UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

**Doctoral Programme in**
**Information Engineering and Computer Science**

# SMT-based Verification of Parameterized Systems

Gianluca Redondi

Advisor
Alessandro Cimatti
Fondazione Bruno Kessler

Co-Advisor
Alberto Griggio
Fondazione Bruno Kessler

April 2024

# Abstract

*SMT-based verification analyzes reachability for transition systems represented by SMT formulae. Depending on the theories and the kinds of systems considered, various approaches have been proposed. Together, they form the Verification Modulo Theory (VMT) framework.*

*This thesis delves into SMT-based verification of parameterized systems, emphasizing the challenges and novel solutions in verifying systems with an unbounded number of components. In this thesis, we first introduce a general framework to model such systems. Then, we introduce two novel algorithms that leverage the strengths of SMT for the verification of parameterized systems, focusing on the automation and reduction of computational complexity inherent in such tasks.*

*These algorithms are designed to improve upon existing verification methods by offering enhanced scalability and automation, making them particularly suited for the analysis of distributed systems, network protocols, and concurrent programming models where traditional approaches may fail.*

*Moreover, we introduce an algorithm for compositional verification that advances the capability to modularly verify complex systems by decomposing the verification task into smaller, more manageable sub-tasks.*

*Additionally, we discuss the potential and ongoing application of these algorithms in an industrial project focusing on the design of interlocking logic. This particular application demonstrates the practical utility of our algorithms in a real-world setting, highlighting their effectiveness in improving the safety and reliability of critical infrastructure.*

*The theoretical advancements proposed in this thesis are complemented by a rigorous experimental evaluation, demonstrating the applicability and effectiveness of our methods across a range of verification scenarios. Our work is implemented within an extended framework of the MathSAT SMT solver, facilitating its integration into existing verification workflows.*

*Overall, this research contributes to the theoretical underpinnings of Verification Modulo Theories (VMT) and offers tools and methodologies for the verification community, enhancing the capability to verify complex parameterized systems with greater efficiency and reliability.*

## Keywords

*Parameterized Verification, SMT, Inductive Invariant, Quantifiers*

# Contents

# Chapter 1

# Introduction

In today's society, where computer and software systems are integral to safety-critical tasks in domains such as transportation, healthcare, and avionics, the reliability of these systems is not just preferable but essential. Failures in such systems can lead to severe consequences, including loss of life and substantial economic repercussions. Given the widespread impact of these technologies, even non-safety-critical systems can have profound global effects if compromised. Hence, it becomes crucial to integrate various methods early in the development lifecycle to minimize potential failures.

Among these methods, testing is the most commonly used. Testing involves running a system under specific conditions and observing its outputs to detect any anomalies or failures. However, while testing is invaluable for identifying existing defects, it does not guarantee the absence of errors. This limitation is particularly critical in safety-sensitive systems where even a minor undetected flaw could lead to catastrophic outcomes. In such scenarios, stakeholders require not just evidence of system robustness but formal assurances that errors will not occur. This demand for higher reliability is where formal verification methods, particularly automated ones, become indispensable.

Model checking is a formal verification technique that involves the exhaustive exploration of the possible states of a system to ensure it behaves correctly according to its specifications. Historically, explicit-state exploration of systems has evolved from Binary Decision Diagrams-based approaches to other symbolic approaches based on the Boolean satisfiability problem (SAT). Model checking algorithms can automatically verify whether a formal model of a system adheres to specified properties, often represented in logical formulas. By exploring all potential states, model checking can conclusively identify violations of specifications or affirm their correctness, thereby providing insights into system reliability.

Model checking has been instrumental in various high-stakes applications, such as verifying the correctness of aerospace software where a single error could result in loss of life. It has also proven effective in the automotive industry, where it is used to ensure that control software for vehicles adheres to critical safety standards.

However, the binary nature of SAT limited its applications in verifying systems with complex data types and conditions. The advent of Satisfiability Modulo Theories (SMT) marked a crucial evolution, extending verification capabilities to encompass theories

like arithmetic, arrays, and bit-vectors. This development was not just incremental but represented a paradigm shift, offering a more expressive framework suitable for complex systems. Over the past decade, significant progress in SMT solvers' handling of quantified expressions has made SMT-based verification a frontrunner, particularly for parameterized systems with inherently unbounded characteristics due to variable components.

Parameterized systems are models that characterize a class of systems with a potentially infinite number of configurations, typically defined by the number of components or processes they encompass. These systems cannot be exhaustively explored using traditional finite-state methods due to their scalability issues, as each component might interact with others in a myriad of ways depending on system size and configuration.

An important example for parameterized systems are railway interlocking systems. These are safety-critical applications where the configuration can vary based on the number of tracks and junctions involved. Each configuration requires precise control to ensure safe and efficient train movement, which is ideal for SMT-based verification due to the complex interdependencies and the critical need for error-proof operation.

Traditional verification methods, while effective for finite-state or quantifier-free systems, often falter with the complexity presented by parameterized systems. This thesis, therefore, focuses on developing SMT-based verification methodologies that enhance scalability, efficiency, and automation. By introducing novel algorithms and adapting SMT approaches to tackle the unique challenges of parameterized systems, this work aims to deliver robust solutions that ensure the safety and reliability of critical software and hardware systems, thereby safeguarding societal welfare.

## 1.1 Contributions

The contributions of this thesis are manifold:

1. **Theoretical Exploration:** A formalism for modeling parameterized systems within an SMT framework is described, offering a foundation for subsequent algorithmic development.

2. **Algorithm Development:** The introduction of novel SMT-based verification algorithms designed to automatically verify properties of systems. We first present two algorithms for proving (or disproving) safety properties of parameterized systems:

   - **Updria:** An algorithm that extends the UPDR [81] algorithm with predicate abstraction to handle a general SMT theory representing the data of the parameterized system. This approach leverages several theoretical properties of the original algorithm but necessitates frequent (and costly) quantified queries to an external solver.
   - **Lambda:** A procedure that first constrains the cardinality of the parameter, allowing for quantifier-free system verification via existing methods, and

then attempts to generalize the resulting inductive invariant to a quantified candidate for the original system. This algorithm explores two avenues for validating the generalized invariant: an extension of Parameter Abstraction [88] for constructing a quantifier-free system and property verification, and a resource-bounded approach to quantified SMT reasoning, aiming to mitigate the risk of divergence in validity checks.

Additionally, a third algorithm applicable to asynchronous systems is proposed, further extending the versatility of our approach.

3. **Application:** The description of ongoing project applying the latter algorithms to an industrial setting related to the design of interlocking logics for railways is detailed, showcasing the practical applicability of our methods.

4. **Experimental Validation:** An exhaustive experimental assessment underscores the effectiveness and practical utility of the proposed algorithms. This includes case studies and a comparative analysis with related techniques.

## 1.2 Thesis Structure

This thesis is organized as follows:

- Chapter 2 lays the groundwork by providing the necessary background on Satisfiability Modulo Theories (SMT) and Verification Modulo Theories (VMT). It introduces the formalism used for describing parameterized systems and formally defines the problem addressed in the thesis.

- Chapter 3 details the novel SMT-based verification algorithms for parameterized systems developed in this study. It covers their theoretical foundations, design rationale, and the specific verification challenges they address.

- Chapter 4 explores compositional verification strategies with a focus on an algorithm tailored for asynchronously composed symbolic transition systems. Although the algorithm is discussed in a general context, emphasis is placed on its application to parameterized systems.

- Chapter 5 describes the application of the algorithms introduced in the previous chapters to a project concerning the design of railway interlocking logic. This chapter illustrates the practical implications and benefits of the developed methodologies.

- Chapter 6 presents an exhaustive experimental evaluation of the algorithms, employing diverse benchmarks to assess their efficacy, efficiency, and practical utility. The evaluation includes a comparative analysis with existing techniques to underscore the novel contributions of this research.

- Chapter 7 concludes the thesis, summarizing the contributions and considering the broader implications of this research. It also outlines potential future research directions, indicating areas for further investigation or application.

# Part I

# Background

# Chapter 2

# Background notions

## 2.1 Satisfiability Modulo Theory

Satisfiability Modulo Theories (SMT) addresses the problem of determining the satisfiability of first-order formulas with respect to various underlying theories or their combinations. At its core, SMT extends the concept of propositional satisfiability (SAT) by incorporating the ability to handle rich theories like linear and nonlinear arithmetic, bit-vectors, arrays, and uninterpreted functions, among others. This inclusion allows for a more expressive framework, enabling the analysis and verification of more complex and detailed system properties.

In this section, we introduce basic definitions and notation that will be used throughout all chapters. For a more detailed introduction to SAT and SMT, see [12, 87, 126].

In this thesis, we work in a multi-sorted setting: we outline here the main differences with the standard case. A sorted signature $\Sigma$ is given by a set of sorts and a set of sorted functions, such that the domain and range sorts of every function in $\Sigma$ is also in $\Sigma$. A multi-sorted structure is a map that takes every sort symbol to a set called its universe (disjoint from other universes) and every function symbol to a function between the universes of the corresponding sorts. Predicates are seen as functions to the sort `bool`, while constants are 0-ary functions.

A term is a variable or an $n$-ary function symbol applied to n arguments of the appropriate sort. An atom is an $n$-ary predicate applied to $n$ arguments of the appropriate sort, and a literal is an atom or the negation of an atom. A disjunction of literals is called a clause. A conjunction of clauses is a formula in conjunctive normal form (CNF). A formula is an atom, the negation of a formula, the conjunction or disjunction of two formulas, or the application of a universal quantifier to a formula; the other logical connectives can be expressed in terms of the other logical symbols.

A term, atom, or formula is ground if it does not contain variables. A formula is closed if all variables are bound by a quantifier. Variables that are not bound by a quantifier are called free. A substitution is a (partial) mapping from variables to terms. A formula that is obtained by substituting the variables in the body of a universally quantified formula with ground terms is called an instance of the quantified formula. If $F$ denotes a set of formulae, with a slight abuse of notation, we may use again the

symbol $F$ to denote instead the formula given by the conjunction of all the elements in $F$. If a formula has free variables in it, the universal (resp. existential) closure of the formula is the formula obtained by application of a universal (resp. existential) quantification to all the free variables.

In this thesis, a theory $\mathcal{T}$ is a pair $\mathcal{T} = (\Sigma, \mathcal{C})$, where $\Sigma$ is a first-order signature and $\mathcal{C}$ is a class of structures over $\Sigma$ (called the *models* of the theory). Given a structure $\mathcal{M}$ over $\Sigma$, the restriction of $\mathcal{M}$ over a sorted vocabulary $\Sigma' \subset \Sigma$ is given by the universe restriction of $\mathcal{M}$ to $\Sigma'$. Given two theories $\mathcal{T} = (\Sigma, \mathcal{C})$ and $\mathcal{T}' = (\Sigma', \mathcal{C}')$, the *combination* of the two theories is a theory whose signature is $\Sigma \cup \Sigma'$, and a model for it is a structure such that its restriction onto $\Sigma$ is in $\mathcal{C}$ and its restriction onto $\Sigma'$ is in $\mathcal{C}'$. A structure is finite iff all its universes are finite sets.

If $\Sigma$ is a signature, and $X$ is a set of symbols not contained in $\Sigma$, we write $\phi(X)$ to denote a formula in the new signature expanded with such symbols (denoted with $\Sigma(X)$). Symbols in $\Sigma$ are usually referred to as *interpreted* since their values are fixed by the models in $\mathcal{C}$. Thus, with $\phi(X)$ we mean that the only non-interpreted symbols that occur in $\phi$ are in $X$. If $\phi$ is a formula, $t$ is a term and $x$ is a symbol that occurs in $\phi$, we write $\phi[x/t]$ for the substitution of every occurrence of $x$ with $t$. If $T$ and $X$ are vectors of the same length, we write $\phi[X/T]$ for the simultaneous substitution of each $x_i$ with the corresponding term $t_i$.

A ground formula $\phi$ is satisfiable with respect to $\mathcal{T}$, or $\mathcal{T}$-satisfiable, if there exists a $\mathcal{T}$-model for it, i.e., a structure $\mathcal{M}$ in the class $\mathcal{C}$, such that $\mathcal{M} \models \phi$. A formula is $\mathcal{T}$-valid if it holds in each model of the theory. Note that a formula is $\mathcal{T}$-valid if and only if its negation is not $\mathcal{T}$-satisfiable.

Given a structure $\mathcal{M}$ of the theory, a valuation $s$ for a formula $\phi(X)$ maps the function symbols in the formula to functions and relations over the universes of $\mathcal{M}$ specified by the symbols' sorts. A valuation under which a formula evaluates to true is called a model for $\phi(X)$, written $s, \mathcal{M} \models \phi(X)$. We say that a formula is *satisfiable* if there exists a structure $\mathcal{M}$ and a valuation $s$ such that $s, \mathcal{M} \models \phi(X)$. Otherwise, the formula is unsatisfiable.

The satisfiability modulo theories (SMT) problem refers to the question of whether a given formula is $\mathcal{T}$-satisfiable. A decision procedure for (a fragment of) a theory $\mathcal{T}$ is an algorithm that determines whether a given formula in $\mathcal{T}$ is satisfiable. If a decision procedure exists, (the fragment of) the theory is decidable.

### 2.1.1 Theories of interest

We now introduce the main theories that we will use in this thesis.

The Theory of Equality and Uninterpreted Functions (EUF), represented as $\mathcal{T}_{\text{EUF}}$, serves as a foundational theory in the field of verification. Its signature contains the equality symbol and a set of function symbols, while the models are all possible sets of functions. The primary utility of $\mathcal{T}_{\text{EUF}}$ lies in its capacity to abstract the functionalities of complex functions, facilitating a more manageable approach to verification tasks. Moreover, other important theories are reduced to EUF. Although the full theory is undecidable, since it subsumes first-order logic, its quantifier-free fragment is decidable.

The standard method to address the decidability of the quantifier-free EUF involves the computation of congruence closures. Additional details on satisfiability procedure and complexity can be found in [105].

Another important theory, originating from McCarthy's seminal work, [97], is the theory of arrays often used in modeling and reasoning about complex data structures such as arrays in programming languages.

Denoted as $\mathcal{T}_A$, this theory has uninterpreted sorts for indices ($\tau_I$) and elements ($\tau_E$). The signature $\Sigma_A$ of the theory contains, alongside the index and element sort, and array sort denoted by $(\tau_I \Rightarrow \tau_E)$. Moreover, two function symbols are defined. The read function, denoted as $\texttt{rd} : (\tau_I \Rightarrow \tau_E) \times \tau_I \to \tau_E$, is used for accessing an element at a given index within an array; the write function, $\texttt{wr} : (\tau_I \Rightarrow \tau_E) \times \tau_I \times \tau_E \to (\tau_I \Rightarrow \tau_E)$, defines the result of updating an array at a specific index with a new element. The class of models for the theory is given by interpreting the array sorts as functions from a universe of the index theory to a universe of the element theory. Moreover, the following axioms should hold:

$$\forall a : (\tau_I \Rightarrow \tau_E), i : \tau_I, v : \tau_E.\ \mathrm{rd}(\mathrm{wr}(a, i, v), i) = v,$$
$$\forall a : (\tau_I \Rightarrow \tau_E), i : \tau_I, j : \tau_I, v : \tau_E.\ i \neq j \to \mathrm{rd}(\mathrm{wr}(a, i, v), j) = \mathrm{rd}(a, j).$$

Additionally, the signature of the theory can have the $\texttt{const} : \tau_E \Rightarrow (\tau_I \Rightarrow \tau_E)$ function symbol, enabling the definition of constant arrays where every index maps to the same element. This is axiomatized as follows:

$$\forall i : \tau_I, v : \tau_E.\ \mathrm{rd}(\mathrm{const}(v), i) = v.$$

While the full theory, inclusive of quantifiers, is undecidable, its quantifier-free fragment and certain quantified fragments, such as the array property fragment, maintain decidability, offering practical avenues for automated reasoning in this domain [80, 122, 22].

Another important theory in the realm of formal verification is the Theory of Linear Integer Arithmetic ($\mathcal{T}_{\mathrm{LIA}}$) for integer domains, and similarly, the Theory of Linear Real Arithmetic ($\mathcal{T}_{\mathrm{LRA}}$) for real numbers. These theories deal with variables that can assume integer or real values, respectively, and are constrained by linear equations and inequalities. The signature of $\mathcal{T}_{\mathrm{LIA}}$ and $\mathcal{T}_{\mathrm{LRA}}$ includes the usual arithmetic operators: addition ($+$), subtraction ($-$), and multiplication by a constant, alongside relational operators ($=, \neq, <, \leq, >, \geq$). The class of models of those theories is a singleton containing only the standard set of integers or real numbers, with the obvious interpretation of the symbols.

Linear arithmetic is fundamental in verifying properties that involve numerical calculations, constraints, and optimizations. For instance, $\mathcal{T}_{\mathrm{LIA}}$ is essential in verifying array bounds, loop counters, and arithmetic properties in algorithms, whereas $\mathcal{T}_{\mathrm{LRA}}$ is often applied in the verification of hybrid systems, involving continuous changes.

The decidability of both $\mathcal{T}_{\mathrm{LIA}}$ and $\mathcal{T}_{\mathrm{LRA}}$ in their quantifier-free fragments provides a solid foundation for their application in SMT solving. Techniques such as the Simplex algorithm are employed for $\mathcal{T}_{\mathrm{LRA}}$, while extensions or variations are used for $\mathcal{T}_{\mathrm{LIA}}$,

ensuring efficient satisfiability checks. References detailing these procedures include [15, 2, 20].

In practical verification tasks, it is often necessary to reason about systems that utilize multiple theories, necessitating a combined approach. The Nelson-Oppen framework [104, 17] provides a methodology for combining decision procedures of different theories, assuming they are stably infinite. If this does not hold, other approaches for theory combination are possible [16, 28]. An example of a combined theory is the Theory of Arrays with Integer Indices and Real Elements, which allows for reasoning about arrays indexed by integers with real-numbered elements, showcasing the versatility of combined theories in modeling and verifying diverse systems.

The SMT-LIB standard [9] describes all standard and combined theories in the context of SMT.

### 2.1.2 Interpolants

Craig interpolants are a tool from mathematical logic, with significant implications for SMT and model checking. An interpolant can be seen as a logical bridge, showing shared logical structure between two contradictory formulae, aiding in isolating and understanding inconsistencies within the two.

**Definition 1.** *Given a pair of formulae $(\phi, \psi)$, with $\phi \wedge \psi$ is unsatisfiable, a Craig interpolant for $(\phi, \psi)$ is a formula $\iota$ satisfying:*

*(i)* $\phi \models \iota$;

*(ii)* $\iota \wedge \psi$ *is unsatisfiable;*

*(iii) the only uninterpreted symbols occurring in $\iota$ occurs both $\phi$ and $\psi$.*

The concept extends also to sequences of formulae as follows:

**Definition 2.** *For an ordered sequence of formulae $\{\phi_0, \ldots, \phi_n\}$, such that their conjunction in unsatisfiable, an interpolant sequence $\{\iota_1, \ldots, \iota_n\}$ is a sequence of formulae such that:*

*(i)* $\bigwedge_{0 \leq k < i} \phi_k \models \iota_i$;

*(ii)* $\bigwedge_{i \leq k \leq n} \phi_k \wedge \iota_i$ *is unsatisfiable;*

*(iii)* $\iota_i \wedge \phi_{i+1} \models \iota_{i+1}$ *for all* $1 \leq i < n$;

*(iv) all the uninterpreted symbols occurring in $\iota_i$ occur in both $\bigwedge_{0 \leq k < i} \phi_k$ and $\bigwedge_{i \leq k \leq n} \phi_k$.*

The computation of interpolants and sequence interpolants has been effectively streamlined across various theories, playing a pivotal role in decomposing and analyzing complex logical structures and system behaviors, with notable techniques and implementations documented, for instance, in [38, 98, 103].

Although Craig interpolants always exist in first-order logic, their applicability is not uniform across all logical theories. Specifically, the existence and computation of quantifier-free interpolants pose significant challenges in certain contexts. The ability to generate such interpolants depends on the specific properties and structure of the theory under consideration. For instance, while theories like Linear Real Arithmetic and EUF support the construction of quantifier-free interpolants, more complex theories involving non-linear arithmetic or specific data structures may not guarantee this property [64]. This limitation can significantly impact the effectiveness of interpolation-based techniques in certain verification tasks, particularly where the preservation of a quantifier-free context is crucial for subsequent analysis or processing[33, 24].

The computation of interpolants becomes notably more complex when the involved theories or formulae include quantifiers. The presence of quantifiers introduces additional logical layers that must be accounted for, complicating the interpolation process. Although there are methods for generating interpolants in the presence of quantifiers, such as using quantifier elimination techniques or specialized decision procedures, these approaches often come with a high computational cost and may not always be feasible or efficient in practice [30, 101].

### 2.1.3 Handling Quantifiers in SMT Solving

Quantified formulas pose significant challenges in Satisfiability Modulo Theories (SMT) solving due to their inherent complexity and the infinite nature of their domains. A predominant method for managing quantified formulas in SMT solvers is through instantiation-based approaches. First, each quantified formula is transformed into an equisatisfiable one, that only contains universal quantifiers, thanks to the process of Skolemization [106]. Then, these approaches involve generating instances of quantified formulas by substituting the universally quantified variables with concrete terms, aiming to reduce the problem to a quantifier-free one that is more tractable for SMT solvers. If the SMT solver is capable of proving that an instance of a quantified formula is unsatisfiable, then also the original formula must be.

The instantiation-based method primarily relies on selectively generating instances of the quantified formulas that are relevant to the current solving context. This relevance is determined through a combination of heuristic and systematic strategies, such as E-matching [116, 49], which identifies potential instantiations by matching patterns in the quantified formula with terms present in the current formula being solved. E-matching plays a critical role in identifying effective instances by ensuring that only those terms that could potentially lead to a conflict or aid in further propagation are considered for instantiation. This selective approach helps in mitigating the combinatorial explosion of possible instances, thereby enhancing the efficiency and scalability of the solver. However, other instantiation-based approaches are possible [62, 117].

In summary, the instantiation-based approach to handling quantifiers in SMT solving represents the main tactic toward addressing the challenges posed by quantified formulas. Through strategic instance generation and integration with existing solving frameworks, SMT solvers can achieve greater efficiency and effectiveness in proving

unsatisfiability of quantified formulae.

The dual method of finite model finding in SMT solving is instead a technique aimed at verifying the satisfiability of quantified formulas through the construction of explicit models. Traditionally tackled by MACE-style model finding [41], which converts first-order logic into propositional logic for resolution, recent advancements have been directed towards enhancing efficiency, scalability, and tight integration with SMT solvers. These advancements include the introduction of term definitions and incremental SAT solving, which simplify clauses by introducing fresh constants for deep ground terms and optimize the search process through the reuse of search information across consecutive model sizes. Furthermore, static symmetry reduction, applied by adding extra constraints to the SAT problem, curtails the search space by circumventing isomorphic models. Sort inference, which automatically deduces sort information for unsorted problems, enables more detailed symmetry reduction and potentially diminishes the complexity of the model search problem. Moreover, the integration of solvers for sort cardinality constraints, coupled with the application of quantifier instantiation over finite domains, circumvents the need for the explicit introduction of domain constants, thereby facilitating a more streamlined handling of quantified formulas [117, 118].

## 2.2 Verification Modulo Theory

Symbolic model checking offers a significant advancement over traditional explicit-state model checking by symbolically representing and manipulating state spaces, thereby enabling the analysis of systems that are impractical to verify explicitly.

By leveraging SMT techniques, symbolic model checking continues to evolve, offering enhanced scalability, precision, and the ability to handle an ever-widening scope of verification challenges. The Verification Modulo Theories [32, 39] framework describes this synergy.

In the following, we consider a fixed theory $\mathcal{T} = (\Sigma, \mathcal{C})$. For a set of variables $X$, the notation $X'$ represents $\{x' | x \in X\}$, indicating successor state variables, and $X^i$ denotes $\{x^i | x \in X\}$ for $i \in \mathbb{N}$, signifying states at different time steps. When we have a formula $F(X)$, $F'$ and $F^i$ are the results of the substitution of the variables $X$ replaced by those in $X'$ and $X^i$, respectively.

**Definition 3.** *A (symbolic) transition system $C$ is a triple $C = (X, I(X), T(X, X'))$ where:*

- *$X$ are a set of state variables;*
- *$I(X)$ is the initial formula;*
- *$T(X, X')$ is the transition formula.*

Given a model $\mathcal{M}$ for $\mathcal{T}$, a state is a valuation $s$ of the state variables $X$ in the universe of $\mathcal{M}$. A state is initial iff it is a model of $I(X)$, i.e. $\mathcal{M}, s \models I(X)$. A couple of states $(\mathcal{M}, s)$, $(\mathcal{M}, s')$ denote a transition iff $\mathcal{M}, s, s' \models T(X, X')$, also denoted as $T(s, s')$. When clear from the context, we may omit the model $\mathcal{M}$ from the notation, and consider it fixed. A variable $x$ is a *frozen* variable iff its value is fixed during every evolution of the system, i.e. if the constraint $x' = x$ is implied by $T$. 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 $i$. If $\pi$ is a path, we denote with $\pi[j]$ the $j$-th element of $\pi$. A state $s$ is reachable iff there exists a path $\pi$ such that $\pi[i] = s$ for some $i$.

A formula $\phi(X)$ is an invariant of the transition system $C = (X, I(X), T(X, X'))$ iff it holds in all the reachable states. Following the standard model checking notation, we denote this with $C \models \phi(X)$.[1]

A formula $\phi(X)$ is an inductive invariant for $C$ iff $I(X) \models \phi(X)$ and $\phi(X) \wedge T(X, X') \models \phi(X')$. Given a formula $\psi(X)$, we say that a formula $\phi$ is an inductive invariant for $\psi$ and $C$ if $\phi$ is an inductive invariant and $\phi \models \psi$.

It is easy to see that every inductive invariant for a transition system is also an invariant for it, whereas the vice-versa is not true. Moreover, *checking* that a formula is an inductive invariant for a system can be reduced to the problem of $\mathcal{T}$-satisfiability.

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

However, only trivial invariants are inductive themselves, and many symbolic algorithms were developed to synthesize inductive invariants for symbolic transition systems and generic formulae.

### 2.2.1 Symbolic model checking

In this section, we summarize the main basic techniques in the context of checking that a formula is an invariant for a symbolic transition system.

#### Bounded Model Checking

Bounded Model Checking (BMC) is a verification technique pioneered in [10] for finite-state systems, with subsequent extensions to infinite-state systems leveraging SMT solvers. BMC operates by translating the problem of finding violations of a property over a system into the problem of checking the satisfiability of a specific formula up to a certain depth.

The core of BMC for a symbolic transition system $C$, involves checking whether a property $\phi$ can be violated within $k$ steps. The BMC formula for a given bound $k$ is constructed as follows:

$$\mathrm{BMC}(C, \phi, k) = I(X_0) \wedge \left( \bigwedge_{i=0}^{k-1} T(X_i, X_{i+1}) \right) \wedge \neg\phi(X_k) \tag{2.1}$$

The formula is satisfiable if and only if there exists an initial state $s_0$ that satisfies $I$, and there exists a sequence of states $s_1, \ldots, s_k$ following the transition relation $T$, leading to a state $s_k$ where the property $\phi$ is violated.

BMC's approach to incrementally increasing the bound $k$ allows for an effective exploration of the system's state space, with the potential to detect errors at minimal depths. For infinite-state systems, the transition formula $T$ and property $\phi$ are expressed using theories compatible with SMT solvers, enabling the handling of complex data types and operations. BMC techniques can be adapted also for searching for lasso-shaped falsification of LTL properties [11].

Despite its computational efficiency for early error detection, the effectiveness of BMC is contingent upon the power of the underlying SAT or SMT solver. The technique, however, does not guarantee in general the absence of errors beyond the explored bound $k$, necessitating complementary verification methods for complete coverage [43]. The diameter [43] of a system is a bound $k$ that represents the shortest path length, beyond which extending the bound does not yield additional counterexamples to the property being verified. Diameters do not exist in general for infinite-state systems, and even for finite-state ones, their computation is very expensive.

#### K-induction

K-induction is an extension of the classical mathematical induction principle, tailored for the verification of properties in both finite and infinite-state systems. Used together

with, Bounded Model Checking (BMC), K-induction aims to prove properties for all possible execution paths. The technique consists of two main steps: the base case and the inductive step.

The base case of K-induction is a call to BMC($C$, $\phi$, $k$) to check that a property $\phi$ holds up to depth $k$. Then, the inductive step assumes that the property $\phi$ holds for any sequence of $k$ states and proves that it continues to hold for the $(k+1)$-th state. This is represented by:

$$\text{Inductive}(C, \phi, k) = \left( \bigwedge_{i=0}^{k} \phi(X_i) \wedge T(X_k, X_{k+1}) \right) \wedge \neg\phi(X_{k+1}) \tag{2.2}$$

If the formula is unsatisfiable, then the formula $\phi$ is an invariant of the system: assuming that $\phi$ holds for a sequence of $k+1$ consecutive states, if no model is satisfying its negation after the next transition, then the property's invariance over the system is demonstrated.

The selection of $k$ is critical, as too small a value may not capture the system's behavior adequately, while too large a value can lead to computational inefficiency. The standard algorithm starts with a value of $k$ equal to zero, and increments such value until either a counterexample is encountered (BMC$(C, \phi, k)$ is satisfiable) or the property is proved (Inductive$(c, \phi, k)$ is unsatisfiable).

One notable strength of K-induction is its ability to prove properties that are beyond the reach of BMC, especially when BMC fails to find a counterexample within the bounded depth. However, K-induction may require additional invariants or lemmas to bridge the induction gap, which are properties that hold at every step and are used to strengthen the induction hypothesis.

Despite its power, K-induction's effectiveness is highly dependent on the underlying SMT or SAT solver's ability to handle the complexities introduced by the inductive step. Advanced techniques, such as incremental and property-directed K-induction [78], have been developed to enhance its applicability and efficiency in practical verification tasks. However, for infinite-state systems, differently from the finite-state case, K-induction is not a complete procedure, meaning that the bound $k$ may grow indefinitely.

### Interpolation-based model checking

Interpolation-based model checking [99, 98] and its variants emerges as a fusion of BMC and interpolation techniques, providing a complete technique for symbolic model checking in the finite-state case.

Given a transition system $C$ and a property $\phi$ to be verified, the method involves first checking that the unrolling BMC($C$, $\phi$, $k$) is unsatisfiable (again, initially $k$ is equal to zero). If a counterexample is not found, the resulting formula is divided into two parts, $A$ and $B$, such that $A \wedge B$ is unsatisfiable. Here, $A$ represents the initial state and the transitions up to a certain point, while $B$ represents the remainder of the transitions up to depth $k$ and the negation of the property $\phi$. Formally, this can be expressed as follows:

$$A = I(X_0) \wedge \bigwedge_{i=0}^{j-1} T(X_i, X_{i+1})$$
$$B = \left( \bigwedge_{i=j}^{k-1} T(X_i, X_{i+1}) \right) \wedge \neg\phi(X_k)$$

The unsatisfiability of $A \wedge B$ indicates that the property $\phi$ cannot be violated within $k$ steps. The interpolant between the two formulae effectively serves as an over-approximation of the states reachable within $j$ steps that cannot lead to a violation of $\phi$ within $k$ steps. By iteratively computing such interpolants (or interpolant sequences) for increasing values of $j$ (and thus $k$), the method and its variants construct a sequence of over-approximations that, in the case of finite-state systems, converge towards an inductive invariant sufficient to prove $\phi$. In the original method, $j$ was set to be 1, and a single interpolant was computed as a candidate inductive invariant. In variations of the method, the value of $j$ can vary, and interpolant sequences are used.

As the depth $k$ increases, if an interpolant is found that is also an inductive invariant, the property is verified. If, however, a counterexample is discovered at any depth $k$, the property is refuted. The iterative nature of this process, combined with the strategic use of interpolants, enables interpolation-based model checking to efficiently verify properties or identify counterexamples in finite-state systems. When adapted to infinite-state systems using SMT solvers, the principle remains the same, though the specifics of interpolant generation and the handling of theories become more complex, and completeness is lost.

### IC3

More recently, *IC3*[21], which stands for "Incremental Construction of Inductive Clauses" (referred to also as PDR, "Property Directed Reachability"), emerged as an efficient algorithm for verification of finite-state systems. The algorithm distinguishes itself by not computing unrollings associated with the transition relation. Instead, it opts for a methodical construction of state space over-approximations.

We introduce here some foundational concepts to simplify the exposition of one of the algorithms we will discuss in this thesis, which is based on IC3. The algorithm maintains a *trace*, i.e. an ordered sequence $F_0, \ldots, F_N$ of formulae, called frames, where the formula $F_i$ over-approximate the set of states reachable in up to $i$ transitions. More formally, the trace of IC3 is an *approximate reachability sequence*:

**Definition 4.** *Let $C = (X, I(X), T(X, X'))$ be a symbolic transition system and $\phi(X)$ a formula. A sequence of formulas $F_0(X), \ldots, F_N(X)$ is an approximate reachability sequence for $C$ and $\phi$ if:*

- $I(X) \models F_0(X)$;
- $F_i(X) \models F_{i+1}(X)$ *for all* $0 \leq i < N$;

- $F_i(X) \wedge T(X, X') \models F_{i+1}(X')$ *for all* $0 \leq i < N$;
- $F_i(X) \models \phi(X)$ *for all* $0 \leq i < N$.

An approximate reachability sequence can be used to prove that a formula $\phi$ is an invariant for a system, thanks to the following:

**Proposition 1** ( [21]). *Let* $C = (X, I(X), T(X, X'))$ *be a transition system and* $\phi$ *a formula. Let* $F_0, \ldots, F_N$ *be an approximate reachability sequence for* $C$ *and* $\phi$. *If for some* $0 \leq i < N, F_{i+1} \models F_i$, *then* $C \models \phi$. *Moreover,* $F_i$ *is an inductive invariant for* $\phi$ *and* $C$.

IC3 operates through a series of refinement steps, each aimed at either extending the trace with a new frame that preserves the safety property or refining an existing frame to exclude states that could lead to a safety violation. This is achieved through a process of *propagation* and *blocking*. During propagation IC3 attempts to propagate facts known about earlier frames to later frames in the trace, thereby extending the safe over-approximation forward through the state space. When a potential counterexample to the safety property is instead identified, IC3 blocks this counterexample by generating a *blocking clause*—a formula that excludes the counterexample and all similar unsafe states—and adding it to the appropriate frame in the trace.

The algorithm employs SAT solvers or SMT solvers for finite and infinite-state systems, respectively, to efficiently handle the logical operations involved in propagation and blocking. The SMT variants [33, 13, 31] of IC3 extend its applicability to a broader range of systems. One of the principal challenges in adapting IC3 for use with various theories lies in the representation of blocking clauses. As detailed in [33], devising a generalized procedure for this task could theoretically be accomplished through quantifier elimination. However, this approach proves to be computationally expensive and is not universally applicable across all theories, limiting its practical utility. Predicate abstraction presents a viable alternative, offering a means to efficiently integrate with IC3 by abstracting system states into a finite set of predicates that describe their properties. Despite its potential for efficiency, the effectiveness of predicate abstraction in conjunction with IC3 heavily depends on the judicious selection of predicates.

Moreover, the process of refining these approximations to exclude spurious counterexamples and strengthen the verification results necessitates efficient interpolation techniques, which do not always exist, as we have mentioned, for example for array theories.

### 2.2.2 Abstraction and Refinement

Abstraction and refinement are fundamental concepts in the field of formal verification [47]. These concepts are often essential in making the verification process more efficient and manageable, especially for complex systems.

Generally speaking, abstraction in verification refers to the process of simplifying a system by changing its state space or by omitting certain details that are not relevant to the verification goal. The main aim is to create a more manageable model that retains

the essential properties of the system, making it easier to analyze and verify. In SMT-based verification, abstraction might involve creating a higher-level representation of the system where certain operations or data are simplified. For example, a variable might be abstracted to represent a range of values rather than a specific value, or complex data structures might be represented in a simplified manner.

The use of abstraction helps in tackling the state explosion problem, which is a significant challenge also in the verification of parameterized systems. By working with a simplified model, the computational resources and time required for verification can be substantially reduced. However, the abstraction process must be carefully designed to ensure that the abstract model is sound, meaning that if the abstract model satisfies the property being verified, then the original, concrete model also satisfies that property.

However, a verification algorithm may find some counterexample in the abstract system, that does not correspond to a concrete counterexample. This happens if the abstraction is too coarse, and thus it needs to be refined.

Refinement is the complementary process to abstraction. It involves incrementally adding detail or specificity back into the abstract model based on the results of the verification attempt. This is typically achieved through counterexample-guided abstraction refinement [42](CEGAR), a popular technique in formal verification. In CEGAR, a counterexample produced by the verification attempt on the abstract model is analyzed to determine if it is also a counterexample for the concrete model. If not, the abstraction is refined to eliminate the spurious counterexample, and the verification process is repeated.

More formally, an abstraction between two systems is represented by a *simulation*, which is a specific relation among the states of the systems. Let $C = (X, I(X), T(X, X'))$ and $\tilde{C} = (\tilde{X}, \tilde{I}(\tilde{X}), \tilde{T}(\tilde{X}, \tilde{X}'))$ be two symbolic transition systems. Let $S$ be the set of states of $C$, and $\tilde{S}$ the set of states of $\tilde{C}$. Let $\alpha$ be a relation between $S$ and $\tilde{S}$; we write $\alpha(s, \tilde{s})$ to denote that two states $s$ and $\tilde{s}$ are in relation.

**Definition 5.** *We say that $\tilde{C}$ $\alpha$-simulates $C$ (written $C \to_\alpha \tilde{C}$) if the following two conditions hold:*

*i. For each initial state $s$ of $C$, there exists an initial state $\tilde{s}$ of $\tilde{C}$ such that $\alpha(s, \tilde{s})$.*

*ii. For each couple $(s, \tilde{s})$ such that $\alpha(s, \tilde{s})$, and for each $s' \in S$ such that $s, s' \models T(X, X')$, there exists a state $\tilde{s}'$ such that $\alpha(s', \tilde{s}')$ and $\tilde{s}, \tilde{s}' \models \tilde{T}(\tilde{X}, \tilde{X}')$.*

If $\alpha$ is clear in the context, we might say that $\tilde{C}$ simulates (or *abstracts*) $C$. We refer to $C$ as the concrete system, while $\tilde{C}$ is the abstract system. The following proposition captures the fact that paths in the concrete system can be mirrored in the abstract.

**Proposition 2.** *Let $C \to_\alpha \tilde{C}$ be an $\alpha$-simulation between two transition systems. Then, given any run $\pi = s_0, \ldots, s_n$ of $C$, there exists a run $\tilde{\pi} = \tilde{s}_0, \ldots, \tilde{s}_n$ of $\tilde{C}$ such that, for all $0 \leq i \leq n$, $\alpha(s_i, \tilde{s}_i)$.*

*Proof.* Follows immediately by induction on $n$ and Definition 5. □

A weaker notion of simulation that still preserves paths (even if with different lengths) is the following:

**Definition 6** (Stuttering Simulation)**.** *We say that $\tilde{C}$ stutter $\alpha$-simulates $C$ (written $C \to_\alpha \tilde{C}$) if the following two conditions hold:*

*i. For each initial state $s$ of $C$, there exists an initial state $\tilde{s}$ of $\tilde{C}$ such that $\alpha(s, \tilde{s})$.*

*ii. For each couple $(s, \tilde{s})$ such that $\alpha(s, \tilde{s})$, and for each $s' \in S$ such that $s, s' \models T(X, X')$, either there exists a state $\tilde{s}'$ such that $\alpha(s', \tilde{s}')$ and $\tilde{s}, \tilde{s}' \models \tilde{T}(\tilde{X}, \tilde{X}')$, or there exists two states $\tilde{s}', \tilde{s}''$ such that $\alpha(s', \tilde{s}'')$ and $\tilde{s}, \tilde{s}' \models \tilde{T}(\tilde{X}, \tilde{X}')$. $\tilde{s}', \tilde{s}'' \models \tilde{T}(\tilde{X}, \tilde{X}')$ .*

Similarly to the non-stuttering case, we have:

**Proposition 3.** *Let $C \to_\alpha \tilde{C}$ be a stutter $\alpha$-simulation between two transition systems. Then, given any run $\pi = s_0, \ldots, s_n$ of $C$, there exists a run $\tilde{\pi} = \tilde{s}_0, \ldots, \tilde{s}_m$ of $\tilde{C}$ such that $\alpha(s_0, \tilde{s}_0)$ and, , for all $1 \leq i \leq n$, there exists a $k_i < m$ such that $\alpha(s_i, \tilde{s}_{k_i})$.*

Usually, simulation relations can be defined inside the theory: we say that a relation $\alpha$ is $\mathcal{T}$-definable if there exists a $\Sigma(X, \tilde{X})$-formula $H(X, \tilde{X})$ such that $\alpha(s, \tilde{s}) \Leftrightarrow s, \tilde{s} \models H(X, \tilde{X})$. In such cases, it is always possible to find a transition system that simulates $C$, called the existential abstraction. This abstraction is given by a transition system defined with the initial formula $\tilde{I}(\tilde{X}) = \exists X. I(X) \wedge H(X, \tilde{X})$ and the transition formula $\tilde{T}(\tilde{X}, \tilde{X}') = \exists X, X'. T(X, X') \wedge H(X, \tilde{X}) \wedge H(X', \tilde{X}')$. The existential abstraction is the 'most precise' abstraction possible (*with respect to* $\alpha$), but it is in general hard to compute.

**Example 1.** One of the most used abstraction paradigms in model checking of infinite-state systems is Predicate Abstraction. This technique is usually used to reduce to the Boolean case: given a symbolic transition system $C$ and a set of $\Sigma(X)$-predicates $\mathcal{P}$, the predicate abstraction of $C$ is a new system $\tilde{C}$, defined only over boolean variables $\tilde{X}$ (one for each element of $\mathcal{P}$). Formally, the simulation relation is given by

$$\alpha(s, \tilde{s}) \leftrightarrow s, \tilde{s} \models \Big( \bigwedge_{p(X) \in \mathcal{P}} \tilde{x}_{p(X)} \leftrightarrow p(X) \Big),$$

where $\tilde{x}_p$ is the boolean variable in $\tilde{X}$ corresponding to the predicate $p$. Note that the simulation relation is $\mathcal{T}$-definable, thus one way to compute $\tilde{C}$ is the existential abstraction. In general, the explicit computation of the system $\tilde{C}$ can be very expensive, requiring typically ALLSAT techniques [92]. However, more efficient implicit encoding were proposed in the literature [127, 90]. □

In many cases, the abstract variables $\tilde{X}$ of the system $\tilde{C}$ will be completely different from the original variables $X$, as seen in the example above. However, in some cases the abstract variables can be a subset of the original one. We start by analyzing this simpler case. Let $V \subseteq X \cap \tilde{X}$ be a set of variables, and $F(V)$ a formula. Let $\alpha$ be a relation between states.

**Definition 7.** *We say that the relation $\alpha$ preserves the formula $F$ if, for all states $s$ such that $s \models \neg F$, then, for all $\tilde{s}$ such that $\alpha(s, \tilde{s})$, we have that $\tilde{s} \models \neg F$.*

**Proposition 4.** *Given a simulation $C \rightarrow_\alpha \tilde{C}$ that preserves $F$,*

$$\tilde{C} \models F \Rightarrow C \models F.$$

*Proof.* Suppose by contradiction that $\tilde{C} \models F$ but there exists a finite path $\pi$ in $C$ such that $\pi[n] \models \neg F$. By Proposition 2, there exists a finite path $\tilde{\pi}$ in $\tilde{C}$ such that $\alpha(\pi[n], \tilde{\pi}[n])$. Since $\alpha$ preserves $F$, we have that $\tilde{\pi}[n]$ is reachable in $C$ and $\tilde{\pi}[n] \models \neg F$, a contradiction. □

Similarly, we can prove the following:

**Corollary 5.** *Given a stutter simulation $C \rightarrow_\alpha \tilde{C}$ that preserves $F$,*

$$\tilde{C} \models F \Rightarrow C \models F.$$

We also have:

**Proposition 6.** *Given a (sutter) simulation $C \rightarrow_\alpha \tilde{C}$ that preserves $F$, if $F$ is inductive for $\tilde{C}$, then $F$ is inductive for $C$.*

Let's now consider the more general case where $\tilde{X}$ and $X$ might be unrelated. Usually, in these cases, one defines some rewriting operator $R$ from formulas defined over $X$ to formulas defined over $\tilde{X}$. Therefore, we need a more detailed definition of preservation:

**Definition 8.** *We say that the relation $\alpha$ preserves the formula $F$ modulo the rewriting $R$ if, for all states $s$ such that $s \models \neg F$, then, for all $\tilde{s}$ such that $\alpha(s, \tilde{s})$, we have that $\tilde{s} \models \neg R(F)$.*

Thus a similar result yields:

**Proposition 7.** *Given a (stutter) simulation $C \rightarrow_\alpha \tilde{C}$ that preserves $F$ modulo the rewriting $R$,*

$$\tilde{C} \models R(F) \Rightarrow C \models F.$$

**Example 2.** In predicate abstraction, given a set of predicates $\mathcal{P}$ and corresponding abstract boolean variables $\tilde{X}$, let $R$ be a rewriting operator that computes the substitution between abstract variables in $\tilde{X}$ and corresponding atoms in $\mathcal{P}$. Then, the simulation preserves each formula whose atoms are contained in $\mathcal{P}$.

If a counterexample is found in $\tilde{C}$, in general this does not implies the existence of a counterexample in $C$. We say that a counterexample $\pi$ in $\tilde{C}$ is spurious if there exists no path $s_0, \ldots, s_k$ in $C$ such that $s_n \models \neg F$ and, for all $0 \leq i \leq k$, $\alpha(s_i, \pi[i])$. In such cases, the abstraction yields no helpful information, and need to be *refined*. Given a simulation $C \rightarrow_\alpha \tilde{C}$, and a spurious counterexample $\pi$, we say that the simulation $C \rightarrow_{\alpha'} \tilde{C}'$ *refines* $C \rightarrow_\alpha \tilde{C}$ iff: $(i)$ each reachable state of $\tilde{C}'$ is a reachable state of $\tilde{C}$; $(ii)$ $\pi$ is not a valid path for $\tilde{C}'$. Refinements are a way to ensure that CEGAR methods make progress, by eliminating newer and newer counterexamples. For finite-state systems, this is enough for ensuring termination of the procedure (in the infinite-state case, termination is not ensured). As an example of refinement, in predicate abstraction, interpolants are usually used to discover new predicates and eliminate counterexamples [33, 13, 100].

### 2.2.3 Asynchronous Composition of Systems

The notion of composing systems to function concurrently yet independently represents a important part of software architectures. This section delves into the concept of asynchronous composition of systems, in the setting of Verification Modulo Theories.

In particular, we will investigate the relations between asynchronous composition and abstractions.

Let $C_1 = (X_1, I_1(X_1), T(X_1, X_1'))$ and $C_2 = (X_2, I(X_2), T(X_2, X_2'))$ be two symbolic transition systems. If $V$ is a set of variables, we denote with $Inertia(V)$ the formula $\bigwedge_{v \in V} v = v'$.

**Definition 9.** *The asynchronous product between $C_1$ and $C_2$, is the transition system $C_1 \parallel C_2 = \big(X_1 \cup X_2, I_{C_1 \parallel C_2}(X_1, X_2), T_{C_1 \parallel C_2}(X_1, X_2, X_1', X_2')\big)$, where:*

- *$I_{C_1 \parallel C_2}(X_1, X_2)$ is the formula $I_1(X_1) \wedge I_2(X_2)$;*
- *$T_{C_1 \parallel C_2}$ is the formula $(T_1(X_1, X_1') \wedge Inertia(X_2 \setminus X_1)) \vee (T_2(X_2, X_2') \wedge Inertia(X_1 \setminus X_2))$.*

The asynchronous product is generally considered with components that have some shared variables.

**Example 3.** As sketched example of asynchronous composition, consider a web application composed of two main components: a User Interface (UI) Manager, $C_1$, and a Database Update Manager, $C_2$. These components interact with each other in an asynchronous manner to handle user requests and update the database, respectively. In the system $C_1 = (X_1, I_1(X_1), T(X_1, X_1'))$, $X_1$ represents the state variables associated with user interactions (e.g., input fields, request status), $I_1(X_1)$ specifies the initial conditions of the UI (e.g., all fields are clear, no pending requests), and $T(X_1, X_1')$ defines the transition relation that captures the changes in the UI state based on user actions (e.g., submitting a form). Similarly, $C_2 = (X_2, I_2(X_2), T(X_2, X_2'))$ represents the Database Update Manager, with $X_2$ including variables related to the database state (e.g., pending updates, connection status), $I_2(X_2)$ indicating the initial state of the database connection, and $T(X_2, X_2')$ describing how database updates are processed and applied. The intersection among $X_1$ and $X_2$ consists of variables representing tasks common to both the UI and the database, such as the user session information. The asynchronous product of $C_1$ and $C_2$, denoted as $C_1 \parallel C_2$, models the web application as a whole, allowing the UI Manager and the Database Update Manager to operate concurrently. □

Since $Inertia(V \cup V') = Inertia(V) \wedge Inertia(V')$, and by distributing conjunction over disjunction, we observe that the composition $(C_1 \parallel C_2) \parallel C_3$ is defined with equivalent formulae to $C_1 \parallel (C_2 \parallel C_3)$. Given a set of variables $V \subseteq X_1 \cup X_2$ and a formula $F(V)$, asynchronous verification aims to prove or disprove if $(C_1 \parallel C_2) \models F(V)$.

A method to address this verification problem is to search for an inductive invariant for the system $(C_1 \parallel C_2)$. This approach, known as the *monolithic approach*, involves finding a single, global inductive invariant that is strong enough to prove the property

$F(V)$ for the entire system composed of $C_1$ and $C_2$. The main challenge of this approach is the complexity of discovering a suitable inductive invariant that encompasses the behaviors of all components in the composition.

Another approach leverages the compositional nature of the system, exploiting the fact that each component might already have its invariants. This strategy, known as the *compositional approach*, seeks to combine invariants from individual components into a single global invariant for the entire system. The advantage of this approach is that it can potentially reduce the complexity of finding a suitable invariant by building upon the properties already proven for individual components. However, ensuring that combined invariants provide sufficient coverage for the composite system's behavior, especially regarding interactions through shared variables, can be challenging.

Both approaches aim to establish a form of inductive reasoning that guarantees the system's behavior aligns with the specified properties $F(V)$. The choice between a monolithic or compositional approach often depends on the specific characteristics of the system being verified, including the complexity of its components, the nature of their interactions, and the properties to be verified.

We now state a proposition that will be useful in our compositional approach. In general, if $V \not\subseteq X_i$, we may write $C_i \models F(V)$ with the meaning that we add to the $X_i$ the remaining $V \setminus X_i$ variables, and we modify $T_i$ by adding inertia on $V \setminus X_i$. This technicality is necessary to state propositions like the following:

**Proposition 8.** *If a formula $F$ is not inductive for $C_1 \parallel C_2$, then there exists an $i \in \{1, 2\}$ such that $F$ is not inductive for $C_i$.*

*Proof.* If $\Psi$ is not inductive for $C_1 \parallel C_2$, then either $I_{C_1 \| C_2} \wedge \neg F$ is satisfiable, or $F \wedge T_{C_1 \| C_2} \wedge \neg F'$ is satisfiable. If $I_{C_1 \| C_2} \wedge F = I_1 \wedge I_2 \wedge \neg F$, is satisfiable, then by restricting the model over $X_1$ or $X_2$ we get the $F$ is not inductive for both the systems. Otherwise, suppose we have a model for

$$F \wedge \big((T_1(X_1, X_1') \wedge Inertia(X_2 \setminus X_1)) \vee (T_2(X_2, X_2') \wedge Inertia(X_1 \setminus X_2))\big) \wedge \neg F'$$

which, distributing the conjunction, is logically equivalent to

$$(F \wedge (T_1(X_1, X_1') \wedge Inertia(X_2 \setminus X_1) \wedge \neg F') \vee (F \wedge (T_2(X_2, X_2') \wedge Inertia(X_1 \setminus X_2) \wedge \neg F').$$

Therefore, it must exists an $i$ such that $(F \wedge (T_i(X_i, X_i') \wedge Inertia(X_{3-i} \setminus X_i) \wedge \neg F')$ is satisfiable, meaning that $F$ is not inductive for $C_i$. □

If we have a family of simulation relation, and we consider the asynchronous composition of the 'concrete' transition systems, then it is possible (under some hypothesis on the abstract initial formula) to build a simulation relation with the asynchronous composition of the abstract systems. We say that two transition system $C_1$ and $C_2$ are *compatible* if each assignment to the shared variables is part of an initial state of $C_1$ if and only if it is an initial state of $C_2$. In practice, this means that the shared variables are initialized in the same way in the two systems.

**Proposition 9.** *Consider $C_1 \to_{\alpha_1} \tilde{C}_1$ and $C_2 \to_{\alpha_2} \tilde{C}_2$ two simulation relation. Suppose that $\tilde{C}_1$ and $\tilde{C}_2$ are compatible. Consider $\alpha_1 \parallel \alpha_2$ (called the product simulation) defined as*

$$\alpha_1 \parallel \alpha_2(s, \tilde{s}) \text{ iff } \alpha_1(s|_{X_1}, \tilde{s}|_{X_1}) \text{ and } \alpha_2(s|_{X_2}, \tilde{s}|_{X_2}).$$

*Then, $C_1 \parallel C_2 \to_{\alpha_1 \parallel \alpha_2} \tilde{C}_1 \parallel \tilde{C}_2$.*

*Proof.* Let $s$ be an initial state of $C_1 \parallel C_2$, i.e. $s \models I_{C_1 \parallel C_2}$. Therefore, there exist states $\tilde{s}_1 \models \tilde{I}_1$ and $\tilde{s}_2 \models \tilde{I}_2$ such that $\alpha_1(\tilde{s}_1, s|_{X_1})$ and $\alpha_2(\tilde{s}_2, s|_{X_2})$. Since $\tilde{C}_1$ and $\tilde{C}_2$ are compatible, then we have that the assignment $(\tilde{s}_1, \tilde{s}_2) \models \tilde{I}_1 \wedge \tilde{I}_2$, and the first condition of Definition 5 is satisfied.
Suppose now that there is a couple of state $s, \tilde{s}$ such that $\alpha(s, \tilde{s})$, and suppose that $s, s' \models T_{C_1 \parallel C_2}$. Therefore, there exists an $i \in \{1, 2\}$ such that $s|_{X_i}, s'|_{X_i} \models T_i(X_i, X_i')$ and $s|_{X_i}, s'|_{X_i} \models \mathit{Inertia}(X_{3-i} \setminus X_i)$. Thus, it exists a state $\tilde{s}_i'$ such that $\alpha_i(s'|_{X_i}, \tilde{s}_i')$ and $\tilde{s}|_{X_i}, \tilde{s}_i' \models \tilde{T}_i$. Let $\tilde{s}'$ defined as $\tilde{s}_i'$ on $X_i$ and as $\tilde{s}$ on $X_{3-i} \setminus X_i$. We have that $\tilde{s}, \tilde{s}' \models \tilde{T}_i \wedge \mathit{Inertia}(\tilde{X}_{3-i} \setminus \tilde{X}_i)$, and $\alpha_1(\tilde{s}_1', s'|_{X_1})$ and $\alpha_2(\tilde{s}_2', s'|_{X_2})$. □

By Definition of product simulation, we have the following corollary:

**Corollary 10.** *Consider $C_1 \to_{\alpha_1} \tilde{C}_1$ and $C_2 \to_{\alpha_2} \tilde{C}_2$ two simulation relation such that they both preserve a formula $F$. Assume that $\tilde{C}_1$ and $\tilde{C}_2$ are compatible. Then, $\alpha_1 \parallel \alpha_2$ also preserves $F$.*

## 2.3 Problem statement

The main problem discussed in this thesis is the problem of invariant checking over a class of symbolic transition systems. Such a class is a generalization of the formalism used in [67, 112], especially suited for the verification of parameterized systems. We call this class *array-based transition systems*, following [67], even if our formalism is more general, allowing general transition formulae. Note that this is also different from standard symbolic transition systems over the extensional theory of arrays.

Parameterized systems are systems composed of an arbitrary number of components or processes, which can be identical or exhibit variations based on certain parameters. These systems are particularly prevalent in distributed computing, where numerous nodes or agents operate concurrently, potentially with varying roles or states, but are designed to achieve a collective goal or maintain a global property. The verification of parameterized systems thus revolves around ensuring that for any number of components, the system adheres to its specified correctness properties.

The challenge in verifying parameterized systems arises from their inherent infinity: because the system can comprise an arbitrary number of components, traditional finite-state verification techniques are not directly applicable. This necessitates the development of verification techniques that can reason about an infinite state space in a efficient manner. Array-based transition systems provide a framework for modeling such systems by abstracting the state of an arbitrary number of components into functions, where each value of the function represents the state of one component. The transitions of the system, then, are described by transition formulae that specify how such functions (and thus the states of the components) can change from one state to the next.

This formalism is more general than standard symbolic transition systems over the theory of arrays in several ways. In particular, the transition formulae can encompass a broader range of operations on the arrays, including non-deterministic updates and operations that affect an arbitrary number of components simultaneously. This flexibility makes the array-based transition system formalism especially suited for modeling and verifying parameterized systems, as it can more naturally express the behaviors and interactions of an arbitrary number of components.

### 2.3.1 Array-based transition systems

We start by considering a theory $\mathcal{T}_{\mathcal{I}} = (\Sigma_I, \mathcal{C}_I)$, called the *index* theory, whose class of models does not contain infinite universes. In practice, this is often the theory of an uninterpreted sort with equality, whose class of models includes all possible finite (but unbounded) structures. In addition, we consider a quantifier-free theory of *elements* $\mathcal{T}_E = (\Sigma_E, \mathcal{C}_E)$, used to model the data of the system. Relevant examples consider as $\mathcal{T}_E$ the theory of an enumerated datatype, linear arithmetic (integer or real), or a combination of those. In addition, we consider a third theory (that we refer as the *array* theory), whose signature only contains the function symbol $\{\cdot[\cdot]\}$, which is interpreted as the function application. Finally, with $A_I^E$ we denote the combination

of the index, the element, and the function theory. Therefore, the signature of $A_I^E$ is $\Sigma = \Sigma_I \cup \Sigma_E \cup \{\cdot[\cdot]\}$, and a model for it is given by a set of total functions from the universe of a model of the index sort, to a universe of the element sort, and $\cdot[\cdot]$ is interpreted as the function application. Therefore, $A_I^E$ satisfiability can be reduced to satisfiability in the combination of $\mathcal{T}_{EUF}$, $\mathcal{T}_I$ and $\mathcal{T}_E$. In the following, we will denote with letters $I, J$ variables of index sort, while we use the letter $X$ to denote array symbols. We restrict ourselves to one index theory and one element theory for the sake of simplicity, but typically applications include a combination of several index theories and several element theories.

**Definition 10.** *An array-based transition systems is a symbolic transition system*

$$S = (X, I(X), T(X, X'))$$

*where:*

*(i.) $X$ is a set of symbols of array sorts;*

*(ii.) $I(X), T(X, X')$ are $\Sigma(X, X')$-formulae (possibly containing quantified variables only of sort $\mathcal{T}_I$).*

Note that arrays can be also 0-ary or constant functions, thus allowing to have state variables of index or element sort.

**Example 4.** We show how to model a simplified version of the Bakery algorithm in our formalism. The Bakery Algorithm is a ticket-based algorithm designed for ensuring mutual exclusion in distributed environments, where multiple processes need to access a shared resource without conflicts. It was named after the numbered ticket system used in bakeries to serve customers in order. The algorithm assigns a unique number to each process that wants to enter the critical section. This number determines the order in which processes are granted access. A process receives a number higher than any other process currently waiting, ensuring that each process enters the critical section in the correct order.

To model the algorithm, we use as $\mathcal{T}_I$ the theory of equality over a simple sort. Each element of a universe of $\mathcal{T}_I$ can be thought as a process entering the algorithm. As for $T_E$, we need a combination of two theories: an enumerated datatype with values $\{idle, wait, crit\}$, and $\mathcal{QF}\_\mathcal{LIA}$ (quantifier-free linear integer arithmetic). We define four state variables: one array *state* with values in $\{idle, wait, crit\}$, one array *ticket* with values in $\mathbb{Z}$, and two integer variables *next_ticket* and *to_serve*. The initial formula of the system is:

$$\forall i. state[i] = idle \wedge ticket[i] = 0 \wedge next\_ticket = 1 \wedge to\_serve = 1$$

The transition formula is the disjunction of the three formulae; the first one models a

process that goes from idle to wait and takes a ticket:

$$\exists i.\big(state[i] = idle \wedge state'[i] = wait \wedge ticket'[i] = next\_ticket \wedge \\ next\_ticket' = next\_ticket + 1 \wedge to\_serve' = to\_serve \\ \wedge \forall j.(j \neq i \rightarrow state'[j] = state[j] \wedge ticket'[j] = ticket[j])\big).$$

In the second disjunct, a process enters the critical section if its ticket is selected:

$$\exists i.\big(state[i] = wait \wedge state'[i] = crit \wedge ticket[i] = to\_serve \\ \wedge next\_ticket' = next\_ticket \wedge to\_serve' = to\_serve \wedge \\ \forall j.(j \neq i \rightarrow state'[i] = state[i]) \wedge (\forall j.ticket'[j] = ticket[j])\big).$$

Finally, a process exits from the critical state and reset its ticket:

$$\exists i.\big(state[i] = crit \wedge state'[i] = idle \wedge ticket'[i] = 0 \wedge \\ next\_ticket' = next\_ticket \wedge to\_serve' = to\_serve + 1 \wedge \\ \forall j.(j \neq i \rightarrow state'[j] = state[j] \wedge ticket'[j] = ticket[j])\big).$$

□

Given a model $\mathcal{M}$ of $A_I^E$, a state of an array-based transition system is a valuation of the elements in $X$ in $\mathcal{M}$, i.e. an assignment of the state variables to functions from a finite universe of the index sort, to an universe of an element sort. In modeling parameterized systems, the number of components is given by the cardinality of the universe of the index sort.

The presence of quantifiers in the theory $\mathcal{T}_I$ make the general problem of satisfiability of $A_I^E$ formulae to be undecidable (even in the case where the element sort is boolean). However, a fragment of first-order logic used to describe parameterized problems is EPR [107, 108, 111] (Effective Propositional Logic), which is decidable. Effective Propositional Logic (EPR), also known as the Bernays-Schönfinkel class, is a fragment of classical first-order logic characterized by formulae that start with a sequence of existential quantifiers followed by a universally quantified formula, that does not contains any function symbols of ariety greater than zero (i.e., it contains only constants).

The decidability of EPR stems from the fact that, after Skolemization, the satisfiabilty of the formula can be reduced to the satisfiabilty of a finite set of ground instances of the formula. Moreover, formulae in EPR enjoys the finite model property, which asserts that formulae have finite models whose universe is a finite set.

In our SMT setting, we have the following decidability result, which combines classical EPR with ground SMT theories:

**Proposition 11.** *[67] If $\Sigma_I$ does not contain any function symbol with ariety greater than zero, and the quantifier-free fragment of the theories $\mathcal{T}_I$ and $\mathcal{T}_E$ are decidable, then the $A_I^E$-satisfiability of formulae of the form*

$$\exists I.\forall J.\phi(I, J, X) \tag{2.3}$$

*is decidable. Moreover, a formula of type 2.3 is satisfiable if and only if it is satisfiable in a finite index model.*

To decide the satisfiability of formulae of kind 2.3, one can follow a procedure similar to the EPR case. In fact, the fact that $\Sigma_I$ is relational implies that there are only finitely many terms (even after Skolemization) to instantiate the universal quantifiers with. Therefore, one can reduce the problem to the satisfiability of a finite set of ground formulae of the combination of $\mathcal{T}_I$, $\mathcal{T}_E$ and $\mathcal{T}_{EUF}$.

Although in our context we do not require that an array-based transition system has initial and transition formulae that falls into the decidable fragment 2.3, many of the systems considered actually falls into this category.

#### The invariant problem for array-based transition systems

We can now formally introduce the main problem that will be discussed in this thesis.

Let $S$ be an array-based transition system, and $\phi(X)$ a $\Sigma(X)$-formula, possibly containing quantified variables of sort $\mathcal{T}_I$. The *Invariant Problem* we consider is the problem of deciding whether $S \models \phi(X)$. The problem is in general undecidable since it subsumes undecidable problems such as safety of parameterized systems or infinite-state systems [67, 14, 110].

**Example 5.** Following the last example, the property we want to prove is mutual exclusion:

$$\forall i, j.(i \neq j \rightarrow \neg(state[i] = crit \wedge state[j] = crit)).$$

The latter example of mutual represents a typical aspect of safety within parameterized systems. Safety properties, such as mutual exclusion, assert that "bad things" never happen during the execution of the system, regardless of the number of components or the specific states those components may be in. In the context of array-based transition systems, proving such a property involves demonstrating that it is impossible for two distinct components to simultaneously be in a critical state. Note that the property is not inductive itself. The undecidability of the invariant problem underscores the challenges and significance of the research in advancing the methods and tools available for tackling these verification problems in parameterized and infinite-state systems.

### 2.3.2 Ground instances

One of the key challenges in parameterized verification is managing the complexity that arises from the system's potential to grow indefinitely in terms of its components. To facilitate the verification of such systems, the concept of ground instances can be used as a way to facilitate things. This concept essentially involves considering a specific instantiation of the parameterized system, where the number of components is fixed to a finite number $n$.

More formally, suppose that we fix a finite cardinality $n$ for the $\mathcal{T}_I$-models. Then, one can construct a quantifier-free under-approximation of $S$ by considering another symbolic transition system, called ground $n$-instance, by considering as states functions

with, as domains, sets of only that size. In this section, we define how to construct such a system.

First, we consider $C = \{c_1, \ldots, c_n\}$ a set of fresh constants of index sort. These will be frozen variables of the ground instance; moreover, they will be also considered all implicitly different. In the following, if $\phi(X)$ is a formula with quantifiers of only sort $\mathcal{T}_I$, we denote $\phi_n(C, X)$ the quantifier-free formula obtained by grounding the quantifiers in $C$, i.e. by recursively applying the rewriting rules:

$$\forall i.\phi'(i, X) \mapsto \bigwedge_{k=1}^{n} \phi'(i, X)[i/c_k] \tag{2.4}$$

$$\exists i.\phi'(i, X) \mapsto \bigvee_{k=1}^{n} \phi'(i, X)[i/c_k] \tag{2.5}$$

Finally, we also need to restrict function symbols with values in $\mathcal{T}_I$. To do so, for each function symbol $a$ in $\Sigma$ whose codomain type is $\mathcal{T}_I$, we define the formula

$$\alpha(a) := \forall i_1, \ldots, i_m \exists j.a(i_1, \ldots, i_m) = j,$$

where $m$ is the ariety of $a$, and $i_1, \ldots, i_m, j$ are fresh variables of index sort.

**Definition 11** (Ground $n-$instance)**.** *Let $S = (X, I(X), T(X, X'))$ be an array-based transition system, and $n$ an integer. Let $\bar{I}(X) := I(X) \wedge \bigwedge_a \alpha(a)_n$, and $\bar{T}(X, X') := T(X, X') \wedge \bigwedge_a \alpha(a)_n$, where $a$ ranges over each function symbol in $\Sigma$ whose codomain type is $\mathcal{T}_I$. The ground $n$-instance of the system $S$ is a symbolic transition system $S_n$ defined by:*

$$S_n = \big(C \cup X, \bar{I}_n(C, X), \bar{T}_n(C, X, X') \wedge C' = C\big).$$

To simplify the use of notation, we will denote with $I_n$ and $T_n$ the initial and transition formula of the $n$-instance $S_n$. Observe that a state of $S_n$ is given by: $(i)$ a valuation of the symbols $C$ in a finite index universe, and $(ii)$ an interpretation of the state variables X as functions from that universe to an element sort. Note that even if the models of $T_I$ are all finite, the set of states of $S_n$ can be infinite, since $\mathcal{T}_E$ could have an infinite model, e.g. if integer or real variables are in the system. Nevertheless, the system can be model-checked efficiently by modern symbolic SMT techniques like [33].

We finally state the following result, linking the invariant problem for array-based transition systems to the invariant problem for ground instances:

**Proposition 12.** *Let $S$ be an array-based transition system, and $\phi$ a formula. Then, $S \models \phi$ if and only if $S_n \models \phi_n$ for all possible $n$.*

*Proof.* Recall that $S \models \phi$ does not hold if there exists a model $\mathcal{M}$ of $A_I^E$and a sequence of state $s$ such that $s$ is reachable in $S$ and $s \models \phi$. Suppose that the restriction of $\mathcal{M}$ over the index sort has a universe with cardinality $n$ (recall that we assumed that the index sort has only finite models). Thus, $s$ is also a reachable state of $S_n$ such that $s \models \phi_n$. □

### Symmetries of ground instances

As already observed in previous works [88, 70, 34], transition systems obtained by instantiating quantified formulae have a certain degree of symmetry. We report here the notion that will be useful to our description. Let $\sigma$ be a permutation of $1, ..., n$ (also called $n$-permutation), and $\phi$ a formula in which $c_1, \ldots, c_n$ occur free. We denote with $\sigma\phi$ the formula obtained by substituting every $c_i$ with $c_{\sigma(i)}$. The following proposition follows directly from the fact that $I_n$ and $T_n$ are obtained by instantiating a quantified formula with a set of fresh constants $C$[88]:

**Lemma 13.** *For every permutation $\sigma$, we have that: $(i)$ the formula $\sigma I_n$ is logically equivalent to $I_n$; $(ii)$ the formula $\sigma T_n$ is logically equivalent to $T_n$.*

From this lemma and an induction proof, the following holds:

**Proposition 14** (Invariance for permutation)**.** *Let $s$ be a state of $S_n$, reachable in $k$ steps. Then $s \models \phi(C, X)$, if and only if, for every $n$-permutation $\sigma$, there exists a state $s'$ reachable in $k$ steps such that $s' \models \sigma\phi(C, X)$*

We will exploit this property both for the verification of ground instances and in the generalization process. In fact, from the last proposition, we can simplify every invariant problem $S_n \models \bigwedge_\sigma \sigma\phi(C, X)$ – where $\sigma$ ranges over all possible substitutions – to $S_n \models \phi(C, X)$. This simplification is of great help when checking properties which are the result of instantiating a formula with only universal quantifiers.

# Part II

# Algorithms

# Chapter 3

# Algorithms for the Invariant Problem of Parameterized Systems

## 3.1 Introduction

In this section, we describe algorithms designed to solve the problem stated in Section 2.3. This chapter is based on the works [35, 34, 115].

In Section 2.2.1, we have delineated the principal algorithms deployed for addressing symbolic model checking challenges. Predominantly, these algorithms have been tailored for finite-state systems, characterized through Boolean formulas and analyzed using SAT solvers. A query emerges regarding the feasibility of substituting SAT solvers with SMT solvers equipped to handle quantifiers. While theoretically conceivable for methodologies like bounded model checking (BMC) and k-induction, practical implementation encounters significant hurdles.

Although BMC keeps its utility for finding counterexamples, the likelihood of k-induction converging to an inductive invariant diminishes. In fact, for infinite-state systems, the method is no longer complete. Furthermore, as the bound $k$ escalates, the complexity of the formulas increases proportionally due to an augmented count of quantifiers, imposing a substantial computational burden on solvers. Consequently, a direct transposition of BMC and k-induction methodologies to contexts requiring quantifier handling proves impracticable.

Interpolation-based techniques encounter analogous obstacles. Besides the challenges shared with BMC, the task of generating interpolants for formulas with quantifiers is scarcely supported by current SMT solvers, as previously noted. Conversely, IC3 presents a more viable pathway since it does not rely on explicit unrolling and does not inherently lead to an increase in quantifier complexity within satisfiability checks. To formulate representations of counterexamples, two primary strategies emerge. One involves quantifier elimination, a solution not universally applicable across theories, particularly where function variables occur. A solution to this problem is imposing stringent syntactic restrictions on the system-defining formulas to facilitate an effective quantifier elimination approach—a tactic akin to the one employed in [65] through functional assignments in transition formulas. However, this thesis aspires to devise

algorithms devoid of such restrictive prerequisites, aiming for a more general applicability. A second solution is to use forms of abstraction to produce approximations of counterexamples. In this direction, the UPDR [81] algorithm, extending IC3 to systems with quantifiers, was recently proposed, albeit limited to pure first-order logic absent of any theories. In the subsequent section, we introduce the first main contribution of this thesis: elaborating on how to adapt and apply this algorithm within our formalism, thereby extending its utility to a broader spectrum of systems.

Thus, the first algorithm we described is based on UPDR. Such procedure combines UPDR with implicit abstraction [127], to deal with a general SMT theories of elements. In the past, implicit abstraction was used in [33] to extend IC3 to deal with quantifier-free infinite-state systems. In the section, we show how to modify that technique to handle quantified systems as well. We also lift several properties of the original algorithm to our setting. However, the resulting algorithm heavily relies on quantified queries to an external solver: in practice, this is still an expensive subroutine.

The second algorithm consists of a simple yet general procedure based on the interaction of two key ingredients. First, we restrict the cardinality of the uninterpreted sort to a fixed natural number. This results in a quantifier-free system that can be model-checked with existing techniques: upon termination, we either get a counterexample, in which case the system is unsafe, or a proof for the property in the form of a quantifier-free inductive invariant. Second, the invariant is generalized to a quantified candidate invariant for the original system, and its validity is checked using either a form of abstraction, or quantified SMT reasoning. If the candidate invariant is valid, then the system is safe. Otherwise, further reasoning is required, e.g. by increasing the cardinality of the domain, and iterating the first step. The first option to check that a candidate invariant holds is based on an extension of the works on Parameter Abstraction of [29, 88] to our formalism. This technique computes a quantifier-free system and a quantifier-free property that, in case it holds, ensures that the original property is an invariant of the system. As a second option, we could use instead any off-the-shelf solver supporting SMT and quantifiers (e.g. Z3[50]) to discharge validity checks. However, a black-box approach to checking the validity of quantified formulae may cause the procedure to diverge in practice. Therefore, we also propose a more careful, resource-bounded approach to instantiation, that can be used to discharge quantified queries in a more controlled way.

It is important to note that these algorithms represent the first efforts to synthesize universal invariants for such a general class of systems. The effectiveness and efficiency of these algorithms are empirically validated in a subsequent section, where their experimental evaluation demonstrates advancements over existing methods. This empirical evidence not only underscores the practicality of our approaches but also establishes a solid foundation for future research in the field.

## 3.2 UPDR with implicit abstraction

### 3.2.1 Overview of UPDR

The paper [81] proposes a variant of the PDR/IC3 [21] algorithm to solve the problem stated in 2.3.1, but only in the case of $\mathcal{T}_E$ equal to the theory of Boolean. The algorithm, called UPDR (Universal Property Directed Reachability), represents first-order models by formulae with the notion of diagrams. When extending UPDR to our general setting, a main challenge is how to extend the computation of diagrams, to handle a general SMT theory $\mathcal{T}_E$. In this section, we combine UPDR with a form of predicate abstraction, to keep the computation of diagrams similar to the original case. We first introduce the necessary notions for the predicate abstraction and describe how to use them in our context. Then, we illustrate the general algorithm, and we discuss some approaches that we used for its implementation. Finally, we study its theoretical properties, most of which are a direct lifting of the ones given in [81]. In the last section, we give the proof of the main results.

As we already mentioned, this algorithm is based on the notion of diagrams. Diagrams are a method to represent a set of finite first-order models with an existentially quantified formula. We report the notion from [81]:

**Definition 12.** *Given a finite model $\mathcal{M}$, let $x_{m_i}$ be a fresh variable for each element $m_i$ in the universe of the model. The diagram of $\mathcal{M}$ over $\Sigma$ is the existential closure of the conjunction of the following literals:*

*(i). inequalities of the form $x_{m_i} \neq x_{m_j}$ for every pair of distinct elements $m_i, m_j$ in the model;*

*(ii). equalities of the form $c = x_m$ for every constant symbol $c$ in $\Sigma$ such that $\mathcal{M} \models c = m$;*

*(iii). the atomic formula $f(x_{m_{i_1}}, \ldots, x_{m_{i_n}}) = x_m$ for every function symbol $f$ of ariety $n$ if $\mathcal{M} \models f(m_{i_1}, \ldots, m_{i_n}) = m$.*

### 3.2.2 Implicit Indexed Predicate Abstraction

Predicate abstraction has been commonly considered for quantifier-free systems, but indexed predicates were introduced to abstract some restricted classes of systems with quantifiers [91]. In general, building the abstract version of a system can be expensive; however, various techniques that encode the abstract transition relation with logical formulas, such as [127, 90], have proven to be successful in the quantifier-free and Boolean case. In this section, we introduce a form of abstraction suitable for our purposes: instead of Boolean variables, we consider a set of first-order predicates, where free variables may occur.

In the following, we fix a set $I = \{i_1, \cdots, i_k\}$ of variables of index sort, and a finite set $\mathcal{P}(I) = \{p_1(I, X), \ldots, p_n(I, X)\}$ of $\Sigma(I)$ atoms (i.e. predicates over $\Sigma$ possibly containing free variables $I$ of sort $\mathcal{T}_I$). We can write $\mathcal{P}$ instead of $\mathcal{P}(I)$ for the sake of simplicity.

**Example 6.** Continuing the bakery example of the previous chapter, we can consider as a set of index predicates:

$$\mathcal{P}(i_1) = \{state[i_1] = idle, ticket[i_1] = 0, next\_ticket = 1, to\_serve = 1, state[i_1] = crit\},$$

where $i_1$ is a free variable of index sort.

We now introduce and generalize the main concepts of implicit abstraction for the quantified case.

**Definition 13.** *Given $\mathcal{P}(I)$ a set of index predicates, we define:*

- *$X_{\mathcal{P}(I)}$ is a set of fresh $\mathcal{T}_I$ predicates, one for each element $p \in \mathcal{P}(I)$, and with ariety the number of free variables in $p$:*

$$X_{\mathcal{P}(I)} := \{x_{p(I,X)}(I) \mid p(I,X) \in \mathcal{P}(I)\};$$

- *the formula*

$$H_{\mathcal{P}}(X_{\mathcal{P}}, X) := \forall I.\big(\bigwedge_{p(I,X)\in\mathcal{P}} x_{p(I,X)}(I) \leftrightarrow p(I,X)\big)$$

- *the formula*

$$EQ_{\mathcal{P}}(X, X') := \forall I.\big(\bigwedge_{p(I,X)\in\mathcal{P}} p(I,X) \leftrightarrow p(I,X'))$$

In the above definition, the new predicates $X_{\mathcal{P}}$ will act as new state variables of the abstract system. The formula $H_{\mathcal{P}}$ is used to define the simulation relation between an abstract and a concrete state. Finally, the formula $EQ_{\mathcal{P}}$ is used to compute the abstraction implicitly.

**Example 7.** Considering again the bakery algorithm, given the set of predicates of the previous example, we have a set of abstract variables

$$X_{\mathcal{P}(i_1)} = \{x_{state[i_1]=idle}(i_1), x_{ticket[i_1]=0}(i_1), x_{next\_ticket=1}, x_{to\_serve=1}, x_{state[i_1]=crit}(i_1)\}$$

Moreover, we have that $H_{\mathcal{P}}(X_{\mathcal{P}}, X)$ is equal to:

$$\begin{aligned}\forall i_1.(x_{state[i_1]=idle}(i_1) \leftrightarrow state[i_1] = idle) \wedge \forall i_1.(x_{ticket[i_1]=0}(i_1) \leftrightarrow ticket[i_1] = 0)\\ \wedge x_{next\_ticket=1} \leftrightarrow next\_ticket = 1 \wedge x_{to\_serve=1} \leftrightarrow to\_serve = 1 \wedge\\ \forall i_1.(x_{state[i_1]=critical}(i_1) \leftrightarrow state[i_1] = critical)\end{aligned}$$

Given a formula $\phi$, the predicate abstraction of $\phi$ with respect to $\mathcal{P}$, denoted $\hat{\phi}_{\mathcal{P}}$, is obtained by adding the abstraction relation to it and then existentially quantifying the variables $X$, i.e., $\hat{\phi}_{\mathcal{P}}(X_{\mathcal{P}}) := \exists X.(\phi(X) \wedge H_{\mathcal{P}}(X_{\mathcal{P}}, X))$, and similarly for a (transition)

formula over $X$ and $X'$ we have $\hat{\phi}_{\mathcal{P}}(X_{\mathcal{P}}, X'_{\mathcal{P}}) := \exists X, X'.(\phi(X, X') \wedge H_{\mathcal{P}}(X_{\mathcal{P}}, X) \wedge H_{\mathcal{P}}(X'_{\mathcal{P}}, X'))$. Moreover, given $\phi$, we denote with $\bar{\phi}$ the formula obtained by replacing the atoms in $\mathcal{P}$ with the corresponding predicates. Dually, if $\phi$ is a formula containing the $X_{\mathcal{P}}$ predicates, with $\phi[X_{\mathcal{P}}/\mathcal{P}]$ we denote the formula obtained by restoring the original predicates in place of the abstract ones. We remark that these are not direct substitutions, since the index variables may have been renamed, and thus some form of substitution could be needed. It is easy to see that, for any formula $\psi$, if all atoms of $\psi$ occur in $\mathcal{P}$, then $\hat{\psi}$ and $\bar{\psi}$ are logically equivalent under the assumption $\mathcal{H}_{\mathcal{P}}$. The indexed predicate abstraction of a system $S = (X, \iota(X), \tau(X, X'))$ is obtained by abstracting the initial and the transition conditions, i.e. $\hat{S}_{\mathcal{P}} = (X_{\mathcal{P}}, \hat{\iota}_{\mathcal{P}}(X_{\mathcal{P}}), \hat{\tau}_{\mathcal{P}}(X_{\mathcal{P}}, X'_{\mathcal{P}}))$. When clear from context, we will omit the subscript $\mathcal{P}$. We have:

**Proposition 15.** *Let $S = (X, \iota(X), \tau(X, X'))$ an array-based transition system, and $\phi(X)$ a formula. Let $\mathcal{P}$ a set of index predicates which contains all the atoms occurring in $\phi$. If $\hat{S}_{\mathcal{P}} \models \hat{\phi}$ then $S \models \phi$.*

In our algorithm, we start with a system $S$ and use a combination of UPDR, implicit abstraction, and indexed predicates to check if the property holds in an abstract system $\hat{S}$. However, the abstraction may be too coarse, and spurious counterexamples can occur. Notably, such counterexamples are actually a proof that no universal invariant exists over the set of predicates $\mathcal{P}$: this result is a direct consequence of the fact that our algorithm is a lifting of the original UPDR[81].

In the following, when working in the abstract space of $S$, the critical step for the algorithm is repeatedly checking whether a formula representing a model $\psi$ is inductive relative to the frame $F$. Frames will be defined as a set of negation of diagrams (except for the initial frame $F_0$ which is $\bar{\iota}$). The insight underlying implicit abstraction is to perform the check without actually computing the abstract version of $\tau$, but by encoding the simulation relation with a logical formula. This is done by checking the formula:

$$AbsRelInd(F, \tau, \psi, \mathcal{P}) = F(X_{\mathcal{P}}) \wedge \psi(X_{\mathcal{P}}) \wedge H_{\mathcal{P}}(X_{\mathcal{P}}, X)$$
$$\wedge EQ_{\mathcal{P}}(X, \bar{X}) \wedge \tau(\bar{X}, \bar{X}') \wedge EQ_{\mathcal{P}}(\bar{X}', X') \wedge \neg\psi(X'_{\mathcal{P}}) \wedge H_{\mathcal{P}}(X'_{\mathcal{P}}, X')$$

Where the variables $\bar{X}$ and $\bar{X}'$ are just a copy of the variables of $X$ and $X'$ that are used to describe the abstraction. We have:

**Proposition 16.** *Let $S = (X, \iota(X), \tau(X, X'))$ and let $\hat{S}_{\mathcal{P}}$ be its indexed predicate abstraction. Given any formulae $F, \psi$, then the formulae $AbsRelInd(F, \tau, \psi, \mathcal{P})$ and $F(X_{\mathcal{P}}) \wedge \hat{\tau}(X_{\mathcal{P}}, X'_{\mathcal{P}}) \wedge \psi(X_{\mathcal{P}}) \wedge \neg\psi(X'_{\mathcal{P}})$ are equisatisfiable.*

### 3.2.3 Algorithm description and pseudocode

We can now describe the whole procedure, depicted in the Algorithm 1.

As inputs, we have the array-based transition system $S = (X, \iota(X), \tau(X, X'))$ and a candidate invariant property $\phi$. At line 2, we set our initial set of predicates to be the set of $\Sigma(I)$-atoms occurring in the initial formula of the system, and in the property $\phi$.

**Algorithm 1:** UPDR + IA

```
1  Input: S = (X, ι(X), τ(X, X')), φ(X)
2  P(I) = {set of atoms occuring in ι, φ}
3  if not ῑ ∧ H_P ⊨ φ̄:
4      return cex  # cex in initial state
5  F_0 = {ῑ}
6  k = 1, F_k = ∅
7  while True:
8      while F_k ∧ H_P ∧ ¬φ̄ is sat:
9      # let σ be an A_I^E model, and σ_I its restriction on the index sort;
10         ψ = diag(σ_I)
11         if not RecBlock(ψ, k):
12             # an abstract counterexample π is provided
13             if not Concretize(π):
14                 P = P ∪ Refine(π)
15             else:
16                 return unsafe
17     # if no models, add new frame:
18     k = k + 1, F_k = ∅
19     # propagation phase
20     for i = 1, ..., k − 1:
21         for each lemma ψ ∈ F:
22             if AbsRelInd(F_i, τ, ψ, P) is unsat:
23                 add ψ to F_{i+1}
24             if F_i = F_{i+1}:
25                 Return safe
```

This choice simplifies the description of the algorithm, but it is also possible to start with a different set of predicates, as long as all the atoms in $\phi$ are contained in $\mathcal{P}$.

Then, at line 3, the algorithm checks whether there is a violation of the property in the initial formula. If a counterexample is not encountered, the algorithm is initialized and the main PDR/IC3 loop starts. The algorithm maintains a set of frames $F_0, \ldots, F_N$, which are an approximate reachability sequence of $\hat{S}_{\mathcal{P}}$, as defined in Definition 4. Initially, we set $F_0 = \{\bar{\iota}\}$, $F_1$ to be empty, and we set $N$, the counter of the length of the frame, to be equal to 1. Then (line 8), we loop over models in the intersection between the last frame $F_N$ and $\neg\hat{\phi}$. For each of those models, we consider its restriction on the index sort as defined in the preliminaries.

Then, the function *diag* computes the diagram of the (finite) model over the signature $\Sigma_I \cup X_P$ (Definition 12). It is important to note here that we compute the diagram only using the universe of the index sort, but we extend the index signature with the new abstract (index) predicates.

Afterward, the procedure *RecBlock* (Algorithm 2) tries to either block such a diagram, showing that the corresponding models are unreachable from the previous frame

**Algorithm 2:** $RecBlock(\psi, N)$

```
1  if N = 0
2      return False
3  While AbsRelInd(F_{i-1}, τ, ¬ψ, P) is sat:
4  # let σ' be an A_I^E model, and σ'_I its restriction on the index sort;
5      ψ' = diag(σ'_I)
6      if not RecBlock(ψ', N − 1):
7          return cex
8  g = Generalize(¬ψ, N)
9  add g to F_1, ..., F_N
10 Return True
```

with an abstract transition, or computes a diagram from the set of backward reachable (abstract) states, and recursively calls itself. If eventually a diagram is blocked, then the corresponding frames are strengthened by adding a (generalization of) the diagrams to them. If the procedure finds a diagram in the first frame, then we are in the presence of an abstract counterexample of $\hat{S}_{\mathcal{P}}$:

**Definition 14.** *(Abstract Counterexample) Let $F_0, \ldots, F_N$ be an approximate reachability sequence for $\hat{S}_{\mathcal{P}}$ and $\hat{\phi}$. An abstract counterexample is a sequence of models $\pi := \sigma_0, \ldots, \sigma_N$ such that:*

- $\sigma_i \models F_i,\ \forall i. 0 \leq i < N$;
- $diag(\sigma_i) \wedge \hat{\tau} \wedge diag'(\sigma_{i+1})$ *is satisfiable;*
- $\sigma_N \models \neg\hat{\phi}$.

Note the fact that we compute diagrams only over index model and over the signature $\Sigma_I \cup X_P$. This is enough to capture abstract counterexamples for $\hat{S}_{\mathcal{P}}$; we have:

**Proposition 17.** *Given a set of predicates $\mathcal{P}$, if the procedure RecBlock (Algorithm 2) returns false, then there exists an abstract counterexample for $\hat{S}_{\mathcal{P}}$.*

In line 12, the abstract counterexample is analyzed: if the counterexample is spurious, then we try to refine the set of predicates, and the loop continues. Our refinement procedure is described in the next section. Else, if a concrete counterexample is found, the algorithm terminates with an unsafe result. Finally, the propagation phase of the algorithm tries to push diagrams in $F_i$ to $F_{i+1}$. If two frames are equal during this phase, then an inductive invariant is found by the algorithm. Otherwise, the loop restarts with a larger trace.

**Example 8.** We give an overview of some steps of the procedure over the bakery algorithm. Given $\mathcal{P}(i_1)$ and $X_P$ of the previous examples, we have that

$$\bar{\iota} = \forall i_1. \big(x_{state[i_1]=idle}(i_1) \wedge x_{t[i_1]=0}(i_1) \wedge x_{next\_ticket=1} \wedge x_{to\_serve=1}\big)$$

and

$$\bar{\phi} = \forall i, j.\big(i \neq j \rightarrow \neg(x_{state[i_1]=crit}(i) \wedge x_{state[i_1]=crit}(j)\big).$$

$H_{\mathcal{P}}(X_{\mathcal{P}}, X)$ was also shown in the previous example. It is easy to see that $\bar{\iota} \wedge H_P \wedge \neg\bar{\phi}$ is unsatisfiable. Therefore, we have $F_0 = \hat{\iota}$ and $F_1 = \top$. We enter the main loop, and consider models of $H_P \wedge \neg\bar{\phi}$; this formula is satisfiable, and we consider the restriction of a model over the index sort. Suppose that this restriction is given by:

- A finite index universe $\{a, b\}$ with $a \neq b$
- $x_{next\_ticket=1}$ is true in the model, $x_{to\_serve=1}$ is false;
- the predicate $x_{state[i_1]=idle}$ does not hold for $a, b$ but $x_{state[i_1]=crit}$ does.

So, the corresponding diagram over $X_{\mathcal{P}}$ is

$$\begin{aligned}\psi = \exists i_1, i_2.\big(&i_1 \neq i_2 \wedge x_{next\_ticket=1} \wedge \neg x_{to\_serve=1} \wedge \neg x_{state[i_1]=idle}(i_1) \\ &\wedge \neg x_{state[i_1]=idle}(i_2) \wedge x_{state[i_1]=crit}(i_1) \wedge x_{state[i_1]=crit}(i_2)\big).\end{aligned}$$

We now call $RecBlock(\psi, 1)$, and therefore consider $AbsRelInd(F_0, \tau, \neg\psi, \mathcal{P})$. This is unsatisfiable, so we can add (a generalization of) $\neg\psi$ to $F_1$. Eventually, $F_1 \wedge H_P \wedge \neg\hat{\phi}$ will be no longer satisfiable, so we introduce a new frame $F_2$. (For simplicity, we skip the propagation phase). At this point, $F_2 \wedge H_P \wedge \neg\hat{\phi}$ is satisfiable, and it is possible to have a sequence of models and corresponding diagrams $\psi, \psi', \psi''$ such that $RecBlock(\psi, 2)$ recursively calls $RecBlock(\psi', 1)$ which again calls $RecBlock(\psi'', 0)$. This corresponds to an abstract counterexample. □

The generalization phase, at line 7 of Algorithm 2, is not relevant to the soundness of the algorithm, but it is a crucial part for its efficiency. During this phase, the diagram $\neg\psi$ is weakened to a more general formula $g$, which implies the negation of the diagram but blocks more models. In our implementation, we used a technique based on unsat core extraction, used also in other PDR/IC3 variants [81, 33].

### 3.2.4 Concretizing counterexamples and refinement

Given an abstract counterexample, we can try to associate it with a concrete counterexample, thus concluding the algorithm with a negative result, or we could discover that no concrete counterexample corresponds to the abstract one. More formally, together with an abstract counterexample $\pi$, the algorithm finds a sequence of diagrams $\psi_0(X_{\mathcal{P}}), \ldots, \psi_k(X_{\mathcal{P}})$. However, differently from the Boolean and quantifier-free case, the abstract unrolling

$$\psi_0(X^0_{\mathcal{P}}) \wedge \bigwedge_{i=1,\ldots,k} \hat{\tau}(X^{i-1}_{\mathcal{P}}, X^i_{\mathcal{P}}) \wedge \psi_i(X^i_{\mathcal{P}}) \tag{3.1}$$

can still be unsatisfiable. That is, we may have two models $\sigma_1, \sigma_2$ such that $diag(\sigma_1) \wedge \hat{\tau} \wedge diag'(\sigma_2)$ is satisfiable, but there is no abstract transition between $\sigma_1$ and $\sigma_2$. A

possible reason for this is that $\sigma_1$ and $\sigma_2$ may have different universes, and the diagrams abstract $\sigma_1$ and $\sigma_2$ by upward-closing them w.r.t. the submodel relation (see section 5 of [81] for examples and more details). This is also a proof that, to eliminate the abstract counterexample, a universal invariant over $\mathcal{P}$ is not enough. The automatic discovery of invariants with quantifier alternation is an active area of research [129, 113, 71]; however, in this thesis, we focus only on universal invariants (note that the frames are the conjunction of negations of diagrams, therefore they are universally quantified in prenex normal form). Therefore, when we encounter an abstract counterexample, we try to expand our predicate set $\mathcal{P}$ and search for a new universal invariant on a larger language. In practice, in order to check whether a sequence of diagrams corresponds to a concrete counterexample, we could check the satisfiability of the concrete unrolling of the system, by replacing atoms in the diagram with their non-abstracted version, i.e. checking if the formula

$$\psi_0[X_{\mathcal{P}}/\mathcal{P}](X^0) \wedge \bigwedge_{i=1,\dots,k} \tau(X^{i-1}, X^i) \wedge \psi_i[X_{\mathcal{P}}/\mathcal{P}](X^i). \tag{3.2}$$

is satisfiable. If such a query is satisfiable, then we are given a sequence of models which leads to a violation of the property. In case of unsatisfiability of the (3.2), a typical approach would be to consider the sequence of formulae $\iota_0 = \psi_0(X^0)[X_{\mathcal{P}}/\mathcal{P}]$, and $\iota_i = \tau(X^{i-1}, X^i) \wedge \psi_i[X_{\mathcal{P}}/\mathcal{P}](X^i)$ (for all $1 \leq i \leq n-1$), and extract an interpolant sequence (2.1.2) from it. However, there are two main practical issues in using directly this encoding: $(i)$ since the formula (4.3) contains quantifiers, proving its (un)satisfiability is very challenging, and $(ii)$ extracting interpolants from quantified queries is scarcely supported by existing solvers.

We propose instead to consider an under-approximation of (4.3), i.e. to consider a finite model encoding of it in a certain size. That is to say, instead of trying to unroll the counterexample in all the instances of the system, we try to block it in a ground instance $S_n$. To choose an appropriate size, recall that a diagram $\psi_i$ is an existentially closed formula that is built from an index model with cardinality $n_i$, where $n_i$ is the number of existentially quantified variables in $\psi_i$. Thus, we need to choose a size able to represent at least all the diagrams in the abstract counterexample. Therefore, we take $n = max\{n_i | i = 1, \dots, k\}$, which is the least integer such that each diagram is satisfiable (we take the least for tractability reasons). Then, we consider

$$(\psi_0[X_{\mathcal{P}}/\mathcal{P}])_n(C^0, X^0) \wedge \bigwedge_{i=1,\dots,k} \tau_n(C^i, X^{i-1}, X^i) \wedge (\psi_i[X_{\mathcal{P}}/\mathcal{P}])_n(C^i, X^i). \tag{3.3}$$

where $C$ and $\tau_n$ are defined in 2.3.2. If such under-approximated unrolling is satisfiable, we have a counterexample in $S_n$, and we can terminate the algorithm with an unsafe result. Otherwise, we can compute an interpolant sequence, and add all the atoms of the interpolants (after replacing the symbols in $C$ with free variables) to the predicate set $\mathcal{P}$. Note however that such a refinement will rule out (3.3), but in general not guaranteed to rule out (4.3): that is, the counterexample can occur again in a greater size. If this happens, we will consider larger and larger finite encodings, and repeat the

process. In this way, such a refinement will diverge when an invariant with not only universal quantification is needed, as the formula 4.3 remains sat, and larger and larger ground instances will be explored, always yielding the same predicates.

**Example 9.** Continuing the latter example, suppose we are given a sequence of diagrams $\psi'', \psi', \psi$, each with two existentially-quantified variables. Therefore, we consider the unrolling of (3.3) in $S_2$, by introducing two fresh index constants $c_1$, $c_2$. Such a formula is unsatisfiable, yielding and a possible interpolant $t[c_1] = to\_serve$. Therefore, we restart the loop with $\mathcal{P} = \mathcal{P} \cup \{t[i_1] = to\_serve\}$. □

We finally remark that, since we are working with infinite-state systems, the choice of the correct predicates make a huge impact in the overall convergence of the problem. Even if our procedure synthesize the predicates that are able to rule out the counterexample given by the formula 4.3, we may diverge in the overall procedure. Predicates that are too specific may fail to generalize across the state space, leading to a need for an ever-increasing number of predicates and risking divergence. On the other hand, predicates that are overly general might not refine the state space sufficiently, failing to eliminate spurious paths effectively. Moreover, navigating the vast space of potential predicates to find a set that is both minimal and sufficient for convergence is a significant challenge in itself. Efficiently selecting and synthesizing such predicates is still a challenge for research in verification problems. highlighting the intricate relationship between predicate selection and the success of interpolant-based methods in handling infinite-state systems [33].

### 3.2.5 Properties

We have the following properties of the algorithm:

**Proposition 18.** *(Soundness) If the algorithm terminates with safe, then $S \models \phi$. If the algorithm terminates with unsafe, then $S \not\models \phi$.*

Moreover, given propositions 16 and 17, we can establish the following fact about Algorithm 1: once a set of indexed predicates $\mathcal{P}$ is fixed, our procedure simulates the standard UPDR [81] algorithm over the system $\hat{S}_{\mathcal{P}}$. As already anticipated, the algorithm has a result of partial completeness over a certain set of universally quantified formulae. To be more precise, let $\mathcal{L}$ be the set of formulae of the form $\forall \underline{i}.\psi(\underline{i})$, with $\psi(\underline{i})$ a Boolean combination of $\Sigma_I(\underline{i})$-atoms and elements of $\mathcal{P}(\underline{i})$. We have the following proposition (by lifting Proposition 5.6 in [81]):

**Proposition 19.** *(Partial Completeness) If the algorithm 1 finds an abstract counterexample, then no inductive invariant in $\mathcal{L}$ exists for $S$ and $\phi$.*

We already mentioned that the problem we are addressing in this paper is undecidable, and there are indeed many possible causes of non-termination of this algorithm. First, we rely continuously on first-order reasoning, which is in general undecidable. However, a nice property of our procedure is the following, which follows from simple logical manipulation after noticing that the formulae defining the implicit abstraction contain only universal quantification:

**Proposition 20.** *If the input system $S = (X, \iota(X), \tau(X, X'))$ is defined by formulae falling in the decidable fragment of Proposition 11, then each satisfiability check of Algorithm 1 falls again in the decidable fragment.*

Proposition 20 ensures that, if the initial and the transition formulae are described in a decidable setting, then we are not stuck in first-order reasoning. However, the whole procedure may still be non-terminating: the sources of divergence may be an infinite series of refinements, or the unbounded exploration of an infinite-search space (in fact, even after fixing a set of predicates $\mathcal{P}$, the abstract system $\hat{S}_\mathcal{P}$ is still infinite-state: every model is finite, but there is no bound on the cardinalities of their universes). For the latter problem, it is a common paradigm in the literature to use well-quasi-orders (wqo) [1] to prove the termination of reachability procedures for infinite-state systems. In [81], it is shown that UPDR terminates as well if the state of the system forms a wqo. In our context, we can state the following proposition:

**Proposition 21.** *Given a set of predicates $\mathcal{P}$, if the states of $\hat{S}_\mathcal{P}$ form a well-quasi-order, then the algorithm either finds an inductive invariant or a proof of the non-existence of an inductive invariant over $\Sigma_I \cup \mathcal{P}$.*

As a use case of the latter result, in [110] it is shown that the theory of linear order in a signature containing arbitrary uninterpreted constants and unary predicates has the wqo property. Therefore, we have:

**Corollary 22.** *Suppose $\mathcal{T}_I$ is the theory of a linear order. Suppose also that, at any point of the procedure, the index predicates $\mathcal{P}$ contains only unary or constant predicates. Then, if there exists a universal invariant over $\Sigma_I \cup \mathcal{P}$, algorithm 1 is guaranteed to find it.*

### 3.2.6 Proof of main results

In the following, we will refer for the sake of simplicity to a system with a single array state variable $S = (x, \iota(x), \tau(x, x'))$. All the results here can be extended without loss of generality to any finite number of array variables $X = \{x_1, \ldots, x_n\}$.

#### Proofs of subsection 3.2.2

In this section, we start by proving Proposition 15. We consider fixed a set of index predicates $\mathcal{P}(I) = \{p_1(I, x), \ldots, p_n(I, x)\}$. Given a system $S = (x, \iota(x), \tau(x, x'))$, we denote with $\hat{S}_\mathcal{P} = (X_\mathcal{P}, \hat{\iota}_\mathcal{P}(X_\mathcal{P}), \hat{\tau}_\mathcal{P}(X_\mathcal{P}, X'_\mathcal{P}))$ its indexed predicate abstraction, as defined in 3.2.2.

We start by defining the simulation relation between the abstract and the concrete system. Note that $S$ is defined over the theory $A_I^E$, whereas $\hat{S}$ is defined only over the theory $\mathcal{T}_I$. Therefore, a state of $S$ is given by a model $\mathcal{M}$ of $A_I^E$, and a valuation $s$ of the array symbol $x$ as a function. Instead, a state of $\hat{S}_\mathcal{P}$ is given by an index model $\mathcal{M}'$ and a valuation $\hat{s}$ of the index predicates $X_\mathcal{P}$ as a subset of (a Cartesian product of) the universe of $\mathcal{M}'$. Given a model $\mathcal{M}$ of $A_I^E$, we denote with $\mathcal{M}_{|I}$ its restriction on the index sort.

**Definition 15.** *Let $(\mathcal{M}, s)$ be a state of $S$, and $(\mathcal{M}', \hat{s})$ a state of $\hat{S}$. The two states are in the relation $\alpha_{\mathcal{P}}$ if and only if $\mathcal{M}_{|I} = \mathcal{M}'$ and $s, \hat{s} \models H_{\mathcal{P}}(X_{\mathcal{P}}, x)$.*

Recall that, for a formula $\phi(X)$ whose atoms are all contained in $\mathcal{P}$, we write $\bar{\phi}$ to denote the formula obtained by substituting the $\mathcal{P}$-atoms with the corresponding predicates in $X_{\mathcal{P}}$. We have that (see Definition 8):

**Proposition 23.** *The relation $\alpha_{\mathcal{P}}$ preserves all the formulae whose set of atoms is contained in $\mathcal{P}$, modulo the rewriting operator $-$.*

We also define the restriction of a state to its abstract counterpart:

**Definition 16.** *Let $s, \mathcal{M}$ be a state of $S$. We define a state of $\hat{S}_{\mathcal{P}}$ by considering $\mathcal{M}_{|I}$, and $\hat{s}_{\mathcal{P}}$ a valuation of the predicates $X_P$ into $\mathcal{M}_{|I}$ given by:*

$$\hat{s}_{\mathcal{P}}(x_{p(I,x)}) = \{M \in \mathcal{U}(\mathcal{M}_{|I}) \times ... \times \mathcal{U}(\mathcal{M}_{|I}) \ | \mathcal{M}, s \models p(M, x)\}.$$

*i.e. the predicate $x_{p(I,x)}$ is interpreted as the set of all elements of the universes of the index models such that they satisfy the predicate $p(I, x)$ (under the valuation induced by $s$).*

The following proposition follows directly from this definition:

**Proposition 24.** *Let $\mathcal{M}$ a model of $A_I^E$, and $s$ a valuation into $\mathcal{M}$. Let $\hat{s}_{\mathcal{P}}$ as in Definition 16. Then, we have that $s, \hat{s}_{\mathcal{P}} \in \alpha_{\mathcal{P}}$.*

It is now easy to prove that the index predicate abstraction simulates the original system:

**Lemma 25.** *Given $(\mathcal{M}, s)$ a state of $S$ such that $\mathcal{M}, s \models \iota(x)$, then there exists a state $(\mathcal{M}', \hat{s})$ of $\hat{S}_{\mathcal{P}}$ such that $\mathcal{M}', \hat{s} \models \hat{\iota}_{\mathcal{P}}(X_{\mathcal{P}})$, and $s, \hat{s} \in \alpha_{\mathcal{P}}$.*

*Proof.* Recall that $\hat{\iota}_{\mathcal{P}}(X_{\mathcal{P}}) := \exists x.(\iota(x) \wedge H_{\mathcal{P}}(X_{\mathcal{P}}, x))$. Let $\mathcal{M}' = \mathcal{M}_{|I}$, and $\hat{s} = \hat{s}_{\mathcal{P}}$ as defined in the previous definition. From 24, we have that $s, \hat{s} \in \alpha_{\mathcal{P}}$, and the lemma follows. □

Similarly, we can prove that:

**Lemma 26.** *Let $\mathcal{M}$ a model of $A_I^E$, and let $s, \hat{s}$ a couple of states such that $s, \hat{s} \in \alpha_{\mathcal{P}}$. Then, for every $s'$ such that $\mathcal{M}, s, s' \models \tau(x, x')$, there exists an $\hat{s}'$ such that $\mathcal{M}_{|I}, \hat{s}, \hat{s}' \models \hat{\tau}_{\mathcal{P}}(X_{\mathcal{P}}, X'_{\mathcal{P}})$.*

We already mentioned that the index predicates abstraction preserves the validity of all the formulas whose atoms are contained in $\mathcal{P}$. Therefore, we have the following (stated as Proposition 15 in Section 3.2):

**Proposition 27.** *Let $S = (X, \iota(X), \tau(X, X'))$ an array-based transition system, and $\phi(X)$ a formula. Let $\mathcal{P}$ a set of index predicates which contains all the atoms occurring in $\phi$. If $\hat{S}_{\mathcal{P}} \models \hat{\phi}$ then $S \models \phi$.*

*Proof.* By Lemmas 25, 26, we have that the relation $\alpha_{\mathcal{P}}$ is a simulation relation, and therefore it preserves reachability (2). Suppose that $S \models \neg\phi$: then, there exists a finite path to a state $s$ such that $s \models \neg\phi$. By using definition 16, it follows that there exists a reachable state $\hat{s}$ of $\hat{S}$ such that $s, \hat{s} \in \alpha_{\mathcal{P}}$. Since $\mathcal{P}$ contains all the atoms of $\phi$, we have that $\hat{s} \models \neg\hat{\phi}$, a contradiction. □

Moreover, can now establish the following fact. Recall that

$$AbsRelInd(F, \tau, \psi, \mathcal{P}) = F(X_{\mathcal{P}}) \wedge \psi(X_{\mathcal{P}}) \wedge H_{\mathcal{P}}(X_{\mathcal{P}}, X)$$
$$\wedge EQ_{\mathcal{P}}(X, \bar{X}) \wedge \tau(\bar{X}, \bar{X}') \wedge EQ_{\mathcal{P}}(\bar{X}', X') \wedge \neg\psi(X'_{\mathcal{P}}) \wedge H_{\mathcal{P}}(X'_{\mathcal{P}}, X')$$

We have:

**Proposition 28.** *Given a system $S = (X, \iota(X), \tau(X, X'))$ and its abstraction $\hat{S}_{\mathcal{P}} = (X_{\mathcal{P}}, \hat{\iota}_{\mathcal{P}}(X_{\mathcal{P}}), \hat{\tau}_{\mathcal{P}}(X_{\mathcal{P}}, X'_{\mathcal{P}}))$, given any formulae $F, \psi$, then the formulae $AbsRelInd(F, \tau, \psi, \mathcal{P})$ and $F(X_{\mathcal{P}}) \wedge \hat{\tau}(X_{\mathcal{P}}, X'_{\mathcal{P}}) \wedge \psi(X_{\mathcal{P}}) \wedge \neg\psi(X'_{\mathcal{P}})$ are equisatisfiable. Moreover, if $s, s' \models AbsRelInd(F, \tau, \psi, \mathcal{P})$, then $\hat{s}_{\mathcal{P}}, \hat{s'}_{\mathcal{P}} \models F(X_{\mathcal{P}}) \wedge \hat{\tau}(X_{\mathcal{P}}, X'_{\mathcal{P}}) \wedge \psi(X_{\mathcal{P}}) \wedge \neg\psi(X'_{\mathcal{P}})$.*

*Proof.* (sketch) The proof is similar to Theorem 1 in [33], by using Definition 16 instead of the projection on Boolean values. □

Proposition 17 is an immediate consequence of the latter, once noticing that we compute diagrams from the restriction of models over the signature $\Sigma_I \cup X_{\mathcal{P}}$: every step of the algorithm is equivalent to performing UPDR [81] on $\hat{S}_{\mathcal{P}}$.

### Proofs of subsection 3.2.3

We give here the proof of the Proposition 18. First, we have

**Proposition 29.** *The frames $F_0(X_{\mathcal{P}}), \ldots, F_n(X_{\mathcal{P}})$ are an approximate reachability sequence for $\hat{S}_{\mathcal{P}}$.*

*Proof.* (sketch) The proof follows the same steps as the one in [33], Lemma 1. □

**Proposition 30.** *If algorithm 1 terminates with SAFE, then $S \models \phi$. If algorithm 1 terminates with UNSAFE, then $S \not\models \phi$*

*Proof.* The algorithm terminates with SAFE when there exists an approximate reachability sequence $F_0(X_{\mathcal{P}}), \ldots, F_n(X_{\mathcal{P}})$ such that, for some $i$, $F_{i+1} \models F_i$. From Proposition 1, we have that $\hat{S}_{\mathcal{P}} \models \hat{\phi}$. From Proposition 27, this implies that $S \models \phi$. If the algorithm terminates with UNSAFE, then the unrolling 4.3 is satisfiable, and we have a counterexample. □

## 3.3 Lambda: Learning lemmas from ground instances

In the preceding section, we introduced UPDR with implicit abstraction as a technique for the automatic discovery of universally quantified inductive invariants for array-based transition systems. The method entails the execution of satisfiability checks that necessitate quantification for both deriving models of quantified formulae and computing diagrams, or for proving unsatisfiability to incorporate clauses into frames. Despite its theoretical prowess, this approach incurs a significant computational overhead.

This section introduces an alternative strategy that, while potentially less comprehensive, lacking any definitive assurance of invariant discovery, it exhibits better practical performance. The method is based on the investigation of concrete instances, that has long been acknowledged as a valuable heuristic in the verification of parameterized systems [114, 46, 70]. The intuition is that in most cases if a counterexample to a property exists, it can be detected for small values of the parameter. Moreover, if a property holds, the reason for that should be the same for all values of the parameter (at least after a certain threshold value).

In this section, we present a second algorithm for solving the invariant problem for array-based transition systems, inspired by the ideas above: we try to generalize an invariant for the system by exploring small ground instances. In contrast to the algorithm of the previous section, this algorithm tries to guess an invariant, rather than constructing one incrementally. Even if we do not provide any theoretical guarantees that this guess will eventually be correct, this approach relies less on quantified reasoning, which turns out to be better from a practical standpoint. We start by giving a high-level overview of our method, depicted also in Fig. 3.1. Then, we will go into the details of the algorithm, discussing our generalization technique 3.3.2, and presenting two different approaches to check whether our generalization is correct (the Candidate Checking box) 3.3.3, 3.3.4. We finish the section by discussing the properties of the algorithm 3.3.5.

### 3.3.1 High level algorithm

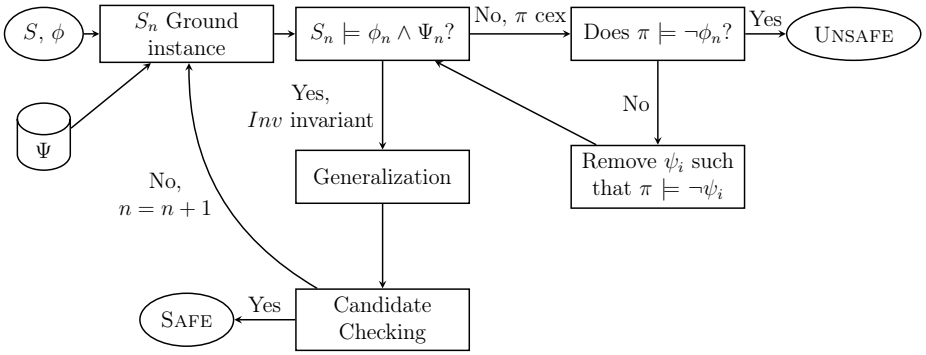


Figure 3.1: An overview of the algorithm.

We consider as inputs for the algorithm an array-based transition system $S$ and a

candidate invariant $\phi$. We also maintain a set of formulae $\Psi = \{\psi_1, \dots, \}$, initially empty, that contains universally quantified *candidate lemmas*. We also initialize $n$, a counter for the size of the ground instance we explore, equal to 1. We perform the following steps:

- we consider $S_n$, as defined in 2.3.2, and then use a model checker to check whether $\phi_n \wedge \Psi_n$ is an invariant for the ground instance. If we get a counterexample, we check if the property itself is falsified (thus terminating the algorithm with UNSAFE result), or we remove the lemmas that are proved to be false in size $n$. In the latter case, we repeat the ground model checking query, until either a true counterexample is obtained, or the property holds.
- If the model checker proves the property, we consider an inductive invariant $Inv$ of $S_n$. From such a formula, we synthesize a set of new lemmas, that we add to $\Psi$ (Generalization).
- In the Candidate Checking box, we check if the property $\phi$ (together with the lemmas $\Psi$) is an inductive invariant for the original system $S$. We propose two methods for performing this check, described in 3.3.3, 3.3.4.
- In case of success, the property is proved and we have found an inductive invariant. In case of a failure, we need a better candidate invariant: we restart the loop with a new exploration from size $n + 1$.

Note there are many possible causes of non-termination of the algorithm (again, since we are dealing with undecidable problems, this is not avoidable): the main problems are the candidate checking box, which involves quantified reasoning, and the existence of an integer $n$ such that the generalized formula obtained after model checking a ground instance of size $n$ is inductive also for all other instances. The pseudocode of the procedure is reported in Algorithm 3. In the next sections, we describe all the sub-procedures in detail.

### 3.3.2 Generalization

A key step of this algorithm is the one of producing candidate invariants by generalizing proofs from a ground instance. In this section, we show our approach. We suppose to have a model checker capable of proving or disproving that $S_n \models \phi_n$ (e.g. [33]). If a counterexample is not found, we also suppose that the model checker returns a formula $Inv$ which witnesses the proof, i.e. an inductive invariant of $S_n$ and $\phi_n$. From this witness, we generalize a set of quantified lemmas.

**Definition 17** (Generalization)**.** *Let $S$ be an array-based transition system, $\phi$ a candidate invariant, and suppose $S_n \models \phi_n$. A* $n$-generalization *is a (quantified) formula $\Psi$ such that $\Psi_n$ is an inductive invariant for $S_n$ and $\Psi_n \models \phi_n$.*

In practice, we exploit the technique that we used in [34], inspired by [114], which works as follows. Suppose that $Inv$ is in CNF. Then, $Inv = \mathcal{C}_1 \wedge \cdots \wedge \mathcal{C}_m$ is a conjunction

**Algorithm 3:** Learning lemmas from ground instances

```
1  Input: S = (X, ι(X), τ(X, X')), φ(X)
2  n = 1
3  while True:
4      S_n = grounding(S, n)
5      while not S_n ⊨ φ_n ∧ Ψ_n:
6          # model checker returns a cex π from size n
7          if π ⊨ ¬φ_n:
8              return UNSAFE, π
9          else:
10             Ψ = Ψ \ {ψ_i | π ⊨ ¬ψ_i}
11     # model checker returns an inductive invariant Inv from size n
12     Ψ = Ψ ∪ generalization(Inv)
13     if InvariantChecking(S, φ, Ψ):
14         return SAFE, φ ∧ Ψ
15     else:
16         n = n + 1
```

of clauses. From every one of such clauses, we will obtain a universally quantified formula. Let $\mathit{AllDiff}(I)$ be the formula that states that all variables in $I$ are different from each other. For all $j \in \{1, \ldots, m\}$, let $\psi_j = \forall I. \mathit{AllDiff}(I) \rightarrow \mathcal{C}_j[C/I]$ - where $C$ are the variables introduced in 2.3.2. Let $\Psi = \bigwedge_{j=1}^{m} \psi_j$. It follows from Proposition (14) that such a $\Psi$ is a $n$-generalization.

It should be clear that our technique can infer lemmas with only universal quantifiers, but more generalizations are possible [70, 52]. For example, if a clause of the inductive invariant were $l(c_1) \vee ... \vee l(c_n)$, a possible generalization of that clause would be $\exists x.l(x)$.

### Minimizing modulo symmetries

Our next step will be to try to prove that the generalized lemmas from size $n$ also hold for all other ground instances. Therefore, it is intuitive to try to weaken as much as possible the generalization $\Psi$, to increase the chances that its inductiveness will be preserved in other instances. So, before generalization, we exploit the invariant minimization techniques described in [77] to weaken the inductive invariant $Inv$ by removing unnecessary clauses. However, note that, with our generalization technique, two symmetric clauses produce the same quantified formula: if $\sigma$ is a substitution of the $C$'s, the formulae obtained by generalizing a clause $\mathcal{C}(C)$ or $\sigma\mathcal{C}(C)$ are logically equivalent. So, we apply the following strategy: given a clause $\mathcal{C}(C)$ in $Inv$, we add to the invariant all the 'symmetric' versions $\sigma\mathcal{C}$, where $\sigma$ ranges over all possible substitutions of the $C$'s. By Proposition (14), we can safely add those clauses to $Inv$ and it will remain inductive. Then, during the minimization process, a clause is removed from the invariant only if all its 'symmetric' versions are.

**Example 10.** Following the bakery example, by using the model checker described [33] on a 2-instance of the protocol we obtain an inductive invariant which is made of 27 inductive clauses. After minimizing the invariant modulo the symmetries of the finite isntance, we are left with the following eight clauses ($c_1, c_2$ are considered implicitly different):

$$
\begin{aligned}
&(\neg(to\_serve = ticket[c_1]) \vee \neg(state[c_2] = crit))\\
&\big(\neg(to\_serve = ticket[c_1]) \vee \neg(state[c_1] = idle)\big),\\
&\big(ticket[c_1] = ticket[c_2] \vee (\neg(state[c_1] = idle) \wedge \neg(state[c_2] = idle))\big),\\
&(1 \leq next\_ticket - ticket[c_2]),\\
&\neg(next\_ticket = ticket[c_1]),\\
&\big(ticket[c_2] = 0 \vee \neg(state[c_2] = idle)\big),\\
&(1 \leq to\_serve),\\
&\big(to\_serve = ticket[c_1] \vee \neg(state[c_1] = crit)\big),
\end{aligned}
$$

Once generalized, we obtain the following set of lemmas, which is indeed an inductive strengthening for the property:

$$
\begin{aligned}
&\forall j \forall i.(i \neq j \rightarrow (\neg(to\_serve = ticket[i]) \vee \neg(state[j] = crit))),\\
&\forall i.\big(\neg(to\_serve = ticket[i]) \vee \neg(state[i] = idle)\big),\\
&\forall j \forall i.\big(i \neq j \rightarrow (ticket[i] = ticket[j] \vee (\neg(state[i] = idle) \wedge \neg(state[j] = idle)))\big),\\
&\forall j.(1 \leq next\_ticket - ticket[j]),\\
&\forall i.\neg(next\_ticket = ticket[i]),\\
&\forall j.\big(ticket[j] = 0 \vee \neg(state[j] = idle)\big),\\
&(1 \leq to\_serve),\\
&\forall i.\big(to\_serve = ticket[i] \vee \neg(state[i] = crit)\big).
\end{aligned}
$$

□

### 3.3.3 Candidate Checking via Parameter Abstraction

In this section, we discuss how we can check that a formula $\phi$ is an invariant of $S$, by using only a quantifier-free model checker instead of a theorem prover. We do this by constructing a new system $\tilde{S}$, called the parameter abstraction of $S$, which is quantifier-free and simulates $S$. The first version of this approach was introduced in [88]; 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 over-approximates the behavior of the system

in the array locations that are not explicitly tracked, we use *n environmental (index) variables* which are allowed to change non-deterministically 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 all other indexes.

**Hypothesis**: In the remaining of this section, we suppose additional hypothesis on the shape of the system $S$. These hypothesis are needed for the construction of the abstraction system $\tilde{S}$, and they turn out not to be particularly restrictive: for instance, large classes of parameterized systems can still be described in this setting. In particular, we suppose that the candidate property $\phi(X)$ and the initial formula $\iota(X)$ are universal formulae. Moreover, we suppose that the transition formula $\tau(X, X')$ is a disjunction of formulae of the form $\exists I \forall J.\psi(I, J, X, X')$, with $\psi$ quantifier-free.The formulae defined in this way are close to the one in 11, altough we do not require the absence of functions with, as codomain, the index sort.

### Preprocessing steps

Before starting constructing the abstract system, we apply some preprocessing steps to $S$, justified by the following two propositions. The first proposition is basically an induction principle:

**Proposition 31** (Guard strengthening [88]). *Let $C = (X, \iota(X), \tau(X, X'))$ be a symbolic transition system and let $\phi(X)$ be a formula. Let $C_\phi := (X, \iota(X), \tau(X, X') \wedge \phi(X))$ (called the guard-strengthening of $C$ with respect to $\phi(X)$). If $\phi$ is an invariant of $C_\phi$, then it is also an invariant of $C$.*

The second proposition is used to remove quantified variables in candidate properties:

**Proposition 32** (Removing quantifiers[102]). *Let $C = (X, \iota(X), \tau(X, X'))$ be a symbolic transition system. Then, $\forall X.\phi(X)$ is an invariant for $C$ iff the formula $\phi[X/P]$ is an invariant for $C_P = (X \cup P, \iota(X), \tau(X, X') \wedge P' = P)$, where $P$ is a set of fresh frozen variables, called prophecy variables.*

In order to build the parameter abstraction of the system $S$, we first apply Proposition 31, and add the candidate property itself in conjunction with the transition formula. Then, by applying Proposition 32, we remove the universal quantifiers in the formula $\phi$ and introduce accordingly a set of prophecy variables $P$. In this way, we reduce the problem of checking $S \models \forall I.\phi'(I, X)$ to $S \models \phi'(P, X)$.

### Abstraction Computation

We continue the computation of the abstract system by defining a set of fresh variables, called the environmental variables $E$, in a number determined by the greatest existential quantification depth in the disjuncts of the transition formula of $S$. While the prophecies are frozen variables, the interpretation of the environmental variables is not fixed. Moreover, we assume that the values taken by $P$ and $E$ are different. We now define the formulae for $\tilde{S}$, the parameter abstraction of $S$.

**Initial formula**

Let $\iota(X)$ be the initial formula of $S$, which by hypothesis contains only universal quantifiers. The initial formula of the abstract system is a formula $\tilde{\iota}(P, X)$, obtained by expanding all the universal quantifiers in $\iota$ over the set of prophecy variables $P$.

**Transition formula**

For simplicity, we can assume that we have only one disjunct in $\tau(X, X')$. First, we compute the set of all substitutions of the existentially quantified variables $I$ over $P \cup E$, and we consider the set of formulae $\{\tilde{\tau}_j(P, P', E, X, X')\}$, where $j$ ranges over the substitutions, and $\tilde{\tau}_j$ is the result of applying the substitution to $\tau$.

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

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

$$\tilde{\tau}_S := \bigwedge_{x \in X} \bigwedge_{p \in P} x'[p] = x[p] \wedge p' = p$$

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

$$\tilde{S} := (\{X, P, E\}, \tilde{\iota}(P, X), \tilde{\tau}(P, P', E, X, X')).$$

**Stuttering Simulation**

We state here the property of our version of the Parameter Abstraction, the proof of which can be found in the last section. Formally, the abstraction induces a stuttering simulation, where the stuttering is given by $\tilde{\tau}_S$: this is a weaker version than the simulation induced by [88], yet it is sufficient for preserving invariants. Intuitively, in our abstraction, we require that every abstract array is equal to its concrete counterparts only on the locations referred to by the prophecy variables. We then have the following:

**Proposition 33.** *There exists a stuttering simulation between $S$ and $\tilde{S}$. Moreover, the simulation preserves $\Phi(P, X)$. Therefore, if $\tilde{S} \models \Phi(P, X)$, then $S \models \Phi(P, X)$.*

Unfortunately, if $\tilde{S} \not\models \Phi(P, X)$, we cannot conclude anything, since the counterexample may be spurious, due to a too coarse abstraction. If that is the case, we restart the loop of the algorithm by increasing the size of ground instance $n$, and either find a counterexample in a larger size or consider again the abstraction with a new candidate invariant.

### 3.3.4 Candidate Checking via SMT solving

An alternative – and more standard – approach for performing the Candidate Checking procedure, which does not require particular syntactical restrictions, is to check directly if the formula $\Psi$ is an *invariant strengthening* for $\phi$.

**Definition 18.** *Let $S = (X, \iota(X), \tau(X, X'))$ be an array-based transitions transition system, and $\phi$ a candidate invariant. An invariant strengthening $\Psi$ is a formula such that the following formulae are $A_I^E$-unsatisfiable:*

$$\begin{aligned} \iota(X) \wedge \neg(\phi(X) \wedge \Psi(X)) \\ \tau(X, X') \wedge \phi(X) \wedge \Psi(X) \wedge \neg(\phi(X') \wedge \Psi(X')). \end{aligned} \tag{3.4}$$

Since a formula is valid iff its negation is unsatisfiable, it follows from the definition that if $\Psi$ is an invariant strengthening for $\phi$ iff $\phi \wedge \Psi$ is an inductive invariant for $S$.

Checking the unsatisfiability of the formulae (3.4) could be implemented with the usage of any prover supporting SMT reasoning and quantifiers, e.g. [50, 8]. However, especially for satisfiable instances, such solvers can diverge easily. Thus, since many queries can be SAT, a naive usage of such tools will cause the procedure to get stuck in quantified reasoning with no progress obtained. Therefore, we propose in this section a 'bounded' sub-procedure of Candidate Checking, in which instead of relying on an off-the-shelf SMT solver supporting quantifiers, we 'manually' apply standard instantiation-based techniques for quantified SMT reasoning [51], in which however we carefully manage the set of terms used to instantiate the quantifiers, in order to prevent divergence.

Given a candidate inductive invariant, we perform Skolemization on the inductive query (3.4), obtaining a universal formula. Then, we look for a set of terms $G$ such that the ground formula obtained by instantiating the universals with $G$ is unsatisfiable. This is the standard approach used in SMT solvers for detecting unsatisfiability of quantified formulae [51, 62]; the main difference is that instead of relying on heuristics to perform the instantiation lazily during the SMT search (e.g [51, 62]), we carefully control the quantifier instantiation procedure, and expand the quantifiers eagerly so that we can use only quantifier-free SMT reasoning.

Let $\phi_S = \forall I.\phi'_S(I, X)$ be the result of the Skolemization process, where $\phi'_S$ is a quantifier-free formula over a signature $\Sigma'$, obtained by expanding $\Sigma$ with new Skolem symbols. Initially, we simply let $G$ be the set of 0-ary symbols of the index sort in the formula. Note that apart from constants in the original signature, new (Skolem) constants arise by eliminating existential quantifiers. Since we use only universal quantification for the generalized invariant strengthening, $\Psi$ is a conjunction of universal formulae, and we can swap the conjunction and the universal quantification to obtain a formula with only $n$ universal quantified variables, where $n$ is the size of the last ground instance visited. Notice that, since the candidate inductive strengthening occurs also negated in the quantified formula, this will produce $n$ new Skolem constants.

Finally, we can add to the inductive query an additional constraint. By induction on the structure of our algorithm, if $\Psi$ is generalized from size $n$, we have proven already that the property $\phi$ holds in $S$ for all the ground instances of size equal to or less than $n$. Thus, we impose that in our universe $G$ there are at least $n$ different terms.

To sum up, let $\phi_S = \forall I.\phi'_S(I, X)$ be the universal formula obtained after Skolemizing the formulae in (3.4), and let $m$ be the length of $I$. Let $n$ be the cardinality of the last visited ground instance. Let $G$ be the set of constants of index sort in $\phi'_S$ (by the

previous discussion, $|G| \geq n$). Let $c_1, \ldots, c_n$ be a set of fresh variables of index sort. We test with an SMT solver the satisfiability of the following formula

$$\bigwedge_{\underline{g} \in G^m} \phi'_S[I/\underline{g}] \wedge \mathit{AllDiff}(C) \wedge \bigwedge_{j=1}^{n} (\bigvee_{g \in G} c_j = g) \tag{3.5}$$

We have that:

**Proposition 34.** *For any set of $\Sigma_I$-terms $G$, if (3.5) is unsatisfiable, then $\psi$ is an inductive strengthening for $\phi$.*

### Refinement

If the former formula is SAT, there are two possibilities. Either we have a real counterexample to induction, and we need a better candidate, or our instantiation set $G$ was too small to detect unsatisfiability. In general, if $G$ covers all possible $\Sigma_I$-terms, then we can deduce that the counterexample is not spurious.

**Definition 19.** *Given an index theory $\mathcal{T}_I$ with signature $\Sigma_I$, we say that a set of $\Sigma_I$-terms $G$ is saturated if, for all terms $\Sigma_I$-term $t$, there exists a $g \in G$ such that $\mathcal{T}_I \models t = g$.*

So, if $G$ is saturated, any model of (3.5) corresponds to a counterexample to induction, and we need a better strengthening. However, in case (3.5) is satisfiable, but $G$ is not saturated, we use the following heuristic to decide whether we need a better candidate or a larger $G$. We consider the inductive query in $S^{n+1}$,using as a candidate inductive invariant $(\Psi \wedge \phi)^{n+1}$. If the candidate invariant is still good (the query is UNSAT), we try to increase $G$ to get the unsatisfiability of the unbounded case. Our choice is to add to $G$ terms of the form $f(x)$ where $f$ is a function symbol of index type, and $x$ are constants already in $G$. Note that if no function symbols are available, i.e. if $\Sigma_I$ is a relational signature, then saturation of $G$ follows already by considering 0-ary terms. Therefore, in case $G$ is initially not saturated, the existence of at least one function symbol is guaranteed.

If the query (3.5) is now UNSAT, we have succeeded. Otherwise, we continue to add terms to $G$, until either all function symbols have been used, or an UNSAT result is encountered. If the candidate invariant strengthening is not inductive for size $n+1$ (the query is SAT), then we remove the bad lemmas, and we fail to prove the property.

An important remark is necessary to put more insight on the reasons why our instantiation procedure is effective, especially for the protocols we considered. As already mentioned, in many systems descriptions, especially the ones arising from parameterized verification, the formulae describing array-based systems fall into the decidable fragment of Proposition 11. In this case, no function symbols are introduced during Skolemization: therefore, the set $G$ of 0-ary terms already is saturated. Even in the case of $\forall\exists$ alternation (but in a multi-sorted setting), saturation can be achieved after a few refinement steps (as long as the Skolem functions introduced in the signature do not combine in cycles). More details about the completeness of instantiation methods,

especially for the verification of parameterized systems, can be found in [67, 60, 125]. Since we limit ourselves to terms of depth one, our method can fail to prove invariants requiring some more complex instantiations. Note that in that case, it is always possible to change the choice and the refinement of the set $G$ with more sophisticated methods [116, 62].

### 3.3.5 Properties

In this section, we state the general properties of the algorithm, that hold regardless of the strategies used during the procedure.

**Proposition 35.** *(Soundness) If Algorithm 3 returns* SAFE*, then* $S \models \phi$*. If the algorithm returns* UNSAFE*, then* $S \not\models \phi$*.*

*Proof.* The first claim follows directly from propositions 33 or 34. For the second claim, the algorithm terminates with unsafe only when $S^n \not\models \phi$, meaning that there exists a model $\mathcal{M}$ of $A_I^E$ of cardinality $n$, and a sequence of states $s_0, \ldots, s_k$, such that $\mathcal{M}, s_0 \models \iota$ and $\mathcal{M}, s_i, s_{i+1} \models \tau$ for all $0 \leq i < k$. This also means that $S \not\models \phi$. □

In general, we do not have any theoretical guarantees that our algorithm will terminate. In fact, even the ground Candidate Checking $S_n \models \phi_n \wedge \Psi_n$ can be non-terminating. However, our algorithm guarantees (it follows from how we do the generalization, i.e. Definition 17) that every lemma in $\Psi$ can be removed only after checking an $n'$ instance, with $n' > n$. Therefore, the algorithm always makes progress in the following sense.

**Proposition 36.** *(Progress) During every execution of the loop of Algorithm 3, the same pair* $(n, \Psi)$ *never occurs twice.*

Finally, by checking instances of bigger and bigger size, we semi-decide the problem of falsifying invariant problems:

**Proposition 37.** *(Semi-completeness for counterexamples) Suppose* $S \not\models \phi$*, and the minimal counterexample with respect to the sizes of index models is* $n$*. If all the model checking problems* $S_{n'} \models \phi_{n'}$*, with* $n' < n$*, terminate, then algorithm 3 eventually finds the counterexample.*

### 3.3.6 Proofs of main results

We report here the technical results for the proof of Proposition 33. We consider an array-based transition system $S = (x, \iota(x), \tau(x, x'))$, a candidate property $\phi(x)$, and its parameter abstraction $\tilde{S} = (\{\tilde{x}, P, E\}, \tilde{\iota}(P, \tilde{x}), \tilde{\tau}(P, P', E, \tilde{x}, \tilde{x}'))$, as defined in 3.3.3, where the $\tilde{x}$ are a renaming of the $x$. Note that a state $\tilde{s}$ of $\tilde{S}$ consists of both an assignment of the array variable $x$ and of the index variables $P \cup E$. With a small abuse of notation, we do not distinguish the two cases. The simulation of the abstraction is given by the formula:

**Definition 20.** *Let $S$ an array based transition system and $\tilde{S}$ its parameter abstraction. Let $P = \{p_1, \ldots, p_n\}$ be the set of prophecy variables. We define*

$$\tilde{H}(x, \tilde{x}) := \bigwedge_{i=1,\ldots,n} \tilde{x}[p_i] = x[p_i]$$

*Let $\mathcal{M}$ be a model of $A_I^E$. If $s$ is a state of $S$, and $\tilde{s}$ of $\tilde{S}$, we define a relation $\alpha$ among states in this way:*

$$s, \tilde{s} \in \alpha \Leftrightarrow \mathcal{M}, s, \tilde{s} \models \tilde{H}(x, \tilde{x}).$$

First, we prove the following Proposition.

**Proposition 38.** *Let $\mathcal{M}$ a model of $A_I^E$. Let $\tilde{s}$ be a state of $\tilde{S}$. Let $\mu$ be an interpretation of $P$ such that $\mu(P) = \tilde{s}(P)$. Let $\phi(P, x)$ be a quantifier free formula which contains only prophecies as free index variables. Then, for any state $s$ of $S$ such that $\alpha(s, \tilde{s})$,*

$$\tilde{s} \models \phi(P, x) \Leftrightarrow s, \mu \models \phi(P, x)$$

*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

$$\mu, s \models \phi(P, x)$$

where $s$ is a valuation to into $\mathcal{M}$, is equivalent to

$$\mu, s' \models \phi(P, x), \tag{3.6}$$

where $s'$ is a valuation to $\mathcal{N}$, which 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 $\mu(P)$. Similarly, for any model $\mathcal{M}'$ and valuation $\tilde{s}$ into it,

$$\tilde{s} \models \phi(P, x)$$

is equivalent to

$$\tilde{s}' \models \phi(P, x), \tag{3.7}$$

where $\mathcal{N}'$ defined similarly as above. Since $\mu(P) = \tilde{s}(P)$, we have $\mathcal{N} = \mathcal{N}'$. Moreover, from the definition of $\alpha$, $s'$ and $\tilde{s}'$ assign $x$ to the same function, so (3.6) and (3.7) are equivalent. □

As a corollary, we have We have (see Definition 7)

**Proposition 39.** *The relation $\alpha$ preserves all the formulae of $A_I^E$ of the form $\phi(P, x[p_1], \ldots, x[p_n])$.*

**Lemma 40.** *Let $\mathcal{M}$ be a model of $A_I^E$. If $s \models \iota(x)$, then there exists some $\tilde{s}$ such that $\alpha(s, \tilde{s})$ and $\tilde{s} \models \tilde{\iota}(P, x)$.*

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

$$\tilde{\iota}(P,x) = \bigwedge_{p_{i_1},\ldots,p_{im} \subseteq P^m} \phi(p_{i_1},\ldots,p_{i_m}, x[p_{i_1},\ldots,p_{i_m}]).$$

Let $\tilde{\mu}$ an (injective) assignment of the prophecy variables $P$ into $\mathcal{M}_I$. Let $\tilde{s}$ defined as the restriction of $s$ over $\tilde{\mu}(P)$ and such that $\tilde{s}(P) = \tilde{\mu}(P)$. By definition, $(s,\tilde{s}) \in \alpha$. Then, by Proposition 38, we have

$$\tilde{s} \models \tilde{\iota}(P,x) \Leftrightarrow s, \tilde{\mu} \models \tilde{\iota}(P,x).$$

Since the formula $\iota(x) \to \tilde{\iota}(P,x)$ is valid, and $s \models \iota(a)$ by hypothesis, the claim follows. □

**Lemma 41.** *Let $\mathcal{M}$ be a model of $A_I^E$. If $s, s' \models \tau(x,x')$, then for every $\tilde{s}$ such that $(s,\tilde{s}) \in \alpha$, either:*

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

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

$$s, s' \models \exists i \forall J. \psi(i, J, a, a[J], a', a'[J]).$$

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

$$s, s', \mu \models \forall J. \psi(i, J, a, x[J], x', x'[J]). \tag{3.8}$$

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

- Suppose $\mu(i) = \tilde{s}(p)$. Then, the transition of $\tilde{S}$ labeled by the substitution $i \mapsto p$ is:
$$\tilde{\tau}_{\sigma: i \mapsto p} = \bigwedge_{j \in p,x} \psi(p, J, x, x[J], x', x'[J]).$$
Let $\tilde{s}'$ defined as $\tilde{s}'(p) := \mu(i)$ and $\tilde{s}'(x)[\tilde{s}'(p)] := s'(x)[\mu(i)]$. Note that $\alpha(s',\tilde{s}')$ by definition. Since (3.8) is universal and $\mu(i) = \tilde{s}(p)$, with an argument similar to lem 40, we have that $\tilde{s}, \tilde{s}' \models \tilde{\tau}_{\sigma: i \mapsto p}$.

- Suppose $\mu(i) \neq \tilde{s}(p)$ but $\mu(i) = \tilde{s}(e)$ and $\tilde{s}(x)[\tilde{s}(e)] = s(x)[\mu(i)]$. Then, consider the transition labeled by the substitution $i \mapsto e$. Similarly to the first case, we can define $\tilde{s}'$ to be the restriction of $s'$ over $p$ and $e$, and we have that $\tilde{s}, \tilde{s}' \models \tilde{\tau}_{\sigma: i \mapsto e}$. Moreover, $\alpha(s',\tilde{s}')$ by definition.

- If instead $\mu(i) \neq \tilde{s}(x)$ or $\tilde{s}(x)[\tilde{s}(e)] \neq s(x)[\mu(i)]$, we can reduce to the previous case with a stuttering transition. Let $\tilde{s}'$ defined as $\tilde{s}$ on $p$, but $\tilde{s}'(e) := \mu(i)$ and $\tilde{s}'(x)[\tilde{s}(e)] := s(x)[\mu(i)]$. Note that we also have $(s, \tilde{s}') \in \alpha$. Then $\tilde{s}, \tilde{s}' \models \tilde{\tau}_S$, and we have reduced to the previous case. So, there exists an $\tilde{s}''$ such that $\tilde{s}', \tilde{s}'' \models \tilde{\tau}_{\sigma:i \mapsto e}$ and $\alpha(s', \tilde{s}'')$.

In general, suppose $P = (p_1, \ldots, p_n)$. By hypothesis,

$$s, s' \models \exists I \forall J. \psi(i, J, x, x[J], x', x'[J]).$$

Since the length of $E$ is the maximum length of the existentially quantified index variables in the rules of $S$, we can assume without loss of generality that $I = (i_1, \ldots, i_m)$ and $E = (e_1, \ldots, e_m)$. By hypothesis there exists an interpretation $\mu$ of $I$ such that

$$s, s', \mu \models \forall J. \psi(I, J, x, x[J], x', x'[J]).$$

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

- if $\mu(I) \subseteq \tilde{s}(P)$, then there exist $P_J = (p_{j_1}, \ldots, p_{j_m})$ such that $\mu(I) = \tilde{s}(P_J)$. We can define $\tilde{s}'$ to be the restriction of $s$ over, $P$ and we have again $\tilde{s}, \tilde{s}' \models \tilde{\tau}_{\sigma:I \mapsto P_J}$.

- Suppose now there exists a $0 \leq h < m$ such that $\mu(i_1, \ldots, i_h) = \tilde{s}(p_{j_1}, \ldots, p_{j_h})$, and moreover $\mu(i_{h+1}, \ldots, i_m) = \tilde{s}(e_1, \ldots, e_{m-h})$, and also $\tilde{s}(a)[\tilde{s}(e_1, \ldots, e_{m-h})] = s(a)[\mu(i_{h+1}, \ldots, i_m)]$. Then, if we define $\tilde{s}'$ to be the restriction of $s'$ over $P \cup E$, we have that $\tilde{s}, \tilde{s}' \models \tilde{\tau}_\sigma$ where $\sigma : I \mapsto \{p_{j_1}, \ldots, p_{j_h}, e_1, \ldots, e_{m-h}\}$.

- If instead $\mu(i_{h+1}, \ldots, i_m) \neq \tilde{s}(e_1, \ldots, e_{m-h})$ or $\tilde{s}(a)[\tilde{s}(e_1, \ldots, e_{m-h})] \neq s(a)[\mu(i_{h+1}, \ldots, i_{m-h})]$, we can reduce to the previous case with a stuttering transition. Let $\tilde{s}'$ defined as $\tilde{s}$ on $P$, but $\tilde{s}'(e_1, \ldots, e_{m-h}) := \mu(i_{h+1}, \ldots, i_m)$ and $\tilde{s}'(a)[\tilde{s}(e_1, \ldots, e_{m-h})] := s(a)[\mu(i_{h+1}, \ldots, i_{m-h})]$. Note that $\alpha(s, \tilde{s}')$ by definition. Moreover, $\tilde{s}, \tilde{s}' \models \tau_S$. We have now reduced to the previous case, and the claim follows.

□

**Theorem 42.** *The relation $\alpha$ is a stuttering simulation between $S$ and $\tilde{S}$.*

*Proof.* Follows directly from Lemmas 40 and 41. □

**Theorem 43.** *Let $S$ be an array-based transition system, $\tilde{S}$ its parameter abstraction. Let $\forall I. \Phi(I, a)$ a candidate invariant, and $P$ a set of frozen variables with same length as $I$. If $\tilde{S} \models \Phi(P, a)$, then $S \models \Phi(P, a)$*

*Proof.* The statement follows from the fact that stutter simulations preserve reachability (6), and from Proposition 38.

□

## 3.4 Related Work

This section aims to provide an overview of the state-of-the-art in parameterized system verification, highlighting the advancements and limitations of existing techniques. Verification of systems with quantifiers ranging over finite but unbounded domains has always received a lot of attention from the literature, particularly in the applications in the field of parameterized verification.

Various techniques have been developed based on *cut-off* results, where a cut-off is the size of a ground instance that already contains all possible behaviors. Cut-off values exist for large varieties of classes of systems but depend strongly on assumptions such as topology and data [14]. Once a cut-off is obtained for a particular class, one can reduce parameterized model checking to model checking only a finite number of ground instances. However, such results are obtained only on specific systems, for example, token-passing systems in ring structures or broadcast in rendezvous protocols [7, 79, 54, 55]. Nonetheless, such results are quite useful, since they can also be used for the verification of liveness properties. Our methodology aims to be more general and deductive, contrasting with these approaches primarily reliant on specific systems and properties for cut-off determination.

The technique of invisible invariants, proposed in [114, 56, 130], uses ground exploration to generate candidate universally quantified invariants, marking the first instance of such an approach for Boolean systems. Such an invariant is obtained with the formula of reachable states computed from a ground instance. In the original paper, the method was proved also complete for a subclass of systems (effectively providing a cut-off result). Despite the advancements it introduced, its applicability is limited to a subclass of the systems we consider. Nonetheless, the concept of generalizing invariants from ground instances is foundational also to our more general algorithms, demonstrating the significance of cut-off results and invisible invariants in our work indirectly.

Abstraction methods, such as those on Parameter Abstraction [88, 29], or Environmental Abstraction [44, 120], have shown success also in verifying some industrial protocols [124]. The general idea is to replace the verification of the parameterized system with an abstract one that encapsulates all the possible behaviors of the components. However, these methods often require significant manual effort and expertise to achieve desired outcomes. Our abstraction algorithm in Section 3.3.3 draws inspiration from [102] and [88], seeking to enhance automation and reduce the need for expert intervention.

Tools like Mcmt [67] and Cubicle [45] are model checkers designed to solve the invariant problem of array-based transition systems. The formalism used by the framework [65] is a subclass of the one we consider. Both tools implement fully symbolic backward reachability algorithms but they need strong syntactic restrictions on the shape of the formulae defining systems. Such restriction allows for their procedure to produce satisfiability checks that always falls into the decidable fragment 2.3. If this is not possible, the procedure introduces approximations that may lead to spurious counterexamples. The approaches that we propose do not rely on backward computation of pre-images, aiming for direct and potentially more accurate verification methods.

The tool SAFARI [3] is based on the MCMT formalism and is used to prove properties of systems defined over the theory of arrays, by leveraging an ad-hoc interpolation procedure.

Ivy [112, 60] is a tool for the verification and design of parameterized systems. It aims to be a bridge between automated theorem proving and verification. The tool itself is not capable of inferring inductive invariants, but it guides the use of writing correct ones by showing counterexamples to induction. The formalism used by Ivy can be embedded in the one of this thesis, simply by considering as $\mathcal{T}_E$ the theory of booleans. A related tool is MYPYVY [83], which is a model checker based on the Ivy formalism, that implements algorithms for the automatic discovery of inductive invariants like UPDR [81]. Such algorithms represent important ground for the work described in this thesis. In particular, our work extends these methodologies to accommodate a general theory $\mathcal{T}_E$, showcasing the breadth of our approach in handling more complex system models.

Other tools and methods that leverage finite instance exploration for invariant generalization are [70, 95, 129, 76, 93], often domain-specific and relying on external provers for quantified queries. Our algorithm in Section 3.3 shares similarities with IC3PO [70] but aims for broader applicability beyond pure first-order logic contexts.

The SMT-based approach for parametric verification in [74] presents a reduction to non-linear Constrained Horn Clauses (CHCs) but is more restrictive in its input language and syntactic structure of invariants. Our methodology contrasts by offering a more flexible input language and a broader approach to invariant structuring.

Moving outside the realm of parameterized verification, there are other verification techniques capable of synthesizing invariants with universally quantified variables. For example, the use of prophecy variables in [96, 128] introduces a technique to infer universally quantified invariants for quantifier-free transition systems over the theory of arrays. The algorithm [75] introduces an IC3 variant for the same kind of systems, capable of inferring invariants with quantifiers, thanks to the techniques of approximated quantifier elimination of [84].

# Chapter 4

# Compositional Verification of Parameterized Systems

In this chapter, we present a novel algorithmic approach that represents a departure from the methods discussed in earlier chapters. Our focus shifts to the invariant verification of parameterized systems through the lens of asynchronous composition of diverse subsystems. Actually, the algorithm we discuss is not limited to array-based symbolic transition systems but is applicable to a broader range of system types. Initially, in Section 4, we introduce the algorithm in a general context, not targeting only parameterized systems. Subsequently, we will refine our discussion to specifically address its application to array-based transition systems.

The context for our algorithm involves systems composed of multiple components, each designed independently yet interconnected through shared variables. These components operate asynchronously, with each capable of altering the shared variables through transitions. Our aim is to devise a method for automatically verifying invariant properties across the entire system, focusing on reducing the computational overhead typically associated with model checking the entire system monolithically.

Understanding that direct application of model checking to such a comprehensive system description can be computationally prohibitive, we propose an alternative strategy. This strategy is predicated on the insight that, in many cases, an inductive invariant for the entire system can be discerned by examining a smaller composition of components. Crucially, this examination only needs to account for information deemed 'relevant' to the invariant being verified. The primary challenge, therefore, is to identify which components are relevant and how they should be abstracted for analysis.

Our proposed solution seeks to identify specific subsets of components for which, by conducting model checking on an abstracted composition, we might uncover an inductive invariant. We hope this invariant will also hold true for the complete system composition. This crucial verification step can be facilitated through a single automated call to a theorem prover. Although this step remains computationally demanding, it is significantly less so than the full model checking process.

This approach can produce spurious counterexamples or even false proofs due to the abstraction process. To mitigate this, we employ a CEGAR framework. Through this

iterative process, the abstraction can be refined to either enhance the existing model or to incorporate new components for abstraction, thereby gradually converging on a more accurate verification outcome.

Building on the foundational principles outlined, Section 4.2 delves deeper into the application of our algorithm within the context of a specific family of parameterized systems, notably those capable of modelling railway interlocking logic. This domain, as referenced with [26], is inherently complex, characterized by the asynchronous interaction of myriad components and a substantial variable set. Such systems are emblematic of the challenges faced when ensuring safety in highly dynamic and distributed environments.

In this vein, our algorithm is not only designed to address these challenges but also structured to incorporate, as subprocedures, the algorithms discussed in the preceding chapter. By leveraging these subprocedures, we aim to enhance the algorithm's efficiency and efficacy in abstracting non-essential variables and focusing on key components. This methodology underscores our commitment to advancing verification practices for complex, parameterized systems, built as a composition of several (parameterized) sub-systems.

## 4.1 An algorithm for the Verification of Asynchronous Composition of Symbolic Transition Systems

In this section, we outline a procedural framework that remains parametric, considering a generic family of transition systems, a generic abstraction procedure, and a property to prove denoted as $F$. In the next section, we delve into a case study where we provide a more concrete setting.

Formally, suppose that we have a finite family of transition systems $\{C_i\}_{i\in I}$. Let $C = \|_{i\in I}\ C_i$ be the asynchronous composition of the systems, and consider a formula $F(V)$ with $V \subseteq \bigcup_{i\in I} X_i$. The problem that we face is to prove of disprove whether $C \models F$.

Recall that the problem is solved if either we find a counterexample, i.e. a path $\pi$ of finite length $n$, such that $\pi[n] \models F$, or if we find an inductive invariant $\Psi$ for $F$. If $F$ is not inductive itself, then by Proposition 8 that there exists a subset $J$ of $I$ such that $F$ is not inductive for $\|_{j\in J}\ C_j$. To describe the whole algorithm, we suppose to have some sub-procedures, namely:

- a model checker, capable of automatically proving if an invariant holds in a transition system. If so, the model checker provides an inductive invariant for it. Otherwise, the model checker find a counterexample;
- a theorem prover, capable of checking whether a formula is inductive for a transition system, or if a counterexample can be simulated (e.g by bounded model checking).

Moreover, let $\tilde{C}$ be a transition system such that there exists a simulation $\|_{j\in J}\ C_j \rightarrow$

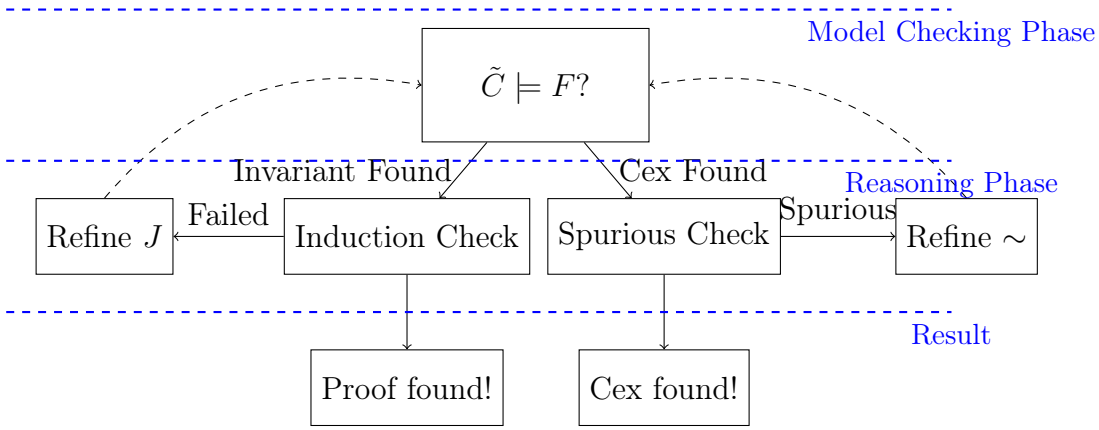


Figure 4.1: The procedure

$\tilde{C}$. Such a simulation should be such that it preserves all inductive invariants found by the model checker.

We consider the following procedure, depicted in Figure 4.1:

- we start by asking a model checker if $\tilde{C} \models F$. The model checker can either find an inductive invariant, $\Psi$, or a counterexample, $\pi$;
- If an invariant is found, we ask the prover to check if $\Psi$ is also inductive for the whole asynchronous composition $C$. Note that, since the simulation preserves $\Psi$, we already know that it is inductive for the components $\{C_j\}_{j \in J}$ by Proposition 6.
- If the prover proves the induction, then we are done. Otherwise, there must exists a new set of components $J' \subseteq I \setminus J$ for which the induction check fails. We thus update the set $J$ to be equal to $J \cup J'$, and we restart the loop by updating the abstraction $\tilde{C}$.
- Suppose instead that the model checker finds a counterexample in $\tilde{C}$. Then, we ask the prover if the counterexample can be simulated by $C$. If so, the algorithm terminates with a counterexample. Otherwise, we refine the abstraction to remove the abstract counterexample.

The key differences of our approach from conventional CEGAR methods in compositional verification lies in the fact that the system $\tilde{C}$ doesn't abstract the entire composition $C = \|_{i \in I} C_i \to \tilde{C}$; instead, it abstracts only the 'sub'-composition $\|_{j \in J} C_j \to \tilde{C}$. We could eventually abstract all components, when $J = I$, and $C \to \tilde{C}$ is a simulation. However, this scenario isn't the primary use-case for our procedure. Ideally, our method is best suited for situations where we aim to verify a property relevant only to a specific subset of components.

Even when $\tilde{C}$ doesn't represent the entire system, the algorithm's soundness should be clear. This is because we conduct an additional induction check to verify whether the

invariant identified during model checking is also inductive for the broader composition $C$.

Addressing termination, various challenges may arise; in fact, model checking itself might be non-terminating in case of infinite-state systems. Moreover, the refinement process may be incomplete; finally, in case of undecidable theories, also the induction check can be undecidable. However, there are various approaches that, although incomplete, are proved to be practical efficient for this problems.

## 4.2 Verification of concurrent parameterized systems

In this section, we illustrate the application of the procedure outlined in Section 4 through a specific use case.

As already mentioned, this algorithm was conceived to facilitate the verification of interlocking logic within railway stations, as part of a larger initiative to integrate various formal methods into railway design [26], described also in Chapter 5. In this scenario, the system's components are each represented by transition systems. These components interact by sharing multiple variables, allowing them to transmit commands among themselves. Our objective is to ascertain the general safety of the composition of the systems. Despite the presence of a vast number of components, only a subset is critical for ensuring safety. This distinction underscores the potential utility of the algorithm described in the preceding section. Furthermore, while the systems may involve a large number of variables, we recognize that only a limited number are pertinent to safety concerns. Consequently, our approach to abstraction focuses on minimizing the complexity by narrowing down to these essential variables.

The symbolic transition systems that we consider are a subclass of the array-based transition systems. In particular, we ask that the formulae defining those systems have a precise syntactic shape; we will use these constraints to better describe how to compute the abstraction algorithm that we will use. We want to remark that these syntactic conditions do not impose a particular burden in modeling; rather, they are a natural way to encode parameterized transition systems, especially those arising from the description of interlocking logic.

**Definition 21.** *A functional array-based transition system $S = (X, I(X), T(X, X'))$ is an array-based transition system such that:*

- *$X$ are a set of array symbols;*
- *$I(X)$ is a $\Sigma(X)$ formula of the form $\forall j. \bigwedge_{x \in Y} x(j) = val_x$ - where $val_x$ is a constant of the appropriate signature, and $Y \subseteq X$;*
- *$T(X, X')$ is a $\Sigma(X, X')$ formula that is a disjunction of formulae of the form (called transition rules)*

$$\exists i.(\phi_G(i, X) \wedge \ \phi_U(i, X, X')) \tag{4.1}$$

*where $\phi_G(i, X)$ is called the guard, and $\phi_U(i, X, X')$ is the functional update, i.e. a formula of the form*

$$\forall j. \bigwedge_{x \in X} x'(j) = F_x(i, j, X, X')$$

*with $\{F_x\}_{x \in X}$ a family of case-defined function.*

In the remainder of this section, when we refer to an array-based transition system, we will always implicitly consider only *functional* ones.

**Example 11** (Interlocking Logic)**.** As an example, we consider two transition systems (modelling two classes of the interlocking logics). The first represents the class of tracks, while the second represents the class of routes. Therefore, for $C_1$ the index theory is an uninterpreted sort named *track*, while for $C_2$ will be named *route*. For the element theories, we use a combination of an enumerative datatype with values $\{locked, free\}$, another enumerative datatype with values $\{idle, wait, active\}$, and the booleans. The state variables for $C_1$ are the array symbols $state\_t : track \mapsto \{locked, free\}$ and $unlock\_command : track \mapsto Bool$. The initial formula is

$$I_1 \equiv \forall j : \text{track}\big(\text{state\_t}[j] = \text{free} \wedge \text{unlock\_command}[j] = \text{false}\big).$$

As an example for a transition rule of the first class, we have the following formula, modelling the unlocking of a track segment

$$T_1 \equiv \exists i : \text{track}\Big( \underbrace{\big(\text{state\_t}[i] = \text{locked} \wedge \text{unlock\_command}[i] = \text{true}\big)}_{\text{guard}} \wedge \underbrace{\big(\forall j.\text{state\_t}'[j] = F_1 \wedge \text{lock\_command}'[j] = F_2\big)}_{\text{update}} \Big),$$

where $F_1$ is given by $\big((i = j), free\big)$, $\big((i \neq j), state\_t[j]\big)$; $F_2$ is given by $\big((i = j), false\big)$, $\big((i \neq j), lock\_command[j]\big)$.

The state variables for $C_2$ are an array $state\_r : route \mapsto \{idle, wait, active\}$, a frozen array predicate $tracks\_used : track \times routes \mapsto Bool$, and the shared variables $unlock\_command$ and $state\_t$. The initial formula is

$$\begin{aligned} I_2 \equiv \forall j_0 : \text{route}, j_1 : \text{track}.\big(&\text{unlock\_command}[j_1] = \text{false} \wedge \\ &\text{state\_t}[j_0] = \text{idle} \wedge \\ &\text{state\_i}[j_0] = \text{Idle}\big). \end{aligned}$$

and, as an example for a transition rule for $C_2$, we have the following formula, modelling the activation of a route

$$T_2 \equiv \exists i\colon \text{route}\big(\text{state\_r}[i] = \text{wait} \wedge$$
$$(\forall t\colon \text{track}\,(\text{tracks\_used}(t,i) \rightarrow (\text{state\_t}[t] = \text{free} \wedge \text{unlock\_command}[t] = \text{false})))\wedge$$
$$(\forall j0\colon \text{route}, j1\colon \text{track}\,(\text{state\_r}'[j0] = F_1\wedge$$
$$\text{state\_t}'[j1] = F_2 \wedge \text{unlock\_command}'[j1] = F_3)\big)$$

where $F_1$ is given by $(i = j0, \mathit{active}), (i \neq j0, \mathit{state\_r}[j0])$, $F_2$ is given by $(\mathit{tracks\_used}(i, j1), \mathit{locked})$, $(\neg \mathit{tracks\_used}(i, j1), \mathit{state\_t}[j1])$, and $F_3$ is given by $(\mathit{true}, \mathit{lock\_command}[j1])$.

A property of interest is that routes that share a track are never active in the same moment. Let NotCompatible($i$ : route, $j$ : route) be a predicate which holds if and only if

$$\big(i \neq j \wedge \exists t : \text{track}\,(\text{tracks\_used}(i,t) \wedge \text{tracks\_used}(j,t))\big),$$

and let

$$F = \forall r0 : \text{route}, r1 : \text{route}\big(\text{NotCompatible}(r0, r1) \rightarrow$$
$$\neg(\text{state\_r}[r0] = \text{active} \wedge \text{state\_r}[r1] = \text{active})\big).$$

We are interested to prove that $C_1 \parallel C_2 \models F$. □

We start by defining the simulation relation that we will use.

**Definition 22.** *Let $V \subseteq X$ a set of variables. Given $\mathcal{M}$ a model of $A_I^E$, we define a relation $\alpha$ between an assignment $s, \tilde{s}$ of $X$ and $V$ into elements of the domain of $\mathcal{M}$ by*

$$\alpha(s, \tilde{s}) \Leftrightarrow s|_V \equiv \tilde{s}|_V,$$

*i.e. we ask that the two states are in relation iff they assign the same value to the variables in $V$.*

Given a (sub)set of variables $V$ and a transition system $C$ defined as in Definition 21, we now define a new transition system $\tilde{C}^V$ such that there exists a simulation between the two system. The new variables will be $V \cup B \cup E$, with $B$ a set of new boolean variables and $E$ a set of new element variables. The abstract initial formula of the system, denoted $\tilde{I}(V)$, is simply obtained from $I$ by dropping the conjuncts that are not assigning variables in $V$; that is, we have that

$$\tilde{I}(V) = \bigwedge_{v \in V} \forall j.v(j) = val_v.$$

For the abstract transition formula, denoted as $\tilde{T}(V, B, E, V')$, we need more steps. We will work on the single transition rules of the concrete transition, that are of the form (4.1). The abstract transition will be a disjunction of formulae of the form $\exists i.(\tilde{\phi_G}(i, V) \wedge \tilde{\phi_U}(i, V, B, E, V'))$ where

- $\tilde{\phi_G}(i, V)$ is obtained by $\phi_G$ by replacing each atom that contains variables in $X \setminus V$ with the constant $true$, if it occurred positively in the formula, or $false$ otherwise;
- $\tilde{\phi_U}(i, V, B, E, V')$ is the formula $\bigwedge_{v \in V} \forall j. v'(j) = \tilde{F_v}(i, j, V, B, E, V')$ where $\tilde{F_v}$ is a case-define function with a sequence of $\tilde{case_i}(V, B)$ statements and a sequence of corresponding terms $\tilde{val_i}(V, B, E)$ such that:
  - $\tilde{case_i}(V, B)$ is either equal to $case_i(V)$ if the original case predicate is defined only over the $V$ variables, or is a fresh boolean constant $b \in B$ otherwise;
  - $\tilde{val_i}(V, B, E)$ is either equal to $val_i(V)$ if the original term is defined only over $V$, or is a fresh element constant $e \in E$ otherwise.

Finally, let $\tilde{C}^V = (\{V, B, E\}, \tilde{I}(V), \tilde{T}(V, B, E, V'))$. We have:

**Proposition 44.** *$\tilde{C}^V$ simulates $C$.*

*Proof.* (sketch) From $\models I(X) \rightarrow \tilde{I}(V)$, since the consequent only contains fewer constraints, it follows that condition $i.$ of Definition 5 holds. For condition $ii.$, suppose that $s, s' \models T$, and let $\tilde{s}$ such that $\alpha(s, \tilde{s})$. Note that, since $\tilde{s} \models \tilde{\phi_G}$ - the abstract guard, by definition of the simulation relation and by construction of the rewriting process. We need to find an abstract state $\tilde{s}'$ such that $\alpha(s', \tilde{s}')$ and $\tilde{s}, \tilde{s}' \models \tilde{T}$. Such a state is defined according to which values are assigned to the variables $V$ in the update $\phi_U$. If only cases and values with variables in $V$ are used, then $\tilde{s} = s|_V$ suffices. Otherwise, $\tilde{s}$ assigns to the variables in $B$ and $E$ values according to $s$, and the claim follows. □

Moreover, since the simulation relation is the equality on $V$, then counter-models of formulae defined only over $V$ are preserved. Thus (see Definition 7), we have that:

**Proposition 45.** *The simulation preserves each formula $F(V)$ defined only over the set of variables $V$.*

Thus, from Corollary 10, we have:

**Corollary 46.** *$\| C_i \rightarrow \| \tilde{C}_i$ via the product simulation. Moreover, the product simulation preserves all the formulas defined over $V$.*

### 4.2.1 Refinement

Suppose that, during the model checking phase, we found an abstract counterexample $\pi$ leading to a violation of the property. Such a counterexample can be viewed as a model for the formula

$$\tilde{I}^0 \wedge \tilde{T}^0 \wedge ... \wedge \tilde{T}^n \wedge \neg F^{n+1}. \tag{4.2}$$

To check for spuriousness, the standard approach would require to check the satisfiability of the concrete unrolling

$$I^0 \wedge T^0 \wedge ... \wedge T^n \wedge \neg F^{n+1}. \tag{4.3}$$

In case of satisfiability, we are in the presence of a real counterexample, and we can exit from the procedure. In case of unsatisfiability, we need to refine the abstraction to eliminate the counterexample. We describe here our procedure for refinement. We start by computing an unsat core of the latter formula. Then, let $V'$ be the set of variables that occur in at least one literal of the core. We update $V$ to be $V \cup V'$. We have the following result, that ensures that $V' \cap V$ is never empty:

**Proposition 47.** *In case (4.2) is satisfiabile and (4.3) is not, then there exists a literal in the unsat core of (4.3) that contains a variable not occurring in $V$.*

*Proof.* (sketch) Since the $C_i$ are compatible and the simulation relation is the identity on $\alpha$, if (4.2) is satisfiable but (4.3) is not, this means that there is no way of extending an assignment on the $V$ variables to an assignment on the $X$ variables that satisfies (4.3). □

This result allows us to have a notion of progress since at each spurious counterexample we decrease the number of variables not abstracted. Moreover, note that by adding new variables, the abstraction process will introduce more constraints to the abstract system. Thanks to this, we can see that $\tilde{C}^{V \cup V'}$ refines $\tilde{C}^{V}$.

In practice, checking unsatisfiability and extracting unsat cores from formulae such as (4.3) can be done by tools such as Z3 [50]. However, such queries can become very hard to reason about if the length of the unrolling becomes very large. This may also be because of the presence of quantifiers in the transition formula. Actually, it is possible to reduce the check to a quantifier-free one. Suppose that the counterexample $\pi$ is given by an assignment of (4.2) in an index model $\mathcal{M}_I$ of finite cardinality. Then, we can check the unsatisfiability of the formula (4.3) with the additional assumption that all quantifiers ranges only over that finite set, and Proposition 47 still holds.

## 4.3 Related Work on compositional verification

The results we use by mixing abstraction and asynchrnous composition are inspired by Sifakis et al.'s seminal paper [94], which explores the theoretical aspects of compositional verification. It introduces concepts such as simulations, parallel composition, and the computation of an abstraction for a composition of systems through the composition of abstraction on singular components. In this section, we build on these ideas but within an SMT and infinite-state framework. The papers [42] and [27] introduce Counterexample-Guided Abstraction Refinement (CEGAR) and a form of localization reduction, similar to the abstraction we use (although adapted in a parameterized setting) for our case study.

Alternative approaches to compositional verification are automated assumption guarantee methods, such as those found in [73], [63], and [121]. In these works, components of the system make assumptions about their environment and provide guarantees about their behavior. Subsequently, some associated verification conditions are checked to verify global properties by ensuring compatibility between assumptions and guarantees. Additionally, the paper [58] introduces an approach to mixing invariants from

both data flow graphs and control, specifically applied to parameterized concurrent programs such as Linux device drivers. Furthermore, the chapter [68] provides a comprehensive overview of general ideas surrounding compositional reasoning.

A different symbolic approach is presented in the work [19], which leverages WSkS logic to reason about parameterized component-based systems, and [18], where a description logic is employed to reason about systems with a re-configurable network.

The distinctive feature of this algorithm is that our abstraction procedure over-approximates only a subset of the components. The result is that the abstraction simulates only part of the whole system; however, we check with an automated prover whether the property found by analyzing such an incomplete abstraction is correct for the whole system. If the procedure terminates by abstracting only a subset of the components, then we can determine, *a posteriori*, a split invariant [68] between the abstracted and non-abstracted components.

# Part III

# Case Studies and Experimental Evaluation

# Chapter 5

# Application of parameterized model checking to the verification of interlocking logics

This section draws upon the work [36] and describes some parts of a large project aimed at applying the algorithms developed in this thesis. Currently, the project is still in progress.

Interlocking systems are complex, safety-critical systems controlling the operation of the devices in a railway station. The main function is the creation of safe routes for trains from different points in the station. This requires for example that the devices insisting upon a given route (e.g. semaphores, switches, level crossing) must be properly operated and that mutual exclusion between interfering routes is ensured. At a high level of abstraction, an interlocking system can be thought of as implementing a very articulated protocol, that we refer to as interlocking logic.

In this section, we describe how we intend to use a formal approach to ensure the reliability and integrity of their operations, as a complement to the standard validation and certification techniques. The context is an ongoing activity between our research group and RFI (the company managing the Italian railways network), aiming to develop an in-house, framework to design interlocking logics and support the development of interlocking systems [6, 26].

Starting from a high-level controlled natural language to describe the interlocking logic, a model-based Integrated Development Environment supports the railways signaling engineers in specifying the interlocking logic in a well-structured and semantically unambiguous way. Interestingly, the interlocking logic is *generic* in that it describes the procedures without specific reference to a single, given station; rather, it applies to any station in a given class. Therefore, at this level each component of the logic can be seen as a parameterized system. The resulting specification is then translated into a SysML model, and from there compiled into executable code (C and Python). Then, the user can define a specific station configuration (e.g. the station of the city of Trento), detailing the exact number of components and their interactions. Upon configuration, the code can be tested in a closed loop integration with a simulator mod-

eling the behavior of trains and physical devices. Formally verifying the interlocking logic is a very important goal, and several attempts have been made in this direction [57, 61]. In our context, attempts were made to apply (non-parameterized) software model checking techniques on the code configured with respect to a specific station. This approach hit a scalability barrier, due to the sheer size of the resulting, instantiated model. Even more importantly, the results of the verification would be applicable to a specific station only. Given these challenges, our research pivots towards applying parameterized verification to interlocking logic. This approach aims to overcome the limitations of station-specific verification by generalizing the verification process to accommodate any possible station within a defined category. Furthermore, we believe that adopting a parameterized verification strategy may offer enhanced manageability compared to the complexity encountered in models instantiated with a multitude of components.

In this context, the starting point is the integration of Dafny [82] alongside C and Python during the code generation phase. Dafny, an object-oriented programming language with formal verification capabilities, enriches our framework by providing a high-level, intuitive platform for the verification of generic interlocking logics. Its object-oriented nature simplifies the modeling of complex systems and enhances the verification process's comprehensiveness and accuracy.

However, a notable limitation within Dafny's ecosystem is its current inability to autonomously generate inductive invariants, which are crucial for establishing the correctness of complex systems through formal verification.

To address this gap, we intend to leverage the algorithms discussed in the previous chapter. These algorithms are designed to generate the necessary inductive invariants, thereby furnishing Dafny with the tools to complete the proofs effectively and in an autonomous way. The integration of these algorithms into our verification process represents a novel approach to enhancing the capability of formal verification tools, ensuring that our framework can not only handle the complexity of parameterized systems but also maintain the rigor required for safety-critical applications like railway interlocking systems.

In the subsequent sections, we will outline the framework of this research project for incorporating these inductive invariants into the verification process. This exploration is part of an ongoing research effort, aiming to bridge the gap between theoretical algorithms and practical applications in formal verification. Through this endeavor, we seek to demonstrate the feasibility and effectiveness of our approach in real-world settings.

## 5.1 Current framework for developing interlocking logics

We start with an overview of the approach for the development of a generic interlocking logic [6, 26, 5]. The development of the interlocking logic is model-based and starts from a domain-specific controlled natural language (CNL), supported by the tool AIDA [26].

The interlocking logic is designed through the creation of *sheets*, each defining a logical entity, also referred to as a class. Examples of entities include shunting routes, i.e. the high-level process devoted to creating a safe path for a train within the station, and lower level entities such as track segments, switches, level crossing, axal counters, and semaphores. Each sheet is divided into two parts. The first one defines the structure of the class, i.e. variables, parameters, and notably lists of other components that are connected to the class. For example, a shunting route will have lists of the entities of track segments it insists upon, e.g. the track segments that must be locked before the green light is signaled.

The second part of the sheet describes the behavior of a single component (an instance of the class), which can be thought of as an extended Finite State Machine. A distinguished state variable, taking values from an enumerative set, is used to define the current location in the FSM. Each state transition is characterized by the following elements:

- Source and Destination;
- Guards, i.e. the conditions that must be satisfied to enable a particular transition. Determinism is ensured by explicitly prioritizing guards;
- Effects: when a transition is executed, effects are applied that alter the internal state of the component, and possibly the state of components that are connected to it.

Guards and effects of a transition may read and or write the value of variables of the objects in the list of the connected components connected to the class.

The structured natural language used in the sheets has been designed to be comprehensible even to those not trained in formal languages and incorporates grammatical structures drawn from domain-specific jargon. Phrases are structured to ensure traceability to pertinent provisions and regulations. For instance, an engineer might specify a transition guard as follows: "Check that all the track segments of the routes are in a free state."

In AIDA, the sheets written in controlled natural language are associated with a number of syntactic and semantic checks and are translated into a SysML model. From the SysML model, it is possible to extract graphical views of the FSM for each class.

Within this comprehensive IDE, preliminary experiments in applying formal verification have been attempted, aiming at proving safety properties of the generated C code. The tool leverages symbolic model-checking techniques for software verification, with integration into the Kratos2 model checker [72]. This approach, however, did not yield the expected results. On the one hand, even if the generation of C code is generic to all configurations, its subsequent verification can only take place once the configuration of a specific station has been provided. This is necessary to meet the limitations of Kratos2, that is unable to find invariants for objects of unspecified size. As a result, our current capability allows us to assess the safety of individual stations with regard to specific properties. Furthermore, the verification of the C implementation of the

interlocking logic configured for a given station incurs scalability problems, due to the large number of components and their complex connections.

For this goal, we could use the generic generated code as an input for Kratos2, but the invariant generation engine of the model checker is not able to synthesize the correct parameterized invariants. Hence, in the rest of this section, we discuss the parameterized verification of the *generic* interlocking logic, without assuming that a specific station configuration is given.

## 5.2 Dafny Encoding

In this subsection, we delve into the encoding process using Dafny [82], a state-of-the-art programming language renowned for its formal verification capabilities. Dafny's object-oriented nature is particularly suited to the complex structure of interlocking systems, offering a robust framework for ensuring their integrity and reliability. At the core of Dafny's verification strategy is the use of verification conditions, grounded in the principles of Hoare logic.

Moreover, Dafny equips developers with a comprehensive toolbox for formal verification, inherently supporting compositional verification methodologies. Compositional verification in Dafny allows developers to verify parts of a program in isolation and then combine these verifications to ascertain the correctness of the entire system. This approach is particularly beneficial for managing the complexity of large systems by breaking them into smaller, more manageable components, each verified separately but within the context of the whole system.

These verification conditions are critical logical assertions that must be valid for a program to be deemed correct in relation to its defined preconditions and postconditions. Dafny automates the generation of these conditions and employs theorem provers for their verification. A notable challenge within Dafny's framework is its reliance on manually defined invariants for loops and recursive methods; the fact that Dafny cannot generate auxiliary invariants is a recognized limitation. Therefore, failure to prove a verification condition often signifies the need for stronger invariants, rather than direct evidence of a counterexample in the code. This aspect underscores the potential benefits of integrating algorithms for automatic invariant inference, such as those developed in this thesis, to enhance Dafny's verification capabilities.

Through the combination of Dafny and a parameterized model checker, we aim to bring a high level of rigor and precision to the verification of interlocking systems, ensuring their safety and reliability in operational scenarios. This section will provide insights into how Dafny's unique features facilitate a natural and effective encoding of interlocking logics, highlighting the language's modular verification approach. This modularity is key to managing the complexity of verifying individual components within a system, making Dafny an invaluable tool in our formal verification toolkit.

During the course of the project, we studied how to automatically generate Dafny code from SysML diagrams (and thus from the AIDA sheets). While this thesis does not explore the intricacies of translating SysML into Dafny, this section provides illustrative examples of the Dafny encoding process and discusses how parameterized model

checking techniques can be integrated with it.

We take as an example a simple station with only two components: routes and tracks. Each component in the Dafny model is equipped with an ‘initialize‘ method, which sets it to an initial state, and an ‘execute‘ method, which mirrors the firing of an appropriate transition. This modeling approach reflects the operational semantics of the interlocking logic, where each component's state transition is critical for the safety and efficiency of the railway system.

The track file in Dafny, shown in Figure 5.1, defines the track component of our simple railway system model. This module encapsulates the properties and behaviors associated with a single track segment, such as its occupancy status and the ability to lock or unlock the track for train movement. Each track is modeled as an object with states indicating whether it is free or occupied. The module includes methods to change these states in response to the train's movements, ensuring that the track's current state is accurately reflected.

The route Dafny file can be seen in Figures 5.2 and 5.3, and it is designed to model the route components within the railway interlocking system. Routes are entities composed of multiple track segments and other infrastructure elements like switches and signals. In this simple example, we only consider a set of associated tracks. This file specifies the logic for activating and deactivating routes, incorporating checks to ensure that all components of a route are in the correct state before a route can be activated, such as verifying that all track segments are free.

By generating these files, the verification checks performed by Dafny are focused on ensuring that the body of the methods actually satisfies the specified requirements. The invariants required for this verification are primarily found in the while loop of the ‘activate Route‘ method. We assume that these invariants can be generated automatically, via some schemata, without the need for an external tool, as they closely reflect the structure of the generated code, which is under our control.

Formally proving that the generated code satisfies the specifications set forth by railway engineers constitutes an initial goal of our project. However, this objective does not mark the culmination of our efforts nor does it necessitate the use of parameterized model checking.

To achieve the verification of generic properties of the logic, we present the concluding segment of the Dafny code, as depicted in Figures 5.4 and 5.5. Within this last code segment, we introduce the concept of a station, which is portrayed as a collection of routes and tracks. Subsequently, we define the station's initializer and the scheduler. The initializer set all the elements of the station to their initial state. The scheduler nondeterministically selects a method that can be executed and proceeds with its execution. Additionally, we define the property that we seek to validate, named "Secure Station," which corresponds in this example to the property ‘two incompatible routes cannot be active together’. The loop of the scheduler tries to establish that the property is preserved by each method. Notably, this preservation is not true for the initial property itself, "Secure Station." However, the verification succeeds when we provide a stronger inductive invariant that implies the original property - called "Secure Inductive Station".

```
datatype TrackState = locked | free
class Track
{
    var state : TrackState
    var unlock_command : bool

    method stayFree()
        modifies this
        requires this.state =free
        ensures this.state =free
        ensures this.unlock_command =old(this.unlock_command)
    {
        this.state :=free;
    }



    method goFree()
        modifies this
        requires this.state =locked
        requires this.unlock_command =true
        ensures this.state =free
        ensures this.unlock_command =false
    {
        this.state :=free;
        this.unlock_command :=false;
    }

    method initialize()
        modifies this
        ensures this.state =free
        ensures this.unlock_command =false
    {
        this.state :=free;
        this.unlock_command :=false;
    }

    method execute()
        modifies this

        ensures (old(state =free)) ⟹(state =free ∧unlock_command =old(unlock_command))

        ensures (old(state =locked) ∧old(unlock_command)) ⟹(state =free ∧¬unlock_command)

        ensures (old(state =locked) ∧¬old(unlock_command)) ⟹(state =old(state) ∧unlock_command =
old(unlock_command))
        {
            match state {
                case free ⇒
                        {
                            stay_free();
                        }

                case locked ⇒
                    if unlock_command
                    {
                        go_free();
                    }
            }
        }

}
```

Figure 5.1: Dafny encoding of the track component

```
include "track.dfy"

datatype RouteState = idle | wait | active

class Route
{
    const usedtracks : seq<Track>
    var state : RouteState

    method go_wait()
        modifies this
        requires state =idle
        ensures this.state =wait
        {
            this.state :=wait;
        }

    predicate precondiation_activate()
        reads this, usedtracks
        {
            (∀ t •t in usedtracks ⟹(t.state =free ∧t.unlock_command =false))
        }

    method activateRoute()
        modifies this
        modifies this.usedtracks‘state
        requires this.state =wait
        requires precondiation_activate()
        ensures this.state =active
        ensures ∀t •(t in this.usedtracks ⟹t.state =locked)
        {
            var i :=0;

            while i < |this.usedtracks|
                invariant 0 ≤i ≤|this.usedtracks|
                invariant ∀t •t in this.usedtracks[..i] ⟹t.state =locked
            {
                this.usedtracks[i].state :=locked;
                i :=i + 1;
            }

            this.state :=active;
        }

    method deactivateRoute()
        modifies this, this.usedtracks‘unlock_command
        requires this.state =active
        ensures this.state =idle
        ensures ∀t •t in this.usedtracks ⟹t.unlock_command =true
        {
            var i :=0;

            while i < |this.usedtracks|
                invariant 0 ≤i ≤|this.usedtracks|
                invariant ∀t •t in this.usedtracks[..i] ⟹t.unlock_command
            {
                this.usedtracks[i].unlock_command :=true;
                i :=i + 1;
            }

            this.state :=idle;
        }
```

Figure 5.2: Dafny encoding of the Route (Part 1)

```
    method initialize()
        modifies this
        ensures this.state =idle
    {
        this.state :=idle;
    }

    method execute()
        modifies this, this.usedtracks

        ensures old(state) =idle ⟹(state =wait
            ∧(∀ t •(t in this.usedtracks ⟹t.state =old(t.state)))
            ∧(∀ t •(t in this.usedtracks ⟹t.unlock_command =old(t.unlock_command))))

        ensures (old(state) =wait ∧old(precondiation_activate())) ⟹(state =active
            ∧(∀ t •(t in this.usedtracks ⟹t.state =locked))
            ∧(∀ t •(t in this.usedtracks ⟹t.unlock_command =old(t.unlock_command))))

        ensures (old(state) =active) ⟹(state =idle
            ∧(∀ t •(t in this.usedtracks ⟹t.unlock_command))
            ∧(∀ t •(t in this.usedtracks ⟹t.state =old(t.state))))

        ensures (old(state) =wait ∧¬old(precondiation_activate()) ⟹(state =old(state)
            ∧(∀ t •(t in this.usedtracks ⟹t.state =old(t.state)))
            ∧(∀ t •(t in this.usedtracks ⟹t.unlock_command =old(t.unlock_command)))))
        {
            match state
            {
                case idle ⇒
                    go_wait();

                case wait ⇒
                    if precondiation_activate()
                    {
                        activateroute();
                    }

                case active ⇒
                    deactivateroute();
            }

        }
}
```

Figure 5.3: Dafny encoding of the Route (Part 2)

Our plan involves employing a parameterized model checker to automate the discovery of these inductive invariants, thereby streamlining and completely automatizing the verification process for generic safety properties within the railway logic.

```
include "route.dfy"

class Station

{
    const tracks : set<Track>
    const routes : set<Route>

    ghost predicate ValidStation()
     reads routes
    {
        ∧routes ≠{}
        ∧tracks ≠{}
        ∧(∀ i, t •i in routes ∧t in i.usedtracks ⟹t in tracks)
    }

    ghost predicate NotCompatibleRoutes( a : Route, b : Route)
        reads a, b
    {
        (∃ i •(i in a.usedtracks) ∧(i in b.usedtracks)) ∧a ≠b
    }

    ghost predicate SecureStation ()
        requires ValidStation()
        reads this, this.routes
    {
        ∀i,j •(i in this.routes) ∧(j in this.routes) ∧(NotCompatibleRoutes(i,j))
            ⟹¬(i.state =active ∧j.state =active)
    }

    ghost predicate SecureInductiveStation ()
        requires ValidStation()
        reads this, this.routes, this.tracks
    {
        ∧SecureStation()
        ∧(∀ i, j •((i in this.routes) ∧(j in this.tracks) ∧(j in i.usedtracks) ∧(i.state =active)) ⟹j.state =locked)
        ∧(∀ i, j •((i in this.routes) ∧(j in this.tracks) ∧(j in i.usedtracks) ∧(i.state =active)) ⟹j.unlock_command =
false)

    }


    method StationInitializer()
        modifies this, this.routes, this.tracks
        requires ValidStation()
        ensures SecureInductiveStation()
        {
            var a :=routes;
            var b :=tracks;
            while (a ≠{})
                invariant (∀ i •i in routes ∧i ∉a ⟹i.state =idle)
                {
                    var i : | i in a;
                    i.initialize();
                    a :=a − {i};
                }

            while (b ≠{})
                invariant ∀i •i in routes ⟹i.state =idle
                invariant (∀ i •i in tracks ∧i ∉b ⟹i.state =free ∧i.unlock_command =false)
                {
                    var i : | i in b;
                    i.initialize();
                    b :=b − {i};
                }
        }
```

Figure 5.4: Dafny encoding of the Station (Part 1)

```
    method StationScheduler()
        modifies this, this.routes, this.tracks
        requires ValidStation()
        requires SecureInductiveStation()
        ensures ValidStation()
        ensures SecureInductiveStation()
        {
            if *
            {
                var i : | i in routes;
                i.execute();
            }
            else
            {
                var i : | i in tracks;
                i.execute();
            }
        }

}
```

Figure 5.5: Dafny encoding of the Station (Part 2)

## 5.3 Invariant Inference with a Parameterized Model Checker

Alongside the Dafny encoding, our methodology includes encoding each component of the interlocking logic as a functional array-based transition system (Definition 21). More precisely, the transition system models an unbounded family of system components of the same type, facilitating scalable and modular verification.

The reader may have noticed that the behaviors exemplified in the Dafny code of the previous section, and discussed in Example 11 depict identical underlying systems. To further illustrate this connection, we highlight specific transition formulae that correspond to operations within the Dafny encoding.

The unlocking of a track segment, implemented in the Dafny method '*goFree*', is captured by the following transition formula:

$$
\begin{aligned}
T_1 \equiv \exists i : \text{track} \Bigg( & (\text{state\_t}[i] = \text{locked} \wedge \text{unlock\_command}[i] = \text{true}) \wedge \\
& \Bigg( \forall j.\, \text{state\_t}'[j] = \begin{cases} \text{free} & \text{if } i = j \\ \text{state\_t}[j] & \text{otherwise} \end{cases}, \\
& \text{unlock\_command}'[j] = \text{false}) \Bigg),
\end{aligned}
$$

which mirrors the postconditions of the associated 'execute' method in the Dafny code, illustrating the process for unlocking track segments.

Similarly, the activation of a route, encapsulated in the method '*activateRoute*', is formalized as:

$$
\begin{aligned}
T_2 \equiv \exists i : \text{route} \Bigg( & (\text{state\_r}[i] = \text{wait} \wedge \\
& (\forall t : \text{track}\, (\text{tracks\_used}(t, i) \rightarrow (\text{state\_t}[t] = \text{free} \wedge \text{unlock\_command}[t] = \text{false}))) \wedge \\
& (\forall j0 : \text{route}, j1 : \text{track}\, (\text{state\_r}'[j0] = \begin{cases} \text{active} & \text{if } i = j0 \\ \text{state\_r}[j0] & \text{otherwise} \end{cases}, \\
& \text{state\_t}'[j1] = \begin{cases} \text{locked} & \text{if tracks\_used}(j1, i) \\ \text{state\_t}[j1] & \text{otherwise} \end{cases}, \\
& \text{unlock\_command}'[j1] = \text{false})) \Bigg),
\end{aligned}
$$

indicating the logical sequence for activating routes based on system state and available track usage.

These formulae demonstrate the close alignment between the logical representation of system behaviors and their implementation in Dafny, underscoring our methodological approach to generate logical encodings alongside Dafny.

Continuing the example, the property that we desire to prove is the following, en-

suring that routes sharing a track cannot be active simultaneously:

$$F = \forall r0 : \text{route}, r1 : \text{route}\big(\text{NotCompatible}(r0, r1) \rightarrow \neg(\text{state\_r}[r0] = \text{active} \wedge \text{state\_r}[r1] = \text{active})\big).$$

where NotCompatible($i$ : route, $j$ : route) was defined as

$$\big(i \neq j \wedge \exists t : \text{track}\,(\text{tracks\_used}(i, t) \wedge \text{tracks\_used}(j, t))\big).$$

This property, can be automatically verified with respect to the system $C_1 \parallel C_2$, as delineated in Section 4, where $C_1$ is the transition system representing routes, and $C_2$ is the one for tracks.

Initial experiments have suggested that modeling the full complexity of the system's components results in excessively large system representations. As such, ongoing efforts are directed towards refining the encoding to manage system size more effectively. Successful verification with parameterized model checking will then necessitate the translation of inferred inductive invariants back into Dafny, enabling the verification of the railway logic's security beyond abstract models. In this example, the inductive property 'Secure Inductive Station' can be obtained also by running the algorithm of Chapter 3 on $C_1 \parallel C_2$.

## 5.4 Summary and Ongoing Work

We discussed the problem of formally verifying an interlocking logic expressed in a domain specific language. The interesting point is that the logic is parameterized, in the sense that it is intended to control any station with an arbitrary number of components. We analyzed a case study, with two main insights. First, we confirm that it is possible to directly encode the main features of the interlocking logic in Dafny in a very natural way. Second, we investigate verification and the relation to simple invariants (from predefined schemata) and to more complex invariants (resulting from the application of parameterized model checker algorithms).

Currently, we are working on extending the IDE for the Interlocking logic to support parameterized verification. The first step will be to devise an encoder to automatically generate Dafny code automatically from SysML. We expect this step to be relatively simple, given that the interlocking logic constructs have a direct correspondence to Dafny ones, and back-and-forth traceability can be achieved.

The Dafny code will incorporate both the method bodies, mirroring the C and Python code, and the preconditions and postconditions, echoing the engineers' natural language specifications. Second, we will integrate a way to express the properties to be proved, likely leveraging the language for specifying the abstract scenarios in the TOSCA environment [6].

Third, we will integrate the generation of invariants required to show that the interlocking logic satisfies the expected properties. On the one side, we will instrument the encoding to automatically generate "simple" invariants via templates, to check the

compliance of the generated code to specification. On the other, we will integrate a parameterized model checker (and possibly other invariant generators) to infer invariants for general safety properties of the interlocking logic.

In this direction, we aim to apply the algorithm of compositional verification to the array-based transition systems modelling dafny components. We will present preliminary results in this direction in Section (6.4). One of the main challenges we currently face lies in the manual translation process required to convert Dafny code into SMT formulae, and then inversely translating the derived invariants back into Dafny for further verification steps. The aspiration to streamline this process through automation has led to preliminary investigations, which suggest that achieving a seamless translation is non-trivial. The complexity of accurately mapping Dafny's rich, type-structured code into the logical expressions required by SMT solvers involves parsing and semantic interpretation challenges.

Given these complexities, it becomes apparent that a collaborative approach, incorporating more abstract modeling techniques and leveraging the domain expertise of railway engineers, might be necessary. Such collaboration could facilitate the identification of critical safety properties and the development of a more intuitive understanding of the system's behavior, potentially simplifying the invariant generation process. This strategy aligns with our broader objective to not only automate the verification process but also to ensure that it remains closely aligned with the real-world operational requirements and safety standards of railway systems.

A figure representing a summary our intended workflow is depicted in 5.6.

Finally, we do not dismiss the idea of a semi-automated approach, where railway engineers can contribute lemmas or provide guidance in the abstraction process, potentially in controlled natural language, to help the verification process.

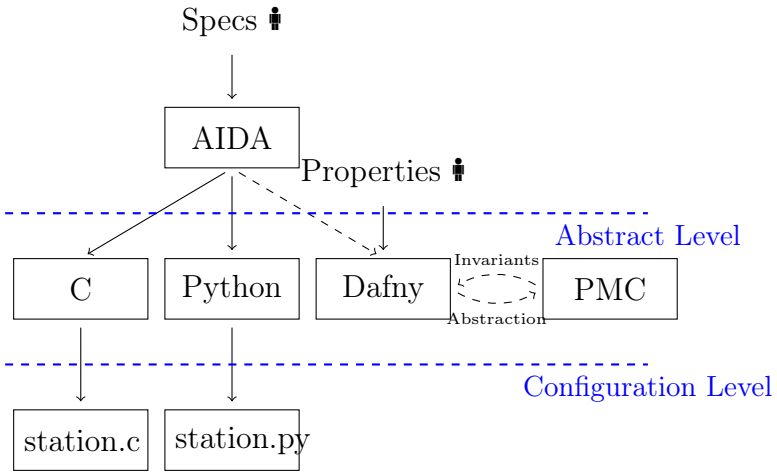


Figure 5.6: Illustration of the integrated development environment (IDE) designed for crafting interlocking logic systems. This environment enables engineers to specify the logic of a generic (parameterized) railway station using controlled natural language within the AIDA tool. Subsequently, AIDA facilitates the automated generation of corresponding C, Python, and Dafny code. The Dafny code is then subjected to an interaction with parameterized model checking algorithms aimed at identifying inductive invariants or uncovering potential counterexamples based on predefined safety properties. For specific station configurations, the IDE can further compile this verified logic into executable code, ensuring the reliable operation of the station's interlocking system.

# Chapter 6

# Experimental Evaluation

This section presents the empirical evaluation of the algorithms developed in Chapters 3 and 4. Our objective is to assess the efficiency and effectiveness of these algorithms across a range of benchmarks. These benchmarks are selected to represent various scenarios in the verification mainly of parameterized systems, emphasizing the practical applicability of our approach.

## 6.1 Implementation

We have implemented the algorithms described in chapter 3 using Python3. In this section, we test the algorithms on various benchmarks, by considering different options and back-end engines.

For the first algorithm, described in section 3.2, which we will denote in the following as UPDRIA, we used the SMT solver Z3 [50] for satisfiability queries (using the approach [84] for quantifiers). We also implemented a simple loop to ensure that every index model found by the solver would be of minimal (finite) size: in theory, when working outside the decidable fragment 11, there is no guarantee that such a finite model exists. However, this was never an issue in our experiments. We used instead MATHSAT [37] to extract interpolants from ground unrollings, as explained in Section 3.2.4.

For the second algorithm, described in section 3.3, denoted as LAMBDA, we used IC3IA [33] as a quantifier-free model checker. Finally, the sub-procedure of invariant checking with SMT solving explained in 3.3.4, has been implemented on top of MATHSAT.

Upon termination, both algorithms return either a counterexample trace in a ground instance, or an inductive invariant which can also be checked externally with automatic provers such as Z3, Vampire or CVC5 [50, 86, 8].

For the algorithm of compositional verification, described in section 3.3, we leveraged both Z3 and MATHSAT. Given that MATHSAT lacks support for quantified formulae, we exclusively utilized Z3 for the 'Induction Check' sub-procedure. For 'Spurious Check' and 'Refinement' sub-procedures, both MATHSAT (with quantifiers instantiated as described in 4.2.1) and Z3 were employed.

We tested the algorithms over different benchmarks. In Section 6.2, we will discuss

the results of applying the algorithm to parameterized protocols. Then, in Section 6.3 we will show how it is possible to use Lambda and Updria for invariant checking of transition systems over the theory of arrays. Finally, we will show some results on asynchronous composition of families of array-based transition systems.

## 6.2 Application to parameterized protocols

For this family of experiments, we organized the experimental evaluation as follows: first, we compare the two algorithms and discuss different options in Section 6.2.2; then, we compare the algorithms to other tools in Section 6.2.3. The tools we considered are the model checkers Mcmt, Cubicle, Ic3po, and MyPyvy, all previously mentioned in Section 3.4. All benchmarks and our implementations are available at the following link: `https://drive.google.com/file/d/1_lIUa_Y-yKAhj5bXnX1hLEovFmP3Jos9/view?usp=sharing`.

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.

### 6.2.1 Benchmarks

The first family of this benchmarks consists of 238 array-based systems in Mcmt format, modelling parameterized protocols, timed systems, programs manipulating arrays, and more, taken from [40, 25, 23] and from the standard Mcmt distribution. Often, in those systems, the index theory is pure equality or the theory of a linear order; instead, the theory of element is usually $\mathcal{QF}\_\mathcal{LIA}$, $\mathcal{QF}\_\mathcal{LRA}$, or the theory of an enumerated datatype. On this set of benchmarks, we could run all the versions of the algorithms presented in this paper, as well as Mcmt. In principle, such benchmarks are supported also by Cubicle; in practice, however, Cubicle and Mcmt use two (quite) different input languages, so we could only run the former on the 42 instances of this family for which the Cubicle translation is available.

The second family of benchmarks considered are 52 array-based systems in VMT [39] and MyPyvy format, taken from [70, 113]. This family does not use theories for array elements (i.e. $\mathcal{T}_E$ is simply the theory of Booleans), but is more liberal than the previous one in the shape of the formulae used in the transitions. Therefore, we could not run Mcmt (nor Cubicle) on this family, nor we could use the parameter abstraction technique of Section 3.3.3 on them. This family of systems can be used as inputs to the model checkers Ic3po and MyPyvy.

Finally, the third family consists of a set of 25 array-based transition systems modeling simple train verification problems [6], on which only the two algorithms presented in this paper can be applied, due to the presence of various SMT theories and non-restricted transition formulae.

### 6.2.2 Comparison of UPDRIA and LAMBDA

We compare now the first two algorithms described in this chapter 3, UPDRIA and LAMBDA. In addition to the basic versions described in Sections 3.2 and 3.3, for UPDRIA we also implemented a variant of the algorithm (denoted with option –**size-**$n$) which combines some of the ideas of Section 3.3. With this option, the algorithm starts by considering a ground instance of size $n$, and extracts a set of lemmas from it (as in 3.3.2) which can help the algorithm to converge faster or can be discarded if falsified later. For the majority of the benchmarks considered, this strategy (with $n = 3$) improved the performances of UPDRIA. For LAMBDA, we consider various options; with the flag –**no-symm**, we do not use the results of Section 2.3.2 to help the model checking of ground instances. With the flag –**no-invgen**, we do not apply the invariant minimization technique explained in 3.3.2. Moreover, we distinguish three options for implementing the invariant checking sub-procedure of Algorithm 3: the option –**param** consists of the implementation of the parameter abstraction technique of Section 3.3.3; the option –**ind** applies the procedure of Section 3.3.4. Finally, the option –**z3** simply calls Z3 on inductive queries.

We tested all such options on the first family of benchmarks, where it was possible to compare them all. A summary of the results is reported in Table 6.1, where we show the number of solved benchmarks and the total time taken by the tool; we also report plots comparing different LAMBDA options in Figure 6.1, and different UPDRIA options in Figure 6.2. Those plots show, for each point in time, the number of solved instances up to that time. We consider as solved instances the ones where the algorithms terminate with SAFE or UNSAFE.

Table 6.1: Summary of experimental results on MCMT benchmarks.

| | **Tot solved** | **Tot time** |
|---|---|---|
| LAMBDA–ind | 216 | 2773s |
| LAMBDA–nosymm–ind | 216 | 8740s |
| LAMBDA–z3 | 214 | 11615s |
| LAMBDA–nosymm–z3 | 214 | 14615s |
| UPDRIA–size-3 | 208 | 24038s |
| LAMBDA–param | 206 | 1055s |
| LAMBDA–noinvgen–ind | 204 | 4073s |
| LAMBDA–nosymm–param | 203 | 17313s |
| LAMBDA–noinvgen–param | 202 | 1914s |
| LAMBDA–noinvgen–nosymm–param | 202 | 10158s |
| LAMBDA–noinvgen–z3 | 191 | 8897s |
| LAMBDA–noinvgen–nosymm–ind | 180 | 7986s |
| UPDRIA–size-2 | 197 | 17488s |
| UPDRIA–size-4 | 197 | 20943s |
| LAMBDA–noinvgen–nosymm–z3 | 166 | 3025s |
| UPDRIA | 162 | 27649s |

In the case of UPDRIA, most of the time used by the algorithm is consumed by Z3 for proving the (un)satisfiability of quantified queries. In this case, when the procedure diverges, it usually does so by extending the frame sequence without converging to an inductive invariant. In general, the algorithm can also diverge with a sequence of refinement failures: as explained in Section 3.2.4, this can also happen when and all the predicates of the interpolants extracted by MATHSAT are already in the predicate set $\mathcal{P}$.

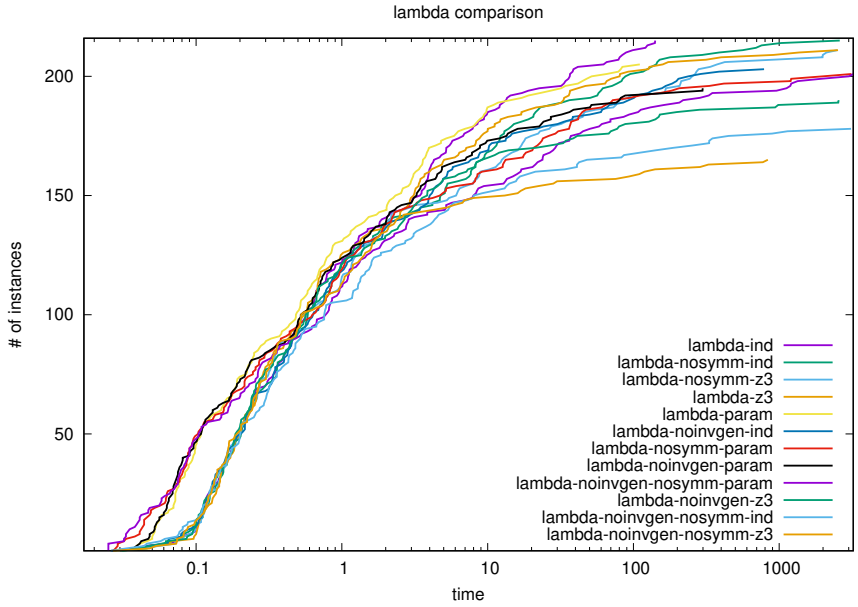


Figure 6.1: Plot comparing different options of LAMBDA

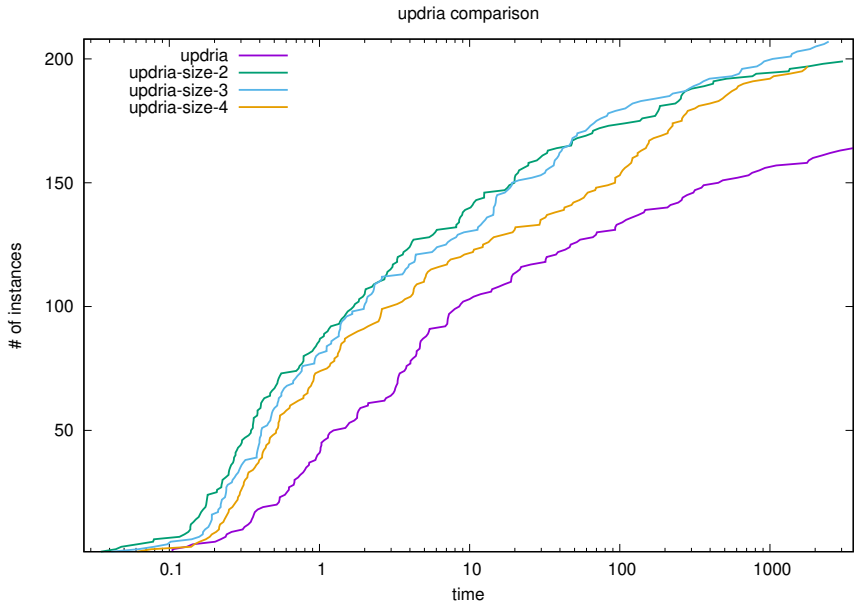


Figure 6.2: Plot comparing different options of UPDRIA

However, the latter possibility is rarer in our experiments: in the MCMT benchmarks, this happened only once. Interestingly, we discovered that for that model, no universal inductive invariant exists.

The performance profile of LAMBDA, instead, is very different from the above. The procedure relies less on quantified reasoning, and most of its time is spent in the model checking of ground instances. When the tool diverges, it is usually during this phase:

if an invariant is not found, larger and larger (ground) instances are analyzed.

From the experiments, it was clear that the results about symmetries of Section 2.3.2 reduced drastically the time of model checking of ground instances, thus improving the overall performances. The technique of invariant reduction of Section 3.3.2 was also helpful, in particular, to obtain better lemmas from finite instances (meaning that they generalized better to other instances). In this experimental evaluation, the usage of the parameter abstraction technique does not show particular benefits in contrast to the usage of SMT-solving techniques on inductive queries. Although for some instances the computation and model checking of the abstract system was quicker, it is possible to see the prophecy variables of the parameter abstraction as an *a priori* fixed instantiation strategy, thus reducing the generality of the approach. Finally, we can also see how our simple instantiation technique was more efficient than a naive usage of Z3: this is because, in the case of non-inductive lemmas, the solver takes a lot of time in building models for counterexamples to induction. Instead, our approach discards lemmas with an additional ground instance exploration, as explained in Section 3.3.4.

Finally, we remark on the difference in time consumed by the two algorithms, UPDRIA and LAMBDA. As already mentioned, this is due to the fact that most of the UPDRIA queries are quantified, whereas LAMBDA tries to avoid that as much as possible. To better illustrate the difference, in Figure 6.3, we report a scatter plot comparing the two tools with their respective best options.

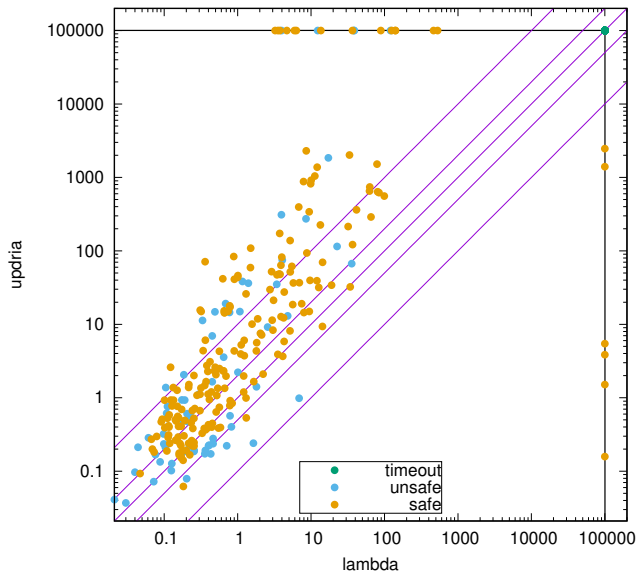


Figure 6.3: Scatter plot comparing UPDRIA and LAMBDA

### 6.2.3 Comparison with other tools

We report in Table 6.3 a summary of the comparison of the algorithms described in this paper and the other available model checkers. The table shows, for each tool/configura-

tion, the number of solved instances for each of the family of benchmarks we considered. In the second column, we show the number of benchmarks solved in the subset of the first family available also for CUBICLE. When a tool is not applicable to a family, we use the symbol '-'. For UPDRIA and LAMBDA, we used the best options according to the previous section. For MCMT, we used standard settings: the tool has however different strategies, but we did not investigate the best ones for each benchmark. For CUBICLE, we used the **–brab 2** option. For MYPYVY, we used the implementation and the best options given in the artifact of [83]. For IC3PO, we used the option **–finv=2** to discharge unbounded checks with Z3. The properties of the tools we consider are summarized in Table 6.2.

Table 6.2: Summary of the characteristics of the tools considered

| | $\mathcal{T}_E$ **only Boolean** | $\mathcal{T}_E$ **generic** | $T(X, X')$ **generic** | $\forall\exists$ invariants |
|---|---|---|---|---|
| UPDRIA | ✓ | ✓ | ✓ | - |
| LAMBDA | ✓ | ✓ | ✓ | - |
| MCMT | ✓ | ✓ | - | - |
| CUBICLE | ✓ | ✓ | - | - |
| IC3PO | ✓ | - | ✓ | ✓ |
| MYPYVY | ✓ | - | ✓ | ✓ |

Table 6.3: Summary of experiments on all benchmarks.

| | **MCMT (238 tot)** | **CUBICLE (42 tot)** | **VMT (52 tot)** | **Trains (25 tot)** |
|---|---|---|---|---|
| UPDRIA | 208 | 30 | 29 | 24 |
| LAMBDA | 214 | 35 | 29 | 25 |
| MCMT | 172 | 24 | - | - |
| CUBICLE | - | 31 | - | - |
| IC3PO | - | - | 36 | - |
| MYPYVY | - | - | 27 | - |
| **VBS** | 220 | 36 | 37 | 25 |

We also show, in Figures 6.4, 6.5, 6.6, plots comparing LAMBDA and UPDRIA with the other tools on single benchmarks families. We can see from the table and the plots the generality of our approach, and the fact that both UPDRIA and LAMBDA compare well with the state of the art.

In particular, LAMBDA is outperformed only by IC3PO in the second family of benchmarks, solving 7 more instances. In these cases, IC3PO finds an inductive invariant that contains an existential quantifier, while LAMBDA (and UPDRIA) only search for universally quantified inductive invariants. In particular, UPDRIA diverges on those instances by discovering always the same predicates from a finite instance, as explained in Section 3.2.4.

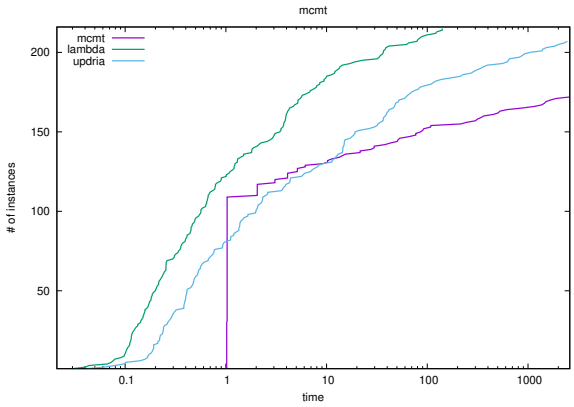


Figure 6.4: Plot for Mcmt family

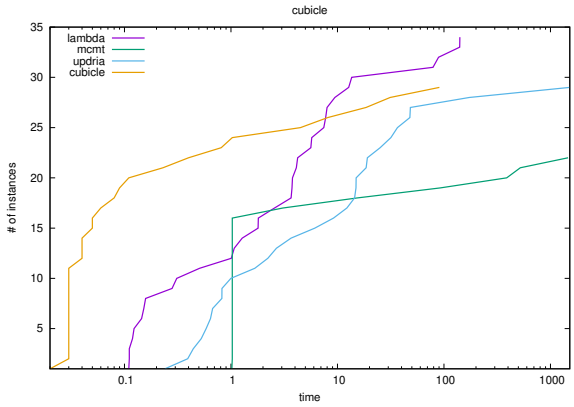


Figure 6.5: Plot for Cubicle family

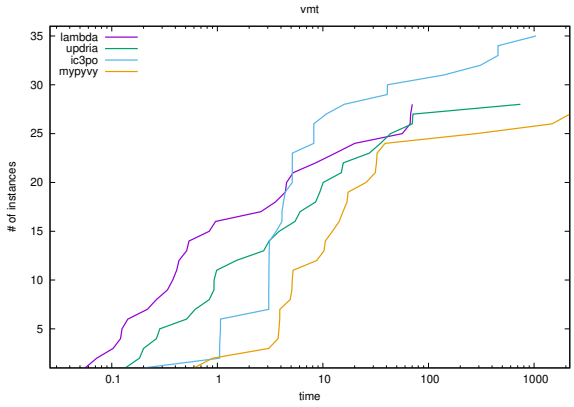


Figure 6.6: Plot for VMT family

## 6.3 Application to array systems

In this section, we elaborate on how the algorithms discussed in the previous chapter can be applied to symbolic transition systems defined over the quantifier-free theory of arrays (with constant arrays), as outlined in Section 2.1.1. These symbolic transition systems are pivotal in modeling programs that manipulate arrays or, more broadly, the heap.

A significant aspect of these systems is that the properties intended for verification are often expressed through universally quantified formulae. However, we always reduce to the quantifier-free cases by applying Proposition 32.

We therefore consider systems denoted with symbolic transition systems $S = (X, I(X), T(X, X'))$, over the theory $\mathcal{T}_A$ of arrays, and a formula $\phi(X)$, where $I, T, \phi$ are quantifier-free formulae. It is well known [12] that the satisfiability problem of array formulae can be reduced to the theory of equality with uninterpreted functions $\mathcal{T}_{\mathcal{EUF}}$; by mirroring such transformations, we can reduce the verification of array systems to our formalism as well. In case that the index sort of the array theory is uninterpreted, then the transformation is straightforward. However, in case of integers, we need to address certain nuances. Therefore, in the following, we fix the index sort of the array theory to be the integers (allowing the element sort to be arbitrary). Recall that in our definition of array-based transition system (definition 10), we necessitate that the index sort's universes are finite. The key insight here is the fact that satisfiability of quantifier-free array formulae is equivalent to EUF satisfiability *over a finite integer subset*, and thus the conversion is still sound.

Despite the system being defined through quantifier-free formulae, it is common for the inductive invariants to necessitate quantifiers over the index sort. This arises because the operations of writing to and reading from arrays implicitly encapsulate quantification within such systems.

The integration of transition systems over arrays into our formalism entails constructing an equivalent array-based transition system, $\widetilde{S}$, which maintains a bisimulation relationship with the original system. A bisimulation relation is a relation between states that preserves reachability in both directions. Therefore, a safe result for $\widetilde{S}$ is a safe result for $S$, and the same applies to counterexamples.

We only discuss this transformation informally, not focusing on all the details. The main question we are facing is to understand if the algorithms in the previous sections can be applied successfully to these systems as well.

To define $\widetilde{S}$, we introduce a theory $\mathcal{T}_I$ for an interpreted index sort. Moreover, we define an injective mapping $\mu$ that maps this index sort to integers.

Then, for every array variable $x \in X$, a corresponding uninterpreted function $\widetilde{x}$ is defined from the uninterpreted index sort to the element sort. To define the initial and transition formula for $\widetilde{S}$, we apply transformation rules on the formulae in $I$ and $T$. First, we can assume that all atoms occurring in formulas in $I$ and $T$ are flat, i.e., the only array atoms occurring in the formulae are equalities among arrays or equalities of the form $wr(a, k, e) = b$, where $a$ and $b$ are array constant symbols and $k, e$ are constants of index and element type, respectively. Then, we remove the array theory

by using the following rewriting rules:

i Array equalities $a = b$ for arrays $a$ and $b$ are transformed into $\forall i.\widetilde{a}(i) = \widetilde{b}(i)$.

ii Array updates $wr(a, k, e) = b$ morph into
$\forall i.(\mu(i) = k \to \widetilde{a}(i) = e) \wedge (\mu(i) \neq k \to \widetilde{a}(i) = \widetilde{b}(i))$.

iii Access operations of the form $rd(a, k)$ convert into $\widetilde{a}(i)$, introducing a fresh variable $i$ for the index sort and incorporating the condition $\mu(i) = k$ within the formula.

The reformulated system $\widetilde{S}$ bisimulates the original system while ensuring that every function $\widetilde{x}$ is defined over an index set $U$, with $\mu$ mapping $U$ to a finite set of integers. The correspondence with the original array $x$ is defined via $x[i] := \widetilde{x}(\mu^{-1}(i))$ for all the integers $i$ within the image of $U$ under $\mu$; otherwise, the array value can be set as a default deducible constant. If there are no constant arrays in the formula, such constant can be any element value. Otherwise, additional reasoning is required.

These transformations adeptly prepare the system for the application of Updria and Lambda algorithms.

We amassed a dataset consisting of 309 benchmarks on array-based transition systems, derived from the paper [59] and the array category of the *CHC-COMP 2020* [119]. Our analysis involved a comparison of two algorithms against Prophic3, a tool specifically tailored for verifying array systems. Other notable tools capable of handling these systems and synthesizing universally quantified invariants include **FreqHorn** [59], **QUIC3** [75], and **gSpacer** [85]. However, these tools necessitate systems described as CHCs, whereas both Prophic3 and our algorithms are compatible with the same input format (VMT files). As documented in [96], Prophic3 performs comparably—and often better—than these alternatives in solving instances.

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.

The summary of the results from our experimental evaluation are presented in the following table:

| Tool | Solved Safe | Solved Unsafe |
|---|---|---|
| Prophic3 | 158 | 34 |
| Lambda | 105 | 33 |
| Updria | 115 | 32 |

Table 6.4: Experimental results on transition systems over $\mathcal{T}_A$

These results demonstrate that, while the algorithms may not outperform specialized tools in verifying properties of arrays, they exhibit competitive performance, particularly in generating counterexamples. Notably, the optimal performance for Lambda was achieved using the **–no-symm** flag, differing from previous cases. This suggests that the inclusion of integer constants possibly complicates the application of index

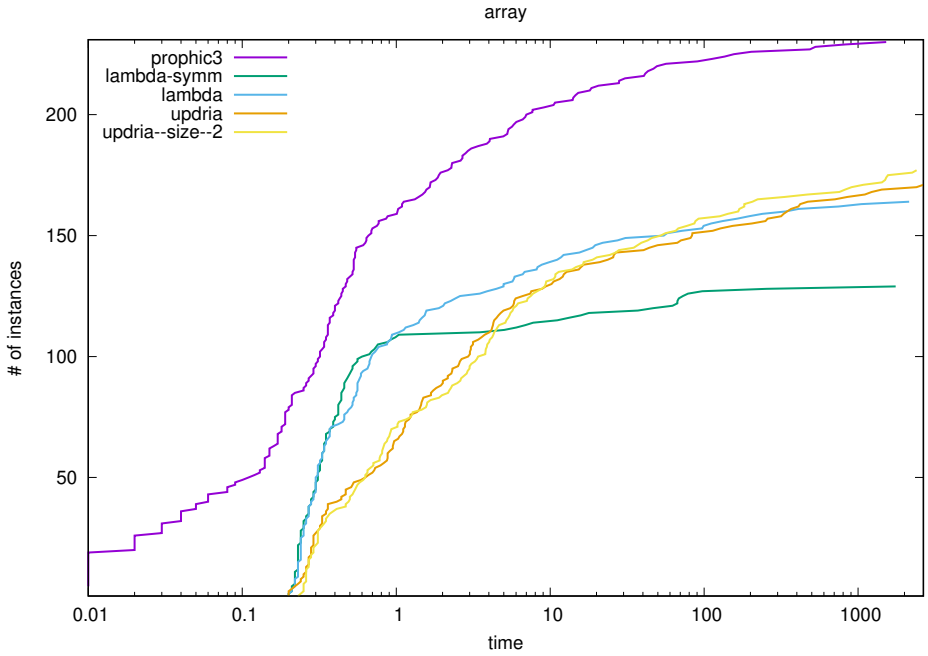


Figure 6.7: Comparison of LAMBDA, UPDRIA and PROPHIC3 on array benchamrks.

domain symmetries in finite instance model checking. Moreover, LAMBDA's performance was outpaced by UPDRIA, aligning with expectations since LAMBDA's strategy—generalizing proofs from finite instances to protocols—may not extend effectively to array programs. The most effective setting for UPDRIA was –**initial-concretization 2**, though the difference from the default configuration was minimal, unlike in prior scenarios. A cactus plot comparing different option can be seen in Figure 6.7.

Although these experiments are preliminary, they affirm the applicability of the described algorithms to array systems. Further investigation is warranted to match or surpass state-of-the-art solutions, yet we remain optimistic about the potential advancements.

## 6.4 Application to asynchronous composition of systems

We discuss in this last section of experimental evaluation the application of the algorithm presented in 4.2.

As we already mentioned, this algorithm is intended for application in a project focused on the verification of interlocking logic; in that project, we aim to automatically generate symbolic transition systems that encapsulate all components of interlocking logic, directly from the specification or railway engineers in the tool discussed in Chapter 5. The current design of the logic encompasses approximately 100 components, each featuring between 2 to 10 variables. As this is still work in progress, we report here our results on a simplified case study on interlocking logic, modeling only 5 parameterized components: routes, tracks, level crossings, switches, and signals. It's important to note that these components represent significantly simplified versions compared to their real-world counterparts.

In this case study, we evaluated the algorithm against two properties. The first, a true assertion, states that two routes sharing a track cannot be active simultaneously. The results are in 6.5. The discrepancies observed during refinement between MATHSAT and Z3 are due to the fact that the solvers find different variables in the unsat cores. Generally, Z3, which processes quantified formulae during the 'Spurious Check', tends to be slower but, in this specific instance, it finds the 'correct' variables (i.e. the ones needed for the inductive invariant) before then MATHSAT. We also test the algorithm on a second property, a false assertion whose counterexample involves most of the components and variables of the system. This is a situation that demonstrate the limitations of our procedure compared to a monolithic approach, where it is applied on a context where 'most' of the behaviours of the various components is needed. Results are detailed in Table 6.6.

Table 6.5: Interlocking: Safe Property

| Method | Solver | Time(s) | #Refinement_steps | #var | #components |
|---|---|---|---|---|---|
| Monolithic | - | 9.28 | - | 15 | 5 |
| Compositional | z3 | 4.20 | 3 | 5 | 2 |
| Compositional | msat | 4.63 | 4 | 7 | 2 |

Table 6.6: Interlocking: Unsafe Property

| Method | Solver | Time(s) | #Refinement_steps | #var | #components |
|---|---|---|---|---|---|
| Monolithic | - | 0.88 | - | 15 | 5 |
| Compositional | z3 | 7.56 | 4 | 12 | 3 |
| Compositional | msat | 4.64 | 4 | 12 | 3 |

A second source of benchmarks derives from two parameterized protocols, which we

have adapted by integrating $N$ components capable of altering certain shared variables. These protocols are a basic mutual exclusion protocol and a variant of the bakery protocol. We test the algorithm on properties that are true and are independent of the modifications enacted by these additional components. This scenario ideally showcases the strengths of our algorithm, as the inductive invariant for the properties can be determined by examining only a fraction of the system, ignoring the rest. The results are depicted in Figure 6.8. The x-axis represents the number of components, while the y-axis measures the time taken by the procedures in seconds (presented on a logarithmic scale). Compared to the monolithic approach, our algorithm's verification process is significantly less time-consuming and remains relatively constant and similarly, the number of refinements required by the algorithm stays steady.

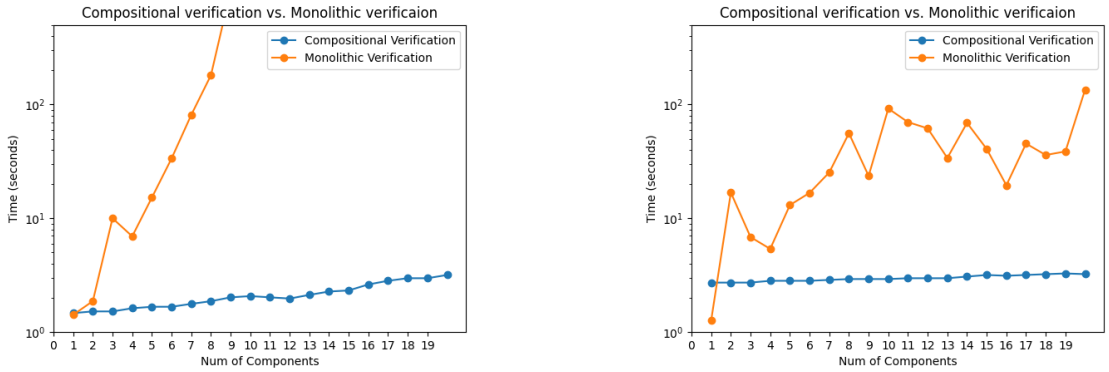


Figure 6.8: Protocols with additional components

# Chapter 7

# Conclusions and Future Work

## 7.1 Conclusions

In this thesis, we tackled the problem of SMT-based verification of parameterized systems. The formalism we use to describe such systems is based on the ones already present in the literature, which we have extended to suit a more general setting. This is motivated by the need for automated verification tools capable of handling the intricate nature of parameterized systems, which are not only infinite-state but also encapsulate quantifiers within their system descriptions, thereby complicating the verification process.

We proposed two novel algorithms designed to automate the verification of safety properties in such systems. These algorithms are grounded in existing techniques but incorporate innovative strategies to overcome the challenges posed by parameterized verification.

The first algorithm, known as Updria, is an enhancement of the UPDR algorithm with the addition of predicate abstraction to effectively manage a general SMT theory representing the data of the parameterized system. This approach builds on the strengths of the original UPDR method but introduces the need for frequent, computationally intensive quantified queries to an external solver. This aspect poses a significant challenge for its practical deployment, as these queries can considerably decelerate the verification process. Despite these hurdles, Updria presents a powerful solution for discovering safety invariants within complex parameterized systems.

The second algorithm is called Lambda, which embarks on a different path by initially limiting the parameter's cardinality. This restriction permits the verification of the system without quantifiers, utilizing existing methodologies. Lambda then seeks to generalize the found inductive invariant into a quantified candidate applicable to the original system. To validate this generalized invariant, Lambda explores two main strategies: leveraging Parameter Abstraction for creating a quantifier-free system and its property verification, and employing a bounded approach to quantified SMT reasoning. These approach aims to curtail the potential for validity check divergences, positioning Lambda as an efficient method for parameterized system verification.

Moreover, our exploration into the verification of asynchronous composition of sys-

tems, with a particular focus on industrial applications in interlocking logic design, yielded promising initial results. The potential for integrating the invariants identified by our algorithms into Dafny code could significantly bolster the robustness and reliability of system designs in an industrial setting.

The experimental evaluations of our algorithms underscored their efficacy and efficiency in verifying the safety properties of parameterized systems.

## 7.2 Future Work

The research presented in this thesis opens several avenues for future exploration.

An immediate direction is the generation of invariants that include not only universal quantifiers but also existential ones. We plan to leverage a generalization approach that introduces existential quantifiers, as discussed in [70]. Additionally, interpolation techniques could be employed within our framework to infer quantifier alternation, as suggested by [53].

Extending our verification algorithms to encompass liveness properties, in addition to safety, presents another promising research trajectory. Techniques for proving liveness properties can vary from reductions to safety, which are often not automated, as outlined in [109], to more direct approaches like [56], which could offer insights into generalizing proofs of liveness from finite domains. This concept aligns with the spirit of the Lambda algorithm and could use a liveness model checker for ground instances such as [48].

A notable limitation of our current methodology is its inability to explicitly reason about cardinality constraints. We aim to address this limitation by integrating specialized approaches, like those found in [89, 69, 66, 123, 4], with our proposed verification procedures, in order to create a more comprehensive framework that accommodates a broader spectrum of properties and systems.

In conclusion, this thesis lays the groundwork for further advancements in the SMT-based verification of parameterized systems. The algorithms and methodologies we have developed contribute to the academic discourse and could also offer practical solutions to real-world verification challenges.

# Bibliography


[1] Abdulla, P. A., Čerāns, K., Jonsson, B., and Tsay, Y.-K. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation 160*, 1 (2000), 109–127.

[2] Achterberg, T. Scip: Solving constraint integer programs. *Mathematical Programming Computation 1* (07 2009), 1–41.

[3] Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., and Sharygina, N. Safari: Smt-based abstraction for arrays with interpolants. In *Computer Aided Verification* (Berlin, Heidelberg, 2012), P. Madhusudan and S. A. Seshia, Eds., Springer Berlin Heidelberg, pp. 679–685.

[4] Alberti, F., Ghilardi, S., and Pagani, E. Counting constraints in flat array fragments. In *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706* (Berlin, Heidelberg, 2016), Springer-Verlag, p. 65–81.

[5] Amendola, A., Becchi, A., Cavada, R., Cimatti, A., Ferrando, A., Pilati, L., Scaglione, G., Tacchella, A., and Zamboni, M. Norma: a tool for the analysis of relay-based railway interlocking systems. In *Tools and Algorithms for the Construction and Analysis of Systems* (Cham, 2022), D. Fisman and G. Rosu, Eds., Springer International Publishing, pp. 125–142.

[6] Amendola, A., Becchi, A., Cavada, R., Cimatti, A., Griggio, A., Scaglione, G., Susi, A., Tacchella, A., and Tessi, M. A model-based approach to the design, verification and deployment of railway interlocking system. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications* (Cham, 2020), T. Margaria and B. Steffen, Eds., Springer International Publishing, pp. 240–254.

[7] Aminof, B., Kotek, T., Rubin, S., Spegni, F., and Veith, H. Parameterized model checking of rendezvous systems. *Distrib. Comput. 31*, 3 (jun 2018), 187–222.

[8] Barbosa, H., Barrett, C. W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., and Zohar, Y. cvc5: A versatile and industrial-strength SMT solver. In *Tools and*

*Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022* (2022), vol. 13243 of *Lecture Notes in Computer Science*, Springer, pp. 415–442.

[9] Barrett, C. W., Stump, A., and Tinelli, C. The smt-lib standard version 2.0.

[10] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. Symbolic model checking without bdds. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 1999), W. R. Cleaveland, Ed., Springer Berlin Heidelberg, pp. 193–207.

[11] Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., and Zhu, Y. Bounded model checking. *Adv. Comput. 58* (2003), 117–148.

[12] Biere, A., Heule, M., van Maaren, H., and Walsh, T., Eds. *Handbook of Satisfiability*, vol. 185. IOS Press, 2009.

[13] Birgmeier, J., Bradley, A. R., and Weissenbacher, G. Counterexample to induction-guided abstraction-refinement (CTIGAR). In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings* (2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 831–848.

[14] Bloem, R., Jacobs, S., and Khalimov, A. *Decidability of Parameterized Verification*. Morgan & Claypool Publishers, 2015.

[15] Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Mahboubi, A., Mebsout, A., and Melquiond, G. A simplex-based extension of fourier-motzkin for solving linear integer arithmetic. In *Automated Reasoning* (Berlin, Heidelberg, 2012), B. Gramlich, D. Miller, and U. Sattler, Eds., Springer Berlin Heidelberg, pp. 67–81.

[16] Bonacina, M. P., Fontaine, P., Ringeissen, C., and Tinelli, C. Theory combination: Beyond equality sharing. In *Description Logic, Theory Combination, and All That - Essays Dedicated to Franz Baader on the Occasion of His 60th Birthday* (2019), C. Lutz, U. Sattler, C. Tinelli, A. Turhan, and F. Wolter, Eds., vol. 11560 of *Lecture Notes in Computer Science*, Springer, pp. 57–89.

[17] Bonacina, M. P., Ghilardi, S., Nicolini, E., Ranise, S., and Zucchelli, D. Decidability and undecidability results for nelson-oppen and rewrite-based decision procedures. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings* (2006), U. Furbach and N. Shankar, Eds., vol. 4130 of *Lecture Notes in Computer Science*, Springer, pp. 513–527.

[18] Bozga, M., Bueri, L., and Iosif, R. On an invariance problem for parameterized concurrent systems. In *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland* (2022), B. Klin, S. Lasota, and A. Muscholl, Eds., vol. 243 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 24:1–24:16.

[19] Bozga, M., Iosif, R., and Sifakis, J. Verification of component-based systems with recursive architectures. *Theor. Comput. Sci. 940*, Part (2023), 146–175.

[20] Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T. A., van Rossum, P., Schulz, S., and Sebastiani, R. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings* (2005), N. Halbwachs and L. D. Zuck, Eds., vol. 3440 of *Lecture Notes in Computer Science*, Springer, pp. 317–333.

[21] Bradley, A. R. Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation* (Berlin, Heidelberg, 2011), R. Jhala and D. Schmidt, Eds., Springer Berlin Heidelberg, pp. 70–87.

[22] Bradley, A. R., Manna, Z., and Sipma, H. B. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation* (Berlin, Heidelberg, 2006), E. A. Emerson and K. S. Namjoshi, Eds., Springer Berlin Heidelberg, pp. 427–442.

[23] Bruschi, D., Pasquale, A. D., Ghilardi, S., Lanzi, A., and Pagani, E. A formal verification of arpon - A tool for avoiding man-in-the-middle attacks in ethernet networks. *IEEE Trans. Dependable Secur. Comput. 19*, 6 (2022), 4082–4098.

[24] Bruttomesso, R., Ghilardi, S., and Ranise, S. Rewriting-based quantifier-free interpolation for a theory of arrays. In *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia* (2011), M. Schmidt-Schauß, Ed., vol. 10 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 171–186.

[25] Carioni, A., Ghilardi, S., and Ranise, S. Mcmt in the land of parametrized timed automata. In *VERIFY-2010. 6th International Verification Workshop* (2012), M. Aderhold, S. Autexier, and H. Mantel, Eds., vol. 3 of *EPiC Series in Computing*, EasyChair, pp. 47–64.

[26] Cavada, R., Cimatti, A., Griggio, A., and Susi, A. A formal ide for railways: Research challenges. In *Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops* (Cham, 2023), P. Masci, C. Bernardeschi, P. Graziani,

M. Koddenbrock, and M. Palmieri, Eds., Springer International Publishing, pp. 107–115.

[27] Chaki, S., Ouaknine, J., Yorav, K., and Clarke, E. Automated compositional abstraction refinement for concurrent c programs: A two-level approach. *Electronic Notes in Theoretical Computer Science 89*, 3 (2003), 417–432. SoftMC 2003, Workshop on Software Model Checking (Satellite Workshop of CAV '03).

[28] Chocron, P. D., Fontaine, P., and Ringeissen, C. Politeness and combination methods for theories with bridging functions. *J. Autom. Reason. 64*, 1 (2020), 97–134.

[29] Chou, C.-T., Mannava, P. K., and Park, S. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer-Aided Design* (Berlin, Heidelberg, 2004), A. J. Hu and A. K. Martin, Eds., Springer Berlin Heidelberg, pp. 382–398.

[30] Christ, J., Hoenicke, J., and Nutz, A. Smtinterpol: An interpolating smt solver. In *Model Checking Software* (Berlin, Heidelberg, 2012), A. Donaldson and D. Parker, Eds., Springer Berlin Heidelberg, pp. 248–254.

[31] Cimatti, A., and Griggio, A. Software model checking via ic3. In *Computer Aided Verification* (Berlin, Heidelberg, 2012), P. Madhusudan and S. A. Seshia, Eds., Springer Berlin Heidelberg, pp. 277–293.

[32] Cimatti, A., Griggio, A., Mover, S., Roveri, M., and Tonetta, S. Verification modulo theories. *Formal Methods Syst. Des. 60*, 3 (2022), 452–481.

[33] Cimatti, A., Griggio, A., Mover, S., and Tonetta, S. Infinite-state invariant checking with IC3 and predicate abstraction. *Formal Methods Syst. Des. 49*, 3 (2016), 190–218.

[34] Cimatti, A., Griggio, A., and Redondi, G. Universal invariant checking of parametric systems with quantifier-free smt reasoning. In *Automated Deduction – CADE 28* (Cham, 2021), A. Platzer and G. Sutcliffe, Eds., Springer International Publishing, pp. 131–147.

[35] Cimatti, A., Griggio, A., and Redondi, G. Verification of SMT systems with quantifiers. In *Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022, Proceedings* (2022), A. Bouajjani, L. Holík, and Z. Wu, Eds., vol. 13505 of *Lecture Notes in Computer Science*, Springer, pp. 154–170.

[36] Cimatti, A., Griggio, A., and Redondi, G. Towards the verification of a generic interlocking logic: Dafny meets parameterized model checking, 2024.

[37] Cimatti, A., Griggio, A., Schaafsma, B. J., and Sebastiani, R. The mathsat5 smt solver. In *TACAS'13* (Berlin, Heidelberg, 2013), TACAS'13, Springer-Verlag.

[38] Cimatti, A., Griggio, A., and Sebastiani, R. Efficient generation of craig interpolants in satisfiability modulo theories. *ACM Trans. Comput. Logic 12*, 1 (nov 2010).

[39] Cimatti, A., Griggio, A., and Tonetta, S. The VMT-LIB language and tools. *CoRR abs/2109.12821* (2021).

[40] Cimatti, A., Stojic, I., and Tonetta, S. Formal specification and verification of dynamic parametrized architectures. In *FM 2018* (2018).

[41] Claessen, K. New techniques that improve mace-style finite model finding.

[42] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement. In *Computer Aided Verification* (Berlin, Heidelberg, 2000), E. A. Emerson and A. P. Sistla, Eds., Springer Berlin Heidelberg, pp. 154–169.

[43] Clarke, E., Kroening, D., Ouaknine, J., and Strichman, O. Completeness and complexity of bounded model checking. In *Verification, Model Checking, and Abstract Interpretation* (Berlin, Heidelberg, 2004), B. Steffen and G. Levi, Eds., Springer Berlin Heidelberg, pp. 85–96.

[44] Clarke, E., Talupur, M., and Veith, H. Environment abstraction for parameterized verification. vol. 3855, pp. 126–141.

[45] Conchon, S., Goel, A., Krstic, S., Mebsout, A., and Zaïdi, F. Cubicle: A Parallel SMT-based Model Checker for Parameterized Systems. In *CAV 2012* (2012).

[46] Conchon, S., Goel, A., Krstic, S., Mebsout, A., and Zaïdi, F. Invariants for finite instances and beyond. In *Formal Methods in Computer-Aided Design, FMCAD 2013* (2013).

[47] Dams, D., and Grumberg, O. Abstraction and abstraction refinement. In *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, pp. 385–419.

[48] Daniel, J., Cimatti, A., Griggio, A., Tonetta, S., and Mover, S. Infinite-state liveness-to-safety via implicit abstraction and well-founded relations. In *Computer Aided Verification* (Cham, 2016), S. Chaudhuri and A. Farzan, Eds., Springer International Publishing, pp. 271–291.

[49] de Moura, L., and Bjørner, N. Efficient e-matching for smt solvers. In *Automated Deduction – CADE-21* (Berlin, Heidelberg, 2007), F. Pfenning, Ed., Springer Berlin Heidelberg, pp. 183–198.

[50] de Moura, L. M., and Bjørner, N. Z3: an efficient SMT solver. In *TACAS* (2008).

[51] Detlefs, D., Nelson, G., and Saxe, J. B. Simplify: a theorem prover for program checking. *J. ACM 52*, 3 (2005), 365–473.

[52] Dooley, M., and Somenzi, F. Proving parameterized systems safe by generalizing clausal proofs of small instances. In *CAV 2016* (2016).

[53] Drews, S., and Albarghouthi, A. Effectively propositional interpolants. In *Computer Aided Verification* (Cham, 2016), S. Chaudhuri and A. Farzan, Eds., Springer International Publishing, pp. 210–229.

[54] Emerson, E. A., and Kahlon, V. Reducing model checking of the many to the few. In *Automated Deduction - CADE-17* (Berlin, Heidelberg, 2000), D. McAllester, Ed., Springer Berlin Heidelberg, pp. 236–254.

[55] Esparza, J., Finkel, A., and Mayr, R. On the verification of broadcast protocols. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999* (1999), IEEE Computer Society, pp. 352–359.

[56] Fang, Y., Piterman, N., Pnueli, A., and Zuck, L. Liveness with invisible ranking. *International Journal on Software Tools for Technology Transfer 8* (06 2006), 261–279.

[57] Fantechi, A., Gori, G., Haxthausen, A. E., and Limbrée, C. Compositional verification of railway interlockings: Comparison of two methods. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification* (Cham, 2022), S. Collart-Dutilleul, A. E. Haxthausen, and T. Lecomte, Eds., Springer International Publishing, pp. 3–19.

[58] Farzan, A., and Kincaid, Z. Verification of parameterized concurrent programs by modular reasoning about data and control. *SIGPLAN Not. 47*, 1 (jan 2012), 297–308.

[59] Fedyukovich, G., Prabhu, S., Madhukar, K., and Gupta, A. Quantified invariants via syntax-guided synthesis. In *Computer Aided Verification* (Cham, 2019), I. Dillig and S. Tasiran, Eds., Springer International Publishing, pp. 259–277.

[60] Feldman, Y. M. Y., Padon, O., Immerman, N., Sagiv, M., and Shoham, S. Bounded quantifier instantiation for checking inductive invariants. *Log. Methods Comput. Sci.* (2019).

[61] Ferrari, A., and Beek, M. H. T. Formal methods in railways: A systematic mapping study. *ACM Comput. Surv. 55*, 4 (nov 2022).

[62] Ge, Y., Barrett, C., and Tinelli, C. Solving quantified verification conditions using satisfiability modulo theories. *Annals of Mathematics and Artificial Intelligence* (feb 2009), 101–122.

[63] Gheorghiu Bobaru, M., Păsăreanu, C. S., and Giannakopoulou, D. Automated assume-guarantee reasoning by abstraction refinement. In *Computer Aided Verification* (Berlin, Heidelberg, 2008), A. Gupta and S. Malik, Eds., Springer Berlin Heidelberg, pp. 135–148.

[64] Ghilardi, S., Gianola, A., Kapur, D., and Naso, C. Interpolation results for arrays with length and maxdiff. *ACM Trans. Comput. Log. 24*, 4 (2023), 28:1–28:33.

[65] Ghilardi, S., Nicolini, E., Ranise, S., and Zucchelli, D. Towards smt model checking of array-based systems. In *Automated Reasoning* (Berlin, Heidelberg, 2008), A. Armando, P. Baumgartner, and G. Dowek, Eds., Springer Berlin Heidelberg, pp. 67–82.

[66] Ghilardi, S., and Pagani, E. Higher-order quantifier elimination, counter simulations and fault-tolerant systems. *J. Autom. Reason. 65*, 3 (mar 2021), 425–460.

[67] Ghilardi, S., and Ranise, S. Mcmt: A model checker modulo theories. In *Automated Reasoning* (Berlin, Heidelberg, 2010), J. Giesl and R. Hähnle, Eds., Springer Berlin Heidelberg, pp. 22–29.

[68] Giannakopoulou, D., Namjoshi, K. S., and Pasareanu, C. S. Compositional reasoning. In *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, pp. 345–383.

[69] Gleissenthall, K. v., Bjørner, N., and Rybalchenko, A. Cardinalities and universal quantifiers for verifying parameterized systems. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2016), PLDI '16, Association for Computing Machinery, p. 599–613.

[70] Goel, A., and Sakallah, K. A. On symmetry and quantification: A new approach to verify distributed protocols. In *NFM 2021* (2021).

[71] Goel, A., and Sakallah, K. A. Towards an automatic proof of lamport's paxos. In *FMCAD 2021* (2021), IEEE, pp. 112–122.

[72] Griggio, A., and Jonáš, M. Kratos2: An smt-based model checker for imperative programs. In *Computer Aided Verification* (Cham, 2023), C. Enea and A. Lal, Eds., Springer Nature Switzerland, pp. 423–436.

[73] Gupta, A., McMillan, K. L., and Fu, Z. Automated assumption generation for compositional verification. In *Computer Aided Verification* (Berlin, Heidelberg, 2007), W. Damm and H. Hermanns, Eds., Springer Berlin Heidelberg, pp. 420–432.

[74] Gurfinkel, A., Shoham, S., and Meshman, Y. Smt-based verification of parameterized systems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2016), FSE 2016, Association for Computing Machinery, p. 338–348.

[75] Gurfinkel, A., Shoham, S., and Vizel, Y. Quantifiers on demand. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings* (2018), S. K. Lahiri and C. Wang, Eds., vol. 11138 of *Lecture Notes in Computer Science*, Springer, pp. 248–266.

[76] Hance, T., Heule, M., Martins, R., and Parno, B. Finding invariants of distributed systems: It's a small (enough) world after all. In *NSDI 2021* (2021), USENIX Association, pp. 115–131.

[77] Ivrii, A., Gurfinkel, A., and Belov, A. Small inductive safe invariants. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014* (2014), IEEE, pp. 115–122.

[78] Jovanović, D., and Dutertre, B. Property-directed k-induction. In *2016 Formal Methods in Computer-Aided Design (FMCAD)* (2016), pp. 85–92.

[79] Kaiser, A., Kroening, D., and Wahl, T. Dynamic cutoff detection in parameterized concurrent programs. In *Computer Aided Verification* (Berlin, Heidelberg, 2010), T. Touili, B. Cook, and P. Jackson, Eds., Springer Berlin Heidelberg, pp. 645–659.

[80] Kapur, D., Majumdar, R., and Zarba, C. G. Interpolation for data structures. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2006), SIGSOFT '06/FSE-14, Association for Computing Machinery, p. 105–116.

[81] Karbyshev, A., Bjørner, N., Itzhaky, S., Rinetzky, N., and Shoham, S. Property-directed inference of universal invariants or proving their absence. In *Computer Aided Verification* (2015), D. Kroening and C. S. Păsăreanu, Eds.

[82] Koenig, J., and Leino, R. Getting started with dafny: A guide.

[83] Koenig, J. R., Padon, O., Immerman, N., and Aiken, A. First-order quantified separators. In *PLDI* (2020).

[84] Komuravelli, A., Bjørner, N. S., Gurfinkel, A., and McMillan, K. L. Compositional verification of procedural programs using horn clauses over integers and arrays. In *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015* (2015), R. Kaivola and T. Wahl, Eds., IEEE, pp. 89–96.

[85] Komuravelli, A., Gurfinkel, A., and Chaki, S. Smt-based model checking for recursive programs. In *Computer Aided Verification* (Cham, 2014), A. Biere and R. Bloem, Eds., Springer International Publishing, pp. 17–34.

[86] Kovács, L., and Voronkov, A. First-order theorem proving and vampire. In *CAV 2013,* (2013).

[87] Kroening, D., and Strichman, O. *Decision Procedures: An Algorithmic Point of View*. Springer, 2016.

[88] Krstic, S. Parametrized system verification with guard strengthening and parameter abstraction.

[89] Kuncak, V., Nguyen, H. H., and Rinard, M. An algorithm for deciding bapa: Boolean algebra with presburger arithmetic. In *Automated Deduction – CADE-20* (Berlin, Heidelberg, 2005), R. Nieuwenhuis, Ed., Springer Berlin Heidelberg, pp. 260–277.

[90] Lahiri, S. K., Ball, T., and Cook, B. Predicate abstraction via symbolic decision procedures. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings* (2005), K. Etessami and S. K. Rajamani, Eds., vol. 3576 of *Lecture Notes in Computer Science*, Springer, pp. 24–38.

[91] Lahiri, S. K., and Bryant, R. E. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Logic 9*, 1 (dec 2007), 4–es.

[92] Lahiri, S. K., Nieuwenhuis, R., and Oliveras, A. Smt techniques for fast predicate abstraction. In *Computer Aided Verification* (Berlin, Heidelberg, 2006), T. Ball and R. B. Jones, Eds., Springer Berlin Heidelberg, pp. 424–437.

[93] Li, Y., Duan, K., Jansen, D. N., Pang, J., Zhang, L., Lv, Y., and Cai, S. An automatic proving approach to parameterized verification. *ACM Trans. Comput. Logic* (Nov. 2018).

[94] Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., and Bensalem, S. Property preserving abstractions for the verification of concurrent systems. *Form. Methods Syst. Des. 6*, 1 (jan 1995), 11–44.

[95] Ma, H., Goel, A., Jeannin, J.-B., Kapritsos, M., Kasikci, B., and Sakallah, K. A. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *SOSP '19* (2019).

[96] Mann, M., Irfan, A., Griggio, A., Padon, O., and Barrett, C. W. Counterexample-guided prophecy for model checking modulo the theory of arrays. *CoRR abs/2101.06825* (2021).

[97] McCarthy, J. *Towards a Mathematical Science of Computation.* Springer Netherlands, Dordrecht, 1993, pp. 35–56.

[98] McMillan, K. L. Interpolation and sat-based model checking. In *Computer Aided Verification* (Berlin, Heidelberg, 2003), W. A. Hunt and F. Somenzi, Eds., Springer Berlin Heidelberg, pp. 1–13.

[99] McMillan, K. L. Applications of craig interpolants in model checking. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2005), N. Halbwachs and L. D. Zuck, Eds., Springer Berlin Heidelberg, pp. 1–12.

[100] McMillan, K. L. Lazy abstraction with interpolants. In *Computer Aided Verification* (Berlin, Heidelberg, 2006), T. Ball and R. B. Jones, Eds., Springer Berlin Heidelberg, pp. 123–136.

[101] McMillan, K. L. Interpolants from Z3 proofs. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011* (2011), P. Bjesse and A. Slobodová, Eds., FMCAD Inc., pp. 19–27.

[102] McMillan, K. L. Eager abstraction for symbolic model checking. In *Computer Aided Verification* (Cham, 2018), H. Chockler and G. Weissenbacher, Eds., Springer International Publishing, pp. 191–208.

[103] McMillan, K. L. *Interpolation and Model Checking*. Springer International Publishing, Cham, 2018, pp. 421–446.

[104] Nelson, G., and Oppen, D. C. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst. 1*, 2 (oct 1979), 245–257.

[105] Nelson, G., and Oppen, D. C. Fast decision procedures based on congruence closure. *J. ACM 27* (1980), 356–364.

[106] Nonnengart, A., and Weidenbach, C. Computing small clause normal forms. In *Handbook of Automated Reasoning* (2001).

[107] Padon, O. Deductive verification of distributed protocols in first-order logic. In *2018 Formal Methods in Computer Aided Design (FMCAD)* (2018), pp. 1–1.

[108] Padon, O. Invited talk: Deductive verification of distributed protocols in decidable logics. In *Proceedings of the 21st International Workshop on Satisfiability Modulo Theories (SMT 2023) co-located with the 29th International Conference on Automated Deduction (CADE 2023), Rome, Italy, July, 5-6, 2023* (2023), S. Graham-Lengrand and M. Preiner, Eds., vol. 3429 of *CEUR Workshop Proceedings*, CEUR-WS.org, p. 1.

[109] Padon, O., Hoenicke, J., Losa, G., Podelski, A., Sagiv, M., and Shoham, S. Reducing liveness to safety in first-order logic. *Proc. ACM Program. Lang. 2*, POPL (dec 2017).

[110] Padon, O., Immerman, N., Shoham, S., Karbyshev, A., and Sagiv, M. Decidability of inferring inductive invariants. *SIGPLAN Not. 51*, 1 (jan 2016), 217–231.

[111] Padon, O., Losa, G., Sagiv, M., and Shoham, S. Paxos made EPR: decidable reasoning about distributed protocols. *Proc. ACM Program. Lang. 1*, OOPSLA (2017), 108:1–108:31.

[112] Padon, O., McMillan, K. L., Panda, A., Sagiv, M., and Shoham, S. Ivy: Safety verification by interactive generalization. *SIGPLAN Not. 51*, 6 (June 2016), 614–630.

[113] Padon, O., Wilcox, J. R., Koenig, J. R., McMillan, K. L., and Aiken, A. Induction duality: Primal-dual search for invariants. *POPL 6*, POPL (2022).

[114] Pnueli, A., Ruah, S., and Zuck, L. D. Automatic deductive verification with invisible invariants. In *TACAS* (2001).

[115] Redondi, G., Cimatti, A., Griggio, A., and McMillan, K. Invariant checking for smt-based systems with quantifiers, 2024.

[116] Reynolds, A. Quantifier instantiation beyond e-matching. In *(CAV 2017)* (2017), M. Brain and L. Hadarean, Eds.

[117] Reynolds, A., Tinelli, C., and de Moura, L. Finding conflicting instances of quantified formulas in smt. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design* (Austin, Texas, 2014), FMCAD '14, FMCAD Inc, p. 195–202.

[118] Reynolds, A., Tinelli, C., Goel, A., and Krstić, S. Finite model finding in smt. In *Computer Aided Verification* (Berlin, Heidelberg, 2013), N. Sharygina and H. Veith, Eds., Springer Berlin Heidelberg, pp. 640–655.

[119] Rümmer, P. Competition report: Chc-comp-20. *Electronic Proceedings in Theoretical Computer Science 320* (Aug. 2020), 197–219.

[120] Sanchez, A., Sankaranarayanan, S., Sanchez, C., and Chang, B.-Y. Invariant generation for parametrized systems using self-reflection. vol. 7460, pp. 146–163.

[121] Shoham, S., and Grumberg, O. Compositional verification and 3-valued abstractions join forces. *Information and Computation 208*, 2 (2010), 178–202.

[122] Stump, A., Barrett, C. W., Dill, D. L., and Levitt, J. A decision procedure for an extensional theory of arrays. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS '01)* (June 2001), IEEE Computer Society, pp. 29–37. Boston, Massachusetts.

[123] Suter, P., Steiger, R., and Kuncak, V. Sets with cardinality constraints in satisfiability modulo theories. In *Verification, Model Checking, and Abstract Interpretation* (Berlin, Heidelberg, 2011), R. Jhala and D. Schmidt, Eds., Springer Berlin Heidelberg, pp. 403–418.

[124] Talupur, M., and Tuttle, M. R. Going with the flow: Parameterized verification using message flows. In *2008 Formal Methods in Computer-Aided Design* (2008), pp. 1–8.

[125] Tamir, O., Taube, M., McMillan, K. L., Shoham, S., Howell, J., Gueta, G., and Sagiv, M. Counterexample driven quantifier instantiations with applications to distributed protocols. *Proc. ACM Program. Lang. 7*, OOPSLA2 (2023), 1878–1904.

[126] Tinelli, C., and Barrett, C. Satisfiability modulo theories: Introduction and applications. *Communications of the ACM 54*, 9 (2011), 69–77.

[127] Tonetta, S. Abstract model checking without computing the abstraction. In *FM 2009: Formal Methods* (Berlin, Heidelberg, 2009), A. Cavalcanti and D. R. Dams, Eds., Springer Berlin Heidelberg, pp. 89–105.

[128] Vick, C., and McMillan, K. L. Synthesizing history and prophecy variables for symbolic model checking. In *Verification, Model Checking, and Abstract Interpretation - 24th International Conference, VMCAI 2023, Boston, MA, USA, January 16-17, 2023, Proceedings* (2023), C. Dragoi, M. Emmi, and J. Wang, Eds., vol. 13881 of *Lecture Notes in Computer Science*, Springer, pp. 320–340.

[129] Yao, J., Tao, R., Gu, R., Nieh, J., Jana, S., and Ryan, G. DistAI: Data-Driven automated invariant learning for distributed protocols. In *(OSDI 21)* (July 2021).

[130] Zuck, L. D., and McMillan, K. L. Invisible invariants are neither. In *From Reactive Systems to Cyber-Physical Systems* (2019).