Universal Invariant Checking of Parametric Systems with Quantifier-Free SMT Reasoning

Alessandro Cimatti, Alberto Griggio, and Gianluca Redondi

Fondazione Bruno Kessler
{cimatti, griggio, gredondi}@fbk.eu

Abstract. The problem of invariant checking in parametric systems – which are required to operate correctly regardless of the number and connections of their components – is gaining increasing importance in various sectors, such as communication protocols and control software. Such systems are typically modeled using quantified formulae, describing the behaviour of an unbounded number of (identical) components, and their automatic verification often relies on the use of decidable fragments of first-order logic in order to effectively deal with the challenges of quantified reasoning.

In this paper, we propose a fully automatic technique for invariant checking of parametric systems which does not rely on quantified reasoning. Parametric systems are modeled with array-based transition systems, and our method iteratively constructs a quantifier-free abstraction by analyzing, with SMT-based invariant checking algorithms for non-parametric systems, increasingly-larger finite instances of the parametric system. Depending on the verification result in the concrete instance, the abstraction is automatically refined by leveraging canditate lemmas from inductive invariants, or by discarding previously computed lemmas.

We implemented the method using a quantifier-free SMT-based IC3 as underlying verification engine. Our experimental evaluation demonstrates that the approach is competitive with the state of the art, solving several benchmarks that are out of reach for other tools.

Keywords: Parametric Systems · Array-based transitions systems · Abstraction-refinement · SMT

1 Introduction

Parametric systems consist of a finite but unbounded number of components. Examples include communication protocols (e.g. leader election), feature systems, or control algorithms in various application domains (e.g. railways interlocking logics). The key challenge is to prove the correctness of the parametric system for all possible configurations corresponding to instantiations of the parameters.

Parametric systems can be described as symbolic array-based transition systems [9], where the dependence on the configuration is expressed with first-order quantifiers in the initial condition and the transition relation of the model.

In this paper, we propose a fully automated approach for solving the universal invariant problem of array-based systems. The distinguishing feature is that the approach, grounded in SMT, does not require dealing with quantified theories, with obvious computational advantages. The algorithm implements an abstraction-refinement loop, where the abstract space is a quantifier-free transition system over some SMT theories. Our inspiration and starting point is the Parameter Abstraction of [3,14], which we extend in two directions. First, we modify the definition of the abstraction, by introducing a set of different *environment variables*, which intuitively overapproximate the behaviour of all the instances not precisely tracked by the abstraction, and by introducing a special *stuttering transition* in which the environment is allowed to change non-deterministically. Second, we combine the abstraction with a method for *automatically* inferring candidate universal lemmas, which are used to strengthen the abstraction in case of spurious counterexamples. The candidate lemmas are obtained by generalization from the spuriousness proof carried out in a finite-domain instantiation of the concrete system. However, we do not require quantified reasoning to prove that they universally hold; rather, the algorithm takes into account the fact that candidate lemmas may turn out not to be universally valid. In such cases, the method is able to automatically discover such bad lemmas and discard them, by examining increasingly-higher-dimension bounded instances of the parametric system.

We implemented the method in a tool called Lambda. At its core, Lambda leverages modern model checking approaches for quantifier-free infinite-state systems, i.e. the SMT-based approach of IC3 with implicit abstraction [4], in contrast to other approaches [18] where the abstract space is Boolean. In our experimental evaluation, we compared Lambda with the state-of-the-art tools MCMT [10] and Cubicle [6]. The results show the advantage of the approach, that is able to solve multiple benchmarks that are out of reach for its competitors.

The rest of the paper is structured as follows. In Section 2 we present some logical background, and in Section 3 we describe array-based systems. We give an informal overview of the algorithm in Section 4. In Section 5 we define the abstraction and state its formal properties. In Section 6 we discuss the approach to concretization and refinement, and we present the techniques for inferring candidate lemmas. We discuss the related work in Section 7, and we present our experimental evaluation in Section 8. Finally, in Section 9 we draw some conclusions and present directions for future work. For lack of space, the proofs of our theoretical results are reported in an Appendix.

2 Preliminaries

Our setting is standard first order logic. A theory $\mathcal{T}$ in the SMT sense is a pair $\mathcal{T} = (\Sigma, \mathcal{C})$, where Σ is a first order signature and $\mathcal{C}$ is a class of models over Σ. A theory $\mathcal{T}$ is closed under substructure if its class $\mathcal{C}$ of structures is such that whenever $\mathcal{M} \in \mathcal{C}$ and $\mathcal{N}$ is a substructure of $\mathcal{M}$, then $\mathcal{N} \in \mathcal{C}$. We use the standard notions of Tarskian interpretation (assignment, model, satisfiability, validity, logical consequence). We refer to 0-arity predicates as Boolean variables, and to 0-arity uninterpreted functions as (theory) variables. A literal is an atom or its negation. A clause is a disjunction of literals. A formula is in conjunctive normal form (CNF) iff it is a conjuction of clauses. If $x_1, ..., x_n$ are variables and ϕ is a formula, we might write $\phi(x_1, ..., x_n)$ to indicate that all the variables occurring free in ϕ are in $x_1, ..., x_n$.

If ϕ is a formula, t is a term and v is a variable which occurs free in ϕ, we write $\phi[v/t]$ for the substitution of every occurrence of v with t. If $\underline{t}$ and $\underline{v}$ are vectors of the same length, we write $\phi[\underline{v}/\underline{t}]$ for the simultaneous substitution of each v_i with the corresponding term t_i. We use an if-then-else notation for formulae. We write **if** ϕ_1 **then** ψ_1 **elif** ϕ_2 **then** ψ_2 **elif** $\ldots \psi_{n-1}$ **else** ψ_n to denote the formula $(\phi_1 \rightarrow \psi_1) \wedge ((\neg\phi_1 \wedge \phi_2) \rightarrow \psi_2) \wedge \ldots ((\neg\phi_1 \ldots \neg\phi_{n-1} \wedge \neg\phi_n) \rightarrow \psi_n)$.

Given a set of variables $\underline{v}$, we denote with $\underline{v}'$ the set $\{v' | v \in \underline{v}\}$. A symbolic transition system is a triple $(\underline{v}, I(\underline{v}), T(\underline{v}, \underline{v}'))$, where $\underline{v}$ is a set of variables, and $I(\underline{v})$, $T(\underline{v}, \underline{v}')$ are first order formulae over some signature. An assignment to the variables in $\underline{v}$ is a state. A state s is initial iff it is a model of $I(\underline{v})$, i.e. $s \models I(\underline{v})$. The states s, s' denote a transition iff $s \cup s' \models T(\underline{v}, \underline{v}')$, also written $T(s, s')$. A path is a sequence of states $s_0, s_1, \ldots$ such that s_0 is initial and $T(s_i, s'_{i+1})$ for all i. We denote paths with π, and with $\pi[j]$ the j-th element of π. A state s is reachable iff there exists a path π such that $\pi[i] = s$ for some i. A variable v is frozen iff for all π, i it holds that $\pi[i](v) = \pi[0](v)$. In the following, when we define a frozen variable v, we assume that this is done by having a constraint $v' = v$ as a top-level conjunct of the transition formula. A formula $\phi(\underline{v})$ is an invariant of the transition system $C = (\underline{v}, I(\underline{v}), T(\underline{v}, \underline{v}'))$ iff it holds in all the reachable states. Following the standard model checking notation, we denote this with $C \models \phi(\underline{v})$.[1] A formula $\phi(\underline{v})$ is an inductive invariant for C iff $I(\underline{v}) \models \phi(\underline{v})$ and $\phi(\underline{v}) \wedge T(\underline{v}, \underline{v}') \models \phi(\underline{v}')$.

3 Modeling Parametric Systems as Array-based Transition Systems

3.1 Array-based Transition Systems

In order to describe parametric systems, we adapt from [9] the notion of array-based systems. In the following, we fix a theory of indexes $\mathcal{T}_I = (\Sigma_I, \mathcal{C}_I)$ and a theory of elements $\mathcal{T}_E = (\Sigma_E, \mathcal{C}_E)$. In order to model the parameters, we require that the class $\mathcal{C}_I$ is closed under substructure. Then with A_I^E we denote the theory whose signature is $\Sigma = \Sigma_I \cup \Sigma_E \cup \{[\_]\}$, and a model for it is given by a set of total functions from a model of $\mathcal{T}_I$ to a model of $\mathcal{T}_E$. In general, we

[1] Note that we use the symbol $\models$ with three different denotations: if ϕ, ψ are formulae, $\phi \models \psi$ denotes that ψ is a logical consequence of ϕ; if μ is an interpretation, and ψ is a formula, $\mu \models \psi$ denotes that μ is a model of ψ; if C is a transition system, $C \models \psi$ denotes that ψ is an invariant of C. The different meanings will be clear from the context.

can have several array theories with multiple sorts for indexes and elements. For simplicity, we fix only an *index* sort and an *elem* sort. In the following, an array-based transition system

$$C = (a, \iota(a), \tau(a, a'))$$

is a symbolic transition system, with the additional constraints that:

- a is a variable of sort *index* $\mapsto$ *elem*. We use a single variable for the sake of simplicity: additional variables of arbitrary type (also of index or element type) can be added without loss of generality.
- $\iota(a)$ is a first-order formula of the form $\forall \underline{i}.\phi(\underline{i}, a[\underline{i}])$, where $\underline{i}$ is of index sort and ϕ is a quantifier-free formula.
- $\tau(a, a')$ is a finite disjunction of formulae, $\vee_{k=1}^{n} \tau_k$, such that every τ_k is a formula of the following type (with $\underline{i}, \underline{j}$ of index sort):

  $$\exists \underline{i} \forall \underline{j}.\psi(\underline{i}, \underline{j}, a[\underline{i}], a[\underline{j}], a'[\underline{i}], a'[\underline{j}])$$

  with ψ a quantifier-free formula.

This syntactic requirement subsumes the common guard and update formalism used for the description of parametric systems, used e.g in [9, 11, 14].

In the following, we shall refer to the disjuncts τ_k of τ as *transition rules* (or simply *rules* when clear from the context).

An array-based transition system can be seen as a family of transition systems, one for each cardinality of the finite models $\mathcal{M}_I$ of $\mathcal{T}_I$. In the following, given d an integer, we denote with C^d *the finite instance of* C *of size* d obtained by instantiating the quantifiers of C over a set of fresh index variables of cardinality d (considered implicitly different from each other). Note that this C^d is a *symmetric presentation* [14]: if $\underline{c} = \{c_1, \dots, c_d\}$ are the fresh index variables, and σ is a permutation of $\underline{c}$, we have that, for every formula $\phi(\underline{c}, a[\underline{c}])$, $C^d \models \phi(\underline{c}, a[\underline{c}]) \Leftrightarrow C^d \models \phi(\sigma(\underline{c}), a[\sigma(\underline{c})])$.

Example 1 (Mutex Protocol for Ring Topology). Here we describe a simple protocol for accessing a shared resource, with processes in a ring-shaped topology. As an index theory, we use the finite sets of integers. As an element theory, we use both the Booleans and an enumerated data type of two elements, namely $\{idle, critical\}$. The array variable t, with sort *index* $\mapsto$ *boolean*, is true in an index variable x if x holds the token. The variable s, with sort *index* $\mapsto$ $\{idle, critical\}$ holds the current state of the process. In addition, we have an integer frozen variable *length*, which represents the length of the ring. The transition system is described by the following formulae:

Initial states. Initially, only one process holds the token, and every process is idle. We model this initial process with an additional constant *init_token* of sort *index*. Moreover, each index is bounded by the value of *length*. The initial formula is:

$$\forall j.p[j] = \mathit{idle} \wedge j \geq 1 \wedge j \leq \mathit{length} \wedge \mathit{length} > 0$$
$$\wedge \begin{cases} \textbf{if } j = \mathit{init\_token} \textbf{ then } t[j] = \mathit{true} \\ \textbf{else } t[j] = \mathit{false} \end{cases}$$

Transition rule 1. A process which holds the token can enter the critical section:

$$\exists i.s[i] = \mathit{idle} \wedge t[i] = \mathit{true} \wedge s'[i] = \mathit{critical} \wedge t'[i] = t[i] \wedge$$
$$\forall j, j \neq i.(s'[j] = s[j] \wedge t'[j] = t[j])$$

Transition rule 2. A process exits from the critical section and passes the token to the process at its right:

$$\exists i. \wedge s[i] = \mathit{critical} \wedge s'[i] = \mathit{idle} \wedge t'[i] = \mathit{false} \wedge$$
$$\forall j, j \neq i. \begin{cases} \textbf{if } j = 1 \wedge i = \mathit{length} \textbf{ then } s'[j] = s[j] \wedge t'[j] = \mathit{true} \\ \textbf{elif } j = i + 1 \wedge i < \mathit{length} \textbf{ then } s'[j] = s[j] \wedge t'[j] = \mathit{true} \\ \textbf{else } s'[j] = s[j] \wedge t'[j] = t[j] \end{cases}$$

3.2 Universal invariant problem for array-based systems

In the following, given an array-based transition system

$$C = (a, \iota(a), \tau(a, a')),$$

the *universal invariant problem* is the problem of proving (or disproving) that a formula of the form $\Phi \overset{\text{def}}{=} \forall \underline{i}.\phi(\underline{i}, a[\underline{i}])$ is an invariant for C.

Guard Strengthening In order to prove that $\forall \underline{i}.\phi(\underline{i}, a[\underline{i}])$ is an invariant of a system $C = (a, \iota(a), \tau(a, a'))$, we can first strengthen the rules of C by adding the candidate invariant in conjunction with the transition relation, and then prove that the formula is an invariant of the newly-restricted system. This induction principle is justified by the following proposition:

Proposition 1 (Guard strenghtening [14]) *Let $C = (a, \iota(a), \tau(a, a'))$ be a transition system and let Φ be $\forall \underline{i}.\phi(\underline{i}, a[\underline{i}])$. Let $C_\Phi = (a, \iota(a), \tau(a, a') \wedge \Phi)$ be the guard-strengthening of C with respect to Φ. Then, if Φ is an invariant of C_Φ, it is also an invariant of C.*

Prophecy variables The universal quantifiers in the candidate invariant can be replaced with fresh frozen variables, called *prophecy variables*, that intuitively contain the indexes of the processes witnessing the violation of the property.

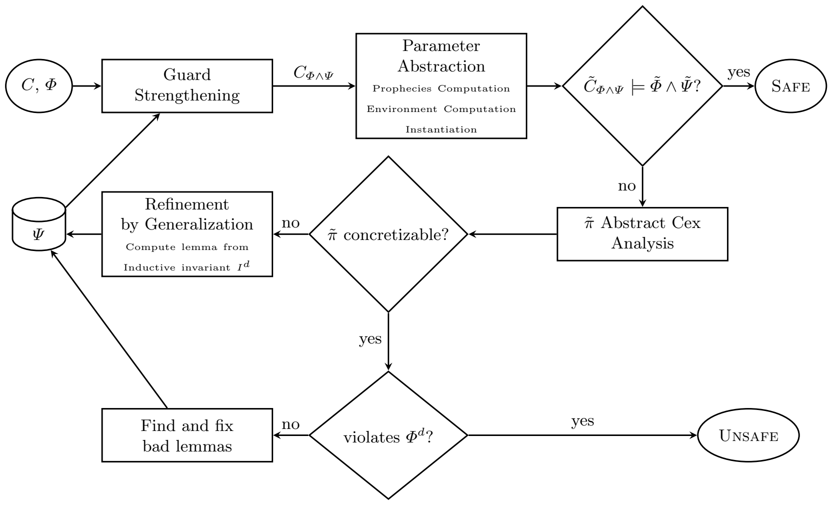
C, Φ
Guard Strengthening
$C_{\Phi\wedge\Psi}$
Parameter Abstraction
Prophecies Computation
Environment Computation
Instantiation
$\tilde{C}_{\Phi\wedge\Psi} \models \tilde{\Phi} \wedge \tilde{\Psi}$?
yes
Safe
no
$\tilde{\pi}$ Abstract Cex Analysis
$\tilde{\pi}$ concretizable?
no
Refinement by Generalization
Compute lemma from Inductive invariant I^d
Ψ
yes
violates Φ^d?
no
Find and fix bad lemmas
yes
Unsafe

Fig. 1. An overview of the algorithm. C is an array-based transition system; Φ is a quantified candidate invariant; $\Psi \stackrel{\text{def}}{=} \{\psi_1, \ldots \psi_n\}$ is the set of candidate lemmas; $C_{\Phi\wedge\Psi}$ is a quantified transition system resulting from the strengthening of C; $\tilde{C}_{\Phi\wedge\Psi}$ is a quantifier-free transition system.

Proposition 2 (Removing quantifiers [18]) *Let $C = (a, \iota(a), \tau(a, a'))$ be an array-based system. The formula $\forall \underline{i}.\phi(\underline{i}, a[\underline{i}])$ is an invariant for C iff the formula $\phi(\underline{p}, a[\underline{p}])$ is an invariant for $C_{+\underline{p}} = (a \cup \underline{p}, \iota(a), \tau(a, a'))$, where $\underline{p}$ is a set of fresh frozen variables of index sort.*

For better readability, in the following we will omit the subscript $+\underline{p}$. Moreover, we assume that the index variables universally quantified in the candidate invariant are considered to be different. This does not limit expressiveness, and simplifies our discourse. Therefore, the prophecy variables induced by a candidate invariant are considered to be *implicitly different*.

Example 2. In the ring protocol example, the universal invariant property that we want to prove is mutual exclusion, i.e. $\forall i, j.\neg(s[i] = \mathit{critical} \wedge s[j] = \mathit{critical})$. By introducing the prophecy variables we obtain $\neg(s[p_1] = \mathit{critical} \wedge s[p_2] = \mathit{critical})$.

4 Overview of the Method

In the following, let an array-based transition system $C \stackrel{\text{def}}{=} (a, \iota(a), \tau(a, a'))$, and a candidate universal invariant $\Phi \stackrel{\text{def}}{=} \forall \underline{i}.\phi(\underline{i}, a[\underline{i}])$ for C be given.

We now summarize the algorithm that attempts to solve the universal invariant problem for C and Φ. The algorithm, depicted in Figure 4, iterates trying either to construct an abstraction sufficiently precise to prove the property (exit with SAFE), or to find a finite instantiation of the problem exhibiting a concrete counterexample (exit with UNSAFE). The abstract space is quantifier-free, and obtained by instantiating the universally quantified formulae over two sets of index variables: the prophecy variables, which arise from the candidate invariant (as explained in Proposition 2), and are denoted with $\underline{p}$; and the *environmental* variables, denoted with $\underline{x}$, which arise from the transition formula and are intended to represent the environment surrounding the $\underline{p}$ indexes, interacting with them in the behaviour leading to the violation. While prophecy variables are frozen, thus representing the same indexes for the whole run, environmental variables are free to change at each time step, hence producing possibly spurious behaviours. The algorithm maintains a set of *candidate lemmas* $\Psi \stackrel{\text{def}}{=} \{\Psi_i\}_i$, composed of universally quantified formulae, that are used to strengthen the property and to tighten the abstraction. Initially, Ψ is empty. In the following, if C^d is a finite instance of C and Φ is a candidate universal invariant, with Φ^d we denote the formula obtained from Φ by instantiating the quantifiers in variables used for the domain of cardinality d.

At each iteration, we carry out the following high-level steps (described in detail in the next sections):

- the property Φ to be proved is conjoined with the candidate lemmas in Ψ, and its quantifiers are moved in prenex form;[2]
- we construct the guard-strengthening $C_{\Phi\wedge\Psi}$ (cfr. Proposition 1), conjoining $\Phi \wedge \Psi$ to the transition rules of C;
- we compute our modified Parameter Abstraction of $C_{\Phi\wedge\Psi}$ (defined in §5.1). First, we define the necessary prophecy variables $\underline{p}$ and environmental variables $\underline{x}$. Then, we instantiate the quantifiers obtaining the quantifier-free array transition system $\widetilde{C}_{\Phi\wedge\Psi}$.
- we (try to) solve the invariant checking problem $\widetilde{C}_{\Phi\wedge\Psi} \models \tilde{\Phi} \wedge \tilde{\Psi}$ by calling a model checker for quantifier-free transition systems. $\tilde{\Phi} \wedge \tilde{\Psi}$ is obtained from $\Phi \wedge \Psi$ by removing quantifiers with prophecy variables, as in Proposition 2
- if the model checker concludes that there is no violation, then Φ holds in C (for the properties of the Parameter Abstraction), and we exit with SAFE.
- otherwise, we try to check whether the property violation in the abstract space corresponds to a real counterexample. We do so by checking whether the current property $\Phi \wedge \Psi$ is falsified in C^d, a suitable finite instance of C. That is, we check whether $C^d \models (\Phi \wedge \Psi)^d$.
- if $C^d \models (\Psi \wedge \Phi)^d$, then the abstraction must be tightened. When the verification of the finite instance succeeds, an inductive invariant I^d is produced, which is used to compute (candidate) lemmas by generalization from d to the universal case.
- if $C^d \not\models (\Psi \wedge \Phi)^d$, two cases are possible. First, we check if the (instantiation of the) property Φ is indeed violated. If so, we exit with UNSAFE, and we produce a concrete counterexample to the original problem, finitely witnessed in C^d.

[2] In the following, with $\Phi \wedge \Psi$ we denote the prenex form $\Phi \wedge \bigwedge_i \Psi_i$

– However, it is also possible that C^d does not violate Φ^d, but it falsifies some lemmas. In fact, the candidate lemmas obtained at previous iterations, by generalization on C^{d^-} with $d^- \neq d$, may not hold universally in C. In that case, the bad lemmas must be fixed, and the iteration is restarted.

When the algorithm terminates with Unsafe, we are able to exhibit a finite counterexample trace in a finite instance of C violating the property. When the algorithm terminates with safe, then the property holds in C. The result is obtained by the following chain of implications: from Theorem 3, stated in the next section, we have that $\tilde{C}_{\Phi \wedge \Psi} \models \tilde{\Phi} \wedge \tilde{\Psi}$ implies $C_{\Phi \wedge \Psi} \models \tilde{\Phi} \wedge \tilde{\Psi}$. From Proposition 2, we have that $C_{\Phi \wedge \Psi} \models \Phi \wedge \Psi$. Therefore, from Proposition 1, we have $C \models \Phi \wedge \Psi$. In particular, we have $C \models \Phi$.

5 Modified Parameter Abstraction

We describe here our Parameter Abstraction. The first version of this approach was introduced in [3], and later formalized in [14]. In the following, we describe a novel version of the abstraction, and how it can be applied to array-based transition systems. The main novelty is that, instead of using a special abstract index "$*$" that overapproximates the behaviour of the system in the array locations that are not explicitly tracked, we use n *environmental (index) variables* which are not abstracted, but are allowed to change nondeterministically in some transitions. This can be achieved by the usage of an additional **stuttering transition**: this rule allows the environmental variables to change value arbitrarily, while not changing the values of the array in the prophecies.

5.1 Abstraction Computation

Let an array-based transition system C and a universal invariant Φ be given[3]. By conjoining Φ to the transition rules in C, we obtain C_Φ, the guard strengthening of C with respect to Φ. Then, we define two sets of variables: the prophecy variables $\underline{p}$, in number determined by Proposition 2, and the environmental variables $\underline{x}$, in number determined by the greatest existential quantification depth in the transition rules of C_Φ. While the prophecies are frozen variables, the interpretation of the environmental variables is not fixed. Moreover, we assume that the values taken by $\underline{p}$ and $\underline{x}$ are different. We now define $\tilde{C}$, the parameter abstraction of C.

Initial formula Let $\iota(a)$ be $\forall \underline{i}.\phi(\underline{i}, a[\underline{i}])$, the initial formula of C in prenex form, with $\phi(\underline{i}, a[\underline{i}])$ quantifier-free. The initial formula of the abstract system is a quantifier-free first order formula, denoted $\tilde{\iota}(\underline{p}, a[\underline{p}])$ obtained by instantiating all the universal quantifiers in ι over the set of prophecy variables $\underline{p}$.

[3] These represent the system and the property in input to each iteration of the loop.

Transition formula The transition formula of C_Φ is still represented by a disjuction of formulae of the form[4]

$$\tau(a, a') \quad \overset{\text{def}}{=} \quad \exists \underline{i} \forall \underline{j}. \psi(\underline{i}, \underline{j}, a[\underline{i}], a[\underline{j}], a'[\underline{i}], a'[\underline{j}]).$$

For simplicity, we can assume that we have only one rule $\tau(a, a')$. First, we compute the set of all substitutions of the $\underline{i}$ over $\underline{p} \cup \underline{x}$, and we consider the set of formulae $\{\tilde{\tau}_j(\underline{p}, \underline{x}, a, a')\}$, where j ranges over the substitutions, and $\tilde{\tau}_j$ is the result of applying the substitution to τ.

Then, for each formula in the set $\{\tilde{\tau}_j\}$, we instantiate the universal quantifiers over the set $\underline{p} \cup \underline{x}$, obtaining a quantifier-free formula over prophecy and environmental variables.

Moreover, we consider an additional transition formula, called the **stuttering transition**, defined by:

$$\tilde{\tau}_S \overset{\text{def}}{=} \bigwedge_{p \in \underline{p}} a'[p] = a[p] \wedge p' = p$$

The disjunction of all the abstracted transition formulae is the transition formula $\tilde{\tau}$. So, we can now define the transition system

$$\tilde{C} \overset{\text{def}}{=} (\{a, \underline{p}, \underline{x}\}, \tilde{\iota}(\underline{p}, a[\underline{p}]), \tilde{\tau}(\underline{p}, \underline{x}, a[\underline{p} \cup \underline{x}], a'[\underline{p} \cup \underline{x}])).$$

Example 3. We apply the abstraction procedure to the transition rule 2 of the token in the ring protocol of Example 1.
Since the invariant is the formula $\forall i, j. \neg(s[i] = \mathit{critical} \wedge s[j] = \mathit{critical})$ it follows that we have two prophecy variables p_1, p_2. Recall that the invariant itself is added to the transition as an additional conjunct. Since the existential quantification depth is one, we have only one environment variable x_1. In the abstraction system we obtain three transition formulae from the original transition; we report the one indexed by the substitution mapping i into x_1; such a formula is equivalent to the following:

$$s[x_1] = \mathit{crit} \wedge t[x_1] = \mathit{true} \wedge s'[x_1] = \mathit{idle} \wedge t'[x_1] = \mathit{false} \wedge$$

$$\bigwedge_{j \in \{p_1, p_2\}} \begin{cases} \textbf{if } j = 1 \wedge x_1 = \mathit{length} \textbf{ then } s'[j] = s[j] \wedge t'[j] = \mathit{false} \\ \textbf{elif } j = x_1 + 1 \wedge x_1 < \mathit{length} \textbf{ then } s'[j] = s[j] \wedge t'[j] = \mathit{false} \\ \textbf{else } s'[j] = s[j] \wedge t'[j] = t[j] \end{cases}$$

$$\bigwedge_{\substack{i,j \in \{p_1, p_2, x_1\} \\ i \neq j}} \neg(s[i] = \mathit{critical} \wedge s[j] = \mathit{critical})$$

[4] Possibly by performing trivial logical manipulations to distribute the guard strengthening inside the rules.

5.2 Stuttering Simulation

We define here the stuttering simulation induced by our version of the Parameter Abstraction. The proof of the main theorem can be found in the appendix. The stuttering is induced by $\tilde{\tau}_S$: this is a weaker version than the simulation induced by [14], yet it is sufficient for preserving invariants.

Definition 1 (Stuttering Simulation) *Given two symbolic transition systems $C_1 = (\underline{x_1}, \iota_1, \tau_1)$ and $C_2 = (\underline{x_2}, \iota_2, \tau_2)$, with sets of states S_1 and S_2, a stuttering simulation $\mathcal{S}$ is a relation $\mathcal{S} \subset S_1 \times S_2$, such that:*

- *for every $s_1 \in S_1$ such that $s_1 \models \iota_1$, there exists some $s_2 \in S_2$ such that $(s_1, s_2) \in \mathcal{S}$ and $s_2 \models \iota_2$;*
- *for every $(s_1, s_2) \in \mathcal{S}$, and for every $s'_1 \in S_1$ such that $s_1 \cup s'_1 \models \tau_1$, there exists either some $s'_2 \in S_2$ such that $(s'_1, s'_2) \in \mathcal{S}$ and $s_2 \cup s'_2 \models \tau_2$, or some $(s'_2, s''_2) \in S_2 \times S_2$ such that $(s'_1, s''_2) \in \mathcal{S}$, and $s_2 \cup s'_2 \models \tau_2$, $s'_2 \cup s''_2 \models \tau_2$.*

If such a relation exists, we say that C_2 stutter simulates C_1.

We write $\mathcal{S}(s_1)$ for $\{s_2 | (s_1, s_2)\} \in \mathcal{S}$. We recall that stutter simulation preserves reachability, i.e. if C_2 stutter simulates C_1, then if s_1 is reachable in C_1 then the set $\mathcal{S}(s_1)$ is reachable in C_2. Formally, the stuttering simulation induced by the Parameter Abstraction is defined as follows.

Definition 2 (Simulation) *Let C be the original transition system and let $\tilde{C}$ be its Parameter Abstraction. Let s and $\tilde{s}$ denote states of C and $\tilde{C}$, respectively. We define $\mathcal{S}$ as follows:*

$$\mathcal{S}(s, \tilde{s}) \text{ iff } s(a)[i] = \tilde{s}(a)[i] \text{ for all } i \in \bigcup_{p \in \underline{p}} \tilde{s}(p).$$

Intuitively, we require that in the concrete state s and the abstract state $\tilde{s}$, the array is interpreted in the same way for all the locations referred by the prophecy variables. We then have the following:

Theorem 3. *The relation $\mathcal{S}$ is a stuttering simulation between C and $\tilde{C}$. Moreover, if $\tilde{C} \models \Phi(\underline{p}, a[\underline{p}])$, then $C \models \Phi(\underline{p}, a[\underline{p}])$.*

6 Refinement

6.1 Concretization

If $\Phi(\underline{p}, a[\underline{p}])$ does not hold in $\tilde{C}$, in general we cannot conclude anything, since the abstraction could be too coarse. So, if an abstract counterexample is encountered, we try to explore a small instance of the system to see if this counterexample occurs in it. To choose the appropriate size, our algorithm keeps a counter d, whose value is equal to the size to explore. Initially, d is equal to the number of (universally-quantified) index variables in the property Φ.[5] When an abstract counterexample is encountered, we check whether $C^d \models (\Phi \wedge \Psi)^d$. For this check, we use a model checker able to return, in case of success, an inductive invariant I^d. From the inductive invariant we compute some first order formulae J which will be a new set of candidate lemmas. We will see later how to obtain this generalization. After computing the new lemmas, we set $d = d + 1$. If a concrete counterexample is found, then there are two cases: (i) the counterexample falsifies the original property, and we exit from the algorithm with a concrete counterexample; (ii) the counterexample falsifies some lemmas; in this case we remove the lemma and restart the loop (without changing d).

6.2 From Invariants to Universal Lemmas

Definition 3 *Let d be an integer, and let I^d be a set of clauses containing d variables. A generalization of I^d is a first-order formula J such that, when evaluating the quantifiers in J in a domain with precisely d elements, we obtain a formula equivalent to I^d.*

We use the following technique for generalization. Suppose that I^d is in CNF, and that we used $c_1, \ldots, c_d$ as variables for an instance with d elements. Then, $I^d = \mathcal{C}_1 \wedge \cdots \wedge \mathcal{C}_n$ is a conjunction of clauses. From each of those clauses we will obtain a new candidate lemma. Let $\mathit{AllDiff}(\underline{i})$ be the formula which states that all variables in $\underline{i}$ are different from each other. Since every C^d is given by a symmetric presentation [14], we have that, for every $i \in \{1, \ldots, n\}$, $C^d \models \forall i_1, \ldots, i_h . \mathit{AllDiff}(i_1, \ldots, i_h) \rightarrow \mathcal{C}_i(i_1, \ldots, i_h)$, where the quantifiers range over $c_1, \ldots, c_d$ and $h \leq d$ is the number of variables which occur in $\mathcal{C}_i$. This means that $J \stackrel{\text{def}}{=} \bigwedge_i \forall \underline{i} . \mathit{AllDiff}(\underline{i}) \rightarrow \mathcal{C}_i(\underline{i})$ is a generalization of I^d. In our algorithm, we add the set $\{\forall \underline{i} . \mathcal{C}_i(\underline{i})\}_{i=1}^n$ of new candidate lemmas to Ψ. Note that we omitted the formula $\mathit{AllDiff}$ for our assumption on the different values of index variables.

Example 4. Let $\tilde{C}$ be the result of the abstraction procedure of the ring protocol of Example 1. An abstract counterexample for the property is given by a trace where the environmental variable takes the token twice (with stuttering transitions), and then it passes it to both prophecy variables, which can then enter the critical section. Therefore, we compute C^2, the finite instance with exactly two indexes, and we check if the property holds in it. The property does hold, and we obtain an inductive invariant:

$$I^2 = (t[c_1] = false \vee t[c_2] = false) \wedge (p[c_1] = idle \vee t[c_1] = true) \wedge \ldots$$

where the dots denote some other clauses which are equal up to a symmetry to c_1 or c_2. If we generalize the first two clauses, we have the two formulae: $\psi_1 \stackrel{\text{def}}{=} \forall i, j . i \neq j \rightarrow t[i] = false \vee t[j] = false$, and $\psi_2 = \forall i . p[i] = idle \vee t[i] = true$,

[5] Recall that we assume that quantified index variables are required to be different. Therefore, the property holds vacuously on instances of size smaller than the number of index variables in Φ.

such that $\psi_1^2 \wedge \psi_2^2$ is equivalent to I^2. After the guard strengthening process, the new lemmas are enough to block all the spurious counterexamples, and the property is proved.

Fixing Unsound Lemmas Unfortunately, we know a priori that a lemma holds only for the instance from which it was generalized. In general, its universal generalization obtained as outlined above might not hold in the system.

Suppose that the formula ψ_1 is a candidate lemma, obtained by generalization after the successful verification of an instance of size d. Suppose that later, a counterexample for ψ_1 is found by exploring a different instance $C^{d'}$ (with $d' > d$). This means that the lemma ψ_1 does not hold universally, but only for some finite instances of the system (including C^d), and not in general. In this case, we simply remove ψ_1 from the set of candidate lemmas Ψ, thus effectively weakening our working property (from $\Phi \wedge \Psi$ to $\Phi \wedge (\Psi \setminus \{\psi_1\})$). While this may cause a particular (abstract) counterexample to be encountered more than once during the main loop of the algorithm, since the finite instances are explored monotonically and their size d is increased after every successful verification of a bounded instance, the overall procedure still makes progress by exploring increasingly-large instances of the system. The hope is that eventually the algorithm will discover enough good lemmas that block the abstract counterexample. This notion of (weak) progress is justified by the following:

Proposition 4 *Let $\tilde{\pi}$ be an abstract counterexample, Ψ be the current set of universally quantified lemmas, and d be the size of the bounded instance to explore. During every execution of the algorithm, the same triple $(\tilde{\pi}, \Psi, d)$ never occurs twice.*

7 Related Work

Parametric verification is a challenging problem, and there is a large body of work in the literature devoted to this problem. Here, we (necessarily) focus on the approaches that are most related to ours.

Several methods are based on quantifier elimination using decidable fragments of first order logic, with notable examples in [6, 9, 21]. These methods guarantee a high degree of automation, but typically impose strong syntactic requirements in the input problem, and may suffer from scalability issues. A second popular approach is based on abstraction and abstraction refinement. Within this family of abstractions, earlier versions of the Paramater Abstraction [3, 14] have been used successfully also for industrial protocols [23]. The main drawback is that the degree of automation is limited, and substantial expertise is required to obtain the desired results. The first steps of our abstraction algorithm are inspired by the ones in [18] and [14]. The key difference from [18] is that in that work the abstract transition system $\tilde{C}$ is given by an eager propositional abstraction, with the axioms of the background theories recovered by the usage of some schemata. Here we retain the theory of arrays in the abstract space $\tilde{C}$. Moreover, differently from both [14] and [18], our procedure includes an automatic refinement of the abstraction in a counterexample-driven manner.

Ivy [19, 21] implements both semi-automatic invariant checking with decidable logics (namely, Effectively Propositional Logic – EPR) and compositional abstraction with eager axioms [18]. MyPyvy [12,13] is a model checker inspired by the language of Ivy. It implements a version of IC3 capable of dealing with universal formulas [12]; the algorithm is completely automatic, but it is still based on quantifier elimination via reduction to decidable logics. In a more recent work, MyPyvy has gained the capability of inferring invariants with quantifier alternations, using a procedure that combines separators and first-order logic [13]. At the moment, our framework is capable of handling only universally quantified invariants. On the other hand, our approach is not limited to EPR, but it can in principle handle formulae with arbitrary SMT theories.

Exploring small instances of a parameterized system for candidate lemmas is a popular approach for parametric verification. In [7], this idea is used to over-approximate backward reachable states inside an algorithm which combines backward search and quantifier elimination. In [15], a finite-instance exploration is used together with a theorem prover to check the validity of candidate lemmas. In [16], candidate invariants are obtained from the set of reachable states of small instances. Similarly to our approach, these lemmas are used to strengthen an earlier version of the parameter abstraction. However, human intervention is still needed for the refinement.

A similar approach is presented in [22], where lemmas are obtained from a generalization of the proof of the property in a small instance of the protocol. The main difference with our technique, besides the methods used to extract such invariants, is the following: in [22], the authors show that to prove that a property (conjoined with lemmas) is inductive for all N, it is enough to prove that it is inductive for a particular N_0, which is computable from the number of variables in the description of the system. This result is obtained from the imposed syntactic structure of the system. On the other hand, we impose less structure, and we rely on proving the property in an abstract version (and not a concrete instance) of the system. Moreover, our approach is integrated in an abstraction/refinement loop, which is missing from [22].

Another SMT-based approach for parametric verification is in [11]. The method is based on a reduction of invariant checking to the satisfiability of non-linear Constrained Horn Clauses (CHCs). Besides differing substantially in the overall approach, the method is more restrictive in the input language, and handles invariants only with a specific syntactic structure.

The use of prophecy variables for inferring universally quantified invariants has been explored also in non-parametric contexts, such as [17]. The main difference with our work is that [17] focuses on finding quantified invariants for quantifier-free transition systems with arrays, rather than array-based systems with quantifiers. The overall abstraction-refinement approach is also substantially different.

8 Experimental Evaluation

We have implemented our algorithm in a tool called LAMBDA (for **L**earning **A**bstractions fro**M** **B**ounde**D** **A**nalysis). LAMBDA is written in Python, and uses the SMT-based IC3 with implicit predicate abstraction of [4] as underlying quantifier-free verification engine.[6] LAMBDA accepts as input array-based systems specified either in the language of MCMT [10] or in VMT format (a light-weight extension of SMT-LIB to model transition systems [24]). In case of successful termination, LAMBDA generates either a counterexample trace (for violated properties) in a concrete instance of the parametric system, or a quantified inductive invariant that proves the property for any instance of the system. In the latter case, LAMBDA can also generate proof obligations that can be independently checked with an SMT solver supporting quantifiers, such as Z3 [20] or CVC4 [2]. More specifically, the quantified inductive invariant can be generated by LAMBDA by simply universally quantifying all the (index) variables in the inductive invariant generated for $\widetilde{C}$, and conjoining it with the lemmas Ψ discovered during the main loop iterations. Computing such an invariant is immediate after the termination of the algorithm, and does not require additional reasoning.

In order to evaluate the effectiveness of our method, we have compared LAMBDA with two state-of-the-art tools for the verification of array-based systems, namely CUBICLE [6] and MCMT. We could not include MYPYVY in the comparison, due to the many differences in input languages and modeling formalisms, which make an automatic translation of the benchmarks very difficult. We would also have liked to compare with the technique of [11], however the prototype tool mentioned in the paper doesn't seem to be available.

For our evaluation, we have collected a total of 116 benchmarks, divided in three different groups:

Protocols consists of 42 instances taken from the MCMT or the CUBICLE distributions, and used in previous works on verifcation of array-based systems. We have used all the instances which were available in both input formats, and we have split benchmarks containing multiple properties into different files.

DynArch consists of 57 instances of verification problems of dynamic architectures, taken from [5]. These benchmarks make use of arithmetic constraints on index terms, which are not supported by CUBICLE. Therefore, we could only compare LAMBDA with MCMT on them.

Trains consists of 17 instances derived by (a simplified version of) verification problems on railway interlocking logics [1]. These benchmarks make use of several features that are not fully supported by CUBICLE and MCMT (such as non-functional updates in the transition relation, transition rules with more than one universally-quantified variable, real-valued variables). None of such restrictions applies to LAMBDA, which in general accepts models with significatly

[6] In our implementation, we use the theory of integers as an index theory. At first, this may seem odd, since we should consider all finite subsets of the integers. However, this is not a problem, since the satisfiability of a quantifier-free UFLIA formula is equivalent to its satisfiability in a finite index model.

Table 1. Summary of experimental results.

		Lambda		MCMT		Cubicle	
Benchmark family	**# of instances**	**Solved**	**Unique**	**Solved**	**Unique**	**Solved**	**Unique**
Protocols	42	34	3	24	0	30	1
DynArch	57	48	5	48	5	–	–
Trains	17	17	–	–	–	–	–

fewer syntactic constraints than Cubicle and MCMT. Since these instances are inspired by relevant real-world verification problems, we believe that it is interesting to include them in the evaluation even though we could only run Lambda on them.

Our implementation, all the benchmarks, and the scripts for reproducing the results are available at http://es-static.fbk.eu/people/griggio/papers/cade21-lambda.tar.gz. We have run our experiments on a cluster of machines with a 2.90GHz Intel Xeon Gold 6226R CPU running Ubuntu Linux 20.04.1, using a time limit of 1 hour and a memory limit of 4GB for each instance. We have used the default settings for MCMT, whereas for Cubicle we have also enabled the BRAB algorithm.[7] A summary of the results of our evaluation are presented in Table 2. More details are provided in Appendix A.2.

Overall, Lambda is very competitive with the state of the art, and in fact it solves the largest number of instances (even when disregarding the Trains group, which cannot be handled by the other tools).When considering the Protocols group, Cubicle is often significantly faster than Lambda (see also Table 2 in Appendix A.2), especially on easier problems, thanks to its explicit-state exploration component (part of the BRAB algorithm). However, the symbolic techniques used by Lambda allow it to generally scale better to larger, more challenging problems: in the end, Lambda solves 4 more instances than Cubicle, and 10 more than MCMT. The situation is different for the DynArch group, in which Lambda and MCMT solve the same number of instances. However, it is interesting to observe that both tools can solve 5 instances that the other tool cannot solve; more in general, it seems that the two approaches have somewhat complementary strengths (this can be seen clearly from Table 3 in Appendix A.2). Moreover, as already stated above, the fact that Lambda imposes significantly less syntactic restrictions than the other two tools considered allowed it to handle all the instances of the Trains group, which cannot be easily modeled in the languages of MCMT or Cubicle.

Finally, we wish to remark that we have generated SMT proof obligations for checking the correctness of all the (universally quantified) inductive invariants produced by Lambda, and checked them with both CVC4 and Z3. None of the solvers reported any error, and overall the combination of the two solvers was able to successfully verify all the proof obligations for 65 of the 67 instances reported

[7] The results reported were obtained using `-brab 2`; we have however experimented also with other (small) values for `-brab`, without noticing any significant difference.

as safe.[8] We believe that the fact that we can easily produce proof obligations that can be independently checked is another strength of our approach. This is in contrast to the approach of Cubicle, where generating proof obligations is nontrivial [8].

9 Conclusions

In this paper we tackled the problem of universal invariant checking for parametric systems. We proposed a fully-automated abstraction-refinement approach, based on quantifier-free reasoning. The abstract model, that stutter simulates the concrete model, is a quantifier-free symbolic transition system refined by (the instantiation of) candidate universal lemmas. These are obtained by analyzing the proofs of validity of the property in a finite instance of the parametric system. We experimentally evaluated an implementation on standard benchmarks from the literature. The results show the effectiveness of the method, also in comparison with state-of-the-art tools (Cubicle, MCMT). We are able to prove, in a fully automated manner and without manual intervention, several benchmarks that are considered challenging. In the future, we plan to work on generalization, to improve the ability of inferring the right lemmas from a small instance, and to find more effective ways to filter out bad candidates. On the theoretical side, we will investigate the relation between the termination of the algorithm and decidable classes of parametric systems (e.g. those that enjoy a cut-off property). Finally, we will work on the verification of temporally extended properties which are also preserved by stuttering simulations (such as fragments of Linear Temporal Logic).

References

1. Amendola, A., Becchi, A., Cavada, R., Cimatti, A., Griggio, A., Scaglione, G., Susi, A., Tacchella, A., Tessi, M.: A model-based approach to the design, verification and deployment of railway interlocking system. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III. Lecture Notes in Computer Science, vol. 12478, pp. 240–254. Springer (2020). https://doi.org/10.1007/978-3-030-61467-6_16, https://doi.org/10.1007/978-3-030-61467-6_16
2. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011)
3. Chou, C.T., Mannava, P.K., Park, S.: A simple method for parameterized verification of cache coherence protocols. In: Hu, A.J., Martin, A.K. (eds.) Formal Methods in Computer-Aided Design. pp. 382–398. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

[8] In the remaining two cases, both solvers returned `unknown` when trying to prove the validity of some of the proof obligations.

4. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state invariant checking with IC3 and predicate abstraction. Formal Methods Syst. Des. **49**(3), 190–218 (2016). https://doi.org/10.1007/s10703-016-0257-4, https://doi.org/10.1007/s10703-016-0257-4
5. Cimatti, A., Stojic, I., Tonetta, S.: Formal specification and verification of dynamic parametrized architectures. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E.P. (eds.) Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10951, pp. 625–644. Springer (2018). https://doi.org/10.1007/978-3-319-95582-7_37, https://doi.org/10.1007/978-3-319-95582-7_37
6. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaïdi, F.: Cubicle: A Parallel SMT-based Model Checker for Parameterized Systems. In: Parthasarathy, M., Seshia, S.A. (eds.) CAV 2012: Proceedings of the 24th International Conference on Computer Aided Verification. Lecture Notes in Computer Science, Springer Verlag, Berkeley, California, USA (July 2012)
7. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaïdi, F.: Invariants for finite instances and beyond. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 61–68. IEEE (2013), http://ieeexplore.ieee.org/document/6679392/
8. Conchon, S., Mebsout, A., Zaïdi, F.: Certificates for parameterized model checking. In: FM. Lecture Notes in Computer Science, vol. 9109, pp. 126–142. Springer (2015)
9. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards smt model checking of array-based systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Automated Reasoning. pp. 67–82. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
10. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. Log. Methods Comput. Sci. **6**(4) (2010). https://doi.org/10.2168/LMCS-6(4:10)2010, https://doi.org/10.2168/LMCS-6(4:10)2010
11. Gurfinkel, A., Shoham, S., Meshman, Y.: Smt-based verification of parameterized systems. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 338–348. FSE 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2950290.2950330, https://doi.org/10.1145/2950290.2950330
12. Karbyshev, A., Bjørner, N., Itzhaky, S., Rinetzky, N., Shoham, S.: Property-directed inference of universal invariants or proving their absence. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 583–602. Springer International Publishing, Cham (2015)
13. Koenig, J.R., Padon, O., Immerman, N., Aiken, A.: First-order quantified separators. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 703–717. ACM (2020). https://doi.org/10.1145/3385412.3386018, https://doi.org/10.1145/3385412.3386018
14. Krstic, S.: Parametrized system verification with guard strengthening and parameter abstraction (2005)
15. Li, Y., Duan, K., Jansen, D.N., Pang, J., Zhang, L., Lv, Y., Cai, S.: An automatic proving approach to parameterized verification. ACM Trans. Comput. Logic **19**(4) (Nov 2018). https://doi.org/10.1145/3232164, https://doi.org/10.1145/3232164

16. Lv, Y., Lin, H., Pan, H.: Computing invariants for parameter abstraction. In: 2007 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007). pp. 29–38 (2007). https://doi.org/10.1109/MEMCOD.2007.371252
17. Mann, M., Irfan, A., Griggio, A., Padon, O., Barrett, C.W.: Counterexample-guided prophecy for model checking modulo the theory of arrays. CoRR **abs/2101.06825** (2021)
18. McMillan, K.L.: Eager abstraction for symbolic model checking. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 191–208. Springer International Publishing, Cham (2018)
19. McMillan, K.L., Padon, O.: Ivy: A multi-modal verification tool for distributed algorithms. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 190–202. Springer International Publishing, Cham (2020)
20. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
21. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: Safety verification by interactive generalization. SIGPLAN Not. **51**(6), 614–630 (Jun 2016), https://doi.org/10.1145/2980983.2908118
22. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2031, pp. 82–97. Springer (2001). https://doi.org/10.1007/3-540-45319-9_7, https://doi.org/10.1007/3-540-45319-9_7
23. Talupur, M., Tuttle, M.R.: Going with the flow: Parameterized verification using message flows. In: 2008 Formal Methods in Computer-Aided Design. pp. 1–8 (2008)
24. VMT-LIB. http://www.vmt-lib.org

A Appendix

A.1 Proofs of section 5

We report here the technical results for the proof of Theorem 3. Note that a state $\tilde{s}$ of $\tilde{C}$ consists of both an assignment of the array variable a and of the index variables $\underline{p} \cup \underline{x}$. With a small abuse of notation we do not distinguish the two cases.

Remark 1 *In the following, since we assumed that the values taken by $\underline{p}$ are different, we also assume that the cardinialities of all index models $\mathcal{M}_I$ are equal or greater than the length of $\underline{p}$. This requirement is not a real limitation: initially, this can be assumed since the property vacuously holds in models with cardinality less than $\underline{p}$. Moreover, the size of $\underline{p}$ can increase only if the property has been proved in all the finite models with lesser size.*

First, we need the following key proposition.

Proposition 5 *Let $\tilde{s}$ be a state of $\tilde{C}$. Let μ be an interpretation of $\underline{p}$ such that $\mu(\underline{p}) = \tilde{s}(\underline{p})$. Let $\phi(\underline{p}, a[\underline{p}])$ be a quantifier free formula which contains only prophecies as index variables. Then, for any state s of C such that $\mathcal{S}(s, \tilde{s})$,*

$$\tilde{s} \models \phi(\underline{p}, a[\underline{p}]) \Leftrightarrow s, \mu \models \phi(\underline{p}, a[\underline{p}])$$

Proof. Note that a model for a function is uniquely determined by the values on its domain. So, if $\mathcal{M}$ is a model for the total functions from $\mathcal{M}_I$ to $\mathcal{M}_E$, then

$$\mathcal{M} \models \phi(\mu(\underline{p}), s(a)[\mu(\underline{p})])$$

is equivalent to

$$\mathcal{N} \models \phi(\mu(\underline{p}), s(a)[\mu(\underline{p})]), \tag{1}$$

where $\mathcal{N}$ is obtained from $\mathcal{M}$ by restricting all the interpretation of the index variables to the substructure of $\mathcal{M}_I$ generated by the elements in $\mu(\underline{p})$ (note that $\mathcal{N}$ is still a model for A_I^E since $\mathcal{T}_I$ is closed under substructure). Similarly, for any model $\mathcal{M}'$

$$\mathcal{M}' \models \phi(\tilde{s}(\underline{p}), \tilde{s}(a)[\tilde{s}(\underline{p})])$$

is equivalent to

$$\mathcal{N}' \models \phi(\tilde{s}(\underline{p}), \tilde{s}(a)[\tilde{s}(\underline{p})]), \tag{2}$$

where $\mathcal{N}'$ defined similarly as above. Since $\mu(\underline{p}) = \tilde{s}(\underline{p})$, we have $\mathcal{N} = \mathcal{N}'$. Moreover, from the definition of $\mathcal{S}$, s and $\tilde{s}$ assign a to the same function, so (1) and (2) are equivalent.

Lemma 1. *If $s \models \iota(a)$, then there exists some $\tilde{s}$ such that $\mathcal{S}(s, \tilde{s})$ and $\tilde{s} \models \tilde{\iota}(\underline{p}, a[\underline{p}])$.*

Proof. Let s be an assignment into a model $\mathcal{M}$, with index domain $\mathcal{M}_I$, and let m be the length of $\underline{i}$. Then,

$$\tilde{\iota}(\underline{p}, a[\underline{p}]) = \bigwedge_{p_{i_1},\ldots,p_{i_m} \subseteq \underline{p}^m} \phi(p_{i_1}, \ldots, p_{i_m}, a[p_{i_1}, \ldots, p_{i_m}]).$$

Let $\tilde{\mu}$ an (injective) assignment of the prophecy variables $\underline{p}$ into $\mathcal{M}_I$. Let $\tilde{s}$ defined as the restriction of s over $\tilde{\mu}(\underline{p})$ and such that $\tilde{s}(\underline{p}) = \tilde{\mu}(\underline{p})$. By definition, $(s, \tilde{s}) \in \mathcal{S}$. Then, by Proposition 5, we have

$$\tilde{s} \models \tilde{\iota}(\underline{p}, a[\underline{p}]) \Leftrightarrow s, \tilde{\mu} \models \tilde{\iota}(\underline{p}, a[\underline{p}]).$$

Since the formula $\iota(a) \to \tilde{\iota}(\tilde{\mu}(\underline{p}), a[\tilde{\mu}(\underline{p})])$ is valid, and $s \models \iota(a)$ by hypothesis, the claim follows. $\square$

Lemma 2. *If $s \cup s' \models \tau(a, a')$, then for every $\tilde{s}$ such that $(s, \tilde{s}) \in \mathcal{S}$, either:*

- *there exists a rule $\tilde{\tau}$ and some $\tilde{s}'$, such that $\tilde{s} \cup \tilde{s}' \models \tilde{\tau}$ and $(s', \tilde{s}') \in \mathcal{S}$; or*
- *there exist a rule $\tilde{\tau}$ and some $\tilde{s}'$, $\tilde{s}''$, such that $\tilde{s} \cup \tilde{s}' \models \tilde{\tau}_S$, $\tilde{s}' \cup \tilde{s}'' \models \tilde{\tau}$, and $(s', \tilde{s}'') \in \mathcal{S}$.*

Proof. We first consider the simpler case of one prophecy variable p and one environmental variable x. By hypothesis,

$$s \cup s' \models \exists i \forall \underline{j}. \psi(i, \underline{j}, a[i], a[\underline{j}], a'[i], a'[\underline{j}]).$$

Hence, there exists an interpretation μ of i in an element of $\mathcal{M}_I$ such that

$$s \cup s' \models \forall \underline{j}. \psi(\mu(i), \underline{j}, a[\mu(i)], a[\underline{j}], a'[\mu(i)], a'[\underline{j}]). \quad (3)$$

Let's also fix a state $\tilde{s}$ of $\tilde{C}$, such that $\mathcal{S}(s, \tilde{s})$. There are now three cases:

- Suppose $\mu(i) = \tilde{s}(p)$. Then, the transition of $\tilde{C}$ labeled by the substitution $i \mapsto p$ is:
$$\tilde{\tau}_{\sigma: i \mapsto p} = \bigwedge_{j \in p, x} \psi(p, \underline{j}, a[p], a[\underline{j}], a'[p], a'[\underline{j}]).$$
Let $\tilde{s}'$ defined as $\tilde{s}'(p) \overset{\text{def}}{=} \mu(i)$ and $\tilde{s}'(a)[\tilde{s}'(p)] \overset{\text{def}}{=} s'(a)[\mu(i)]$. Note that $\mathcal{S}(s', \tilde{s}')$ by definition. Since (3) is universal and $\mu(i) = \tilde{s}(p)$, with an argument similar to Lemma 1, we have that $\tilde{s} \cup \tilde{s}' \models \tilde{\tau}_{\sigma: i \mapsto p}$.
- Suppose $\mu(i) \neq \tilde{s}(p)$ but $\mu(i) = \tilde{s}(x)$ and $\tilde{s}(a)[\tilde{s}(x)] = s(a)[\mu(i)]$. Then, consider the transition labeled by the substitution $i \mapsto x$. Similarly to the first case, we can define $\tilde{s}'$ to be the restriction of s' over p and x, and we have that $\tilde{s} \cup \tilde{s}' \models \tilde{\tau}_{\sigma: i \mapsto x}$. Moreover, $\mathcal{S}(s', \tilde{s}')$ by definition.
- If instead $\mu(i) \neq \tilde{s}(x)$ or $\tilde{s}(a)[\tilde{s}(x)] \neq s(a)[\mu(i)]$, we can reduce to the previous case with a stuttering transition. Let $\tilde{s}'$ defined as $\tilde{s}$ on p, but $\tilde{s}'(x) \overset{\text{def}}{=} \mu(i)$ and $\tilde{s}'(a)[\tilde{s}(x)] \overset{\text{def}}{=} s(a)[\mu(i)]$. Note that we also have $(s, \tilde{s}') \in \mathcal{S}$. Then $\tilde{s} \cup \tilde{s}' \models \tilde{\tau}_S$, and we have reduced to the previous case. So, there exists an $\tilde{s}''$ such that $\tilde{s}' \cup \tilde{s}'' \models \tilde{\tau}_{\sigma: i \mapsto x}$ and $\mathcal{S}(s', \tilde{s}'')$.

In general, suppose $\underline{p} = (p_1, \ldots, p_n)$. By hypothesis

$$s \cup s' \models \exists \underline{i} \forall \underline{j}.\psi(i, \underline{j}, a[i], a[\underline{j}], a'[i], a'[\underline{j}]).$$

Since the the length of $\underline{x}$ is the maximum length of the existentially quantified index variables in the rules of C, we can assume without loss of generality that $\underline{i} = (i_1, \ldots, i_m)$ and $\underline{x} = (x_1, \ldots, x_m)$. By hypothesis there exists an interpretation μ of $\underline{i}$ such that

$$s \cup s' \models \forall \underline{j}.\psi(\mu(\underline{i}), \underline{j}, a[\mu(\underline{i})], a[\underline{j}], a'[\mu(i)], a'[\underline{j}]).$$

Let's also fix a state $\tilde{s}$ of $\tilde{C}$, such that $\mathcal{S}(s, \tilde{s})$. There are again three cases; we omits the details since they are a generalization of the previous ones.

– if $\mu(\underline{i}) \subseteq \tilde{s}(\underline{p})$, then there exist $\underline{p}_{\underline{j}} = (p_{j_1}, \ldots, p_{j_m})$ such that $\mu(\underline{i}) = \tilde{s}(\underline{p}_{\underline{j}})$. We can define $\tilde{s}'$ to be the restriction of s over $\underline{p}$ and we have again $\tilde{s} \cup \tilde{s}' \models \tilde{\tau}_{\sigma:\underline{i} \mapsto \underline{p}_{\underline{j}}}$.
– Suppose now there exists an $0 \leq h < m$ such that $\mu(i_1, \ldots, i_h) = \tilde{s}(p_{j_1}, \ldots, p_{j_h})$, and moreover $\mu(i_{h+1}, \ldots, i_m) = \tilde{s}(x_1, \ldots, x_{m-h})$, and also $\tilde{s}(a)[\tilde{s}(x_1, \ldots, x_{m-h})] = s(a)[\mu(i_{h+1}, \ldots, i_m)]$. Then, if we define $\tilde{s}'$ to be the restriction of s' over $\underline{p} \cup \underline{x}$, we have that $\tilde{s} \cup \tilde{s}' \models \tilde{\tau}_\sigma$ where $\sigma : \underline{i} \mapsto \{p_{j_1}, \ldots, p_{j_h}, x_1, \ldots, x_{m-h}\}$.
– If instead $\mu(i_{h+1}, \ldots, i_m) \neq \tilde{s}(x_1, \ldots, x_{m-h})$ or $\tilde{s}(a)[\tilde{s}(x_1, \ldots, x_{m-h})] \neq s(a)[\mu(i_{h+1}, \ldots, i_{m-h})]$, we can reduce to the previous case with a stuttering transition. Let $\tilde{s}'$ defined as $\tilde{s}$ on $\underline{p}$, but $\tilde{s}'(x_1, \ldots, x_{m-h}) \overset{\text{def}}{=} \mu(i_{h+1}, \ldots, i_m)$ and $\tilde{s}'(a)[\tilde{s}(x_1, \ldots, x_{m-h})] \overset{\text{def}}{=} s(a)[\mu(i_{h+1}, \ldots, i_{m-h})]$. Note that $\mathcal{S}(s, \tilde{s}')$ by definition. Moreover, $\tilde{s} \cup \tilde{s}' \models \tau_S$. We have now reduced to the previous case, and the claim follows.

□

Theorem 6. *The relation $\mathcal{S}$ is a stuttering simulation between C and $\tilde{C}$.*

Proof. Follows directly from Lemmas 1 and 2. □

Theorem 7. *Let C be an array-based transition system, $\tilde{C}$ its parameter abstraction defined in §5. Let $\forall \underline{i}.\Phi(\underline{i}, a[\underline{i}])$ a candidate invariant, and $\underline{p}$ a set of frozen variables with same length as $\underline{i}$. If $\tilde{C} \models \Phi(\underline{p}, a[\underline{p}])$, then $C \models \Phi(\underline{p}, a[\underline{p}])$*

Proof. The statement immediately follows from the fact that stutter simulations preserve reachability, and from Proposition 5. □

A.2 Detailed Experimental Results

We report here detailed results of our experimental evaluation. The results for each group of benchmarks are presented in Tables 2–4. Besides the execution time, the table reports also some statistics on the size of the benchmarks (in number of state variables **V** and transition rules **T**) and on the execution of Lambda (number of iterations of the main loop **I**, number of lemmas added **L** and removed **R**, and maximum size of the concrete instances explored **C**).

Table 2. Experimental results on Protocols benchmarks.

		Size		**Lambda**					**Cubicle**	**MCMT**
Benchmark	**Status**	**V**	**T**	**I**	**L**	**R**	**C**	**Time**	**Time**	**Time**
bakery	safe	1	2	1	3	0	2	0.15	0.03	1.03
bakery_lamport	safe	4	4	1	45	0	2	2.06	0.08	1.03
bakery_lamport_nomax	safe	4	4	1	44	0	2	2.07	UNK	TO
berkeley	safe	1	3	3	6	2	3	0.36	0.03	1.02
chandra_buggy	unsafe	16	27	0	0	0	2	1.54	0.81	3.08
chandra_suggested_inv_01	safe	16	27	1	3	0	2	1.93	0.04	1.03
chandra_suggested_inv_02	safe	16	27	1	1	0	2	0.92	0.03	1.03
chandra_suggested_inv_03	safe	16	27	1	6	0	2	1.69	8.12	92.29
chandra_suggested_inv_04	safe	16	27	1	6	0	2	9.87	18.58	TO
chandra_suggested_inv_05	safe	16	27	1	7	0	2	2.87	31.25	TO
chandra_suggested_inv_06	safe	16	27	6	25	4	4	29.43	0.05	TO
chandra_suggested_inv_07	safe	16	27	1	3	0	2	1.03	TO	15.38
chandra_u_cnj_01	safe	16	27	1	19	0	3	12.95	0.4	TO
chandra_u_cnj_02	safe	16	27	–	–	–	–	TO	90.6	TO
chandra_u_cnj_03	safe	16	27	1	3	0	2	1.83	0.04	TO
chandra_u_cnj_04	safe	16	27	–	–	–	–	TO	TO	TO
chandra_unsafe_01	unsafe	16	27	2	2	1	4	7.41	0.23	1.03
dijkstra	safe	3	7	1	5	0	2	0.34	0.05	1.03
flash_eager	safe	3	5	1	1	0	2	0.17	0.02	1.03
flash_u_cnj_00	safe	41	92	1	81	0	2	61.85	4.49	TO
flash_u_cnj_01	safe	41	92	3	100	1	2	37.02	MO	TO
flash_u_cnj_02	safe	41	92	–	–	–	–	TO	MO	TO
flash_u_cnj_03	safe	41	92	–	–	–	–	TO	MO	TO
flash_u_cnj_04	safe	41	92	–	–	–	–	TO	MO	TO
flash_u_cnj_05	safe	41	92	8	269	5	3	1720.96	MO	TO
flash_u_cnj_06	safe	41	92	–	–	–	–	TO	MO	TO
flash_u_cnj_07	safe	41	92	–	–	–	–	TO	MO	TO
flash_u_cnj_08	safe	41	92	–	–	–	–	TO	MO	TO
german_baukus	safe	9	11	2	65	1	3	5.15	0.09	1463.21
german_pfs	safe	10	13	7	186	11	3	25.97	0.11	521.72
german_undip	safe	9	15	4	51	5	3	4.5	0.06	1.03
illinois	safe	1	9	2	28	0	3	0.86	0.03	1.03
jml	safe	4	8	6	49	4	3	2.25	0.03	1.02
mesi	safe	1	3	1	5	0	2	0.19	0.03	1.03
moesi	safe	1	4	1	3	0	2	0.17	0.03	1.03
mutex	safe	3	2	1	4	0	2	0.16	0.02	1.03
synapse	safe	1	3	3	4	2	3	0.35	0.03	1.03
szymanski_at	safe	4	8	1	59	0	2	2.19	0.04	388.72
ticket	safe	4	2	1	9	0	2	0.19	0.03	1.03
ticket_buggy	unsafe	4	2	0	0	0	2	0.13	0.03	1.03
ticket_o_param	safe	6	5	20	794	16	5	403.01	UNK	TO
xerox	safe	4	20	1	10	0	2	0.51	0.03	1.03
TOTAL	**42**							**34**	**30**	**24**

V: num state vars; **T:** num transition rules; **I:** iterations;
L: num lemmas added; **R:** num lemmas removed; **C:** max concretization size;
TO: time out (> 1h); **MO:** memory out (> 4Gb); **UNK:** unknown result.

Table 3. Experimental results on DynArch benchmarks.

		Size		**Lambda**					**MCMT**
Benchmark	**Status**	**V**	**T**	**I**	**L**	**R**	**C**	**Time**	**Time**
converging_safe_1_a0_b0-src_a0-dst_b0	safe	6	9	1	1	0	1	0.19	1.02
converging_safe_2_a1_b0-src_a1-dst_b0	safe	10	17	1	2	0	1	0.34	1.02
converging_safe_3_a1_b0-src_a1-dst_b0	safe	14	25	1	3	0	1	0.52	1.02
converging_safe_4_a2_b0-src_a2-dst_b0	safe	18	33	1	4	0	1	0.73	1.02
converging_safe_5_a2_b0-src_a2-dst_b0	safe	22	41	1	5	0	1	1.07	1.02
converging_safe_6_a3_b0-src_a3-dst_b0	safe	26	49	1	6	0	1	1.39	4.09
converging_unsafe_1_a0_b0-src_a0-dst_b0	unsafe	6	13	0	0	0	1	0.13	1.03
converging_unsafe_2_a1_b0-src_a1-dst_b0	unsafe	10	25	0	0	0	1	0.2	1.02
converging_unsafe_3_a1_b0-src_a1-dst_b0	unsafe	14	37	0	0	0	1	0.3	1.02
converging_unsafe_4_a2_b0-src_a2-dst_b0	unsafe	18	49	0	0	0	1	0.41	1.02
converging_unsafe_5_a2_b0-src_a2-dst_b0	unsafe	22	61	0	0	0	1	0.53	1.02
converging_unsafe_6_a3_b0-src_a3-dst_b0	unsafe	26	73	0	0	0	1	0.66	2.05
messenger_safe_1_a0_a1-src_a0-dst_a1	safe	10	13	3	11	2	2	0.51	2.05
messenger_safe_2_a0_a2-src_a0-dst_a2	safe	18	25	2	11	1	2	0.84	2361.91
messenger_safe_3_a0_a3-src_a0-dst_a3	safe	26	37	2	12	1	2	1.46	TO
messenger_safe_4_a0_a4-src_a0-dst_a4	safe	34	49	2	14	1	2	2.47	TO
messenger_unsafe_1_a0_a1-src_a0-dst_a1	unsafe	8	13	0	0	0	1	0.15	1.03
messenger_unsafe_2_a0_a2-src_a0-dst_a2	unsafe	14	25	0	0	0	1	0.43	14.35
messenger_unsafe_3_a0_a3-src_a0-dst_a3	unsafe	20	37	0	0	0	1	1.25	1352.5
messenger_unsafe_4_a0_a4-src_a0-dst_a4	unsafe	26	49	0	0	0	1	4.25	TO
messenger_unsafe_5_a0_a5-src_a0-dst_a5	unsafe	32	61	0	0	0	1	8.18	TO
network_safe_1_d_e-src_d-dst_e	safe	10	20	6	31	7	3	4.44	1.02
network_safe_2_d_e-src_d-dst_e	safe	13	28	4	42	4	3	4.87	10.27
network_safe_3_d_e-src_d-dst_e	safe	16	36	5	45	6	3	10.19	53.29
network_safe_4_d_e-src_d-dst_e	safe	19	44	4	45	3	2	7.47	310.28
network_safe_5_d_e-src_d-dst_e	safe	22	52	5	47	6	3	17.81	1675.7
network_safe_6_d_e-src_d-dst_e	safe	25	60	7	53	7	3	31.52	TO
network_unsafe_1_d_e-src_d-dst_e	unsafe	9	19	0	0	0	1	0.16	1.02
network_unsafe_2_d_e-src_d-dst_e	unsafe	12	27	0	0	0	1	0.26	1.02
network_unsafe_3_d_e-src_d-dst_e	unsafe	15	35	0	0	0	1	0.36	4.09
network_unsafe_4_d_e-src_d-dst_e	unsafe	18	43	0	0	0	1	0.48	20.47
network_unsafe_5_d_e-src_d-dst_e	unsafe	21	51	0	0	0	1	0.6	84.07
network_unsafe_6_d_e-src_d-dst_e	unsafe	24	59	0	0	0	1	0.78	309.26
ring_safe_1_s_d-src_s-dst_d	safe	4	7	1	2	0	1	0.13	1.02
ring_safe_2_s_d-src_s-dst_d	safe	6	13	3	11	1	2	0.6	1.02
ring_safe_3_s_d-src_s-dst_d	safe	8	19	17	36	19	4	18.84	1.02
ring_safe_4_s_d-src_s-dst_d	safe	10	25	76	157	114	5	631.81	3.08
ring_safe_5_s_d-src_s-dst_d	safe	12	31	–	–	–	–	MO	49.16
ring_safe_6_s_d-src_s-dst_d	safe	14	37	–	–	–	–	TO	MO
ring_unsafe_1_s_d-src_s-dst_d	unsafe	4	7	2	2	1	2	0.27	1.02
ring_unsafe_2_s_d-src_s-dst_d	unsafe	6	13	6	11	4	3	2.27	1.03
ring_unsafe_3_s_d-src_s-dst_d	unsafe	8	19	25	39	33	4	52.07	4.1
ring_unsafe_4_s_d-src_s-dst_d	unsafe	10	25	103	225	158	5	2285.03	215.17
ring_unsafe_5_s_d-src_s-dst_d	unsafe	12	31	–	–	–	–	MO	MO
ring_unsafe_6_s_d-src_s-dst_d	unsafe	14	37	–	–	–	–	TO	MO
sequence_safe_1_s_d-src_s-dst_d	safe	4	5	1	2	0	1	0.11	1.03
sequence_safe_2_s_d-src_s-dst_d	safe	6	11	11	21	8	4	4.11	1.03
sequence_safe_3_s_d-src_s-dst_d	safe	8	17	–	–	–	–	TO	1.02
sequence_safe_4_s_d-src_s-dst_d	safe	10	23	–	–	–	–	MO	3.07
sequence_safe_5_s_d-src_s-dst_d	safe	12	29	–	–	–	–	MO	44.07
sequence_safe_6_s_d-src_s-dst_d	safe	14	35	–	–	–	–	TO	MO
sequence_unsafe_1_s_d-src_s-dst_d	unsafe	4	5	0	0	0	1	0.08	1.03
sequence_unsafe_2_s_d-src_s-dst_d	unsafe	6	11	2	2	1	2	0.35	1.02
sequence_unsafe_3_s_d-src_s-dst_d	unsafe	8	17	5	7	4	3	2.49	1.03
sequence_unsafe_4_s_d-src_s-dst_d	unsafe	10	23	11	20	15	4	13.95	1.02
sequence_unsafe_5_s_d-src_s-dst_d	unsafe	12	29	28	61	52	5	234.49	5.12
sequence_unsafe_6_s_d-src_s-dst_d	unsafe	14	35	–	–	–	–	TO	104.44
TOTAL	**57**							**48**	**48**

V: num state vars; **T:** num transition rules; **I:** iterations;
L: num lemmas added; **R:** num lemmas removed; **C:** max concretization size;
TO: time out (> 1h); **MO:** memory out (> 4Gb); **UNK:** unknown result.

Table 4. Experimental results on Trains benchmarks.

		Size		**Lambda**				
Benchmark	**Status**	**V**	**T**	**I**	**L**	**R**	**C**	**Time**
trains-switches-sorted-real_0	safe	6	5	1	4	0	3	0.51
trains-switches-sorted-real_1	safe	6	5	1	1	0	1	0.16
trains-switches-sorted-real_2	unsafe	6	5	0	0	0	3	0.43
trains-switches-sorted-real_3	safe	6	5	1	1	0	1	0.17
trains-switches-sorted-real_4	safe	6	5	1	1	0	1	0.16
trains-switches-sorted-real_5	safe	6	5	1	1	0	1	0.17
trains-switches-sorted-real_6	safe	6	5	1	1	0	1	0.17
trains-switches-sorted-real_7	unsafe	6	5	0	0	0	4	1.64
trains-switches-sorted-real_8	unsafe	6	5	0	0	0	8	12.31
trains-switches-sorted-real_9	safe	6	5	1	2	0	4	0.79
trains-switches-sorted-real_10	safe	6	5	1	2	0	3	0.4
trains-switches-sorted-real_11	safe	6	5	1	2	0	3	0.41
trains-switches-sorted-real_12	safe	6	5	1	7	0	4	1.12
trains-switches-sorted-real_13	safe	6	5	1	6	0	3	0.52
trains-switches-sorted-real_14	safe	6	5	1	7	0	3	0.71
trains-switches-sorted-real_15	safe	6	5	1	1	0	2	0.24
trains-switches-sorted-real_16	safe	6	5	1	2	0	2	0.28
TOTAL	**17**							**17**

V: num state vars; **T:** num transition rules; **I:** iterations;
L: num lemmas added; **R:** num lemmas removed; **C:** max concretization size;
TO: time out ($>$ 1h); **MO:** memory out ($>$ 4Gb); **UNK:** unknown result.