MŰEGYETEM 1782

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

Abstraction-based timed model checking for software-intensive system models

Master's Thesis

Author
Dóra Cziborová

Advisor
dr. Kristóf Marussy

December 17, 2023

Contents

Kivonat

A valós idejű szoftverintenzív rendszerek biztonságának biztosítása kulcsfontosságú számos kritikus alkalmazásban, mint az automatikus vezetéstámogató rendszerek, vasúti rendszerfigyelők, okos városok. A formális verifikációs módszerek, mint az automatizált modellellenőrzés matematikailag precíz bizonyítékot adhatnak a rendszerek helyességére a biztonságos működés érdekében.

A valós idejű időzítési követelmények és a bonyolult időbeli viselkedések megnehezítik a valós idejű szoftverintenzív rendszerek verifikációját. Ezek a rendszerek tartalmaznak továbbá külső adatokkal, például beérkező szenzor adatokkal végzett számításokat, melyek erősen adatfüggő viselkedéseket eredményeznek. Emiatt kétféle kihívással is találkozunk: (1) a szoftverek fejlesztéséhez gyakran összetett eszközök szükségesek, mint például az állapottérkép alapú modellezőeszközök a viselkedés megfelelő kifejezőerejű leírásához, valamint (2) a formális verifikációt hátrátatja az esetleges időzítések és bejövő adatok által okozott állapottér-robbanás.

Az utóbbi probléma ellensúlyozására az irodalomban absztrakció alapú modellellenőrző eszközök kerültek kifejlesztésre. Korábbi munkám során kifejlesztettem absztrakció alapú algoritmusok egy olyan kombinációját, mely képes az időzített viselkedések és összetett bejövő adatok egyidejű kezelésére. Az időzített modellellenőrzésben használt alacsonyszintű formalizmusok azonban nem rendelkeznek kellő kifejezőerővel a magasszintű modellező eszközökből származó rendszer modellek reprezentálására. Bár az időzített viselkedés diszkrét közelítése elterjedt megoldás az olyan, kifejezőbb verifikációs algoritmusok használatának lehetővé tételére, melyek natívan nem támogatják az időzített viselkedést, ezzel a diszkretizáció hibája miatt elveszhet a helyesség.

A dolgozat célja egy köztes formalizmus javaslata a valós idejű rendszer modellek reprezentálására, valamint ehhez a formalizmushoz létező absztrakció alapú modellellenőrző algoritmusok adaptálása. Ehhez (i) javasolok egy kiterjesztést a *Gamma* állapottérkép alapú modellezőeszközben köztes modellként használt *kiterjesztett szimbolikus tranzíciós rendszer* (XSTS) formalizmushoz, hogy lehetővé tegyem időzített rendszerek leírását, kihasználva a Gamma és az XSTS létező képességeit az összetett adatvezérelt viselkedés és a rendszer komponensek közti kommunikáció reprezentálására. Ezen kívül (ii) absztrakció alapú modellellnőrzési megközelítéseket adaptálok az időzített XSTS modellek verifikációjára, beleértve (iii) a komponensek kommunikációja és a hierarchikus modellezés által okozott összetett vezérlési folyamok kezelését a kombinált verifikációs algoritmusomban. Továbbá, (iv) az időzített XSTS formalizmust és a verifikációs algoritmusokat implementálom a nyílt forráskódú *Theta* verifikációs keretrendszerben. Végül (v) a javasolt megközelítéseket ipari projektetből származó esettanulmányokon és szintetikus benchmark modelleken értékelem ki.

Munkám eredményeképp lehetővé válik az összetett valós idejű szoftverintenzív rendszerek verifikációja a kifejezőerő korlátozása és diszkretizációs hiba nélkül.

Abstract

Ensuring the safe operation of real-time software-intensive systems is crucial in many critical applications, such as automatic driver assist systems, monitors for railway systems, and smart cities. Automated model checking provides a mathematically precise way to maintain safety and prevent damage to property by formally verifying correctness.

Real-time scheduling requirements and complex timed behavior make the verification of real-time software-intensive systems difficult. These systems also contain computations with external data, e.g. sensor data, and heavily data-dependent computations. These properties pose a twofold challenge: (1) development of the software often requires complex toolchains, such as statechart-based modeling to adequately express behaviors, and (2) formal verification is adversely affected by state space explosion caused by the possible scheduling and data inputs.

To counteract the latter problem, abstraction-based model checking tools have been developed in the literature. In our previous work, we developed a combination of abstraction-based algorithms to simultaneously handle timed behaviors and complex data acquisition. However, existing low-level formalisms for timed model checking lack the expressive power to represent system models from high-level modeling toolchains. While discrete approximations of timed behaviors are a common solution to enable the use of more expressive verification algorithms that do not natively support timed behaviors, this may introduce unsoundness due to discretization error.

This work aims to propose an intermediate formalism to represent real-time software-intensive system models, and adapt existing abstraction-based model checking algorithms. In particular, we (i) propose an extension to the *Extended Symbolic Transition System* (XSTS) formalism used as an intermediate model in the *Gamma* statchart-based modeling toolchain to represent timed systems, leveraging the existing capabilities of Gamma and XSTS to represent complex data-driven behaviors and communication between system components. We also (ii) adapt abstraction-based model checking approaches to handle the verification of timed XSTS, including (iii) handling complex control flow caused by component communication and hierarchical modeling in our combined verification algorithm. We (iv) implement the timed XSTS formalism and the verification algorithms in the open-source *Theta* verification framework and (v) evaluate the proposed approaches on case studies from industrial projects, as well as synthetic benchmarks.

As a result, verification of complex real-time software-intensive systems is enabled without limitations to expressive power or unsoundness introduced by discretization.

Chapter 1

Introduction

Software-based systems are becoming more and more widespread in many areas, including safety-critical systems, where software faults can result in severe consequences. Ensuring the safe operation of real-time software-intensive systems is crucial in critical applications, such as automatic driver assist systems, monitors for railway systems, and smart cities. In one hand, testing is widely used to find faults in software. However, testing can not prove functional correctness alone. On the other hand, automated model checking provides a mathematically precise way to find errors or to prove correctness of software systems.

Real-time scheduling requirements and complex timed behavior make the verification of real-time software-intensive systems difficult. These systems also contain heavily data-dependent computations. The development of the software often requires complex toolchains and high-level modeling formalism, such as statechart-based modeling to adequately express behaviors. Furthermore, formal verification is adversely affected by state space explosion caused by the possible scheduling and data-intensive operations.

Various techniques and algorithms were introduced in the literature to tackle the state space explosion problem. Abstraction-based model checking proved its efficiency to reduce the complexity of verifying software systems. Advanced refinement techniques help finding the right abstraction to prove the correctness or find the software faults. These techniques are efficient for software, but they are less suitable for handling timing information.

The verification of timed systems often relies on a modeling formalism that restricts both the data dependent and also the timing operations in order to be able to use efficient algorithms for the timed verification. As a consequence, existing low-level formalisms for timed model checking lack the expressive power to represent system models from high-level modeling toolchains.

In this work we propose an expressive intermediate formalism to represent real-time software-intensive system models, and introduce a novel extension to existing model checking methods for the efficient verification. In particular, we (i) propose an extension to the *Extended Symbolic Transition System* (XSTS) formalism used as an intermediate model in the *Gamma* statchart-based modeling toolchain to represent timed systems, leveraging the existing capabilities of Gamma and XSTS to represent complex data-driven behaviors and communication between system components. We also (ii) introduce abstraction-based model checking approaches for the efficient verification of timed XSTS, including (iii) handling complex control flow caused by component communication and hierarchical modeling in our combined verification algorithm. We (iv) implement the timed XSTS formalism and the verification algorithms in the open-source *Theta* verification framework and (v) evaluate the proposed approaches on case studies from industrial projects, as well as synthetic benchmarks.

Our contributions enable engineers to rely on an expressive intermediate formalism to formally represent engineering models, and our toolchain provides sound, abstraction-based techniques to efficiently verify the engineering models.

We presented our solutions in [17] at this year's Students' Scientific Conference at BME. In this work we cover additional related work, as well as provide further details of the implementation of the proposed solutions.

Chapter 2

Preliminaries

2.1 Model checking

Formal verification is the term used for proving the correctness of a system using mathematical techniques. Model checking [15] is a formal verification method that exhaustively explores all possible behaviors (the *state space*) of a formal model of the system to check it against some formally specified requirements. The desired properties are typically safety properties (the defined unsafe state of the system is never reached) and reachability properties (the desired state can be reached).

Both safety and reachability properties define a reachability problem. The result of the reachability analysis is either a path to the given state, or a mathematical proof that the state is not reachable. In case of safety properties, a path to the unsafe state is a counterexample to the safety property, while the desired result is a proof of the unsafe state being unreachable. For reachability properties this is reversed, the desired result is a path to the given state, while a proof of unreachability is a counterexample to the property.

Explicitly enumerating all states on all paths would not be an efficient or even viable approach to reachability analysis. The size of the state space of the system grows exponentially in the number of variables, resulting in what is called *state space explosion*, which is one of the main challenges of model checking.

To counteract the problem of state space explosion, many reachability analysis techniques have been developed in the literature, including *bounded model checking* [13], *symbolic model checking* [8], *abstract interpretation* [16], and *abstraction* [14], the latter forming the basis of the algorithms presented in this work.

2.2 Modeling formalisms

Model checkers operate on formal models with well-defined syntax and semantics, e.g. *Kripke structures*, *timed automata* [6] and *symbolic transition systems* [26]. In the following we describe two modeling formalisms that are relevant to our work, *timed automata with data variables* and *extended symbolic transition systems*.

2.2.1 Timed automata with data variables

This work focuses on the verification of systems containing both timed behavior and data operations. Timed automata are suitable for modeling timed behavior. Although not without certain limitations, data variables can also be included in timed automata.

Timed behavior is modeled using *clock variables*. Clock variables are continuous, non-negative variables. All clock variables are initialized at zero, can be reset, and are incremented equally within the model. For a set of clock variables V_C, a *clock valuation* $val_C \colon V_C \to \mathbb{R}_{\geq 0}$ maps each clock $c \in V_C$ to a non-negative value. We denote the set of clock valuations by Val_C.

Let V_D be a set of *data variables* with (finite or infinite) domains $D_1, D_2, \ldots, D_{|V_D|}$. A *data valuation* $val_D \colon V_D \to \bigcup_{i=1}^{|V_D|} D_i$ maps each variable $x \in V_D$ to a value in its corresponding domain, i.e. $\forall x_i \in V_D \colon val_D(x_i) \in D_i$. We denote the set of data valuations by Val_D.

Definition 1 (Timed automaton with data variables). A timed automaton with data variables is a tuple $TA = \langle \mathcal{L}, \mathcal{E}, V_C, V_D, l^0, val_D^0 \rangle$, where

- $\mathcal{L}$ is a finite set of *locations*;
- $\mathcal{E} \subset \mathcal{L} \times \mathcal{O}_{TA} \times \mathcal{L}$ is a set of *edges* labeled with operations $op \in \mathcal{O}_{TA}$;
- V_C and V_D are finite sets of *clock* and *data variables*;
- $l^0 \in \mathcal{L}$ is the *initial location*;
- $val_D^0 \in Val_D$ is the *initial valuation* of data variables. ▪

A *state* of the automaton is a tuple $\langle l, \langle val_D, val_C \rangle \rangle$ where $l \in \mathcal{L}$, val_D is a data valuation over V_D, and val_C is a clock valuation over V_C.

Let $[\![op]\!]_{TA}(\langle val_D, val_C \rangle)$ denote the set of possible results of applying *op* to the valuation $\langle val_D, val_C \rangle$. Thus, $\{l'\} \times [\![op]\!]_{TA}(\langle val_D, val_C \rangle)$ is the set of states reachable by edge $\langle l, op, l' \rangle$ from $\langle l, \langle val_D, val_C \rangle \rangle$.

We use the same notation for denoting the result set of applying *op* on any valuation, i.e. $[\![op]\!]_{TA}(val_D)$, $val_D \in Val_D$ denotes applying *op* on a data valuation val_D.

In timed automaton models, operations must not contain interdependence of data and clocks, i.e. $[\![op]\!]_{TA}(\langle val_D, val_C \rangle) = [\![op]\!]_{TA}(val_D) \times [\![op]\!]_{TA}(val_C)$. Therefore, operations can be divided into two categories, *data operations* that leave clock valuations unchanged (for a data operation op_D and clock valuation val_C, $[\![op_D]\!]_{TA}(val_C) = \{val_C\}$), and *clock operations* that leave the data valuations unchanged (for a clock operation op_C and data valuation val_D, $[\![op_C]\!]_{TA}(val_D) = \{val_D\}$).

(a) Data operations:

- A *data guard* has the form $[\varphi]$, where φ is a predicate over V_D. They control which edges are enabled, $[\![[\varphi]]\!]_{TA}(val_D) = \{val_D\}$ if val_D satisfies φ, otherwise $[\![[\varphi]]\!]_{TA}(val_D) = \emptyset$.
- An *assignment* $x := \varphi$ assigns the value of an expression φ evaluated on val_D to the variable $x \in V_D$ with domain D, i.e. $[\![x := \varphi]\!]_{TA}(val_D) = \{val'_D\}$, such that $val'_D(x) = \varphi(val_D)$, $\varphi(val_D) \in D$, and $\forall x' \in V_D \backslash \{x\} \colon val'_D(x') = val_D(x')$.

- A *havoc* operation $havoc(x)$ is a non-deterministic assignment that sets a data variable $x \in V_D$ to any value of its domain D, i.e. $[\![havoc(x)]\!]_{TA}(val_D) = \{val'_D \mid val'_D(x) \in D,\ \forall x' \in V_D \backslash \{x\} \colon val'_D(x') = val_D(x')\}$.

(b) Clock operations:

- $[constr]$ is a *clock guard* if *constr* is a *clock constraint*, i.e. a formula of the form $c_i \sim k$ or $c_i - c_j \sim k$, where $c_i, c_j \in V_C$, $\sim \in \{<, \leq, =, \geq, >\}$, and $k \in \mathbb{Z}$. Similarly to data guards, $[\![[constr]]\!]_{TA}(val_C) = \{val_C\}$ if val_C satisfies *constr*, otherwise $[\![[constr]]\!]_{TA}(val_C) = \emptyset$.

- A *reset* operation $c := n$ sets a clock $c \in V_C$ to a value $n \in \mathbb{N}_0$, $[\![c := n]\!]_{TA}(val_C) = \{val'_C\}$ such that $val'_C(c) = n$, and $\forall c' \in V_C \backslash \{c\} \colon val'_C(c') = val_C(c')$.

- A *delay* operation increments all clocks, $[\![delay]\!]_{TA}(val_C) = \{val^\delta_C \mid \delta \in \mathbb{R}_{\geq 0}\}$ where $val^\delta_C(c) = val_C(c) + \delta$ for all clocks $c \in V_C$.

Lastly, *compound operations* of the form op_1, op_2, where $op_1, op_2 \in \mathcal{O}_{TA}$ are executed sequentially: $[\![op_1, op_2]\!]_{TA}(val) = \{val'' \mid val'' \in [\![op_2]\!]_{TA}(val'), val' \in [\![op_1]\!]_{TA}(val)\}$.

The *initial states* of the automaton form a set $\mathcal{I} = \{\langle l^0, \langle val^0_D, val^0_C \rangle\rangle \mid val^0_C = \{c \mapsto \delta \mid c \in V_C\}, \delta \in \mathbb{R}_{\geq 0}\}$ with data variables initialized to val^0_D and clock variables to some initial delay $\delta \geq 0$.

A *run* of the automaton is an alternating finite sequence of states and operations $\sigma_{TA} = \langle l^0, \langle val^0_D, val^0_C \rangle\rangle \xrightarrow{op_1} \langle l^1, \langle val^1_D, val^1_C \rangle\rangle \xrightarrow{op_2} \cdots \xrightarrow{op_k} \langle l^k, \langle val^k_D, val^k_C \rangle\rangle$, with $\langle val^i_D, val^i_C \rangle \in [\![op_i]\!]_{TA}(\langle val^{i-1}_D, val^{i-1}_C \rangle)$ and $\langle l^{i-1}, op_i, l^i \rangle \in \mathcal{E}$ for all $1 \leq i \leq k$. A location l is *reachable* in the model if and only if there exists a run σ_{TA} such that $l^k = l$.

2.2.2 Extended symbolic transition systems

The modeling improvements proposed in this work are based on a higher-level modeling formalism, *extended symbolic transition systems* [36].

Definition 2 (Extended symbolic transition system). An extended symbolic transition system is a tuple $XSTS = \langle V, V_{ctrl}, val^0, init, env, tran \rangle$, where

- $V = \{x_1, x_2, \ldots, x_{|V|}\}$ is a set of variables with domains $D_1, D_2, \ldots, D_{|V|}$;

- $V_{ctrl} \subseteq V$ is a set of *control variables*, it is strongly recommended to track these variables explicitly (see explicit abstraction in 2.3.1 and product abstraction in 2.3.4);

- val^0 is the *initial valuation* that maps each variable $x \in V$ to the initial value of the variable in its corresponding domain, or $\top$ if unknown, i.e. $val^0(x_i) \in D_i \cup \{\top\}$;

- $init \subseteq \mathcal{O}_{XSTS}$ is a set of operations representing the *initialization transition set*, it describes more complex initialization that cannot be described by val^0;

- $env \subseteq \mathcal{O}_{XSTS}$ is a set of operations representing the *environment transition set*, it describes the interactions of the system with the environment;

- $tran \subseteq \mathcal{O}_{XSTS}$ is a set of operations representing the *internal transition set*, it describes the internal behavior of the system. ▪

Let $\mathcal{T}$ denote the set of transition sets $\{init, env, tran\}$. Each transition set in $\mathcal{T}$ consists of one or more operations taken from a set of operations $\mathcal{O}_{XSTS}$. When executing a transition set, the operation to be executed is selected from the transition set in a non-deterministic manner.

A *state* of an XSTS model is a pair $\langle val, \tau \rangle$, where $val : V \to \bigcup_{i=1}^{|V|} D_i$ is a valuation that maps each variable $x \in V$ to a value in the domain of the variable $(\forall x_i \in V : val(x_i) \in D_i)$, and $\tau \in \mathcal{T}$ denotes the transition set to be executed in this state, which is the only one that can be executed in this state. In the *initial state* $\tau = init$, and the valuation is $val = val^0$, so there is exactly one initial state, $\langle val^0, init \rangle$.

Let $[\![op]\!]_{XSTS}(\langle val, \tau \rangle)$ denote the result of applying the operation $op \in \mathcal{O}_{XSTS}$ on the XSTS state $\langle val, \tau \rangle$. We will write $[\![op]\!]_{XSTS}(val)$ for the result of applying op on val, and $[\![op]\!]_{XSTS}(\tau)$ for applying op on τ. The two components of the XSTS states are independent, hence $[\![op]\!]_{XSTS}(\langle val, \tau \rangle) = [\![op]\!]_{XSTS}(val) \times [\![op]\!]_{XSTS}(\tau)$.

Since τ is the only transition set that can be executed in a state $\langle val, \tau \rangle$, $[\![op]\!]_{XSTS}(\tau) = \emptyset$ if $op \notin \tau$. The transition set *init* is executed only once, from the initial state. The transition sets *env* and *tran* are executed in an alternating manner, but only after *init*. In accordance with this consecution of transition sets, $[\![op]\!]_{XSTS}(init) = \{env\}$, $[\![op]\!]_{XSTS}(env) = \{tran\}$ and $[\![op]\!]_{XSTS}(tran) = \{env\}$.

The set of valuations resulting from $[\![op]\!]_{XSTS}(val)$ depends on the kind of operation op. The operations in $\mathcal{O}_{XSTS}$ are either basic operations or compound operations describing more complex semantics, with other (basic or compound) operations embedded. The following operations are defined for the XSTS formalism:

(a) Basic operations:

- *Assumptions* have the form $[\varphi]$, where φ is a predicate over V, $[\![[\varphi]]\!]_{XSTS}(val) = \{val\}$ if val satisfies φ, otherwise $[\![[\varphi]]\!]_{XSTS}(val) = \emptyset$.
- *Assignments* have the form $x := \varphi$. They assign the value of an expression φ evaluated on val to the variable $x \in V$ with domain D, i.e. $[\![x := \varphi]\!]_{XSTS}(val) = \{val'\}$ such that $val'(x) = \varphi(val)$, $\varphi(val) \in D$, and $\forall x' \in V \backslash \{x\} \colon val'(x') = val(x')$.
- *Havoc* operations are non-deterministic assignments denoted by $havoc(x)$ where $x \in V$ with domain D. They assign a non-deterministically chosen value to the variable from its domain, i.e. $[\![havoc(x)]\!]_{XSTS}(val) = \{val' \mid val'(x) \in D,\ \forall x' \in V \backslash \{x\} \colon val'(x') = val(x')\}$.
- *No-op* operations are denoted simply by *skip*, $[\![skip]\!]_{XSTS}(val) = \{val\}$. The main purpose of this operation is to enable writing composite operations in a generic form.

(b) Compound operations:

- *Sequences* have the form $op_1, op_2, \ldots, op_n$, where $op_i \in \mathcal{O}_{XSTS}$. The operations $op_1, op_2, \ldots, op_n$ are executed one after the other, $[\![op_1\ op_2\ \ldots\ op_n]\!]_{XSTS}(val) = \{val^{(n)} \mid val^{(n)} \in [\![op_n]\!]_{XSTS}(val^{(n-1)}),\ \ldots,\ val'' \in [\![op_2]\!]_{XSTS}(val'),\ val' \in [\![op_1]\!]_{XSTS}(val)\}$.
- *Non-deterministic choices* have the form $\{op_1\}\ or\ \{op_2\}\ or\ \ldots\ or\ \{op_n\}$, where $op_i \in \mathcal{O}_{XSTS}$. Exactly one operation $op \in \{op_1, op_2, \ldots, op_n\}$ is executed, chosen in a non-deterministic manner, $[\![\{op_1\}\ or\ \{op_2\}\ or\ \ldots\ or\ \{op_n\}]\!]_{XSTS}(val) = [\![op]\!]_{XSTS}(val)$ such that $op \in \{op_1, op_2, \ldots, op_n\}$.

- *Conditional operations* have the form $if(\varphi)\, then\, \{op_1\}\, else\, \{op_2\}$, where φ is a predicate over V, and $op_1, op_2 \in \mathcal{O}_{XSTS}$. Exactly one of op_1 and op_2 is executed, determined by the evaluation of the predicate φ on val. If val satisfies φ, then $[\![if(\varphi)\, then\, \{op_1\}\, else\, \{op_2\}]\!]_{XSTS}(val) = [\![op_1]\!]_{XSTS}(val)$, otherwise $[\![if(\varphi)\, then\, \{op_1\}\, else\, \{op_2\}]\!]_{XSTS}(val) = [\![op_2]\!]_{XSTS}(val)$.
- *Loops* have the form $for\ i\ from\ \varphi_a\ to\ \varphi_b\ do\ \{op\}$, where i is an integer variable, φ_a and φ_b are expressions such that $\varphi_a(val) \in \mathbb{Z}$, $\varphi_b(val) \in \mathbb{Z}$, and $op \in \mathcal{O}_{XSTS}$. The semantics of this operation is that of usual for-loops, $[\![for\ i\ from\ \varphi_a\ to\ \varphi_b\ do\ \{op\}]\!]_{XSTS}(val) = [\![if(\varphi_a(val) < \varphi_b(val))\ then\ \{i := \varphi_a(val),\ op,\ for\ i\ from\ \varphi_a + 1\ to\ \varphi_b\ do\ \{op\}\}\ else\ \{skip\}]\!]_{XSTS}(val)$.

A *run* of an XSTS is an alternating finite sequence of states and operations $\sigma_{XSTS} = \langle val^0, init\rangle \xrightarrow{op_1} \cdots \xrightarrow{op_k} \langle val^k, \tau^k\rangle$, with $\langle val^i, \tau^i\rangle \in [\![op_i]\!]_{XSTS}(\langle val^{i-1}, \tau^{i-1}\rangle)$ for all $1 \leq i \leq k$. A state $\langle val, \tau\rangle$ is *reachable* in the model if there exists a run σ_{XSTS} such that $val^k = val$ and $\tau^k = \tau$.

FOL operation semantics

The operations of XSTS models can also be interpreted as FOL formulas containing variables from V and their primed versions, such that primed variables represent the variables in the next state. We use the notation $[\![op]\!]_{FOL}$ to represent the FOL semantics of an operation op. The FOL semantics of XSTS operations are as follows:

- An assumption does not change any variables, but the state should satisfy the formula: $[\![[\varphi]]\!]_{FOL} = \varphi \wedge \bigwedge_{x\in V} x = x'$;
- After an assignment, the value of the assigned variable should be equal to the assigned value: $[\![x := \varphi]\!]_{FOL} = (x' = \varphi) \wedge \bigwedge_{y \in V\setminus\{x\}} y' = y$;
- A havoc operation binds all variables in the successor except the argument of the havoc operation to their current values: $[\![havoc(x)]\!]_{FOL} = \bigwedge_{y\in V\setminus\{x\}} y' = y$;
- A no-op binds all variables in the successor to their current values: $[\![skip]\!]_{FOL} = \bigwedge_{x\in V} x' = x$;
- For the FOL representation of sequences we introduce the notation $[\![op]\!]^i_{FOL}$ that represents $[\![op]\!]_{FOL}$ with primed variables replaced by their indexed versions with index i, and non-primed variables replaced by their indexed versions with index $i-1$. Then $[\![op_1, op_2, \ldots, op_n]\!]_{FOL} = (\bigwedge_{i=1}^n [\![op_i]\!]^i_{FOL}) \wedge (\bigwedge_{x\in V} x = x^{(0)} \wedge x' = x^{(n)})$.
- Conditional operations can be represented by a FOL formula as a disjunction where only one of the two operands can hold, selected by the condition of the operation: $[\![if(\varphi)\, then\, \{op_1\}\, else\, \{op_2\}]\!]_{FOL} = (\varphi \wedge [\![op_1]\!]_{FOL}) \vee (\neg\varphi \wedge [\![op_2]\!]_{FOL})$
- The FOL representation of a non-deterministic choice consists of a disjunction of the FOL representation of its branches, with a new variable $a \in \mathbb{N}$ introduced to ensure that no more than one operand of the disjunction can evaluate to true: $[\![\{op_1\}\ or\ \{op_2\}\ or\ \ldots\ or\ \{op_n\}]\!]_{FOL} = \bigvee_{i=1}^n (a = i \wedge [\![op_i]\!]_{FOL})$

Loops do not have a closed FOL representation. With some constraints and additional steps, however, they can be reduced to a sequence of other operations that can already be represented by a FOL formula (see loop unfolding in section 5.2.1).

Let $(val, val') \models \varphi$ denote that assigning values from val to the non-primed variables and assigning values from val' to the primed variables in φ evaluates to *true*. A valuation val' is a successor of the valuation val by an operation op if and only if $(val, val') \models [\![op]\!]_{FOL}$.

In the following we will write $\mathcal{O}$ to denote the set of operations regardless of the used modeling formalism, and $[\![op]\!](val)$ to denote the result of applying $op \in \mathcal{O}$ on a concrete state val.

2.3 Abstract domains

The state spaces of models involved in verification tasks are usually large, often infinite. *Abstractions* replace concrete states with a finite, tractable representation to enable verification [16].

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

- $\mathcal{V}$ is the set of *concrete states*;
- $\mathcal{S}$ is the set of *abstract states*;
- $\sqsubseteq \subseteq \mathcal{S} \times \mathcal{S}$ is a *preorder*, i.e. it is a reflexive and transitive binary relation;
- $\gamma \colon \mathcal{S} \to 2^{\mathcal{V}}$ is the *concretization function* that maps an abstract label to the corresponding set of concrete states, such that $s_1 \sqsubseteq s_2$ implies $\gamma(s_1) \subseteq \gamma(s_2)$ for all $s_1, s_2 \in \mathcal{S}$,
- $T \colon \mathcal{S} \times \mathcal{O} \to 2^{\mathcal{S}}$ is the (abstract) *transition function* that maps an abstract state $s \in \mathcal{S}$ to its successor states $s' \in \mathcal{S}$ with respect to an operation $op \in \mathcal{O}$, such that $\bigcup_{val \in \gamma(s)} [\![op]\!](val) \subseteq \bigcup_{s' \in T(s,op)} \gamma(s')$ for all $s \in \mathcal{S}$. ▪

The transition function T is often associated with a *precision* π describing the information retained by the abstraction [12]. In this case, we will write $T^{\pi}(s, op)$ for the successors at precision π.

In most abstract domains, abstract states can be represented as FOL formulas. We write $[\![s]\!]_{FOL}$ to denote the FOL representation of an abstract state $s \in \mathcal{S}$. The transition functions often utilize an SMT solver to compute successor states, using FOL representations of states and operations.

2.3.1 Explicit value abstraction

In *explicit value abstraction* [4] the values of some subset of data variables are explicitly tracked, while the rest of the variables may take any value. The precision is defined by the variables tracked, i.e. π is a set of variables, $\pi \subseteq V$ where V is the set of variables in the model.

The abstract states are *partial valuations* that map variables $x \in \pi$ to a value in their corresponding domains. E.g., an abstract state representing the set of concrete states $\{\{x_1 \mapsto 1, x_2 \mapsto 0, x_3 \mapsto n\} \mid n \in \mathbb{Z}\}$ with precision $\pi = \{x_1\}$ is a partial valuation $pval = \{x_1 \mapsto 1\}$. The FOL representation of an abstract state $pval$ is the formula $[\![pval]\!]_{FOL} = \bigwedge_{x \in \pi} x = pval(x)$.

For two abstract states the preorder relation $pval_1 \sqsubseteq_e pval_2$ holds if and only if $pval_2(x) \in \{pval_1(x), \top\}$ for all $x \in \pi$. E.g., $\{x_1 \mapsto 1, x_2 \mapsto 0\} \sqsubseteq_e \{x_1 \mapsto 1\}$ and $\{x_1 \mapsto 1, x_2 \mapsto 0\} \sqsubseteq_e \{x_1 \mapsto 1, x_2 \mapsto \top\}$, but $\{x_1 \mapsto 1\} \not\sqsubseteq_e \{x_1 \mapsto 1, x_2 \mapsto 0\}$.

The concretization function $\gamma_e(pval)$ yields all valuations val such that $val(x) = pval(x)$ for all variables $x \in \pi$. E.g., for a set of variables $V = \{x_1, x_2\}$ with domains $D_1 = D_2 = \mathbb{Z}$ and an abstract state $pval = \{x_1 \mapsto 1\}$, $\gamma_e(pval) = \{val \mid val(x_1) = 1, val(x_2) \in \mathbb{Z}\}$.

The abstract transition function T_e assigns values to variables in π according to the operation op executed on $pval$ where it can be evaluated and assigns $\top$ to all other variables in π. E.g., for a precision $\pi = \{x_1\}$, an abstract state $pval$ and a compound operation $op = (havoc(x_1), [-180 \leq x_1 \leq 180])$, $T_e^\pi(pval, op) = \{\{x_1 \mapsto \top\}\}$.

We can also define an alternate transition function $T_{e,n}$ ($n \in \mathbb{N}$) that tries to enumerate the possible values for variables in π that cannot be evaluated in an exact manner, and assigns $\top$ to the variable only if there are more than n possible values. Otherwise, multiple successor states are included in the result, with all possible values for the variable. For the same precision, abstract state and compound operation as in the previous example, $T_{e,400}^\pi(pval, op) = \{\{x_1 \mapsto -180\}, \{x_1 \mapsto -179\}, \ldots, \{x_1 \mapsto 180\}\}$.

2.3.2 Predicate abstraction

In *predicate abstraction* [30], the abstract states are defined by predicates instead of explicit value assignments to variables. The precision π is a set of first order logic predicates, e.g. $\pi = \{p_1, p_2\}$ where $p_1 = (x_1 \geq 0)$ and $p_2 = (x_1 = x_2)$.

There are multiple approaches to representing states using predicates, here we describe *Cartesian predicate abstraction* [1] and *Boolean predicate abstraction* [1] with *predicate splitting* [25]. We also include examples, where S denotes the set of concrete states $\{\{x_1 \mapsto 1, x_2 \mapsto 0, x_3 \mapsto n\} \mid n \in \mathbb{Z}\}$, the used precision is the above example precision $\pi = \{p_1, p_2\}$, and op denotes a compound operation $havoc(x_1), [-180 \leq x_1 \leq 180]$.

Cartesian predicate abstraction represents states as a *conjunction* of positive or negative forms of *some* of the tracked predicates, the representation of S is $s_{Cart} = (p_1 \land \neg p_2)$. The transition function $T_{p,Cart}^\pi$ yields only one state, which is the strongest conjunction of predicates that is entailed by the state and operation. In our example both p_1 and $\neg p_1$ may hold in the successor, as well as both p_2 and $\neg p_2$, therefore $T_{p,Cart}^\pi(s_{Cart}, op) = \{\emptyset\}$.

Boolean predicate abstraction represents states as a *Boolean combination* of the positive or negative form of *each* tracked predicate. The representation of S is again $s_{Bool} = (p_1 \land \neg p_2)$. With predicate splitting, the transition function $T_{p,Bool}^\pi$ enumerates multiple successors when both the positive and negative form of a predicate may hold in a successor, hence $T_{p,Bool}^\pi(s_{Bool}, op) = \{(p_1 \land p_2), (p_1 \land \neg p_2), (\neg p_1 \land p_2), (\neg p_1 \land \neg p_2)\}$.

In both cases the abstract states are already FOL formulas: $[\![s_p]\!]_{FOL} = s_p$.

The preorder relation $\sqsubseteq_p$ of predicate abstraction corresponds to implication, i.e. $s_1 \sqsubseteq_p s_2$ if and only if $s_1 \Rightarrow s_2$. The concretization function $\gamma_p(s_p)$ yields all valuations that satisfy the predicates in s_p, e.g. $\gamma_p(\{p_1 \land \neg p_2\}) = \{val \mid val(x_1) \geq 0 \land val(x_1) \neq val(x_2)\}$.

2.3.3 Zone abstraction

Zone abstraction [2] is used for time abstraction. Zone abstraction represents sets of clock valuations as *zones*. A zone is described by a conjunction of clock constraints, e.g.

$z = (c_1 \geq 0) \wedge (c_2 \geq 0) \wedge (c_1 - c_2 \leq 2)$. Let $\mathcal{Z}$ denote the set of zones. Zones can also be represented as FOL formulas, in fact, a zone is already a FOL formula: $[\![z]\!]_{FOL} = z$.

The preorder relation $\sqsubseteq_{\mathcal{Z}}$ corresponds to the implication of zones, e.g. $(c \geq 1) \wedge (c \leq 4) \sqsubseteq_{\mathcal{Z}} (c \geq 0)$. The concretization function $\gamma_{\mathcal{Z}}$ yields all clock valuations satisfying the zone.

The transition function $T_{\mathcal{Z}}$ of zone abstraction produces at most one successor state, such that it does not introduce unreachable states, i.e. all concretizations of the successor states are reachable from some concretization of the source by the given operation: if $T_{\mathcal{Z}}(z, op) = \{z'\}$, then $\gamma_{\mathcal{Z}}(z') = \bigcup_{val_C \in \gamma_{\mathcal{Z}}(z)} [\![op]\!](val_C)$.

2.3.4 Product abstraction

Product abstraction combines multiple abstractions into one abstract domain. It is essential when working with systems that contain both data and clock variables, e.g. timed automata with data variables, but it also enables using more elaborate abstractions, e.g. we might use explicit value abstraction for the control variables of an XSTS model, and predicate abstraction for all other variables. Here we describe product abstraction for two domains, but this abstraction can be easily generalized for more abstract domains as well.

The product of two abstract domains $\mathcal{D}_1 = \langle \mathcal{V}_1, \mathcal{S}_1, \sqsubseteq_1, \gamma_1, T_1 \rangle$ and $\mathcal{D}_2 = \langle \mathcal{V}_2, \mathcal{S}_2, \sqsubseteq_2, \gamma_2, T_2 \rangle$ is the abstract domain $Prod(\mathcal{D}_1, \mathcal{D}_2) = \langle \mathcal{V}_1 \times \mathcal{V}_2, \mathcal{S}_1 \times \mathcal{S}_2, \sqsubseteq_\times, \gamma_\times, T_\times \rangle$, such that

- $\langle s_{1,1}, s_{1,2} \rangle \sqsubseteq_\times \langle s_{2,1}, s_{2,2} \rangle$ if and only if $s_{1,1} \sqsubseteq_1 s_{2,1}$ and $s_{1,2} \sqsubseteq_2 s_{2,2}$;
- $\gamma_\times(\langle s_1, s_1 \rangle) = \gamma_1(s_1) \times \gamma_2(s_2)$;
- $T_\times(\langle s_1, s_2 \rangle, op) = T_1(s_1, op) \times T_2(s_2, op)$.

The FOL representation of an abstract state in product abstraction exists if the FOL representation of both components exists: $[\![\langle s_1, s_2 \rangle]\!]_{FOL} = [\![s_1]\!]_{FOL} \wedge [\![s_2]\!]_{FOL}$.

2.4 Abstract reachability graphs

The reachability algorithms discussed in this work represent the state space of the model as an *abstract reachability graph* (ARG).

Definition 4 (Abstract reachability graph). An abstract reachability graph using an abstract domain $\mathcal{D} = \langle \mathcal{V}, \mathcal{S}, \sqsubseteq, \gamma, T \rangle$ for a model with operations $\mathcal{O}$ is a tuple $ARG = \langle N, E, C, L_n, L_e \rangle$, where

- $\langle N, E \rangle$ is a finite directed tree with nodes N and edegs E, rooted at $n_0 \in N$;
- $C \subseteq N \times N$ is the set of *covered-by edges*;
- $L_n \colon N \to \mathcal{S}$ is the node labeling by abstract states of $\mathcal{D}$;
- $L_e \colon E \to \mathcal{O}$ is the edge labeling by operations of the model. ▪

A node $n \in N$ is *expanded* if and only if for all operations $op \in \mathcal{O}$ enabled from $L_n(n)$ there is an outgoing edge from n labeled with op. A node n is *covered* if and only if $\langle n, n' \rangle \in C$ for some node n'. A node is *pending* if and only if it is not expanded and not covered. An ARG is *complete*, if and only if all its nodes are either expanded or covered.

We require the following properties of ARGs:

1. Initiation: if *val* is an initial state of the model, then $val \in \gamma(L_n(n_0))$.
2. Inductive labeling: for each expanded node $n \in N$ and operation $op \in \mathcal{O}$ enabled from $L_n(n)$ there exists a node $n' \in N$ and an ARG edge $\langle n, n' \rangle \in E$ for each $s' \in T(L_n(n), op)$ such that $s' \sqsubseteq L_n(n')$.
3. Coverage: $L_n(n) \sqsubseteq L_n(n')$ for each $\langle n, n' \rangle \in C$.

When performing reachability analysis, we seek an *abstract run* $n_0 \xrightarrow{op_1} n_1 \xrightarrow{op_2} \cdots \xrightarrow{op_k} n_k$ such that $n_i \in N$ for all $0 \leq i \leq k$, $\langle n_{i-1}, n_i \rangle \in E$ and $L_e(\langle n_{i-1}, n_i \rangle) = op_i$ for all $1 \leq i \leq k$ that can be concretized into a run with $\gamma(L_n(n_k))$ containing the *target state* of the reachability analysis. If such abstract run exists, then the target state is reachable, otherwise, if the ARG becomes complete without containing such abstract run, then the target state is unreachable.

2.5 CEGAR

Counterexample-guided abstraction refinement (CEGAR) [4, 3, 31, 32, 45] is an abstraction-based model checking technique. The idea of the CEGAR approach is to start with a very coarse initial abstraction (e.g. a precision of no tracked variables in explicit value abstraction, or no tracked predicates in predicate abstraction), then refine it iteratively in alternating *abstraction* and *refinement* steps.

Algorithm 1 shows a CEGAR algorithm where a state is a target state if and only if it satisfies the target predicate φ_t. The CEGAR algorithm utilizes an *abstractor* providing the Build algorithm that returns the abstractor result R_a, and a *refiner* providing the Refine algorithm that returns the refiner result R_r.

Algorithm 1 CEGAR algorithm

```
1: function Check($\mathcal{M}$: XSTS model, $\mathcal{D} = \langle \mathcal{V}, \mathcal{S}, \sqsubseteq, \gamma, T \rangle$: abstract domain, $\varphi_t$: target predicate, $\pi_0$: initial precision)
2:     $\pi \leftarrow \pi_0$
3:     $arg \leftarrow \langle \emptyset, \emptyset, \emptyset, L_n, L_e \rangle$
4:     loop
5:         $R_a$, $arg \leftarrow$ Build($\mathcal{M}$, $\mathcal{D}$, $\varphi_t$, $arg$, $\pi$)
6:         if $R_a$ = unreachable then
7:             return unreachable, $arg$
8:         else
9:             $R_r$, $arg$, $\pi \leftarrow$ Refine($arg$, $\pi$)
10:            if $R_r$ = reachable then
11:                return reachable, $arg$
```

CEGAR is a suitable algorithm for the verification of XSTS models. The nodes of the ARG are labeled by abstract states: in the case of XSTS models we use a product domain as the abstract domain, where the first domain is an arbitrary abstract domain suitable for the abstraction of valuations over data variables. The second abstract domain explicitly tracks the transition sets to be executed next, for this purpose we define an XSTS-specific abstract domain $\mathcal{D}_\tau = \langle \mathcal{T}, \mathcal{T}, =, id_{\mathcal{T}}, T_\tau \rangle$, where $id_{\mathcal{T}}$ is the identity function of $\mathcal{T}$, $T_\tau(init) = \{env\}$, $T_\tau(env) = \{tran\}$ and $T_\tau(tran) = \{env\}$, in accordance with the semantics of the XSTS formalism.

2.5.1 The CEGAR abstractor

The abstractor builds the ARG and determines whether the target is reachable in the abstract state space. The algorithm of the abstractor is shown in Algorithm 2. It initializes the ARG if necessary, using an *initialization function* INIT that conforms to the initiation property of ARGs. The abstractor maintains a waitlist of nodes to be processed. The main loop of the algorithm takes a node from the waitlist and checks if the node is a target. If it is not a target, then the algorithm continues with attempting to cover the node by an already reached node. If no such coverage is possible, then the node is expanded, i.e. a new node is created for each successor of the abstract state represented by the node.

Algorithm 2 Constructing an ARG in CEGAR

```
1: function BUILD($\mathcal{M}$: XSTS model, $\mathcal{D} = \langle \mathcal{V}, \mathcal{S}, \sqsubseteq, \gamma, T \rangle$: abstract domain, $\varphi_t$: target
   predicate, $arg = \langle N, E, C, L_n, L_e \rangle$: ARG, $\pi$: precision)
2:     $N \leftarrow N \cup$ INIT($\mathcal{M}$, $\mathcal{D}$, $\pi$)
3:     $waitlist \leftarrow \{n \in N \mid n$ is not covered and not expanded$\}$
4:     while $n \in waitlist$ for some $n$ do
5:         $waitlist \leftarrow waitlist \setminus \{n\}$
6:         if $L_n(n)$ satisfies $\varphi_t$ then
7:             return reachable, $arg$
8:         if $L_n(n) \sqsubseteq L_n(n')$ for some $n' \in N$ then
9:             $C \leftarrow C \cup \{\langle n, n' \rangle\}$
10:        else
11:            $\langle s, \tau \rangle \leftarrow L_n(n)$
12:            for all $op \in \tau$ do
13:                for all $s' \in T^{\pi}(L_n(n), op)$ do
14:                    $L_n(n') \leftarrow s'$
15:                    $L_e(\langle n, n' \rangle) \leftarrow op$
16:                    $N \leftarrow N \cup \{n'\}$
17:                    $E \leftarrow E \cup \{\langle n, n' \rangle\}$
18:                    $waitlist \leftarrow waitlist \cup \{n'\}$
19:    return unreachable, $arg$
```

2.5.2 The CEGAR refiner

The refiner is called if the abstractor deems the target reachable. It determines whether the abstract run is concretizable, and thus reachable in the concrete state space. If it is concretizable, then it proves the reachability of the target. Otherwise, more information has to be included in the precision to exclude the infeasible abstract run from the next iteration. Then, the ARG has to be pruned to allow rebuilding it with the new, stronger precision. Algorithm 3 shows a high-level pseudocode for the refinement algorithm.

2.6 Lazy abstraction

Lazy abstraction [27, 28, 33, 41, 43] is an other technique for abstraction-based model checking. The lazy abstraction algorithm does not use fixed precisions, instead, it stores more information in the ARG in the form of an additional labeling, and occasionally modifies the L_n labeling of some nodes.

Algorithm 3 Abstraction refinement in CEGAR

```
1: function REFINE($arg = \langle N, E, C, L_n, L_e \rangle$: ARG, $\pi$: precision, $\varphi_t$: target predicate)
2:     $N_t \leftarrow \{n_t \mid n_t \in N,\ L_n(n_t)$ satisfies $\varphi_t\}$
3:     $\psi \leftarrow$ abstract run to some $n_t \in N_t$
4:     if CONCRETIZE($\psi$) = infeasible then
5:         $\pi, i \leftarrow$ REFINE($arg$, $\pi$, $\psi$)
6:         $arg \leftarrow$ PRUNE($arg$, $i$)
7:         return infeasible, $arg$, $\pi$
8:     else
9:         return reachable, $arg$, $\pi$
```

The additional labeling maintained by the lazy abstraction algorithm is the *concrete node labeling* $L_{concr}\colon N \to \mathcal{S}_{concr}$ that maps nodes to abstract states in the domain $\mathcal{D}_{concr} = \langle \mathcal{V}, \mathcal{S}_{concr}, \sqsubseteq_{concr}, \gamma_{concr}, T_{concr} \rangle$. The concrete labels $s_{concr} \in \mathcal{S}_{concr}$ are also abstract states, but restricted to only represent actually reachable states (each $val \in \gamma(L_{concr}(n))$ is a reachable state for all $n \in N$), hence the name *concrete label.* We require some additional properties to hold for the concrete labeling besides the usual properties of ARGs:

1. Concrete initiation: $Val^0 = \gamma(L_{concr}(n_0))$ where Val^0 is the set of initial states of the model.
2. Concrete inductive labeling: $T(L_{concr}(n), L_e(\langle n, n' \rangle)) = \{L_{concr}(n')\}$ for each $\langle n, n' \rangle \in E$.
3. Simulation: for each expanded node $n \in N$ and operation $op \in \mathcal{O}$ it holds that if $T_{concr}(L_{concr}(n), op) = \emptyset$ then $T(L_n(n), op) = \emptyset$.

Lazy abstraction can handle zone abstraction efficiently, and with some constraints we can also use lazy abstraction for the verification of timed automata with data variables. For both $\mathcal{D}$ and $\mathcal{D}_{concr}$ we use product abstraction combining two abstract domains: one for the locations of the automaton, and a second product domain for the combination of data abstraction and time abstraction.

The locations of the automaton are tracked explicitly, the purpose of the location abstraction is to be able to store location information in the nodes of the ARG: $\mathcal{D}_{\mathcal{L}} = \langle \mathcal{L}, \mathcal{L}, =, id_{\mathcal{L}}, T_{\mathcal{L}} \rangle$ where $\mathcal{L}$ is the set of locations of the model, $id_{\mathcal{L}}$ is the identity function of $\mathcal{L}$ and $T_{\mathcal{L}}(l, op)$ is the set of locations $l' \in \mathcal{L}$ such that $\langle l, op, l' \rangle \in \mathcal{E}$.

The used data abstraction may be different in $\mathcal{D}$ and $\mathcal{D}_{concr}$, since the efficiency of the algorithm is heavily dependent on interpolation in $\mathcal{D}$, while $\mathcal{D}_{concr}$ should be suitable for representing states with the required granularity. Let $\mathcal{D}_{concr} = \langle Val_D, \mathcal{S}_D^{concr}, \sqsubseteq_D^{concr}, \gamma_D^{concr}, T_D^{concr} \rangle$ denote the data abstraction used in $\mathcal{D}$, and $\mathcal{D}_{abstr} = \langle Val_D, \mathcal{S}_D^{abstr}, \sqsubseteq_D^{abstr}, \gamma_D^{abstr}, T_D^{abstr} \rangle$ denote the data abstraction used in $\mathcal{D}_{concr}$.

The zone domain $\mathcal{D}_{\mathcal{Z}} = \langle Val_C, \mathcal{Z}, \sqsubseteq_{\mathcal{Z}}, \gamma_{\mathcal{Z}}, T_{\mathcal{Z}} \rangle$ is a suitable and commonly used choice for time abstraction in both $\mathcal{D}$ and $\mathcal{D}_{concr}$.

Thus, lazy abstraction for the verification of timed automata with data variables uses the abstract domains $\mathcal{D} = Prod(\mathcal{D}_{\mathcal{L}}, Prod(\mathcal{D}_{abstr}, \mathcal{D}_{\mathcal{Z}}))$ and $\mathcal{D}_{concr} = Prod(\mathcal{D}_{\mathcal{L}}, Prod(\mathcal{D}_{concr}, \mathcal{D}_{\mathcal{Z}}))$. We denote the abstract transition function for $Prod(\mathcal{D}_{abstr}, \mathcal{D}_{\mathcal{Z}})$ by T_{abstr} and the abstract transition function for $Prod(\mathcal{D}_{concr}, \mathcal{D}_{\mathcal{Z}})$ by T_{concr}. The interpolation algorithms currently used in lazy abstraction presume that the transition functions T_{abstr} and T_{concr} produce at most one successor state.

Algorithm 4 shows lazy abstraction for a timed automaton with data variables. The algorithm uses an initialization function InitLazy to initialize the ARG if necessary, that conforms to the initiation properties of both node labelings. Lazy abstraction also maintains a waitlist of nodes to be processed, as well as a set of expanded nodes.

The possibility of coverage is checked on the concrete label of the potentially covered node and the abstract label of the covering node candidate. To be able to compare these abstract states, the algorithm uses the $\gamma_{abstr} : \mathcal{S}_{concr} \to \mathcal{S}$ operator that maps a concrete label to its representation in $\mathcal{D}$. When a new covered-by edge is added to the ARG, Cover is called to ensure that the coverage property of the ARG holds, the abstract label of the covered node may have to be refined to ensure this.

When expanding a node, the successors on the concrete node labeling domain are computed by the transition function, while abstract data and time states of the abstract node labels are initially set to $\top$. In case an operation cannot be fired from the concrete state ($T_{concr} = \emptyset$), then Disable is called to ensure the simulation property of the ARG by refining the abstract node labeling.

Algorithm 4 Lazy abstraction algorithm

```
1:  function Check($\mathcal{M}$: model, $\mathcal{D} = \langle \mathcal{V}, \mathcal{S}, \sqsubseteq, \gamma, T \rangle$: abstract domain, $\mathcal{D}_{concr} = \langle \mathcal{V}, \mathcal{S}_{concr}, \sqsubseteq_{concr}, \gamma_{concr}, T_{concr} \rangle$: concrete domain, $\varphi_t$: target predicate, $arg = \langle N, E, C, L_n, L_e \rangle$: ARG)
2:      $N \leftarrow N \cup$ InitLazy($\mathcal{M}$, $\mathcal{D}$, $\mathcal{D}_{concr}$)
3:      $waitlist \leftarrow \{n \in N \mid n$ is not covered and not expanded$\}$
4:      $expanded \leftarrow \{n \in N \mid n$ is expanded$\}$
5:      while $n \in waitlist$ for some $n$ do
6:          if $L_n(n)$ satisfies $\varphi_t$ then
7:              return reachable, $arg$
8:          if $\gamma_{abstr}(L_{concr}(n)) \sqsubseteq L_n(n')$ for some $n' \in expanded$ then
9:              $C \leftarrow C \cup \{\langle n, n' \rangle\}$
10:             Cover($n$, $n'$)
11:         if $n$ is not covered then
12:             $\langle l, \langle s_d, s_c \rangle \rangle \leftarrow L_{concr}(n)$
13:             for all $\langle l, op, l' \rangle$ outgoing edge from $l$ in $\mathcal{M}$ do
14:                 if $T_{concr}(\langle s_d, s_c \rangle, op) = \{\langle s'_d, s'_c \rangle\}$ then
15:                     $L_{concr}(n') \leftarrow \langle l', \langle s'_d, s'_c \rangle \rangle$
16:                     $L_n(n') \leftarrow \langle l', \langle \top, \top \rangle \rangle$
17:                     $L_e(\langle n, n' \rangle) \leftarrow op$
18:                     $N \leftarrow N \cup \{n'\}$
19:                     $E \leftarrow E \cup \{\langle n, n' \rangle\}$
20:                     $waitlist \leftarrow waitlist \cup \{n'\}$
21:                 else
22:                     Disable($n$, $op$)
23:             $expanded \leftarrow expanded \cup \{n\}$
24:     return unreachable, $arg$
```

Procedures Cover and Disable use *interpolation* [33] to refine the abstract node labeling. However, to maintain the inductive labeling property of the ARG, it is often not enough to refine the label of a single node, instead, the labels are refined on the backwards path towards the root of the ARG. These interpolation algorithms are further detailed in [43, 19].

2.7 Related work

Timed automata are widely used for the modeling and verification of timed systems, despite being a simpler, less expressive formalism than XSTS models. For this reason, models are usually manually developed by experts in the literature. An overview of the state-of-the-art in zone-based verification of timed automata is provided in [5]. Several methods have been proposed for the SMT-based analysis of timed systems. Sorea [40] proposed a model checking algorithm for timed automata extended with integer variables using *Bounded Model Checking* (BMC). Chen et al. [10] investigated several potential improvements for this approach. *K-induction* has also been employed for the verification of timed automata [21, 42]. An *IC3-based* [7] verification approach was proposed for timed automata in [29]. *Learning-based* timed automata verification was proposed in [39]. Our approach extends the ideas of these papers to iteratively verify timed behaviour and we also employ abstract domains to efficiently represent data valuations, as a significant algorithmic contribution.

Lazy abstraction for model checking was introduced in [27]. McMillan [33] proposed a modified version of the lazy algorithm using interpolants. Herbreteau et al. [28] adapted this approach to timed automata and introduced *Adaptive Simulation Graphs* (ASG) as the abstraction built during state-exploration. The results of [41] improves [28] by generalizing the abstraction to DBMs. In [43], ASG-based lazy abstraction was extended with explicit value abstraction of data variables. These approaches are efficient for timed behaviour, but their efficiency to handle data-dependent behaviour is limited.

CEGAR [12] has proven its efficiency for the verification of software-based, data intensive systems [31]: various abstract domains [4], refinement strategies [32, 25, 45] and tools [3] were introduced. Theoretical background for the applicability of CEGAR with predicate abstraction to clock variables was provided in [34], but without directions to implement the verification in practice. The algorithm proposed in [38] can be considered an implementation of this framework. Apart from a direct explicit implementation using DBMs, the authors also provide a version based on decision diagrams. However, this requires specialized data structures and variable ordering information for efficiency. In addition to running in a CEGAR loop, [38] also presents a lazy version similar to [41]. CEGAR for UPPAAL models with explicit value abstraction was implemented in [20]. Instead of creating a direct abstraction-based model checker, they use UPPAAL to check *abstract UPPAAL models* derived from the original ones by ignoring variables. These approaches use coarser abstractions for the timed domains than the second one of our proposed solutions.

The input language of the NUXMV [9] model checker was extended with clock variables in [11] for verifying LTL and MTL properties. Similarly to the first approach for verification presented in this work, they perform the analysis via a reduction to verification problems in the discrete-time case.

Functional verification of timed engineering models are often solved by transforming the engineering models to the timed automata formalism, such as it can be done in *Gamma* [23]. However, using timed automata as the background formalism restricts the expressiveness of the timing operations of the high-level formalism.

Chapter 3

Overview

In this section, an overview of the novel verification process is given. The details of the steps are given in the latter sections.

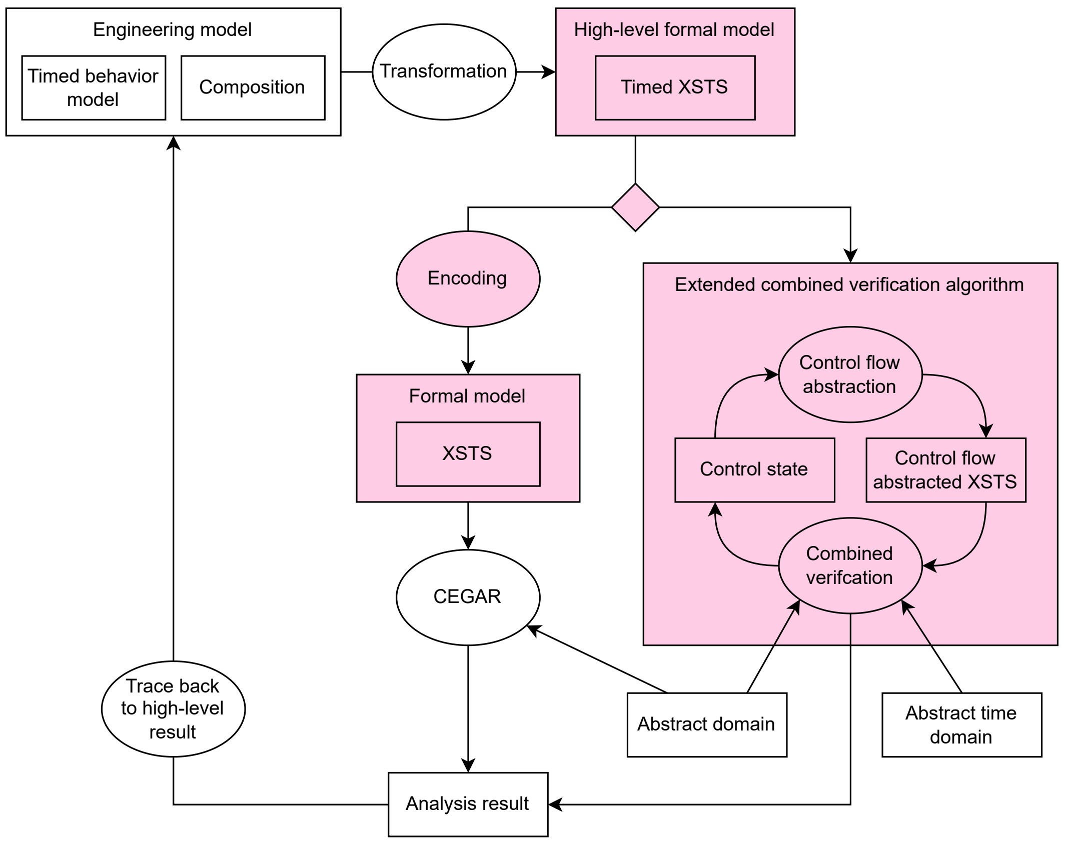

Figure 3.1: Overview of the proposed approaches

Figure 3.1 shows the architecture of the proposed abstraction-based verification approach for timed software-intensive systems.

The *inputs* of our approach are the *engineering models* describing the system to be verified, i.e.,

- the *timed behavior models* corresponding to software and platform components,

- the *composition models* describing the instances and connections of the components, and
- the *property* of the system to be verified.

We use the *Gamma Statechart Composition Language* (GSCL) [35] both as a modeling language for the component models, as well as for the composition models.

We consider *reachability properties*, where a either a given state of the system should be unreachable (e.g., an unsafe situation in the case of safety properties), or contrariwise a given state should be reached (e.g., for generating test cases).

We propose *Timed XSTS* (TXSTS), an extension to the XSTS [36] formal modeling language as a *high-level formal model* for real-time software-intensive systems. The TXSTS model is derived from the GSCL models in the *Gamma* framework by *model transformations*. Likewise, the property specification of the engineering model can be formalized as a *target predicate* in the XSTS language.

We propose two abstraction-based verification approaches for TXSTS models:

1. *encoding* into an XSTS model without timing information, and verifying the model using CEGAR [26], or
2. an *extended combined verification algorithm* that combines CEGAR with lazy timed abstraction [43].

The *parameters* of our approach are

- whether to use *encoding* or *extended combined verification*,
- the *abstract domain* (section 2.3) to employ for data abstraction, and
- in the latter algorithm, the configuration for the *abstract domain* for time abstraction (section 2.3.3).

Both algorithms provide an *analysis result*: a *trajectory* if the state encoded by the target formula is reachable, or a *proof* of unreachability. These results can be interpreted in the context of the engineering property (i.e., as a proof of safety/unsafety, or as a generated test case).

3.1 The Gamma Statechart Composition Language

Statecharts are an expressive formalism widely used in engineering applications. The Gamma Statechart Composition Framework supports the automatic import of models defined in integrated modeling front-ends [24], such as Yakindu, UML, SCXML and SysMLv2 models. The Gamma Statechart Composition Language offers powerful constructs, such as

- *Composition*: various types of composition are supported [24], including
  - *Synchronous composition*, where all system components are executed in response to ticks emitted by a clock (not to be confused with clock variables),

  - *Asynchronous composition*, where components communicate with each other using message queues,
  - *Cascade composition*, where components are executed one after another in a specific order, in a cycle initiated by a tick.

- *Hierarchical state refiement*, with *transition priorities* according to the state hierarchy (following the transition priorities of UML);
- *Transition timeouts* for modeling timed behavior.
- *Ports* for the communication of components, using messages with data.
- *Environment model* providing non-deterministic input for representing the behavior of the environment of the system.

The above constructs make statechart-based modeling highly expressive, suitable for the modeling of complex real-time software-intensive systems. However, the cost of easier modeling is more difficult analysis. An example of the challenges in analysis posed by statechart-based modeling is handling priorities and timeouts in an asynchronous environment.

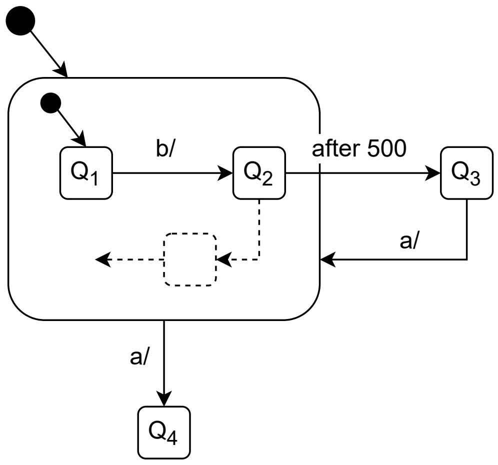

Figure 3.2: Statechart model with hierarchical state refinement

Example 1. *Figure 3.2 is a schematic example of hierarchical state refinement in Gamma. To illustrate the difficulties of the verification of these models, consider the possible transitions executions triggered by a in an asynchronous composition:*

```
if $(a)$ then
    if $((timeout \geq 500 \wedge state = Q_2) \vee state = Q_3)$ then
        $\{state := Q_1\}$
    else
        if $(state = Q_1 \vee state = Q_2 \vee \dots)$ then
            $\{state := Q_4\}$
        else
            $\{skip\}$
```

The order of the conditions follows the transition priorities of UML: the transition from the inner state Q_2 gets priority over the transition from the outer state containing Q_1 and Q_2 among other states. Note that the next state of the model cannot be determined by considering data and time successors independently. E.g., if timeout < 500, the transition *to Q_1 may still be executed if state $= Q_3$, however, if state $\neq Q_3$, then the transition to Q_4 is executed. From an other point of view, if state $= Q_2$, then the transition to Q_1 is executed if timeout ≥ 500, otherwise the transition to Q_4 is taken. Now consider the case when timeout < 500 and state $= Q_2$. The transition to Q_1 can not be executed in this case, even though it is in the intersection of transitions that can be executed from the current data state and the current time state.*

3.2 Proposed algorithms

In the *encoding* approach, an *XSTS formal model* may be derived from TXSTS by *encoding* timed behaviors as data variables. This apporach can exploit existing abstraction-based techniques for verifying data-intensive behaviors, but cannot take advantage of algorithms for verifying timed systems. In particular, we use the *CEGAR* [26] algorithm for performing the analysis.

We also propose an *extended combined verification algorithm* to exploit verification techniques for both data-intensive and timed systems. *Control flow abstraction* based on the *control state* of the XSTS model produces a *control flow abstracted TXSTS*. The abstracted model can be analyzed by a *combined verification algorithm* proposed in our previous work [18] that relies on CEGAR for data-intensive behavior and lazy timed abstraction [41] for timed behavior. Newly discovered control states are used to further expand the abstracted model.

Chapter 4

The timed XSTS formalism

Complex engineering models are easier mapped to more high-level modeling formalisms, such as the extended symbolic transition system formalism. However, the verification of real-time systems requires using modeling formalisms with the ability to represent timed behavior. We propose an extension to the XSTS formalism that enables using clock variables and operations affecting clock variables.

We explicitly reuse existing XSTS semantics (section 2.2.2) for behaviors concerning data variables. As a benefit, transformations that already produce XSTS formal models (e.g., the transformation from GCSL implemented in the Gamma framework [36, section 5.2]) can be easily extended to take timed behavior into account using the appropriate TXSTS constructs.

Definition 5 (Timed extended symbolic transition system). A *timed extended symbolic transition system* is a tuple $\mathit{TXSTS} = \langle V_D, V_C, V_{\mathit{ctrl}}, \mathit{val}^0, \mathit{init}, \mathit{env}, \mathit{tran} \rangle$ where

- V_D and V_C are finite sets of data variables and clock variables;
- $V_{\mathit{ctrl}} \subseteq V_D$ is a set of control variables;
- val^0 is the initial valuation over V_D that maps each variable $x \in V_D$ to the initial value of the variable, or $\top$ if unknown;
- $\mathit{init}, \mathit{env}, \mathit{tran} \subseteq \mathcal{O}_{\mathit{TXSTS}}$ are sets of operations representing the initialization, environmental and internal transition sets of the system. ▪

A *state* of a TXSTS model is a tuple $\langle \langle \mathit{val}_D, \mathit{val}_C \rangle, \tau \rangle$, where $\mathit{val}_D \in \mathit{Val}_D$ is a data valuation, $\mathit{val}_C \in \mathit{Val}_C$ is a clock valuation and $\tau \in \{\mathit{init}, \mathit{env}, \mathit{tran}\}$ is a transition set, which is the only transition set that can be executed in this state. We will continue using $\mathcal{T}$ to denote the set of transition sets $\{\mathit{init}, \mathit{env}, \mathit{tran}\}$. However, in TXSTS models, the transition sets in $\mathcal{T}$ take their contained operations from an extended operation set $\mathcal{O}_{\mathit{TXSTS}}$.

We use $[\![\mathit{op}]\!]_{\mathit{TXSTS}}(\langle \langle \mathit{val}_D, \mathit{val}_C \rangle, \tau \rangle)$ to denote the result of applying the operation $\mathit{op} \in \mathcal{O}_{\mathit{TXSTS}}$ on a TXSTS state. We also use the same notation for the result of applying op on some components of a state, e.g. $[\![\mathit{op}]\!]_{\mathit{TXSTS}}(\langle \mathit{val}_D, \mathit{val}_C \rangle)$ denotes applying op on the pair of valuations $\langle \mathit{val}_D, \mathit{val}_C \rangle$.

The semantics of TXSTS models regarding the τ component of states is the same as with XSTS models, $[\![\mathit{op}]\!]_{\mathit{TXSTS}}(\tau) = \emptyset$ if $\mathit{op} \notin \tau$, otherwise $[\![\mathit{op}]\!]_{\mathit{TXSTS}}(\mathit{init}) = \{\mathit{env}\}$, $[\![\mathit{op}]\!]_{\mathit{TXSTS}}(\mathit{env}) = \{\mathit{tran}\}$ and $[\![\mathit{op}]\!]_{\mathit{TXSTS}}(\mathit{tran}) = \{\mathit{env}\}$.

The set of operations $\mathcal{O}_{TXSTS}$ is a superset of $\mathcal{O}_{XSTS}$, it contains operations that affect clock variables as well. In $\mathcal{O}_{TXSTS}$, assumptions are extended so that they may contain clock constraints as well. We also introduce two new basic operations, *clock resets* and *delays*. The semantics of basic operations in $\mathcal{O}_{TXSTS}$ regarding the $\langle val_D, val_C\rangle$ component of TXSTS states is given as follows:

- *Assumptions* have the form $[\varphi]$, where φ is a Boolean combination of predicates over V_D and clock constraints over V_C. $[\![[\varphi]]\!]_{TXSTS}(\langle val_D, val_C\rangle) = \{\langle val_D, val_C\rangle\}$ if $\langle val_D, val_C\rangle$ satisfies φ, otherwise $[\![[\varphi]]\!]_{TXSTS}(\langle val_D, val_C\rangle) = \emptyset$.
- *Data assignments* have the form $x := \varphi$, they assign the value of an expression φ evaluated on $\langle val_D, val_C\rangle$ to the data variable $x \in V_D$ with domain D. The presence of clock variables in φ is restricted to clock constraints, e.g. the concrete value of a clock cannot be assigned to a data variable, but a data variable can be set to a Boolean value representing whether the value of a clock is more than a given integer value. $[\![x := \varphi]\!]_{TXSTS} = \{\langle val'_D, val_C\rangle\}$ such that $val'_D(x) = \varphi(\langle val_D, val_C\rangle)$, $\varphi(\langle val_D, val_C\rangle) \in D$, and $\forall x' \in V_D \backslash \{x\}\colon val'_D(x') = val_D(x')$.
- *Clock resets* have the form $c := n$, they set the value of a clock $c \in V_C$ to a value $n \in \mathbb{N}_0$, $[\![c := n]\!]_{TXSTS}(\langle val_D, val_C\rangle) = \{\langle val_D, val'_C\rangle\}$ such that $val'_C(c) = n$ and $\forall c' \in V_C \backslash \{c\}\colon val'_C(c') = val_C(c')$.
- A *havoc* operation denoted by $havoc(x)$ is a non-deterministic assignment to a data variable $x \in V_D$ with domain D. The semantics of havoc operations is the same as in XSTS models, $[\![havoc(x)]\!]_{TXSTS}(\langle val_D, val_C\rangle) = \{\langle val'_D, val_C\rangle \mid val'_D(x) \in D,\ \forall x' \in V_D \backslash \{x\}\colon val'_D(x') = val_D(x')\}$.
- *Delay* operations are denoted simply by *delay*. A delay increments all clocks, $[\![delay]\!]_{TXSTS}(\langle val_D, val_C\rangle) = \{\langle val_D, val^{\delta}_C\rangle \mid \delta \in \mathbb{R}_{\geq 0}\}$ where $val^{\delta}_C(c) = val_C(c) + \delta$ for all clocks $c \in V_C$.
- A *no-op* operation is denoted by *skip*, $[\![skip]\!]_{TXSTS}(\langle val_D, val_C\rangle) = \{\langle val_D, val_C\rangle\}$.

Compound operations in $\mathcal{O}_{TXSTS}$ are the same as compound operations in $\mathcal{O}_{XSTS}$: sequences, non-deterministic choices, conditional operations, and loops. They have the same semantics as well, with valuations *val* substituted by pairs of valuations $\langle val_D, val_C\rangle$.

Based on the variables affected, data assignments and havoc operations are data operations, while clock resets and delays are clock operations. Assumptions and compound operations clearly fall into both categories.

A *run* of a TXSTS is an alternating finite sequence of TXSTS states and operations $\sigma_{TXSTS} = \langle\langle val^0_D, val^0_C\rangle, init\rangle \xrightarrow{op_1} \cdots \xrightarrow{op_k} \langle\langle val^k_D, val^k_C\rangle, \tau^k\rangle$, with $\langle\langle val^i_D, val^i_C\rangle, \tau^i\rangle \in [\![op_i]\!]_{TXSTS}(\langle\langle val^{i-1}_D, val^{i-1}_C\rangle, \tau^{i-1}\rangle)$ for all $1 \leq i \leq k$. A state $\langle\langle val_D, val_C\rangle, \tau\rangle$ is *reachable* in the model if there exists a run σ_{TXSTS} such that $val^k_D = val_D$, $val^k_C = val_C$, and $\tau^k = \tau$.

Chapter 5

Verification of timed XSTS models

The possible interdependence of data and timed behavior in the composite operations of TXSTS models makes it impossible to determine successors of a state on the data domain and time domain independently, e.g. as in a product domain that then takes the Cartesian product of independently computed data and time successors. Therefore, the model checking algorithms CEGAR and lazy abstraction, discussed in sections 2.5 and 2.6 cannot be used without modifications.

We propose two approaches to handle the interdependence of clocks and data in TXSTS models. The first one transforms TXSTS models to the XSTS formalism by encoding clock variables and operations using data variables only, thus eliminating timing. Our second, more refined method transforms the operations and uses an SMT solver to obtain and enumerate purely sequential control flows without branching, where data and time successors can be computed independently, while using a combined abstraction algorithm for the verification. We present these two approaches in the following sections.

5.1 Encoding timed behavior using data variables

We define a transformation that transforms a TXSTS model to an XSTS model. This transformation takes place as a preprocessing step, the verification is then run on the resulting XSTS model. This way, the CEGAR algorithm discussed in section 2.5 can be used without any modification.

The transformation is done by replacing clock variables with data variables and encoding clock operations of the TXSTS as data operations of the XSTS. Since clock variables are continuous variables, they are mapped to rational variables to avoid discretization errors.

Let $rat(c)$ denote the rational data variable that corresponds to $c \in V_C$ in the transformed model. The mapping $R\colon \mathcal{O}_{TXSTS} \to \mathcal{O}_{XSTS}$ maps operations to their corresponding replacements, e.g. $R(c := 0) = (rat(c) := 0)$. In the following, we define the precise semantics of these replacements for a TXSTS model with clock variables V_C.

- Assumptions: $R([\varphi]) = [\varphi']$, where φ' is φ with each clock variable $c \in V_C$ appearing in φ replaced by $rat(c)$ in φ'. Since clock variables are restricted in $\mathcal{O}_{TXSTS}$ to appear only in clock constraints, only clock constraints are transformed, to expressions of the form $rat(c_i) \sim k$ or $rat(c_i) - rat(c_j) \sim k$, where $c_i, c_j \in V_C$, $\sim \in \{<, \leq, =, \geq, >\}$, and $k \in \mathbb{Z}$. E.g., for variables $x \in V_D$ and $c_1, c_2 \in V_C$, $R([(x = 0) \vee (c_1 - c_2 < 2)]) = ((x = 0) \vee (rat(c_1) - rat(c_2) < 2))$.

- Data assignments: The assigned expression may contain clock constraints, therefore the assigned expression also has to be transformed. If $R([\varphi]) = [\varphi']$, then $R(x := \varphi) = (x := \varphi')$ for any $x \in V_D$.
- Clock resets: The clock is replaced by the corresponding rational variable, the resulting operation is a data assignment $R(c := n) = (rat(c) := n)$ for any $c \in V_C$ and $n \in \mathbb{N}_0$.
- Delays: We introduce a new rational variable δ, which can be the same variable for all delay operations. To this new variable we assign a value using a havoc operation, followed by an assumption to ensure that it is non-negative, then each clock variable replacement is incremented by this non-deterministically chosen non-negative value: $R(delay)$ is a sequence of $havoc(\delta)$, the assumption $[\delta \geq 0]$, and the data assignments $rat(c) := rat(c) + \delta$ for each clock variable $c \in V_C$.
- Havoc and no-op operations are left unmodified: $R(havoc(x)) = havoc(x)$ for any $x \in V_D$, and $R(skip) = skip$.

In composite operations, all components are transformed individually:

- Sequences: $R(op_1, op_2, \ldots, op_n) = R(op_1), R(op_2), \ldots, R(op_n)$
- Non-deterministic choices: $R(\{op_1\}\, or\, \{op_2\}\, or\, \ldots\, or\, \{op_n\}) = \{R(op_1)\}\, or\, \{R(op_2)\}\, or\, \ldots\, or\, \{R(op_n)\}$
- Conditional operations: if $R([\varphi]) = [\varphi']$, then $R(if(\varphi)\, then\, \{op_1\}\, else\, \{op_2\}) = if(\varphi')\, then\, \{R(op_1)\}\, else\, \{R(op_2)\}$
- Loops: $R(for\ i\ from\ \varphi_a\ to\ \varphi_b\ do\ \{op\}) = for\ i\ from\ \varphi_a\ to\ \varphi_b\ do\ \{R(op)\}$

Given a TXSTS model $\mathcal{M} = \langle V_D, V_C, V_{ctrl}, val^0, init, env, tran\rangle$, the transformed model is the XSTS $\mathcal{M}' = \langle V', V_{ctrl}, val'_0, init', env', tran'\rangle$, where $V' = V_D \cup \{rat(c) \mid c \in V_C\} \cup \{\delta\}$, $val'_0(x) = val^0(x)$ for each $x \in V_D$ and $val'_0(rat(c)) = 0$ for each $c \in V_C$ in accordance with the properties of clock variables, $init' = \{R(op) \mid op \in init\}$, $env' = \{R(op) \mid op \in env\}$, and $tran' = \{R(op) \mid op \in tran\}$.

Example 2. *Consider the TXSTS model $\mathcal{M} = \langle V_D, V_C, V_{ctrl}, val^0, init, env, tran\rangle$, where*

- $V_D = V_{ctrl} = \{state\}$,
- $V_C = \{c_1, c_2\}$,
- $val^0 = \{state \mapsto Q_0\}$,
- $init = env = \{skip\}$, *and*
- *tran consist of one sequence operation:*

```
  delay,
  if ((c_1 ≥ 500 ∧ state = Q_2) ∨ state = Q_3) then
      {state := Q_1, c_2 := 0}
  else
      ...
```

Then, in the transformed model, the clock c_1 is replaced by a rational variable $r_1 = rat(c_1)$ and c_2 is replaced by the rational variable $r_2 = rat(c_2)$. The transformed model is the XSTS $\mathcal{M}' = \langle V', V_{ctrl}, val'_0, init', env', tran' \rangle$.

The sets of data and clock variables $V_D = \{state\}$ and $V_C = \{c_1, c_2\}$ are merged into one set of data variables with the additional variable for delays: $V' = \{state, r_1, r_2, \delta\}$. The set of control variables is unchanged: $V_{ctrl} = \{state\}$. The initial valuation is extended with the new rational variables: $val'_0 = \{state \mapsto Q_0, r_1 \mapsto 0, r_2 \mapsto 0, \delta \mapsto \top\}$.

In our example, the init and env transition sets consist of a single skip operation, therefore the $init'$ and env' sets of the transformed model stay the same. The delay operation in the tran set of the TXSTS is replaced by a sequence of data operations. First the δ variable is set: $havoc(\delta), [\delta \geq 0]$. Then r_1 and r_2 are incremented: $r_1 := r_1 + \delta, r_2 := r_2 + \delta$. In other sub-operations the clocks c_1 and c_2 are replaced by r_1 and r_2, changing clock operations into data operations without syntactical changes. The whole transformed operation forming $tran'$ can be seen below:

```
$havoc(\delta)$,
$[\delta \geq 0]$,
$r_1 := r_1 + \delta$,
$r_2 := r_2 + \delta$,
if $((r_1 \geq 500 \land state = Q_2) \lor state = Q_3)$ then
    $\{state := Q_1, r_2 := 0\}$
else
    ...
```

The transformed model has the same semantics as the original model but does not contain any clock variables. It can be verified with CEGAR as described in section 2.5, although the verification may take longer, as zone abstraction is usually more efficient for representing timed behavior than abstractions using rational variables with SMT solvers.

5.2 Control flow splitting with Boolean flags

In our previous work [18], we proposed a novel algorithm combining CEGAR and lazy abstraction for the verification of timed automata with data variables. This combined algorithm preserves the advantages of both approaches, it allows the use of non-deterministic data operations, as well as efficient data abstractions of CEGAR, while using the more efficient time abstraction of lazy abstraction. Despite its efficiency, the combined algorithm still requires data and clock operations to be independent in the analyzed models.

We propose a novel method that splits the control flow of the operations of transition sets into operations without branching, where data and time behavior are independent. With a slight modification that utilizes our control flow splitting method, our combined abstraction algorithm can be adapted to verify TXSTS models.

The main steps of our control flow splitting method are presented in Algorithm 5. In the following, we give a detailed explanation of each step, then show how control flow splitting can be incorporated in the combined verification algorithm.

5.2.1 Operation simplification by substitution

The first step of our control flow splitting algorithm is *operation substitution* [37]. This means that the explicitly tracked variables in the abstract state s are inlined in

Algorithm 5 Control flow splitting

```
1: function $\textsc{CfSplit}(op\text{: operation}, s\text{: abstract state})$
2:     $op' \leftarrow \textsc{Simplify}(op, s)$                ▷ variable inlining, loop unfolding, etc.
3:     $root \leftarrow \textsc{SelectFrom}(V_B)$      ▷ select one from the set of unused Boolean flags $V_B$
4:     $\langle op'', A\rangle \leftarrow F(op', root)$    ▷ get transformed operation and a FOL formula over $V_B$
5:     $flags \leftarrow V_B \cap \textsc{Vars}(op'')$       ▷ collect Boolean flags of the transformed operation
6:     $\varphi \leftarrow \textsc{SmtFormula}(root, A, s, op'')$         ▷ create formula for the SMT solver
7:     $M \leftarrow \textsc{AllSat}(\varphi, flags)$           ▷ get all flag assignments satisfying the formula
8:     $Op \leftarrow \{\textsc{Simplify}(op'', m) \mid m \in M\}$   ▷ extract ctrl flows for satisfying assignments
9:     return $Op$
```

the operation *op*, as well as loops with known bounds are unfolded, i.e. for a loop $for\ i\ from\ \varphi_a\ to\ \varphi_b\ do\ \{op\}$, if the values of φ_a and φ_b are known, then the loop is substituted by a sequence where *op* is repeated the correct number of times and the loop variable i is assigned to the correct value before each instance of *op* in the sequence operation. The variable inlining may also simplify some control flows, e.g. conditional operations with the condition *true* or $false$ can be substituted by the corresponding branch operation.

Example 3. *For a sequence of a loop and a conditional operation*

$for\ i\ from\ 0\ to\ x\ do\ \{y := y + 1\},$
$if(x < 5)\ then\ \{x := x + 5\}\ else\ \{y := y/2\}$

the result of the substitution with the abstract state $\{x \mapsto 2\}$ *is the following sequence:*

$i := 0,$
$y := y + 1,$
$i := 1,$
$y := y + 1,$
$x := 7$

Since the bound of the loop x is an explicitly tracked variable in this example, the loop could be unfolded, eliminating the loop operation from the result. Furthermore, as the condition $x < 5$ evaluates to true, the conditional operation can be substituted by its $x := x + 5$ branch, which can be further simplified to $x := 7$.

5.2.2 Operation transformation with Boolean flags

The next step of the algorithm further transforms the operations, including introducing new Boolean variables in the operation. The two main objectives of the transformation are that each satisfying assignment to these Boolean flags should determine a sequential control flow where data and clock operations are independent, and secondly, given the transformed operation and a satisfying flag assignment, the corresponding sequential control flow can be extracted.

The transformation is denoted by $F\colon \mathcal{O}_{TXSTS} \times V_B \rightarrow \mathcal{O}_{TXSTS} \times \Phi$, where V_B is the set of Boolean variables not contained by the model ($V_D \cap V_B = \emptyset$) and Φ is the set of first order logic formulas over V_B.

The input of the transformation is the operation and a *parent flag*. The value of the parent flag is unknown at the time of the operation transformation, in a resulting flag assignment it evaluates to *true* if and only if the operation is on an *active branch*, i.e. it is contained by the control flow selected by the given flag assignment. This enables handling compound operations.

The *root flag* is the Boolean flag that the transformation of a separate operation of a transition set (as opposed to sub-operations of a compound operation) is parameterized with. The root flag always evaluates to *true*.

The output consists of the transformed operation and a FOL formula, which is the conjunction of the *flag constraints* that should hold.

In the following, we specify the transformation F for each type of operation, starting with composite operations. Of course, the operations not containing any clock operations can be left unmodified to optimize the algorithm, in these cases $F(op, p) = \langle op, \top \rangle$.

5.2.2.1 Sequences

A sequence is transformed by transforming all its sub-operations, while all flag constraints entailed by the individual transformations should hold: $F((op_1, op_2, \dots, op_n), p) = \langle (op'_1, op'_2, \dots, op'_n), (A_1 \wedge A_2 \wedge \dots \wedge A_n) \rangle$ if $F(op_i, p) = \langle op'_i, A_i \rangle$ for all $1 \leq i \leq n$.

5.2.2.2 Conditional operations

Conditional operations may directly introduce cases where timing and data state have to be considered simultaneously to determine control flows, e.g. if a condition is $(x = 0 \wedge c \leq 5) \vee x = 1$ where $x \in V_D$ and $c \in V_C$, it cannot be determined whether the clock constraint $c \leq 5$ holds in the executed branch without taking into consideration the data state as well.

To avoid these cases, a new conditional operation is created for both first and second branch of the original conditional operation, where the conditions are newly introduced Boolean flags. For better understanding of the transformation, we show it decomposed into multiple steps, starting from the original conditional operation $if(\varphi)\, then\, \{op_1\}\, else\, \{op_2\}$:

1. the operation is replaced by a sequence of two operations:
   - $if(b_1)\, then\, \{op_1\}\, else\, \{skip\}$,
   - $if(b_2)\, then\, \{op_2\}\, else\, \{skip\}$,
2. in the result, the branch containing op_1 is extended by an assumption of the original condition φ, and the branch containing op_2 is extended by an assumption of the negation of φ:
   - $if(b_1)\, then\, \{[\varphi], op_1\}\, else\, \{skip\}$,
   - $if(b_2)\, then\, \{[\neg\varphi], op_2\}\, else\, \{skip\}$,
3. the sub-operations $[\varphi], op_1$ and $[\neg\varphi], op_2$ are replaced by their transformations, with b_1 and b_2 as parent flags:
   - $if(b_1)\, then\, \{op'_1\}\, else\, \{skip\}$, where op'_1 is determined by $F(([\varphi], op_1), b_1) = \langle op'_1, A_1 \rangle$,
   - $if(b_2)\, then\, \{op'_2\}\, else\, \{skip\}$, where op'_2 is determined by $F(([\neg\varphi], op_2), b_2) = \langle op'_2, A_2 \rangle$.

To ensure that the transformation allows the same control flows as the original operation, we introduce constraints on the new Boolean flags:

- A_1 and A_2, resulting from transforming $[\varphi], op_1$ and $[\neg\varphi], op_2$;
- $p \Rightarrow (b_1 \not\Leftrightarrow b_2)$, which intuitively means that exactly one branch of the original operation is executed (by $b_1 \not\Leftrightarrow b_2$), if this operation is the active branch (i.e. if p is true); selecting exactly one branch, even though the original condition and its negation are still present in the operations as assumptions, ensures that a satisfying flag assignment identifies a single control flow;
- $\neg p \Rightarrow \neg b_1$ and $\neg p \Rightarrow \neg b_2$, to optimize the algorithm by ensuring that if this operation is not on the active branch, then only one satisfying flag assignment exists for the flags of this operation (that sets both flags to $false$).

The resulting flag constraint of the transformation is the conjunction of the above expressions.

Example 4. *The transformation of the conditional operation*

```
if ($state = Q_2$) then
    $state := Q_1$
else
    if ($state = Q_3$) then
        $state := Q_4$
    else
        skip
```

with the parent flag p yields the operation

```
if ($b_1$) then
    [$state = Q_2$]
    $state := Q_1$
if ($b_2$) then
    [$state \neq Q_2$]
    if ($b_3$) then
        [$state = Q_3$]
        $state := Q_4$
    if ($b_4$) then
        [$state \neq Q_3$]
        skip
```

and the flag constraint $(p \Rightarrow (b_1 \not\Leftrightarrow b_2)) \wedge (\neg p \Rightarrow \neg b_1) \wedge (\neg p \Rightarrow \neg b_2) \wedge (b_2 \Rightarrow (b_3 \not\Leftrightarrow b_4)) \wedge (\neg b_2 \Rightarrow \neg b_3) \wedge (\neg b_2 \Rightarrow \neg b_4)$.

The satisfying assignments for the flag constraint such that $p = true$ *are the following:*

- $b_1 \mapsto true,\ b_2 \mapsto false,\ b_3 \mapsto false,\ b_4 \mapsto false$, *which corresponds to the control flow* $[state = Q_2], state := Q_1$,
- $b_1 \mapsto false,\ b_2 \mapsto true,\ b_3 \mapsto true,\ b_4 \mapsto false$, *corresponding to the control flow* $[state \neq Q_2], [state = Q_3], state := Q_4$,
- $b_1 \mapsto false,\ b_2 \mapsto true,\ b_3 \mapsto false,\ b_4 \mapsto true$, *corresponding to* $[state \neq Q_2], [state \neq Q_3], skip$.

5.2.2.3 Non-deterministic choices

The transformation of non-deterministic choices is similar to conditional choices. For each branch of the non-deterministic choice, a new conditional operation is created, where the condition is a new Boolean flag. The flag constraints that accompany the transformed operation must ensure that exactly one branch of the non-deterministic choice is executed, i.e. exactly one of the Boolean flags serving as conditions may evaluate to *true*.

For a non-deterministic choice $\{op_1\}\ or\ \{op_2\}\ or\ \dots\ or\ \{op_n\}$, the resulting operation is a sequence of operations of the form $if(b_i)\ then\ \{op'_i\}\ else\ \{skip\}$, where $b_i \in V_B$ is a new Boolean flag, and op'_i is determined by the transformation of op_i: $F(op_i, b_i) = \langle op'_i, A_i \rangle$.

The required expressions are the following, their conjunction forming the resulting flag constraint of the transformation:

- $A_1, A_2, \dots, A_n$, resulting from transforming $op_1, op_2, \dots, op_n$;
- $p \Rightarrow \bigvee_{i \in N}(b_i \land \bigwedge_{j \in N \setminus \{i\}} \neg b_j)$, where $N = \{1, 2, \dots, n\}$, to ensure that exactly one branch is executed;
- $\neg p \Rightarrow \neg b_i$ for all $1 \le i \le n$, to optimize the algorithm by ensuring that all flags are always set to $false$ if the non-deterministic choice is not on the active branch.

Example 5. *The transformation of the non-deterministic choice*

$\{x := 0,\ c := 0\}\ or\ \{y := 5\}\ or\ \{y := 6\}$

produces the operation

if (b_1) ***then*** $\{x := 0,\ c := 0\}$
if (b_2) ***then*** $\{y := 5\}$
if (b_3) ***then*** $\{y := 6\}$

where branches that contain only a skip operation were omitted.

With a parent flag p, the resulting flag constraint is $(p \Rightarrow (b_1 \land \neg b_2 \land \neg b_3) \lor (\neg b_1 \land b_2 \land \neg b_3) \lor (\neg b_1 \land \neg b_2 \land b_3)) \land (\neg p \Rightarrow \neg b_1) \land (\neg p \Rightarrow \neg b_2) \land (\neg p \Rightarrow \neg b_3)$. *In the satisfying flag assignments such that p = true, exactly one of* b_1, b_2, b_3 *evaluates to true, corresponding to exactly one branch of the non-deterministic choice operation.*

5.2.2.4 Loops

F is not defined for loops; after loop unfolding, the resulting operation should not contain any loop operations.

5.2.2.5 Assumptions

Given an assumption $[\varphi]$, the formula φ is first transformed so that the only logical connectives that may appear in it are $\neg$, $\land$ and $\lor$, and its negated subformulas do not contain any logical connectives. E.g., $p \Rightarrow (\neg(q \land r) \land (c > 0))$ is transformed to $\neg p \lor ((\neg q \lor \neg r) \land (c > 0))$.

If φ in the here described form does not contain any logical connectives except for negation, then data and clock interdependence is not possible, therefore $F([\varphi], p) = \langle [\varphi], \top \rangle$.

An assumption where the assumed formula is the conjunction of some subformulas is equivalent to the sequence of separate assumptions with those subformulas. Therefore, the transformation of these assumptions can be reduced to the transformation of a sequence: $F([\bigwedge_{i=1}^{n} \varphi_i], p) = F(([\varphi_1], [\varphi_2], \ldots, [\varphi_n]), p)$.

If the assumed formula is the disjunction of some subformulas, then the assumption is equivalent to a non-deterministic choice between the assumptions of the subformulas, so the transformation can be reduced to the transformation of a non-deterministic choice: $F([\bigvee_{i=1}^{n} \varphi_i], p) = F(\{[\varphi_1]\}\ or\ \{[\varphi_2]\}\ or\ \ldots\ or\ \{[\varphi_n]\}, p)$.

Note that the assumptions that should be further transformed (as parts of a sequence or non-deterministic choice) contain smaller formulas that the original assumption. This ensures that the transformation of assumptions does not get stuck in an infinite loop.

Example 6. *We show the transformation of the assumption* $[(timeout \geq 500 \wedge state = Q_2) \vee state = Q_3]$*. First it is transformed to a non-deterministic choice* $\{[timeout \geq 500 \wedge state = Q_2]\}$ *or* $\{[state = Q_3]\}$*, which transforms to*

```
    if (b_1) then {op'_1}
    if (b_2) then {op'_2}
```

such that op'_1 *is determined by* $F([timeout \geq 500 \wedge state = Q_2], b_1) = \langle op'_1, A_1 \rangle$ *and* op'_2 *is determined by* $F([state = Q_3], b_2) = \langle op'_2, A_2 \rangle$*, with the flag constraints* A_1*,* A_2 *and* $(p \Rightarrow (b_1 \wedge \neg b_2) \vee (\neg b_1 \wedge b_2)) \wedge (\neg p \Rightarrow \neg b_1) \wedge (\neg p \Rightarrow \neg b_2)$ *if the parent flag is* p*.*

To obtain op'_1*, the assumption* $[timeout \geq 500 \wedge state = Q_2]$ *is transformed to* $[timeout \geq 500], [state = Q_2]$ *without introducing any flag constraints* ($A_1 = \top$)*. The assumption* $[state = Q_3]$ *is not transformed further to get* op'_2*, it is already in the desired form* ($A_2 = \top$)*.*

With the substeps of the transformation put together, the transformation of the assumption $[(timeout \geq 500 \wedge state = Q_2) \vee state = Q_3]$ *is the operation*

```
    if (b_1) then [timeout ≥ 500], [state = Q_2]
    if (b_2) then [state = Q_3]
```

with the flag constraint $(p \Rightarrow (b_1 \wedge \neg b_2) \vee (\neg b_1 \wedge b_2)) \wedge (\neg p \Rightarrow \neg b_1) \wedge (\neg p \Rightarrow \neg b_2)$ *if the parent flag is* p*.*

5.2.2.6 Data assignments

A data assignment may contain both data and clock variables only if an expression containing a clock constraint is assigned to a Boolean variable. In that case, the transformation is reduced to the transformation of a conditional operation: $F(x := \varphi, p) = F(if(\varphi)\, then\, \{x := true\}\, else\, \{x := false\}, p)$. Otherwise, the assignment is left unmodified: $F(x := \varphi, p) = \langle x := \varphi, \top \rangle$.

5.2.2.7 Other basic operations

All other basic operations are left unmodified, and consequently there are no flag constraints: for an operation op that is either a havoc, no-op, clock reset or delay, $F(op, p) = \langle op, \top \rangle$.

5.2.3 Using an SMT solver to determine control flows

We provide two different methods for obtaining assignments for the Boolean flags introduced in the operation transformation. In both cases, the flag constraints have to be satisfied and the root flag should evaluate to *true*. The first variant uses the SMT solver to solve an all solutions SAT problem for the above constraints. The second variant also solves an all solutions SAT problem, but includes the FOL representations of the abstract state and the transformed operation in the SMT query as well (in this case, the transformation R is applied to the operation, since the delay operation does not have a FOL representation). This approach leaves out results representing control flows that are infeasible from the current abstract state, although the SMT solver has to handle larger formulas with more variables.

The control flow splitting transforms operations in such manner, that all remaining branching in the operation comes from conditional operations with a single Boolean flag as its condition. This way, substituting the values to these Boolean flags from a satisfying flag assignment results in an operation where each condition of conditional operations evaluates to either *true* or $false$. Therefore, variable inlining and control flow simplification described in section 5.2.1 yields a sequential control flow without branching. It should also be noted that none of the operations in this purely sequential control flow contains data and clock variables simultaneously, since that can only occur in assumptions and data assignments, but assumptions are also split during the control flow splitting transformation, and data assignments containing clocks are transformed so that the assigned expression becomes the condition of a conditional operation, which+ is further transformed as an assumption.

Extracting all control flows identified by the satisfying Boolean flag assignments in the above described way concludes our control flow splitting algorithm.

5.2.4 Soundness of control flow splitting

To prove the soundness of our control flow splitting algorithm, we have to show that each satisfying Boolean flag assignment where the root flag evaluates to *true* identifies a control flow of the original operation, and that for each control flow of the original operation there exists a Boolean flag assignment that identifies that same control flow.

It is easy to see that all transformations satisfy the above two statements using structural induction with the induction hypothesis that the statements hold for transformations of sub-operations. As an example, we show the correctness of the transformation of conditional operations.

Using the same notation as before, the only satisfying assignment of the flags of this operation where $p = false$ is $\{p \mapsto false, b_1 \mapsto false, b_2 \mapsto false\}$, and it identifies the control flow where this operation is not executed. The second statement also holds, as for any control flow that does not contain this operation, $\{p \mapsto false, b_1 \mapsto false, b_2 \mapsto false\}$ is a satisfying flag assignment of the flags of this operation.

There are two satisfying flag assignments where $p = true$, it is either $\{p \mapsto true, b_1 \mapsto true, b_2 \mapsto false\}$ or $\{p \mapsto true, b_1 \mapsto false, b_2 \mapsto true\}$. In the first case inlining the flags results in an operation op'_1, determined by $F(([\varphi], op_1), b_1) = \langle op'_1, A_1 \rangle$. Assuming that the induction hypothesis holds, i.e. op'_1 results from the sound transformation of the sequence $[\varphi], op_1$, this corresponds to the control flow of the original operation where the condition φ evaluates to *true* and the first branch (op_1) is executed. The second case after variable inlining is the result of the sound transformation of the sequence $[\neg\varphi], op_2$, which corresponds to the control flow where the condition evaluated to $false$ (hence $\neg\varphi$ holds) and the second branch (op_2) is taken. Only these two control flows are possible in the original operation if it is on the active branch (the condition either holds or not), therefore the second statement also holds.

5.3 Verification of timed XSTS with combined abstraction

Combined abstraction, presented in our previous work [18], utilizes lazy abstraction for the zone abstraction of timed behavior, while still using CEGAR for efficient data abstraction.

The principle idea of the combined algorithm is to use lazy abstraction as the abstraction step of CEGAR instead of the usual CEGAR abstractor. On the level of the CEGAR loop only the data projection of the model is taken into consideration. In the lazy abstractor, data are not included in the concrete labeling, only in the abstract labeling. The data component of abstract labels is computed the same way as in CEGAR, with the given precision. Conversely, the time projection of the model is only considered inside the lazy abstractor, where time abstraction is handled the same way as in pure lazy abstraction, as described in section 2.6.

States of TXSTS models consist of a data component, a clock component, and the next transition set to be executed. The ARG being built by the combined algorithm contains information about all three state components in its abstract labels, but only clock information is necessary to include in the concrete labels, since lazy abstraction is run only on the time projection. Therefore, the abstract labeling domain is a product domain of a data domain $\mathcal{D}_{data} = \langle \mathcal{V}, \mathcal{S}_{data}, \sqsubseteq_{data}, \gamma_{data}, T_{data} \rangle$, the zone domain $\mathcal{D}_{\mathcal{Z}}$ used for time abstraction, and the $\mathcal{D}_{\mathcal{T}}$ domain introduced in section 2.5 for preserving information about which transition set should be executed next. The concrete labeling domain is the simple zone domain $\mathcal{D}_{\mathcal{Z}}$.

We use the notation $\mathcal{D}_{\text{CEGAR}}$ to refer to $Prod(\mathcal{D}_{data}, \mathcal{D}_{\mathcal{T}})$, which is the abstract domain of the components handled by CEGAR in the combined algorithm, with its corresponding transfer function T_{CEGAR} and preorder operator $\sqsubseteq_{\text{CEGAR}}$. For simplicity, we will refer to this domain as the data domain. For a node n with abstract label $L_n(n) = \langle s_{data}, s_{zone}, \tau \rangle$, $s_{data} \in \mathcal{S}_{data}$, $s_{zone} \in \mathcal{Z}$, $\tau \in \mathcal{T}$, we use $d(n)$ to refer to $\langle s_{data}, \tau \rangle$, i.e. the components handled by CEGAR. We also use $z_a(n)$ and $z_c(n)$ to refer to the abstract and concrete zone of n, i.e. s_{zone} and $L_{concr}(n)$.

The adaptation of the combined algorithm to TXSTS models, completed with control flow splitting is shown in Algorithm 6, and its abstraction substep in Algorithm 7.

The main loop of the combined algorithm is the well-known CEGAR loop, however, the ARG is built by a different abstractor. The abstractor, in contrast with lazy abstraction, is parameterized by a precision π, however, this is a data precision, relevant only for the data abstraction.

The combined abstractor checks the possibility of coverage on the data domain in the same way as in CEGAR, by checking the preorder relation between the two abstract labels, while on the time domain it checks the preorder relation between the concrete and abstract zones, as in lazy abstraction. The interpolation algorithm Cover is also called when adding a covered-by edge, as in lazy abstraction, however, here only the time abstraction has to be refined.

Algorithm 6 Combined algorithm main loop

```
1: function CHECK($\mathcal{M}$: TXSTS model, $\mathcal{D}_{\text{CEGAR}}$: abstract data domain, $\mathcal{D}_{\mathcal{Z}}$: abstract time domain, $\varphi_t$: target predicate, $\pi_0$: initial precision)
2:     $\pi \leftarrow \pi_0$
3:     $arg \leftarrow \langle \emptyset, \emptyset, \emptyset, L_n, L_e \rangle$
4:     loop
5:         $R_a$, $arg \leftarrow$ BUILDCOMBINED($\mathcal{M}$, $\mathcal{D}_{data}$, $\mathcal{D}_{\mathcal{Z}}$, $\varphi_t$, $arg$, $\pi$)
6:         if $R_a$ = unreachable then
7:             return unreachable, $arg$
8:         else
9:             $R_r$, $arg$, $\pi \leftarrow$ REFINE($arg$, $\pi$)
10:            if $R_r$ = reachable then
11:                return reachable, $arg$
```

Algorithm 7 Combined abstractor

```
1: function BUILDCOMBINED($\mathcal{M}$: TXSTS model, $\mathcal{D}_{\text{CEGAR}}$: abstract data domain, $\mathcal{D}_{\mathcal{Z}}$: abstract time domain, $\varphi_t$: target predicate, $arg = \langle N, E, C, L_n, L_e \rangle$: ARG, $\pi$: precision)
2:     $N \leftarrow N \cup$ INITCOMBINED($\mathcal{M}, \mathcal{D}_{\text{CEGAR}}, \mathcal{D}_{\mathcal{Z}}, \mathcal{D}_{\mathcal{Z}}$)
3:     $waitlist \leftarrow \{n \in N \mid n$ is not covered and not expanded$\}$
4:     $expanded \leftarrow \{n \in N \mid n$ is expanded$\}$
5:     while $n \in waitlist$ for some $n$ do
6:         if $L_n(n)$ satisfies $\varphi_t$ then
7:             return reachable, $arg$
8:         if $d(n) \sqsubseteq_{\text{CEGAR}} d(n')$ and $z_c(n) \sqsubseteq_{\mathcal{Z}} z_a(n')$ for some $n' \in expanded$ then
9:             $C \leftarrow C \cup \{\langle n, n' \rangle\}$
10:            COVER($n$, $n'$)
11:        if $n$ is not covered then
12:            for all $op \in \tau$ such that $\langle \cdot, \tau \rangle = d(n)$ do
13:                for all $op' \in$ CFSPLIT($op$, $s$) do
14:                    if $T_{\mathcal{Z}}(z_c(n), op') = \{s'_{zone}\}$ then
15:                        for all $\langle s'_{data}, \tau' \rangle \in T^{\pi}_{CEGAR}(d(n), op')$ do
16:                            $L_{concr}(n') \leftarrow s'_{zone}$
17:                            $L_n(n') \leftarrow \langle s'_{data}, \top, \tau' \rangle$
18:                            $L_e(\langle n, n' \rangle) \leftarrow op'$
19:                            $N \leftarrow N \cup \{n'\}$
20:                            $E \leftarrow E \cup \{\langle n, n' \rangle\}$
21:                            $waitlist \leftarrow waitlist \cup \{n'\}$
22:                    else
23:                        DISABLE($n$, $op'$)
24:            $expanded \leftarrow expanded \cup \{n\}$
25:    return unreachable, $arg$
```

The combined algorithm determines the successors of a state in a more complex way. The operations contained by the correct transition set are first processed by our control flow splitting algorithm CfSplit, yielding a new set of operations that are already compatible with the abstraction algorithms using product abstraction. The successor on the time domain is computed the same way as in lazy abstraction: it is computed from the concrete zone, while the abstract zone of the new node is $\top$. If the successor on the concrete labeling domain does not exists, the Disable method of lazy abstraction is called, again refining only the time abstraction. Otherwise, the data successors are computed, on the abstract data domain with the current data precision π, as in CEGAR.

In contrast with lazy abstraction, reachable result in the combined abstractor does not mean that the target is actually reachable. It only indicates that it is reachable on the concrete time domain and the abstract data domain. Therefore, the Refine method of CEGAR is called to check the feasibility of the abstract path to the target state on the data domain, and refine the data precision π if needed.

Chapter 6

Evaluation

6.1 Implementation

We implemented our proposed solutions in the Theta open source configurable model checking framework [44]. Theta has already supported the XSTS formalism, and the CEGAR algorithm for the verification of XSTS models, among other formalisms (non-timed formalisms and timed automata). We extended the XSTS language to support TXSTS. We also implemented the transformation of TXSTS models to XSTS. Theta has already supported the combined abstraction algorithm as the result of our previous work. To adapt the combined algorithm for verifying TXSTS models directly, we implemented the control flow splitting algorithm, including the operation transformation.

In the Theta framework the XSTS formalism and the algorithms associated with it are implemented in three separate modules. The `xsts` module contains the domain-specific language for the XSTS formalism and classes for representing XSTS models. Classes of the `xsts-analysis` module enable the model checking of XSTS models. The `xsts-cli` module provides a command-line user interface for the model checker.

6.1.1 The TXSTS formalism

The TXSTS formalism was implemented by extending the domain-specific language for the XSTS formalism in the `xsts` module written in Antlr[1]. We added support for declaring variables of a new clock type with the keyword `clock`, as well as for delay operations with the keyword `__delay`. We also implemented handling clock resets and clocks appearing in assumptions and data assignments in the parser, taking into consideration that clocks can only be reset to integer literals and that clocks in assumptions and data assignments are restricted to clock constraints.

6.1.2 Transformation to the XSTS formalism

Our first approach proposed for the verification of TXSTS models, transformation to XSTS, was implemented as a preprocessing step of the verification, taking place in the construction of the verification configuration. A lookup table is created for the rational replacements of clocks that contains the mapping $c \mapsto rat(c)$ for each $c \in V_C$. Then the

[1]`https://www.antlr.org`

transformation is done strictly corresponding to section 5.1, with $rat(c)$ obtained for a clock variable $c \in V_C$ from the previously created lookup table.

6.1.3 Control flow splitting

The abstractors in Theta, including the combined abstractor described in section 5.3, use a *labeled transition system* (LTS) component for obtaining the enabled actions in a given state. The LTS classes used in the analysis of XSTS and TXSTS models can be found in the `xsts-analysis` module.

In our solution, the default LTS used in combined abstraction is replaced by an LTS implementing the control flow splitting algorithm.

For operation simplification performed at both the beginning and the end of our control flow splitting algorithm, we use the already implemented operation simplifier class `StmtSimplifier` of the `analysis` module.

Lines 2-5 of Algorithm 5, with the operation transformation F as the main step, are referred to as the control flow splitting algorithm in the implementation. This control flow splitting method produces a complex result, consisting of the Boolean flags introduced by the transformation, the transformed operation and the conjunction of flag formulas. The operation is transformed recursively, supporting arbitrary embedding in compound operations. The introduced Boolean flags and flag constraints are collected continuously during the transformation.

To optimize the control flow splitting algorithm, we only transform operations and sub-operations that contain clock variables or delay operations.

For further optimization, we also implemented a cache for the results. The result of control flow splitting may vary depending on the state, because of the initial operation substitution step, which is necessary for the elimination of loops. Therefore, the cache stores the control flow splitting results identified by the simplified operations obtained in line 1 of Algorithm 5. This prevents running the control flow splitting algorithm multiple times for state-operation pairs producing the same simplified operation. Note that the results of simplifying an operation with different states may produce the same simplified operation, e.g. in conditional operations where the condition evaluates to $true$ or $false$, variables appearing only in the branch removed by the simplification become irrelevant, therefore states that differ only on those variables produce the same simplified operation.

6.1.4 Configuration options

Our solutions provide new configuration options that were also added to the command-line user interface provided by the `xsts-cli` module. The following new command-line parameters were added to the user interface:

- `-combined`: run the combined abstraction algorithm (see section 5.3).
- `-clockstrategy`, the strategy for handling clock variables in the model, with the following possible values:
  - `RAT_ENCODE`: encoding with rational variables (see section 5.1),
  - `CF_SPLIT_ALL`: control flow splitting (see section 5.2), keeping all control flows,

  - CF_SPLIT_FILTERCF: control flow splitting, with filtering out infeasible control flows (see 5.2.3).

- -zonestrategy, the refinement strategy for zones, if the combined abstraction algorithm is used, with the following possible values:
  - BW_ITP: backward interpolation [41],
  - FW_ITP: forward interpolation [41].

6.2 Benchmarks

We evaluated our approaches on two TXSTS models automatically exported from statechart models using Gamma. The first one is a model of a crossroad[2] with traffic lights, a controller, and an interrupted mode that may be triggered by the police, inspired by industrial system models and demonstrating the capabilities of Gamma. The other model is a subsystem of two antivalence checkers and a signaller, taken from a railway-related industrial case study [22].

We used BenchExec[3] to execute measurements on virtual computers, with each task limited to 3 CPU cores, 15 GB memory, and 20 minutes of runtime. By executing the measurements, we aim to answer the following research questions:

1. How does the performance of the proposed verification approaches compare for checking reachability properties?
2. How does the performance of the proposed approaches compare for checking timed reachability, i.e. reachability of states under a given time limit in the analyzed model?

The first question targets the reachability of various properties. Some of the considered properties are safety properties, where deciding reachability is the main objective. In other cases the reachability of the state is known, still we might be interested in a trajectory leading to the given state, as it comes useful in test generation applications.

Even though all analyzed models contain timing, the reachability of states is not always dependent on both timing and data flow, in many cases reachability can be decided without taking timing into consideration. To better assess the future applicability of our solutions, we also examined performance for the timed versions of the properties, as our second research question. To answer this, we introduced a new clock in each model, and checked the same reachability properties as before, but under a given time limit, represented by the newly introduced clock. We set a time limit uniformly for all properties checked on the crossroad model, and a different time limit uniformly for all properties checked on the model of the antivalence checker, based on the values appearing in clock constraints in the models.

6.2.1 Configurations

In the measurements we compare our two main approaches for verifying TXSTS models, with two main configurations of the second approach: RAT_ENCODE, CF_SPLIT_ALL and CF_SPLIT_FILTERCF, as described in 6.1.4.

[2]https://github.com/ftsrg/gamma/tree/master/tutorial
[3]https://github.com/sosy-lab/benchexec

For all mentioned approaches we consider two different abstract domains for the data abstraction that usually perform well on XSTS models:

- EXPL: explicit value abstraction with transition function $T^{\pi}_{e,250}$ (see explicit value abstraction in 2.3.1),
- EXPL_PRED: product domain of explicit value abstraction with $T^{\pi}_{e,250}$ and predicate abstraction with predicate splitting (see predicate abstraction in 2.3.2).

The strategy used for pruning the ARG at the end of the CEGAR refinement step produces quite different runtimes in the results, therefore we run the measurements with two ARG pruning strategies:

- PRUNE_FULL: the ARG is completely discarded,
- PRUNE_LAZY: only a subtree of the ARG is removed, depending on where the abstract path becomes infeasible in the ARG (lazy pruning is discussed in more detail in [25]).

6.2.2 Results and discussion

Table 6.1 shows an overview of our benchmarking results. For each configuration the upper number shows the number of tasks the configuration completed in the given category (checking reachability or timed reachability properties), and the lower number shows the number of cases where the given configuration performed best regarding runtime.

Overall, control flow splitting without infeasible control flow filtering could solve the most tasks: all reachability tasks and 18 out of 30 timed reachability tasks, which is also the maximum of solved timed reachability tasks among all configurations.

We further inspected the results of benchmarking for configurations that performed best. Our main objective is the number of verification tasks a given configuration can solve, therefore we select the configurations with the most solved tasks, and in case of a tie, we choose depending on the number of tasks where the given configuration performed best. We inspected these configurations separately for the categories of reachability and timed reachability. We also made sure to include at least one configuration of both our main approaches: encoding with rational variables and control flow splitting.

In general, configurations using the product domain and full pruning solved the most verification tasks for reachability properties without a given time limit. It is worth noting that although the CF_SPLIT_ALL method with explicit abstraction and full pruning could not check one of the properties, it is the fastest configuration, producing the shortest runtime in 11 out of 30 cases.

Reachability properties with time limits (timed reachability) proved to be more difficult to check. The control flow splitting method with keeping all control flows (CF_SPLIT_ALL) produced the best results, verifying up to 18 properties out of 30, with all configurations outperforming the RAT_ENCODE and CF_SPLIT_FILTERCF methods.

Table 6.2 shows the detailed results produced by the best configurations for reachability without time limits in the properties: the RAT_ENCODE and CF_SPLIT_FILTERCF methods using the EXPL_PRED product domain for data, and full pruning.

Both approaches could verify all properties, with similar runtimes. Since they completed the same verification tasks, we can also compare the average CPU time needed for the two

Method	Parameters	Reachability	Timed reachability	Total tasks
RAT_ENCODE	EXPL, PRUNE_LAZY	26 2	12 1	38 3
	EXPL, PRUNE_FULL	27 6	12 3	39 9
	EXPL_PRED, PRUNE_LAZY	29 0	10 1	39 1
	EXPL_PRED, PRUNE_FULL	30 3	10 3	40 6
CF_SPLIT_ALL	EXPL, PRUNE_LAZY	27 0	16 3	43 3
	EXPL, PRUNE_FULL	29 11	17 1	46 12
	EXPL_PRED, PRUNE_LAZY	30 0	17 1	47 1
	EXPL_PRED, PRUNE_FULL	30 1	18 2	48 3
CF_SPLIT_FILTERCF	EXPL, PRUNE_LAZY	29 0	13 1	42 1
	EXPL, PRUNE_FULL	29 3	13 3	42 6
	EXPL_PRED, PRUNE_LAZY	30 0	14 0	44 0
	EXPL_PRED, PRUNE_FULL	30 4	15 1	45 5
	Total number of tasks:	30	30	60

Table 6.1: Number of solved tasks and number of fastest results by each configuration for reachability and timed reachability tasks

methods to complete a task, where encoding with rational variables slightly outperforms control flow splitting, with an average runtime of 7.477 seconds, against the average of 11.089 seconds by control flow splitting.

This result indicates that the efficient time abstraction provided by the zone domain in the combined algorithm does not improve the result significantly for the properties analyzed here. A probable cause of this is that the reachability of the states given in the properties is not dependent of time, and therefore control flow splitting unnecessarily introduces some overhead, while the simplicity of the encoding method turns into an advantage in this case.

Table 6.3 shows detailed results produced by the best configuration of both main approaches for timed reachability properties: the encoding method with explicit value abstraction for data, and control flow splitting with keeping all control flows, using the product domain for data.

Although both are the best among configurations of one of the main approaches, these two configurations produce quite different results, which means that extending the models and properties to check reachability under a given time limit indeed necessitated checking behavior that is dependent on both data and timing. In many cases, timing could not be handled by rational variables, pointing out the need for handling time in a more refined way.

Control flow splitting enabled the verification of 50 percent more properties than rational variable encoding. Moreover, control flow splitting could verify each one of these properties in less than 3 minutes, as opposed to rational variable encoding exceeding the 20 minutes time limit.

The overall results show that the proposed approaches can provide a solution for verifying real-time software-intensive systems. Encoding TXSTS models in the XSTS formalism using rational variables is most suitable for verifying less time-dependent reachability properties and generating trajectories to given states, while control flow splitting provides a method for handling more complex verification tasks.

Property	CPU time in seconds by `RAT_ENCODE` (`EXPL_PRED`, `PRUNE_FULL`)	CPU time in seconds by `CF_SPLIT_FILTERCF` (`EXPL_PRED`, `PRUNE_FULL`)
crossroad-1	1.601	2.142
crossroad-2	1.866	1.789
crossroad-3	20.402	19.545
crossroad-4	2.861	2.275
crossroad-5	6.742	6.342
crossroad-6	1.815	1.514
crossroad-7	1.868	2.034
crossroad-8	1.724	1.582
crossroad-9	1.859	1.880
crossroad-10	2.436	2.422
crossroad-11	5.416	2.972
crossroad-12	3.156	3.040
crossroad-13	1.778	1.879
crossroad-14	2.541	3.696
crossroad-15	2.736	1.802
crossroad-16	2.183	1.874
crossroad-17	2.235	1.955
crossroad-safety	10.620	9.269
antivalence-1	2.396	2.064
antivalence-2	6.233	7.642
antivalence-3	3.464	5.773
antivalence-4	19.198	20.281
antivalence-5	1.804	1.813
antivalence-6	3.224	2.726
antivalence-7	3.932	7.850
antivalence-8	2.119	2.174
antivalence-9	4.028	4.910
antivalence-10	6.931	6.732
antivalence-safety-1	29.734	182.484
antivalence-safety-2	67.419	20.210
Average runtime in seconds:	**7.477**	**11.089**

Table 6.2: CPU time in seconds for *reachability* properties by the best configuration of both main approaches for reachability

Property	CPU time in seconds by `RAT_ENCODE` (`EXPL`, `PRUNE_FULL`)	CPU time in seconds by `CF_SPLIT_ALL` (`EXPL_PRED`, `PRUNE_FULL`)
crossroad-timed-1	1.697	1.840
crossroad-timed-2	**timeout**	**126.777**
crossroad-timed-3	**timeout**	**151.514**
crossroad-timed-4	**timeout**	**136.996**
crossroad-timed-5	**timeout**	**153.888**
crossroad-timed-6	1.910	1.906
crossroad-timed-7	timeout	timeout
crossroad-timed-8	1.789	2.243
crossroad-timed-9	1.824	2.140
crossroad-timed-10	timeout	timeout
crossroad-timed-11	timeout	timeout
crossroad-timed-12	**timeout**	**4.162**
crossroad-timed-13	1.999	2.157
crossroad-timed-14	2.099	3.852
crossroad-timed-15	2.060	2.686
crossroad-timed-16	1.946	2.324
crossroad-timed-17	1.664	2.236
crossroad-timed-safety	timeout	timeout
antivalence-timed-1	timeout	timeout
antivalence-timed-2	timeout	timeout
antivalence-timed-3	timeout	timeout
antivalence-timed-4	timeout	timeout
antivalence-timed-5	2.191	2.287
antivalence-timed-6	3.255	6.315
antivalence-timed-7	4.724	23.083
antivalence-timed-8	timeout	timeout
antivalence-timed-9	timeout	timeout
antivalence-timed-10	timeout	timeout
antivalence-timed-safety-1	timeout	timeout
antivalence-timed-safety-2	**timeout**	**111.407**
Number of solved verification tasks:	12	18

Table 6.3: CPU time in seconds for *timed reachability* properties by the best configuration of both main approaches for timed reachability

Chapter 7

Conclusion

The verification of timed engineering models used for the modeling of real-time software-intensive critical systems is a challenging task. Various techniques are introduced in the literature to verify either software systems or timed automata models, and some approaches can also handle engineering models with limited time-dependent properties.

Our goal was to introduce a formalism to support the formal description of high-level engineering models of software-intensive real-time systems. In addition, efficient algorithms are needed to solve the verification problem. We devised a combined algorithm by integrating CEGAR and lazy abstraction, and a novel iterative exploration is used to provide a step-wise refinement of the verification problem.

The theoretical contributions of the paper:

- We defined an extension to the XSTS formalism used as an intermediate formal model in the Gamma statechart-based modeling toolchain to represent timed systems. The formalism can serve as a general intermediate language between high-level engineering modeling languages and low level verification tools.
- We defined a mapping from timed XSTS models to XSTS models to enable utilizing existing verification tools for the analysis.
- We developed a novel algorithm to iteratively explore complex control flows in timed XSTS models caused by component communication and hierarchical modeling.
- We adapted an abstraction-based model checking approach for the verification of timed XSTS models. The new methods are able to handle the data and time aspects of engineering models efficiently.

The engineering contributions of the paper:

- We integrated the timed XSTS formalism in the open-source THETA verification framework.
- We implemented the proposed mapping as a transformation of timed XSTS to XSTS in the THETA framework.
- We implemented the proposed control flow splitting algorithm in THETA and integrated it into the existing combined verification algorithm.

7.1 Future work

In the future, we plan to further evaluate the approach on more engineering models from other domains. In addition, the proposed integrated approach is a good candidate for temporal logic model checking. This could further help engineers to analyze even complex temporal properties. Our fine-grained abstraction techniques can also be used to analyze parameterized timed properties; in the future we plan to explore this research direction as well.

Bibliography

[1] Thomas Ball, Andreas Podelski, and Sriram K Rajamani. Boolean and cartesian abstraction for model checking c programs. In *TACAS*, pages 268–283. Springer, 2001.

[2] Johan Bengtsson and Wang Yi. *Timed Automata: Semantics, Algorithms and Tools*, pages 87–124. Springer, 2004. DOI: `10.1007/978-3-540-27755-2\_3`.

[3] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, pages 184–190. Springer, 2011.

[4] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *FASE*, pages 146–162. Springer, 2013.

[5] Patricia Bouyer, Paul Gastin, Frédéric Herbreteau, Ocan Sankur, and B. Srivathsan. Zone-based verification of timed automata: Extrapolations, simulations and what next? In *FORMATS*, volume 13465 of *LNCS*, pages 16–42. Springer, 2022. DOI: `10.1007/978-3-031-15839-1\_2`.

[6] Patricia Bouyer, Paul Gastin, Frédéric Herbreteau, Ocan Sankur, and B Srivathsan. Zone-based verification of timed automata: extrapolations, simulations and what next? In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 16–42. Springer, 2022.

[7] Aaron R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011. DOI: `10.1007/978-3-642-18275-4\_7`.

[8] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.

[9] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 334–342, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08867-9.

[10] Zuxi Chen, Zhongwei Xu, Junwei Du, Meng Mei, and Jing Guo. Efficient encoding for bounded model checking of timed automata. *IEEJ Tran. Electrical Electronic Eng.*, 12(5):710–720, 2017. DOI: `10.1002/tee.22457`.

[11] Alessandro Cimatti, Alberto Griggio, Enrico Magnago, Marco Roveri, and Stefano Tonetta. Extending nuxmv with timed transition systems and timed temporal properties. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 376–386, Cham, 2019. Springer International Publishing. ISBN 978-3-030-25540-4.

[12] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169. Springer, 2000.

[13] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19:7–34, 2001.

[14] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5): 1512–1542, 1994.

[15] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. *Introduction to Model Checking*, pages 1–26. Springer, Cham, 2018. DOI: `10.1007/978-3-319-10575-8\_1`.

[16] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[17] Dóra Cziborová. Abstraction-based model checking for real-time software-intensive system models. Scientific students' association report, Budapest University of Technology and Economics, 2023.

[18] Dóra Cziborová and Béla Ákos Vizi. Abstraction-based model checking techniques for real-time systems. Scientific students' association report, Budapest University of Technology and Economics, 2022.

[19] Dóra Cziborová. Generalizing lazy abstraction refinement algorithms with partial orders. Bachelor's thesis, Budapest University of Technology and Economics, 2021.

[20] Henning Dierks, Sebastian Kupferschmid, and Kim Guldstrand Larsen. Automatic abstraction refinement for timed automata. In *FORMATS*, volume 4763 of *LNCS*, pages 114–129. Springer, 2007. DOI: `10.1007/978-3-540-75454-1\_10`.

[21] Bruno Dutertre and Maria Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *FORMATS*, volume 3253 of *LNCS*, pages 199–214. Springer, 2004. DOI: `10.1007/978-3-540-30206-3\_15`.

[22] Bence Graics, Vince Molnár, and István Majzik. Integration test generation for state-based components in the gamma framework.

[23] Bence Graics, Vince Molnár, András Vörös, István Majzik, and Dániel Varró. Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Softw. Syst. Model.*, 19(6):1483–1517, 2020. DOI: `10.1007/s10270-020-00806-5`.

[24] Bence Graics, Vince Molnár, András Vörös, István Majzik, and Dániel Varró. Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Software and Systems Modeling*, 19:1483–1517, 2020.

[25] Ákos Hajdu and Zoltán Micskei. Efficient strategies for CEGAR-based model checking. *J. Autom. Reasoning*, 64(6):1051–1091, 2020.

[26] Ákos Hajdu, Tamás Tóth, András Vörös, and István Majzik. A configurable cegar framework with interpolation-based refinements. In *Formal Techniques for Distributed Objects, Components, and Systems: 36th IFIP WG 6.1 International Conference,*

FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings 36, pages 158–174. Springer, 2016.

[27] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[28] Frédéric Herbreteau, B. Srivathsan, and Igor Walukiewicz. Lazy abstractions for timed automata. In *CAV*, volume 8044 of *LNCS*, pages 990–1005. Springer, 2013. DOI: `10.1007/978-3-642-39799-8\_71`.

[29] Tobias Isenberg and Heike Wehrheim. Timed automata verification via IC3 with zones. In *ICFEM*, volume 8829 of *LNCS*, pages 203–218. Springer, 2014. DOI: `10.1007/978-3-319-11737-9\_14`.

[30] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. *Predicate Abstraction for Program Verification*, pages 447–491. Springer, 2018. DOI: `10.1007/978-3-319-10575-8\_15`.

[31] Martin Leucker, Grigory Markin, and Martin R. Neuhäußer. A new refinement strategy for CEGAR-based industrial model checking. In *Hardware and Software: Verification and Testing*, pages 155–170. Springer, 2015.

[32] K. L. McMillan. Applications of Craig interpolants in model checking. In *TACAS*, pages 1–12. Springer, 2005.

[33] Kenneth L McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136. Springer, 2006.

[34] M. Oliver Möller, Harald Rueß, and Maria Sorea. Predicate abstraction for dense real-time system. In *Theory and Practice of Timed Systems*, volume 65 of *Elec. Notes Theor. Comput. Sci.*, pages 218–237. Elsevier, 2002. DOI: `10.1016/S1571-0661(04)80478-X`.

[35] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The gamma statechart composition framework: design, verification and code generation for component-based reactive systems. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 113–116, 2018.

[36] Milán Mondok. Extended symbolic transition systems: an intermediate language for the formal verification of engineering models. Scientific students' association report, Budapest University of Technology and Economics, 2020.

[37] Milán Mondok. Efficient abstraction-based model checking using domain-specific information. Scientific students' association report, Budapest University of Technology and Economics, 2021.

[38] Victor Roussanaly, Ocan Sankur, and Nicolas Markey. Abstraction refinement algorithms for timed automata. In *CAV*, volume 11561 of *LNCS*, pages 22–40. Springer, 2019. DOI: `10.1007/978-3-030-25540-4\_2`.

[39] Ocan Sankur. Timed automata verification and synthesis via finite automata learning. In *TACAS*, volume 13994 of *LNCS*, pages 329–349. Springer, 2023. DOI: `10.1007/978-3-031-30820-8\_21`.

[40] Maria Sorea. Bounded model checking for timed automata. In *MTCS@CONCUR*, volume 68 of *Elec. Notes Theor. Comput. Sci.*, pages 116–134. Elsevier, 2002. DOI: `10.1016/S1571-0661(04)80523-1`.

[41] Tamás Tóth and István Majzik. Lazy reachability checking for timed automata using interpolants. In *FORMATS*, volume 10419 of *LNCS*, pages 264–280. Springer, 2017. DOI: `10.1007/978-3-319-65765-3\_15`.

[42] Tamás Tóth, András Vörös, and István Majzik. K-induction based verification of real-time safety critical systems. In *DEPCOS*, volume 224 of *Advances in Intelligent Systems and Computing*, pages 469–478. Springer, 2013. DOI: `10.1007/978-3-319-00945-2\_43`.

[43] Tamás Tóth and István Majzik. Configurable verification of timed automata with discrete variables. *Acta Informatica*, 2020. DOI: `10.1007/s00236-020-00393-4`.

[44] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: A framework for abstraction refinement-based model checking. In *FMCAD*, pages 176–179, 2017. DOI: `10.23919/FMCAD.2017.8102257`.

[45] Yakir Vizel and Orna Grumberg. Interpolation-sequence based model checking. In *FMCAD*, pages 1–8. IEEE, 2009. DOI: `10.1109/FMCAD.2009.5351148`.