal{S}$ is the set of all *total valuations* $\{val\colon X \to V\}$,
- $\varphi_1 \Rightarrow \varphi_2$ iff $[\![\varphi_1 \Rightarrow \varphi_2]\!] = \mathcal{S}$,
- $\gamma(\varphi) = [\![\varphi]\!]$.

For brevity, we will omit the signature and the interpretation if they are clear from context and use the notation $\mathbf{Form}(X)$. ▪

3.2.4 Zone abstraction

Due to clock variables being real-valued, the state spaces of timed automata are infinite. In order to make verification of timed systems feasible, clock valuations have to be abstracted. For this purpose we use *zone abstraction* [1].

Definition 11 (Zone). A zone is a set of clock constraints. For a set of clock variables $\mathcal{C}$ and a zone Z, let $(\!|Z|\!)$ denote the set of clock valuations $\{cval : \mathcal{C} \to \mathbb{R}_{\geq 0}\}$ such that $cval$ satisfies all clock constraints in Z, i.e. $(\!|Z|\!)$ is the solution set of the conjunction of clock constraints in Z. ▪

Note that two clock valuations $c_1, c_2 \in (\!|Z|\!)$ are indistinguishable by the clock constraints in Z, however, this does not necessarily hold for an arbitrary clock constraint. Nonetheless, for the purpose of efficient model checking it is sufficient for these clock valuations to be indistinguishable only by a finite set of clock constraints, e.g. the set of clock constraints that appear in the model being analysed.

Definition 12 (Zone abstraction). Zone abstraction over the clock variables $\mathcal{C}$ uses the domain $\mathbf{Zone}(\mathcal{C}) = \langle \mathcal{S}, \mathcal{Z}, \sqsubseteq, \gamma \rangle$, where

- $\mathcal{S}$ is the set of all clock valuations $\{cval\colon \mathcal{C} \to \mathbb{R}_{\geq 0}\}$,
- $\mathcal{Z}$ is the set of all zones and the *inconsistent zone* $\bot$,
- $\gamma(Z) = (\!|Z|\!)$,
- $Z_1 \sqsubseteq Z_2$ iff $(\!|Z_1|\!) \subseteq (\!|Z_2|\!)$. ▪

Zones contain clock constraints of the form $c_1 \sim n$ or $c_1 - c_2 \sim n$ where $c_1, c_2 \in \mathcal{C}$, $n \in \mathbb{N}$ and $\sim \in \{<, \leq, =, \geq, >\}$. By introducing a reference clock c_0 with constant zero value, all clock constraints can be written uniformly as $c_1 - c_2 \prec n$ where $c_1, c_2 \in \mathcal{C} \cup \{c_0\}$, $n \in \mathbb{Z}$ and $\prec \in \{<, \leq\}$. This enables the representation of a zone as a *difference bound matrix* (DBM) [1].

Definition 13 (Difference bound matrix). For a timed automaton with clock variables $c_1, c_2, ..., c_k$, a difference bound matrix (DBM) is a square matrix D of dimension $(k+1) \times (k+1)$ such that an element of this matrix is either $D_{ij} = (n, \prec)$ where $n \in \mathbb{Z}$ and $\prec \in \{<, \leq\}$, representing the clock constraint $c_i - c_j \prec n$, or $D_{ij} = \infty$, indicating the absence of a bound. ▪

DBMs enable the effective representation of zones and the efficient execution of zone operations used in reachability analysis.

3.2.5 Product abstraction

For systems that contain variables of different kinds (e.g. a timed automaton with data and clock variables) we often cannot use a single abstraction for all variables, as one abstraction may not be applicable for all of them. In these cases we use two or more different domains and use product abstraction to handle them as one.

Definition 14 (Product abstraction). The product of two abstractions $\mathbf{D_1} = \langle \mathcal{S}_1, \mathcal{D}_1, \sqsubseteq_1, \gamma_1 \rangle$ and $\mathbf{D_2} = \langle \mathcal{S}_2, \mathcal{D}_2, \sqsubseteq_2, \gamma_2 \rangle$ is the domain $\mathbf{Prod}(\mathbf{D_1}, \mathbf{D_2}) = \langle \mathcal{S}, \mathcal{D}, \sqsubseteq, \gamma \rangle$, where

- $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$,
- $\mathcal{D} = \mathcal{D}_1 \times \mathcal{D}_2$,
- $(d_1, d_2) \sqsubseteq (d_1', d_2')$ iff $d_1 \sqsubseteq_1 d_1' \wedge d_2 \sqsubseteq_2 d_2'$, where $d_1, d_1' \in \mathcal{D}_1$ and $d_2, d_2' \in \mathcal{D}_2$,
- $\gamma(d_1, d_2) = (\gamma_1(d_1), \gamma_2(d_2))$, where $d_1 \in \mathcal{D}_1$ and $d_2 \in \mathcal{D}_2$. ▪

3.2.6 Location abstraction

The goal of reachability analysis is checking the reachability of an error location. Therefore, locations of the automaton also have to be tracked. Locations are always explicitly tracked, as the sets of possible transitions (and thus all guards, actions, invariants, etc.) are dependent on the location of the automaton. Location abstraction can be viewed as a special form of product abstraction.

Definition 15 (Location abstraction). Location abstraction with the abstract domain $\mathbf{D} = \langle \mathcal{S}, \mathcal{D}, \sqsubseteq, \gamma \rangle$ is $\mathbf{Loc}(\mathbf{D}) = \langle \mathcal{S}_L, \mathcal{D}_L, \subseteq_L, \gamma_L \rangle$, where

- $\mathcal{S}_L = L \times \mathcal{S}$,
- $\mathcal{D}_L = L \times \mathcal{D}$,
- $(l, d) \sqsubseteq_L (l', d')$ iff $l = l' \wedge d \sqsubseteq d'$,
- $\gamma_L(l, d) = (l, \gamma(d))$. ▪

3.3 General lazy abstraction algorithm

Our algorithm generalizes the abstraction-based lazy algorithm presented in [8]. Given a target state (e.g. a state containing an error location), reachability analysis is performed by constructing an *abstract reachability tree* (ART) of reachable states, rooted at the initial state, and checking whether the error state is present in the resulting graph.

The general algorithm labels the nodes of the ART by labels from two abstract domains. Let $\mathbf{D}_c = \langle \mathcal{S}, \mathcal{D}_c, \sqsubseteq_c, \gamma_c \rangle$ denote the concrete labelling domain and $\mathbf{D}_a = \langle \mathcal{S}, \mathcal{D}_a, \sqsubseteq_a, \gamma_a \rangle$ the abstract labelling domain.

Definition 16 (Abstract reachability tree). An *abstract reachability tree* (ART) for a timed automaton $\mathcal{A}$ is a tuple $ART = \langle N, E, n_0, d_c, d_a, t, C \rangle$, where

- (N, E) is a directed rooted tree of nodes N and edges $E \subset N \times N$, with root node $n_0 \in N$,
- $d_c : N \to \mathcal{D}_c$ is the labelling of nodes by concrete labels,
- $d_a : N \to \mathcal{D}_a$ is the labelling of nodes by abstract labels,
- $t : E \to T$ is the labelling of edges by transitions,
- $C \subseteq N \times N$ is the set of covering edges, $(n, n') \in C$ expressing that all states reachable from n are also reachable from n'. ▪

An important restriction on the abstraction $\mathbf{D}_c$ is that the abstract states do not produce unreachable states when concretized. Still, it is an abstract domain, as its abstract states may represent multiple concrete states (e.g. zones). Abstract states in $\mathbf{D}_a$ over-approximate the abstract states in $\mathbf{D}_c$ and likely include actually unreachable states as well. The purpose of this second abstract domain in the lazy algorithm is to have an abstraction that is as coarse as possible, for as long as possible.

Given an abstract state d and a transition t, an abstract post-image operator (post-image operator for an abstract domain) computes a set of states that is the strongest postcondition of d with respect to t. The abstract post-image $post'_t(d)$ has the following properties:

- $\bigcup_{s \in \gamma(d)} post_t(s) \subseteq \bigcup_{d' \in post'_t(d)} \gamma(d')$ (the concretizations of all possible abstract post-images contain all concrete post-images of the concretized abstract state),
- it is the strongest possible postcondition for the given abstract state and transition.

Let $\overline{post}$ denote the abstract post-image operator for $\mathbf{D}_c$. The $\overline{post}$ operator does not introduce unreachable states in the post-image. We refer to post-image operators that have this property as *exact* post-image operators. Note that the consequence of the $\overline{post}$ operator being exact is that for all nodes n of the ART there exists a run of the automaton to all states $s \in d_c(n)$.

Let $\widetilde{post}$ denote the abstract post-image operator for $\mathbf{D}_a$. The $\widetilde{post}$ operator is not exact, post-images computed by this operator contain all reachable states in accordance with the definition of abstract post-image operators, but they may also contain unreachable states.

With abstract domains $\mathbf{D_1} = \langle \mathcal{S}_1, \mathcal{D}_1, \sqsubseteq_1, \gamma_1 \rangle$ and $\mathbf{D_2} = \langle \mathcal{S}_2, \mathcal{D}_2, \sqsubseteq_2, \gamma_2 \rangle$, $\gamma_{1\to 2} : \mathcal{D}_1 \to \mathcal{D}_2$ is a concretization function that maps abstract states in $\mathcal{D}_1$ to abstract states in $\mathcal{D}_2$, i.e. $\gamma_{1\to 2}(d)$ is $d \in \mathcal{D}_1$ represented in the $\mathbf{D}_2$ domain.

Let $d_1 \vdash d_2$ denote that d_1 *proves* d_2. The proves operator is a reflexive and transitive binary relation with the following semantics: $d_1 \vdash d_2$ iff $\gamma_{1\to 2}(d_1) \sqsubseteq_2 d_2$, where $d_1 \in \mathcal{D}_1$, $d_2 \in \mathcal{D}_2$ from abstract domains $\mathbf{D_1} = \langle \mathcal{S}_1, \mathcal{D}_1, \sqsubseteq_1, \gamma_1 \rangle$ and $\mathbf{D_2} = \langle \mathcal{S}_2, \mathcal{D}_2, \sqsubseteq_2, \gamma_2 \rangle$.

The ART is constructed gradually during the exploration of the state space. It is constructed and labelled in such a way that its nodes over-approximate the set of reachable states. To ensure this, the labelling has to conform to the following properties:

1. $s_0 \in \gamma_c(d_c(n_0))$ (the concrete label of the initial node over-approximates the initial state),
2. $d_c(n) \vdash d_a(n)$ (the concrete label of any node proves the abstract label of the same node),
3. (a) for all edges $e = (n, n') \in E$ there exists a concrete label $d'_c \in \overline{post}_{t(e)} d_c(n)$ such that $d_c(n') = d'_c$ (a successor concrete label present in the ART is a post-image of its parent),
   (b) for all transitions t enabled from an expanded node n and all concrete labels $d'_c \in \overline{post}_t d_c(n)$, $d'_c \neq \bot$ there exists an edge $(n, n') \in E$ labelled by t such that $d_c(n') = d'_c$ (the ART contains all post-images of the concrete label of an expanded node),
4. (a) for all edges $e = (n, n') \in E$ there exists an abstract label $d'_a \in \widetilde{post}_{t(e)} d_a(n)$ such that $d'_a \sqsubseteq_a d_a(n')$ (a successor abstract label present in the ART over-approximates a post-image of its parent),
   (b) for all transitions t enabled from an expanded node n and all abstract labels $d'_a \in \widetilde{post}_t d_a(n)$ there exists an edge $(n, n') \in E$ labelled by t such that $d'_a \sqsubseteq_a d_a(n')$ (the ART contains an over-approximating abstract label for all post-images of the abstract label of an expanded node),
5. $(n, n') \in C \Rightarrow d_a(n) \sqsubseteq_a d_a(n')$ (if a covering edge is present then the covering node over-approximates the covered node on their abstract labels).

Before presenting the general lazy abstraction algorithm, the functions *target* and *initabstr* have to be defined. Function *target*(d) returns `true` iff d is the target state (e.g. an error state, a state containing an error location).

Function *initabstr*(d) returns an initial abstract label for the concrete label d, in compliance with the relevant labelling properties of ARTs described above (2. and 4.). A suitable initial abstraction can be, for example, the top element of the abstract labelling domain, or, if we do not wish to use any abstractions, it can be equivalent to the concrete label as well.

The reachability of an error location is checked by the general lazy abstraction algorithm. The algorithm builds an ART of nodes N, edges E and covering edges C, while maintaining a list of passed (expanded) nodes and a waitlist. In the waitlist, the reached but not yet closed or expanded nodes are queued. It is initialized by the initial node that consists of the initial state represented in the concrete labelling domain and a suitable initial abstract label created by *initabstr*.

Algorithm 1 Reachability of an error state

Ensure: return SAFE iff the error state is not reachable in the automaton $\mathcal{A}$

1: **function** CHECK($\mathcal{A}$)
2: $n_0 \leftarrow$ node(d_c, d_a) such that $s_0 \in \gamma_c(d_c)$ and $d_a =$ initabstr(d_c)
3: $N \leftarrow \{n_0\}$
4: $E \leftarrow \emptyset$
5: $C \leftarrow \emptyset$
6: $waitlist \leftarrow \{n_0\}$
7: $passed \leftarrow \emptyset$
8: **while** $n \in waitlist$ for some n **do**
9: $waitlist \leftarrow waitlist \backslash \{n\}$
10: CLOSE(n)
11: **if** n is not covered $\wedge$ EXPAND(n) **then**
12: **return** UNSAFE
13: **return** SAFE

Algorithm 2 Attempt at covering a node

Ensure: the labelling of nodes satisfies the properties of an ART

1: **procedure** CLOSE(n)
2: **for all** $n' \in passed$ **do**
3: **if** $d_c(n) \vdash d_a(n')$ **then**
4: $C \leftarrow C \cup \{(n, n')\}$
5: COVER(n, n')

Algorithm 3 Expand a node

Ensure: return `true` iff the error state is reachable as an immediate successor of n
Ensure: the labelling of nodes satisfies the properties of an ART

1: **function** EXPAND(n)
2: **for all** $t \in T$ enabled from $d_c(n)$ **do**
3: **for all** $d'_c \in \overline{post}_t(d_c(n))$ **do**
4: **if** $\gamma_c(d'_c) = \emptyset$ **then**
5: DISABLE(n, t)
6: **else if** target(d'_c) **then**
7: **return** `true`
8: **else**
9: $d'_a \leftarrow$ initabstr(d'_c)
10: $n' \leftarrow$ node(d'_c, d'_a)
11: $N \leftarrow N \cup \{n'\}$
12: $E \leftarrow E \cup \{$edge$(n, n', t)\}$
13: $waitlist \leftarrow waitlist \cup \{n'\}$
14: $passed \leftarrow passed \cup \{n\}$
15: **return** `false`

Adjust the abstract labels after coverage

Require: $d_c(n) \vdash d_a(n')$
Ensure: the labelling of nodes satisfies the properties of an ART
procedure COVER(n, n')

Disable a transition to an infeasible state

Require: transition t is infeasible from n
Ensure: transition t is disabled from $d_a(n)$
Ensure: the labelling of nodes satisfies the properties of an ART
procedure DISABLE(n, t)

Algorithm 4 Strengthen the abstract label of a node

Ensure: the labelling of nodes satisfies the properties of an ART
1: **procedure** STRENGTHEN(n, d)
2: $d_a(n) \leftarrow d$
3: **for all** n' such that $(n', n) \in C$ and $d_a(n') \not\sqsubseteq d_a(n)$ **do**
4: $C \leftarrow C \backslash \{(n', n)\}$
5: $waitlist \leftarrow waitlist \cup \{n\}$

Function CHECK consumes nodes from the waitlist one by one. First it attempts to cover the node n by calling to CLOSE. If it is unsuccessful, then the node has to be expanded. If the result of procedure EXPAND is true, then by its contract the error state is reachable by a successor of n, the exploration of the state space is stopped and the automaton $\mathcal{A}$ is deemed unsafe. If the waitlist becomes empty without encountering the error state in the process, then $\mathcal{A}$ is deemed safe.

Procedure CLOSE covers the node n, if it is possible. It iterates through the passed nodes, and checks the labellings. If the coverage is possible (the concrete label of n proves the abstract label of the other node n'), then it adds a covering edge and strengthens the abstract label of n by calling to COVER in order to enforce coverage on the abstract labels as well.

Function EXPAND creates the successor nodes for n, puts them in the waitlist and puts n in the list of passed nodes. EXPAND iterates through each transition t enabled from $d_c(n)$ and each post-image of $d_c(n)$ with respect to t. First, it is checked, whether the post-image can represent any actual states. If not (i.e. its concretization gives an empty set), then the abstract label of n is strengthened by calling to DISABLE to disable the transition to the infeasible state. Otherwise, the successor represents states that are actually reachable in $\mathcal{A}$. If the state is the target state, then the creation of successor nodes is interrupted and `true` result is returned. Otherwise, a new node is created from the post-image and a suitable initial abstract label.

Procedures COVER and DISABLE perform abstraction refinement, they may strengthen the abstract labelling of some nodes. Strengthening the abstract label of n by d is done by procedure STRENGTHEN. After setting the abstract label to d, all nodes covered by n have to be inspected: when the requirement for coverage no longer holds, then the covering edge is removed, and the previously covered node is put back in the waitlist. For this reason, coverage has to be checked once more in CHECK after calling to CLOSE.

Note that how CLOSE and STRENGTHEN work is a key to understanding the lazy abstraction algorithm. Informally, the initial abstraction is very coarse (the initial abstract labels are indeed very abstract), so lots of nodes are covered at first and not many nodes are put in the waitlist. When as a consequence of updating an abstract label it turns out that a node cannot cover an other one (there are some runs of the automaton that the covering node does not cover), only then is a covered node uncovered and put in the waitlist for later processing. Hence, it is a lazy algorithm.

3.3.1 Proof of correctness

For now let us assume that the contracts of procedures COVER and DISABLE are correct. These assumptions are proven later in this work. Now we show that the contracts of CHECK, CLOSE, EXPAND and STRENGTHEN are correct.

CHECK: When the procedure returns UNSAFE, the following conditions have to hold: there is a node n such that it is not covered and EXPAND(n) returns true. By contract EXPAND(n) returns true iff the target state is reachable as an immediate successor of n, and here n is also reachable, therefore the target state is reachable in $\mathcal{A}$. At all times, a node is either covered, passed or in the waitlist. When the reachability of the target state is discovered in EXPAND, the algorithm stops immediately, it does not put any new nodes in the waitlist, so a target state is never put into the waitlist. Because passed and covered nodes are nodes removed from the waitlist, they also do not contain the target state. When CHECK returns SAFE, the waitlist is empty, i.e. all nodes are either passed or covered, which means that their concrete labels has already been checked and they do not contain the target state, hence the target state is not reachable in $\mathcal{A}$.

CLOSE: The call to COVER ensures that all labelling properties of the ART are satisfied, even if a new covering edge is added.

EXPAND: $\gamma_c(d_c(n)) \neq \emptyset$, as all nodes (except for n_0) are created in EXPAND, where nodes labelled by d such that $\gamma(d) = \emptyset$ are never created. From the $\overline{post}$ operator being exact it follows, that $d_c(n)$ is a reachable state in the automaton. When EXPAND returns `true`, the following conditions have to hold: there exists a transition t and a state $s_c \in \overline{post}_t(d_c(n))$ such that $\gamma_c(s_c) \neq \emptyset$ and $target(s_c)$ returns true. This means that the concretization of the post-image yields a non-empty set and because $\overline{post}$ is exact, the target state is reachable in the automaton as well, as an immediate successor of n. When EXPAND returns `false`, all successors with all transitions taken into consideration either do not reach the target state or their concretization is empty (otherwise EXPAND would return `true`), meaning that the target state is not reachable in $\mathcal{A}$ in the immediate successors of n. It is easy to verify that the creation of new nodes in EXPAND also conforms to the labelling properties of an ART.

STRENGTHEN: The ART property $d_a(n') \sqsubseteq_a d_a(n)$ for covering edges $(n', n) \in C$ is satisfied before the call to STRENGTHEN. In STRENGTHEN, the covering edges where $d_a(n') \sqsubseteq_a d_a(n)$ no longer holds are removed from C, so the 5. ART labelling property still holds after the call. Later in this work we show that STRENGTHEN is used in such a way that preserves the other ART labelling properties as well.

The initial graph that consists of only n_0 is clearly a proper ART. The contracts of the procedures imply that the tree rooted at n_0 and built by the algorithm is indeed a proper ART.

Chapter 4

Interpolation techniques

A possible implementation of procedures COVER and DISABLE is based on strengthening the abstract labels of nodes by *interpolants*.

The interpolation algorithms for lazy abstraction use an other abstract domain in addition to $\mathbf{D}_c$ and $\mathbf{D}_a$. This *interpolation domain* enables further configuration of the algorithm, and with a well chosen interpolation domain the abstract labelling can remain coarser as well, leading to a faster solution. As an example, when the concrete labelling domain uses identity abstraction and the abstract labelling domain uses explicit value abstraction, the interpolation domain can use formula abstraction. The interpolation domain is denoted by $\mathbf{D}_i = \langle \mathcal{S}, \mathcal{D}_i, \sqsubseteq_i, \gamma_i \rangle$.

Let $d_1 \dashv d_2$ denote that d_1 *refutes* d_2. The refutes operator is a binary relation with the following semantics: if $d_1 \dashv d_2$, then $\gamma_1(d_1) \cap \gamma_2(d_2) = \emptyset$, where $d_1 \in \mathcal{D}_1$ and $d_2 \in \mathcal{D}_2$ from abstract domains $\mathbf{D_1} = \langle \mathcal{S}_1, \mathcal{D}_1, \sqsubseteq_1, \gamma_1 \rangle$ and $\mathbf{D_2} = \langle \mathcal{S}_2, \mathcal{D}_2, \sqsubseteq_2, \gamma_2 \rangle$.

Definition 17 (Interpolant). Given $A \in \mathbf{D}_a$ and $B \in \mathbf{D}_i$ where $A \dashv B$, an interpolant for (A, B) is $I = \mathrm{ITP}(A, B) \in \mathbf{D}_a$ such that $A \vdash I$ and $I \dashv B$. ▪

Note that according to the definition of interpolants, A can be a trivial interpolant for (A, B). However, as the abstract labels are strengthened by these interpolants, an efficient implementation needs to produce interpolants that are as abstract as possible. For example, in zone abstraction a further property of an interpolant can be that it may only contain constraints on clock variables that are constrained in both A and B.

For the interpolation algorithms, the computation of weakest existential preconditions is needed, done by a *pre-image* operator. A pre-image $pre_t(s)$ is the set of states s' such that some state in $post_t(s')$ is s.

Abstract pre-image operators (pre-image operators for abstract domains) are defined similarly to abstract post-image operators. Given an abstract state d and a transition t, an abstract pre-image operator computes a set of states that is the weakest existential precondition of d with respect to t. The abstract pre-image $pre'_t(d)$ has the following properties:

- $\bigcup_{s \in \gamma(d)} pre_t(s) \subseteq \bigcup_{d' \in pre'_t(d)} \gamma(d')$ (the concretizations of all possible abstract pre-images contain all concrete pre-images of the concretized abstract state),
- it is the weakest possible precondition for the given abstract state and transition.

We refer to abstract pre-image operators that do not over-approximate the weakest precondition as *exact* pre-image operators. The pre-image operator for the interpolation domain is exact and it is denoted by $\widehat{pre}$.

For the interpolation algorithms presented in this chapter to work correctly, the refutes operator has to form the following connection between pre- and post-images:
$(d_1 \dashv d_{pre}$ for all $d_{pre} \in \widehat{pre}_t(d_2)) \Leftrightarrow (d_{post} \dashv d_2$ for all $d_{post} \in \widetilde{post}_t(d_1))$

In contrast with the definition of the proves operator, $\gamma_1(d_1) \cap \gamma_2(d_2) = \emptyset \Rightarrow d_1 \dashv d_2$ does not necessarily hold for the refutes operator, as it could conflict with the above property. The section about using the interpolation techniques with explicit value abstraction elaborates on this and provides an example as well. Regardless, we can state that the refutes operator is commutative and if $d_1 \dashv d_2$ and $d_1' \sqsubseteq d_1$, then $d_1' \dashv d_2$.

In the following, the abstraction refinement algorithms are presented.

Algorithm 5 Disable a transition to an infeasible state

Require: transition t is infeasible from n
Ensure: transition t is disabled from $d_a(n)$
Ensure: the labelling of nodes satisfies the properties of an ART
1: **procedure** DISABLE(n, t)
2: $\tau \leftarrow \top \in \mathcal{D}_i$
3: **for all** $B \in \widehat{pre}_t(\tau)$ **do**
4: BLOCK(n, B)

Algorithm 6 Adjust the abstract labels after coverage

Require: $d_c(n) \vdash d_a(n')$
Ensure: the labelling of nodes satisfies the properties of an ART
1: **procedure** COVER(n, n')
2: $\rho \leftarrow \gamma_{a \to i}(d_a(n'))$
3: **for all** $B \in \neg\rho$ **do**
4: BLOCK(n, B)

Remove states from the abstract label of a node

Require: $d_c(n) \dashv B$
Ensure: $d_a(n) \dashv B$
Ensure: the labelling of nodes satisfies the properties of an ART
procedure BLOCK(n, B)

In procedure DISABLE, all pre-images of the top element of the interpolation domain with respect to transition t are removed from the abstract label of the node by calling to procedure BLOCK. By the contract of BLOCK, $d_a(n) \dashv B$ for all $B \in \widehat{pre}_t(\tau)$. By the connection of the $\widehat{pre}$ and $\widetilde{post}$ operators, $d_{post} \dashv \tau$ for all $d_{post} \in \widetilde{post}_t(d_a(n))$. Since the post-images of $d_a(n)$ refute the top element, from the definition of the refutes operator we have that $\gamma_a(d_{post}) = \emptyset$ for all $d_{post} \in \widetilde{post}_t(d_a)$, which means that t is indeed disabled from $d_a(n)$ and the labelling properties of the ART are preserved by the contract of BLOCK.

Procedure COVER removes from the abstract label of the covered node all states that are not in the abstract label of the covering node by calling to BLOCK, which ensures that the labelling properties of the ART are preserved.

Both DISABLE and COVER rely on the procedure BLOCK that performs the main part of the abstraction refinement. It strengthens the abstract label of the node, and additionally,

to preserve the correctness of abstract labelling of the ART, it also has to strengthen the abstract labels of multiple nodes on the path to the node.

In order to be able to use the interpolating abstraction refinement algorithms presented later in this chapter, some limitations have to be introduced on the post-image operators. Both $\overline{post}$ and $\widetilde{post}$ must produce only one post-image. Moreover, $\widetilde{post}_t(\gamma_{c\to a}(d)) = \overline{post}_t(d)$, i.e. the abstract post-image operators $\overline{post}$ and $\widetilde{post}$ must produce the same post-image for a concrete label (resp. its representation in the abstract labelling domain).

In the following sections two algorithms for BLOCK are presented. They both move backward in the ART from the node that BLOCK was initially called with, however, they differ in the order in which the interpolants are computed and the abstract labels strengthened.

4.1 Backward interpolation

Backward interpolation is a possible method of abstraction refinement. It first strengthens the abstract label of the node by a suitable interpolant, then refines the abstract label of its parent node in the same way, moving backward towards the root of the ART until it reaches the root node or a node that already satisfies the requirement.

Algorithm 7 Block with backward interpolation

Require: $d_c(n) \dashv B$
Ensure: $d_a(n) \dashv B$
Ensure: the labelling of nodes satisfies the properties of an ART

```
 1: procedure BLOCK_BW(n, B)
 2:     if d_a(n) ⊣ B then
 3:         return
 4:     A ← γ_{c→a}(d_c(n))
 5:     I ← ITP(A, B)
 6:     d_new ← d_a(n) ⊓_a I
 7:     STRENGTHEN(n, d_new)
 8:     if (p, n) ∈ E for some p then
 9:         t ← t((p, n))
10:         I_i ← γ_{a→i}(I)
11:         for all B' ∈ ¬I_i do
12:             for all B_pre ∈ pre^_t(B') do
13:                 BLOCK_BW(p, B_pre)
```

For proving the correctness of BLOCK_{BW}, we first show that the contract of the procedure is correct. The procedure does not change the concrete labelling and procedure STRENGTHEN ensures the correctness of covering edges in the ART, so we only have to show that BLOCK_{BW} maintains the 2. and 4. labelling properties of the ART ($d_c(n) \vdash d_a(n)$ and the consecution of abstract labels). Then, we show that the interpolant in line 5 can be computed and that the recursive call to BLOCK_{BW} in line 13 is possible.

If initially $d_a(n) \dashv B$, then the procedure is a no-op and all postconditions trivially hold. Because of this, we concentrate only on the case when $d_a(n) \dashv B$ does not hold at the time of calling BLOCK_{BW}.

The new abstract label refutes B ($d_a(n) \dashv B$): For the interpolant I computed in line 5 we have $I \dashv B$ by the definition of interpolants. From this it follows by the properties of the refutes operator that $d \sqcap_a I \dashv B$ for any $d \in \mathcal{D}_a$ where $d \sqcap_a I$ denotes the meet of d and I as two elements of the lattice of abstract labels. As the abstract label of n is updated by $d_{new} = d_a(n) \sqcap_a I$, the new labelling satisfies the postcondition.

The concrete label of n proves its abstract label ($d_c(n) \vdash d_a(n)$): For the interpolant I computed in line 5 we have $d_c(n) \vdash I$ by the definition of interpolants. Initially $d_c(n) \vdash d_a(n)$ by the labelling property of the ART. From these it follows that $d_c(n) \vdash d_a(n) \sqcap_a I$. The new abstract label of n is $d_{new} = d_a(n) \sqcap_a I$, so the ART labelling property is maintained.

Consecution of abstract labels: The edges of the ART are not modified, only the labelling, so we only have to show that $\widetilde{post}_t(d_a(p)) \sqsubseteq_a d_a(n)$ holds for the new labelling. In line 13 we have $d_a(p) \dashv B_{pre}$ for all $B_{pre} \in \widehat{pre}_t(B')$ for all $B' \in \neg(\gamma_{a \to i}(I))$ by contract. By the connection of $\widetilde{post}$ and $\widehat{pre}$ it follows that $\widetilde{post}_t(d_a(p)) \dashv B'$ for all $B' \in \neg(\gamma_{a \to i}(I))$. This implicates that $\widetilde{post}_t(d_a(p)) \sqsubseteq_a I$. Initially $\widetilde{post}_t(d_a(p)) \sqsubseteq_a d_a(n)$ by the labelling property of the ART. From these we have $\widetilde{post}_t(d_a(p)) \sqsubseteq_a d_a(n) \sqcap_a I$ and the new abstract label is $d_{new} = d_a(n) \sqcap_a I$, so $\widetilde{post}_t(d_a(p)) \sqsubseteq_a d_{new}$, the ART labelling properties are maintained.

The interpolant can be computed: For computing an interpolant for A and B, $A \dashv B$ must hold. By the precondition of BLOCK$_{BW}$ $d_c(n) \dashv B$, so the interpolant can be computed.

The recursive call to BLOCK$_{BW}$ is possible: Calling BLOCK$_{BW}$ is possible if the precondition of the procedure is met. From $d_c(n) \dashv B$ and $\overline{post}_t(d_c(p)) = d_c(n)$ we have $\widetilde{post}_t(\gamma_{c \to a}(d_c(p))) \dashv B$ by the previously introduced properties of abstract post-image operators. By the connection of $\widetilde{post}$ and $\widehat{pre}$ it follows that $d_c(p) \dashv B_{pre}$ for all $B_{pre} \in \widehat{pre}_t(B)$, hence BLOCK$_{BW}$ can be called with parameters p and B_{pre}.

4.2 Forward interpolation

An other method of abstraction refinement is forward interpolation. In this case the backward step in the ART precedes the strengthening of the abstract label. This means that it first moves backward towards the root of the ART until it reaches the root node or a node that already satisfies the requirement. Then, it moves *forward* on the same path, while computing the interpolants and strengthening the abstract labels of nodes.

For proving the correctness of BLOCK$_{FW}$ we have to show that the contract of BLOCK$_{FW}$ is correct, that the interpolant in line 11 or 15 can be computed and that calling BLOCK$_{FW}$ in line 8 is possible. Regarding the labelling properties of the ART, for the same reason as in BLOCK$_{BW}$, we only have to show that BLOCK$_{FW}$ maintains the 2. and 4. properties ($d_c(n) \vdash d_a(n)$ and the consecution of abstract labels).

If initially $d_a(n) \dashv B$, then the abstract label of n is returned. This clearly satisfies all postconditions of the procedure, therefore, similarly to BLOCK$_{BW}$, we concentrate only on the case when $d_a(n) \dashv B$ does not hold initially.

The return value of BLOCK$_{FW}$ refutes B ($I \dashv B$): If n is the root node, then the returned value is the interpolant for $d_c(n)$ and B, which refutes B by the definition of interpolants. Otherwise, for every interpolant i computed in line 11 we have $i \dashv B$. I is the meet of these interpolants, so $I \sqsubseteq_i i$ for all i. From this and $i \dashv B$ it follows that $I \dashv B$ by the properties of the refutes operator.

Algorithm 8 Block with forward interpolation

Require: $d_c(n) \dashv B$
Ensure: $I \dashv B$
Ensure: $d_a(n) \vdash I$
Ensure: $d_a(n) \dashv B$
Ensure: the labelling of nodes satisfies the properties of an ART

```
1: procedure BLOCK_FW (n, B) returns I
2:     if d_a(n) ⊣ B then
3:         return d_a(n)
4:     if (p, n) ∈ E for some p then
5:         t ← t(p, n)
6:         I ← ⊤ ∈ D_a
7:         for all B_pre ∈ pre^_t(B) do
8:             A_pre ← BLOCK_FW(p, B_pre)
9:             A ← post~_t(A_pre)
10:            if A ⊣ B then
11:                i ← ITP(A, B)
12:                I ← I ⊓_a i
13:    else
14:        A ← γ_{c→a}(d_c(n))
15:        I ← ITP(A, B)
16:    d_new ← d_a(n) ⊓_a I
17:    STRENGTHEN(n, d_new)
18:    return I
```

The new abstract label of n proves the return value of BLOCK$_{FW}$ ($d_a(n) \vdash I$): $d_a(n)$ is the meet of the original abstract label and I, so $d_a(n) \vdash I$ clearly holds.

The new abstract label refutes B ($d_a(n) \dashv B$): $d_a(n) \dashv B$ is the direct consequence of the previous two proven statements and the properties of the refutes operator.

The concrete label of n proves its abstract label ($d_c(n) \vdash d_a(n)$): If n has a parent node, then in every iteration of the loop in lines 7-12 $d_a(p) \vdash A_{pre}$ holds by the already proven part of the contract of BLOCK$_{FW}$. By the labelling property of the ART we have $d_c(p) \vdash d_a(p)$. From these it follows that $d_c(p) \vdash A_{pre}$, and by the monotony of the post operator $\widetilde{post}_t(d_c(p)) \vdash \widetilde{post}_t(A_{pre})$, which is equivalent to $d_c(n) \vdash A$. For every A it holds that $A \vdash i$ by the definition of interpolants. This means that $d_c(n) \vdash i$ for every interpolant i computed in line 11. From this it follows that $d_c(n) \vdash I$. If n is the root node, the same holds simply by the definition of interpolants. From this point onward the proof is the same as for BLOCK$_{BW}$.

Consecution of abstract labels: As in BLOCK$_{BW}$, the edges of the ART are not modified, only the labelling, so we only have to show that $\widetilde{post}_t(d_a(p)) \sqsubseteq_a d_a(n)$ holds for the new labelling. In every iteration of the loop in lines 7-12 $d_a(p) \vdash A_{pre}$ holds. By the monotony of the post operator $\widetilde{post}_t(d_a(p)) \vdash \widetilde{post}_t(A_{pre})$, which is equivalent to $\widetilde{post}_t(d_a(p)) \vdash A$. As shown previously, $A \vdash i$ for every interpolant i computed in line 11. From these it follows that $\widetilde{post}_t(d_a(p)) \vdash i$ for all i, so $\widetilde{post}_t(d_a(p)) \vdash I$ holds, from this it follows that $\widetilde{post}_t(d_a(p)) \sqsubseteq_a I$. Initially $\widetilde{post}_t(d_a(p)) \sqsubseteq_a d_a(n)$, so the new abstract label $d_{new} = d_a(n) \sqcap_a I$ also over-approximates $\widetilde{post}_t(d_a(p))$, maintaining the ART labelling property.

The interpolant can be computed: It is clear that the interpolant can be computed in the case of n not being the root node. If n is the root node, then the interpolant for $d_c(n)$ and B can be computed too, as the precondition of the procedure is $d_c(n) \dashv B$.

The recursive call to BLOCK$_{FW}$ is possible: The proof is the same as for BLOCK$_{BW}$.

4.3 Interpolation techniques in practice

4.3.1 Explicit value abstraction with formula abstraction

A simple abstraction for data variables is explicit value abstraction, when the abstract labels are partial valuations. When using this kind of abstraction, the straightforward way of representing concrete labels is by total valuations.

The efficiency of the presented algorithms is dependent on the ability of computing interpolants that are coarse enough. For an interpolant to be as coarse as possible, the opposite should hold for the state it refutes. However, partial valuations do not allow for fine representation of states, hence explicit value abstraction is not a suitable abstraction for the interpolation domain. Therefore, we use formula abstraction for this.

A suitable *initabstr* function for explicit value abstraction is such that it always gives the top element of the abstract labelling domain, which in this case is the empty partial valuation $\top$. This initial abstract labelling complies with the labelling properties of the ART: it always over-approximates both the concrete label and the post-image of the abstract label of its parent node.

To use these interpolation techniques, one must provide operators $\vdash$ (proves) and $\dashv$ (refutes). For the proves operator only a concretization function $\gamma_{c\to a}$ is needed. The $\gamma_{c\to a}$ function provides the representations of concrete labels in the abstract labelling domain. In the case of total and partial valuations this is quite simple: total valuations are partial valuations where the values of all variables are tracked.

The refutes operator has to satisfy the previously presented property:
$(d_1 \dashv d_{pre}$ for all $d_{pre} \in \widehat{pre}_t(d_2)) \Leftrightarrow (d_{post} \dashv d_2$ for all $d_{post} \in \widetilde{post}_t(d_1))$.

Because of this, it is incorrect to state that $v \dashv \varphi$ holds for a partial valuation v and a formula φ if $\gamma(v) \cap [\![\varphi]\!] = \emptyset$. As an example, for an empty partial valuation $v = \top$, formula $\varphi = (x \neq y)$ and transition $t = \{y \leftarrow x\}$ (assign x to y) we have $\varphi_{pre} = \widehat{pre}_t(\varphi) = (x \neq x)$ and $v_{post} = \widetilde{post}_t(v) = \top$. In this case $\gamma(v) \cap [\![\varphi_{pre}]\!] = \emptyset$ holds but $\gamma(v_{post}) \cap [\![\varphi]\!] = \emptyset$ does not. Therefore we say that $v \dashv \varphi$ holds if $[\![\varphi]\!]_v = \texttt{false}$. This way all requirements for the refutes operator are satisfied.

4.3.2 Zone abstraction

Zones provide a suitable abstraction for both concrete and abstract labels. They allow for very coarse and also very fine representation of states, hence the interpolation domain can use zone abstraction as well.

The *initabstr* function for zone abstraction always gives the top element of the zone domain, which is a zone that contains only those clock constraints that ensure that all clock variables have non-negative values. Using the top element of the abstract labelling do-

main, just as with explicit value abstraction, complies with the labelling properties of the ART.

In this case all three abstract labelling domains are the same, which reduces the proves and refutes operators to common operations on zones. The proves operator corresponds to checking the inclusion of two zones. This can be done by comparing the values of the DBM representation of zones.

The refutes operator corresponds to checking the inconsistency of the intersection of zones. The intersection is computed from the DBM representation of the two zones by taking the minimal value for all elements. Checking inconsistency is done by checking whether there is a negative cycle in the resulting DBM interpreted as the adjacency matrix of a weighted graph.

Chapter 5

Implementation

I implemented the presented general framework in Theta, a generic, modular and configurable model checking framework, aiming to support the development and evaluation of abstraction refinement-based algorithms for the reachability analysis of different formalisms [9]. I used several already existing components of Theta: the ARG and XtaLts classes, the Abstractor and Analysis interfaces, the PartialOrd interface for partial orders and the TransFunc interface for post operators.

5.1 General lazy abstraction framework

Figure 5.1 shows a simplified class diagram of the prototype implementation in the Theta framework for the XTA formalism (Uppaal timed automata [7]). The main loop of the general lazy abstraction algorithm (Algorithm 1) is implemented in the LazyXtaAbstractor class, which also implements the Abstractor interface, i.e. it can initialize an abstract reachability graph (ARG) and check the reachability of a target node. The LazyXtaAbstractor class uses an XtaLts (labelled transition system) that provides enabled actions for a state and an Analysis that can provide the initial states and the successors for a state (corresponding to the $\overline{post}$ operator).

The configurability of the algorithm is provided by an algorithmStrategy object that implements the generic AlgorithmStrategy interface. This interface provides methods for obtaining an initial abstract label for a given concrete label, as well as for checking whether a given node can be covered by an other given node (mightCover). It also provides methods for the additional actions that should be performed when a node is covered (cover) and when a transition cannot be taken from a node (disable).

Currently there are three classes implementing the AlgorithmStrategy interface. BasicStrategy is the implementation that uses no abstraction, i.e. all abstract labels are the same as the corresponding concrete label and therefore abstraction refinement is never performed, meaning that cover and disable are no-op in this case. ItpStrategy uses the interpolation techniques presented in the previous chapter. Prod2Strategy is the product of two strategies, it is useful when we use two abstractions at the same time, e.g. explicit value abstraction with backward interpolation for the data variables and zone abstraction with forward interpolation for the clock variables of a timed automaton.

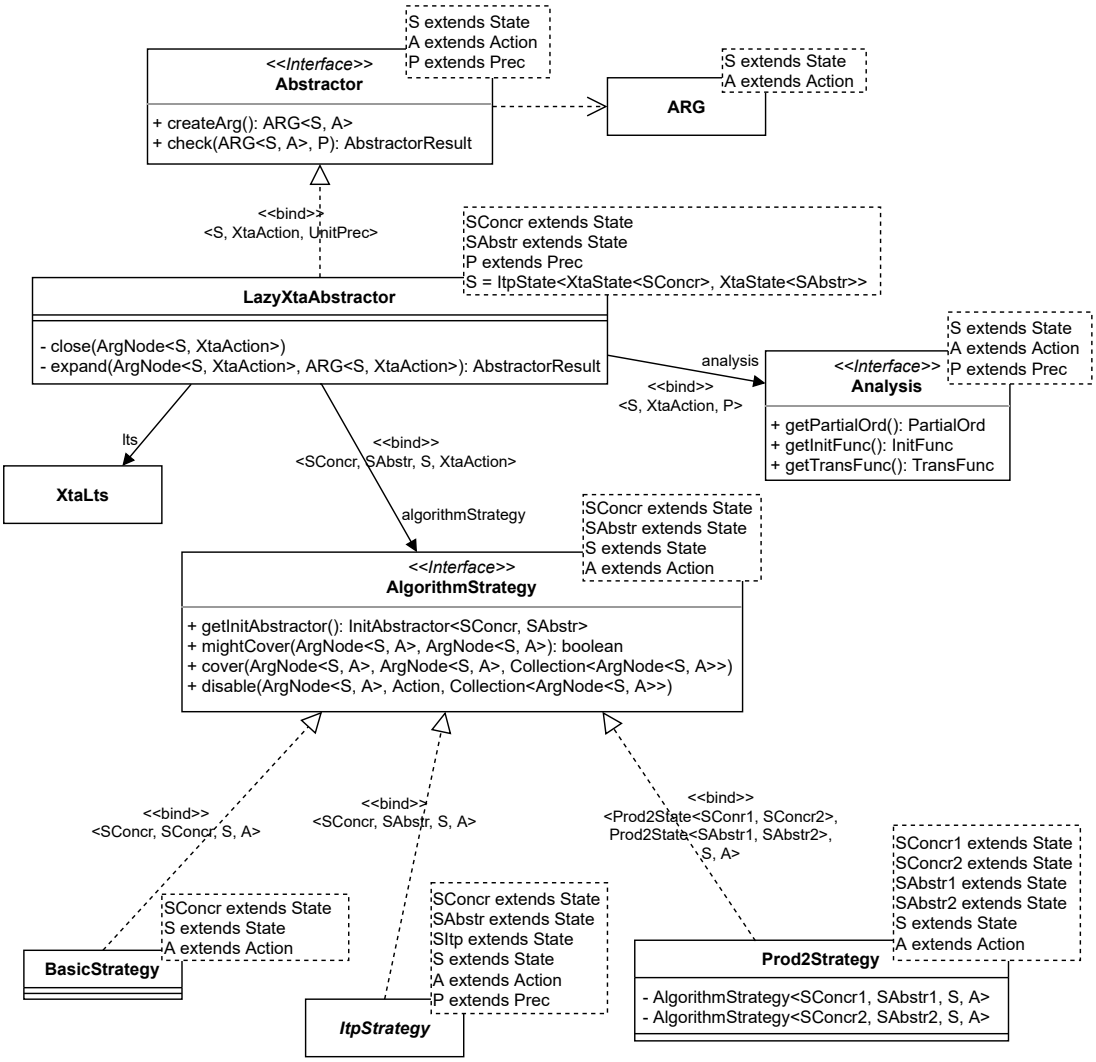


Figure 5.1: Class diagram of the general framework

5.2 Interpolation algorithms

A simplified class diagram of classes that participate in the interpolation algorithms is shown in Figure 5.2. The previously mentioned ItpStrategy has an abstract block method that corresponds to the procedure of the same name presented in the previous chapter. Its backward and forward variants are implemented in the descendants of ItpStrategy: BwItpStrategy and FwItpStrategy.

For the proves and refutes operators new interfaces were introduced that comprise operations that are usually used in the same context. The Concretizer interface contains methods corresponding to the $\gamma_{c \to a}$ and $\vdash$ operators. This interface is used by the ItpStrategy class for the concrete and abstract labelling domains.

The Interpolator interface is used for the abstract labelling and interpolation domains. It contains a method for computing an interpolant, as well as methods corresponding to the $\dashv$, $\gamma_{a \to i}$ and $\neg$ (complement) operators.

In the interpolation algorithms the meet of two states of the abstract labelling domain has to be computed and the top element of the domain has to be obtained. Therefore the ItpStrategy class uses a lattice for the abstract labelling domain. For the sake of

completeness the newly introduced Lattice interface provides some other operations as well that can be expected of lattices.

For the $\widehat{pre}$ operator the introduction of a new interface was needed. The InvTransFunc interface follows the pattern of the TransFunc interface and is used by the interpolation algorithms to compute pre-images of states of the interpolation domain. In forward interpolation a $\widetilde{post}$ operator is also needed, hence the FwItpStrategy class uses a TransFunc in addition to the InvTransFunc inherited from its parent class.

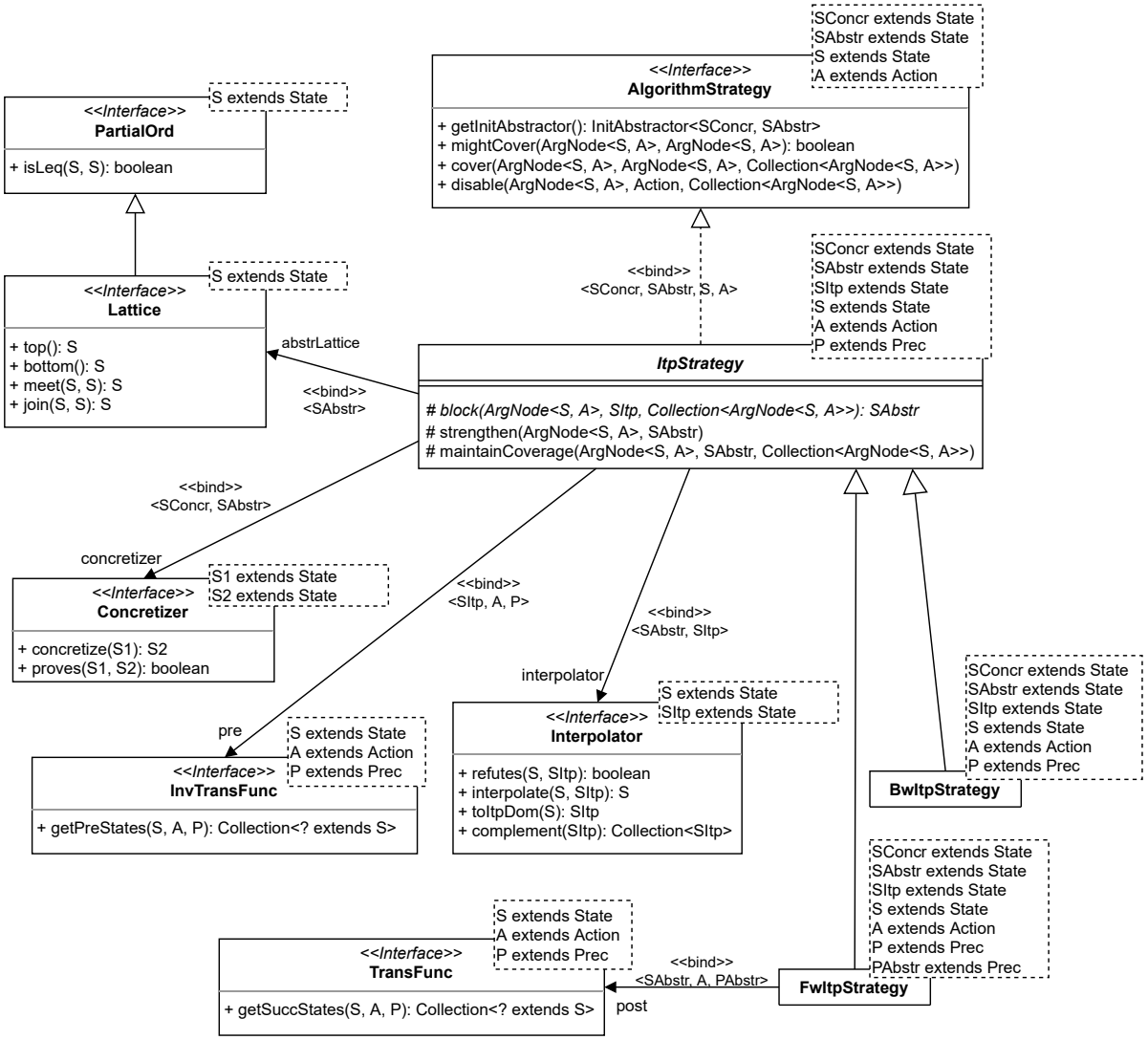


Figure 5.2: Diagram of classes that participate in abstraction refinement with interpolants

Chapter 6

Evaluation

To evaluate the applicability of the prototype of the general lazy abstraction framework, I used six models of different size and measured the number of nodes in the ART, the number of successful coverages, the number of refinement steps (refined nodes in all refinements performed) and execution time. The used algorithms are deterministic, i.e. all results except for execution time are always the same for a given model and configuration. Execution times were obtained by taking the average execution time of 10 runs.

Table 6.1 shows the results of running the algorithm for all six models with six different configurations. The configurations were chosen so that execution times can be compared with the original lazy abstraction framework that only supports identity or explicit value abstraction for data and zone abstraction for clocks, hence I used the same abstractions for evaluating the algorithms. For abstraction refinement the interpolation algorithms BLOCK_{BW} and BLOCK_{FW} were used for all abstractions. In the table, column Time* contains execution times with the original implementation and the last column contains the results with the new, general implementation.

Figure 6.1 shows a comparison of execution times for all models, where blue colour corresponds to the original implementation and orange corresponds to the general one. The configurations are written in an abbreviated form where the first part is the abstraction refinement strategy used for data variables and the second part is the same for zones.

The comparison of the original and the general implementation shows that there is a slight difference in execution times, mostly in favour of the original implementation. The reason for the small loss in efficiency can be the introduction of many new components that are necessary for a general approach. It can also be the result of allowing multiple pre-images of a state. Even if there are only pre-images consisting of a single state, the implementation has to be prepared for handling pre-images consisting of multiple states. Moreover, computing the meet of the original abstract label and the interpolant in explicit value abstraction has become more complex, as the general approach uses a lattice for the abstract domain, where it cannot be presumed that the two partial valuations do not contain the same variable with different values.

6.1 Comparison of configurations

In four cases (FDDI-2, Fischer-diag-3-32-64, Lynch-7-16, CSMA-9) using no abstraction for the data variables reduced execution time, the number of refinement steps and sometimes even the size of the ART. The explanation of this is that in these models most

data variables have an important role in determining possible behaviours and therefore they should be tracked. In contrast, with explicit value abstraction all data variables are initially abstracted and nodes have to be refined in order to track these variables. Moreover, regardless of these refinements, new nodes are always created with all data variables abstracted.

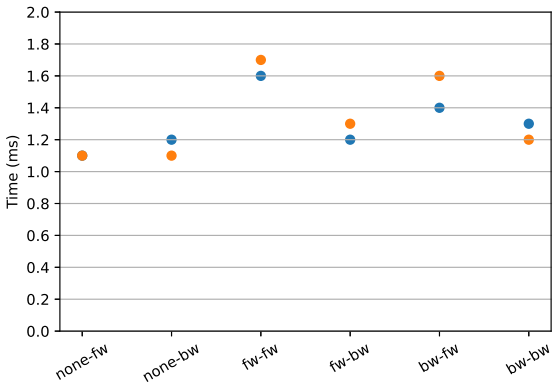
On the other hand, if there are more data variables in a model, it has a higher probability that not all variables are relevant everywhere, therefore in these cases using an abstraction for the data variables is beneficial. This is the case with the models Engine-classic and Fischer-split-3-32-64 as well.

The forward and backward variants of the interpolation algorithm have two notable differences that determine their efficiency. BLOCK$_{FW}$ uses post-image computation in addition to pre-image computation, whereas BLOCK$_{BW}$ uses only pre-images. On the other hand, the possibility of the complement of the interpolant being a set of multiple states can cause branching recursion in BLOCK$_{BW}$, while in BLOCK$_{FW}$ this is not possible. Therefore, the best configuration is dependent on the characteristics of the given model.

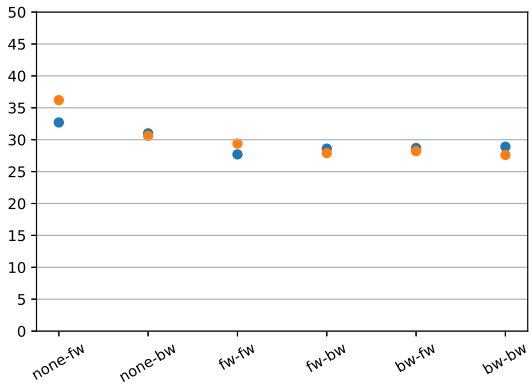
This theory is mirrored by the measured execution times as well. Although with the currently analysed models there are no significant differences in forward and backward interpolation for explicit value abstraction, the better variant for zones can easily be identified for an individual model in nearly all cases. For FDDI-2, Fischer-diag-3-32-64, Lynch-7-16 and CSMA-9 the algorithms performed better with using backward interpolation for zones, whereas for the Fischer-split-3-32-64 model forward interpolation outperformed backward interpolation.

Model	Data	Clocks	ART size	Coverages	Refin. steps	Time* (ms)	Time (ms)
FDDI-2	-	FW	40	11	22	1.1	1.1
	-	BW	40	11	22	1.2	1.1
	FW	FW	40	12	61	1.6	1.7
	FW	BW	40	12	61	1.2	1.3
	BW	FW	40	12	61	1.4	1.6
	BW	BW	40	12	61	1.3	1.2
Engine-classic	-	FW	561	82	590	32.7	36.2
	-	BW	561	75	646	31.0	30.6
	FW	FW	418	60	716	27.7	29.4
	FW	BW	418	60	772	28.6	27.9
	BW	FW	418	60	716	28.7	28.2
	BW	BW	418	60	772	28.9	27.6
Fischer-diag-3-32-64	-	FW	193	95	72	3.9	4.3
	-	BW	199	103	77	4.0	4.0
	FW	FW	193	95	225	5.3	5.8
	FW	BW	1037	103	234	4.7	5.4
	BW	FW	193	95	225	5.2	5.6
	BW	BW	1037	103	234	4.9	5.5
Fischer-split-3-32-64	-	FW	1947	1516	1248	76.5	80.5
	-	BW	2408	2139	2167	97.0	106.5
	FW	FW	423	329	739	24.9	22.6
	FW	BW	675	844	1849	56.2	60.8
	BW	FW	423	325	728	23.4	24.2
	BW	BW	689	860	1902	55.3	56.4
Lynch-7-16	-	FW	46915	36938	22379	1346.1	1492.7
	-	BW	46915	36938	22379	1204.7	1285.5
	FW	FW	46915	45848	85940	1982.1	2101.0
	FW	BW	46915	45848	85940	1838.1	1936.7
	BW	FW	46915	45848	85940	1966.1	2081.9
	BW	BW	46915	45848	85940	1825.4	1936.2
CSMA-9	-	FW	78552	48076	22950	3728.2	3940.7
	-	BW	78552	48076	22950	3330.6	3568.0
	FW	FW	78552	67985	103111	5878.7	6164.0
	FW	BW	78552	67985	103111	5436.7	5966.2
	BW	FW	78552	67985	103111	6196.8	6199.0
	BW	BW	78552	67985	103111	5534.2	6039.0

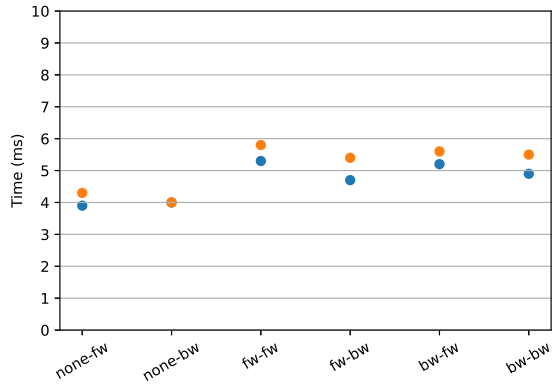
Table 6.1: Comparison of abstraction refinement strategies



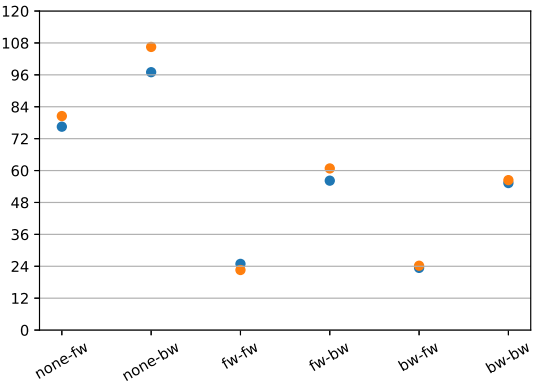
FDDI-2



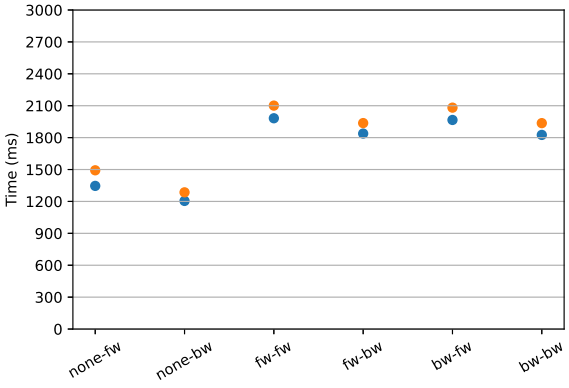
Engine-classic



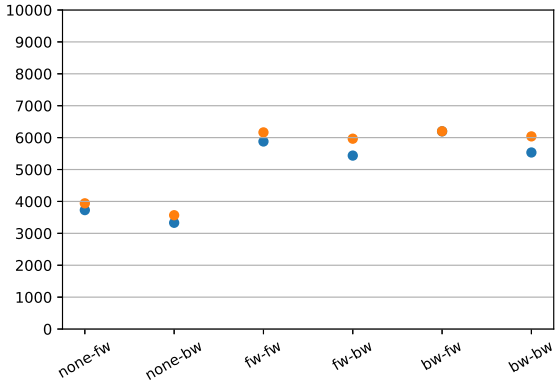
Fischer-diag-3-32-64



Fischer-split-3-32-64



Lynch-7-16



CSMA-9

Figure 6.1: Execution times

Chapter 7

Related work

CEGAR Counterexample-guided abstraction refinement (CEGAR) [3][4] is an abstraction-based model checking technique. The idea of the CEGAR approach is to start with a very coarse initial abstraction, then refine it iteratively. In each iteration the model either satisfies the requirement (when the unsafe state cannot be reached) or a counterexample is found (a path to the unsafe state). In the latter case it has to be checked, whether the counterexample is feasible in the concrete model. If it is feasible, then the model indeed does not satisfy the requirement. If it is unfeasible, then the counterexample is spurious, and the abstraction has to be refined in such a way that eliminates the spurious counterexample. The CEGAR algorithm can be divided into two alternating phases. The first one is the abstraction phase when the abstract reachability graph is built according to the current level of abstraction. The second phase is the refinement phase when the precision of the abstraction is increased to eliminate spurious counterexamples.

The lazy abstraction techniques presented in this work show some similarity to the CEGAR method, especially the abstraction phase, in both cases an abstract reachability tree is built for checking the reachability of an error state. However, there are some notable differences. The CEGAR method uses a precision that determines the level of abstraction for the whole ART and then computes successors from the abstract states. In lazy abstraction no such global precision is present, the successors are computed from the concrete labels instead. It is also notable that abstraction refinement has different purposes in these two approaches. In CEGAR, the abstraction is refined after reaching an error state and identifying it as a spurious counterexample, when the purpose is to have a more precise abstraction in the next iteration. In lazy abstraction, abstraction refinement does not depend on the reachability of the error state in the ART. Actually, the automaton is deemed unsafe as soon as a non-bottom error state is reached, without the need for further abstraction refinements. The purpose of abstraction refinement in this case is to uncover and later expand nodes that allow for behaviours that are not possible from other nodes.

Lazy abstraction A lazy abstraction approach applicable to various formalisms, including timed automata is proposed in [5]. It is related to counterexample-guided abstraction refinement, although it does not traverse the same part of the state space repeatedly but reuses the results of previous runs by abstracting locally.

In [2] a zone-based abstraction is proposed that uses the maximal lower and maximal upper bounds (LU-bounds) that occur in guards in addition to zones to obtain a coarser abstraction. In [6] this abstraction is adapted to the reachability problem for timed automata.

Apart from CEGAR, the Theta framework has already supported lazy abstraction for zone abstraction, zone abstraction with LU-bounds and explicit value abstraction [8].

Chapter 8

Conclusions

In this work we presented numerous abstractions, defined a general lazy abstraction framework, generalized interpolation algorithms for abstraction refinement and described the usage of the existing abstractions with the new framework. I also implemented and tested a prototype of this configurable lazy abstraction framework in the Theta tool. Our results show that the generalization of the algorithms preserved their efficiency, they performed well on benchmark models as well.

8.1 Future work

The applicability of the proposed general lazy abstraction framework can be expanded by lifting some of the limitations that were introduced while generalizing the algorithms. A possible theoretical work for the future is the further generalization of the approach by allowing the use of such post operators in the interpolation algorithms that produce multiple post-images. It could also be useful to integrate other abstraction techniques into the general framework in Theta and broaden the scale of supported abstractions.

Bibliography

[1] Rajeev Alur. *Timed automata*, pages 8–22. Springer-Verlag Berlin Heidelberg, 1999. ISBN 978-3-540-48683-1. DOI: 10.1007/3-540-48683-6.

[2] Gerd Behrmann, Patricia Bouyer, Kim G Larsen, and Radek Pelánek. Lower and upper bounds in zone-based abstractions of timed automata. *International Journal on Software Tools for Technology Transfer*, 8(3):204–215, 2006.

[3] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45047-4.

[4] Ákos Hajdu and Zoltán Micskei. Efficient strategies for cegar-based model checking. *Journal of Automated Reasoning*, 64(6):1051–1091, 2020.

[5] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, 2002.

[6] Frédéric Herbreteau, Balaguru Srivathsan, and Igor Walukiewicz. Lazy abstractions for timed automata. In *International Conference on Computer Aided Verification*, pages 990–1005. Springer, 2013.

[7] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1(1):134–152, 1997.

[8] Tamás Tóth and István Majzik. Configurable verification of timed automata with discrete variables. *Acta Informatica*, 2020. ISSN 1432-0525. DOI: 10.1007/s00236-020-00393-4. URL https://doi.org/10.1007/s00236-020-00393-4.

[9] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: A framework for abstraction refinement-based model checking. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 176–179, 2017. DOI: 10.23919/FMCAD.2017.8102257.