Acta Cybernetica — *online–first paper version* — pages 1–49.

Sequential and Parallel Implementation of Lazy Abstraction Strategies*

Mihály Dobos-Kovács[ab], Dóra Cziborová[ac], and András Vörös[ad]

Abstract

Verifying timed systems is essential in safety-critical applications and poses significant challenges due to the huge number of possible behaviors. Model checking is a powerful tool of verification exploring the possible behaviors of systems, but it struggles with industrial applications, particularly when time is involved. Abstraction-based techniques are often used for the model checking of complex systems, as they can simplify the representation of the state space. Lazy abstraction is such a technique that incrementally adjusts (refines) abstractions only as needed, thus offering a promising solution by balancing precision and efficiency.

This paper explores both sequential and parallel implementations of lazy abstraction strategies for timed systems. The sequential approach generalizes former implementations, while parallel implementations aim to leverage modern multi-core architectures for improved scalability and efficiency. Through empirical evaluation of benchmark systems, the research compares the various approaches, identifying key factors influencing parallelization effectiveness. The findings offer insights into lazy abstraction and opens new directions to further enhance the verification of complex timed systems.

Keywords: formal verification, model checking, lazy abstraction

1 Introduction

As the complexity of real-time systems is growing, ensuring their correctness has become increasingly challenging. These systems, characterized by the necessity to

*Supported by the **ÚNKP-23-3-I-BME-337** New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund. The project was supported by the Doctoral Excellence Fellowship Programme (DCEP) is funded by the National Research Development and Innovation Fund of the Ministry of Culture and Innovation and the Budapest University of Technology and Economics, under a grant aggreement with the National Research, Development and Innovation Office.

[a]Department of Measurement and Information Systems, Budapest University of Technology and Economics, Hungary

[b]E-mail: mdobosko@mit.bme.hu, ORCID: 0000-0002-0064-2965

[c]E-mail: cziborova@mit.bme.hu, ORCID: 0009-0003-5624-1372

[d]E-mail: vori@mit.bme.hu, ORCID: 0000-0001-7617-3563

DOI: 10.14232/actacyb.311397

meet strict timing constraints, are pervasive in critical domains such as aerospace, automotive, telecommunications, and industrial automation. A failure in these systems due to a missed deadline or incorrect timing can lead to catastrophic outcomes, making their verification a matter of utmost importance.

Model checking is a formal verification technique that systematically explores the state space of a system to verify whether the system satisfies certain properties, including timing constraints. However, model checking faces the state space explosion problem, where the number of system states grows exponentially with the complexity of the system, exacerbated further in timed systems due to the continuous nature of time.

To mitigate this problem, abstraction techniques have been developed, which approximate the system model by focusing on essential details relevant to the properties being verified. Lazy abstraction is one such technique, particularly effective for timed systems, as it incrementally refines the abstraction only when necessary, which is crucial for handling the intricate timing constraints of real-time systems.

This paper explores both sequential and parallel implementations of lazy abstraction strategies. Lazy abstraction is a sequential algorithm that navigates the set of reachable states using abstraction, applying refinement only when necessary. We provide a flexible framework that is designed to be highly versatile and compatible with any abstract domain that meets some special properties, allowing the integration of various abstractions and refinement strategies.

Given the computational demands of formal verification, the implementation of lazy abstraction strategies could benefit from parallelization. With multi-core processors, modern computing architectures offer substantial opportunities to parallelize verification: our goal is to exploit the available computing power in a parallel implementation of the lazy algorithm.

First, in Chapter 2, we provide a detailed overview of model checking and the formal model used later on, then Chapter 3 introduces the relevant related work. In Chapter 4, we propose a generic lazy abstraction framework summarizing the sequential implementations of lazy abstraction. Chapter 5 explores the parallel implementation of lazy abstraction strategies, discussing the approach of parallelization and the trade-offs identified. Finally, Chapter 6 offers a comparative analysis of the sequential and parallel implementations, highlighting key findings and implications for the verification of timed systems.

2 Background

2.1 Model Checking

Model checking (Figure 1) [13] is a formal verification technique to prove that certain properties (called requirements) hold in a model. The properties under investigation are often categorized into two main groups: safety properties (an unsafe state of the system is never reached) and liveness properties (the desired state is always reached eventually).

The inputs of a model checking algorithm are the model and the requirements, both formally specified. The algorithm then outputs whether the given model satisfies the given requirements or, in case the model does not satisfy a requirement, it may also output a corresponding counterexample, leading to some erroneous states in the model.

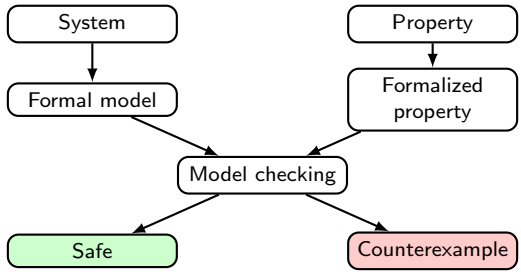


Figure 1: Model checking

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

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

Many reachability analysis techniques are based on *abstraction*, which applies an *over-approximation* of the reachable states. The over-approximated state space may also contain behavior that is not present in the original model. Still, more importantly, it contains all possible behaviors of the model, while the state space can be represented in a more compact way. Abstraction, therefore, ensures that if a state is unreachable in the abstract model, then neither is it reachable in the original model. However, a model checking algorithm may find counterexamples in the abstract model that are not present in the original one, and thus, false positives may occur when the abstraction is too coarse.

2.2 Timed Automata

Timed automata [1] provide a graph-based modeling formalism, where a finite set of nodes represent locations and a finite set of labeled directed edges represent transitions. Time is modeled using *clock variables*, while the use of *data variables* is also allowed.

Clock variables are continuous, non-negative variables that represent the passage of time. All clock variables are initialized at zero, are always incremented

equally within the model, and can be reset individually. Although clocks are continuous variables, most timed analysis techniques consider resets and comparisons with integers only. However, this is barely a limitation, as models can still use resets and comparisons with rational constants, in that case they need to be multiplied by the least common multiple of their denominators as a preprocessing step [1].

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. *Clock constraints* are formulas 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 $k \in \mathbb{Z}$.

Let V_D be a set of data variables with 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 2.1. *A timed automaton is a tuple* $\mathcal{A} = \langle \mathcal{L}, \mathcal{T}, V_C, V_D, l^0, val_D^0 \rangle$*, where*

- $\mathcal{L}$ *is a finite set of* locations*;*
- $\mathcal{T} \subset \mathcal{L} \times \mathcal{O} \times \mathcal{L}$ *is a set of* transitions *labeled with operations* $op \in \mathcal{O}$*;*
- 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.*

2.2.1 Semantics of Timed Automata

The semantics of timed automata is defined by a transition system, with a set of states $\mathcal{S}$. A state is a tuple $\langle l, \langle val_D, val_C \rangle \rangle \in \mathcal{S}$, where $l \in \mathcal{L}$, $val_D \in Val_D$ and $val_C \in Val_C$.

The *initial states* form a set $\mathcal{S}_0 = \{\langle l^0, \langle val_D^0, val_C^0 \rangle \rangle \mid val_C^0 = \{c \mapsto \delta \mid c \in V_C\}, \delta \in \mathbb{R}_{\geq 0}\}$ with data variables initialized to val_D^0 and clock variables to some initial delay $\delta \geq 0$.

Let $[\![op]\!]\langle val_D, val_C \rangle$ denote the set of possible results of applying the operation $op \in \mathcal{O}$ on a pair $\langle val_D, val_C \rangle$, where $val_D \in Val_D$ and $val_C \in Val_C$. The set $[\![op]\!]\langle val_D, val_C \rangle$ is defined as follows:

- If *op* is a *data guard* of the form $[\varphi]$, where φ is a predicate over V_D, then $[\![op]\!]\langle val_D, val_C \rangle = \{\langle val_D, val_C \rangle\}$ if val_D satisfies φ, otherwise we have $[\![op]\!]\langle val_D, val_C \rangle = \emptyset$.
- If *op* is a *clock guard* of the form $[constr]$, where *constr* is a clock constraint over V_C, then $[\![op]\!]\langle val_D, val_C \rangle = \{\langle val_D, val_C \rangle\}$ if val_C satisfies *constr*, otherwise $[\![op]\!]\langle val_D, val_C \rangle = \emptyset$.

- If op is a *data assignment* $x := \varphi$, where $x \in V_D$ with domain D_x, φ is an expression over V_D such that $\varphi(val_D) \in D_x$, then $[\![op]\!]\langle val_D, val_C\rangle = \{\langle val'_D, val_C\rangle\}$ such that $val'_D(x) = \varphi(val_D)$ and $val'_D(x') = val_D(x')$ for each $x' \in V_D \backslash \{x\}$.

- If op is a *clock reset* $c := n$, where $c \in V_C$, $n \in \mathbb{N}_0$, then $[\![op]\!]\langle val_D, val_C\rangle = \{\langle val_D, val'_C\rangle\}$ such that $val'_C(c) = n$ and $val'_C(c') = val_C(c')$ for each $c' \in V_C \backslash \{c\}$.

- If op is a *compound operation* of the form $(op_1; op_2)$, where $op_1, op_2 \in \mathcal{O}$, then $[\![op]\!]\langle val_D, val_C\rangle = \{\langle val''_D, val''_C\rangle \mid \langle val''_D, val''_C\rangle \in [\![op_2]\!]\langle val'_D, val'_C\rangle, \langle val'_D, val'_C\rangle \in [\![op_1]\!]\langle val_D, val_C\rangle\}$. If the result of the first operation op_1 is empty, e.g., op_1 is a data guard not satisfied by val_D, then $\langle val'_D, val'_C\rangle$ and consequently $\langle val''_D, val''_C\rangle$ cannot exist, therefore the result of the compound operation will also be empty.

There are two kinds of transitions in the transition system representing the semantics of the timed automaton:

- An *action transition* $\langle l, \langle val_D, val_C\rangle\rangle \xrightarrow{op} \langle l', \langle val'_D, val'_C\rangle\rangle$ with an operation $op \in \mathcal{O}$ is enabled if and only if there exists an edge $\langle l, op, l'\rangle \in \mathcal{E}$ and $\langle val'_D, val'_C\rangle \in [\![op]\!]\langle val_D, val_C\rangle$.

- A *delay transition* $\langle l, \langle val_D, val_C\rangle\rangle \xrightarrow{\delta} \langle l', \langle val'_D, val'_C\rangle\rangle$ with delay $\delta \in \mathbb{R}_{\geq 0}$ is enabled if and only if $l' = l$, $val'_D = val_D$ and $val'_C(c) = val_C(c) + \delta$ for each $c \in V_C$.

The postcondition of a state $s \in \mathcal{S}$ with respect to a transition $t = \langle l, op, l'\rangle \in \mathcal{T}$ is given by the *concrete post-image* operator: $post_t(s) = \{s'' \mid \exists s' \colon s \xrightarrow{op} s' \text{ is enabled}, s' \xrightarrow{\delta} s'' \text{ is enabled}\}$. Conversely, the existential precondition of a state $s \in \mathcal{S}$ with respect to a transition $t \in \mathcal{T}$ is given by the *concrete pre-image* operator: $pre_t(s') = \{s \mid s' \in post_t(s)\}$.

A *run* of a timed automaton is a finite sequence of states $\sigma = \langle l^0, \langle val^0_D, val^0_C\rangle\rangle \xrightarrow{t_1} \langle l^1, \langle val^1_D, val^1_C\rangle\rangle \xrightarrow{t_2} \cdots \xrightarrow{t_n} \langle l^n, \langle val^n_D, val^n_C\rangle\rangle$, where $\langle l^0, \langle val^0_D, val^0_C\rangle\rangle \in \mathcal{S}_0$ and $\langle l^i, \langle val^i_D, val^i_C\rangle\rangle \in post_{t_i}(\langle l^{i-1}, \langle val^{i-1}_D, val^{i-1}_C\rangle\rangle)$ for all $1 \leq i \leq n$. A state $\langle l, \langle val_D, val_C\rangle\rangle \in \mathcal{S}$ is *reachable* if and only if a run exists where $l^n = l$, $val^n_D = val_D$ and $val^n_C = val_C$.

3 Related Work

Lazy abstraction was introduced in [21], as a model checking approach that integrates the building and refinement phases of abstraction, with different parts of the abstract model exhibiting different degrees of precision. Craig interpolant-based abstraction refinement is introduced into the lazy abstraction paradigm in [27], serving as the basis for the here-described lazy abstraction building and abstraction

refinement techniques. Both [21] and [27] work over formulas, which are continuously increasing in size as the state-space grows. Abstract domains to efficiently summarize the state space have been introduced with zone abstraction for timed automata.

For the verification of timed automata, zone-based abstractions are used. Lazy abstraction was adapted to the reachability analysis of timed automata in [5]. In addition to zones, it also distinguishes LU-bounds, i.e., maximal lower and upper bounds, to obtain a coarser abstraction. In [22], an algorithm was proposed that updates LU-bounds dynamically. A similar lazy search algorithm was presented in [35] that uses an abstraction technique that extends zones with difference-bound constraint sets to reduce the state space.

Our work builds directly on [33] and [34]. In [33], a lazy method was proposed for location reachability checking of timed automata with two refinement strategies that use zone interpolation and backward propagation. For timed automata with discrete variables, an algorithmic framework using the lazy method of [33] was presented in [34] that allows the combination of abstractions for clocks and discrete variables. The framework provides a lazy abstraction algorithm for checking location reachability. Nevertheless, the inventory of abstract domains for lazy analysis remained limited.

In this paper, we propose a generalization of the framework presented in [34]. In [34], the abstraction refinement strategies are provided specifically for the direct product of zone abstraction and explicit value abstraction. In contrast, we propose both abstraction building and refinement strategies operate over *generic abstract domains*, allowing greater flexibility in the use of various abstractions. Furthermore, we identify the operations of lazy abstraction and their properties in a domain-independent approach, while directly addressing the possibility of handling a set of variables by multiple different but compatible abstractions. Moreover, we provide guidance on designing efficient abstract domains for lazy abstraction.

Using parallel algorithms for verification tasks is an established idea in the literature. Ideas to conduct model checking in parallel were discussed in the early 2000s [19, 3]. Parallel versions of explicit model checking tend to be highly distributed [10, 25, 24], using computer clusters to enumerate the state space, but other approaches are known as well: there are explicit model checking variants that utilize the GPU for massively parallel processing [15, 37] and there is even a variant that employs hardware acceleration via an FPGA [18].

Model checking algorithms often use abstraction to tackle state-space explosion. There are search-based techniques utilizing abstraction that were successfully parallelized [12], and for computer software, both bounded model checking [23] and the algorithm of counterexample-guided abstraction refinement (CEGAR) [31] were extended for multi-processor environments. Modern approaches tend to use portfolios of algorithms to tackle verification tasks [7, 9], and parallelization can be achieved by running different algorithms in parallel.

Our paper focuses on the reachability analysis of timed systems. Parallel verification of timed systems was previously achieved by LTL-based algorithms [16, 26, 4] and by CEGAR as well [28]. The latter algorithms uses a cluster of computers to

calculate counterexamples in parallel, and a master node to consolidate the results. In contrast, our proposed parallel algorithm works in a single computer using shared memory.

4 Lazy Abstraction Framework

We propose a generic, configurable lazy abstraction framework that works with any abstract domain that adheres to the properties given later in this chapter. The framework is independent of the modeling formalism used; however, we also present examples through some specific domains that one may use to verify timed automaton models.

Our approach explores the state space by building a graph-like representation of reachable states with additional information at each node. The exploration follows the general idea of lazy abstraction depicted in Figure 2.

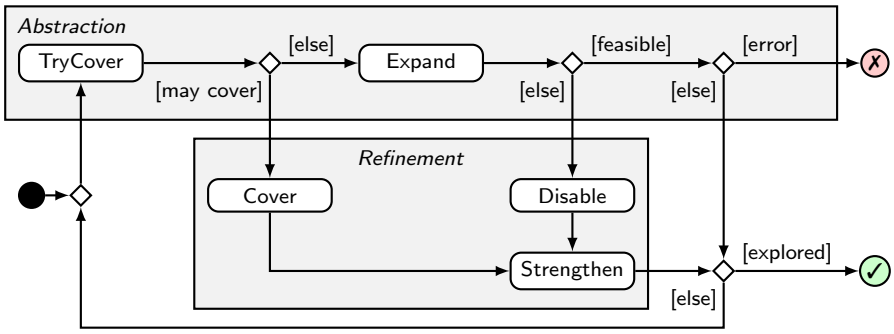


Figure 2: Overview of lazy abstraction

Lazy abstraction first tries to *cover* the states it encounters. If a state can be covered, then its node needs to be *refined* to maintain correctness. If it is not possible to cover a state, then the state space needs to be explored further from that state, i.e., the node is *expanded*. If an infeasible transition is encountered while expanding, then the transition needs to be *disabled* from the given node by refinement. In both cases, refinement works by *strengthening* some states, which helps eliminate infeasible paths.

When an error state is encountered on a path during the exploration, then that path is already a proof of incorrect behavior, without requiring further checks of feasibility. On the other hand, if there is nothing more left to explore, then the resulting graph built by the algorithm proves the correctness of the model.

4.1 Abstract Domains

Abstractions over-approximate concrete states with a tractable representation called *abstract states* to support verification. We define abstractions through *abstract do-*

mains that in addition to the abstract states also comprise the operators closely associated with them.

Definition 4.1. *An abstract domain is a tuple* $\mathbf{D} = \langle \mathcal{S}, \mathcal{D}, \sqsubseteq, \gamma, post' \rangle$*, where*

- $\mathcal{S}$ *is the set of* concrete states*;*
- $\mathcal{D}$ *is the set of* abstract states*;*
- $\sqsubseteq \subseteq \mathcal{D} \times \mathcal{D}$ *is a* preorder*, i.e. it is a reflexive and transitive binary relation;*
- $\gamma \colon \mathcal{D} \to 2^{\mathcal{S}}$ *is the* concretization function *that maps an abstract label to the corresponding set of concrete states, such that* $d_1 \sqsubseteq d_2$ *implies* $\gamma(d_1) \subseteq \gamma(d_2)$ *for each* $d_1, d_2 \in \mathcal{D}$*,*
- $post' \colon \mathcal{D} \times \mathcal{T} \to 2^{\mathcal{D}}$ *is the* abstract post-image *operator that maps an abstract state* $d \in \mathcal{D}$ *to its successor states with respect to a transition* $t \in \mathcal{T}$*, such that* $\bigcup_{s \in \gamma(d)} post_t(s) \subseteq \bigcup_{d' \in post'_t(d)} \gamma(d')$ *for each* $d \in \mathcal{D}$ *and each* $t \in \mathcal{T}$*, where* $post'_t(d)$ *is the shortened notation for* $post'(d, t)$*. We also require* $\gamma(d') \neq \emptyset$ *for all* $d' \in post'_t(d)$*,* $d \in \mathcal{D}$*,* $t \in \mathcal{T}$*.*

Henzinger et al. [21] and McMillan [27] proposed lazy abstraction with *predicates* as abstract states. In the following, we show some other examples of abstract domains that are useful for the verification of timed automata in the lazy abstraction framework.

4.1.1 Identity Abstraction

We may encounter problems where we do not wish to use any abstraction. However, the presented lazy abstraction framework works over abstract domains. *Identity abstraction* represents the concrete domain as an abstract domain without losing any information about the concrete states.

Identity abstraction over a set of states $\mathcal{S}$ is the abstract domain $\mathbf{Id}(\mathcal{S}) = \langle \mathcal{S}, \mathcal{S}, =_{\mathcal{S}}, id_{\mathcal{S}}, post \rangle$, i.e., the set of abstract states is the same as the set of concrete states, the preorder relation is the equivalence relation on $\mathcal{S}$, the conceretization function is the identity function on $\mathcal{S}$, and the abstract post-image operator is the concrete post-image operator.

4.1.2 Explicit Value Abstraction

In *explicit value abstraction* [8], the values of some subset of variables $V' \subseteq V$ are explicitly tracked, while the rest of the variables $V \backslash V'$ may take any value.

The abstract states are *partial valuations* that map variables in V' to values 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 $V' = \{x_1\}$ is a partial valuation $pval = \{x_1 \mapsto 1\}$.

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 $x \in V'$. 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 total valuations val over V such that $val(x) = pval(x)$ for all variables $x \in V'$. 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 post-image operator $post'_E$ assigns values to variables in V' according to the transition $t \in \mathcal{T}$ where it can be evaluated on V', and assigns $\top$ to all other variables in V'. E.g., if $V' = \{x_1\}$, then $post'_E(\{x_1 \mapsto 1\}, x_1 := x_1 + 1) = \{\{x_1 \mapsto 2\}\}$ and $post'_E(\{x_1 \mapsto 1\}, x_1 := x_2) = \{\{x_1 \mapsto \top\}\}$.

4.1.3 Zone Abstraction

Zone abstraction [1] is used for capturing time information in a compact representation. 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.

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, e.g., for a set $V_C = \{c\}$, $\gamma_{\mathcal{Z}}((c \leq 4)) = \{val_C \mid val_C(c) \geq 0 \wedge val_C(c) \leq 4\}$.

The abstract post-image operator $post'_{\mathcal{Z}}$ yields exactly one zone, which is the exact successor for the given zone with respect to the given transition. Therefore, if $post'_{\mathcal{Z}}(z, t) = \{z'\}$ for some $z \in \mathcal{Z}$ and $t \in \mathcal{T}$, then $\bigcup_{val_C \in \gamma_{\mathcal{Z}}(z)} post_t(val_C) = \gamma_{\mathcal{Z}}(z')$. A possible implementation of $post'_{\mathcal{Z}}$ using DBMs is presented in [6].

Difference Bound Matrices Zones contain clock constraints of the form $c_1 \sim n$ or $c_1 - c_2 \sim n$ where $c_1, c_2 \in V_C$, $n \in \mathbb{N}$ and $\sim \in \{<, \leq, =, \geq, >\}$. By introducing a reference clock c_0 with constant zero value, all clock constraints can be written uniformly as $c_1 - c_2 \prec n$ where $c_1, c_2 \in V_C \cup \{c_0\}$, $n \in \mathbb{Z}$ and $\prec \in \{<, \leq\}$. This enables the representation of any zone as a *difference bound matrix* (DBM) [1]. DBMs enable the effective representation of zones and the efficient execution of time operations used in reachability analysis.

Definition 4.2. *For clock variables $c_1, c_2, ..., c_k$, a difference bound matrix (DBM) is a square matrix $\mathbb{D}$ of dimension $(k + 1) \times (k + 1)$ such that an element of this matrix is either $\mathbb{D}_{i,j} = (n, \prec)$ where $n \in \mathbb{Z}$ and $\prec \in \{<, \leq\}$, representing the clock constraint $c_i - c_j \prec n$, or $\mathbb{D}_{i,j} = \infty$, indicating the absence of a bound.*

4.1.4 Product Abstraction

For systems that contain variables of different kinds (e.g., a timed automaton with data and clock variables), there may not exist a suitable simple abstraction that is applicable to all kinds of variables and handles all of them well. *Product abstraction* combines multiple abstractions and handles them as a single abstract domain.

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

- $\langle d_{1,1}, d_{1,2} \rangle \sqsubseteq_\times \langle d_{2,1}, d_{2,2} \rangle$ if and only if $d_{1,1} \sqsubseteq_1 d_{2,1}$ and $d_{1,2} \sqsubseteq_2 d_{2,2}$;
- $\gamma_\times(\langle d_1, d_1 \rangle) = \gamma_1(d_1) \times \gamma_2(d_2)$;
- $post'_\times(\langle d_1, d_2 \rangle, t) = post'_1(d_1, t) \times post'_2(d_2, t)$.

4.1.5 Location Abstraction

In many cases, the goal of reachability analysis is checking the reachability of an error location. Locations should always be explicitly tracked, as the sets of enabled transitions are dependent on the location. *Location abstraction* provides a uniform representation of locations and states of an automaton, so that the algorithms can work over a generic abstract domain without having to handle locations explicitly.

Location abstraction tracks locations in addition to abstract states of some other abstract domain. With a set of locations $\mathcal{L}$ and an abstract domain $\mathbf{D}$, location abstraction can be considered as a special form of product abstraction: $\mathbf{Loc}(\mathcal{L}, \mathbf{D}) = \mathbf{Prod}(\mathbf{Id}(\mathcal{L}), \mathbf{D})$.

4.2 Reachability Analysis

Our algorithm generalizes the abstraction-based lazy algorithm presented in [34]. Reachability analysis of a target state is performed by dynamically constructing an *abstract reachability graph* (ARG) of reachable states.

4.2.1 Abstract Reachability Graph for Lazy Abstraction

The generic algorithm labels the nodes of the ARG by labels from two generic abstract domains. Let $\mathbf{D}_c = \langle \mathcal{S}, \mathcal{D}_c, \sqsubseteq_c, \gamma_c, \widetilde{post} \rangle$ be the *concrete labeling domain* and $\mathbf{D}_a = \langle \mathcal{S}, \mathcal{D}_a, \sqsubseteq_a, \gamma_a, \widetilde{post} \rangle$ be the *abstract labeling domain.*

Definition 4.3. *An* abstract reachability graph *(ARG) is a tuple* $ARG = \langle N, E, C, n_0, d_c, d_a, e \rangle$*, where*

- $\langle N, E \rangle$ *is a directed tree of nodes* N *and edges* $E \subset N \times N$*, rooted at* n_0*,*
- $C \subseteq N \times N$ *is the set of covering edges,*
- $d_c : N \to \mathcal{D}_c$ *is the labeling of nodes by concrete labels,*
- $d_a : N \to \mathcal{D}_a$ *is the labeling of nodes by abstract labels,*
- $e : E \to \mathcal{T}$ *is the labeling of edges by transitions.*

A node of the ARG $n \in N$ is *expanded* if and only if for all transitions $t \in \mathcal{T}$ enabled from $d_a(n)$ there is an edge $\langle n, n' \rangle \in E$ for some $n' \in N$ labeled with t. A node n is *covered* if and only if $\langle n, n' \rangle \in C$ for some node $n' \in N$. 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.

The ARG is constructed so that the concrete labeling represents some of the concrete paths of the model, while the abstract labeling over-approximates all concrete paths of the model. Furthermore, if $\langle n, n' \rangle \in C$, i.e., n is covered by n', then each state reachable from the abstract label of n is also reachable from the abstract label of n'.

The concrete labeling d_c is never modified during the run of the algorithm. The abstract post-image operator $\overline{post}$ of the concrete labeling domain $\mathbf{D_c}$ is *exact*, which means that it does not introduce unreachable states in the post-image. Still, $\mathbf{D}_c$ is an abstract domain since the concrete labels are abstract states that may represent multiple concrete states (e.g., zones). Identity abstraction and zone abstraction are examples of domains that can serve as the concrete labeling domain.

The abstract labeling d_a is a coarser abstraction. It is refined when needed in a lazy manner. The abstract post-image operator $\widetilde{post}$ of the abstract labeling domain $\mathbf{D_a}$ is not exact. The abstract labeling overapproximates the concrete labeling; the concretizations of abstract labels may include unreachable states as well.

Figure 3 shows an example ARG. For both nodes n_0 and n_1, the concrete label can be seen in the first row of the node (marked with C) and the abstract label in the second row (marked with A). This ARG is used for the reachability analysis of a timed automaton, so both the concrete and abstract labels consist of three states: a location (L), a data state (D), and a time state (T). The location is tracked explicitly in both concrete and abstract labels. Data is represented as a total valuation (e.g., $\{x \mapsto 0\}$) in the concrete label and as a predicate (e.g., *true*) in the abstract label. Time is represented as a zone in both cases; however, concrete labels contain non-overapproximating zones, while the zones in abstract labels may overapproximate the concrete ones, e.g., zone $(c \geq 0)$ overapproximates zone $(c \geq 2)$ in node n_1.

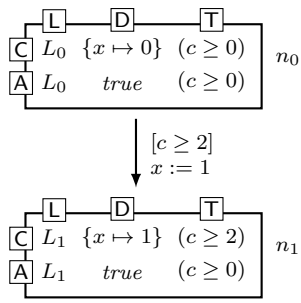


Figure 3: ARG Example

The edge from n_0 to n_1 is labeled by a transition with a compound operation consisting of a clock guard $[c \geq 2]$ and a data assignment $x := 1$. For simplicity, we have omitted the source and target locations of the transition, as they are evident from the location (L) column of nodes. The clock guard $[c \geq 2]$ further constrains $(c \geq 0)$ in the concrete label to $(c \geq 2)$ in node n_1, while the data assignment $x := 1$ yields the data valuation $\{x \mapsto 1\}$.

4.2.2 Requirements for Lazy Abstraction

In this section, we discuss the constraints that the used abstract domains and the ARG should conform to when checking reachability with lazy abstraction.

We only consider those abstract domains $\mathbf{D_c}$ and $\mathbf{D_a}$ for which the post-images produced by $\overline{post}$ and $\widetilde{post}$ consist of at most one abstract state.

With $\mathbf{D_1} = \langle \mathcal{S}, \mathcal{D}_1, \sqsubseteq_1, \gamma_1, post'_1 \rangle$ and $\mathbf{D_2} = \langle \mathcal{S}, \mathcal{D}_2, \sqsubseteq_2, \gamma_2, post'_2 \rangle$ abstract domains, let $\gamma'(\mathbf{D_1}, \mathbf{D_2})$: $\mathcal{D}_1 \to \mathcal{D}_2$ denote a function that represents abstract states of domain $\mathbf{D}_1$ as abstract states of domain $\mathbf{D}_2$. The γ' function can represent the mapping of, e.g., a total valuation to a partial valuation with all variables tracked, forming a connection between identity abstraction and explicit value abstraction. For the labeling domains $\mathbf{D_c}$ and $\mathbf{D}_a$, the function $\gamma_{c\to a} = \gamma'(\mathbf{D_c}, \mathbf{D_a})$ must exist for which $\widetilde{post}_t(\gamma_{c\to a}(d)) = \gamma_{c\to a}(d')$ where $\{d'\} = \overline{post}_t(d)$.

Moreover, let $\vdash \subseteq \mathcal{D}_1 \times \mathcal{D}_2$ denote the *proves* relation, a reflexive and transitive binary relation with the following semantics: $d_1 \vdash d_2$ if and only if $\gamma'(\mathbf{D_1}, \mathbf{D_2})(d_1) \sqsubseteq_2 d_2$, where $d_1 \in \mathcal{D}_1$, $d_2 \in \mathcal{D}_2$. The $d_1 \vdash d_2$ relation also implies that $\gamma_1(d_1) \subseteq \gamma_2(d_2)$.

The ARG for lazy abstraction should conform to the following properties:

- Initiation: $\bigcup_{s_0 \in \mathcal{S}_0} s_0 = \gamma_c(d_c(n_0))$ and $d_c(n_0) \vdash d_a(n_0)$
- Inductive labeling: for each expanded node $n \in N$, transition $t \in \mathcal{T}$, and abstract state $d'_a \in \widetilde{post}_t(d_a(n))$, there exists an edge $\langle n, n' \rangle \in E$ labeled by t to a node $n' \in N$ such that $d'_a \sqsubseteq_a d_a(n')$
- Concrete labeling: $\overline{post}_{e(\langle n,n' \rangle)}(d_c(n)) = \{d_c(n')\}$ for each $\langle n, n' \rangle \in E$
- Simulation: for each expanded node $n \in N$ and transition $t \in \mathcal{T}$ it holds that $\overline{post}_t(d_c(n)) = \emptyset$ implies $\widetilde{post}_t(d_a(n)) = \emptyset$
- Coverage: $d_a(n) \sqsubseteq_a d_a(n')$ for each $\langle n, n' \rangle \in C$

4.2.3 Building an ARG with Lazy Abstraction

The generic lazy abstraction algorithm that builds an ARG with the above properties is presented in Algorithm 1.

The inputs of the algorithm are an initialized ARG (i.e., containing only the labeled initial node n_0), the set of transitions $\mathcal{T}$, and the *error* predicate. The *error* predicate determines whether the abstract label of a given node represents an error state. Therefore, the abstract labels should include precise information about whether a state is an error state or not. For example, when checking the reachability

Algorithm 1 Generic lazy abstraction algorithm

with domains $\mathbf{D}_c = \langle \mathcal{S}, \mathcal{D}_c, \sqsubseteq_c, \gamma_c, \overline{post} \rangle$ and $\mathbf{D}_a = \langle \mathcal{S}, \mathcal{D}_a, \sqsubseteq_a, \gamma_a, \widetilde{post} \rangle$

Require: ARG $= \langle N, E, C, n_0, d_c, d_a, e \rangle$ is an initialized ARG, where $N = \{n_0\}$, $E = \emptyset$, $C = \emptyset$, $\gamma_c(d_c(n_0)) = \bigcup_{s_0 \in \mathcal{S}_0} s_0$ and $d_c(n_0) \vdash d_a(n_0)$

function CHECK(ARG, $\mathcal{T}$, *error*)
  waitlist $\leftarrow N$, *expanded* $\leftarrow \emptyset$
  while $n \in$ *waitlist* for some n **do**
    waitlist $\leftarrow$ *waitlist*$\backslash\{n\}$
    if *error*$(d_a(n))$ **then**
      return unsafe, ARG
    TRYCOVER(ARG, *waitlist*, *expanded*, n)
    if n is not covered **then**
      EXPAND(ARG, *waitlist*, *expanded*, $\mathcal{T}$, n)
  return safe, ARG

function TRYCOVER(ARG, *waitlist*, *expanded*, n)
  for all $n' \in$ *expanded* such that $d_c(n) \vdash d_a(n')$ **do**
    COVER(ARG, *waitlist*, n, n')
    if $d_a(n) \sqsubseteq_a d_a(n')$ **then**
      $C \leftarrow C \cup \{\langle n, n' \rangle\}$
      return

function EXPAND(ARG, *waitlist*, *expanded*, $\mathcal{T}$, n)
  for all $t \in \mathcal{T}$ **do**
    if $\overline{post}_t(d_c(n)) = \emptyset$ **then**
      DISABLE(ARG, *waitlist*, n, t)
    else
      create node n', $N \leftarrow N \cup \{n'\}$, $E \leftarrow E \cup \{\langle n, n' \rangle\}$, $e(\langle n, n' \rangle) \leftarrow t$
      $d_c(n') \leftarrow d'_c$ with $\{d'_c\} = \overline{post}_t(d_c(n))$
      $d_a(n') \leftarrow$ INITABSTR(ARG, n, t)
      waitlist $\leftarrow$ *waitlist* $\cup \{n'\}$
  expanded $\leftarrow$ *expanded* $\cup \{n\}$

Ensure: returns d''_a such that $d'_a \sqsubseteq_a d''_a$ with $\{d'_a\} = \widetilde{post}_t(d_a(n))$

function INITABSTR(ARG, n, t) ▷ *Declaration only. Possible definition is* INITABSTR$_{post}$ *or* INITABSTR$_\top$

function INITABSTR$_{post}$(ARG, n, t)
  return d'_a where $\{d'_a\} = \widetilde{post}_t(d_a(n))$

function INITABSTR$_\top$(ARG, n, t)
  return $\top$

Algorithm 1 Generic lazy abstraction algorithm (continued)

Require: $d_c(n) \vdash d_a(n')$
32: **function** COVER(ARG, *waitlist*, n, n') ▷ *Declaration only. Implementation depends on interpolation domain (see Algorithm 2)*

Require: $\overline{post}_t(d_c(n)) = \emptyset$
Ensure: $\widetilde{post}_t(d_a(n)) = \emptyset$
33: **function** DISABLE(ARG, *waitlist*, n, t) ▷ *Declaration only. Implementation depends on interpolation domain (see Algorithm 2)*

of an error location, locations should be explicitly tracked by the abstract labels as well.

Two sets of nodes are maintained, one for the nodes to be processed (*waitlist*) and one for the already expanded nodes (*expanded*). Nodes in the *waitlist* are processed one by one in a loop until there are no more nodes left. After each iteration of the loop, each node of the ARG is either covered, expanded, or in the *waitlist*.

If the processed node represents an error state, then an *unsafe* result is returned, along with the ARG that serves as proof of the reachability of the error state.

If the processed node is not an error node, then the algorithm attempts to cover it. A covered node does not have to be expanded, thus sealing a branch of the ARG and limiting its growth. By the coverage property of the ARG, $d_a(n) \sqsubseteq_a d_a(n')$ has to hold for nodes $n, n' \in N$ if n is covered by n'. However, instead of searching for a node n' that already satisfies the coverage property with n, the lazy algorithm searches for a node n' for which $d_c(n) \vdash d_a(n')$ holds, then refines the abstract labeling with the aim to achieve $d_a(n) \sqsubseteq_a d_a(n')$. The abstract labeling is refined by calling COVER, which ensures that $d_a(n) \sqsubseteq_a d'$ holds, where d' is the abstract label of n' at the point of calling COVER. The covering edge is added to the ARG if and only if $d_a(n) \sqsubseteq_a d_a(n')$ is successfully achieved by the refinement.

If the node is not covered, then it has to be expanded. The successors are computed on the concrete domain. If a successor of n does not exist on the concrete labeling domain for some transition $t \in \mathcal{T}$, then the abstract label of n has to be refined to maintain the simulation property of the ARG by calling DISABLE. Otherwise, a new successor node is created with the given transition $t \in \mathcal{T}$. The concrete label of the successor node n' is the concrete successor computed by $\overline{post}_t(d_c(n))$. Its abstract label is the over-approximation of the abstract successor that could be computed by $\widetilde{post}_t(d_a(n))$, complying with the inductive labeling property of the ARG. The over-approximation of the abstract successor can be, for example, exactly the abstract state in $\widetilde{post}_t(d_a(n))$ (INITABSTR$_{post}$), or simply the $\top$ element of the abstract labeling domain (INITABSTR$_\top$). The new nodes are added to the waitlist, and n becomes expanded after all its successors are created.

If the waitlist becomes empty, then all nodes are either expanded or covered, and therefore, the ARG is complete. The abstract labeling and paths of a complete ARG represent the whole state space and all possible concrete runs. Since no error states were encountered, a *safe* result is returned, along with the complete ARG, proving that the error state is not reachable.

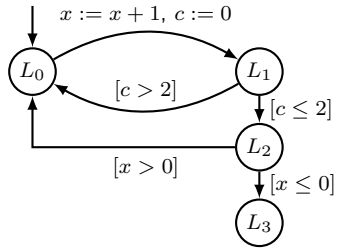


Figure 4: Timed Automaton Example

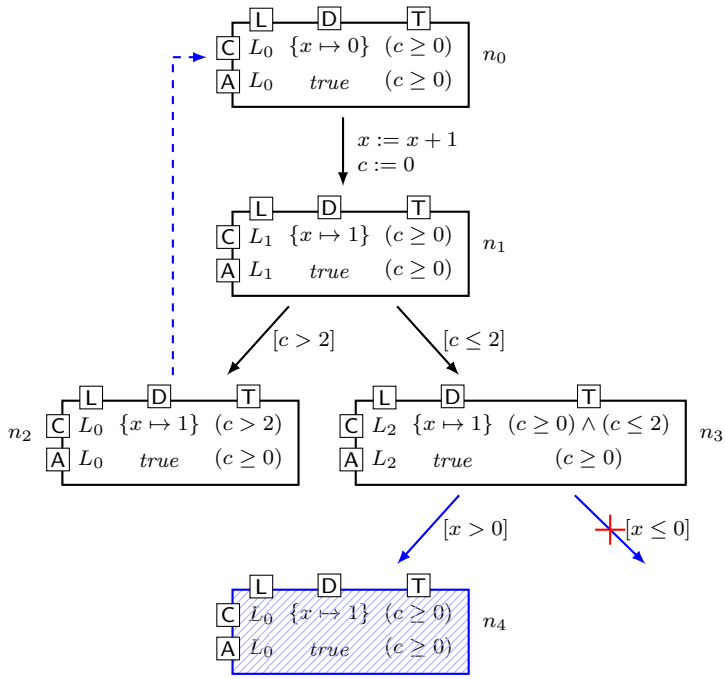


Figure 5: Lazy Abstraction Example

Figure 5 illustrates how the lazy abstraction algorithm builds an ARG. Nodes and edges marked with blue are added to the ARG later, as described in this example. Let us assume that the algorithm has already created nodes n_0, n_1, n_2

and n_3 of the ARG for the timed automaton in Figure 4, and the error predicate is $l(d) = L_3$ where $l(d)$ is the location component of the abstract label d. Nodes n_0 and n_1 of the partial ARG are already expanded, while the two leaf nodes, n_2 and n_3, are not covered and not expanded, i.e., they are pending nodes in the waitlist.

We assume that n_2 is processed first. It does not satisfy the error predicate, so the algorithm tries to cover it. The concrete label of n_2 is $\langle L_0, \{x \mapsto 1\}, (c > 2)\rangle$, which is overapproximated by the abstract label of n_0, $\langle L_0, \mathit{true}, (c \geq 0)\rangle$, therefore, n_2 can be covered by n_0. The algorithm refines the abstract label of n_2 by calling Cover and adds a covering edge to the ARG from n_2 to n_0, represented by a dashed line in the figure. In this case, the refinement done by Cover will be a no-op, since the abstract labels of n_0 and n_2 already satisfy the coverage property.

At this point the only remaining node in the waitlist is n_3, so it is processed next. It cannot be covered, as locations are tracked explicitly and none of the expanded nodes have L_2 as their location. Therefore, n_3 must be expanded.

Transitions with other source than L_2 are not feasible from the concrete label of n_3, so the algorithm presented here calls Disable for them (the algorithm intentionally does not contain timed automata-specific optimizations to be independent of the modeling formalism). In these cases, Disable will be a no-op, since the abstract label of n_3 already contains the information that the location is L_2.

For the transition from L_2 to L_0 with guard $[x > 0]$, there exists a successor of the concrete label of n_3, so a new node n_4 is created, with unchanged data state $\{x \mapsto 1\}$. The concrete zone of n_4 is $(c \geq 0)$, as the constraint $(c \leq 2)$ of the parent node n_3 is not necessarily satisfied anymore due to the delay transition of the timed automaton. The initial data and time states of the abstract label are the $\top$ elements of the respective domains, *true* and $(c \geq 0)$.

For the transition from L_2 to L_3 with guard $[x \leq 0]$, the concrete label of n_3 has no successor, since the guard is not satisfied by the data state $\{x \mapsto 1\}$. Therefore, the abstract label of n_3 will be refined by calling Disable with this transition.

4.2.4 Sound Overapproximation of Safety

We provide a proof for the soundness of the lazy abstraction algorithm w.r.t. safety.

Proposition 4.1. *Algorithm 1 yields an ARG that satisfies the ARG properties for lazy abstraction.*

Proof. The initiation property is presumed to hold at the start of the algorithm. The labeling of the initial node n_0 can only be modified by Cover and Disable. As we prove later in Propositions 4.3 and 4.5, these methods preserve the ARG labeling properties. Then, the initiation property holds in the ARG built by the lazy abstraction algorithm.

When a new node is added to the ARG, then its concrete label is created in accordance with the concrete labeling property, therefore each node satisfies the concrete labeling property when it is created. A node becomes expanded only after all possible transitions are considered. If a transition is infeasible from a node, then Disable is called, thus the simulation property is ensured by the contract of

DISABLE. If the transition is feasible, then a new node is created, for which the abstract label is created in accordance with the inductive labeling property by the contract of INITABSTR. Therefore, a node and its successors satisfy the simulation and inductive labeling properties when the node becomes expanded. Moreover, covering edges are added to the ARG only if the coverage property holds for the nodes, so each covering edge satisfies the coverage property when it is created. The labeling of existing nodes can only be modified by COVER and DISABLE; therefore, the concrete labeling, inductive labeling, simulation, and coverage properties hold later as well. □

Proposition 4.2. *If there is a run σ of the automaton to a state $s \in \mathcal{S}$ and the lazy abstraction algorithm returns a safe result, then there is a non-covered node n in the resulting ARG such that $s \in \gamma_a(d_a(n))$.*

Proof. The algorithm returns a safe result only when the waitlist is empty, and therefore, all nodes are either expanded or covered, i.e., the ARG is complete. The above proposition is then a consequence of the next lemma. □

Lemma 4.1. *If there is a run $\sigma = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots \xrightarrow{t_k} s_k$ of the automaton and the ARG for the automaton is complete, then there is a non-covered node n in the ARG such that $s_k \in \gamma_a(d_a(n))$.*

Proof. The lemma can be proven by induction on the length of the run σ.

It holds for a run consisting of only one state. The first state must be an initial state $s_0 \in \mathcal{S}_0$. By the initiation property $s_0 \in \gamma_a(d_a(n_0))$, therefore n_0 is a witness for the statement.

Suppose that the statement holds for runs with at most k states.

Then there exists a non-covered node n_{k-1} such that

$$s_{k-1} \in \gamma_a(d_a(n_{k-1})). \tag{1}$$

The ARG is complete, and n_{k-1} is not covered, so n_{k-1} must be an expanded node. By the properties of abstract post-image operators,

$$\bigcup_{s \in \gamma_a(d_a(n_{k-1}))} post_{t_k}(s) \subseteq \bigcup_{d' \in \widetilde{post}_{t_k}(d_a(n_{k-1}))} \gamma_a(d'). \tag{2}$$

From (1) and (2) we have

$$post_{t_k}(s_{k-1}) \subseteq \bigcup_{d' \in \widetilde{post}_{t_k}(d_a(n_{k-1}))} \gamma_a(d'). \tag{3}$$

Moreover, from the concrete run we have $s_k \in post_{t_k}(s_{k-1})$, hence from (3) it follows that

$$s_k \in \bigcup_{d' \in \widetilde{post}_{t_k}(d_a(n_{k-1}))} \gamma_a(d'). \tag{4}$$

This implies that $\widetilde{post}_{t_k}(d_a(n_{k-1})) \neq \emptyset$, which means that

$$\exists d^* \in \mathcal{D}_a : \widetilde{post}_{t_k}(d_a(n_{k-1})) = \{d^*\}. \tag{5}$$

By (4) and (5) it holds that

$$s_k \in \gamma_a(d^*). \tag{6}$$

By (5) and the inductive labeling property of the ARG, it follows that there exists an edge $\langle n_{k-1}, n_k \rangle \in E$ labeled by t_k to a node $n_k \in N$ such that

$$d^* \sqsubseteq_a d_a(n_k). \tag{7}$$

From (7) and the properties of the preorder relation, we have

$$\gamma_a(d^*) \subseteq \gamma_a(d_a(n_k)). \tag{8}$$

From (6) and (8) it follows that

$$s_k \in \gamma_a(d_a(n_k)). \tag{9}$$

Therefore, if n_k is not covered, then it is a witness for the statement.

If n_k is covered, then there exists a non-covered node n_c that covers n_k for which $d_a(n_k) \sqsubseteq_a d_a(n_c)$ by the coverage property of the ARG. By the properties of the preorder relation it follows that $\gamma_a(d_a(n_k)) \subseteq \gamma_a(d_a(n_c))$, and from this and (9) we get $s_k \in \gamma_a(d_a(n_c))$. Thus, in this case, n_c is a witness. □

4.3 Refinement of the Abstraction

The lazy abstraction algorithm relies on abstraction refinement to maintain the ARG labeling properties when expanding nodes and to enable more coverages. In the presented framework, various refinement algorithms can be used that conform to the given contracts and preserve the ARG labeling properties. Here, we present two algorithms for abstraction refinement based on binary interpolation. We also introduce an additional generic abstract domain to assist with refinement and to avoid having too many constraints for the concrete and abstract labeling domains.

4.3.1 The Interpolation Domain

For an abstract domain $\mathbf{D} = \langle \mathcal{S}, \mathcal{D}, \sqsubseteq, \gamma, post' \rangle$, let $pre' \colon \mathcal{D} \times \mathcal{T} \to 2^{\mathcal{D}}$ be the *abstract pre-image* operator that maps an abstract state $d \in \mathcal{D}$ to its predecessor states with respect to a transition $t \in \mathcal{T}$, such that $\bigcup_{s \in \gamma(d)} pre_t(s) \subseteq \bigcup_{d' \in pre'_t(d)} \gamma(d')$.

The interpolation algorithms presented here require that we can compute the pre-image of an abstract state d and that the pre-image is *exact*, i.e., it does not contain states from which d is unreachable. However, not all abstract domains are equally suitable for representing pre-images. For example, with explicit value abstraction, we might get too many states as the pre-image, while these can be represented easily as a first-order logic formula. However, it is not required that the pre-images of the concrete or abstract labels can be computed and represented effectively. Instead, pre-images are computed on an *interpolation domain* $\mathbf{D}_i = \langle \mathcal{S}, \mathcal{D}_i, \sqsubseteq_i, \gamma_i, \widehat{post} \rangle$, with its corresponding pre-image operator $\widehat{pre}$. To be able

to represent abstract labels as an abstract state of the interpolation domain, we require that the function $\gamma'(\mathbf{D_a}, \mathbf{D_i})$ exists, we will denote it as $\gamma_{a \to i}$.

With $\mathbf{D_1} = \langle \mathcal{S}, \mathcal{D}_1, \sqsubseteq_1, \gamma_1, post'_1 \rangle$ and $\mathbf{D_2} = \langle \mathcal{S}, \mathcal{D}_2, \sqsubseteq_2, \gamma_2, post'_2 \rangle$ abstract domains, let $\lightning \subseteq \mathcal{D}_1 \times \mathcal{D}_2$ denote the *refutes* relation, a symmetric binary relation that has the following semantics: if $d_1 \lightning d_2$ then $\gamma_1(d_1) \cap \gamma_2(d_2) = \emptyset$, where $d_1 \in \mathcal{D}_1$ and $d_2 \in \mathcal{D}_2$. It should be noted that the implication does not hold in the opposite direction.

Regarding the refutes operator, the relationship between abstract pre-images on the interpolation domain and abstract post-images on the abstract labeling domain is defined for each $d'_i \in \mathcal{D}_i$ and $d_a \in \mathcal{D}_a$ as follows:

$$\forall d_i \in \widehat{pre}_t(d'_i) \colon d_a \lightning d_i \iff \forall d'_a \in \widetilde{post}_t(d_a) \colon d'_a \lightning d'_i \tag{10}$$

We also require that the complement $\neg d$ of an abstract state $d \in \mathcal{D}_i$ can be computed. It should represent all states in $\mathcal{S} \backslash \gamma_i(d)$; however, this often cannot be represented as a single abstract state, e.g., zones are not closed under complementation, but the complement can be represented as the union of a finite number of zones. The complement is, therefore, a set of abstract states such that $\bigcup_{d' \in \neg d} \gamma_i(d') = \mathcal{S} \backslash \gamma_i(d)$. It also holds that $d_1 \vdash d_2$ if and only if $\forall d'_2 \in \neg d_2 \colon d_1 \lightning d'_2$.

4.3.2 Refining the Abstraction Correctly

This section explains the various countermeasures that need to be taken in the abstraction refinement algorithms to prevent violations of the ARG properties.

Abstraction refinement is performed in two cases, as presented in Algorithm 2: when attempting to add a covering edge or when a transition is disabled on the concrete labeling domain. In both cases, refinement is done by removing certain parts from the abstract label of the given node.

For covering edges $\langle n, n' \rangle \in C$ of the ARG, the coverage property $d_a(n) \sqsubseteq_a d_a(n')$ has to hold. Cover ensures that $d_a(n) \sqsubseteq_a d'$ where d' is the abstract label of n' at the point of calling the function. Cover takes the complement of d' and removes it from $d_a(n)$. As a result, $d_a(n)$ does not contain any states that are not in d', hence $d_a(n) \sqsubseteq_a d'$ holds. However, because of refinement propagation, which is discussed later in this section, the abstract label $d_a(n')$ may also change during the refinement if n' is a predecessor of n, and thus $d_a(n) \sqsubseteq_a d_a(n')$ is not necessarily ensured in those cases.

When a node n is being expanded and a transition t is encountered that is disabled from $d_c(n)$, i.e., $\overline{post}_t(d_c(n)) = \emptyset$, then an abstract state $d'_a \in \widetilde{post}_t(d_a(n))$ may still exist, that would violate the simulation property of the ARG when n becomes expanded. Disable refines the abstract labeling to prevent the violation of the simulation property by ensuring that $\widetilde{post}_t(d_a(n)) = \emptyset$.

Lemma 4.2. Disable *is consistent with its contract, i.e., for node n and transition t it ensures that $\widetilde{post}_t(d_a(n)) = \emptyset$.*

Proof. Disable calls Block to remove all pre-images of $\gamma_{a \to i}(\top)$ by the transition t from the abstract label of n. After this, $\forall B \in \widehat{pre}_t(\gamma_{a \to i}(\top)) \colon d_a(n) \lightning B$ holds by

Algorithm 2 Generic lazy abstraction refinement algorithms

Require: $d_c(n) \vdash d_a(n')$
1: **function** COVER(ARG, *waitlist*, n, n')
2: **for all** $B \in \neg\gamma_{a\to i}(d_a(n'))$ **do**
3: BLOCK(ARG, *waitlist*, n, B)

Require: $\widetilde{post}_t(d_c(n)) = \emptyset$
Ensure: $\widetilde{post}_t(d_a(n)) = \emptyset$
4: **function** DISABLE(ARG, *waitlist*, n, t)
5: **for all** $B \in \widehat{pre}_t(\gamma_{a\to i}(\top))$ **do**
6: BLOCK(ARG, *waitlist*, n, B)

Require: $d_c(n) \not\downarrow B$
Ensure: $d_a(n) \not\downarrow B$
7: **function** BLOCK(ARG, *waitlist*, n, B) ▷ *Declaration only. Implementation depends on interpolation strategy. See* $BLOCK_{BW}$ *(Algorithm 4) or* $BLOCK_{FW}$ *(Algorithm 5)*

the contract of BLOCK. By (10) it follows that $\forall d'_a \in \widetilde{post}_t(d_a(n))\colon d'_a \not\downarrow \gamma_{a\to i}(\top)$, which implies that $\widetilde{post}_t(d_a(n)) = \emptyset$. Therefore, DISABLE satisfies its contract. □

Maintaining the ARG properties is necessary not only when adding a covering edge or expanding a node but also when an abstract label is updated. The update of an abstract label may not preserve the inductive labeling and coverage properties of the ARG automatically. Therefore, additional steps need to be taken to maintain these properties:

- After refining an abstract label $d_a(n)$, the inductive labeling property $d'_a \sqsubseteq_a d_a(n)$ may not hold for the ARG edge $\langle p, n\rangle \in E$ with $d'_a \in \widetilde{post}_{e(\langle p,n\rangle)}(d_a(p))$. To maintain the inductive labeling property, the abstract label of the parent node p should also be refined, which may necessitate further refinement, etc. Therefore, abstraction refinement is propagated in the ARG. Sections 4.3.3 and 4.3.4 are dedicated to these refinement propagation algorithms.
- If the refined node n covers a node n_{cov}, the coverage property $d_a(n_{cov}) \sqsubseteq_a d_a(n)$ may not hold after the refinement of $d_a(n)$. The coverage property is maintained immediately when updating the abstract label by removing covering edges that violate this property, as it is shown in Algorithm 3.

In fact, the removal of covering edges is a key part of the lazy abstraction algorithm. Covering edges are added easily to the ARG, taking advantage of the coarseness of the abstract labeling, initially sealing most of the branches of the ARG. The exploration of the state space is propelled forward by the removal of these covering edges.

Algorithm 3 Strengthening of an abstract label

Ensure: $d_a(n) \sqsubseteq_a I$
1: **function** STRENGTHEN(ARG, *waitlist*, n, I)
2: $\quad d_a(n) \leftarrow d_a(n) \sqcap_a I$
3: $\quad$ **for all** $\langle n_{cov}, n\rangle \in C$ such that $d_a(n_{cov}) \not\sqsubseteq_a d_a(n)$ **do**
4: $\quad\quad C \leftarrow C \backslash \{\langle n_{cov}, n\rangle\}$
5: $\quad\quad waitlist \leftarrow waitlist \cup \{n_{cov}\}$

In the following sections 4.3.3 and 4.3.4, we provide two methods for the propagation of abstraction refinement based on [34], using *binary interpolants*.

Definition 4.4. *The binary interpolant* $itp(A, B)$ *of abstract states* $A \in \mathcal{D}_a$ *and* $B \in \mathcal{D}_i$ *for which* $A \not\perp B$ *holds, is the abstract state* $I \in \mathcal{D}_a$ *such that* $A \vdash I$ *and* $I \not\perp B$.

Both refinement propagation methods work on some suffix of the path from the root node to the leaf node that initiates the refinement. The first method propagates the refinement starting from the leaf node and stepping *backward* in the ARG, while the second method starts by finding the suitable suffix of the path that should be refined, then performs refinement in a *forward* direction.

4.3.3 Backward Binary Interpolation

Algorithm 4 shows the backward propagation of abstraction refinement in the ARG. In the backward propagation algorithm, the abstract label of the node is first refined by a suitable interpolant, then the pre-image of the complement of the interpolant is removed from the abstract label of the parent node in a recursive refinement step. In other words, based on (10), the complement of the interpolant is removed from the abstract post-image of the parent. Therefore, the abstract post-image of the parent becomes over-approximated by the computed interpolant, preserving the inductive labeling of the ARG.

Proposition 4.3. *The backward propagation algorithm preserves the ARG properties.*

Proof. Since the concrete labeling is not modified by the algorithm, the concrete labeling property trivially holds. The new abstract labels always contain fewer states than before the strengthening, therefore the simulation property is also preserved. Furthermore, it is easy to see that the strengthening keeps only those covering edges that conform to the coverage property. If the propagation reaches and affects the root node as well, then its abstract label is strengthened by an interpolant I for which $d_c(n_0) \vdash I$ holds, so $d_c(n_0) \vdash d_a(n_0)$ and thus the initiation property is also preserved.

We prove that the labeling stays inductive by showing that $d'_p \sqsubseteq_a d_a(n)$ is preserved, where $\{d'_p\} = \widetilde{post}_{e(\langle p,n\rangle)}(d_a(p))$ is the abstract successor of p, and p is

Algorithm 4 Backward propagation of abstraction refinement

Require: $d_c(n) \not\downarrow B$
Ensure: $d_a(n) \not\downarrow B$
1: **function** BLOCK$_{BW}$(ARG, *waitlist*, n, B)
2: **if** $d_a(n) \not\downarrow B$ **then**
3: **return**
4: $A \leftarrow \gamma_{c\to a}(d_c(n))$
5: $I \leftarrow itp(A, B)$
6: STRENGTHEN(ARG, *waitlist*, n, I)
7: **if** $\langle p, n\rangle \in E$ for some p **then**
8: **for all** $B' \in \neg\,\gamma_{a\to i}(I)$ **do**
9: **for all** $B'_{pre} \in \widehat{pre}_{e(\langle p,n\rangle)}(B')$ **do**
10: BLOCK$_{BW}$(ARG, *waitlist*, p, B'_{pre})

the parent node of n. For all $B' \in \neg\,\gamma_{a\to i}(I)$ by contract it holds that $\forall B'_{pre} \in \widehat{pre}_{e(\langle p,n\rangle)}(B')$: $d_a(p) \not\downarrow B'_{pre}$ and therefore $d'_p \not\downarrow B'$ by (10). From this, it follows that $d'_p \sqsubseteq_a I$. As the abstract label $d_a(n)$ is strengthened by I, it follows that $d'_p \sqsubseteq_a d_a(n)$ is preserved, and therefore the labeling stays inductive. □

Proposition 4.4. *The backward propagation algorithm is consistent with its contract.*

Proof. By the definition of interpolants $I \not\downarrow B$, so $d_a(n) \not\downarrow B$ holds for the new labeling.

We also show that it is ensured that the precondition for the recursive call holds, i.e., $d_c(p) \not\downarrow B'_{pre}$. By the concrete labeling property $\{d_c(n)\} = \widetilde{post}_{e(\langle p,n\rangle)}(d_c(p))$, and $d_c(n) \vdash I$ by the definition of interpolants, therefore $d' \sqsubseteq_a I$ where $\{d'\} = \widetilde{post}_{e(\langle p,n\rangle)}(\gamma_{c\to a}(d_c(p)))$. This implies $\forall B' \in \neg\,\gamma_{a\to i}(I)$: $d' \not\downarrow B'$, and by (10) it follows that $\forall B'_{pre} \in \widehat{pre}_t(B')$: $d_c(p) \not\downarrow B'_{pre}$, satisfying the precondition for the recursive call. □

Figure 6 illustrates the refinement of a path to an infeasible transition. The path shown here is from the previous example in Figure 5, where the concrete label of node n_3 has no successor for the transition with guard $[x \leq 0]$. The successor does not exist due to the data guard, so the refinement affects only the data states. Therefore, we do not consider the location and time states in the example. Similarly to [21, 27], we use predicates as abstract labels in the data domain.

In the following, we describe the refinement process with backward propagation, however, note that in this case the result would be the same with forward propagation. Calling DISABLE for node n_3 and the transition with guard $[x \leq 0]$, the algorithm computes the pre-image of $\top$ for $[x \leq 0]$, which is $x \leq 0$. Then, BLOCK$_{BW}$ is called with node n_3 and the computed predicate $x \leq 0$ as state B:

1. The abstract label of n_3 (*true*) does not refute $x \leq 0$, so BLOCK$_{BW}$ first computes the interpolant for the concrete label of n_3 and $x \leq 0$. More

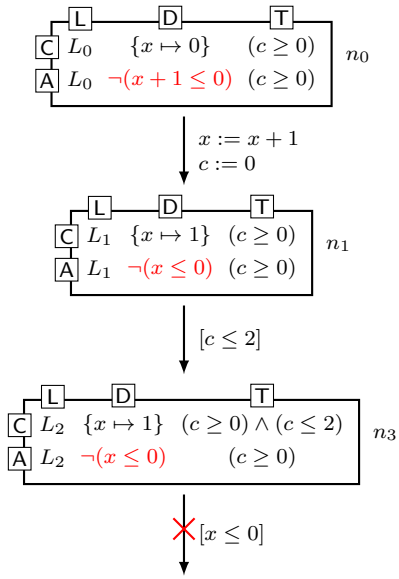


Figure 6: Lazy Refinement Example

precisely, the interpolant is computed for the predicates $x = 1$ (from the concrete label of n_3) as state A, and $x \leq 0$ as state B. The interpolant is $\neg(x \leq 0)$. Node n_3 is strengthened by this interpolant, i.e., its abstract label becomes $\neg(x \leq 0)$. The algorithm then computes the complement, which is $(x \leq 0)$, then computes its pre-image for the clock guard $[c \leq 2]$ on the incoming edge. The pre-image is again $x \leq 0$, which is used as state B for the refinement of the previous node in the path, n_1.

2. As the refinement of node n_1, the same interpolation and strengthening takes place as for n_3. Then, the pre-image of $x \leq 0$ is computed for the data assignment $x := x + 1$ on the incoming edge. The pre-image is $x + 1 \leq 0$, which is used as state B for the refinement of the root node, n_0.

3. The abstract label of n_0 (*true*) does not refute $x + 1 \leq 0$, so the refinement algorithm computes the interpolant for $x = 0$ (from the concrete label of n_0) and $x + 1 \leq 0$. The interpolant is $\neg(x + 1 \leq 0)$. The algorithm then strengthens the abstract label of n_0 with this interpolant. In our previous example (Figure 5), node n_0 covers n_2, which has *true* as its abstract label. However, the coverage property is not satisfied after strengthening n_0, i.e., $true \not\Rightarrow \neg(x + 1 \leq 0)$, so n_2 is uncovered and put back in the waitlist by STRENGTHEN. Then, since n_0 is the root node, the refinement ends here.

4.3.4 Forward Binary Interpolation

Algorithm 5 shows the forward propagation of refinement. In the forward propagation algorithm, the parent of the given node is refined first in a recursive refinement step. If a state B should be removed from the abstract label of the child node, then the pre-images of B are removed from the abstract label of the parent node. In other words, B is removed from the abstract post-image of the parent. Thus, inductive labeling is still preserved when B gets removed from the abstract label of the child node. The recursive refinement steps return interpolants that prove the concrete label of the parent while also refuting the corresponding pre-image of B. Therefore, the abstract post-images of these returned interpolants prove the concrete label of the child node and refute B. At last, the child node is refined by an interpolant of these post-images and B.

Algorithm 5 Forward propagation of abstraction refinement

Require: $d_c(n) \not\downarrow B$
Ensure: $d_a(n) \not\downarrow B$
Ensure: $d_a(n) \vdash I$
function BLOCK$_{FW}$(ARG, *waitlist*, n, B)
  if $d_a(n) \not\downarrow B$ **then**
    return $d_a(n)$
  if $\langle p, n\rangle \in E$ for some p **then**
    $A \leftarrow \top$
    for all $B_{pre} \in \widehat{pre}_{e(\langle p,n\rangle)}(B)$ **do**
      $I' \leftarrow$ BLOCK$_{FW}$(ARG, *waitlist*, p, B_{pre})
      $\{d\} \leftarrow \widetilde{post}_{e(\langle p,n\rangle)}(I')$
      $A \leftarrow A \sqcap_a d$
  else
    $A \leftarrow \gamma_{c\to a}(d_c(n))$
  $I \leftarrow itp(A, B)$
  STRENGTHEN(ARG, *waitlist*, n, I)
  return I

Proposition 4.5. *The forward propagation algorithm preserves the ARG properties.*

Proof. The forward propagation algorithm preserves the initiation, concrete labeling, simulation, and coverage properties in the same way as the backward propagation algorithm.

We also show that the labeling stays inductive. Similarly to the backward propagation case, we show that $d'_p \sqsubseteq_a d_a(n)$ is preserved, where $\{d'_p\} = \widetilde{post}_{e(\langle p,n\rangle)}(d_a(p))$ is the abstract successor of p. By contract $d_a(p) \vdash I'$ holds for interpolants I' computed by the recursive call. By the monotonity of post-images, $d'_p \vdash d$ where $\{d\} = \widetilde{post}_{e(\langle p,n\rangle)}(I')$. By the definition of interpolants $A \vdash I$, where A is the meet

of the post-images computed by $\widetilde{post}_{e(\langle p,n\rangle)}(I')$. Therefore, $d'_p \vdash I$ also holds, and since $d_a(n)$ is strengthened by I, the labeling stays inductive. □

Proposition 4.6. *The forward propagation algorithm is consistent with its contract.*

Proof. The abstract label $d_a(n)$ is strengthened by I, so $d_a(n) \vdash I$ clearly holds. By the definition of interpolants $I \not\downarrow B$, so $d_a(n) \not\downarrow B$ also holds.

We also show that the precondition for the recursive call holds, i.e., $d_c(p) \not\downarrow B_{pre}$. By the concrete labeling property $\{d_c(n)\} = \overline{post}_{e(\langle p,n\rangle)}(d_c(p))$, and $d_c(n) \not\downarrow B$ by contract, therefore $d' \not\downarrow B$ where $\{d'\} = \widetilde{post}_{e(\langle p,n\rangle)}(\gamma_{c\to a}(d_c(p)))$. By (10) it follows that $\forall B_{pre} \in \widehat{pre}_{e(\langle n,p\rangle)}(B)\colon d_c(p) \not\downarrow B_{pre}$, satisfying the precondition. □

4.3.5 Comparison of forward and backward binary interpolation

We have shown two different methods for propagating abstraction refinement in the ARG. The forward method first finds the suffix of the path that needs to be refined, then works forward on that suffix; while the backward method takes backward steps in the ARG and refines nodes immediately, until no further refinement is needed.

Although quite similar approaches, the forward and backward methods differ in the operations they use. Therefore, to get an efficient implementation for propagating abstraction refinement, the used abstract domains must implement these operations efficiently.

The backward propagation algorithm computes the complement of an abstract state in each step on the path. This is usually easy to compute; however, in zone abstraction, sometimes the complement cannot be represented as a single zone, only as a union of zones. If this is the case, then the refinement has to be propagated further through all resulting zones.

The forward propagation algorithm does not contain complement computation, which in theory makes it better suitable for zone domains. On the other hand, in each step on the path, it needs to compute both a pre-image on the interpolation domain and a post-image on the abstract domain, while the backward method only needs to compute a pre-image. Moreover, the forward algorithm needs to keep all previous stack frames, while the backward algorithm uses tail recursion.

In practice, both methods perform similarly well for the abstract domains we discussed in this work, as shown by our experiments (see Figure 10 in Chapter 6). The same conclusions were reached in [34].

5 Parallel Lazy Abstraction for Timed Automata

Modern computational platforms utilize multicore processors and shared memory architectures to enhance performance and efficiency. Multicore processors contain multiple processing units that are able to perform calculations in parallel, while shared memory enables collaboration and data sharing between the processing

units. As a result, modern computational platforms can efficiently handle complex tasks where parts of the workload can be parallelized. In this chapter, we elaborate on how we enhanced the lazy abstraction algorithm to take advantage of the nature of modern computational platforms.

5.1 On the Potential Parallelization of Lazy Abstraction

Lazy abstraction, as described in Algorithm 1, is inherently a sequential algorithm. It maintains a waitlist of nodes to process. While processing a node, first, it tries to cover it with any of the already expanded nodes. If it fails to find a suitable node to cover with, it will calculate the succeeding nodes and expand the node with them. If the node can potentially be covered or an infeasible succeeding node is encountered, the abstract labels are refined along the trace toward the root of the ARG.

Figure 7 illustrates the steps of lazy abstraction in more detail. Figure 7.a depicts a partially complete ARG. Three nodes, l_0, l_1, and l_2, have already been expanded (with grey background), while l_3, l_4, and l_5 are in the waitlist waiting to be processed (white background).

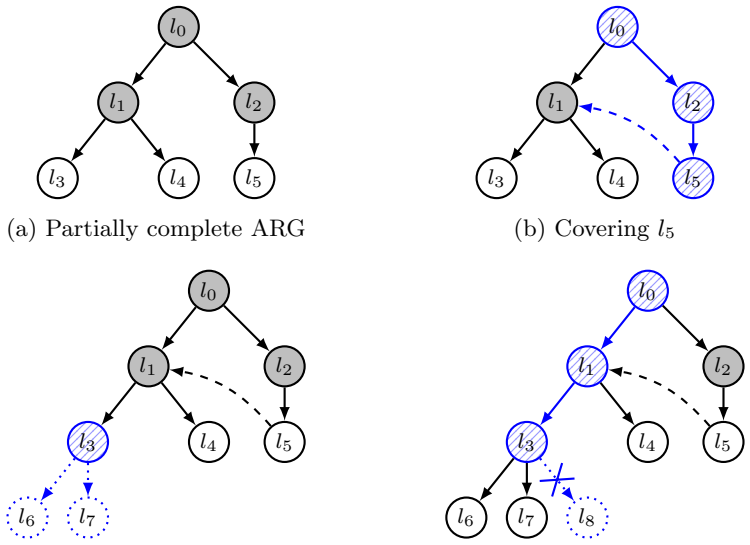
While processing node l_5, the algorithm first tries to cover the node. It will try to cover it with one of the expanded nodes: l_0, l_1 or l_2. In Figure 7.b, l_5 is being covered with l_1 (dashed arrow), and refinement takes place on the trace from l_5 to the root (l_5, l_2 and l_0). The nodes whose abstract labels are (potentially) changed during this operation are highlighted with a blue-striped background.

Figures 7.c and 7.d depicts the operation of expanding l_3. In the first step, l_6 and l_7 are discovered, which are both feasible. They are calculated from the state of l_3 and the operations on the edges. However, l_8 is not feasible, so the abstract label of l_3 needs to be adjusted accordingly. As a result, the nodes along the trace from l_3 towards the root (l_3, l_1, and l_0) need to be refined and might be modified (highlighted with a blue striped background).

We have made a couple of observations regarding the sequential lazy abstraction algorithm.

Observation 1 (Locality of expanding)**.** Our first observation is the locality of the expanding operation. The expand operation only needs the labels of the node it tries to expand to calculate the successor nodes. Moreover, as long as the successor nodes are feasible, there is no need for refinement. Owing to the locality, it might be possible to calculate the successor nodes of two distinct nodes in parallel to each other.

Observation 2 (Read-only nature of covering)**.** Our second observation was made regarding covering. First, the operation searches for a suitable node to cover the processed node with. To this end, it systematically checks for each expanded node if the cover relation holds between the processed and already expanded nodes. To establish if the cover relation holds, the only information needed is the labels of the covering node and the node to be covered. It follows that it is possible to check in



(a) Partially complete ARG (b) Covering l_5

(c) Expanding l_3 (d) Disabling l_8

Figure 7: Overview of the sequential algorithm

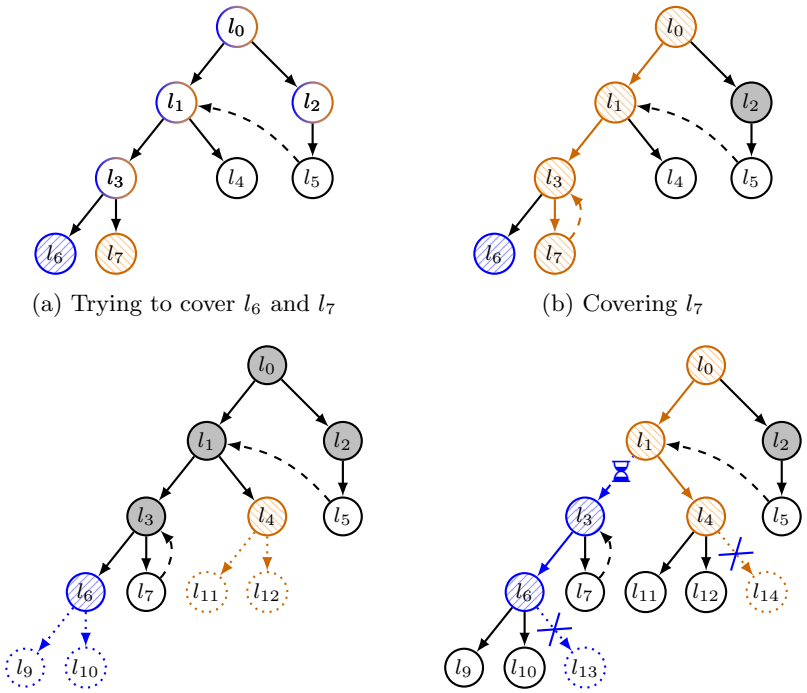
parallel if a node covers multiple other nodes as long as the labels of the potential covering nodes do not change.

Observation 3 (Partial independence of refinement)**.** Our final observation targets the behavior of refinement. Refinement might modify the abstract labels of nodes on the trace from the covered or infeasible node to the root of the ARG. However, with binary interpolation, not necessarily all nodes are modified; only some nodes will be adjusted. Moreover, refinement can be calculated node-by-node towards the root, making it possible to (at least partially) calculate two refinement operations in parallel.

5.2 Illustrating the Parallelization of Lazy Abstraction

Lazy abstraction is sequential as described in Algorithm 1. However, taking into account the observations made in the previous section, it is possible to run parts of the algorithm in parallel. In this section, we assume that multiple processes of a modern computational platform perform the operations of lazy abstraction in parallel.

Figure 8 continues the example from Figure 7 but depicts a scenario where two processes perform the operations of lazy abstraction in parallel. Figure 8.a depicts a scenario where two different processes try to cover l_6 and l_7. Both processes consider each of the already expanded nodes (l_0, l_1, l_2, and l_3, depicted with a blue-orange background). Assuming that the labels of the aforementioned nodes



(a) Trying to cover l_6 and l_7

(b) Covering l_7

(c) Expanding l_6 and l_4

(d) Disabling l_{13} and l_{14}

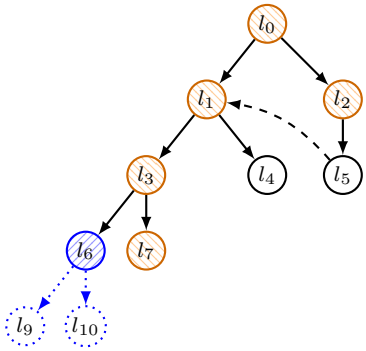
Figure 8: Overview of the parallel algorithm

do not change, their suitability for the cover relation can be checked in parallel. In Figure 8.b node l_7 is being covered by l_3: as a result, the labels of nodes along the trace to the root (l_7, l_3, l_1 and l_0) might change, blocking other processes from considering them for covering. Nonetheless, the other process is still able to check whether l_2 covers l_6, as it is not modified during the refinement stemming from covering l_7.

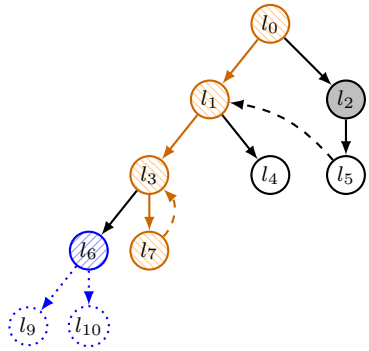
The operation of expanding multiple nodes at once is depicted in Figure 8.c. Nodes l_4 and l_6 are being expanded by calculating the successor nodes: l_{11} and l_{12} for l_4, while l_9 and l_{10} for l_6. As calculating successor nodes only needs localized information found in l_4 and l_6, it can be performed in parallel.

Moreover, Figure 8.d depicts a scenario where two refinements take place at the same time. One of the traces from l_4 (l_4-l_{14}) and l_6 (l_6-l_{13}) is infeasible, so the trace towards the root of the ARG needs to be adjusted accordingly. Stemming from the refinement of l_4, the nodes l_4, l_1 and l_0 might change, while l_6 might incur the change of l_6, l_3, l_1 and l_0, with both refinements targeting l_1 and l_0. Thereby, one of the refinements needs to have precedence, while the other needs to wait for the precedence-having refinement to finish. Let us assume for the sake of this example that the refinement of l_4 gains precedent, while the refinement of l_6 needs to wait

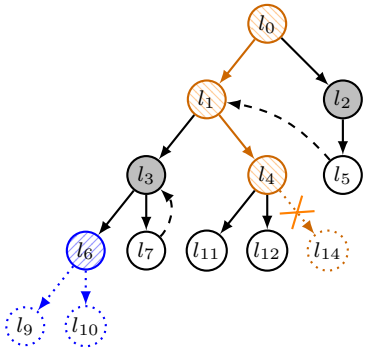
for the former to finish. (It could be the other way around as well.)



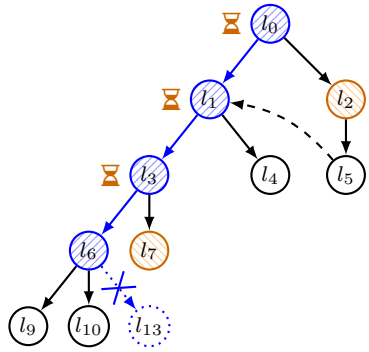
(a) Expanding l_6 and trying to cover l_7



(b) Expanding l_6 and refining l_7 for coverage



(c) Expanding l_6 and disabling l_{14}



(d) Refining l_7 for coverage and disabling l_{13}

Figure 9: Overview of the parallel algorithm (cont'd)

While Figure 8 depicts how the same operations affect the ARG when they are performed in parallel, Figure 9 focuses on scenarios on how different operations may interact.

Figure 9.a shows a situation where l_6 is expanded while l_7 is being covered. As expanding only requires local information, the succeeding nodes (l_9, l_{10}) can be calculated ignoring the rest of the ARG, so covering can check the already expanded nodes (l_3, l_2, l_1 and l_0) for a cover relation uninterrupted. If covering does find a suitable cover relation (Figure 9.b), the refinement stemming from covering l_7 can commence without conflict as well.

On the other hand, Figure 9.c depicts a scenario in which l_6 is expanded while l_{14} is being disabled. Again, as expanding only requires local information, the succeeding nodes (l_9, l_{10}) can be calculated, ignoring the rest of the ARG and the refinement along the trace from l_4 towards l_0.

Finally, Figure 9.d represents a situation where l_7 is being covered while a refinement takes place from l_6 towards l_0 owing to l_{13} being disabled. It is worth

noting that the situation would be the same if the refinement took place in response to the establishment of a cover relation. While trying to cover l_7, the labels of completed nodes (l_3, l_2, l_1, and l_0) need to be checked for a suitable cover relation. In turn, the refinement of l_6 might modify the labels of l_6, l_3, l_1 and l_0. It follows that both operations target l_3, l_1, and l_0, so one of them needs to have precedent over the other, making the other wait for the precedent-having operation to finish. In the figure, the refinement takes precedence. However, the presence of some cover relations (l_2 in this instance) can be checked while the refinement commences.

In conclusion, we have made another set of observations on how the operations of lazy abstraction can be performed in parallel to each other to guide the construction of a parallel lazy abstraction algorithm.

Observation 4 (Independence of expand)**.** The expand operation can commence, disregarding every other operation. First of all, as a node currently being expanded is not considered expanded, covering will not attempt to cover another node with it yet. Moreover, as it is not yet expanded, it cannot be modified as part of the refinement of another node.

Observation 5 (Conflict between refinements)**.** If the expand operation requires disabling a transition, the refinement might have a conflict with other refinements, or partially with covering. Similarly, if the refinement happens as a response to the establishment of a cover relation, the refinement might have a conflict with other refinements or partially with covering.

Observation 6 (Partial independence of cover)**.** Finally, the covering operation targets many nodes in the ARG, but it only reads them: provided the labels of nodes are not modified, multiple covering operations can work on them. However, a refinement might change the labels, which means that the checking for coverage requires the nodes not to be currently refined.

5.3 Parallel Lazy Abstraction Algorithm

We analyzed the observations to determine how to parallelize the sequential lazy abstraction algorithm. Motivated by Observations 2, 3 and 5, we decided to extend the generic lazy abstraction algorithm by introducing a read-write based locking mechanism and a thread-pool-based parallel processing approach to achieve the parallel processing of the nodes of the ARG. A read lock on a node depicts that the process that owns the lock may read the labels of the node, other processes may also read the labels (provided they also acquire a read lock), and no other process may modify the labels. On the other hand, a write lock means that the process that owns the lock may modify the labels of the node, and no other process may modify or read the labels.

5.3.1 Parallel Building of the ARG

The algorithm presented in Algorithm 6 utilizes a thread pool to parallelize the verification process efficiently. Initially, it initializes a thread pool with a specified

number of threads ($thread_{\#}$). Then, it sets up three sets: *waitlist* to keep track of nodes to process, *expanded* to store nodes that have been expanded, and *futures* to hold references to the results of the asynchronous computation tasks submitted to the thread pool.

The algorithm iterates until both the *waitlist* and *futures* sets are empty, indicating that all nodes have been processed and all processing has yielded a verdict. In each iteration, it processes nodes in the *waitlist* by submitting verification tasks to the thread pool for each node. Once a task is completed, it retrieves the result and updates the sets accordingly. If a node is found to be unsafe, the algorithm terminates and returns the verdict along with the unsafe node. Otherwise, it continues by adding newly discovered nodes to the *waitlist* and marking the current node as expanded.

This process continues until all nodes have been explored or an unsafe node is detected. Finally, if no unsafe nodes are found, the algorithm concludes that the system is safe and returns the verdict along with the ARG.

Algorithm 6 Parallel lazy abstraction algorithm

with domains $\mathbf{D}_c = \langle \mathcal{S}, \mathcal{D}_c, \sqsubseteq_c, \gamma_c, \overline{post} \rangle$ and $\mathbf{D}_a = \langle \mathcal{S}, \mathcal{D}_a, \sqsubseteq_a, \gamma_a, \widetilde{post} \rangle$, and the proves operator $\vdash \subseteq \mathcal{D}_c \times \mathcal{D}_a$

Require: ARG $= \langle N, E, C, n_0, d_c, d_a, e \rangle$ is an initialized ARG, where $N = \{n_0\}$, $E = \emptyset$, $C = \emptyset$, $\gamma_c(d_c(n_0)) = \bigcup_{s_0 \in \mathcal{S}_0} s_0$ and $d_c(n_0) \vdash d_a(n_0)$

1: **function** CHECK(ARG, $\mathcal{T}$, *error*, $thread_{\#}$)
2: $tp \leftarrow$ INITTHREADPOOL($thread_{\#}$)
3: $waitlist \leftarrow N$, $expanded \leftarrow \emptyset$, $futures \leftarrow \emptyset$
4: **while** $waitlist \neq \emptyset$ or $futures \neq \emptyset$ **do**
5: **while** $n \in waitlist$ for some n **do**
6: $waitlist \leftarrow waitlist \backslash \{n\}$
7: $f_{v,w,n} \leftarrow$ SUBMITTASK$_{tp}(v, w, n \leftarrow$CHECKNODE(ARG, $\mathcal{T}$, *error*, n))
8: $futures \leftarrow futures \cup f_{v,w,n}$
9: $f_{v,w,n} \leftarrow$ WAITFORANY$_{tp}(futures)$
10: $futures \leftarrow futures \backslash \{f_{v,w,n}\}$
11: **if** $v =$ unsafe **then**
12: **return** unsafe, ARG
13: **else**
14: $waitlist \leftarrow waitlist \cup w$
15: $expanded \leftarrow expanded \cup n$
16: **return** safe, ARG

The algorithm uses the method CHECKNODE (Algorithm 7) for assessing individual nodes in parallel. It begins by acquiring a write lock on the node n, indicating that it is the only verification task that is allowed to read or write the labels of the node. It then checks if the node n has already been expanded (*expanded*) or if it has been marked as covered. If either condition holds, the algorithm releases the

lock and returns, indicating that the processing of this node is complete.

Next, the algorithm checks if the node represents an error in the abstract state space. If so, it releases the lock and returns that the node is unsafe.

If neither of the aforementioned conditions holds, the algorithm initializes sets to track the nodes to be added to the waitlist later (*waitlist*) and nodes that have been newly expanded (*newlyExpanded*). It then attempts to cover the node n and checks if it has been successfully covered. If not, it proceeds to expand the node.

After the node has been processed, the algorithm releases the write lock on the node n and returns that processing is done along with the updated sets of nodes waiting to be processed and newly expanded nodes.

The TryCover method (Algorithm 7) aims to cover a node n with another node n', ensuring that the abstraction of n' entails the abstraction of n. It iterates through the *expanded* set, acquiring read locks first, thereby allowing concurrent read access of multiple verification tasks (Observations 2 and 6). If $d_c(n) \vdash d_a(n')$, it stores the current abstract label of n' in n'_a, releases the read lock, and proceeds to cover by calling Cover. Releasing the read lock and saving the current abstract label in n'_a is necessary to avoid deadlocks: a node can be covered by one of its ancestors, and keeping the locks could result in a deadlock in the case of two refinements targeting the ancestors. After covering, TryCover reacquires the read lock for n' and checks if $d_a(n)$ is subsumed by $d_a(n')$, and establishing a cover relation on successful cover.

The methods Expand and InitAbstr can be taken from the sequential algorithm without any modification owing to Observations 1 and 4. Expand for any node n is called by CheckNode, and the verification task of CheckNode already owns a write lock for n at that point. Moreover, the *waitlist* and *expanded* sets of the Expand were created by the CheckNode method and only affect the global *waitlist* and *expanded* sets of Check, once CheckNode returns. It follows that no other verification task may want to interact with node n, as it is not considered expanded until Expand and CheckNode return. It also follows that no other verification task may want to interact with the newly created nodes, as they are not in the waitlist until Expand and CheckNode return. InitAbstr is only called by Expand on nodes CheckNode already owns a write lock for.

The methods Cover and Disable are part of the refinement and will be elaborated later.

We provide a proof for the soundness of the parallel lazy abstraction algorithm w.r.t, safety.

Proposition 5.1. *The parallel lazy algorithm yields an ARG that satisfies the properties for lazy abstraction.*

Proof. The initiation property trivially holds, as the Check algorithm requires an initialized ARG, where $\gamma_c(d_c(n_0)) = \bigcup_{s_0 \in \mathcal{S}_0} s_0$ and $d_c(n_0) \vdash d_a(n_0)$.

CheckNode starts by acquiring a write lock for node n, depicting that the abstract labels for that node can only be read and modified by the process of CheckNode.

Algorithm 7 Parallel processing of a node

1: **function** CheckNode(ARG, $\mathcal{T}$, *error*, n, *expanded*)
2: WriteLock(n)
3: **if** $n \in$ *expanded* or n is covered **then**
4: WriteUnlock(n)
5: **return** done, $\emptyset$, $\emptyset$
6: **if** $error(d_a(n))$ **then**
7: WriteUnlock(n)
8: **return** unsafe, $\emptyset$, $\emptyset$
9: $waitlist \leftarrow \emptyset$, $newlyExpanded \leftarrow \emptyset$
10: TryCover(ARG, *waitlist*, *expanded*, n)
11: **if** n is not covered **then**
12: Expand(ARG, *waitlist*, *newlyExpanded*, $\mathcal{T}$, n)
13: WriteUnlock(n)
14: **return** done, *waitlist*, *newlyExpanded*

Require: WriteLock(n) acquired
15: **function** TryCover(ARG, *waitlist*, *expanded*, n)
16: **for all** $n' \in$ *expanded* **do**
17: ReadLock(n')
18: **if** $d_c(n) \vdash d_a(n')$ **then**
19: $n'_a \leftarrow d_a(n')$
20: ReadUnlock(n')
21: Cover(ARG, *waitlist*, n, n'_a)
22: ReadLock(n')
23: **if** $d_a(n) \sqsubseteq_a d_a(n')$ **then**
24: $C \leftarrow C \cup \{\langle n, n' \rangle\}$
25: ReadUnlock(n')
26: **return**
27: ReadUnlock(n')

Require: WriteLock(n) acquired
28: **function** Expand(ARG, *waitlist*, *expanded*, $\mathcal{T}$, n)

Ensure: returns d''_a such that $d'_a \sqsubseteq_a d''_a$ with $\{d'_a\} = \widetilde{post}_t(d_a(n))$
29: **function** InitAbstr(ARG, n, t)

Require: WriteLock(n) acquired
Require: $d_c(n) \vdash n'_a$
30: **function** Cover(ARG, *waitlist*, n, n'_a)

Require: WriteLock(n) acquired
Require: $\overline{post}_t(d_c(n)) = \emptyset$
Ensure: $\widetilde{post}_t(d_a(n)) = \emptyset$
31: **function** Disable(ARG, *waitlist*, n, t)

First, TryCover tries to establish a covering relation. To this end, a read lock is acquired for n', and the abstract label of n' is read and stored in the memory of the current process (n'_a). The read lock ensures that no other process modifies the abstract label at that point, and then the read lock is released to prevent deadlocks in the algorithm. Assuming that Cover satisfies the properties of lazy abstraction, we can show that the coverage property holds: after the Cover method returns, the read lock is reacquired for node n', and the coverage relation is only established if $d_a(n) \sqsubseteq_a d_a(n')$.

If TryCover does not cover the node, Expand will expand it. Expand requires a write lock for node n, which is satisfied by CheckNode. Moreover, as Expand does not insert nodes into the *waitlist* or *expanded* sets, other processes will not be able to work on the newly expanded nodes until Expand finishes and CheckNode returns. It follows that assuming Disable satisfies the properties of lazy abstraction, the inductive labeling, concrete labeling, and simulation properties hold in accordance with Proposition 4.1. □

Proposition 5.2. *If there is a run σ of the automaton to a state $s \in \mathcal{S}$ and the parallel lazy abstraction algorithm returns a safe result, then there is a non-covered node n in the resulting ARG such that $s \in \gamma_a(d_a(n))$.*

Proof. The algorithm returns a safe result only when the waitlist and futures are empty, and therefore, all nodes are either expanded or covered, i.e., the ARG is complete. The above proposition is then a consequence of Lemma 4.1 and Proposition 5.1. □

5.3.2 Parallel Refinement of the Abstraction

Abstraction refinement is performed in the following two scenarios: either when attempting to establish a new covering relation or when a transition is disabled. In both cases, the abstract labels will be adjusted along the trace to the root of the ARG.

First of all, the methods of Cover, Disable, and Strengthen (Algorithm 8) need to be adapted from the sequential algorithm.

The arguments of Cover are modified to fit the parallel version of TryCover: instead of the covering node, it takes the abstract label of the covering node as its last parameter. Disable is the same as it was in the sequential case, as it is only called by Expand on nodes it already acquired the write lock for. Both Cover and Disable use Block, which will be defined later.

Finally, Strengthen is adapted for the parallel version. It is called by Block, which in turn is only called on nodes for which the write lock had already been acquired. It iterates through and removes the existing cover relation with the node in question. This deviation from the sequential algorithm is necessary, as checking the cover relations would require acquiring read locks, which could lead to a deadlock situation.

Both backward and forward binary interpolation have been adapted for the parallel lazy abstraction algorithm, taking into account Observations 3 and 5:

- In the case of backward binary interpolation (Algorithm 9), BLOCK_{BW} starts by already having a write-lock for n. First, the abstract label of the node is refined by a suitable interpolant, the node is strengthened, then a write lock is acquired for the parent node, and the pre-image of the complement of the interpolant is removed from the abstract label of the (already locked) parent node via a recursive call to BLOCK_{BW}.
- In the case of forward binary interpolation (Algorithm 10), BLOCK_{FW} also starts by already having a write-lock for n. In contrast to the backward version, the algorithm starts with acquiring a write lock for the parent node, then recursively removing the pre-image of B from the abstract label of the parent. Finally, the interpolant is calculated, and n is strengthened.

Proposition 5.3. *The backward propagation algorithm preserves the ARG properties.*

Proof. The parallel version of the backward propagation algorithm preserves the initiation, concrete labeling, simulation, and coverage properties the same way as its sequential counterpart (Proposition 4.3).

Moreover, we can also show that the labeling also stays inductive reusing the proof of the sequential backward propagation algorithm showing that $d'_p \sqsubseteq_a d_a(n)$ is preserved, where $\{d'_p\} = \widetilde{post}_{e(\langle p,n\rangle)}(d_a(p))$ is the abstract successor of p, the parent node of n, assuming that the abstract labels along the path are not read or modified while the propagation takes place. BLOCK_{BW} requires a write lock being acquired for node n before calling, and the recursive call for p only happens for nodes p for which a write lock has already been acquired, which prevents the reading and modification of the node's abstract label. As the write lock is not released until the backward propagation and strengthening of the parent is finished, the process of the backward propagation algorithm will own the write-locks for all the nodes from n until the root of the ARG (or until $d_a(n) \not\downarrow B$). At that point, no other process can read or modify any information from which the interpolants are calculated, thus $d'_p \sqsubseteq_a d_a(n)$ is preserved along the trace and the labeling stays inductive. □

Proposition 5.4. *The forward propagation algorithm preserves the ARG properties.*

Proof. The parallel version of the forward propagation algorithm preserves the initiation, concrete labeling, simulation, and coverage properties the same way as its sequential counterpart (Proposition 4.5).

Similarly, the labeling also stays inductive. BLOCK_{FW} requires a write lock being acquired for node n and always acquires write locks for nodes before a recursive call on those. Thus, the process of the forward propagation algorithm will own the write lock for all nodes from n until the root of the ARG (or until $d_a(n) \not\downarrow B$). At that point, no other process can read or modify any information from which the interpolants are calculated. The write locks are only released once the information needed to calculate the interpolant is read. As the write lock is released

on the parent, other processes may modify the abstract label of the parent, but these modifications do not affect the inductive labeling on the current trace. Thus $d'_p \sqsubseteq_a d_a(n)$ is preserved along the trace and the labeling stays inductive. □

Proposition 5.5. *The backward and forward propagation algorithms are consistent with their contract.*

Proof. The consistency of the contract of the backward propagation is a direct consequence of Propositions 5.3 and 4.4, while the consistency of the contract of the forward propagation algorithm is a direct consequence of 5.4 and 4.6. □

Algorithm 8 Parallel lazy abstraction refinement algorithms

Require: WriteLock(n) acquired
1: **function** Cover(ARG, *waitlist*, n, n'_a)
2: **for all** $B \in \neg\gamma_{a\to i}(n'_a)$ **do**
3: Block(ARG, *waitlist*, n, B)

Require: WriteLock(n) acquired
4: **function** Disable(ARG, *waitlist*, n, t)
5: **for all** $B \in \widehat{pre}_t(\gamma_{a\to i}(\top))$ **do**
6: Block(ARG, *waitlist*, n, B)

Require: WriteLock(n) acquired
Require: $d_c(n) \not\leq B$
Ensure: $d_a(n) \not\leq B$
7: **function** Block(ARG, *waitlist*, n, B)

Require: WriteLock(n) acquired
Ensure: $d_a(n) \sqsubseteq_a I$
8: **function** Strengthen(ARG, *waitlist*, n, I)
9: $d_a(n) \leftarrow d_a(n) \sqcap_a I$
10: **for all** $\langle n_{cov}, n\rangle \in C$ **do**
11: $C \leftarrow C\backslash\{\langle n_{cov}, n\rangle\}$
12: $waitlist \leftarrow waitlist \cup \{n_{cov}\}$

5.4 Discussion

We developed the parallel lazy abstraction algorithm to enhance the performance of lazy abstraction on modern computational platforms. In this section, we would like to provide a discussion on the different trade-offs that stem from the way in which we constructed the parallel algorithm.

Trade-off 1 (Expanding over covering)**.** The first trade-off was identified in the covering operation. Both the sequential and parallel versions of TryCover (Algorithms 1 and 7) are constructed to iterate over the set of already expanded nodes

Algorithm 9 Backward propagation of abstraction refinement for the parallel lazy algorithm

Require: $d_c(n) \not\downarrow B$
Require: WRITELOCK(n) acquired
Ensure: $d_a(n) \not\downarrow B$
Ensure: the labeling of nodes satisfies the properties of an ARG
1: **function** BLOCK$_{BW}$(ARG, *waitlist*, n, B)
2: **if** $d_a(n) \not\downarrow B$ **then**
3: **return**
4: $I \leftarrow itp(\gamma_{c\to a}(d_c(n)), B)$
5: STRENGTHEN(ARG, *waitlist*, n, I)
6: **if** $\langle p, n\rangle \in E$ for some p **then**
7: WRITELOCK(p)
8: **for all** $B' \in \neg\, \gamma_{a\to i}(I)$ **do**
9: **for all** $B'_{pre} \in \widehat{pre}_{e(\langle p,n\rangle)}(B')$ **do**
10: BLOCK$_{BW}$(ARG, *waitlist*, p, B'_{pre})

Algorithm 10 Forward propagation of abstraction refinement for the parallel lazy algorithm

Require: $d_c(n) \not\downarrow B$
Require: WRITELOCK(n) acquired
Ensure: $d_a(n) \not\downarrow B$
Ensure: $d_a(n) \vdash I$
1: **function** BLOCK$_{FW}$(ARG, *waitlist*, n, B)
2: **if** $d_a(n) \not\downarrow B$ **then**
3: **return** $d_a(n)$
4: **if** $\langle p, n\rangle \in E$ for some p **then**
5: WRITELOCK(p)
6: $A \leftarrow \top$
7: **for all** $B_{pre} \in \widehat{pre}_{e(\langle p,n\rangle)}(B)$ **do**
8: $I' \leftarrow$ BLOCK$_{FW}$(ARG, *waitlist*, p, B_{pre})
9: $\{d\} \leftarrow \widetilde{post}_{e(\langle p,n\rangle)}(I')$
10: $A \leftarrow A \sqcap_a d$
11: WRITEUNLOCK(p)
12: **else**
13: $A \leftarrow \gamma_{c\to a}(d_c(n))$
14: $I \leftarrow itp(A, B)$
15: STRENGTHEN(ARG, *waitlist*, n, I)
16: **return** I

to try to find a suitable covering node. In the case of the sequential variant, the following is true for each node: either an already expanded node covers it, or it must be expanded. However, in the case of the parallel TryCover, a third possibility arises: another node that is actively being expanded is a suitable candidate for covering the node in question. If this third option holds, the parallel version of TryCover will ignore the node, as this node is not yet expanded (Expand is not yet finished), so its abstract labels might change due to a refinement started by Disable; as a result, TryCover will not be able to establish a covering relation, and the node will be expanded instead. In summary, the parallel algorithm prefers expanding nodes rather than waiting for other verification tasks to finish. Waiting for other verification tasks would enable the algorithm to check if a cover relation can be established with the nodes the other tasks expand. However, choosing not to wait might lead to the parallel algorithm having to perform extra, unnecessary calculations.

Trade-off 2 (Reestablishing instead of keeping cover relations)**.** The second trade-off we identified comes from the parallel implementation of Strengthen (Algorithm 8). Compared to the sequential implementation (Algorithm 3), the parallel version removes all covering relations and puts the covered nodes back on the wait-list. This behavior is necessary, as Strengthen cannot acquire arbitrary locks in the ARG without risking deadlocks. However, removing all covering relations causes extra computations to be necessary. In the sequential case, the current covering relation can be checked to see if it holds, and exploring all expanded nodes is only necessary if the cover relation has to be removed. In contrast, the parallel algorithm has to recheck every expanded node for each cover relation and reestablish those that still hold.

Trade-off 3 (Blocking locks)**.** Finally, the third trade-off stems from the parallel algorithm using thread pools and locks (Algorithm 6). The ARG is a tree-like structure, and every refinement modifies the abstract labels along the trace from a node to the root of the ARG. It follows that different refinements may overlap and target the same nodes (at least the root of the ARG). If refinements overlap and they target the same node at the *same time*, the locking mechanism in the parallel refinement algorithm (Algorithms 9 and 10) will cause the verification task and thread of one of the refinements to wait and do nothing before it can acquire the lock. An alternative approach would be to use coroutines (or virtual threads) that are able to suspend their execution if they have to wait for a resource, allocate the underlying thread to another task, and resume execution once the lock can be acquired, thus always utilizing a thread when possible. However, suspending and resuming coroutines also have an overhead [11]. Nonetheless, the binary interpolation strategies do not necessarily iterate over the whole trace to the root, only until it is necessary, reducing the likelihood that the same nodes are targeted at the same time in a big enough ARG. Thus, we opted to favor thread pools.

6 Evaluation

We implemented the proposed techniques in the Theta open-source model checking framework [32]. We evaluated the prototype to study its scalability and compare its performance with that of the sequential and parallel lazy abstraction algorithms.

6.1 Setup and Configurations

We evaluated the algorithms on a benchmark set[1] [17], a collection containing 57 UPPAAL XTA models from various well-known sources. The benchmark set contains models that are regularly used to evaluate well-known tools, such as UPPAAL [20], CosyVerif [2], PAT [30] or MCTA [36], and problems from industrial case studies [14, 29] as well.

Measurements were executed using BenchExec[2] with each task limited to a configuration-dependent number of CPU cores, 20 GB of memory, and 100 seconds of runtime. The evaluation was performed on the E-cores of an Intel Core i7-1360P processor to avoid the effect hyperthreading might have on the results. The 20 GB memory limit was determined empirically: it was raised until none of the tasks terminated due to running out of memory. Other system-level modifications, like disabling swapping, were employed according to the instructions of BenchExec.

We used explicit value abstraction with location abstraction for data abstraction and zone abstraction with location abstraction for handling time. As for refinement, we conducted the experiments with both backward (BW) and forward (FW) binary interpolation. Since we have not identified any previously existing parallel lazy implementation, we have compared the performance of the parallel implementation to our sequential one. The algorithms were evaluated in the following configurations:

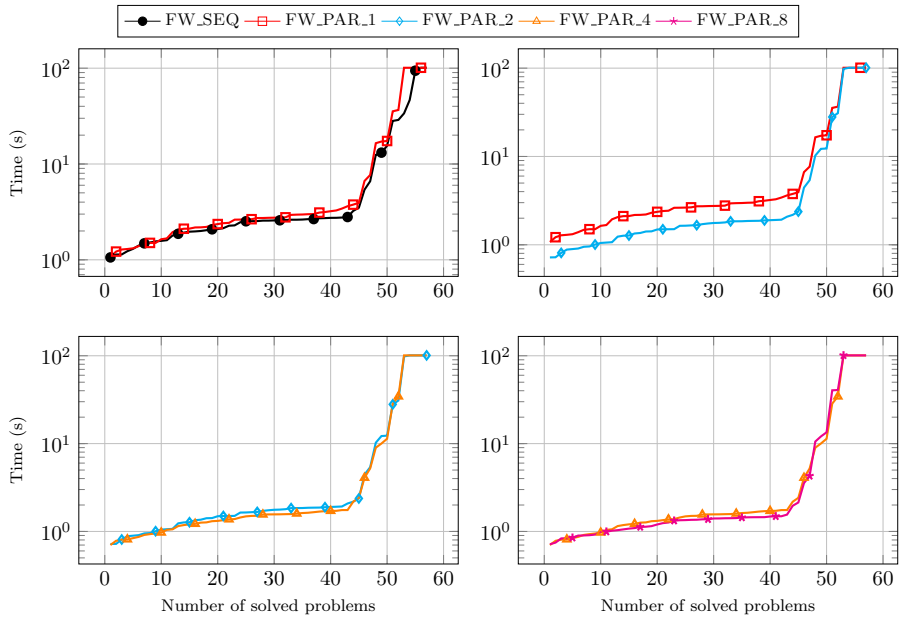
- The sequential algorithm (Algorithm 1) referred to as SEQ onward.
- The parallel algorithm (Algorithm 6) limited to running on one (PAR_1), two (PAR_2), four (PAR_4) and eight (PAR_8) CPU cores, with $thread_{\#}$ being set to respectively 1, 2, 4 and 8.

The experiment was repeated five times for each configuration, and the final result was derived by dropping the best and worst measurements and taking the average of the remaining three.

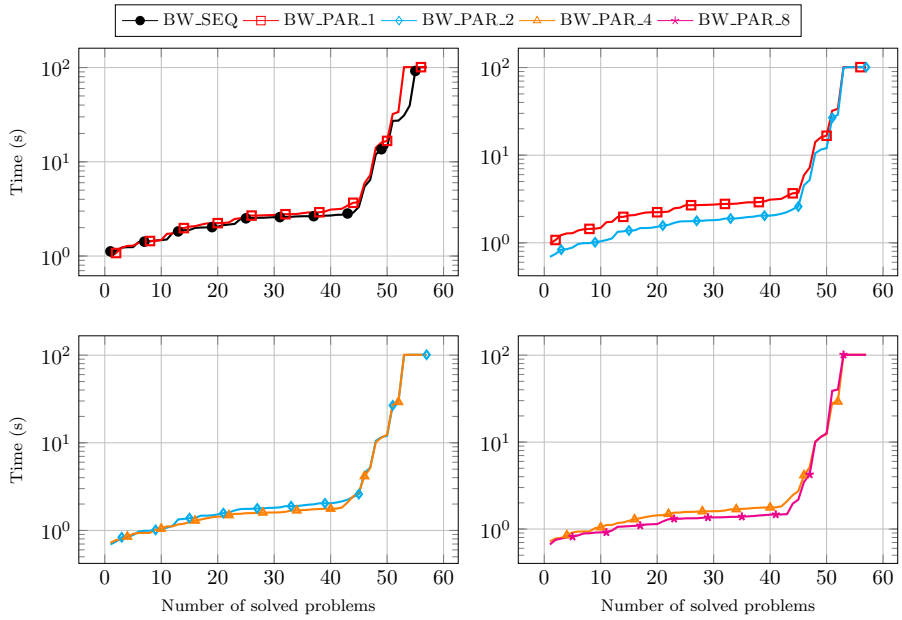
6.2 Results and Discussion

Figure 10 contains the quantile plots comparing the performance of the sequential and parallel configurations. The x-axis describes the number of solved tasks, while the y-axis shows the time required to solve the given number of tasks. With both refinement strategies, the sequential algorithm (FW_SEQ, BW_SEQ) was able to

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



(a) Forward binary interpolation



(b) Backward binary interpolation

Figure 10: Quantile plots describing the performance of the algorithms

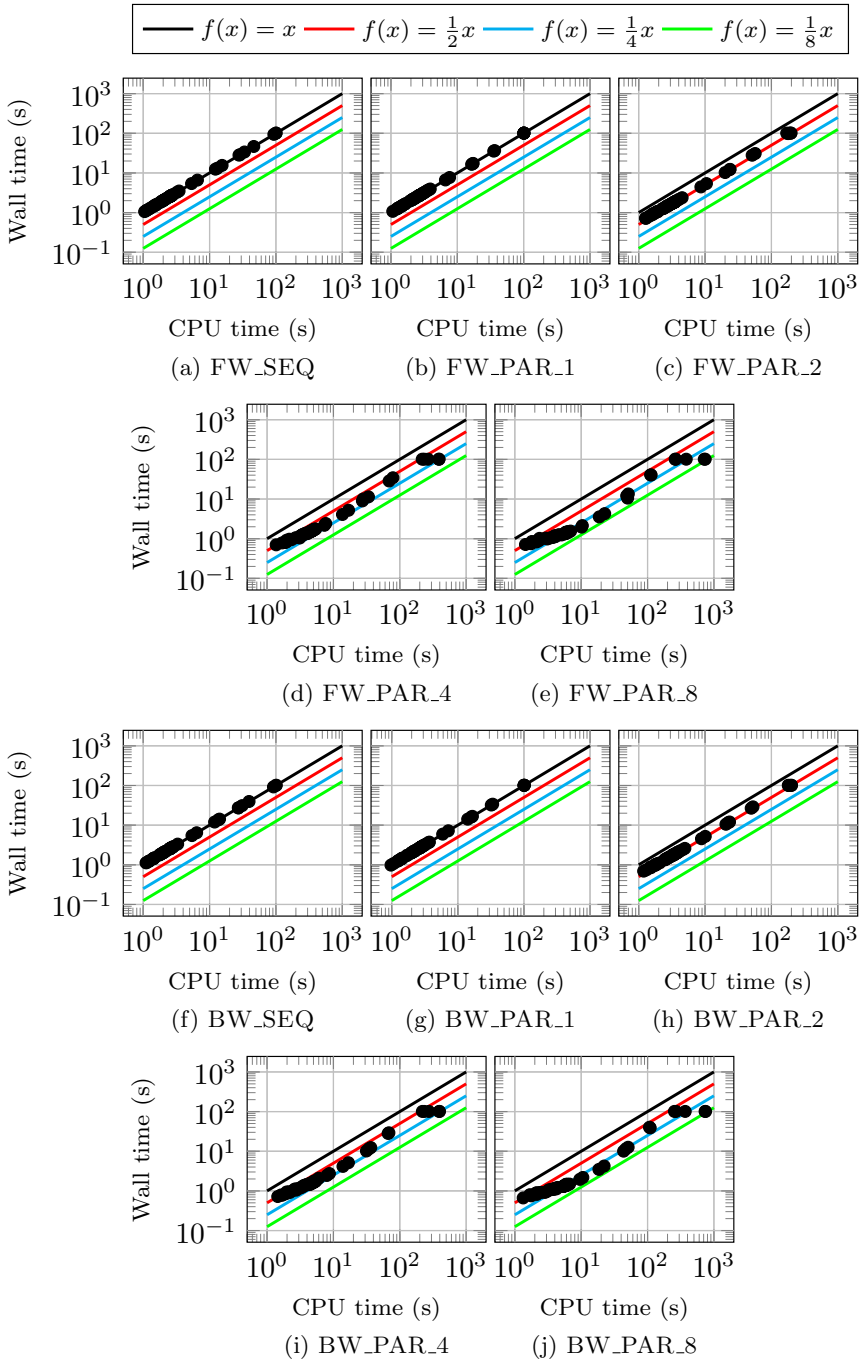


Figure 11: Scatter plots describing the relationship between CPU time and wall time

solve 55 problems, the parallel algorithm on two cores (FW_PAR_2, BW_PAR_2) 53 problems, while the rest of the configurations 52 problems.

In the case of problems that can be solved under 5 seconds, there is a clear trend: the parallel algorithm on eight cores (FW_PAR_8, BW_PAR_8) outperforms the parallel algorithm on four cores (FW_PAR_4, BW_PAR_4), which in turn performs better than the parallel algorithm on two cores (FW_PAR_2, BW_PAR_2). The worst performing configurations are the parallel algorithm on one core (FW_PAR_1, BW_PAR_1), which is slightly outperformed by the sequential algorithm (FW_SEQ, BW_SEQ). This latter comparison is shown to establish the performance hit that the upkeeping of the thread pool and the usage of locks might incur.

However, the situation is less evident in the case of problems requiring more than 5 seconds to be solved. Generally, the performance of the parallel algorithm on eight cores (FW_PAR_8, BW_PAR_8) falls back, and the parallel variants on four cores (FW_PAR_4, BW_PAR_4) come out on top, so questions about scalability arise. An explanation for this phenomenon could arise from Trade-off 1: the more cores the algorithm runs on, the less the number of expanded nodes is available for checking whether it is suitable for a cover relation. Fewer cover relations lead to more nodes to expand. This can also explain why there are problems that the sequential algorithm can solve but the parallel algorithm cannot: by running the operations of lazy abstraction in parallel, some cover relations might never be established, and the parallel algorithm is instead forced to expand a node that could have been covered. Unfortunately, even if a suitable candidate for cover is expanded later on, an expanded node will not be revisited to check coverage again.

Analyzing the scalability further, Figure 11 depicts how the CPU time is proportional to the wall time for each problem. The wall time was measured to be the length of the experiment, while the CPU time is the sum of times that each core spends actively working. In ideal conditions, the CPU time should equal the wall time times the number of cores, but technical (inherent multithreaded nature of JVM) and theoretical (Amdahl's law) limitations exist. It can be seen that in the case of the sequential algorithm (FW_SEQ, BW_SEQ) and the parallel algorithm on one core (FW_PAR_1, BW_PAR_1) the CPU time almost matches the wall time as expected. Moreover, in the case of the parallel algorithm on two cores (FW_PAR_2, BW_PAR_2), the data points primarily lie on the line of the theoretical limit of scalability for two cores ($f(x) = 1/2x$), indicating that algorithm was able to take advantage of both cores. However, the situation is radically different with the parallel algorithm on 4 (FW_PAR_4, BW_PAR_4) and on eight cores (FW_PAR_8, BW_PAR_8). While the data points lie south of the scalability limit for two cores ($f(x) = 1/2x$), many of them are well north of the theoretical limit of scalability for their respective number of cores ($f(x) = 1/4x$ and $f(x) = 1/8x$), making it clear, that the algorithm cannot fully take advantage of the extra cores in case of tasks that can be solved quickly.

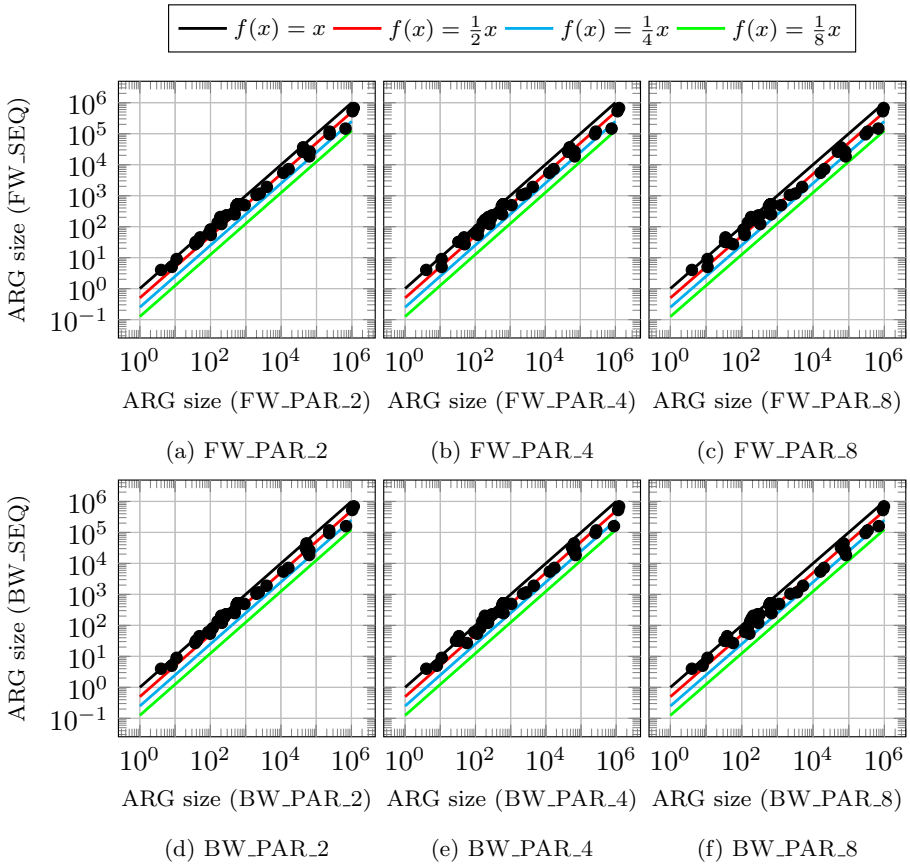
Nevertheless, the problems that timeout (on the $f(x) = 100$ line) provide an interesting insight into the scalability. Given that the problems take advantage of

more than two cores, but not all can take advantage of 4 or 8, it can be concluded that the scalability depends on the algorithm and the model as well. The different structures of the models might allow for a lesser degree of parallelization. At the same time, the structure could also contribute to having to do more conflicting refinements, causing more waiting to acquire locks. As per Trade-off 3, waiting for locks is a blocking operation leading to the observed behavior.

Finally, Figure 12 shows the ARG size of the parallel algorithms compared to the sequential algorithm illustrating the space overhead. Although it can be observed that increasing the number of cores slightly increases the overall ARG size as expected after Trade-off 1, there is no clear correlation between the number of cores and the relative size of the ARG. One possible factor for this behavior is the different structure of the problem under verification: the algorithm can take advantage of problems with more independent branches as there are fewer conflicting refinements. Another possible factor stems from Trade-off 2: the parallel algorithm has to reestablish many cover relations, thus spending time on operations that do not expand the ARG.

To decide how the factors contribute to the phenomenon, we depicted the relative size of the ARG compared to the sequential algorithm and compared it to the ratio of CPU time and wall time (Figure 13). Based on the figures, it can be concluded that for two cores (FW_PAR_2, BW_PAR_2), the model-dependent factor prevails, and with increasing the number of cores, Trade-off 2 starts to dominate.

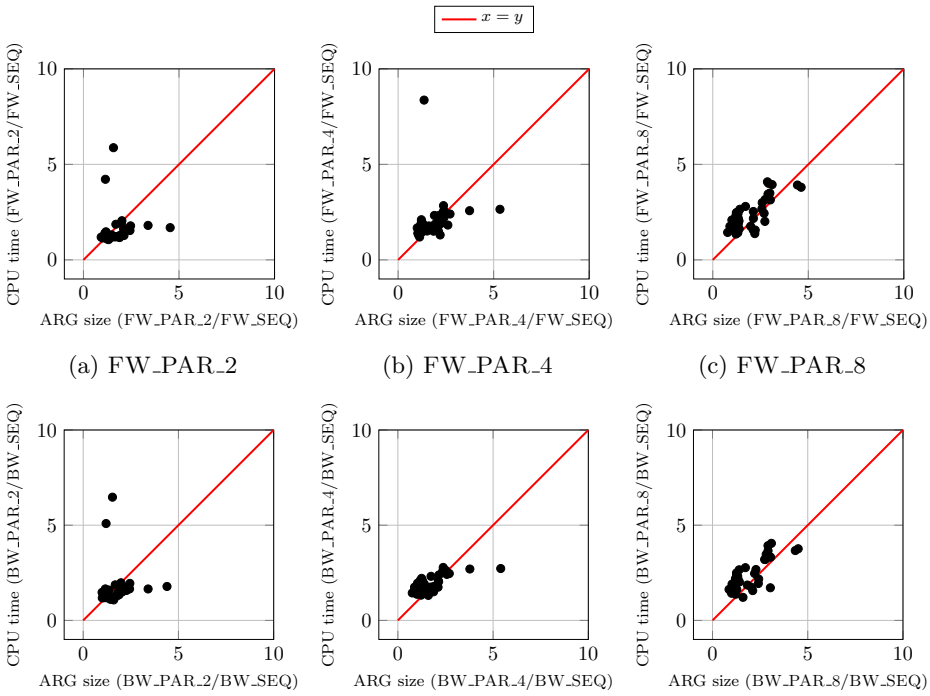
In conclusion, the experiments showed that the parallel lazy abstraction algorithm outperforms the sequential algorithm. In terms of scalability, the algorithm is able to take advantage of two cores in almost all cases. However, although the algorithm can also take advantage of more than two cores, its scalability is limited. Our evaluation showed that by increasing the number of cores, the trade-offs we made regarding the cover relations are also starting to affect the performance: the more the number of cores, the fewer cover relations the algorithm can establish and the more time the algorithm has to spend reestablishing cover relations. Moreover, we also showed that the scalability depends on the model under verification as well.



(a) FW_PAR_2 (b) FW_PAR_4 (c) FW_PAR_8

(d) BW_PAR_2 (e) BW_PAR_4 (f) BW_PAR_8

Figure 12: Scatter plots describing the relationship between the size of the ARGs compared to the sequential algorithm



(a) FW_PAR_2 (b) FW_PAR_4 (c) FW_PAR_8

(d) BW_PAR_2 (e) BW_PAR_4 (f) BW_PAR_8

Figure 13: Scatter plots describing the relationship between the relative size of the ARGs compared to the sequential algorithm and the CPU time compared to wall time

7 Conclusion

This paper has explored the sequential and parallel implementations of lazy abstraction strategies for verifying timed systems, a critical task in ensuring the reliability of real-time applications. We provided a generic lazy abstraction framework for the sequential implementations of lazy abstraction. We analyzed the behavior of the sequential algorithm to design a parallel algorithm that leverages modern multi-core architectures and offers a promising avenue to accelerate the verification process.

Through a detailed empirical evaluation of benchmark timed systems, we have demonstrated that the parallel algorithm can enhance performance compared to its sequential counterpart. However, the study also highlights the challenges associated with parallelization, such as recalculating previous cover relations, which can impact the overall scalability of the approach on higher core counts.

The findings provide valuable insights into lazy abstraction, improving the verification of complex timed systems and ultimately enhancing the safety and reliability of critical real-time systems. In the future, we plan to address the limitations of

scalability by trying to preserve more cover relations during strengthening and by working out a methodology to increase the number of cover relations in the ARG.

References

[1] Alur, R. Timed automata. In *Proceedings of the 11th International Conference on Computer Aided Verification*, pages 8–22. Springer, 1999. DOI: 10.1007/3-540-48683-6_3.

[2] André, É., Lembachar, Y., Petrucci, L., Hulin-Hubard, F., Linard, A., Hillah, L., and Kordon, F. CosyVerif: An open source extensible verification environment. In *Proceedings of the 18th International Conference on Engineering of Complex Computer Systems*, pages 33–36. IEEE, 2013. DOI: 10.1109/ICECCS.2013.15.

[3] Barnat, J., Brim, L., and Chaloupka, J. Parallel breadth-first search LTL model-checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 106–115, 2003. DOI: 10.1109/ASE.2003.1240299.

[4] Barnat, J., Bloemen, V., Duret-Lutz, A., Laarman, A., Petrucci, L., van de Pol, J., and Renault, E. *Parallel Model Checking Algorithms for Linear-Time Temporal Logic.* In *Handbook of Parallel Constraint Reasoning*, pages 457–507. Springer International Publishing, Cham, 2018. DOI: 10.1007/978-3-319-63516-3_12.

[5] Behrmann, G., Bouyer, P., Larsen, K. G., and Pelánek, R. Lower and upper bounds in zone-based abstractions of timed automata. *International Journal on Software Tools for Technology Transfer*, 8(3):204–215, 2006. DOI: 10.1007/978-3-540-24730-2_25.

[6] Bengtsson, J. and Yi, W. *Timed Automata: Semantics, Algorithms and Tools.* In *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, pages 87–124. Springer, 2004. DOI: 10.1007/978-3-540-27755-2_3.

[7] Beyer, D., Haltermann, J., Lemberger, T., and Wehrheim, H. Decomposing software verification into off-the-shelf components: An application to CEGAR. In *Proceedings of the IEEE/ACM 44th International Conference on Software Engineering*, pages 536–548, 2022. DOI: 10.1145/3510003.3510064.

[8] Beyer, D. and Löwe, S. Explicit-state software model checking based on CEGAR and interpolation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 146–162. Springer, 2013. DOI: 10.1007/978-3-642-37057-1_11.

[9] Beyer, D. and Podelski, A. *Software Model Checking: 20 Years and Beyond.* In *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on*

the Occasion of His 60th Birthday, pages 554–582. Springer Nature Switzerland, Cham, 2022. DOI: 10.1007/978-3-031-22337-2_27.

[10] Bingham, B., Bingham, J., Paula, F. M. d., Erickson, J., Singh, G., and Reitblatt, M. Industrial strength distributed explicit state model checking. In *Proceedings of the Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology*, pages 28–36, 2010. DOI: 10.1109/PDMC-HiBi.2010.13.

[11] Buhr, P. A. *Coroutine*. In *Understanding Control Flow: Concurrent Programming Using μC++*, pages 125–190. Springer International Publishing, Cham, 2016. DOI: 10.1007/978-3-319-25703-7_4.

[12] Burns, E. and Zhou, R. Parallel model checking using abstraction. In Donaldson, A. and Parker, D., editors, *Model Checking Software*, pages 172–190, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-31759-0_13.

[13] Clarke, E. M., Henzinger, T. A., Veith, H., and Bloem, R., editors. *Handbook of Model Checking*. Springer, 2018. DOI: 10.1007/978-3-319-10575-8.

[14] David, A., Illum, J., Larsen, K. G., and Skou, A. Model-based framework for schedulability analysis using UPPAAL 4.1. In *Model-based design for embedded systems*, pages 117–144. CRC Press, 2018. DOI: 10.1201/9781315218823-12.

[15] Edelkamp, S. and Sulewski, D. Efficient explicit-state model checking on general purpose graphics processors. In van de Pol, J. and Weber, M., editors, *Model Checking Software*, pages 106–123, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-16164-3_8.

[16] Evangelista, S., Petrucci, L., and Youcef, S. Parallel nested depth-first searches for ltl model checking. In Bultan, T. and Hsiung, P.-A., editors, *Automated Technology for Verification and Analysis*, pages 381–396, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-24372-1_27.

[17] Farkas, R. and Bergmann, G. Towards reliable benchmarks of timed automata. In *25th Mini-Symposium of BME MIT*, pages 20–23. Budapest University of Technology and Economics, 2018. URL: http://real.mtak.hu/id/eprint/79561.

[18] Fuess, M. E., Leeser, M., and Leonard, T. An FPGA implementation of explicit-state model checking. In *Proceedings of the 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 119–126, 2008. DOI: 10.1109/FCCM.2008.36.

[19] Garavel, H., Mateescu, R., and Smarandache, I. Parallel state space construction for model-checking. In Dwyer, M., editor, *Model Checking Software*,

pages 217–234, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. DOI: 10.1007/3-540-45139-0_14.

[20] Hendriks, M., Yi, W., Petterson, P., Hakansson, J., Larsen, K., David, A., and Behrmann, G. UPPAAL 4.0. In *Proceedings of the Third International Conference on the Quantitative Evaluation of Systems*, pages 125–126. IEEE, 2006. DOI: 10.1109/QEST.2006.59.

[21] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70, 2002. DOI: 10.1145/503272.503279.

[22] Herbreteau, F., Srivathsan, B., and Walukiewicz, I. Lazy abstractions for timed automata. In *Proceedings of the International Conference on Computer Aided Verification*, Volume 8044 of *Lecture Notes in Computer Science*, pages 990–1005. Springer, 2013. DOI: 10.1007/978-3-642-39799-8_71.

[23] Inverso, O. and Trubiani, C. Parallel and distributed bounded model checking of multi-threaded programs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 202–216, New York, NY, USA, 2020. Association for Computing Machinery. DOI: 10.1145/3332466.3374529.

[24] Jones, M., Mercer, E. G., Bao, T., Kumar, R., and Lamborn, P. Benchmarking explicit state parallel model checkers. *Electronic Notes in Theoretical Computer Science*, 89(1):84–98, 2003. DOI: 10.1016/S1571-0661(05)80098-2.

[25] Kumar, R. and Mercer, E. G. Load balancing parallel explicit state model checking. *Electronic Notes in Theoretical Computer Science*, 128(3):19–34, 2005. DOI: 10.1016/j.entcs.2004.10.016.

[26] Laarman, A., Langerak, R., van de Pol, J., Weber, M., and Wijs, A. Multi-core nested depth-first search. In Bultan, T. and Hsiung, P.-A., editors, *Automated Technology for Verification and Analysis*, pages 321–335. Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-24372-1_23.

[27] McMillan, K. L. Lazy abstraction with interpolants. In *Proceedings of the International Conference on Computer Aided Verification*, pages 123–136. Springer, 2006. DOI: 10.1007/11817963_14.

[28] Okano, K., Nagaoka, T., Tanaka, T., Sekizawa, T., and Kusumoto, S. Parallel multiple counter-examples guided abstraction loop applying to timed automaton. *International Journal of Informatics Society*, 8(2):103–116, 2016. URL: https://cir.nii.ac.jp/crid/1010282257267711104.

[29] Ravn, A. P., Srba, J., and Vighio, S. Modelling and verification of web services business activity protocol. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–371. Springer, 2011. DOI: 10.1007/978-3-642-19835-9_32.

[30] Sun, J., Liu, Y., Dong, J. S., and Pang, J. PAT: Towards flexible verification under fairness. In Bouajjani, A. and Maler, O., editors, *Computer Aided Verification*, pages 709–714. Springer Berlin Heidelberg, 2009.

[31] Tian, C., Duan, Z., and Duan, Z. Making CEGAR more efficient in software model checking. *IEEE Transactions on Software Engineering*, 40(12):1206–1223, 2014. DOI: 10.1109/TSE.2014.2357442.

[32] Tóth, T., Hajdu, A., Vörös, A., Micskei, Z., and Majzik, I. Theta: a framework for abstraction refinement-based model checking. In Stewart, D. and Weissenbacher, G., editors, *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179, 2017. DOI: 10.23919/FMCAD.2017.8102257.

[33] Tóth, T. and Majzik, I. Lazy reachability checking for timed automata using interpolants. In *Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systems*, Volume 10419 of *Lecture Notes in Computer Science*, pages 264–280. Springer, 2017. DOI: 10.1007/978-3-319-65765-3_15.

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

[35] Wang, W. and Jiao, L. Difference bound constraint abstraction for timed automata reachability checking. In Graf, S. and Viswanathan, M., editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 146–160, Cham, 2015. Springer International Publishing. DOI: 10.1007/978-3-319-19195-9_10.

[36] Wehrle, M. and Kupferschmid, S. Mcta: Heuristics and search for timed systems. In Jurdziński, M. and Ničković, D., editors, *Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systems*, pages 252–266. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-33365-1_18.

[37] Wijs, A., Neele, T., and Bosnacki, D. GPUexplore 2.0: Unleashing GPU explicit-state model checking. In *World Congress on Formal Methods*, 2016. DOI: 10.1007/978-3-319-48989-6_42.