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

Abstraction-based model checking techniques for real-time systems

Scientific Students' Association Report

Author:

Dóra Cziborová
Béla Ákos Vizi

Advisor:

Mihály Dobos-Kovács
Dániel Szekeres
Kristóf Marussy

2022

Contents

Kivonat **i**

Abstract **ii**

1 Introduction **1**

2 Background **3**

2.1 Formal verification 3

2.2 Extended timed automata 4

2.2.1 Properties 6

2.3 Timed control flow automata 7

2.3.1 Transfer function 8

2.4 Abstract domains 9

2.4.1 Explicit value abstraction 9

2.4.2 Predicate abstraction 10

2.4.3 Zone abstraction 10

2.4.4 Product abstraction 11

2.5 CEGAR 12

2.5.1 Abstraction 13

2.5.2 Refinement 13

2.6 Lazy abstraction 14

2.6.1 Lazy abstraction over the zone domain 15

3 Overview **18**

3.1 Mapping Real-Time Systems to TCFA 19

3.1.1 “Error” locations in Real-Time Sytems 20

3.1.2 Converting *RTS* with Error Locations to *TCFA* 20

4 CEGAR with clock activity **22**

5 Combined CEGAR **25**

5.1 Pruning strategies 30
5.2 Correctness of the combined CEGAR algorithm 30

6 Evaluation **33**
6.1 Lazy abstraction 33
6.2 Eager CEGAR 35
6.3 Combined CEGAR 36
6.4 Comparison of approaches 37

7 Related work **39**

8 Conclusions **40**
8.1 Future Work 41

Bibliography **41**

Kivonat

Kritikus valós idejű rendszerek fejlesztésekor egy kulcsfontosságú kihívás a szoftverrendszerek biztonságának verifikációja. Az ilyen rendszerekkel szemben gyakran szigorú időzítési követelményeink is vannak, melyek megsértése akár életvesztéshez vagy jelentős anyagi károkhoz vezethet. Ezen követelmények kielégítésének biztosítására automatizált modellellenőrzés használható, melyben egy automatizált eszköz ellenőrzi a kívánt biztonsági követelményeket a rendszer egy formális leírásán. Az analízis eredménye egy bizonyíték a rendszer helyességére, vagy pedig egy hibaút, ami ellenpéldaként szolgál.

A modellellenőrzés azonban még közepes méretű rendszerek esetén is nehéz feladat, hiszen a bejárandó állapotok száma gyakran a rendszer méretében exponenciálisan növekszik. Ezen kívül az időzítések figyelembe vételéhez az állapotoknak egy megszámlálhatatlanul végtelen halmazával szükséges dolgozni. A mérnöki alkalmazásokban az imént említett kihívásokkal megbirkózó modern verifikációs algoritmusok használatához a rendszerek és kritériumok komplex leírásait alacsonyszintű matematikai formalizmusokra szükséges fordítani.

Ezen munka célja az absztrakció alapú modellellenőrzés támogatása a széles körben használt időzített automata formalizmushoz és a hozzá tartozó tulajdonság specifikációs nyelvhez. A már létező algoritmusok adaptálása mellett a technikák egy olyan újszerű kombinációjára teszünk javaslatot, mely ötvözi az irodalomban elérhető megközelítések hatékonyságát az adatok és az idő absztrakciójára. Ezen kívül támogatást adunk egy már létező modellellenőrző eszközben az időzített automaták tulajdonság specifikációinak kezelésére.

Különösen a következő kontribúciókat mutatjuk be: (i) Visszavezetjük az időzített automaták elérhetőségi tulajdonságait egy keresési problémára. (ii) Kombináljuk az absztrakciófinomítás lusta és mohó megközelítését egy újszerű vegyes stratégiaként. (iii) Implementáljuk a javasolt technikákat a nyílt forráskódú Theta modellellenőrző keretrendszerben. (iv) Kiértékeljük a létező és az általunk javasolt technikákat ipari és szintetikus benchmark modelleken.

Legjobb tudomásunk szerint a miénk az első olyan megközelítés, ami kombinálja a lusta és mohó absztrakciós technikák előnyeit az időzített automaták hatékony verifikációjára.

Abstract

Safety verification of software systems is a key challenge in the development of critical real-time embedded systems. Such systems are often subject to stringent real-time scheduling requirements, the violation of which may lead to loss of life or significant damage to property. Automated model checking can be used to ensure the satisfaction of these requirements, where a formal description of the system is checked against the desired safety properties by an automated tool. The result of the analysis is either a proof of correctness or an error trace serving as a counterexample.

However, model checking systems of even a moderate size can be difficult, as 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. In engineering applications, the use of modern verification algorithms tackling the aforementioned challenges requires translating complex descriptions of systems and properties into low-level mathematical formalisms.

This work aims at providing support for abstraction-based model checking for the widely used extended timed automaton formalism and the associated property specification language. We adapt existing algorithms, as well as propose a novel combination of techniques to take advantage of the efficiency of approaches for data and time abstractions available in the literature. Moreover, we add support to an existing model checking tool for handling property specifications of extended timed automata.

We present the following specific contributions: (i) We reduce symbolic reachability properties of extended timed automata to a search problem. (ii) We combine lazy and eager abstraction refinement approaches into a novel mixed strategy. (iii) We implement the proposed techniques in the open source Theta model checking framework. (iv) We evaluate existing and proposed techniques in the context of industrial and synthetic benchmark models.

To the best of our knowledge, ours is the first approach to combine the advantages of both lazy and eager abstraction techniques for the efficient verification of extended timed automata.

Chapter 1

Introduction

Software systems are becoming prevalent in safety-critical applications, such as aerospace, automotive, transportation, and industrial IoT infrastructures. Safety-critical systems often have strict timing constraints, so they must provide soft or hard real-time services.

The design of real-time safety critical systems is often supported by rigorous design techniques and modeling languages. However, to ensure the functional correctness of the desired system, engineers need advanced verification techniques. Testing is often used to find bugs in critical applications, but testing can not be used to prove functional correctness, especially in the presence of complex timing and data-related requirements.

Formal verification is an automated technique to explore the possible behaviors of systems and find errors or prove functional correctness. However, formal verification faces the problem of state space explosion in the presence of data and timing in the systems. Real-time safety-critical systems process data in a timely manner, which makes the formal verification of such systems a huge challenge.

Abstraction-based techniques were developed in the literature to handle data in verification and various refinement techniques were introduced to compute better and better abstractions during the verification process. The family of algorithms exploiting iterative refinement is called Counterexample Guided Abstraction Refinement, CEGAR for short.

Unfortunately, CEGAR-based algorithms turned out to be less efficient in representing the timed behavior. So far, the most efficient tools for exploring timed behavior were based on explicit state space exploration with fine-grained, lazy abstractions: this approach leads to enumerating all the possible equivalence classes of timed behavior, which is good for answering the verification questions but yields a huge task for the verification algorithms. This conservative abstraction technique prevents the application of timed verification for industrial systems where enumerating the state space does not scale.

Attempts tried to extend the CEGAR-based abstraction refinement approach to timed systems and clock activity was used to describe the actually needed precision of the abstraction. Unfortunately, the clock activity-based refinement strategy still has not provided the desired coarse-grained abstraction that would reduce the state space representation significantly. Lazy abstraction uses a different abstraction refinement strategy, which turned out to be efficient for timing, but unfortunately, this approach could only handle a restricted class of software and system models.

Our motivation is to extend the former approaches and provide a framework with various solutions to verify real-time software systems. In order to process the high-level input models of real-time systems, we developed model transformations, that map the require-

ment and the system models into a formal representation, that serves as input for the verification algorithms. We implemented existing algorithms from the literature and integrated them to work on our formal representation. We extended the CEGAR-based approach to use activity-based abstraction and refinement [14].

From the algorithmic point, we developed a novel algorithm as the combination of eager CEGAR-based data refinement and lazy refinement for time. We have modified the general CEGAR loop and introduced a two-phase refinement strategy. The new algorithm combines the advantages of both algorithms, but without their limitations. So far, this is the first approach to provide efficient automatic abstractions for the timed behavior and also the data-dependent behavior in real-time software systems. Our approach can handle even complex data-dependent and timed behavior together in one framework without requiring manual tuning of the abstractions. From the theoretical point of view, we formalized the new algorithm and proved its correctness.

We have implemented all the formerly mentioned algorithms in the open-source Theta formal verification framework and evaluated on benchmark problems. Measurements show that the new approach is competitive with the existing ones on the benchmark models, without having the same restrictions on the expressive power of the input models.

Chapter 2

Background

2.1 Formal verification

Formal verification is the term used for proving the correctness of a system using mathematical techniques. Model checking is a formal verification method capable of proving that some formalized properties hold in a formal 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 [8] algorithm are the model and the requirements, both formally specified. The algorithm then outputs whether the given model satisfies the given requirements. If case the requirement is satisfied, then it gives a mathematical proof for it, or, in case of the model not satisfying a requirement, it outputs a corresponding counterexample, where the requirement is not met.

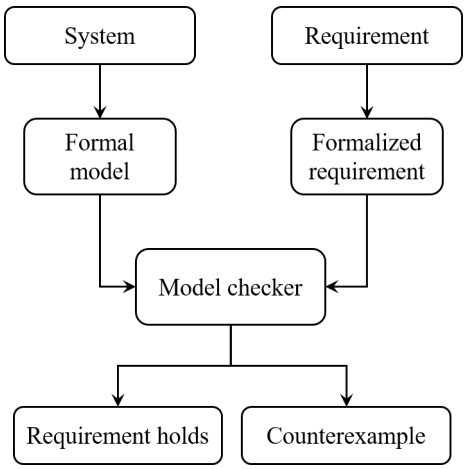

Figure 2.1: Model checking

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 behavior of the original model but also enables behavior 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 Extended timed automata

Timed automata extend finite automata with *clock variables*, which are special variables that model the passage of time. The values of these variables increase continuously at the same rate. Apart from this, the values of any subset of the clock variables can be reset to 0 at any point of operation, which is called *resetting*. Arbitrary assignments are not permitted for them. A clock constraint is a conjunctive formula of atomic constraints of the forms $x \sim n$ or $x - y \sim n$ where $x, y \in \mathcal{C}$, $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. Let $\mathcal{B}(\mathcal{C})$ denote a set of constraints that we will use as guards later.

A *clock valuation* $val_{\mathcal{C}}$ for a set of clock variables V_C is a function $V_C \to \mathbb{R}_+$ that assigns a non-negative real value to each clock variable. For a clock valuation $val_{\mathcal{C}}$ and a set of clock variables $r \subset V_C$, the notation $val_{\mathcal{C}}[r]$ will mean a valuation where each clock in r is reset (the new valuation assigns 0 to them), while all others retain their values from the original valuation.

A *data valuation* for a set of data variables V_D is a function $V_D \to \bigcup_{v \in V_D} Dom(v)$ that assigns to each variable v an element from its domain $Dom(v)$.

An *Extended Timed Automaton (XTA)* is a timed automaton extended with data variables and synchronization channels. In a real-time system, more than one process can be defined, which are separate Extended Timed Automata, and these can synchronize over channels.

Definition 1 (Real-Time System). A Real-Time System $\mathcal{S}$ is a tuple $\langle P, \mathcal{D}_g, \mathcal{C}_g, v_0^g, S = S_b \uplus S_n \uplus \{\tau\} \rangle$, where

- $P = \{p_0, p_1, \ldots, p_{|P|}\}$ is a finite set of extended timed automata, that is called the *processes* of the system,
- $\mathcal{D}_g = \{d_1^g, d_2^g, \ldots, d_N^g\}$ is a finite set of *global data variables* with corresponding domains $D_1^g, D_2^g, \ldots, D_N^g$,
- $\mathcal{C} = \{c_{g_1}, c_{g_2}, \ldots, c_{2_{|\mathcal{C}|}}\}$ is a finite set of *global clock variables* that can take non-negative real values,
- v_0^g is the *initial data valuation*, determining the initial value of each data variable, if no value is assigned to a variable then it takes a nondeterministic value of its domain
- $S = \{s_1, s_1 \ldots, s_{|S|-1}, \tau\}$ is a finite set of channels through which processes can synchronize; it is the disjoint union of *broadchast channels* S_b, *non-broadcast channels* S_n and the "null-channel" τ used for denoting non-synchronized actions ▪

Definition 2 (Extended timed automaton). An XTA $\mathcal{A}$ in the context of a Real-Time System $\langle P, \mathcal{D}_g, \mathcal{C}_g, S \rangle$ is a tuple $\langle \mathcal{L}, l_0, v_0^{loc}, \mathcal{D}_{loc}, \mathcal{C}_{loc}, I, E, T \rangle$, where

- $\mathcal{L} = \{l_0, l_1, \ldots, l_{|\mathcal{L}|}\}$ is a finite set of locations,
- $l_0 \in \mathcal{L}$ is the *initial location*,
- v_0^{loc} is the *initial data valuation*, determining the initial value of each data variable, if no value is assigned to a variable then it takes a nondeterministic value of its domain
- $\mathcal{D}_{loc} = \{d_1^{loc}, d_2^{loc}, \ldots, d_M^{loc}\}$ is a finite set of *local data variables* with corresponding domains $D_1^{loc}, D_2^{loc}, \ldots, D_M^{loc}$,
- $\mathcal{C} = \{c_{loc_1}, c_{l_2}, \ldots, c_{|\mathcal{C}_{loc}|}\}$ is a finite set of *local clock variables* that can take non-negative real values,
- $I : \mathcal{L} \to \mathcal{B}(\mathcal{C})$ assigns an *invariant* to each location
- $E \subset \mathcal{L} \times Op \times S \times \mathcal{L}$ is the set of directed edges between locations, each of them labeled with some operation $op \in Op$ and a channel $s \in S$
- T is a function that maps each location to a type, that can be NORMAL, URGENT, COMMITTED
  - Normal location has no extra functionality
  - Urgent location l_U adds an extra clock variable x that resets in every incoming edge to this location and adds an invariant $x <= 0$ to l_U. So time is not allowed to pass in these locations (clock variables cannot increase).
  - Committed locations are the same as Urgent locations but the next transition in the system must involve an outgoing edge of one of the committed locations if any process is in this type of location. ▪

A global variable can be referenced from all processes, but local variables are visible only in the process that they were defined in. An operation $op \in Op$ consists of a guard, a set of assignments, and resets.

- A guard g is a conjunction of data variable conditions and clock constraints.
- In an assignment a, a data variable can be assigned to any value in its domain.
- Reset r is a set of clock variables, which will be set to 0 after applying the operation.

A *state* in a Real-Time System with process set $P = \{p_0, p_1, \ldots, p_{|P|}\}$ is a tuple $\langle L, val_D, val_{\mathcal{C}} \rangle$, where $L = \langle l_0, \ldots l_{|P|} \rangle \in \times_{p \in P} Loc(P)$ determines the currently active location in each process, val_D is a valuation, $val_{\mathcal{C}}$ is a clock valuation satisfying $I(L) = I(l_0) \wedge I(l_1) \wedge \cdots \wedge I(l_{|P|})$. The initial state contains the initial location of each process, its clock valuation $val_{\mathcal{C}}^0$ assigns 0 to every clock variable, and its data valuation val_D^0 is computed in the following way:

$$val_D^0 = \begin{cases} v_0^g, & \text{if } v \in \mathcal{D}_g \\ v_0^{loc_i} & \text{if } v \in \mathcal{D}_{loc}^i \end{cases},$$

where $\mathcal{D}_{loc}^i$ denotes the set of local data variables of p_i and $v_0^{loc_i}$ denotes its initial data valuation.

We call an edge $e = \langle l, op = \langle g, a, r \rangle, s, l' \rangle$ *enabled*, if the process is in l, val_D satisfies the data conditions of g, $val_{\mathcal{C}}$ satisfies the clock conditions of g and $val_{\mathcal{C}}[r]$ satisfies $I(l')$.

If two processes p_1, p_2 are synchronised there are edges labelled with $\{s!|s \in S\}$ in p_1 and edges labeled with $\{s?|s \in S\}$ in p_2. Initiator edge e_1=$\langle l_1, op = (g_1, a_1, r_1), s!, l'_1\rangle$ can initiate the synchronization if it is enabled, and if a receiving edge e_2=$\langle l_2, op = (g_2, a_2, r_2), s?, l'_2\rangle$ is enabled too. When p_1 and p_2 synchronize e_1, e_2 are fired at the same time and neither of them can be fired individually. There is another type of channel, the broadcast channel. In this case, there is one initiator edge and can be more receiving edge. If the initiator edge is enabled, it can be always fired even if no receiving edge is enabled. Enabled receiving edges will synchronize and will be fired.

The state $\langle L = \langle l_1, \ldots, l_{|P|}\rangle, val_D, val_\mathcal{C}\rangle$ has a *discrete transition* to $\langle L' = \langle l'_1, \ldots, l'_{|P|}\rangle, val'_D, val'_\mathcal{C}\rangle$ if one of the following condition groups holds:

- $\exists i \in 1 \ldots |P|$: there is an enabled edge $e = \langle l_i, op = \langle g, a, r, \rangle, \tau, l'_i\rangle$ in the ith process, $\forall k \neq i \in 1 \ldots |P| : l_k = l'_k$, $val'_\mathcal{C} = val_\mathcal{C}[r]$ and val_D changes according to a
- $\exists s \in S_n : \exists i \in 1 \ldots |P|$: there is an enabled edge $e = \langle l_i, op = \langle g_!, a_!, r_!, \rangle, s!, l'_i\rangle$ in the ith process, $\exists j \in 1 \ldots |P|$: there is an enabled edge $e = \langle l_j, op = \langle g_?, a_?, r_?, \rangle, s?, l'_j\rangle$ in the jth process, $\forall k \neq i, j \in 1 \ldots |P| : l_k = l'_k$, $val'_\mathcal{C} = val_\mathcal{C}[r_! \cup r_?]$ and val_D changes according applying to $a!$ first, then $a?$
- $\exists s \in S_b$: $\exists i \in 1 \ldots |P|$: there is an enabled edge $e = \langle l_i, op = \langle g_!, a_!, r_!, \rangle, s!, l'_i\rangle$ in the ith process, let $J = \{j \mid$ there is at least one enabled outgoing edge from l_j labeled with $s?\}$, then $\forall j \in J$: there is an enabled edge $e_j = \langle l_j, op = \langle g_j, a_j, r_j\rangle, s?, l'_j\rangle$, $val'_\mathcal{C} = val_\mathcal{C}[r_! \cup \bigcup_{j\in J} r_j]$, and val_D changes according applying to $a!$ first, then all a_j assignments for all $j \in J$

If there is at least one committed location in L, then l_i or l_j above must be a committed location in the definitions above - whenever a committed location is active, only transitions that leave a committed location can be fired (with self-loops also considered leaving).

The state has a *time transition* (*delay*) from $\langle L, val_D, val_\mathcal{C}\rangle$ to $\langle L, val_D, val'_\mathcal{C}\rangle$ if $val'_\mathcal{C}$ assigns $val_\mathcal{C}(c) + d$ for a non negative value d to every $c \in \mathcal{C}$, and $val'_\mathcal{C}$ satisfies $I(l_1) \cup I(l_2) \cup \cdots \cup I(l_{|L|})$.

2.2.1 Properties

Timed automata are usually verified with the help of property specification languages (PSL). We can specify properties on data-, clock variables and locations.

Definition 3 (State property). A property is a tuple $prop = \langle loc, d, c\rangle$, where

- loc is the location set that appears in prop
- d is the conjunction of conditions on data variables
- c is the conjunction of conditions on clock variables ▪

A state property $prop = \langle loc, d, c\rangle$ is satisfied in a state $s = \langle L, val_D, val_\mathcal{C}\rangle$ denoted by $\langle L, val_D, val_\mathcal{C}\rangle \models \langle loc, d, c\rangle$ if:

- $loc \in L$,
- val_D satisfies d

- val_C satisfies c

Properties can be divided into two classes in our implementation:

- Possibly: Marked with $E <>$ - evaluates to true iff there is a reachable state s of the automata, where the $s \models prop$
- Invariantly: Marked with $A[]$ - evaluates to true iff in every state s, $s \models prop$. It can be expressed with the Possibly operator: $not\ E <>\ not\ [state\ property]$.

Generally, to check the timed requirements, different kinds of PSL can be used such as LTL, CTL*, and CTL, but we only use a subset of this. The elements of this subset can be mapped to a reachability problem, so our algorithms can run on these systems [7].

2.3 Timed control flow automata

Our algorithms perform model checking on *timed control flow automata*. Our definition of timed control flow automaton in slightly different from the traditional definition of the control flow automata by the inclusion of clock variables.

Definition 4 (Timed control flow automaton). A timed control flow automaton is a tuple $TCFA = \langle \mathcal{L}, l^0, V_D, V_C, val_D^0, val_C^0, E \rangle$, where

- $\mathcal{L}$ is a finite set of locations,
- $l^0 \in \mathcal{L}$ is the initial location,
- $V_D = \{d_1, d_2, \ldots, d_{|V_D|}\}$ is a finite set of data variables with domains $D_1, D_2, \ldots, D_{|V_D|}$,
- $V_C = \{c_1, c_2, \ldots, c_{|V_C|}\}$ is a finite set of clock variables,
- $val_D^0 : V_D \to \bigcup_{i=0}^{|V_D|} D_i$ where $val_D^0(d_i) \in D_i$ for all $d_i \in V_D$ is a total valuation that maps data variables to their initial values in their corresponding domains,
- $val_C^0 : V_C \to \mathbb{R}_{\geq 0}$ is a clock valuation that maps clock variables to their initial values,
- $E \subset \mathcal{L} \times Op \times \mathcal{L}$ is a set of directed edges between locations, each edge labeled with an operation $op \in Op$, representing the execution of that operation. ▪

Let Val_D denote the set of total valuations over V_D and Val_C the set of clock valuations over the set of clocks V_C. We will also refer to Val_D and Val_C as *concrete domains*.

A state of the control flow automaton is a tuple $\langle l, val_D, val_C \rangle$ where $l \in \mathcal{L}$, $val_D \in Val_D$ and $val_C \in Val_C$. The set of these *concrete states* is denoted by S. We can also use the projections of these states to get a partial state corresponding to a specific domain (e.g. clock valuations only).

The following operations are defined on TCFA:

- *Guards* are distinguished by the form $[\varphi]$ where φ is a predicate over $V_D \cup V_C$. Given a valuation val, a guard $[\varphi]$ does not change the value of any variable $v \in val$ but the operation can only be executed if φ evaluates to *true* on val.

- An *assignment* $x := \varphi$ assigns the value of an expression φ over $V_D \cup V_C$ to the variable $x \in V_D \cup V_C$. Given a valuation val, it sets the value of the variable $x \in val$ in the successor state to φ and leaves the values of all other variables in val unchanged.
- A *havoc* operation $havoc(x)$ is a non-deterministic assignment that sets the value of a variable $x \in V_D \cup V_C$ to any value of its domain D_x. Given a valuation val, it produces a successor state for each value $v \in D_x$ such that the value of x is v and the values of all other variables in val are unchanged.

A *run* of the automaton is an alternating finite sequence of states and operations:
$\sigma = \langle l^0, val_D^0, val_C^0 \rangle - op_1 - \langle l^1, val_D^1, val_C^1 \rangle - op_2 - \cdots - op_n - \langle l^n, val_D^n, val_C^n \rangle$.

A location l is *reachable* in the model if and only if there exists a run from the initial state $\langle l_0, val_D^0, val_C^0 \rangle$ such that $l^n = l$.

We define the model checking problem for TFCA as checking for the reachability of a given *target location* in the model.

2.3.1 Transfer function

Transitions execute operations that bring the automaton from one state to its successor. The successor of a state consists of one or possibly multiple states. These states are determined by transfer functions. We refer to transfer functions that operate on concrete states and determine exact successors (i.e. without under- or over-approximation) as *concrete transfer functions*. We can define transfer functions for each domain separately:

Definition 5 ($T_{D,concr}$). The concrete transfer function for data valuations is a function $T_{D,concr} : Val_D \times Op \to 2^{Val_D}$ that maps a total valuation $val_D \in Val_D$ to its successor states with respect to an operation $op \in Op$. ▪

Definition 6 ($T_{C,concr}$). The concrete transfer function for clock valuations is a function $T_{C,concr} : Val_C \times Op \to 2^{Val_C}$ that maps a clock valuation $val_C \in Val_C$ to its successor states with respect to an operation $op \in Op$. ▪

The transfer function that operates on the actual states of the automaton is composed of the now-defined transfer functions for the data and clock domains. The term *concrete transfer function* used later in this work refers to this transfer function.

Definition 7 (Concrete transfer function T_{concr}). The concrete transfer function is a function $T_{concr} : \mathcal{L} \times Val_D \times Val_C \times Op \to 2^{\mathcal{L} \times Val_D \times Val_C}$ such that $(l', val'_D, val'_C) \in T_{concr}(l, val_D, val_C, op)$ if and only if

- $(l', val'_D) \in T_{D,concr}(l, val_D, op)$,
- $(l', val'_C) \in T_{C,concr}(l, val_C, op)$,
- $(l, op, l') \in E$. ▪

2.4 Abstract domains

The state space of models involved in verification tasks is usually large, often infinite. To handle such state space, abstractions are used, which means that we do not use concrete states during model checking, instead, we use abstract states comprising multiple concrete states. With the proper use of abstractions, the state space of a system can be represented by a significantly smaller set of abstract states.

Definition 8 (Abstract domain). An abstract domain is a tuple $D = \langle \mathcal{S}, \mathcal{S}_a, \sqsubseteq, \gamma \rangle$, where

- $\mathcal{S}$ is the set of *concrete states*,
- $\mathcal{S}_a$ is the set of *abstract states*,
- $\sqsubseteq \subseteq \mathcal{S}_a \times \mathcal{S}_a$ is a *preorder*, i.e., it is a reflexive ($s \sqsubseteq s$ for all $s \in \mathcal{S}_a$) and transitive ($s_1 \sqsubseteq s_2$ and $s_2 \sqsubseteq s_3$ implies $s_1 \sqsubseteq s_3$ for all $s_1, s_2, s_3 \in \mathcal{S}_a$) binary relation,
- $\gamma \colon \mathcal{S}_a \to 2^{\mathcal{S}}$ is the *concretization* function that maps abstract labels to the sets of states they represent, such that $s_1 \sqsubseteq s_2$ implies $\gamma(s_1) \subseteq \gamma(s_2)$ for all $s_1, s_2 \in \mathcal{S}_a$ ▪

If there exists a function $\alpha \colon 2^{\mathcal{S}} \to \mathcal{D}$ such that $A \subseteq \gamma(\alpha(A))$, $\alpha(\gamma(b)) \sqsubseteq b$ and $A_1 \sqsubseteq A_2$ implies $\alpha(A_1) \sqsubseteq \alpha(A_2)$, 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*.

We use a set Π of *precisions* to keep track of how coarse the abstraction is. Smaller precision corresponds to less concrete abstraction. Since abstraction-based algorithms use abstract states to explore the state space, we have to define transfer functions that operate on these abstract states with a given precision.

Definition 9 (Abstract transfer function). The *abstract transfer function* for an abstract domain $D = \langle \mathcal{S}, \mathcal{S}_a, \sqsubseteq, \gamma \rangle$ is the function $T_{abstr} : \mathcal{S}_a \times Op \times \Pi \to 2^{\mathcal{S}_a}$ that maps an $s \in \mathcal{S}_a$ to its successor states $\{s' \mid s' \in T_{abstr}(s, op, \pi)\}$ with respect to an operation $op \in Op$ and a precision $\pi \in \Pi$ such that for all $s_a \in \mathcal{S}_a$:
$\bigcup_{s_c \in \gamma(s_a)} T_{concr}(s_c, op) \subseteq \bigcup_{s'_a \in T_{abstr}(s_a, op, \pi)} \gamma(s'_a)$ ▪

There are numerous abstractions in the literature for both data and clock valuations. In the following, we present the ones we work with in our solutions, namely *explicit value abstraction* and *predicate abstraction* for data valuations, and *zone abstraction* for clock valuations, then we define the product of abstractions.

2.4.1 Explicit value abstraction

For the abstraction of total valuations over data variables we might use *explicit value abstraction* [4], where each abstract state explicitly tracks the values of some subset of data variables, while the rest of the variables may take any value. The value of an explicitly tracked variable can either be a value of its domain or $\top$ if the value of the variable is unknown.

Definition 10 (Explicit value abstraction). Explicit value abstraction over the set of data variables $V_D = \{d_1, d_2, \ldots, d_{|V_D|}\}$ with domains $D_1, D_2, \ldots, D_{|V_D|}$, their union $D = \bigcup_{i=0}^{|V_D|} D_i$ and a precision $\pi \subseteq V_D$ is the abstract domain $Expl(V_D, \pi) = \langle \mathcal{S}, \mathcal{E}, \sqsubseteq_{\mathcal{E}}, \gamma_{\mathcal{E}} \rangle$, where

- $\mathcal{S}$ is the set of *total valuations* $\{val_D : V_D \to D \mid val_D(d_i) \in D_i \text{ for all } d_i \in V_D\}$,
- $\mathcal{E}$ is the set of *partial valuations* $\{pval : \pi \to D \cup \{\top\} \mid \forall d_i \in \pi : pval(d_i) \in D_i \cup \{\top\}\}$,
- $pval_1 \sqsubseteq_{\mathcal{E}} pval_2$ if and only if $pval_2(d) \in \{pval_1(d), \top\}$ for all $d \in \pi$,
- $\gamma_{\mathcal{E}}(pval) = \{val_D \in \mathcal{S} \mid val_D(d) = pval(d) \text{ for all } d \in \pi\}$. ▪

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, $d \in supp(\alpha(A))$ and $\alpha(A)(d) = v$ if $val(d) = v$ for all $val \in A$. Otherwise, if there are some $val_1, val_2 \in A$ such that $val_1(d) \neq val_2(d)$, then $d \notin supp(\alpha(A))$. As a special case, we have $\alpha(\emptyset) = \bot$.

The transfer function for the abstract domain $Expl(V_D, \pi)$ is the abstract transfer function $T_{\mathcal{E}}$. Informally, $T_{\mathcal{E}}(pval, op, \pi)$ assigns values to variables in π according to operation op executed on $pval$ where it can be evaluated and assigns $\top$ to all other tracked variables.

2.4.2 Predicate abstraction

In *predicate abstraction* [19] the value of data variables are not tracked explicitly, but we track whether some predicates π hold in the state or not. There are 2^{π} states in the abstract model, because V_D can satisfies a predicate or not. Predicates are logical expressions that can contain constants and data variables of the model.

Definition 11 (Predicate abstraction). Predicate abstraction over the set of data variables $V_D = \{d_1, d_2, \ldots, d_{|V_D|}\}$ with domains $D_1, D_2, \ldots, D_{|V_D|}$, their union $D = \bigcup_{i=0}^{|V_D|} D_i$ and a set π of predicates over V_D is the abstract domain $Pred(V_D, \pi) = \langle \mathcal{S}, \mathcal{P}, \sqsubseteq_{\mathcal{P}}, \gamma_{\mathcal{P}} \rangle$, where

- $\mathcal{S}$ is the set of *total valuations* $\{val_D : V_D \to D \mid val_D(d_i) \in D_i \text{ for all } d_i \in V_D\}$,
- $\mathcal{P} \subseteq 2^{\pi}$ is the set of abstract states,
- $p_1 \sqsubseteq p_2$ if and only if $p_1 \Rightarrow p_2$, i.e. the preorder corresponds to implication,
- $\gamma_{\mathcal{P}}(p) = \{val_D \in \mathcal{S} \mid ((\bigwedge_{\varphi \in p} \varphi) \leftarrow val_D) = true\}$ where $(f \leftarrow val)$ denotes replacing each variable x in the logical expression f by $val(x)$. ▪

The transfer function for the abstract domain $Pred(V_D, \pi)$ is the abstract transfer function $T_{\mathcal{P}} : \mathcal{P} \times Op \times \pi \to \mathcal{P}$. A successor state is the strongest set of predicates in the precision that is implied by the source state and the operation.

2.4.3 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* [2].

Definition 12 (Zone). A *zone* is a set of clock constraints. For a set of clock variables V_C and a zone Z, let $(\!|Z|\!)$ denote the set of clock valuations $\{val_C : V_C \to \mathbb{R}_{\geq 0}\}$ such that val_C 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 $cval_1, cval_2 \in (\!|Z|\!)$ 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. a reasonable subset of clock constraints that appear in the model being analyzed.

Definition 13 (Zone abstraction). Zone abstraction over the set of clock variables $V_C = \{c_1, c_2, \ldots, c_{|V_C|}\}$ and a precision $\pi \subseteq V_C$ is the abstract domain $Zone(V_C, \pi) = \langle \mathcal{S}, \mathcal{Z}, \sqsubseteq_{\mathcal{Z}}, \gamma_{\mathcal{Z}} \rangle$, where

- $\mathcal{S}$ is the set of *clock valuations* $\{val_C : V_C \to \mathbb{R}_{\geq 0}\}$,
- $\mathcal{Z}$ is the set of zones over the set of clock variables in π, the zone $\top$ containing clock constraints $c \geq 0$ and $c - c \leq 0$ for all $c \in \pi$, and the *inconsistent zone* $\bot$,
- $Z_1 \sqsubseteq_{\mathcal{Z}} Z_2$ if and only if $(\!|Z_1|\!) \subseteq (\!|Z_2|\!)$,
- $\gamma_{\mathcal{Z}}(Z) = (\!|Z|\!)$. ▪

The abstraction function $\alpha_{\mathcal{Z}}$ corresponding to $\gamma_{\mathcal{Z}}$ is the least general abstraction $\alpha_{\mathcal{Z}}(A)$ of a set of clock valuations $A \in \mathcal{S}$, such that $\alpha_{\mathcal{Z}}(A)$ is a zone containing the clock constraints $c \leq \max_{val \in A} val(c)$, $c \geq \min_{val \in A} val(c)$, $c - c' \leq \max_{val \in A}(val(c) - val(c'))$, $c - c' \geq \min_{val \in A}(val(c) - val(c'))$ for each pair of clocks $(c \in V_C, c' \in V_C)$.

The transfer function for the zone domain is the abstract transfer function $T_{\mathcal{Z}} : \mathcal{Z} \times Op \times \Pi \to \mathcal{Z}$, which produces exactly one successor state and the concretization of the successor yields only states that are actually reachable from the source state in the concrete domain by the given operation (i.e. it does not introduce unreachable states):

$$\bigcup_{val_C \in \gamma_{\mathcal{Z}}(Z)} T_{concr}(val_C, op) = \gamma_{\mathcal{Z}}(T_{\mathcal{Z}}(Z, op, \pi))$$

2.4.4 Product abstraction

For systems that contain variables of different kinds (e.g. a timed automaton with data and clock variables), we use product abstraction [5] to handle different domains as one abstract domain. Here we define product abstraction for two abstract domains, however, the definition can easily be extended to allow products of more than two domains as well.

Definition 14 (Product abstraction). The product of two abstractions $D_1 = \langle \mathcal{S}_1, \mathcal{S}_{1a}, \sqsubseteq_1, \gamma_1 \rangle$ and $D_2 = \langle \mathcal{S}_2, \mathcal{S}_{2a}, \sqsubseteq_2, \gamma_2 \rangle$ is the domain $Prod(D_1, D_2) = \langle \mathcal{S}, \mathcal{S}_a, \sqsubseteq, \gamma \rangle$, where

- $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$,
- $\mathcal{S}_a = \mathcal{S}_{1a} \times \mathcal{S}_{2a}$,
- $(s_1, s_2) \sqsubseteq (s_1', s_2')$ if and only if $s_1 \sqsubseteq_1 s_1' \wedge s_2 \sqsubseteq_2 s_2'$, where $s_1, s_1' \in \mathcal{S}_{1a}$ and $s_2, s_2' \in \mathcal{S}_{2a}$,
- $\gamma((s_1, s_2)) = (\gamma_1(s_1), \gamma_2(s_2))$, where $s_1 \in \mathcal{S}_{1a}$ and $s_2 \in \mathcal{S}_{2a}$. ▪

Let T_1, T_2 denote the transfer functions for D_1, D_2. The transfer function for the product domain produces the successor states, which are the cartesian product of those states, that are produced by T_1 and T_2

2.5 CEGAR

Counterexample-guided abstraction refinement (CEGAR) [6] 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 an abstract state space is constructed, where a counterexample (e.g. a path to the unsafe state) may be encountered.

Upon encountering a counterexample, its feasibility in the concrete model has to be checked. If it is feasible, then the model indeed does not satisfy the requirement. However, if it is infeasible, then the counterexample is spurious, and the abstraction has to be refined in such a way that excludes the spurious counterexample.

If the abstract state space does not contain a counterexample, then the model is safe, since the abstract state space over-approximates the possible behaviors of the model.

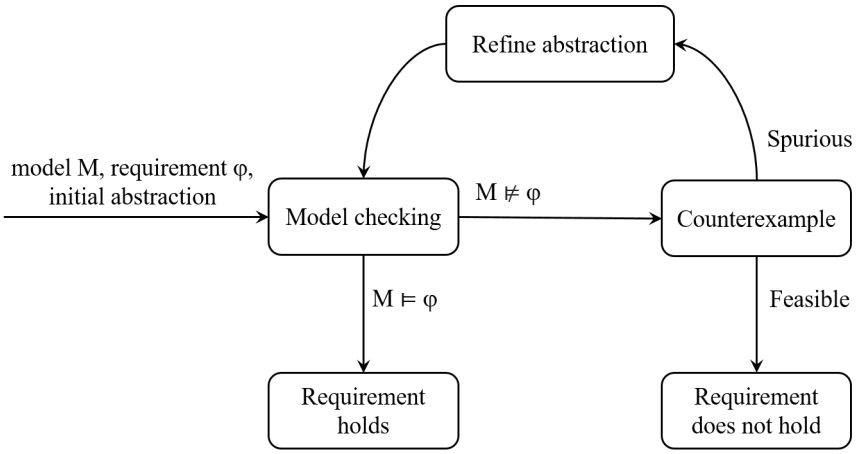

Figure 2.2: The CEGAR method

The abstract state space constructed in each iteration of the CEGAR algorithm is represented by an *abstract reachability graph*.

Let $D = \langle \mathcal{S}, \mathcal{D}, \sqsubseteq, \gamma \rangle$ denote the abstract domain used in the algorithm with T being the abstract transfer function for D.

Definition 15 (Abstract reachability graph). An abstract reachability graph is a tuple ARG $= \langle N, E, C \rangle$, where

- N is the set of *nodes*, consisting of a location and an abstract state,
- $E \subset N \times Op \times N$ is the set of directed *edges* representing transitions, an edge (n, op, n') expressing that n' is a successor of n with respect to operation op,
- $C \subset N \times N$ is the set of *covered-by edges*, a covered-by edge (n_1, n_2) expressing that all states reachable from n are also reachable from n'. ▪

In the CEGAR algorithm, the nodes of the ARG are labeled by a location and an abstract state of an abstract domain $D = \langle \mathcal{S}, \mathcal{S}_{abstr}, \sqsubseteq, \gamma \rangle$, obtained from a node by the following functions:

- $loc : N \to \mathcal{L}$ maps a node to the corresponding location,
- $s : N \to \mathcal{S}_{abstr}$ maps a node to the corresponding abstract state.

The CEGAR algorithm (Algorithm 1) is constructed as a loop consisting of an *abstraction* step (function Build, Section 2.5.1) and a *refinement* step (function Refine, Section 2.5.2).

Algorithm 1 Counterexample-guided abstraction refinement algorithm

```
1: function CHECK(M: TCFA, l_t: target location, D = ⟨S, S_abstr, ⊑, γ⟩: abstract domain,
   T_abstr: abstract transfer function over D, π_0: initial precision)
2:     π ← π_0
3:     arg ← (∅, ∅, ∅)
4:     repeat
5:         abstractorResult, arg ← BUILD(M, l_t, arg, π, D, T_abstr)
6:         if abstractorResult = unsafe then
7:             refinerResult, arg, π ← REFINE(arg, π)
8:     until abstractorResult = safe ∨ refinerResult = unsafe
```

2.5.1 Abstraction

The abstraction step (Algorithm 2) of the CEGAR algorithm corresponds to building an ARG of reachable abstract states with a given precision and determining whether any of the abstract states contains a counterexample for the given property to be checked.

The algorithm starts either with an empty ARG or (in later iterations) a partially constructed ARG. If the ARG is empty, then an initial set of nodes is created by an initialization function $\mathcal{I} : \Pi \to 2^{\mathcal{L} \times S_a}$ that over-approximates the initial states of the model. The algorithm maintains a waitlist for nodes to be processed. The main loop of the abstraction algorithm takes a node from the waitlist and first checks whether it violates the property. If it violates the property, then the path to the node is returned as a counterexample on the abstract states. Otherwise, the algorithm continues with attempting to create a covered-by edge from this node to a node in the set of already reached nodes. If no such coverage is possible, then the node is expanded, i.e. a new node is created for each successor state of the abstract state represented by the node.

Of course, this algorithm can be optimized, for example by checking whether a node violates the safety property as soon as it is reached in the expand step of its parent.

2.5.2 Refinement

If a counterexample was found in the abstract state space, CEGAR uses a refinement algorithm to check whether the path to this counterexample is present in the concrete state space or not. If the counterexample is infeasible according to the refinement algorithm (the path is not concretizable in the concrete state space), then more information need to be added to the precision, for the spurious counterexample to be excluded from the next iteration. If the path is feasible, then the corresponding concrete path is present in the concrete state space, so the model is unsafe.

We present a high-level algorithm for the refinement step (Algorithm 3), without the details of the implementation. In this algorithm, the function Concretize takes a feasible abstract counterexample and returns a concrete path to an unsafe state in that abstract counterexample. Function RefineAndPrune refines the precision with information obtained from the provided spurious abstract counterexample and prunes some nodes of the ARG

Algorithm 2 Constructing and checking an ARG

```
1: function BUILD(M: TCFA, l_t: target location, arg=⟨N, E, C⟩: ARG, π: precision,
   D=⟨S, S_abstr, ⊑, γ⟩: abstract domain, T_abstr: abstract transfer function)
2:     N ← N ∪ I(π)
3:     waitlist ← {n ∈ N | n is not covered and not expanded}
4:     while n ∈ waitlist for some n do
5:         waitlist ← waitlist\{n}
6:         if loc(n) = l_t then
7:             return unsafe, arg
8:         if loc(n) = loc(n') ∧ s(n) ⊑ s(n') for some n' ∈ N then
9:             C ← C ∪ {(n, n')}
10:        else
11:            for all (l, op, l') ∈ {enabled edge from loc(n) in M} do
12:                for all s' ∈ T_abstr(s(n), op, π) do
13:                    N ← N ∪ {s'}
14:                    E ← E ∪ {(s(n), op, s')}
15:                    waitlist ← waitlist ∪ {s'}
16:    return safe, arg
```

so that the spurious counterexample is removed and will be excluded from the following iterations. Some refinement techniques are detailed in [21] and [9].

Algorithm 3 Abstraction refinement

```
1: function REFINE(arg: ARG, π: precision)
2:     ψ ← abstract counterexample in arg
3:     if ψ is feasible then
4:         σ ← CONCRETIZE(ψ)
5:         return unsafe, arg, σ
6:     else
7:         ARG, π ← REFINEANDPRUNE(arg, π, ψ)
8:         return spurious, arg, π
```

2.6 Lazy abstraction

Besides the well-known CEGAR algorithm, there are other approaches to tackle the reachability problem in model checking. One of them is *lazy abstraction*. The lazy abstraction algorithm builds an ARG on-the-fly, without the alternating ARG building and refinement steps seen in the CEGAR algorithm. Abstraction refinement is performed occasionally (as needed) during the construction of the ARG.

As a configurable framework, lazy abstraction can work with various abstract domains and product domains composed of different abstract domains [12]. E.g. data can be represented by explicit abstraction or by first-order logic formulas while timing constraints can be represented by zone abstraction. The lazy abstraction algorithm enables working with zones without binding the level of abstraction to a precision as opposed to the abstraction step of the CEGAR algorithm. Lazy abstraction also enables the use of efficient abstraction refinement algorithms that do not use an SMT solver.

The algorithm may resemble the CEGAR algorithm (it maintains a waitlist of nodes, builds an ARG, checks for coverages, expands nodes, and performs abstraction refinement), even though they have fundamental differences.

Lazy abstraction uses abstract states that must not over-approximate the set of reachable concrete states. However, this granularity of abstract states would not allow many covered-by edges in the ARG. Because of this, lazy abstraction uses an additional abstract domain for each concrete domain, which is called *coverage domain*. This domain has the purpose of over-approximating of the set of reachable concrete states, which is useful when checking for coverage.

For the abstraction refinement algorithms to work, lazy abstraction can only use transfer functions that produce an exact successor state for a given operation. These constraints are satisfied by zone abstraction, however, it is not always the case with data domains, e.g. we cannot use havoc operations on data variables when using this algorithm.

To outline the preliminaries of this work more clearly, we present only a simplified version of the lazy abstraction algorithm, which works over models containing only clock variables, for which we will use zones as abstract states.

2.6.1 Lazy abstraction over the zone domain

For the abstract domain, $Zone(V_C, V_C)$ is capable of representing states with the required granularity. We can use $Zone(V_C, V_C)$ for the coverage domain as well, in this case allowing the over-approximation of concrete states. We denote the abstract domain with $Zone_{abstr} = \langle \mathcal{S}, \mathcal{Z}_{abstr}, \sqsubseteq_{\mathcal{Z}}, \gamma_{\mathcal{Z}} \rangle$ and the coverage domain with $Zone_{cov} = \langle \mathcal{S}, \mathcal{Z}_{cov}, \sqsubseteq_{\mathcal{Z}}, \gamma_{\mathcal{Z}} \rangle$. Both domains use the same preorder, hence we will use the $\sqsubseteq_{\mathcal{Z}}$ relation between zones belonging to different domains as well.

The lazy abstraction algorithm constructs an ARG of reachable states. The nodes of this ARG are labeled with a location and with a zone from both $Zone_{abstr}$ and $Zone_{cov}$. To obtain these labels of nodes, we define the following functions:

- $loc : N \to \mathcal{L}$ maps a node to the corresponding location,
- $z_{abstr} : N \to \mathcal{Z}_{abstr}$ maps a node to the corresponding zone in $Zone_{abstr}$, we refer to this zone as the *abstract zone* of node n,
- $z_{cov} : N \to \mathcal{Z}_{cov}$ maps a node to the corresponding zone in $Zone_{cov}$, we refer to this zone as the *coverage zone* of node n.

Definition 16 (Lazy transfer function). The lazy abstraction algorithm uses a transfer function $T_{lazy} : N \times Op \to N$, which uses the abstract transfer function $T_{\mathcal{Z}}$ of the zone domain, such that $T_{lazy}(n, op) = (l', T_{\mathcal{Z}}(z_{abstr}(n), op, V_C), \top)$ if an edge $(loc(n), op, l')$ exists in the model. ▪

Note that precision is irrelevant here, we always use all clock variables to compute the successors of a state on the abstract domain, while the successor on the coverage domain is always $\top$ (it may get refined later). We will omit the precision parameter when referring to the abstract transfer function $T_{\mathcal{Z}}$ in the lazy algorithm.

To ensure the correctness of the algorithm, we must construct the ARG in such a way that it is well-labeled, which means that the following conditions hold [24]:

1. initiation: $z_{abstr}(n_0) = \alpha_{\mathcal{Z}}(val_C^0)$ and $z_{abstr}(n_0) \sqsubseteq_{\mathcal{Z}} z_{cov}(n_0)$ where n_0 is the root node of the ARG and val_C^0 is the initial valuation of the TCFA,
2. simulation: $\forall n \in N : \ T_{\mathcal{Z}}(z_{abstr}(n), op) = \bot \ \Rightarrow \ T_{\mathcal{Z}}(z_{cov}(n), op) = \bot$,
3. consecution on $Zone_{abstr}$: $\forall (n, op, n') \in E : \ T_{\mathcal{Z}}(z_{abstr}(n), op) = z_{abstr}(n')$,
4. consecution on $Zone_{cov}$: $\forall (n, op, n') \in E : \ T_{\mathcal{Z}}(z_{cov}(n), op) \sqsubseteq_{\mathcal{Z}} z_{cov}(n')$,
5. coverage: $\forall (n, n') \in C : \ z_{cov}(n) \sqsubseteq_{\mathcal{Z}} z_{cov}(n')$.

The algorithm presented here is a version of the lazy abstraction algorithm that can explore the state space starting from either an empty or a partially constructed ARG. The algorithm maintains a waitlist of nodes to be processed and a list of passed nodes, i.e. nodes that are already fully expanded. The main loop of the algorithm, similarly to CEGAR, consumes nodes from the waitlist, and first checks if it is the target location (this can also be checked as soon as the node is reached for a more efficient algorithm). If it is the target location, then the model is unsafe without the need for any further checks, since only actually feasible nodes are created. Then the possibility of covering the abstract zone of the node with the coverage zone of another node is checked, with an additional abstraction refinement step if a covering node is found. For non-covered nodes, the algorithm proceeds to expand the node. During the expansion step, an abstraction refinement is performed for all disabled transitions from the abstract zone to ensure that it is disabled from the coverage zone as well.

In this algorithm, there is no loop of alternating ARG building and refinement steps, rather the ARG building is occasionally interrupted by a necessary refinement step.

Procedures Cover and Disable ensure that the ARG stays well-labeled. In the case of coverage, this is needed to ensure the well-labeledness condition for coverage. When encountering a disabled transition, abstraction refinement is performed to ensure the simulation condition. Unfortunately, refining the coverage zone only in the directly involved node is not enough, as this may produce an ARG that does not satisfy the condition of consecution on the coverage domain. Because of this, these are complex algorithms that traverse the ARG backward on the path toward the root node to ensure well-labeledness. These algorithms are detailed in [24] and [12].

Note that this algorithm builds an inductive ARG on both $Zone_{abstr}$ and $Zone_{cov}$ domains, but the set of reachable states in the model is over-approximated by only the coverage domain.

Algorithm 4 Lazy abstraction algorithm

1: **function** CHECK(M: TCFA, l_t: target location, arg: ARG)
2: **if** $N = \emptyset$ **then**
3: $N \leftarrow \{(l_0, \alpha_{\mathcal{Z}}(val_C^0), \top)\}$
4: $waitlist \leftarrow \{n \in N \mid n \text{ is not covered and not expanded}\}$
5: $passed \leftarrow \{n \in N \mid n \text{ is expanded}\}$
6: **while** $n \in waitlist$ for some n **do**
7: **if** $loc(n) = l_t$ **then**
8: **return** unsafe, arg
9: // try to cover
10: **if** $loc(n) = loc(n') \wedge z_{abstr}(n) \sqsubseteq_{\mathcal{Z}} z_{cov}(n')$ for some $n' \in passed$ **then**
11: $C \leftarrow C \cup \{(n, n')\}$
12: COVER(n, n')
13: // expand
14: **if** n is not covered **then**
15: **for all** (l, op, l') outgoing edge from $loc(n)$ in M **do**
16: $n' \leftarrow T_{lazy}(n, op)$ // $z_{cov}(n') = \top$
17: **if** $\gamma_{\mathcal{Z}}(z_{abstr}(n')) = \emptyset$ **then**
18: DISABLE(n, op)
19: **else**
20: $N \leftarrow N \cup \{n'\}$
21: $E \leftarrow E \cup \{(n, op, n')\}$
22: $waitlist \leftarrow waitlist \cup \{n'\}$
23: $passed \leftarrow passed \cup \{n\}$
24: **return** safe, arg

Chapter 3

Overview

In this chapter, we present our framework to support the verification of software-intensive systems. Our goal is to analyze real-time systems that have data and time-dependent behavior so the analysis methods need to take into consideration both aspects. In our approach (Figure 3.1), we assume that we have a real-time system (RTS for short) with both data and clock variables and a safety property. This model is often derived from high-level engineering models. The verification task is to prove that the system satisfies the given property.

Formal verification algorithms are utilized to answer the verification query. The first step in the verification process is to derive a formal model (in this case, a Timed Control Flow Automaton) via a series of model transformation steps.

The first model transformation step takes the RTS and the safety property and generates a modified RTS encompassing the property by generating a witness process and error locations in the RTS. The modified RTS is safe (based on the reachability of the error location) if the input RTS satisfies the property, or the modified RTS is unsafe if the input RTS does not satisfy the property. The property preserving transformation is detailed in Section 3.1.1.

Next, we map the modified RTS to a TCFA. The RTS supports high-level elements such as synchronization, urgent and committed locations, which makes it easy to model a real-world system in RTS, but they introduce unnecessary complexity in the formal verification process. To combat this issue, we generate a low-level formal model from the RTS, whose behavior is equivalent to the RTS. We describe this model transformation in Section 3.1.2.

Our formal verification methods can be applied on the TCFA formalism, so they are independent of the high-level RTS formalism. The TCFA formalism provides flexibility as any kind of higher-level formalism can be verified through TCFA: we only need a mapping from the high-level formalism to it.

Given a TCFA, we are able to run formal verification algorithms to determine whether the TCFA is safe or unsafe. We extended the Theta open source, modular, formal verification framework to be able to execute the implemented verification algorithms.

We provide lazy abstraction, which is efficient in handling time-dependant behaviour as it provides efficient abstractions. On the other hand, lazy abstraction has significant limitations on the data-dependant behaviour. However, lazy abstraction can solve many problems efficiently, so it is a valuable part of our portfolio. We integrated the lazy abstraction into Theta, and we used this implementation as a baseline to compare our novel approach.

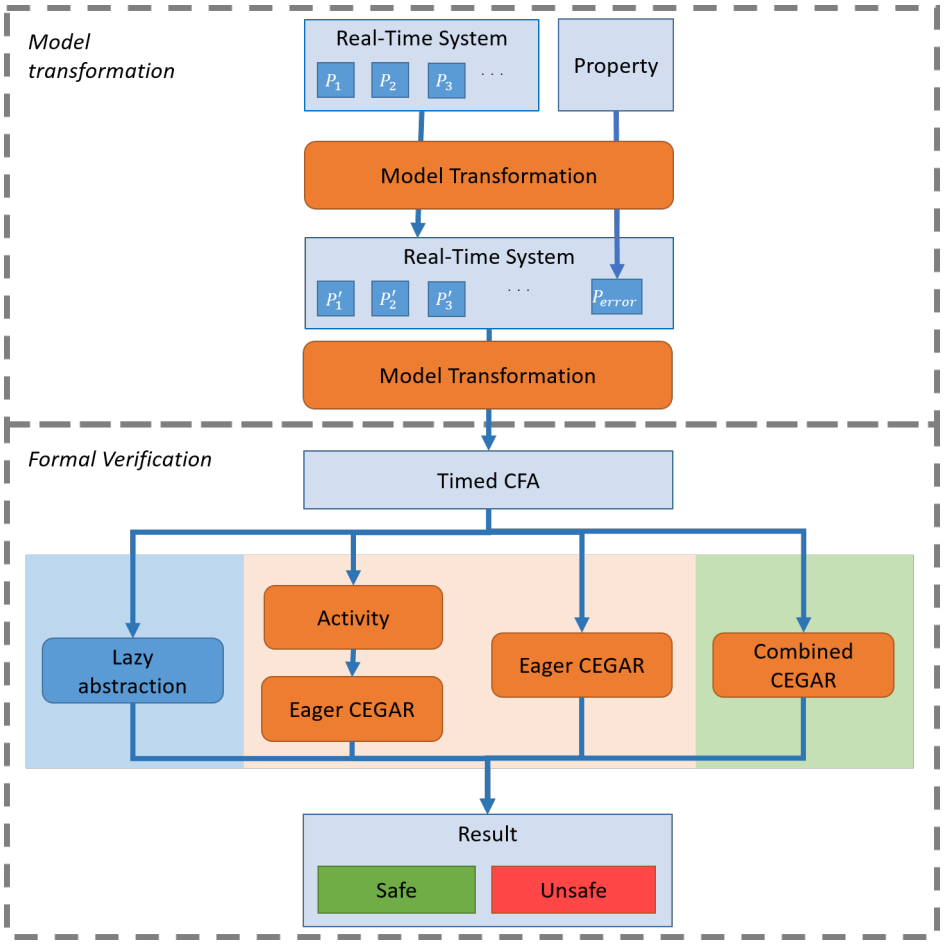

Figure 3.1: Overview of our approach

We implemented two algorithms from the literature based on the eager CEGAR approach. These algorithms can handle data efficiently thanks to the CEGAR-based refinement loop. These algorithms use zone abstraction for time, but while one follows all clock variables, the other performs an activity-based preprocessing step that reduces the number of clock variables during verification. Unfortunately, zone-based handling of clock variables provide a fine grained abstraction preserving too much information during verification, so these algorithms perform better for data than for time. We elaborate on these algorithms in Chapter 4.

Finally, we introduce a novel algorithm that combines the efficient data abstraction strategies of eager CEGAR with the efficient clock abstraction methods of lazy abstraction. This new algorithm is capable of overcoming the difficulties plaguing both lazy abstraction and eager CEGAR when it comes to the verification of real-time software systems that have both data and time-based behavior. We describe the combined approach in Chapter 5.

3.1 Mapping Real-Time Systems to TCFA

We define the algorithms in this work to be able to check whether an error location in a *TCFA* is reachable or not. Therefore, we need to create a method to transform a *Real-Time System* and a property to a *TCFA* with a set of error locations.

3.1.1 "Error" locations in Real-Time Sytems

We introduce *error locations* as a new type of location in an XTA. So the new definition of an $XTA = \langle \mathcal{L}, l_0, v_0^{loc}, \mathcal{D}_{loc}, \mathcal{C}_{loc}, I, E, T, l_{err} \rangle$, where l_{err} is an optional error location (not every process has to have an error location) and everything else is the same as in Definition 2. The connection between the safety of the system and the reachability of an error location depends on the type of property.

We reduce the verification problem with PSL to a search problem. A new process is added to the system, that has an initial and an error location. There is only one edge e in this process from the initial location to the error location, with a single guard, which is the state property with a little modification: if a location is in the state property, it is replaced with a boolean variable that is true if and only if the state, which the system is in, contains the corresponding location. Let *prop* denote this modified property. The guard g of e depends on the type of the state property:

- $g = prop$ if the property's type is *possibly* and the system is safe if an error location is reachable because the system can be brought to a state, where the property holds
- $g = \neg prop$ if the property's type is *invariantly* and the system is unsafe, if the error location is reachable because the system can be brought to a state, where the property not holds

The extra process is not enough to check reachability, because if the system is in a state that contains some *committed locations* and no edge from it to an *error location*, but g holds, the next transition must include an outgoing edge of one of the committed locations, and after this transition, g may never hold again. It is not an option to make the initial location of the extra process committed, because it can cause a deadlock - if g is not satisfied and there are no other committed locations in the state then no edge can be fired. Therefore, each process containing at least one committed location needs to be extended with an error location, and from every committed location, an outgoing edge with guard g must be added to the error location. Thus an error location is reachable from every committed location if the property under analysis holds in the committed location.

3.1.2 Converting *RTS* with Error Locations to *TCFA*

In this section, we use the notations from Definition 1, and Definition 4. The goal of this section is to give a method to transform a *Real-Time System* $RTS = \langle P, \mathcal{D}_g, \mathcal{C}_g, S \rangle$ to a *Timed Control Flow Automaton* $TCFA = \langle \mathcal{L}, l^0, V_D, V_C, val_D^0, val_C^0, E \rangle$. We use L to refer to the location set of the state of the RTS. D_i^{loc} denotes the data variables, and C_i^{loc} denotes the clock variables and L_i denotes the locations of $p_i \mid i \in \{1, 2, \ldots, |P|\}$. In the following list we show how to calculate the parts of the model:

- $V_D = \bigcup_{i=0}^{|P|} D_i^{loc} \cup \mathcal{D}_g$
- $V_C = \bigcup_{i=0}^{|P|} C_i^{loc} \cup \mathcal{C}_g$
- val_D^0 is the initial value of V_D.
- val_C^0 is a clock valuation that maps V_C to 0

- $\mathcal{L} = L_1 \times L_2 \times \cdots \times L_{|P|}$
- $E = \mathcal{L} \times Op \times \mathcal{L}$
- l_0 is the initial location that is the cartesian product of the initial locations of all processes

The edge set of the TCFA consists of edges representing *timed transitions* and edges representing *discrete transitions*, as defined in 2.2. The TCFA has an edge $e_t = \langle loc, op_t, loc' \rangle$, where loc, loc' have $|P|$ components with the ith component being from L_i, if one of the following conditions groups holds:

- $\exists i \in 1 \dots |P|$: there is an edge in the RTS $e = \langle l_i, op = \langle g_i, a_i, r_i, \rangle, \tau, l_i' \rangle$ in the ith process, $\forall k \neq i \in 1 \dots |P|$: the kth component of loc and loc' are the same, the ith component of loc' is l_i' and $op_t = \langle g_i \wedge I(l_i'), a_i, r_i \rangle$
- $\exists s \in S_n : \exists i \in 1 \dots |P|$: there is an edge in the RTS $e = \langle l_i, op = \langle g_!, a_!, r_!, \rangle, s!, l_i' \rangle$ in the ith process, $\exists j \in 1 \dots |P|$: there is an edge $e = \langle l_j, op = \langle g_?, a_?, r_?, \rangle, s?, l_j' \rangle$ in the jth process, $\forall k \neq i, j \in 1 \dots |P|$:,the kth component of loc and loc' are the same, the ith, jth component of loc' is l_i', l_j' and $op_t = \langle g_! \wedge g_? \wedge I(l_i') \wedge I(l_j'), a_! \cup a_?, r_! \cup r_? \rangle$
- $\exists s \in S_b :$ $\exists i \in 1 \dots |P|$: there is an edge in the RTS $e = \langle l_i, op = \langle g_!, a_!, r_!, \rangle, s!, l_i' \rangle$ in the ith process, let $J = \{j \mid$ there is at least one outgoing edge from l_j labeled with $s?\}$, then $\forall j \in J$: there is an edge $e_j = \langle l_j, op = \langle g_j, a_j, r_j \rangle, s?, l_j' \rangle$, the ith component of loc' is l_i', for all $j \in J$ the jth component of loc' is l_j', every other component in loc, loc' are the same, $op_t = \langle g_i \wedge \bigwedge_{j \in J} g_j \wedge \bigwedge_{j \in J} I(l_j'), a_! \cup \bigcup_{j \in J} a_j, r_! \cup \bigcup_{j \in J} r_j \rangle$

If there is at least one committed location in the components of loc, then l_i or l_j above must be a committed location.

In a timed transition, every clock variable is increased by d, an arbitrarily large real number, so there is an edge $\langle l, op, l \rangle$ for every $l \in \mathcal{L}$ and op contains a guard that is the conjunctions of the invariants of the XTA locations in l and increases every value of clock variables $c \in V_C$ by an arbitrary value d (non-deterministically chosen by the concrete transition function). The error location l_{err} in the TCFA has an incoming edge $e = l \times l_e rr$ where l contains an error location from the RTS, and e is the only outgoing edge from l.

Our algorithms can be run on everything that can be mapped to a $TCFA$. In this work, we focused on Real-Time systems from $UPPAAL$ verification software, but system models and real-time software can be mapped to TCFA as well.

Chapter 4

CEGAR with clock activity

We introduce another abstraction to reduce the number of clock variables in *Zone abstraction* without losing any information that affects state space. This abstraction is called *active zone abstraction* [14], and its main idea is omitting those clock variables from the zone that do not affect any future operations. A clock c is active at a location l, denoted by $c \in Act(l)$, if c is part of invariant $I(l)$, appears in the guard of some outgoing edge of l, or $c \in Act(l')$ where l' is a location reachable from l and $c \notin r$, and r is the set of clocks that are reset on the outgoing edge of l.

Definition 17 (Global Zone abstraction). *Global Zone abstraction* over the set of clock variables $V_C = \{c_1, c_2, \ldots, c_{|V_C|}\}$ is the abstract domain $GlobalZone(V_\mathcal{C}) = Zone(V_\mathcal{C}, V_\mathcal{C})$ where *Zone* is an abstract domain (defined in Definition 13), ▪

Definition 18 (Active Zone abstraction). *Active Zone abstraction* over the set of clock variables $V_C = \{c_1, c_2, \ldots, c_{|V_C|}\}$ and a precision $\pi \subseteq V_C$ is the abstract domain $ActiveZone(V_\mathcal{C}, \pi) = Zone(V_\mathcal{C}, Act(l))$ where *Zone* is an abstract domain (defined in Definition 13), and $Act(l)$ is the set of clock variables, which may affect future operations, so this abstraction depends on the locations. ▪

Active clocks of a location can be calculated using the iterative algorithm described below, where $Act(l)$ is the set of active clock variables of l, and $clk : \mathcal{B}(\mathcal{C}) \to 2^\mathcal{C}$ assigns to each clock constraint the set of clocks appearing in it

Algorithm 5 Activity calculation

```
1:  function GET_ACTIVE_CLOCKS(xta : XTA)
2:      for all l ∈ {location in xta} do
3:          Act_0(l) ← clk(I(l))
4:          for all (l, a, g, r, s, l') ∈ {every outgoing edge from l} do
5:              Act_0(l) ← {Act_0(l) ∪ clk(g)}
6:      i ← 1
7:      do
8:          for all l ∈ {locationin}xta do
9:              for all (l, a, g, r, s, l') ∈ {every outgoing edge from l} do
10:                 Act_i(l) ← {Act_{i−1}(l) ∪ Act_{i−1}(l')\r}
11:     while Act(l)_{i−1} ≠ Act(l)_i for every l in xta
```

In the $0th$ iteration $Act(l)$ gets every clock variable appears in the invariant of location and every clock variable appears in the guard of each outgoing edge. Then from the first

iteration $Act(l)$ is expanded with every element of the active clocks of the target location of each outgoing edge excluding the clocks, that is reset in the outgoing edge. The loop stops when a fix-point is reached, so when $Act(l)$ is not expanded for any l.

In the CEGAR algorithm we need to use an abstraction over the data variables, and one over the clock variables. Therefore, we used product abstraction. We use *Zone abstraction* over clock variables, and *Predicate* or *Explicit abstraction* over data variables. CEGAR creates an ARG of a TCFA, where the abstract states are labeled with a location, and a product state of a data state that abstracts over data variables and a zone that abstracts over clock variables. To build the ARG we can use Algorithm 2, to check whether the error location is reachable or not. We can reduce the tracked clock variables in the *zone* if we use Active Zone abstraction, so we expect a reduced abstract state space, which manifests in an increased number of coverage edges in the ARG. We show that in the example below that we need fewer nodes to be expanded in the ARG building algorithm if we use Active Zone abstraction.

Example 1. *Figure 4.2 is the ARG constructed by Active Zone abstractor for the TCFA in Figure 4.1. The construction starts with a preprocess that calculates Act(l) for each location in the TCFA using Algorithm 5. The TCFA has x, y, z clock variables and*

- $Act(L_1) = \{z\}$
- $Act(L_2) = \{x, z\}$
- $Act(L_3) = \{x, y, z\}$
- $Act(L_4) = \{x\}$

When the ARG is expanded from L_3 with two nodes, one is labeled with L_4 and the other one is labeled with L_3. Both of them can be covered by other nodes, but with Global Zone abstraction we would not have the red coverage arrow, because z would appear in the clock constraints in each node. The upper abstract node labeled with L_4 would have a constraint: $z > 0$ and the lower one would have a constraint $z > 1$, so the lower one does not imply the upper one. It follows that both of them need to be expanded, but with Active Zone Abstraction only the upper one needs to be expanded.

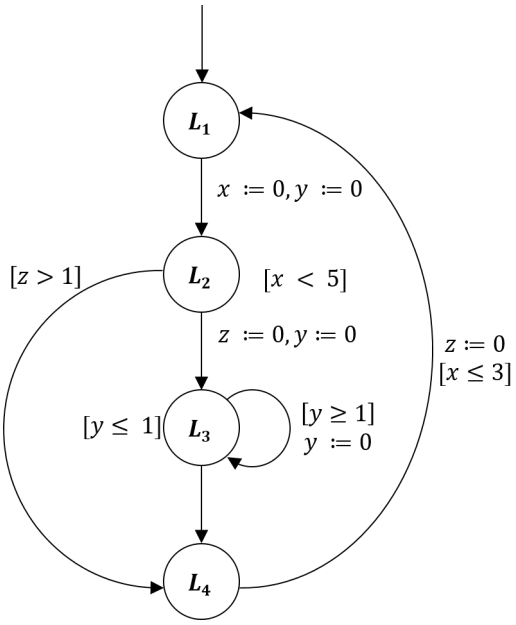

Figure 4.1: Example TCFA

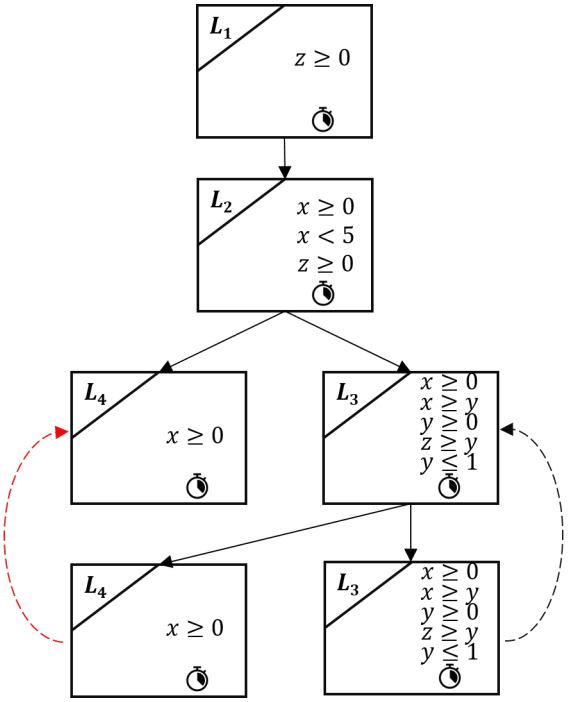

Figure 4.2: ARG constructed by Active Zone abstractor

Chapter 5

Combined CEGAR

TCFA models contain both data and clock variables. In the following, we overview both those algorithms that we integrated to the framework, and also our novel verification algorithms. These algorithms can now be used in the framework for timed verification.

Eager CEGAR

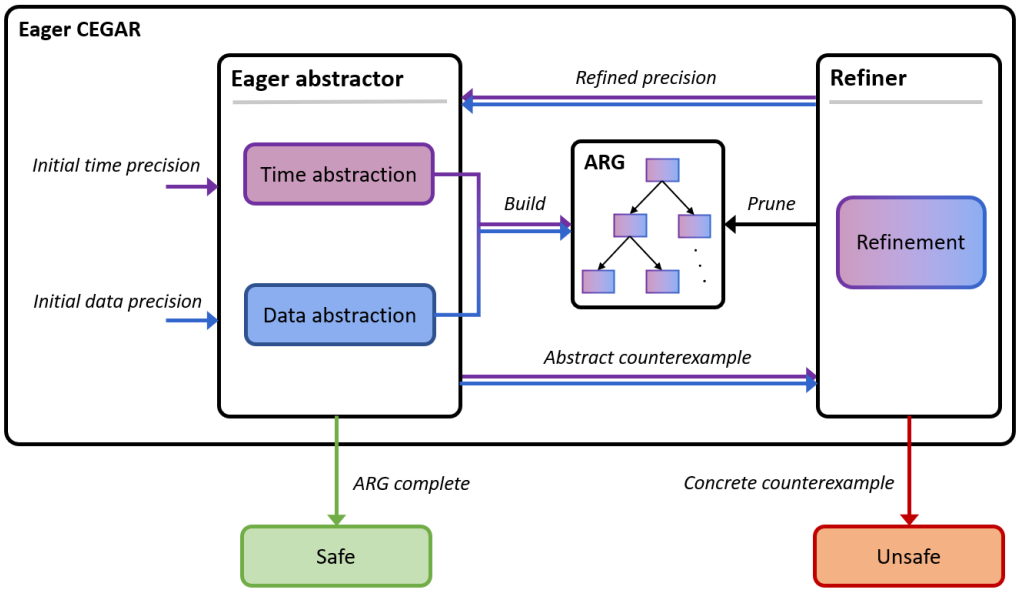

Figure 5.1: Eager CEGAR

Our first option is to use the (eager) CEGAR algorithm. In that case, an eager abstractor builds the ARG with a common precision for both time and data and an eager refiner is responsible for refining the precision and pruning the ARG. With this option, we have efficient abstract domains for data, e.g. explicit value abstraction, and predicate abstraction. On the other hand, we do not have any efficient options for representing time. We can use visible clock abstraction, or active clock abstraction, although both of them are based on removing all information about some of the clocks in the abstract states. In our experience, neither of them proved to be efficient (see evaluation in Chapter 6).

Lazy abstraction

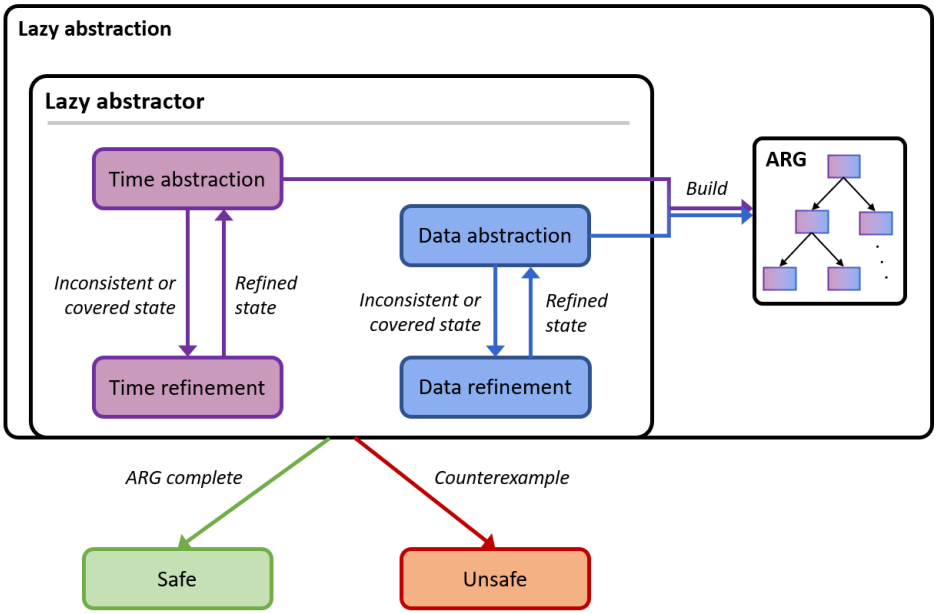

Figure 5.2: Lazy abstraction

Instead of the eager CEGAR algorithm, we could use lazy abstraction. In this case, we do not need a global precision to be able to use abstraction, instead, precision is handled locally in each ARG node. This is advantageous for the timed component, as we can utilize the ability of zones to store some (but not all) information about a clock. We also have efficient abstraction refinement algorithms for it. Lazy abstraction *can* be efficient on the data domain as well, however, non-deterministic operations can hinder this, as we can not give an efficient data domain where non-deterministic operations are supported. [15] Therefore, lazy abstraction is not suitable for the verification of some models, even though lots of models contain non-deterministic operations in practice, e.g. handling user input.

We could turn back to the eager CEGAR option when verifying these models; instead, we propose a novel approach that combines the advantages of the eager and lazy algorithms to give a more efficient solution, which is the combined CEGAR algorithm.

Combined CEGAR

Previously we have seen that the CEGAR algorithm handles discrete data efficiently but it cannot handle timing very well, on the other hand, the efficiency of the lazy algorithm for zone abstraction is outstanding but it also has its weaknesses when it comes to handling data variables, it severely limits what operations we can use in the data domains. To allow the efficient verification of as many models as possible, we combine them into a *Combined CEGAR* algorithm, so that discrete data is handled by an eager CEGAR algorithm, while timing is handled by lazy abstraction. To do so, we run a lazy abstraction algorithm inside a CEGAR algorithm, in place of its abstraction step.

This abstraction algorithm runs with a fixed precision for data, obtained from the last refinement step of the CEGAR algorithm. For data abstraction, it uses the abstract transfer function for the eager algorithm, which allows states that over-approximate the

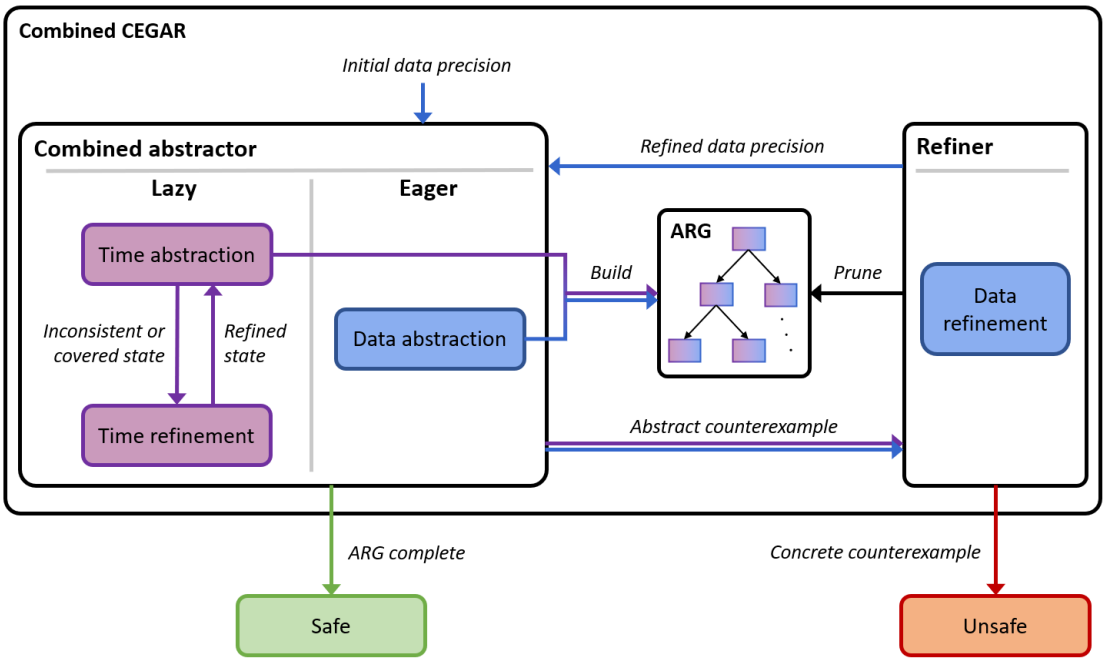

Figure 5.3: Combined CEGAR with lazy abstraction

set of reachable concrete states, as opposed to the lazy algorithm. For time abstraction the lazy transfer function and abstraction refinement are used, as described previously.

In this combined algorithm reaching inconsistent states is possible, as successors are computed for all outgoing edges as in the lazy abstraction algorithm. When encountering a state with an inconsistent time component, the simulation condition of well-labeledness has to be ensured, while states with an inconsistent data component can simply be skipped. Moreover, when adding a covered-by edge, there are no further actions needed to be performed on the data component of states, as it already satisfies the requirement for coverage, while in the timed component it has to be ensured that the coverage domain satisfies the coverage condition of well-labeledness in the lazy algorithm. Therefore, abstraction refinement caused by an inconsistent state or a coverage involves only the time domain, so in these cases, we can use the abstraction refinement algorithms presented previously for lazy abstraction as they are.

When reaching an unsafe state in the combined algorithm, the counterexample projected to the timed component of states is always concretizable, since we run the lazy algorithm on that part. However, when projected to the data domain, we get an abstract counterexample, which may or may not be concretizable. Therefore, when encountering an unsafe abstract state, we do not deem the automaton unsafe immediately but instead, use the refiner of the CEGAR algorithm. The refiner runs as usual in the CEGAR algorithm. The result of the refinement is always either a concrete counterexample or only the data precision being refined since the lazy algorithm does not generate spurious counterexamples.

The ARG built with the combined algorithm is similar to that of the lazy algorithm, however, we have to label the nodes with abstract data states as well. With $D = \langle \mathcal{S}, \mathcal{D}, \sqsubseteq_{\mathcal{D}}, \gamma_{\mathcal{D}} \rangle$ being the abstract domain used for data, let $d_{abstr} : N \to \mathcal{D}$ denote the function that maps a node to the corresponding abstract data state.

Definition 19 (Combined transfer function). Let $T_{\mathcal{D}}$ denote the abstract transfer function used in the combined algorithm for the data domain D. The combined algo-

rithm uses a transfer function $T_{combined} : N \times Op \times \Pi \to 2^N$, which uses transfer functions $T_{\mathcal{D}}$ and $T_{\mathcal{Z}}$. With the decomposition of a node n as $(loc(n), d_{abstr}(n), z_{abstr}(n), z_{cov}(n))$, the successors of a node n with respect to operation op and precision π are $T_{combined}(n, op, \pi) = \{(l', s_d, s_z, \top) \mid s_d \in T_{\mathcal{D}}(d_{abstr}(n), op, \pi),\ s_z = T_{\mathcal{Z}}(z_{abstr}(n), op)\}$ if an edge $(loc(n), op, l')$ exists in the model. ▪

The combined algorithm uses the same outer CEGAR loop as presented in Algorithm 1. The refinement algorithms used in eager CEGAR can also be used here. Here we present the ARG building step of the combined algorithm, which is slightly different than the ones presented before. It is a lazy abstraction algorithm, modified to handle both data and timing as described here. The highlighted parts show where this algorithm differs from the lazy abstraction algorithm presented previously.[1] We refer to this part of the combined CEGAR algorithm as a combined abstraction.

Algorithm 6 Build an ARG in the combined CEGAR algorithm

1: **function** BUILD(M: TCFA, l_t: target location, arg: ARG, π: precision, $D = \langle \mathcal{S}, \mathcal{D}, \sqsubseteq_{\mathcal{D}}, \gamma_{\mathcal{D}} \rangle$: abstract data domain, $T_{\mathcal{D}}$: abstract transfer function for D)
2: $N \leftarrow N \cup \{(l_0, i_d, i_z, \top) \mid i_d \in \mathcal{I}(\pi),\ i_z = \alpha_{\mathcal{Z}}(val^0_C)\}$
3: $waitlist \leftarrow \{n \in N \mid n \text{ is not covered and not expanded}\}$
4: $passed \leftarrow \{n \in N \mid n \text{ is expanded}\}$
5: **while** $n \in waitlist$ for some n **do**
6: **if** $loc(n) = l_t$ **then**
7: **return** unsafe, arg
8: **if** $loc(n) = loc(n') \wedge d_{abstr}(n) \sqsubseteq_{\mathcal{D}} d_{abstr}(n') \wedge z_{abstr}(n) \sqsubseteq_{\mathcal{Z}} z_{cov}(n')$ for some $n' \in passed$ **then**
9: $C \leftarrow C \cup \{(n, n')\}$
10: COVER(n, n')
11: **if** n is not covered **then**
12: **for all** (l, op, l') outgoing edge from $loc(n)$ in M **do**
13: **for all** $n' \in T_{combined}(n, op, \pi)$ **do**
14: **if** $\gamma_{\mathcal{Z}}(z_{abstr}(n')) = \emptyset$ **then**
15: DISABLE(n, op)
16: **else if** $\gamma_{\mathcal{D}}(d_{abstr}(n')) = \emptyset$ **then**
17: continue
18: **else**
19: $N \leftarrow N \cup \{n'\}$
20: $E \leftarrow E \cup \{(n, op, n')\}$
21: $waitlist \leftarrow waitlist \cup \{n'\}$
22: $passed \leftarrow passed \cup \{n\}$
23: **return** safe, arg

Example 2. *Figure 5.5 illustrates the ARG constructed by the combined abstractor for the TCFA in Figure 5.4 in the first iteration of the combined CEGAR algorithm. In this example, explicit value abstraction is used for data. The algorithm starts with an empty precision, i.e. no data variables are tracked. The abstract zone of the initial node of the ARG contains information available from the initial clock valuation. In accordance with the presented algorithm, both the abstract data state and the coverage zone are* $\top$*. The*

[1] Note that there are only a few changes. These can be handled well by writing a generic lazy abstraction algorithm (see [12]), with configurations that provide different implementations for the domain-specific parts, such as the marked places here.

transition from L_1 to L_2 sets x_2 to 0 but the next abstract data state is still $\top$, as x_2 is not tracked. The transition also resets clock c_1, hence the value of c_1 is at most the value of c_2 in the next state. The result of the next transition (from L_2 to L_3) is computed similarly. From this third node the transition with the guard $[c_1 < 1 \wedge c_2 \geq 1]$ is disabled because $c_2 \leq c_1$. The disabled transition triggers an abstraction refinement on zones, setting the coverage zone of this node to $c_2 \leq c_1$. The transition resetting c_2 is enabled, and a new node is created, which is then covered by its parent node since their locations are the same and the abstract zone of the child is over-approximated by the coverage zone of the parent. To ensure that coverage is satisfied on the coverage domain, the coverage zone of the covered node is refined. From L_3 there is a third outgoing edge as well, to the error location L_4, with the guard $[x_1 > 1]$. As x_1 is not tracked, we must assume that this transition is enabled, and the combined abstractor returns an unsafe result.

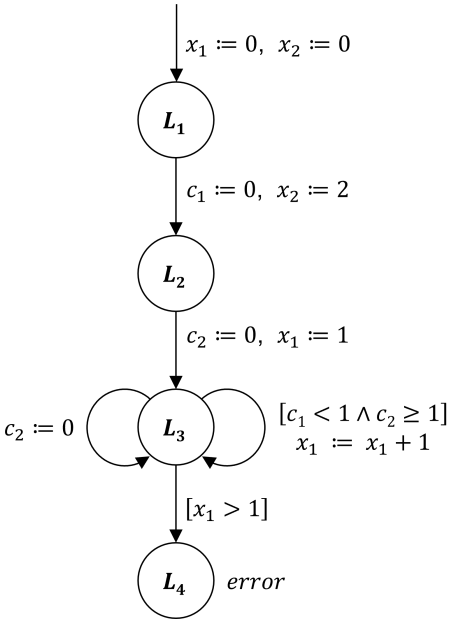

Figure 5.4: Example TCFA

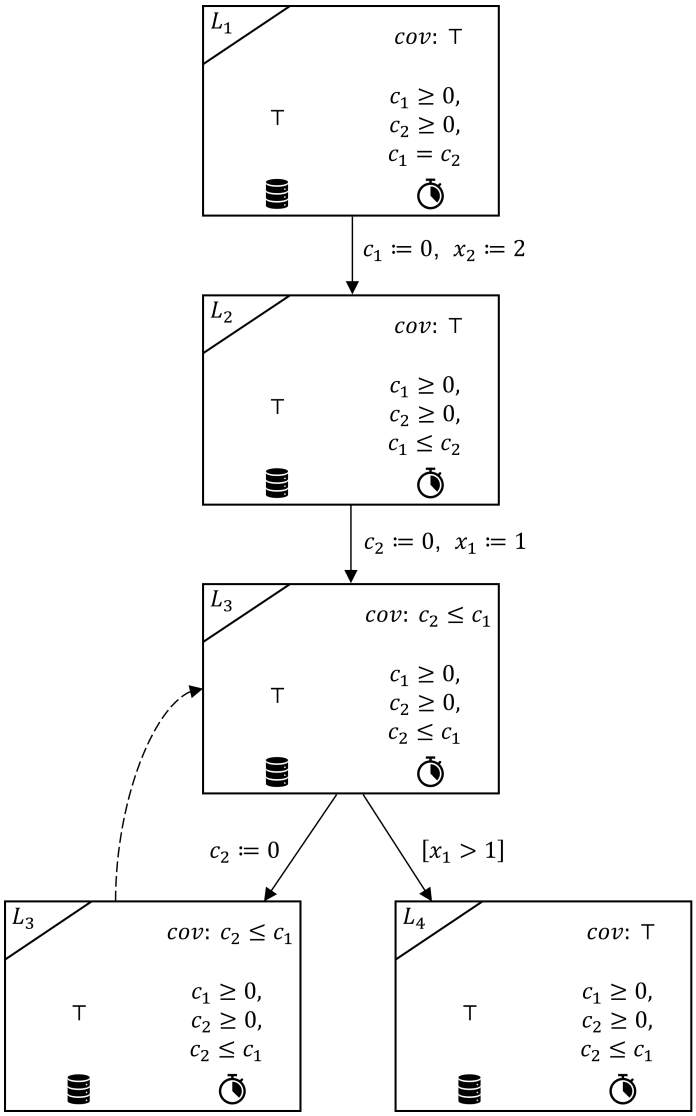

Figure 5.5: ARG constructed by the combined abstractor

5.1 Pruning strategies

We experimented with two different strategies for pruning the ARG at the end of the data refinement step. The most straightforward strategy is to erase the ARG completely and build an entirely new one. We call this strategy *full pruning*. However, full pruning seems inefficient, as no information is preserved that might be useful in the next iteration.

To reuse information, we tried another strategy that we call *lazy pruning*. Lazy pruning determines which node is the first in the provided counterexample where precision should be refined to exclude the spurious counterexample. The subtree of this node is then removed from the ARG since they contain states that are actually infeasible. This refinement strategy requires additional steps to keep the ARG correct. The parent node of the subtree should be removed from the set of passed nodes and put back in the waitlist, as it has an enabled transition for which there is no corresponding child node now. Moreover, all nodes should be put back in the waitlist that covered or was covered by a node in the removed subtree. The algorithm can then continue with the ARG building step with this pruned ARG and the refined precision. Even though the data precision is refined, the information stored in the already existing parts of the ARG is not modified, and because of this it can happen that we get the same counterexample in the next iteration, i.e. the algorithm becomes stuck. When this happens, we do full pruning instead of lazy pruning, then continue the algorithm normally.

Example 3. *The left side of Figure 5.6 illustrates the continuation of the previous example after data abstraction refinement and lazy pruning. The refiner concluded that x_1 should be tracked, and prunes the ARG, leaving only the first two nodes, as the value of x_1 should be set in the transition from the second ARG node. The highlighted parts of the figure are added to the ARG in the second iteration of the combined CEGAR loop. The transition from L_2 to L_3 sets the value of x_1 to 1 and this time we track x_1, so the abstract data state of the successor ARG node is $x_1 = 1$. Regarding the time component, everything is the same as in the previous iteration. From the third ARG node, the transition with guard $[c_1 < 1 \wedge c_2 \geq 1]$ is disabled again, triggering the abstraction refinement, setting the coverage zone of the third node to $c_2 \leq c_1$. The transition resetting c_2 creates a new ARG node (this time with abstract data state $x_1 = 1$), which is then covered, just as in the previous iteration. However, this time the transition from the third ARG node to L_4 is disabled, as now we know that $x_1 > 1$ cannot be true since $x_1 = 1$. At this point all nodes are either expanded or covered, there are no more nodes to be processed; the combined algorithm returns that the model is safe.*

The right side of the figure shows the ARG constructed after full pruning. This leads to the same result, although in this case the whole ARG has to be built again and the value of x_1 is tracked everywhere in the ARG.

5.2 Correctness of the combined CEGAR algorithm

Lastly, we show that the combined CEGAR algorithm is correct, i.e. if it terminates, then it concludes that the model is safe if and only if an unsafe state can not be reached in the model. For this let us assume that the lazy abstraction algorithm presented in this work for zone abstraction builds a well-labeled ARG and that a well-labeled ARG preserves reachable states. Proofs for both statements are given in [24].

With the handling of the data components removed from the combined abstraction algorithm, the resulting algorithm is the previously presented lazy abstraction algorithm for

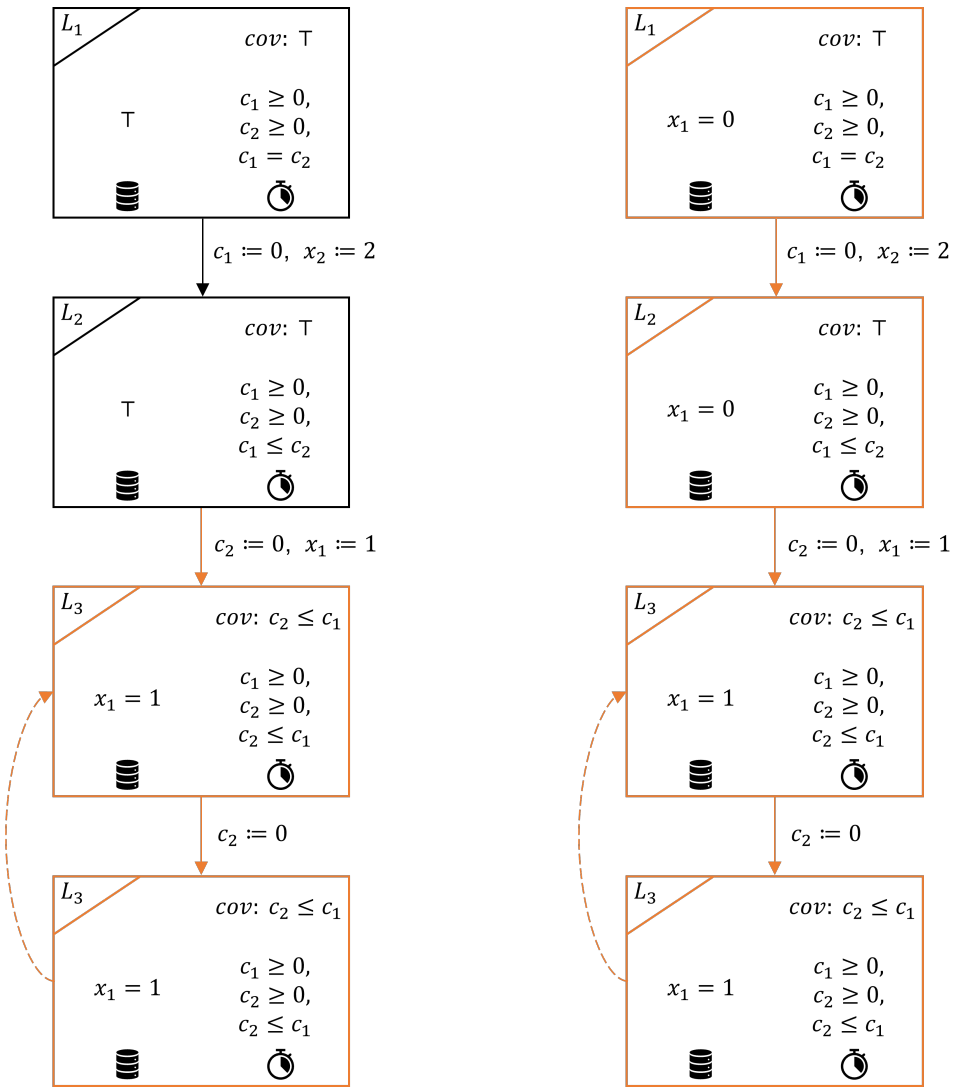

Figure 5.6: ARG constructed after data abstraction refinement and lazy pruning (left) or full pruning (right)

the zone domain: (1) the set of initial states without the initial states for data abstraction consists of a single state, equivalent to the initial state of the lazy algorithm, (2) the check for coverage without checking the data part becomes the same as in the lazy algorithm, (3) it is also easy to see that the abstract transfer function $T_{combined}$ becomes T_{lazy} without the data component. From our previous assumption, the lazy abstraction algorithm builds a well-labeled ARG. We show that re-adding the data components preserves the well-labeledness on the timed projection of the ARG.

Even though the initialization function for the data domain might produce more than one initial state, $z_{abstr}(n_0)$ will be $\alpha_{\mathcal{Z}}(val_C^0)$ for each initial node n_0, and $z_{cov}(n_0)$ will always be $\top \sqsupseteq \alpha_{\mathcal{Z}}(val_C^0)$. On the timed projection of the ARG this means that there might be multiple identical initial nodes, however, assuming the presence of only one initial node identical to the original initial nodes with a no-op operation to all original ones transforms it to an ARG where there is only one initial node, it covers exactly the same concrete states as the union of the ARGs built separately from the original initial nodes, and it satisfies the condition of initiation.

The successor state of the abstract zone of each node is computed with the same $T_{\mathcal{Z}}$ abstract transfer function in both combined and lazy algorithms, so consecution on $Zone_{abstr}$ is trivially preserved.

Even with data present in the algorithm, the successor state of the coverage zone of a node is always $\top$ and time refinement is also not affected by it. Therefore, consecution on $Zone_{cov}$ and simulation are trivially preserved.

The preconditions for creating a covered-by edge (n, n') in the combined algorithm include $z_{abstr}(n) \sqsubseteq_{\mathcal{Z}} z_{cov}(n')$ and the Cover method ensures that the coverage condition $z_{cov}(n) \sqsubseteq_{\mathcal{Z}} z_{cov}(n')$ is satisfied on these nodes in the same way as in the lazy abstraction algorithm, as it works only on the timed projection of the ARG, the presence of data does not affect this aspect of the algorithm.

From these, it follows that the combined algorithm builds such an ARG that its timed projection is well-labeled. This means that it preserves the timed projections of all reachable states.

Next, we show that the data projection of the ARG built by the combined CEGAR algorithm over-approximates the set of reachable states. The initialization function for data over-approximates the set of initial states of the model. Any node that is not in the waitlist is either expanded or covered. When expanding a node, the combined algorithm creates a successor node in the ARG for each outgoing edge that contains an operation that is enabled from both the timed and data projection of the node and computes successor states of $d_{abstr}(n)$ with the abstract transfer function $T_{\mathcal{D}}$ for data domain D that by the definition of abstract transfer functions over-approximates the set of successor states. The preconditions for creating a covered-by edge (n, n') in the combined algorithm contain $d_{abstr}(n) \sqsubseteq_{\mathcal{D}} d_{abstr}(n')$, therefore no reachable states are lost by coverage. The data projection of an existing ARG node is never modified during the run of the combined abstraction algorithm. From these, it follows that the data projection of the ARG preserves the data projection of all reachable states.

Pruning also preserves the desired properties. With full pruning this is trivial, a completely new ARG is constructed. In the case of lazy pruning, if an initial node is removed, then it is re-added at the beginning of the next iteration, hence initiation is ensured. The abstract and coverage zones of the remaining nodes are not modified, so simulation and consecution are trivially preserved, as well as coverage for all such covered-by edges that have both their source and target in the remaining nodes. If either the source or the target is a removed node, then lazy pruning also removes the covered-by edge, so coverage is ensured in this case as well. The data projections of existing ARG nodes are not modified by pruning, and any node that became non-covered or non-expanded is put back into the waitlist at the beginning of the next iteration, hence no reachable states are lost by pruning.

All reachable states are preserved, therefore we do not lose reachable locations either, since for a location to be reachable there has to exist a concrete run of the automaton to that state.

A safe result from the combined abstraction algorithm is returned only if the ARG does not contain the target location. We have seen that the ARG built by this algorithm preserves all reachable locations, so if the ARG does not contain the target location, then it is indeed unreachable in the model.

An unsafe result is returned only if the refiner provides a concrete path to the target location for the given abstract counterexample. This means that if an unsafe result is returned, then assuming that the refiner works correctly, the target location is actually reachable in the model.

Chapter 6

Evaluation

We implemented the proposed techniques in the Theta open source model checking framework [25]. Since Theta is a configurable framework, we were able to implement our solutions as new algorithm configurations in the framework and also experiment with multiple options for some aspects of our algorithms that were already present in the framework (e.g. different abstract domains, abstraction refinement techniques, etc.).

Theta has already supported model checking for XTA models, although previously using the lazy abstraction algorithm was the only option for that. Because of this, we conducted experiments with lazy abstraction as well to use as a baseline for our results.

We evaluated our solutions on a benchmark set for RTS. These systems are usually verified by lazy abstraction, so the models in this benchmark set do not contain elements that are not supported by lazy abstraction algorithms. The benchmark set we used consists of 95 benchmark models [13].

The benchmarks were run using Benchexec[1] on virtual computers with 6 CPU cores. Each task was run with a time limit of 300 seconds and a memory limit of 15 GB.

6.1 Lazy abstraction

In the following, we describe the configuration options used in our experiments with lazy abstraction.

Abstract and coverage data domains:

- EXPL/EXPL (explicit value abstraction for both domains, using over-approximation in the coverage domain)
- EXPL/NONE (explicit value abstraction without over-approximation): this is the approach used by the Uppaal verification tool [20], one of the most efficient approaches for real-time systems
- FORM (first-order logic formulas as abstract states in both domains): this is the only option supporting non-deterministic operations, therefore this configuration can be considered the actual baseline for our results

Data refinement algorithm:

- NONE (no data refinement): used when the coverage domain does not allow over-approximation

[1] https://github.com/sosy-lab/benchexec

- BW (backward propagated interpolation)
- FW (forward propagated interpolation)
- SEQ (sequence interpolation)

Time refinement algorithm:
- BW (backward propagated interpolation)
- FW (forward propagated interpolation)
- LU (propagation of LU bounds)

Not all combinations of the above configuration options are valid: data refinement does not make sense for the EXPL/NONE abstraction, EXPL/EXPL is only used with BW and FW data refinement algorithms, FORM is only used with SEQ refinement.

Data refinement	Time refinement	EXPL		FORM
		EXPL	NONE	
NONE	BW		54 *546*	
	FW		54 *577*	
	LU		53 *385*	
BW	BW	53 *350*		
	FW	53 *351*		
	LU	52 *321*		
FW	BW	48 *211*		
	FW	48 *221*		
	LU	48 *289*		
SEQ	BW			45 *1040*
	FW			45 *1060*
	LU			45 *1110*

Table 6.1: Benchmark results for the lazy abstraction algorithm: the number of runs where the algorithm has given a result within the allowed time frame and the total CPU time in seconds needed for those computations (in italics)

The results can be seen in Table 6.1. From our point of view, the most important part of this table is the part with FORM abstraction. Even though it performed poorly compared to other configurations, it is the only configuration that has the same expressive power as our solutions, so we will use this configuration as the baseline for our results.

Evaluation of the lazy abstraction using measurements with different configurations has already been done before, further analysis of these measurements can be found in [24].

6.2 Eager CEGAR

Our goal was to evaluate the newly implemented eager CEGAR algorithms and compare them with the preexisting solutions. In particular, we look for answers to the following questions:

- How do the eager CEGAR algorithms compare to the lazy abstraction algorithms?
- How do different data abstract domains and data refinement strategies affect the result?
- How does active zone abstraction compared to Global Zone abstraction?
- Does lazy pruning improve the result?

As for the data abstract domains we chose explicit value analysis (EXPL) and predicate abstraction (PRED) because they were shown to be effective in eager CEGAR. For data refinement, we chose sequence interpolation (SEQ), and Newton refinement (NWT) [10]because these two radically different approaches to refinement are proven to be efficient in different circumstances, and the time system is a new application for both of them. We tried each configuration with both full and lazy pruning and with both active (ACT) and global zone abstraction (GLOB) for clock variables.

		EXPL		PRED	
		SEQ	NWT	SEQ	NWT
GLOB	full pruning	48 *780*	47 *802*	50 *1310*	0 -
	lazy pruning	48 *884*	48 *822*	48 *1150*	0 -
ACT	full pruning	35 *696*	35 *723*	48 *1460*	0 -
	lazy pruning	35 *819*	35 *702*	45 *1250*	0 -

Table 6.2: Benchmark results for the eager CEGAR algorithm:
the number of runs where the algorithm has given a result within the allowed time frame and the total CPU time in seconds needed for those computations (in italics)

Predicate abstraction was able to verify more problems than explicit value analysis, but verification required 30-100% more CPU time. As for the refinement, Newton refinement did not work on predicate abstraction, but in the case of explicit value analysis, it verified the same amount of problems as sequence interpolation. Interestingly the cost of full pruning, Newton refinement required 2-4% more time than sequence interpolation, but with lazy pruning, it was the other way around with sequence interpolation requiring 7-16% more time.

In the case of explicit value analysis and lazy pruning, it verified the same amount of problems but required 2-13% more CPU time than full pruning, and in the case of predicate abstraction and lazy pruning, it verified fewer problems, than full pruning.

As for the zone abstractions, active zone abstraction unfortunately is much worse than global value abstraction. In the case of explicit value analysis, global zone abstraction

37% more problems are verified. When we used predicate abstraction and global zone abstraction it verified 4-6% more problems in 8-10% less time than active zone abstraction.

We got the best result when we used Global Zone abstraction with full pruning and SEQ refinement algorithm, in the case of predicatre abstraction it verified 4% more problems, but required 68% more CPU time.

The expressive power of form abstraction in lazy abstraction, and predicate abstraction in eager CEGAR are the same over data variables, and we can see from Table6.1 and Table 6.2 that the eager CEGAR verified 10% more problems.

6.3 Combined CEGAR

We conducted experiments with the combined CEGAR algorithm as well, to answer the following questions:

1. How does the combined CEGAR algorithm compare to the lazy abstraction algorithm?
2. How do different data abstractions, data, and clock refinement techniques affect the result?
3. Does lazy pruning improve the result?

To answer these questions using experiments, we have to specify first what configurations we will use. For the abstract data domain, we use the already implemented options: explicit value abstraction (EXPL) and predicate abstraction (PRED). Previous work ([16], [11]) shows that interpolating techniques give the best performance for data abstraction refinement. For data refinement, we use sequence interpolation (SEQ) and Newton interpolation (NWT) [21],[9] for the same reasons as for the eager CEGAR algorithm. For time refinement we use algorithms that were used with the lazy abstraction algorithm as well: backward propagated interpolation (BW), forward propagated interpolation (FW), or propagation of LU bounds (LU). We also want to evaluate the effect of different pruning strategies, so we run all configurations once with full pruning and once with lazy pruning.

From Table 6.3 we can see that by using the combined CEGAR algorithm our results significantly improved compared to the baseline. We were able to check up to 6 more models, in less than half the time the baseline lazy algorithm needed to check fewer of them. The best results were produced by using explicit value abstraction with sequence interpolation for data, with forward propagated interpolation for time refinement.

Generally, explicit value abstraction performed better than predicate abstraction, by about 20-35%. It can also be concluded that we can not use predicate abstraction with Newton refinement with this algorithm for these models. However, with explicit value abstraction, Newton refinement and sequence interpolation yielded similar results. The same can be said about backward and forward propagated interpolation for time refinement. Compared to them, propagating LU bounds proved to be much less efficient, requiring about 50% more CPU time, although it could check around the same number of models.

It can also be noted that the same time refinement algorithms proved to be the best for this algorithm as for the lazy abstraction algorithm (forward and backward propagated interpolation).

Introducing lazy pruning did not improve our results, on the contrary, it yielded worse results than full pruning by 1-14%. This is not contradictory to our expectations, as lazy

		EXPL		PRED	
		SEQ	NWT	SEQ	NWT
BW	full pruning	51 *462*	51 *479*	49 *723*	0 -
	lazy pruning	49 *498*	51 *495*	47 *671*	0 -
FW	full pruning	51 *457*	51 *487*	49 *704*	0 -
	lazy pruning	49 *498*	51 *486*	47 *657*	0 -
LU	full pruning	51 *653*	51 *617*	49 *949*	0 -
	lazy pruning	49 *676*	51 *625*	47 *834*	0 -

Table 6.3: Benchmark results for the combined CEGAR algorithm:
the number of runs where the algorithm has given a result within the allowed time frame and the total CPU time in seconds needed for those computations (in italics)

pruning does some additional steps and it might also be performed multiple times if more information is needed about data variables in nodes already present in the ARG. Even though it did not improve the results, the loss in performance was not significant either, in most cases being 3-9%.

6.4 Comparison of approaches

From each group of algorithms discussed in this chapter (lazy abstraction, eager CEGAR, combined CEGAR) we took the best-performing configuration to compare all the different approaches. We also included the baseline lazy abstraction configuration (working with first-order formulas) in the comparison.

Figure 6.1 shows the number of models that can be verified within a given time limit. A point with coordinates (x, y) on a line corresponding to a configuration means that the given configuration can verify x models if the time limit for each task is y seconds.

The black line corresponding to the best performance represents lazy abstraction without data abstraction, although it is the only configuration in this figure that cannot handle non-determinism.

The baseline lazy abstraction configuration is represented by the red line. One can see that almost all of our novel techniques require less time and check more models than the lazy abstraction of the same expressive power. The best-performing novel technique turned out to be the combined CEGAR algorithm. The combined CEGAR algorithm with explicit value abstraction performed especially well, its results are comparable with the best-performing lazy abstraction algorithm.

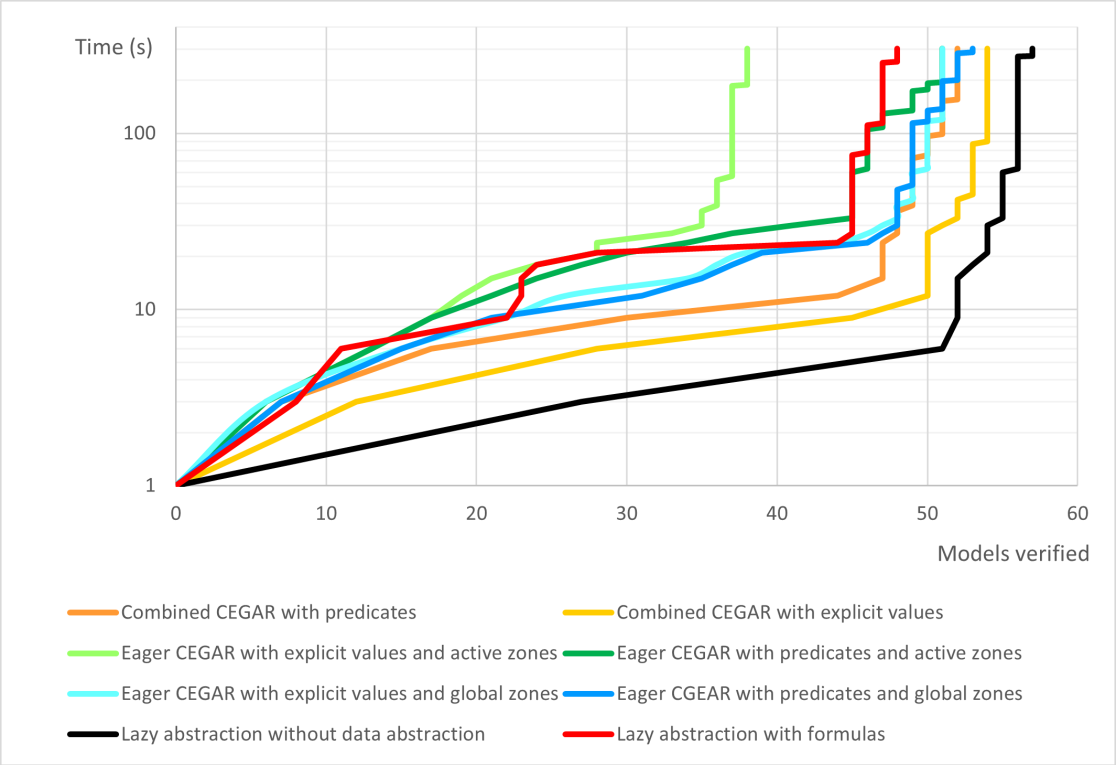

Figure 6.1: Number of models verified by time

Chapter 7

Related work

Model checking has a long standing history [8]. However, in this section, we focus on the verification of timed systems and the handling of data in CEGAR approaches.

The most prominent tool for the verification of timed systems is the UPPAAL tool [20]. UPPAAL uses zones for time representation and explicit state space traversal for the discrete parts. UPPAAL is an efficient tool for verification when only simple data is used. Other zone based techniques for time representation are also available in the literature, where both forward and backward computation for the zone representation was examined [15, 17, 22, 26, 1]. A general CEGAR-based abstraction technique using active clocks for refinement was used in [14]. Efficient lazy abstraction for timed systems was developed in [23] with efficient refinement strategies. The authors also used their former work in the field [12]. These works served as the basis of this paper.

Software verification has a much richer literature with real industrial success stories. Lazy abstraction was introduced in software verification in [18]. Later, when CEGAR was devised [7, 21, 4], it became the dominant technique for software verification for a long time. A wide range of abstraction and refinement algorithms provided an efficient software verification solution. An extensive study, which served the basis for the data handling in our framework can be found here: [16].

Other abstraction based techniques also proved their efficiency is software verification: many of them available in open source tools such as Theta[1] [25] and CPAchecker[2] [3].

[1] https://github.com/FTSRG/theta

[2] https://cpachecker.sosy-lab.org/

Chapter 8

Conclusions

Real-time safety-critical systems are becoming more and more important in industrial environments. Assuring the safe behavior of such systems is of utmost importance, so formal verification is desirable as it is able to check the whole state space of the system for errors, that would otherwise be hard to find with conventional testing. However real-time systems have both data and time-based behavior, which has proved to be a huge challenge for model-checking algorithms, as data and time abstractions require different approaches for efficient verification. No former approaches could give efficient techniques for this problem.

Based on the literature lazy abstraction is efficient for representing time, but have severe limitations with data: formalisms supporting non-deterministic assignments (such as input variables, input data) are not supported by most lazy abstraction techniques. On the other hand, the eager abstraction in the CEGAR approach has numerous efficient domains for data but does not have a flexible abstract domain with a configurable precision to provide efficient representation for time.

In this work, we aimed at providing support and algorithms for efficient abstraction-based model checking for real-time systems that have data and time-based behavior.

We provided the following algorithmic and theoretical results:

- We defined a mapping from a high-level modelling language and property description to the low-level formal representation in Theta.
- We defined a new, two step refinement strategy inside the CEGAR loop to use traditional CEGAR-based abstraction and refinement for data and lazy abstraction for time.
- We defined a novel representation of the nodes in the state space that supports the efficient computation of both the data and time coverage.
- We developed various pruning strategies that preserve the consistency of the abstract reachability graph representation between the steps of the algorithm.
- We proved the correctness of the new algorithms.

In order to provide support for the verification engineers, we extended an existing, open-source framework. Our implementation related results are the following:

- We implemented a chain of model transformations to provide the mapping of the high-level input language to the low-level formal representations.

- We have adapted the lazy abstraction and also activity based CEGAR techniques to the TCFA formalism and integrated the approaches to the Theta framework.
- We have implemented the novel combination of the algorithms and integrated into the Theta framework.

We implemented the aforementioned algorithms in the open source Theta framework and evaluated them on benchmark models. Our measurements show that the novel combined algorithm is competitive and it performed better than lazy and eager abstraction. In conclusion, we extended the Theta framework with various existing and new algorithms to handle real-time, software-based systems, so now a significant verification portfolio can be used by the verification engineers for solving their problems.

8.1 Future Work

In the future we would like to:

- extend the set of properties that the model transformation supports so we can verify a more diverse set of problems and requirements,
- investigate the efficiency of the algorithm calculating the set of active clocks and employ dynamic programming methods to increase its efficiency and make it a competitive alternative.

Bibliography

[1] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Advanced Course on Petri Nets*, pages 87–124. Springer, 2003.

[2] Johan Bengtsson and Wang Yi. *Timed Automata: Semantics, Algorithms and Tools*, pages 87–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-27755-2. DOI: 10.1007/978-3-540-27755-2_3. URL https://doi.org/10.1007/978-3-540-27755-2_3.

[3] Dirk Beyer and M Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011.

[4] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on cegar and interpolation. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, pages 146–162, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37057-1.

[5] Dirk Beyer, Thomas A. Henzinger, and Gregory Theoduloz. Program analysis with dynamic precision adjustment. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 29–38, 2008. DOI: 10.1109/ASE.2008.13.

[6] 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.

[7] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[8] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. *Introduction to Model Checking*, pages 1–26. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_1. URL https://doi.org/10.1007/978-3-319-10575-8_1.

[9] Daniel Dietsch, Matthias Heizmann, Betim Musa, Alexander Nutz, and Andreas Podelski. Craig vs. newton in software model checking. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 487–497, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. DOI: 10.1145/3106237.3106307. URL https://doi.org/10.1145/3106237.3106307.

[10] Daniel Dietsch, Matthias Heizmann, Betim Musa, Alexander Nutz, and Andreas Podelski. Craig vs. newton in software model checking. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 487–497, 2017.

[11] Mihály Dobos-Kovács, Ákos Hajdu, and András Vörös. Bitvector support in the theta formal verification framework. In *2021 10th Latin-American Symposium on Dependable Computing (LADC)*, pages 01–08, 2021. DOI: 10.1109/LADC53747.2021.9672595.

[12] Cziborová Dóra. Generalizing lazy abstraction refinement algorithms with partial orders. Master's thesis, Budapest University of Technology and Economics, 2022.

[13] Rebeka Farkas and Gábor Bergmann. Towards reliable benchmarks of timed automata. In *25th Mini-Symposium*, pages 20–23. Budapest University of Technology and Economics, 2018.

[14] Rebeka Farkas and Akos Hajdu. Activity-based abstraction refinement for timed systems. In *24th PhD Mini-Symposium (Minisy@ DMIS 2017)*, pages 18–20. Budapest University of Technology and Economics, 2017.

[15] Rebeka Farkas, Tamás Tóth, Ákos Hajdu, and András Vörös. Backward reachability analysis for timed automata with data variables. *Electronic Communications of the EASST*, 76, 2019.

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

[17] Thomas A Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and computation*, 111(2):193–244, 1994.

[18] 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.

[19] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. *Predicate Abstraction for Program Verification*, pages 447–491. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_15. URL https://doi.org/10.1007/978-3-319-10575-8_15.

[20] 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.

[21] K. L. McMillan. Applications of craig interpolants in model checking. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–12, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31980-1.

[22] Georges Morbé, Florian Pigorsch, and Christoph Scholl. Fully symbolic model checking for timed automata. In *International Conference on Computer Aided Verification*, pages 616–632. Springer, 2011.

[23] Tamás Tóth and István Majzik. Lazy reachability checking for timed automata with discrete variables. In *International Symposium on Model Checking Software*, pages 235–254. Springer, 2018.

[24] 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.

[25] 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.

[26] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Formal Description Techniques VII*, pages 243–258. Springer, 1995.