Hardware Model Checking Algorithms and Techniques

algorithms

MDPI

Review

Hardware Model Checking Algorithms and Techniques

Gianpiero Cabodi [1,*], Paolo Enrico Camurati [1], Marco Palena [2] and Paolo Pasini [3,*]

[1] DAUIN, Department of Control and Computer Engineering, Politecnico di Torino, 10129 Turin, Italy; paolo.camurati@polito.it
[2] CNIT, National Inter-University Consortium for Telecommunications, 10129 Turin, Italy; marco.palena@cnit.it
[3] DET, Department of Electronics and Telecommunications, Politecnico di Torino, 10129 Turin, Italy
* Correspondence: gianpiero.cabodi@polito.it (G.C.); paolo.pasini@polito.it (P.P.); Tel.: +39-011-090-7082 (G.C.)

Abstract: Digital systems are nowadays ubiquitous and often comprise an extremely high level of complexity. Guaranteeing the correct behavior of such systems has become an ever more pressing need for manufacturers. The correctness of digital systems can be addressed resorting to formal verification techniques, such as model checking. Currently, it is usually impossible to determine a priori the best algorithm to use given a verification task and, thus, portfolio approaches have become the de facto standard in model checking verification suites. This paper describes the most relevant algorithms and techniques, at the foundations of bit-level SAT-based model checking itself.

Keywords: formal verification; model checking; SAT; Boolean functions

1. Introduction

Ranging from commodity devices to business or safety critical environments, digital systems have proliferated into most aspects of our daily lives. Moreover, thanks to Moore's law and the advent of high-level synthesis, contemporary hardware designs often comprise an extremely high level of complexity. As a direct consequence, design verification has become one of the most relevant aspects of the design and production flow. In such a scenario, guaranteeing the correct behavior of digital systems, with respect to their specification, is becoming an ever more pressing need for manufacturers [1]. The increasing complexity of hardware designs, together with the demand for short development cycles imposed by ever evolving markets, makes the delivery of defect-free systems an extremely challenging task. This is particularly true when considering sequential designs, in which the behavior of the system depends not only on the inputs applied at a given time but also on the state the system finds itself in. Modern systems are characterized by the presence of several million elements, those being transistors, logic gates, memory elements or other things. In such a scenario, design verification can represent one of the most time consuming, yet necessary, activities to be performed, with reports purporting it in the range between 40% and 70% of the entire cycle's endeavor [2]. Oftentimes, the usual testing techniques are not enough to prove the correctness of a whole system in a timely fashion, and further problems arise when taking into account the issue of transferring theoretical results to industrial practice [3].

Traditionally, hardware designs are verified through *simulation*. In simulation, a model of the system is solicited by a set of stimuli while the resulting behavior is checked against the expected behavior. Simulation is scalable, being applicable to designs of virtually any size, but is also costly and incomplete. In fact, it is often unfeasible to completely cover the behaviors of a realistic design through simulation. The incompleteness of simulation is often aggravated by the increasing complexity of hardware designs, which not only increases the number of behaviors to check but may also introduce corner cases difficult to cover. Starting from the late 1980s, *formal hardware verification* has become an attractive approach to overcome the coverage limitations of simulation. Formal verification is the use

Citation: Cabodi, G.; Camurati, P.E.; Palena, M.; Pasini, P. Hardware Model Checking Algorithms and Techniques. *Algorithms* **2024**, *17*, 253. https://doi.org/10.3390/a17060253

Academic Editor: Jesper Jansson

Received: 8 May 2024
Revised: 5 June 2024
Accepted: 7 June 2024
Published: 9 June 2024

of the formal methods of mathematics and logic to prove (or disprove) the correctness of a design with respect to a given formal specification of its expected behavior. During the last two decades, research in the field of formal verification has led to the development of many techniques. Such techniques can be subdivided into three main approaches to formal verification: *theorem proving* [4,5], *equivalence checking* [6] and *model checking* [7,8].

Model checking is the most widely used formal verification approach for sequential hardware designs. In model checking, a *model* of the design and a formalized representation of its specification are formulated using some mathematically precise and unambiguous language. Then, an algorithmic procedure automatically checks whether or not the model meets the formal specification. Such a procedure exhaustively traverses the modeled behaviors of the system and either confirms that the system behaves correctly or produces a trace that demonstrates a violating behavior (called a *counterexample*). The main advantages of model checking are its fully automated nature and its ability to produce counterexamples. Usually, the design to be verified is modeled as a *transition system*, comprising states and transitions between states, and the specification is formalized by writing *temporal logic properties*. Invariant verification is a particular model checking task in which the system specification is described as an invariant property that must hold true in every reachable state of the model.

From an algorithmic standpoint, in the beginning, model checking was based on explicit state enumeration; thus, its practical application was severely hindered by mere state space representation problems. The introduction of BDD-based representation [9] was the turning point for model checking applicability to industrial-level scenarios. Despite the huge impact of such an approach to model checking, BDD-based representations still suffered from unpredictable memory requirements and, thus, additional paths of research opened up in order to overcome such limits, such as the introduction of SAT-based approaches.

Today, model checking can exploit explicit state enumeration or symbolic state enumeration, via BDD- or SAT-based representations. Furthermore, solutions based on automatic test pattern generations are admissible. Each of these, in turn, has led to a number of distinct model checking algorithms and techniques.

Up until today, there is not a concrete winner among all the available techniques. The most reliable approach is obtained from running in parallel several of these algorithms, exploiting multi-core environments, and halting verification as soon as one of the engines has found a solution to the verification problem at hand [10,11].

Complementary to the verification phase itself, there is then the additional problem of certifying checkers' results, in order to better support their acquisition in complete design cycles scenarios [12,13].

A recent trend in hardware model checking is to exploit the ability of SMT solvers to perform word-level reasoning based on bit-vectors. This trend is demonstrated by the introduction of word-level tracks in recent editions of the Hardware Model Checking Competition [14]. However, word-level model checking shines for problems with memory modeled with arrays; bit-level solvers are currently the de facto standard in the industry and prove to be still superior on many word-level designs after bit-blasting [15].

This paper (part of the contents of this paper are a redacted version of the introductory chapters of the PhD thesis of two of its authors, Marco Palena [16] and Paolo Pasini [17], which have not been published before besides diffusion within the Politecnico di Torino and the respective thesis defense committees) describes the most relevant algorithms and techniques targeting explicitly bit-level SAT-based hardware model checking, in a formal and theoretically sound fashion. A description of complementary topics and a broader overview of the vast field of model checking and satisfiability is provided in [18,19]. A description of topics concerning model checking, with additional examples and applications, is provided in [8]. Complementary to previous works, a specific focus on the design and verification of cyber-physical systems is provided in [20].

Outline

The rest of the paper is organized as follows:

- In Section 2, we provide some preliminary notions with the purpose of presenting the adopted notation and terminology and make this work self-contained. In particular, we first provide some general concepts from the field of logic and computer science and then we discuss in more depth the necessary concept to contextualize model checking in detail.
- In Sections 3–7, we present a complete description of the most relevant bit-level SAT-based hardware model checking algorithms.
- In Section 8, we present a practical evaluation of model checking algorithms over industrial-level benchmarks.
- In Section 9, we summarize the current work and provide some insights concerning future developments in the field.

2. Background and Notation

In this section, we introduce some notions needed to understand the rest of the paper. In particular, we provide some basic concepts and definitions from the fields of logic and computer science relevant to the topics at hand.

2.1. Mathematical Logic

Model checking aims at verifying whether a finite-state model of a system meets a given specification. In order to solve such a problem algorithmically, both the model of the system and its specification need to be expressed in some precise mathematical language. The fields of mathematical logic and formal language theory provide us the tools we need to reason about the system states in a symbolical and rigorous manner. Mathematical logic is the systematic study of reasoning. By reasoning, we intend the abstract capability to provide *arguments* supporting the fact that a given statement, called the *conclusion*, follows from some other statements, called the *premises*. The steps through which conclusions are reached from premises are termed *inferences*. Mathematical logic relies on *formal languages* in order to provide a symbolic representation of abstract concepts, like statements about the model of the system we are interested to reason about, that can then be manipulated through formal, mathematical processes. We will rely on common concepts stemming from logic, and we will specifically introduce and define just the ones that are more relevant to this work. We refer the interested reader to [21,22] for a comprehensive exposition of those subjects.

2.2. Formal Languages

A formal language is a mathematical object that describes the syntax of a set of finite *strings* of *symbols*. A formal language is a purely syntactic tool and can be completely defined without making any reference to an interpretation for its symbols. In the following, we provide some basic notions from the field of *formal language theory* (we refer the interested reader to [23] for an more in depth discussion on the subject).

Definition 1 (Alphabet and Words)**.** *An alphabet* Σ *is any non-empty finite set of symbols. A word over* Σ *is any finite or infinite string of symbols in* Σ*. The set of all finite words over* Σ *is denoted by* Σ^* *(called the Kleene closure of* Σ*).*

Definition 2 (Formal Languages)**.** *Given an alphabet* Σ*, a formal language* $\mathcal{L}$ *over* Σ *is any set of finite words over* Σ*, i.e.,* $\mathcal{L} \subseteq \Sigma^*$*.*

A formal language $\mathcal{L}$ is often described by a set of rules that determine how its words are formed (*rules of formation*). Many formalisms can be used to describe the rules of formation of a formal language, such as formal grammars, regular expressions and automata [23].

Definition 3 (Well-formed Formulas)**.** *Given a formal language $\mathcal{L}$ over Σ, a word $w \in \Sigma^*$ is said to be a well-formed formula of $\mathcal{L}$, often abbreviated with* wff*, iff $w \in \mathcal{L}$.*

Note that a formal language can be identified with the set of its well-formed formulas. Throughout the rest of this paper, we simply use the term *formula* to refer to a well-formed formula of a given formal language.

Formal languages are entirely syntactic in nature but may be given interpretations that provide meaning to their symbols and sentences. Given a formal language $\mathcal{L}$, from the semantic standpoint, the symbols of $\mathcal{L}$ may be subdivided into logical symbols and non-logical symbols.

Definition 4 (Logical Symbols)**.** *A symbol of a formal language is said to be a logical symbol if it always has the same meaning, independently from the language interpretation.*

Examples of logical symbols in the context of propositional and first-order logic are logical connectives ($\neg$, $\wedge$, $\vee$, etc.), quantifiers ($\forall$, $\exists$, etc.) and the equality predicate ($=$).

Definition 5 (Non-logical Symbols and Signature)**.** *A symbol of a formal language is said to be a non-logical symbol if it has a meaning only when one is assigned to it by means of a language interpretation. Given a formal language $\mathcal{L}$, its signature σ is defined as the set of its non-logical symbols.*

Note that non-logical symbols may have different meanings under different language interpretations. Examples of non-logical symbols in the context of propositional and first-order logic are variables and symbols for constants, function and relation.

A fundamental problem in formal language theory is that of determining the membership of a given word in a formal language.

Definition 6 (Membership Problem)**.** *Given a formal language $\mathcal{L}$ and a word $w \in \Sigma^*$, the membership problem is the problem to determine whether $w \in \mathcal{L}$, i.e., whether w is a wff of $\mathcal{L}$.*

As we show later in Section 2.6, each problem in the important computational class of *decision problems* can be represented as a membership problem for a formal language. SAT-based model checking algorithms encode state reachability problems, defined on a model of the system, as instances of a particular decision problem known as Boolean Satisfiability (SAT).

2.3. Logical Systems

A fundamental concept in mathematical logic is that of *logical consequence*, describing the relationship between statements that hold true when one statement, called the *conclusion*, logically follows from one or more other statements, called the *premises*. Historically, the concept of logical consequence has been studied from two different but interrelated perspectives: syntactic and semantic.

Proof theory is the branch of logic that studies logical consequence from the syntactic standpoint. The formalism of *formal systems* is used to model deductions (a *deduction* is a kind of inference that is always valid, i.e., its conclusion is always a logical consequence of its premises) as syntactical transformations (*inference rules*) applied to strings of symbols (*formulas*). In this context, the concept of logical consequence is related to the ability to derive a *proof*, that is, a sequence of syntactic transformations that allow us to derive the conclusion from the premises.

Model theory is the branch of logic that studies logical consequence from the semantic standpoint. Semantics is about associating a meaning with the well-formed formulas of some formal language. Formulas are given a meaning by relating their constituents to elements of the *domain of discourse* by means of an *interpretation*. Given an interpretation of the symbols of a formal language, each formula can be related to the concept of *truth* based on whether or not its meaning holds true in the domain of discourse, i.e., the world we

are interested to reason about. The set of all possible interpretations of a formal language define a *formal semantic* for that language. In this context, the concept of logical consequence is related to whether or not a conclusion would be evaluated as true under all the possible interpretations that evaluate the premises as true.

Even if the syntactic and semantic notions of logical consequence have different definitions, the concepts from the two branches of logic are systematically related. We give the following definitions.

Definition 7 (Logical System)**.** *Given a formal language $\mathcal{L}$, a logical system (or logic) for $\mathcal{L}$ is defined as a pair $\mathscr{L} \stackrel{def}{=} \langle \mathcal{S}, \mathcal{V} \rangle$, consisting of a formal system $\mathcal{S}$ and a formal semantics $\mathcal{V}$ for $\mathcal{L}$.*

Definition 8 (Soundness and Completeness)**.** *Given a formal language $\mathcal{L}$, a logical system $\mathscr{L} = \langle \mathcal{S}, \mathcal{V} \rangle$ for $\mathcal{L}$ is sound iff every formula that can be derived from $\mathcal{S}$ is valid according to $\mathcal{V}$. A logical system $\mathscr{L}$ is complete iff every formula that is valid according to $\mathcal{V}$ can be derived from $\mathcal{S}$.*

Therefore, in a sound and complete logical system the concepts of semantic and syntactic logical consequence coincide. The same is true for the semantic and syntactic versions of validity, consistency, necessity and impossibility.

2.4. Propositional Logic

Propositional logic (or *Boolean logic*) is a sound and complete logical system concerned with the study of propositions. A *proposition* is an abstract entity bearing a truth value that is expressed as a statement on a domain of discourse. Boolean logic considers (complex) propositions to be composed of *atomic propositions* connected by means of *logical operators*. Atomic propositions are propositions whose structure cannot be further decomposed in terms of logical operators. Later, in Section 2.7, we will use Boolean logic to provide a symbolical representation of hardware-derived state transition systems at the bit level.

2.4.1. Syntax of Propositional Logic

From the syntactic standpoint, the language of propositional logic consists of a set of non-logical symbols called *propositional variables* and a set of logical symbols, comprising *logical connectives* ($\neg$, $\wedge$ and $\vee$) and *logical constants* ($\top$ and $\bot$).

Definition 9 (Syntax of Propositional Formulas)**.** *Given a set of propositional variables AP, the syntax of propositional formulas is defined inductively as follows:*

- *$\top$ and $\bot$ are propositional formulas.*
- *Each $a \in AP$ is a propositional formula.*
- *Given F, a propositional formula, $\neg F$ is a propositional formula.*
- *Given F_1 and F_2, propositional formula $F_1 \wedge F_2$ is a propositional formula.*
- *Given F_1 and F_2, propositional formula $F_1 \vee F_2$ is a propositional formula.*

Note that $\bot$ and $\vee$ can be omitted from Definition 9 without losing expressiveness as they can be easily derived from $\top$, $\neg$ and $\wedge$. Parentheses "(" and ")" can be used to improve the readability of formulas. We also provide the following definitions about propositional formulas.

Definition 10 (Atomic Formulas)**.** *An atomic formula is a propositional formula consisting of a propositional variable only.*

Definition 11 (Support)**.** *Given a propositional formula F, we call support of F, denoted by $\mathrm{Vars}(F)$, the set of propositional variables occurring in F. Accordingly, given a set of propositional formulas Φ, we call support of Φ, denoted by $\mathrm{Vars}(\Phi)$, the set of propositional variables occurring in at least one formula of Φ:*

$$\text{Vars}(\Phi) = \bigcup_{F_i \in \Phi} \text{Vars}(F_i) \tag{1}$$

2.4.2. Semantics of Propositional Logic

Intuitively, from the semantic point of view, propositional variables represent atomic propositions, logical constants represent special propositions that are either always true ($\top$) or always false ($\bot$) and logical connectives represent standard Boolean functions such as logical negation ($\neg$), logical conjunction ($\wedge$) and logical disjunction ($\vee$).

Each propositional variable is interpreted as an atomic proposition about the domain of discourse which can either be true or false. Such an interpretation is described by means of a *truth assignment* defined as follows.

Definition 12 (Truth Assignments)**.** *Given a set of propositional variables $A \subseteq AP$, a truth assignment $\mu : A \to \mathbb{B}$ is a function mapping each propositional variable $a \in A$ to a truth value in $\mathbb{B}$.*

Note that, from the model-theoretic perspective, truth assignments can be seen as equivalent representations of structures along with a definition of truth. Since the only non-logical symbols of propositional logic are propositional variables, a structure for the propositional language is an interpretation of those variables as atomic propositions on the domain of discourse. A definition of truth then maps each atomic proposition to a truth value. The composition of the two is equivalent to a truth assignment.

Definition 13 (Complete or Partial Truth Assignments)**.** *Given a propositional formula F, a complete truth assignment for F is a function $\mu : \text{Vars}(F) \to \mathbb{B}$ assigning a truth value to each variable of F. A partial truth assignment for F is a function $\mu : A \subset \text{Vars}(F) \to \mathbb{B}$ assigning a truth value to a subset A of the variables of F.*

Each logical connective is given a *truth-functional interpretation*, mapping it to a specific *Boolean function*. Logical connectives are therefore thought as having a certain *arity*, according to the Boolean function they represent: logical conjunction and disjunction are binary connectives whereas negation is a unary connective. Under its truth-functional interpretation, each logical connective of arity n corresponds to a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ mapping the truth value of its operands to the truth value of its result. The truth-functional interpretation of logical connectives is usually described by means of truth tables. Additional logical connectives, representing other standard Boolean functions such as implication ($\to$), bi-implication ($\leftrightarrow$) and exclusive disjunction ($\oplus$), may be included in the language of propositional logic according to their usual semantics and and syntactic definition, following.

$$F_1 \to F_2 \stackrel{\text{def}}{=} \neg F_1 \vee \neg F_2 \tag{2}$$

$$F_1 \leftrightarrow F_2 \stackrel{\text{def}}{=} (\neg F_1 \wedge \neg F_2) \vee (F_1 \wedge F_2) \tag{3}$$

$$F_1 \oplus F_2 \stackrel{\text{def}}{=} (\neg F_1 \wedge F_2) \vee (F_1 \wedge \neg F_2) \tag{4}$$

As a way of reducing the number of necessary parentheses, the following precedence order between logical operators is usually applied (from highest precedence to lowest precedence): $\neg$, $\wedge$, $\vee$, $\to$, $\leftrightarrow$ and $\oplus$.

Each propositional formula, therefore, unambiguously represents a given Boolean function, which can be determined combining the function associated with its logical connectives according to the precedence order.

Definition 14 (Semantics of Propositional Logic)**.** *Given a set of propositional variables AP, a propositional formula F over $\text{Vars}(F)$ and a truth assignment μ over $A \subseteq AP$ with $\text{Vars}(F) \subseteq A$,*

we give the following inductive definition of when μ satisfies *F (or equivalently, F evaluates to* $\top$ *under* μ*), denoted as* $\mu \models F$*:*

- $\mu \models \top$ *and* $\mu \not\models \bot$.
- *For each* $a \in \text{Vars}(F)$, $\mu \models a$ *iff* $\mu(a) = \top$.
- $\mu \models \neg F$ *iff* $\mu \not\models F$.
- $\mu \models F_1 \wedge F_2$ *iff* $\mu \models F_1$ *and* $\mu \models F_2$.
- $\mu \models F_1 \vee F_2$ *iff* $\mu \models F_1$ *or* $\mu \models F_2$.

The semantics of other logical connectives can easily be inferred from their truth tables. We provide the following useful definitions:

Definition 15 ((Propositional) Models)**.** *Given a propositional formula F, a truth assignment $\mu : \text{Vars}(F) \to \mathbb{B}$ is a model (or satisfying assignment) of F iff $\mu \models F$.*

Definition 16 ((Propositional or Boolean) Satisfiability)**.** *A propositional formula F is said to be satisfiable iff it has at least one model, i.e., there exists μ such that $\mu \models F$. Otherwise, F is said to be unsatisfiable.*

Definition 17 ((Propositional) Consequence)**.** *Given a set of propositional formulas Φ and a propositional formula F such that $\text{Vars}(F) \subseteq \text{Vars}(\Phi)$, F is said to be a consequence of Φ iff every model of Φ can be restricted to a model for F, denoted as $\Phi \models F$.*

Definition 18 ((Propositional) Equivalence)**.** *Given two propositional formulas F and G such that $\text{Vars}(F) = \text{Vars}(G)$, F is said to be equivalent to G iff every model of F is also a model of G and vice versa, denoted as $F \equiv G$.*

Definition 19 ((Propositional) Validity)**.** *An argument from a set of propositional formulas Φ to a conclusion F is valid iff $\Phi \models F$.*

Definition 20 ((Propositional) Consistency)**.** *A set of propositional formulas Φ is consistent iff there is a complete truth assignment μ that is a model for every formula of Φ.*

Definition 21 (Valid Propositional Formulas)**.** *A propositional formula F is said to be a valid (or a tautology), denoted as $\models F$, iff every truth assignment over its variables is a model for it.*

Definition 22 (Strength of Propositional Formulas)**.** *Given two propositional formulas F and G, if $F \models G$ then F is said to be stronger than G and, conversely, G is said to be weaker than F. From the semantics of implication ($\to$), it is clear that, if $\models F \to G$, then $F \models G$ and, thus, F is stronger than G.*

We also provide the following remarks about notation for models and truth assignment.

Notation 1 (Formulas as Set of Models)**.** *Given a propositional formula F, we denote with $\text{Mods}(F)$ the set of all models of F. Any propositional formula F can be seen as a representation of its set of models $\text{Mods}(F)$.*

Any truth assignment can be represented as a formula satisfied by it or the set of propositional variables it assigns to $\top$.

Notation 2 (Restriction of Models)**.** *Given a propositional formula F, a truth assignment $\mu : A \to \mathbb{B}$, with $\text{Vars}(F) \subset A$ can be restricted to a model of F iff its restriction $\mu|_{\text{Vars}(F)} \models F$.*

Notation 3 (Truth Assignments as Formulas). *A truth assignment μ over a set of propositional variables $A \in AP$ can be represented as the following propositional formula:*

$$F_\mu \stackrel{def}{=} \bigwedge_{a_i \in A} l_i \text{ where } \begin{cases} l_i = a_i, & \text{iff } \mu(a_i) = \top \\ l_i = \neg a_i, & \text{iff } \mu(a_i) = \bot \end{cases} \tag{5}$$

Notation 4 (Truth Assignments as Sets of Propositional Variables). *A truth assignment μ over a set of propositional variables $A \in AP$ can be represented by the following set of propositional variables:*

$$A_\mu \stackrel{def}{=} \{a \in A \mid \mu(a) = \top\} \tag{6}$$

Vice versa, a set of propositional variables $A \in AP$ induces a truth assignment μ_A equal to the characteristic function of A, i.e.:

$$\mu_A(a) = \begin{cases} \top & \text{if } a \in A \\ \bot, & \text{otherwise} \end{cases} \tag{7}$$

Following Notation 4, the satisfaction relation for propositional logic ($\models$) can be extended to sets of propositional variables: $A \models \varphi$ if and only if $\mu_A \models \varphi$.

2.4.3. Conjunctive Normal Form

We provide the following definitions about propositional formulas.

Definition 23 (Literals, Clauses and Cubes). *A* literal *is either an atomic formula or the negation of an atomic formula. A* clause *is a disjunction of literals, whereas a* cube *is a conjunction of literals.*

Definition 24 (Conjunctive Normal Form). *A formula F is said to be in Conjunctive Normal Form (CNF) if is is a conjunction of clauses.*

For instance, a, b, c and d being propositional variables, a is an atomic formula, both $\neg a$ and b are literals, $(\neg a \vee b \vee \neg c)$ is a clause, $(\neg a \wedge b \wedge \neg c)$ is a cube and $(\neg a \vee b \vee \neg c) \wedge (\neg b \vee \neg c \vee d)$ is a CNF formula. Given a truth assignment μ is easy to see the following:

- A literal a is satisfied by μ iff $\mu(a) = \top$.
- A literal $\neg a$ is satisfied by μ iff $\mu(a) = \bot$.
- A clause $C = l_0 \vee \cdots \vee l_n$ is satisfied by μ iff $\mu \models l_i$ for some $0 \leq i \leq n$.
- A cube $Q = l_0 \wedge \cdots \wedge l_n$ is satisfied by μ iff $\mu \models l_i$ for each $0 \leq i \leq n$.
- A CNF formula $F = C_0 \wedge \cdots \wedge C_n$ is satisfied by μ iff $\mu \models C_i$ for each $0 \leq i \leq n$.

2.5. Boolean Functions

In this paper we are interested in verifying the properties of hardware digital systems. Such systems can be thought to be made from large assemblies of logic gates which, in turn, are simple electronic implementations of Boolean functions. As described in Section 2.4, formulas of Propositional Logic can be used to represent Boolean functions and reason about them. In this section, we introduce some basic notions about Boolean functions.

Definition 25 (Boolean Function). *A Boolean function is a function $f : \mathbb{B}^n \to \mathbb{B}$, where $\mathbb{B} = \{\top, \bot\}$ is the Boolean domain and $n \in \mathbb{N}$ identifies the arity of the function.*

Definition 26 (Functions Monotone in a Variable). *Given an n-ary Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ and a value $k \in \mathbb{N}$, with $1 \leq k \leq n$, we say that f is monotone in the variable x_k if $f|_{x_k=\bot} \to f|_{x_k=\top}$. Thus, when f is monotone in x_k, changing the value of x_k from $\bot$ to $\top$ cannot change the value of f from $\top$ to $\bot$. The definition here provided of monotonicity in a given variable is also known as unateness or positive monotonicity in that variable.*

Definition 27 (Monotone Functions)**.** *A Boolean function f is monotone if it is monotone in each of its variables. Thus, changing the value of any of its variables from $\bot$ to $\top$ cannot change the value of f from $\top$ to $\bot$.*

2.6. Decision Problems

In Section 2.1, we have presented logic as a tool to formalize and reason about a world of interest, whereas in Section 2.5 we have provided some background knowledge about commonly used models for representing combinational and sequential hardware circuits. Our aim is to use the tool of logic, and in particular Propositional Logic, to provide a formal representation of hardware sequential systems and reason about some of their properties. In particular, we are interested in asking certain questions about the formalized systems and having those questions answered by algorithmic procedures. We are, thus, interested in solving some computational problems arising from those questions, which typically belong to the class of *decision problems*.

In this section we provide an overview of decision problems and their properties and we present Boolean Satisfiability (SAT), a fundamental decision problem in computer science of particular interest in the context of this paper.

Definition 28 (Decision Problems and Instances)**.** *A decision problem $\mathcal{P}$ is any arbitrary question admitting a YES or NO answer over an infinite set of inputs. An instance of a decision problem $\mathcal{P}$ is a particular input for $\mathcal{P}$. We distinguish between* YES*-instances, instances for which $\mathcal{P}$ gives a* YES *answer, and* NO*-instances, instances for which $\mathcal{P}$ gives a* NO *answer.*

Traditionally, a decision problem is identified by its set of YES-instances. Instances of a decision problem are usually encoded as strings over an alphabet Σ. As a consequence, a decision problem $\mathcal{P}$ is often defined as a *formal language* $\mathcal{L}(\mathcal{P})$ over that alphabet and can be formulated as an instance of the *membership problem* for $\mathcal{L}(\mathcal{P})$.

Theorem 1 (Decision Problems as Membership Problems)**.** *Given a decision problem $\mathcal{P}$, determining whether the answer to a given instance is* YES *is equivalent to determining whether an encoding $w \in \Sigma^*$ of that instance over an alphabet Σ is in the corresponding language $\mathcal{L}(\mathcal{P})$.*

In order to solve a decision problem $\mathcal{P}$, we are interested in algorithms that terminate with a correct answer for every input of $\mathcal{P}$, as formalized by the following definitions.

Definition 29 (Soundness and Completeness of an Algorithm)**.** *An algorithm for a given decision problem $\mathcal{P}$ is sound iff when it returns a YES answer the input is a YES-instance of $\mathcal{P}$. An algorithm for a given decision problem $\mathcal{P}$ is complete iff it always terminates and it returns a YES answer when the input is a YES-instance of $\mathcal{P}$.*

Unsound algorithms are rarely of practical interest. Incomplete algorithms may be effective in solving only YES- or NO-instances of a decision problem. Ideally, however, we would like to have an algorithm for a decision problem $\mathcal{P}$ that is both sound and complete.

Definition 30 (Decision Procedure)**.** *A decision procedure is an algorithmic procedure that, given an instance of a decision problem $\mathcal{P}$, always terminates with a correct* YES *or* NO *answer.*

Definition 31 (Decidability of a Decision Problem)**.** *A decision problem for which there exists a decision procedure is said to be decidable.*

Decidability is an important property to determine whether a given problem may be tackled algorithmically. Many important problems are undecidable, such as the so-called *Entscheidungsproblem* asking for the validity of a first-order formula in the general case (proved undecidable by Church and Turing).

Sometimes, decision procedures are also asked to produce certificates as a way to prove that the results of their computation are correct.

Definition 32 (Certificates)**.** *A certificate is a string of symbols that can be used to prove the answer of a decision procedure on a given instance. We distinguish YES-certificates, produced by the algorithm to certify a YES answer, and NO-certificates, produced by the algorithm to certify a NO answer.*

The typical decision problems we are interested in concern the satisfiability or validity of formulas in some logical system $\mathscr{L}$. Those two problems are interchangeable, as deciding the satisfiability of a formula φ can be seen as the negation of the decision problem to determine whether the formula $\neg\varphi$ is valid. Focusing on the validity problem, we can formulate such a problem as the membership problem of a formula φ to the language of logically valid formulas of $\mathscr{L}$ (or the language theorems, supposing $\mathscr{L}$ to be sound and complete).

In what follows, we assume $\mathscr{L}$ to be both sound and complete. This is because we are mainly interested in decision problems over propositional logics, which are both sound and complete under the usual semantics and proof systems. Considering logical systems that are both sound and complete, we equate the problem of validity (checking whether a formula is a tautology in $\mathscr{L}$) to the problem of derivability (checking whether the formula is a theorem in $\mathscr{L}$).

Propositional (or Boolean) Satisfiability (SAT) is the problem to decide whether a given propositional formula is satisfiable, as defined in Definition 16.

Definition 33 (Boolean Satisfiability Problem)**.** *Given a propositional formula φ over the propositional variables V, SAT is the problem to determine whether $\exists\mu(V) : \mu \models \varphi$. If that is the case, the formula φ is a* YES*-instance of the problem and is said to be satisfiable (*SAT*). The model μ is a* YES*-certificate for the problem on φ. Otherwise, the formula φ is a* NO*-instance of the problem and is said to be unsatisfiable (*UNSAT*).*

SAT is a fundamental decision problem in computer science, important from both the theoretical and the practical perspective. From the theoretical standpoint, SAT plays a central role in computational complexity theory. From the practical standpoint, SAT can be used as a modeling framework to encode many computationally hard problems.

SAT is known to be a computationally hard problem: it is, in fact, a member of the NP-complete class of complexity [24]. This briefly means that no algorithm is currently known that is able to solve all instances of SAT in a polynomial amount of time. Despite this theoretical limitation, the SAT problem is considered in many cases to be tractable, thanks to the fact that its instances often present some kind of structure arising from the application domain they originate from. Currently, state-of-the-art SAT solvers are able to handle SAT instances up to millions of variables and constraints.

Definition 34 (SAT-Solving Algorithm)**.** *A SAT-solving algorithm is a decision procedure to answer the SAT problem.*

A *SAT solver* is software that runs a SAT-solving algorithm, taking a propositional formula (usually in CNF) as input and determining whether or not such a formula can be satisfied by a truth assignment of its variables. SAT-solving algorithms are usually designed as search procedures over the space of truth assignments, following the highly influential DPLL [25] algorithm. To give a better understanding of the subject at hand, Algorithm 1 highlights the top-level procedure of DPLL; a brief description follows as well as a partial example of the assignment's space exploration.

Figure 1 depicts a search tree enumerating all the truth assignments, one for each leaf node, over three variables, taking into account $CNF = \{\{\neg x, y\}, \{\neg y, \neg z\}\}$ as a target formula. Given the visual representation, it is easy to see how looking for a satisfying

assignment is equal to looking for a path, and hence a leaf node, that satisfies a given CNF. Furthermore, taking into account the tree topology and the leaf values, it is possible to identify simplifying steps that reduce the search space, e.g., neither μ_3 nor μ_4 can lead to satisfying assignments; thus, it is irrelevant to evaluate the role of variable z given a partial assignment that reaches such a branch.

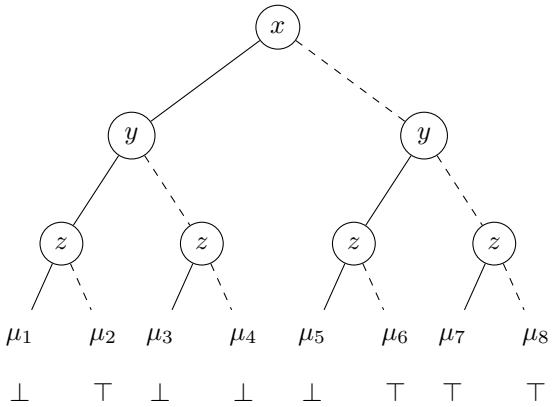


Figure 1. Search tree enumerating all possible assignments for three variables x, y and z with respect to $CNF = \{\{\neg x, y\}, \{\neg y, \neg z\}\}$. Solid (dashed) edges correspond to *true* (*false*) assignments.

In order to evaluate the amount of work that the DPLL algorithm has to perform, a common notion used is that of the *termination tree*, which evaluates the actual search tree considered during exploration. Figure 2 depicts a partial termination tree taking into account the same example introduced in Figure 1. Within a termination tree, one can find conflicts (✗) and valid (✓) assignments, as well as a trace for the conditioned CNF at each level of the search space.

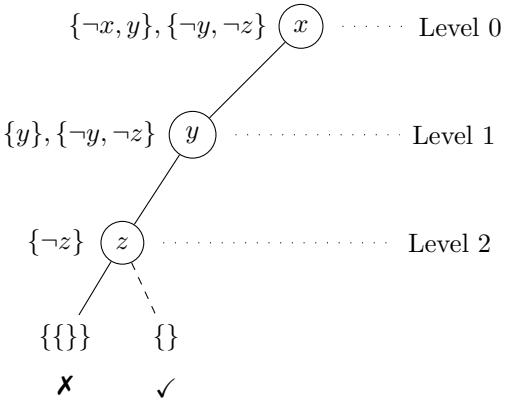


Figure 2. Partial termination tree associated with the example of Figure 1 where each node is labeled with the corresponding CNF. Crosses denote contradiction within a given path whereas check marks denote admissible assignments.

Most modern state-of-the-art SAT solvers employs a SAT-solving algorithm called *Conflict-Driven Clause Learning* (CDCL) [26]. CDCL is based on *clause learning* and *non-chronological backtracking* in order to avoid repeated exploration of regions of the search space that do not lead to a satisfying assignment. During search, CDCL algorithms maintain a *trail* of literals representing the current partial assignment. Under a given partial assignment, each clause loaded in the solver can be either *satisfied*, *conflicts*, *units* or *unresolved*. Clauses are satisfied if at least one of their literals is satisfied. If each literal in a clause is falsified, then the whole clause is a conflict. If all but one literals of the clause are falsified, then the clause is a unit. Otherwise the clause is unresolved. If a clause is a unit, its only non-falsified

literal is implied by the current partial assignment; thus, its variable can be assigned so that the literal is satisfied.

Algorithm 1 Top-level procedure of the DPLL algorithm.

Input: C a set of clauses.
Output: $res \in \{\text{SAT}, \text{UNSAT}\}$.

```
1:  procedure DPLL(C)
2:      while ∃ unit clause l ∈ C do
3:          C ← UnitPropagate(C, l)
4:      while ∃ pure literal l ∈ C do
5:          C ← PureLiteralAssign(C, l)
6:      if C = ∅ then
7:          return SAT
8:      if empty clause ∈ C then
9:          return UNSAT
10:     l ← ChooseLiteral(C)
11:     return DPLL(C ∪ l) or DPLL(C ∪ ¬l)
```

The search starts by selecting a variable, using some suitable heuristic, and assigning it a truth value. This process is called *decision*. After a decision has been made, some of the clauses loaded in the solver may have become conflicts or unit clauses. The solver looks up for unit clauses using efficient data structures, called *watch lists*, in order to perform *Boolean Constraint Propagation* (BCP). During BCP, the solver finds and assigns every clause that is a unit under the current partial assignment. Such a procedure iterates until either no more unit clauses can be found or the solver has found a conflict. In the first case, the solver proceeds with the next decision. In the second case, the solver analyzes the sequence of assignments that have led to the conflict, a process called *conflict analysis* [26]. The result of such an analysis is a *learned clause* catching the causes of the conflict. Such a clause is added to the current formula in order to avoid future iterations to enter the region of the search space that led to the conflict (*learning*). The algorithm, thus, performs non-chronological backtracking canceling enough of its latest decisions (even all of them) to leave such a region of space and then resume search on another region. Eventually, either a conflict is found when no decisions have been made, meaning that the problem at hand is UNSAT, or every variable has been assigned without incurring a conflict, meaning that the problem at hand is SAT.

State-of-the-art SAT solvers are sophisticated artifacts of engineering that often include various techniques and heuristics to enhance their performance. Among such techniques, there are those to periodically *restart search* in order to better explore the space [27], simplifying the problem at hand either before starting searching for a solution or during search [28] and periodically trimming the database of learned clauses.

In many applications, SAT solvers are required to answer a sequence of related calls. In order to preserve useful information about the problem at hand, maintained in the state of the solver, many SAT solvers expose an *incremental interface*. Incremental interfaces enable the user to load a formula in the solver, solve the corresponding SAT problem, then modify the formula and solve the related SAT problem without losing inferences or other useful information about the problem stored in the state. Between incremental calls, the formula can usually be modified by either specifying *variable assumptions* or *adding or removing clauses*.

Many modern SAT solvers can be instructed to provide a *refutation proof* in the case a problem is declared to be UNSAT. Such a proof can be thought of as a NO-certificate that can be used by an independent checker to assess the correctness of the SAT solver's answer. Refutation proofs are usually produced as either *resolution proofs* [29] or *clausal proofs* [30]. A resolution proof is a series of applications of the *resolution rule* that derives the empty clause from the clauses of an unsatisfiable propositional formula.

Definition 35 (Binary Resolution Rule)**.** *Given two clauses C_1 and C_2, the (binary) resolution rule is the inference rule:*

$$\frac{(C_1 \vee a) \wedge (\neg a \vee C_2)}{(C_1 \vee C_2)} \tag{8}$$

Modern SAT solvers are capable, without incurring too large an additional cost, to generate a resolution proof from unsatisfiable runs [31]. Clausal proofs can be seen as logs of the sequence of clauses learned by the solver in the order in which they were learned. A resolution proof can be constructed starting from a clausal proof using BCP [32]. Another common feature of modern SAT solvers is the extraction of *unsatisfiability cores*. Given an UNSAT problem, an unsatisfiability core is a subset of the clauses of the original problem that is still unsatisfiable. The set of clauses in a resolution proof can be seen as an unsatisfiability core.

2.7. Model Checking

Model checking is an automated formal verification technique for determining whether a design meets a given specification. The behavior of the design is usually modeled using a *transition system*. The specification usually consists of a *property* (or a set of properties) expressed in a temporal logic.

Transition systems and temporal logics are discussed in detail in Sections 2.8 and 2.9, respectively. Section 2.10 formalizes the model checking problem as well as a particular subclass of that problem, called invariant verification. In Section 2.11, a way to symbolically represent transition systems is presented, whereas in Section 2.12 the invariant verification problem is reduced to a reachability problem on such a symbolic representation. These concepts form the foundation for the discussion in the remaining sections of this paper, which will each focus on specific model checking algorithms and techniques.

2.8. Transition Systems

Transition systems are often used in computer science as models to describe the behavior of real-world systems. They can be seen as directed graphs, with nodes representing *states* and edges representing *transitions*. A state describes some unique configuration of information about a given system at a certain moment of its behavior. Transitions specify how the system can evolve from one state to another.

Definition 36 (Transition System)**.** *A transition system (also known in the literature as a Kripke structure [33]) is a tuple $\mathcal{TS} \stackrel{def}{=} \langle S, S_0, R, AP, L \rangle$, where S is the set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the transition relation, AP is a set of propositional variables representing atomic propositions and $L : S \to 2^{AP}$ is a labeling function.*

For each state $s \in S$, the labeling function $L(s)$ denotes the set of atomic propositions that hold in s. A transition system can be either *finite* or *infinite*, depending on whether or not it admits an infinite set of states (and consequently transitions). In this paper, we are interested in the model checking of bit-level hardware sequential circuits. Since the behavior of such systems can be modeled using finite transition systems, we restrict the discussion to those models. For this reason, we hereinafter use the term *transition system* as a shorthand for *finite transition systems* without risk of ambiguity.

Example 1. *The sequential hardware system described in Figure 3a, representing a 2-bit synchronous counter that counts up to 2, can be modeled as the following transition system:*

$$\mathcal{TS} = \langle S, S_0, R, AP, L \rangle \tag{9}$$

$$S = \{s_0, s_1, s_2, s_3\} \tag{10}$$

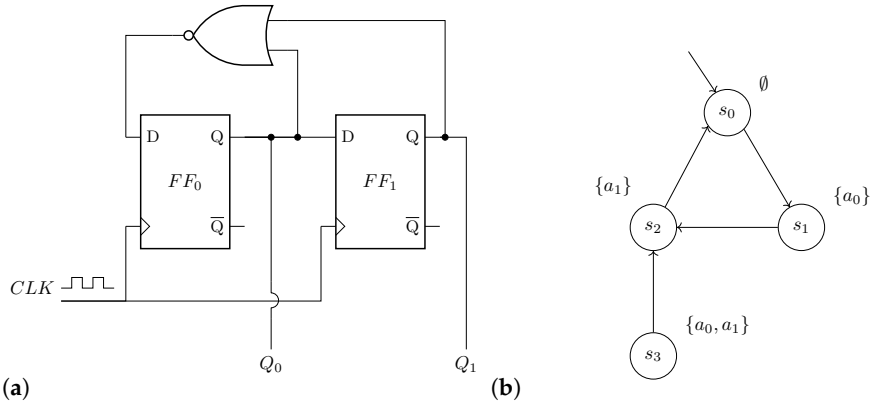
$$S_0 = \{s_0\} \tag{11}$$

$$R = \{(s_0, s_1), (s_1, s_2), (s_1, s_0), (s_3, s_2)\} \tag{12}$$

$$AP = \{a_0, a_1\} \tag{13}$$

$$L = \{(s_0, \emptyset), (s_1, \{a_0\}), (s_2, \{a_1\}), (s_3, \{a_0, a_1\})\} \tag{14}$$

States $S = \{s_0, s_1, s_2, s_3\}$ represent the four possible states of the counter, corresponding to the combined digital values of the two flip flops; s_0 represent the reset state of the system, in which the output of both flip flops is low, R indicates how the system will transition between states at each clock cycle and the labeling function L maps each state to a set of atomic propositions in $AP = \{a_0, a_1\}$ each indicating that either Q_0 or Q_1 is high, respectively. The transition system is visualized in Figure 3b. We will use this system and its model as a running example in the rest of the section.



(**a**) (**b**)

Figure 3. (**a**) A simple sequential hardware design implementing a 2-bit counter from 0 to 2 using D-type flip flops. (**b**) The model of the counter as a transition system.

Let $\mathcal{TS} = \langle S, S_0, R, AP, L \rangle$ be a transition system; we provide the following useful definitions.

Definition 37 (Transition)**.** *A transition in $\mathcal{TS}$ is a pair of states (s_1, s_2) such that $(s_1, s_2) \in R$.*

Definition 38 (Successor and Predecessor States)**.** *Given a pair of states (p, s), p is said to be a predecessor of s iff (p, s) is a transition in $\mathcal{TS}$. Vice versa, s is said to be a successor of p.*

Definition 39 (Paths)**.** *A path in $\mathcal{TS}$ is any sequence of states $\boldsymbol{\pi} = (s_0, s_1, \dots)$ of arbitrary (even infinite) length $|\boldsymbol{\pi}|$ such that (s_i, s_{i+1}) is a transition in $\mathcal{TS}$ for each $0 \leq i < |\boldsymbol{\pi}|$. A path $\boldsymbol{\pi}$ is said to be finite if $|\boldsymbol{\pi}| \in \mathbb{N}$; otherwise, it is said to be infinite. We denote by $\boldsymbol{\pi}_i = (s_i, s_{i+1}, \dots)$ the suffix of $\boldsymbol{\pi}$ starting with the $(i+1)$-th state of $\boldsymbol{\pi}$. We also denote by $\boldsymbol{\pi}[i]$ the $(i+1)$-th state in $\boldsymbol{\pi}$.*

Definition 40 (Initial Paths)**.** *An initial path in $\mathcal{TS}$ is any path $\boldsymbol{\pi} = (s_0, s_1, \dots)$ such that $s_0 \in S_0$.*

Definition 41 (Exact and Bounded Reachability)**.** *A state $s \in S$ is reachable in exactly k steps in $\mathcal{TS}$ iff a finite initial path $\boldsymbol{\pi} = (s_0, \dots, s_{k+1})$ of length $k+1$ such that $s_{k+1} = s$ exists. A state $s \in S$ is reachable within k steps (or reachable bounded by k) in $\mathcal{TS}$ iff there is $i \leq k$ such that s is reachable in exactly i steps in $\mathcal{TS}$.*

Definition 42 (Reachability)**.** *A state $s \in S$ is reachable in $\mathcal{TS}$ if it is reachable within an arbitrary (finite) number of steps in $\mathcal{TS}$.*

Definition 43 (Sets of Exactly Reachable and Sets of Bounded Reachable States)**.** *We denote with $\mathcal{R}_i^E(\mathcal{TS})$ the set of states reachable in exactly i steps in $\mathcal{TS}$. We denote with $\mathcal{R}_i(\mathcal{TS})$ the set of states reachable within i steps in $\mathcal{TS}$, i.e.,*

$$\mathcal{R}_i(\mathcal{TS}) \stackrel{def}{=} \bigcup_{0 \leq j < i} \mathcal{R}_j^E(\mathcal{TS}) \tag{15}$$

Definition 44 (Reachability Diameter)**.** *We define the reachability diameter of $\mathcal{TS}$ to be the minimal number $d \in \mathbb{N}$ of steps required for reaching all reachable states in $\mathcal{TS}$:*

$$d \stackrel{def}{=} \arg\min_{i \in \mathbb{N}} \{i \mid \mathcal{R}_i(\mathcal{TS}) = \mathcal{R}_{i+1}(\mathcal{TS})\} \tag{16}$$

Definition 45 (Set of Reachable States)**.** *We denote with $\mathcal{R}(\mathcal{TS})$ the set of states reachable in $\mathcal{TS}$*

$$\mathcal{R}(\mathcal{TS}) \stackrel{def}{=} \bigcup_{0 \leq j < d} \mathcal{R}_j(\mathcal{TS}) \tag{17}$$

where d is the reachability diameter of $\mathcal{TS}$.

When considering reachability, we use the term *timeframe i*, to denote what happens after i transitions from the initial states. It is trivial to prove that, for a finite-state transition system, there always exists a (finite) reachability diameter.

Example 2. *Given the hardware design and its transition system described in Example 1, $\boldsymbol{\pi} = (s_0, s_1, s_2)$ is both a path and an initial path, the set of reachable states is $\mathcal{R}(\mathcal{TS}) = \{s_0, s_1, s_2\}$ and the reachability diameter of the system is $d = 2$.*

2.9. Temporal Logics

Temporal logics are extensions of classical logics that are used to represent, and reason about, statements qualified in terms of *time*. Both propositional and predicate logic can be extended to include temporal modalities. In this dissertation, we are only interested in *Propositional Temporal Logics* (PTLs), i.e., temporal logics obtained augmenting propositional logic with *temporal operators*. This is because they are the logics typically used to express the requirements for bit-level sequential circuits. Hereinafter, when referring to those logics, we will omit the prefix "propositional" for conciseness.

Different temporal logics, with different expressiveness, can be defined based on the temporal operators they use. The two temporal logics used the most in practice are *Linear Temporal Logic* (LTL) and *Computation Tree Logic* (CTL). LTL adopts a *linear-time* perspective, in which every moment in time is followed by a single successor moment. A possible behavior in LTL can, therefore, be seen as an infinite, ordered sequence of moments. CTL, instead, adopts a *branching-time* perspective, in which every moment in time may be followed by many alternative moments representing different courses of execution. A possible behavior in CTL can, therefore, be seen as an infinite, directed tree of moments. Neither of these logics is a subset of the other, as many temporal properties can be expressed only using one of either LTL or CTL.

The model-theoretic semantics usually given to temporal logics is defined in terms of paths and states of transition systems.

2.9.1. Linear Temporal Logic

Linear Temporal Logic (LTL) was introduced by Pnueli [34] for the specification and verification of reactive systems. Formulas of LTL are constructed from a set of *propositional variables AP* using the usual *logical connectives* (negation $\neg$, conjunction $\wedge$ and disjunction $\vee$) and some *temporal operators* **X** ("next time"), **F** ("eventually"), **G** ("always") and **U** ("until").

LTL formulas are interpreted in terms of paths, i.e., sequences of states of a transition system. Their semantics is also extended to states and whole transition systems.

Definition 46 (Syntax of LTL Formulas)**.** *Given a set of propositional variables AP, the syntax of LTL formulas is defined inductively as follows.*

- *Boolean constants $\top$ and $\bot$ are LTL formulas.*
- *Each $a \in AP$ is an LTL formula.*
- *Given ψ, an LTL formula $\neg\psi$ is an LTL formula.*
- *Given ψ_1 and ψ_2, LTL formula $\psi_1 \wedge \psi_2$ is an LTL formula.*
- *Given ψ_1 and ψ_2, LTL formula $\psi_1 \vee \psi_2$ is an LTL formula.*
- *Given ψ, an LTL formula $\mathbf{X}\psi$ is an LTL formula.*
- *Given ψ, an LTL formula $\mathbf{F}\psi$ is an LTL formula.*
- *Given ψ, an LTL formula $\mathbf{G}\psi$ is an LTL formula.*
- *Given ψ_1 and ψ_2, LTL formula $\psi_1 \mathbf{U} \psi_2$ is an LTL formula.*

Note that $\bot$ and $\vee$ can be omitted from Definition 46 without losing expressiveness as they can be easily derived from $\top$, $\neg$ and $\wedge$. Similarly, **F** and **G** can also be omitted, as they can be derived from **X** and **U** as $\mathbf{F}\psi \stackrel{\text{def}}{=} \top \mathbf{U} \psi$ and $\mathbf{G}\psi \stackrel{\text{def}}{=} \neg\mathbf{F}\neg\psi$. Furthermore, the syntax can be extended to other common Boolean operators by means of their usual definitions in Propositional Logic.

Example 3. *Given the hardware design and its transition system described in Example 1, we can model the specification describing how the system will never reach a state in which the output of both flip flops is high as the following property in LTL logic:*

$$\psi = \neg\mathbf{F}(a_0 \wedge a_1) \tag{18}$$

Intuitively, the semantics of the temporal operators of LTL is the following: $\mathbf{X}\psi$ holds at the current moment if ψ holds at the next moment; $\mathbf{F}\psi$ holds at the current moment if there is a future moment at which ψ holds; $\mathbf{G}\psi$ holds at the current moment if ψ holds at the current moment and at every future moment; $\psi_1 \mathbf{U} \psi_2$ holds at the current moment if there is some future moment at which ψ_2 holds and ψ_1 holds at each moment until that future moment. Formally, the semantics of LTL formulas is defined with respect to paths, states and transition systems, as follows.

Definition 47 (Semantics of LTL Formulas with Respect to Paths)**.** *Given a transition system $\mathcal{TS} \stackrel{def}{=} \langle S, S_0, R, AP, L\rangle$, let $\boldsymbol{\pi} = (s_0, s_1, \dots)$ be an infinite path in $\mathcal{TS}$ and let ψ be an LTL formula over AP. We give the following inductive definition of when $\boldsymbol{\pi}$ satisfies ψ, denoted as $\boldsymbol{\pi} \models \psi$:*

- *$\boldsymbol{\pi} \models \top$ and $\boldsymbol{\pi} \not\models \bot$.*
- *For each $p \in AP$, $\boldsymbol{\pi} \models p$ iff $p \in L(\boldsymbol{\pi}[0])$.*
- *$\boldsymbol{\pi} \models \neg\psi_1$ iff $\boldsymbol{\pi} \not\models \psi_1$.*
- *$\boldsymbol{\pi} \models \psi_1 \wedge \psi_2$ iff $\boldsymbol{\pi} \models \psi_1$ and $\boldsymbol{\pi} \models \psi_2$.*
- *$\boldsymbol{\pi} \models \psi_1 \vee \psi_2$ iff $\boldsymbol{\pi} \models \psi_1$ or $\boldsymbol{\pi} \models \psi_2$.*
- *$\boldsymbol{\pi} \models \mathbf{X}\psi_1$ iff $\boldsymbol{\pi}_1 \models \psi_1$.*
- *$\boldsymbol{\pi} \models \mathbf{F}\psi_1$ iff $\boldsymbol{\pi}_i \models \psi_1$ for some $i \geq 0$.*
- *$\boldsymbol{\pi} \models \mathbf{G}\psi_1$ iff $\boldsymbol{\pi}_i \models \psi_1$ for every $i \geq 0$.*
- *$\boldsymbol{\pi} \models \psi_1 \mathbf{U} \psi_2$ iff $\boldsymbol{\pi}_i \models \psi_2$ for some $i \geq 0$ and $\boldsymbol{\pi}_j \models \psi_1$ for every $0 \leq j < i$.*

Other common Boolean operators have semantics that can easily be inferred from the semantics of $\neg$, $\wedge$ or $\vee$.

Definition 48 (Semantics of LTL Formulas with Respect to States)**.** *Given a transition system* $\mathcal{TS} \stackrel{def}{=} \langle S, S_0, R, AP, L \rangle$*, let* $s \in S$ *be a state and let* ψ *be an LTL formula over* AP*. We say that* s *satisfies* ψ*, denoted as* $s \models \psi$*, iff* $\boldsymbol{\pi} \models \psi$ *for each infinite path* $\boldsymbol{\pi} = (s_0, s_1, \dots)$ *with* $s_0 = s$*.*

Definition 49 (Semantics of LTL Formulas with Respect to Transition Systems)**.** *Given a transition system* $\mathcal{TS} \stackrel{def}{=} \langle S, S_0, R, AP, L \rangle$*, let* ψ *be an LTL formula over* AP*. We say that* $\mathcal{TS}$ *satisfies* ψ*, denoted as* $\mathcal{TS} \models \psi$*, iff* $s_0 \models \psi$ *for each* $s_0 \in S_0$*.*

The interpretation of an LTL formula ψ in terms of a state s requires that all the computations starting from s satisfy ψ, in order for s to satisfy ψ. In LTL semantics, therefore, there is an implicit universal quantification over all computations when determining whether a state satisfies a formula. Thus, LTL can be used to express properties holding for every path from a state but it cannot express properties holding only for some of such paths.

2.9.2. Computation Tree Logic

Computation Tree Logic (CTL) was introduced by Clarke et al. [35] to overcome the semantics limitation of LTL. In CTL, two kinds of formulas are distinguished: *state formulas* and *path formulas*. CTL also introduces two new temporal operators, called *path quantifiers* to allow the specification of properties for some or all the paths starting from a state. These quantifiers are **A** ("for all paths") and **E** ("there exists a path"). CTL formulas are interpreted in terms of trees of paths, i.e., possible alternative paths of a transition system. Their semantics is also extended to states and whole transition systems.

Definition 50 (Syntax of CTL Formulas)**.** *Given a set of propositional variables* AP*, the syntax of CTL state formulas is defined inductively as follows.*

- *Boolean constants* $\top$ *and* $\bot$ *are CTL state formulas.*
- *Each* $a \in AP$ *is a CTL state formula.*
- *Given* ψ*, a CTL state formula* $\neg\psi$ *is a CTL state formula.*
- *Given* ψ_1 *and* ψ_2*, CTL state formula* $\psi_1 \wedge \psi_2$ *is a CTL state formula.*
- *Given* ψ_1 *and* ψ_2*, CTL state formula* $\psi_1 \vee \psi_2$ *is a CTL state formula.*
- *Given* φ*, a CTL path formula* $\mathbf{A}\varphi$ *is a CTL state formula.*
- *Given* φ*, a CTL path formula* $\mathbf{E}\varphi$ *is a CTL state formula.*

The syntax of CTL path formulas is defined inductively as follows:

- *Given* ψ*, a CTL state formula* $\mathbf{X}\psi$ *is a CTL path formula.*
- *Given* ψ*, a CTL state formula* $\mathbf{F}\psi$ *is a CTL path formula.*
- *Given* ψ*, a CTL state formula* $\mathbf{G}\psi$ *is a CTL path formula.*
- *Given* ψ_1 *and* ψ_2*, CTL state formula* $\psi_1 \mathbf{U} \psi_2$ *is a CTL path formula.*

Note that Definition 50 can be restricted to exclude $\top$, $\vee$, **F** and **G**, without losing expressiveness in the same way as Definition 46. Syntax can also be expanded to other common Boolean operators as usual.

Example 4. *Given the hardware design and its transition system described in Example 1, we can model the specification describing how the system will never reach a state in which the output of both flip flops is high as the following property in CTL logic:*

$$\psi = \neg\mathbf{EF}(a_0 \wedge a_1) \tag{19}$$

Intuitively, state formulas describe properties of states, whereas path formulas describe properties of (infinite) paths. Path formulas can be transformed into state formulas by prefixing them with a path quantifier. Note that path quantifiers must immediately precede temporal operators in order for the resulting state formula to be well formed. Intuitively, the semantics of path quantifiers is the following: $\mathbf{A}\varphi$ holds at the current state if φ holds

for all paths starting at the current state; $\mathbf{E}\varphi$ holds at the current state if φ holds for some paths starting at the current state. Formally, the semantics of CTL formulas is defined with respect to paths, states and transition systems, by means of two satisfaction relations: one for state formulas and the other for path formulas.

Definition 51 (Semantics of CTL Path Formulas with Respect to Paths)**.** *Given a transition system $\mathcal{TS} \stackrel{def}{=} \langle S, S_0, R, AP, L\rangle$, let $\boldsymbol{\pi} = (s_0, s_1, \dots)$ be an infinite path in $\mathcal{TS}$ and let φ be a CTL path formula over AP. We give the following inductive definition of when $\boldsymbol{\pi}$ satisfies φ, denoted as $\boldsymbol{\pi} \models \varphi$:*

- $\boldsymbol{\pi} \models \mathbf{X}\psi$ *iff* $\boldsymbol{\pi}[1] \models \psi$.
- $\boldsymbol{\pi} \models \mathbf{F}\psi$ *iff* $\boldsymbol{\pi}[i] \models \psi$ *for some* $i \geq 0$.
- $\boldsymbol{\pi} \models \mathbf{G}\psi$ *iff* $\boldsymbol{\pi}[i] \models \psi$ *for every* $i \geq 0$.
- $\boldsymbol{\pi} \models \psi_1 \mathbf{U} \psi_2$ *iff* $\boldsymbol{\pi}[i] \models \psi_2$ *for some* $i \geq 0$ *and* $\boldsymbol{\pi}[j] \models \psi_1$ *for every* $0 \leq j < i$.

Definition 52 (Semantics of CTL State Formulas with Respect to States)**.** *Given a transition system $\mathcal{TS} \stackrel{def}{=} \langle S, S_0, R, AP, L\rangle$, let $s \in S$ be a state and let ψ be a CTL state formula over AP. We give the following inductive definition of when s satisfies ψ, denoted as $s \models \psi$:*

- $s \models \top$ *and* $s \not\models \bot$.
- *For each* $p \in AP$, $s \models p$ *iff* $p \in L(s)$.
- $s \models \neg\psi_1$ *iff* $s \not\models \psi_1$.
- $s \models \psi_1 \wedge \psi_2$ *iff* $s \models \psi_1$ *and* $s \models \psi_2$.
- $s \models \psi_1 \vee \psi_2$ *iff* $s \models \psi_1$ *or* $s \models \psi_2$.
- $s \models \mathbf{E}\varphi_1$ *iff* $\boldsymbol{\pi} \models \psi$ *for some infinite initial path* $\boldsymbol{\pi}$ *starting in s.*
- $s \models \mathbf{A}\varphi_1$ *iff* $\boldsymbol{\pi} \models \psi$ *for each infinite initial path* $\boldsymbol{\pi}$ *starting in s.*

Definition 53 (Semantics of CTL State Formulas with Respect to Transition Systems)**.** *Given a transition system $\mathcal{TS} \stackrel{def}{=} \langle S, S_0, R, AP, L\rangle$, let $s \in S$ be a state and let ψ be a CTL state formula over AP. We say that $\mathcal{TS}$ satisfies ψ, denoted as $\mathcal{TS} \models \psi$, iff $s_0 \models \psi$ for each $s_0 \in S_0$.*

2.9.3. Temporal Properties

Definition 54 (Temporal Properties)**.** *Given a transition system $\mathcal{TS}$, a temporal property is any property ψ, expressed in a temporal logic, that specifies the admissible (or desired) behaviors of $\mathcal{TS}$.*

Temporal properties typically fall under one of two main categories: safety properties and liveness properties, defined informally as follows.

Definition 55 (Safety Properties)**.** *Given a transition system $\mathcal{TS}$, a safety property asserts that some (undesired) conditions never happen for $\mathcal{TS}$ ("nothing bad ever happens").*

Definition 56 (Liveness Properties)**.** *Given a transition system $\mathcal{TS}$, a liveness property asserts that some (desired) conditions will eventually happen for $\mathcal{TS}$ ("something good eventually happens").*

Typical examples of safety properties include deadlock freedom and mutual exclusion, whereas a typical example of a liveness property is starvation freedom. Both safety and liveness properties can be expressed using LTL and CTL. The main difference between the two categories of properties is that safety properties can be violated by finite computations of a system, whereas liveness properties can only be violated by infinite computations of a system. In practical applications, safety properties are prevalent. The temporal properties defined in Examples 3 and 4 are safety properties.

We distinguish a kind of safety properties of particular interest, called *invariant properties*, defined as follows:

Definition 57 (Invariant Properties)**.** *Given a transition system $\mathcal{TS}$, an invariant property asserts that the condition described by a given propositional formula φ over AP, called an invariant condition, always happen for $\mathcal{TS}$ ("some invariant condition φ always happens").*

Invariant properties can be described in LTL as formulas $\mathbf{G}\varphi$ with the invariant condition φ a formula over AP. Similarly, invariant properties in CTL take the form $\mathbf{AG}\varphi$ with the invariant condition φ a formula over AP.

Example 5. *The corresponding invariant properties of the safety properties defined in Examples 3 and 4 are as follows:*

$$\psi = \mathbf{G}(\neg a_0 \vee \neg a_1) \tag{20}$$

$$\psi = \mathbf{AG}(\neg a_0 \vee \neg a_1) \tag{21}$$

in LTL and CTL, respectively.

Properties management [36,37] goes beyond the scope of this paper, but it is still a topic as relevant as the proper choice of the actual model checking algorithm to be used given a specific scenario.

2.10. Model Checking Problem

Let $\mathcal{TS} = \langle S, S_0, R, AP, L \rangle$ be a transition system; the model checking problem can be formalized as follows:

Definition 58 (Model Checking Problem)**.** *Given a transition system $\mathcal{TS}$ and a temporal property ψ over AP, the decision problem to check whether $\mathcal{TS} \models \psi$ is called a model checking problem. In the case $\mathcal{TS} \not\models \psi$, the model checking procedure is required to provide a counterexample.*

Definition 59 (Counterexample)**.** *Given a transition system $\mathcal{TS}$ and a temporal property ψ over AP such that $\mathcal{TS} \not\models \psi$, a counterexample to ψ for $\mathcal{TS}$ is any initial path $\boldsymbol{\pi}$ of $\mathcal{TS}$ such that $\boldsymbol{\pi} \not\models \psi$.*

Model checking is therefore a decision problem, as defined in Section 2.6, that admits a YES-answer (ψ holds for $\mathcal{TS}$) or a NO-answer (ψ does not hold for $\mathcal{TS}$).

Definition 60 (Model Checking Algorithm)**.** *A model checking algorithm, or model checker, is any decision procedure able to solve a model checking problem.*

When a model checking algorithm terminates with a NO-answer, it must also emit a NO-certificate in the form of a counterexample. Model checking algorithms must systematically traverse all behaviors of the system in order to either confirm that the property holds or provide a counterexample. A model checking algorithm may have either or both of the following qualities.

Definition 61 (Completeness and Soundness (of a Model Checking Algorithm))**.** *A model checking algorithm is said to be complete iff, given a transition system $\mathcal{TS}$ and a property ψ, it is able to either provide a counterexample of ψ for $\mathcal{TS}$ or detect the absence of counterexamples of any length. A model checking algorithm is said to be sound iff, given a transition system $\mathcal{TS}$ and a property ψ, it provides a counterexample of ψ for $\mathcal{TS}$ only when $\mathcal{TS} \not\models \psi$ and it proves the absence of counterexamples of any length only when $\mathcal{TS} \models \psi$.*

Different classes of model checking problems can be distinguished, depending on the category of temporal properties being checked. In this paper, we are interested only in the model checking of invariant properties, also called *invariant verification problems*, which we show can be solved via reachability analysis on the transition system under verification.

Definition 62 (Invariant Verification Problem)**.** *Given a transition system $\mathcal{TS}$ and an invariant property ψ over AP, the invariant verification problem of ψ for $\mathcal{TS}$ is the model checking problem to decide $\mathcal{TS} \models \psi$ or to provide a counterexample.*

Definition 63 (Invariant)**.** *Given a transition system $\mathcal{TS}$ and an invariant property ψ over AP, if $\mathcal{TS} \models \psi$, then the invariant condition φ of ψ is a propositional formula over AP and is said to be an invariant of $\mathcal{TS}$.*

The invariant verification problem can be reduced to a reachability problem over $\mathcal{TS}$, as follows.

Theorem 2 (Reduction of Invariant Verification to Reachability)**.** *Given a transition system $\mathcal{TS}$ and an invariant property ψ over AP, let φ over AP be the invariant condition of ψ; then, the invariant verification problem can be reduced to the problem to check whether a state $s \in S$ such that $s \not\models \varphi$ is reachable in $\mathcal{TS}$. If that is the case, $\mathcal{TS} \not\models \psi$ and any initial path to s is a counterexample for the invariant verification problem. Otherwise, $\mathcal{TS} \models \psi$. Such a problem is called the reachability problem of $\neg\varphi$ in $\mathcal{TS}$.*

Therefore, the invariant verification problem can be simply solved by means of a procedure that traverses the reachable state space of $\mathcal{TS}$ and checks whether or not φ holds for every traversed state. Since the state space is finite, the procedure eventually terminates either providing a counterexample or confirming φ as an invariant of $\mathcal{TS}$. In practice, such a state space is usually too large to be explicitly traversed. State-of-the-art model checking methods rely on a symbolic representation of the system instead.

2.11. Symbolic Representation of Transition Systems

The model checking formulation provided in the previous subsection relies on an explicit representation of states and transitions. Such an explicit representation is not adequate to handle large state spaces. In *symbolic model checking*, the (explicit) transition system is converted into a more concise, implicit representation based on propositional formulas, called *symbolic representation*. Using a symbolic representation, model checking algorithms can traverse the state space of the system more efficiently, by manipulating sets of states and transitions directly, instead of being forced to operate on one state or transition at a time. In addition, since the symbolic representation relies on propositional formulas, symbolic model checking algorithms can leverage well-developed automatic techniques for manipulating such formulas such as BDDs or SAT solvers.

In this Subsection we show how to *encode* a transition system into its symbolic representation and how to formulate the invariant verification problem on such a representation. Given a transition system $\mathcal{TS} = \langle S, S_0, R, AP, L \rangle$, to obtain a symbolic representation of $\mathcal{TS}$, we must define how to encode each of its constituents into propositional formulas.

2.11.1. Symbolic Representation of States

In order to symbolically represent the set of states S of a transition system $\mathcal{TS}$, we introduce a set of propositional variables V, called *state variables*, with $|V| = \lceil log_2 |S| \rceil$. Given a transition system $\mathcal{TS} = \langle S, S_0, R, AP, L \rangle$ and a set of state variables V, we give the following definitions.

Definition 64 (Encoding Function)**.** *An encoding function $\phi : S \to \mathbb{B}^{|V|}$ over V is an injective function mapping states to complete truth assignments over V.*

Definition 65 (Encoding of States)**.** *A state $s \in S$ is encoded as a complete truth assignment $\mu_s = \phi(s)$ over V according to an encoding function ϕ. The complete truth assignment μ_s is called encoding of s over V.*

Note that ϕ is bijective if and only if $|S| = 2^{|V|}$. Otherwise, some pseudo-state representations are introduced in the symbolic representation of S by encoding. Such pseudo-states can be disregarded and treated as unreachable states of $\mathcal{TS}$.

Propositional formulas over V are used to represent sets of states. Any propositional formula F_Q over V represents the set of states $Q \subseteq S$ such that, for each state $q \in Q$, its corresponding encoding μ_q over V satisfies F_Q, i.e., $\mu_q \models F_Q$. Given an encoding function ϕ over V, and letting $q \in S$ be a state of $\mathcal{TS}$, many logically equivalent propositional formulas over V can be used to represent q. We define one example of such a formula as follows.

Definition 66 (Characteristic Formula for States)**.** *The characteristic formula of q is the propositional formula ξ_q over V such that*

$$\xi_q \stackrel{def}{=} \bigwedge_{v_i \in V} l_i \;\; \textit{where} \;\; \begin{cases} l_i = v_i, & \textit{iff } \mu_q(v_i) = \top \\ l_i = \neg v_i, & \textit{iff } \mu_q(v_i) = \bot \end{cases} \tag{22}$$

where $\mu_q = \phi(q)$.

Note that ξ_q is a cube and it is satisfied only by the complete truth assignment μ_q encoding q.

Let $Q \subseteq S$ be a set of states; many logically equivalent propositional formulas over V can be used to represent Q. We define one example of such a formula as follows.

Definition 67 (Characteristic Formula for Set of States)**.** *The characteristic formula of Q is the propositional formula ξ_Q over V such that*

$$\xi_Q \stackrel{def}{=} \bigvee_{q \in Q} \xi_q \tag{23}$$

Note that ξ_Q is satisfied only by the complete truth assignments μ_q encoding states $q \in Q$.

With abuse of notation, hereinafter, we make no distinction among the following concepts:

- A state $s \in S$, any propositional formula F_s that is logically equivalent to its characteristic formula ξ_s and the only model for such formula μ_s.
- A set of states $Q \subseteq S$, any propositional formula F_Q that is logically equivalent to its characteristic formula ξ_Q and the set of models for such formula $\text{Mods}(F_Q) = \{\mu_q \mid q \in Q\}$.

Initial states $S_0 \subseteq S$ of $\mathcal{TS}$ can be represented symbolically by a propositional formula $\mathcal{I}$ such that the encodings μ_{s_0} over V of each initial state $s_0 \in S_0$ satisfy $\mathcal{I}$.

Example 6. *Given the hardware design and its transition system described in Example 1, the states $S = \{s_0, s_1, s_2, s_3\}$ will be encoded using $|V| = \lceil \log_2 |S| \rceil = 2$ state variables $V = \{v_0, v_1\}$ by the encoding function defined as follows:*

$$\phi = \{(s_0, (0,0)), (s_1, (0,1)), (s_2, (1,0)), (s_3, (1,1))\} \tag{24}$$

Individual states will be represented symbolically by their characteristic formulas (cubes):

$$\xi_{s_0} = \neg v_0 \wedge \neg v_1 \quad \xi_{s_1} = v_0 \wedge \neg v_1 \quad \xi_{s_2} = \neg v_0 \wedge v_1 \quad \xi_{s_3} = v_0 \wedge v_1 \tag{25}$$

The set of initial states will be encoded as the following formula:

$$\mathcal{I} = \xi_{s_0} = \neg v_0 \wedge \neg v_1 \tag{26}$$

2.11.2. Symbolic Representation of Transitions

In order to symbolically represent transitions (a binary relation on states), we consider two sets of state variables: one to represent the starting state, called *present state*, and another to represent the final state, called *next state*, of a transition. The set V is used to encode present states while its primed counterpart $V' = \{v' \mid v \in V\}$ is used to encode next states. The same primed notation is used to denote the set of next states $S' = \{s' \mid s \in S\}$ and any propositional formula $F' = F[V \leftarrow V']$ in which the present state variables have been renamed as the next state variables. Furthermore, we use the superscript notation V^i to denote the set of state variables encoding states at timeframe i. The same superscript notation is used accordingly to denote the sets of states S^i after i transitions and any propositional formula $F^i = F[V \leftarrow V^i]$ referring to timeframe i. We also use the notation $\boldsymbol{\pi}^{i,j}$ as a shorthand for a truth assignment over $V^i \cup \cdots \cup V^j$.

Definition 68 (Next State Encoding Function)**.** *Given a transition system $\mathcal{TS}$, a set of present state variables V and a set of next state variables V', let ϕ be an encoding function over S; we define the next state encoding function ϕ' as $\phi' = \phi[V \leftarrow V']$.*

We give the following definitions.

Definition 69 (Encoding of Transitions)**.** *A transition $(s, s') \in R$ is encoded as a complete truth assignment $\mu_{s,s'} = \phi(s) \cup \phi'(s')$ over $V \cup V'$. The complete truth assignment $\mu_{s,s'}$ is called an encoding of (s, s') over $V \cup V'$.*

Propositional formulas over $V \cup V'$ are used to represent sets of transitions. Any propositional formula F_Z over $V \cup V'$ represents the set of transitions $Z \subseteq R$ such that, for each $(z, z') \in Z$, its corresponding encoding $\mu_{z,z'}$ over $V \cup V'$ satisfies F_Z, i.e., $\mu_{z,z'} \models F_Z$. Given the encoding function ϕ and a next state encoding function ϕ', and letting $(z, z') \in R$ be a transition of $\mathcal{TS}$, many logically equivalent propositional formulas over $V \cup V'$ can be used to represent (z, z'). We define one example of such a formula as follows.

Definition 70 (Characteristic Formula for Transitions)**.** *The characteristic formula of (z, z') is the propositional formula $\xi_{z,z'}$ over $V \cup V'$ such that*

$$\xi_{z,z'} \stackrel{def}{=} \bigwedge_{v_i \in V \cup V'} l_i \;\; \textit{where} \;\; \begin{cases} l_i = v_i, & \textit{iff } \mu_{z,z'}(v_i) = \top \\ l_i = \neg v_i, & \textit{iff } \mu_{z,z'}(v_i) = \bot \end{cases} \tag{27}$$

where $\mu_{z,z'} = \phi(z) \cup \phi'(z')$.

Note that $\xi_{z,z'}$ is satisfied only by the complete truth assignment $\mu_{z,z'}$ encoding (z, z').

Let $Z \subseteq R$ be a set of transitions of $\mathcal{TS}$. Many logically equivalent propositional formulas over $V \cup V'$ can be used to represent Z. We define one example of such a formula as follows.

Definition 71 (Characteristic Formula for Sets of Transitions)**.** *The characteristic formula of Z is the propositional formula ξ_Z over $V \cup V'$ such that*

$$\xi_Z \stackrel{def}{=} \bigvee_{(z,z') \in P} \xi_{z,z'} \tag{28}$$

Note that ξ_Z over $V \cup V'$ is satisfied only by the complete truth assignments $\mu_{z,z'}$ over $V \cup V'$ encoding transitions $(z, z') \in Z$.

With abuse of notation, hereinafter, we make no distinction among the following concepts:

- A transition $(s, s') \in R$, any propositional formula $F_{s,s'}$ that is logically equivalent to its characteristic formula $\xi_{s,s'}$ and the only model for such formula $\mu_{s,s'}$.

- A set of transitions $Z \subseteq R$, any propositional formula F_Z that is logically equivalent to its characteristic formula ξ_Z and the set of models for such formula $\text{Mods}(F_Z) = \{\mu_{z,z'} | (z, z') \in Z\}$.

The transition relation R is, thus, encoded as a propositional formula T over $V \cup V'$ such that the encoding $\mu_{s,s'}$ over $V \cup V'$ of each transition $(s, s') \in R$ satisfies T.

Example 7. *Given the hardware design and its transition system described in Example 1, the transition relation R will be encoded as the propositional formula T over $V \cup V'$:*

$$\begin{aligned} T = &(\neg v_0 \wedge \neg v_1 \wedge v_0' \wedge \neg v_1') \vee \\ &(v_0 \wedge \neg v_1 \wedge \neg v_0' \wedge v_1') \quad \vee \\ &(\neg v_0 \wedge v_1 \wedge \neg v_0' \wedge \neg v_1') \vee \\ &(v_0 \wedge v_1 \wedge \neg v_0' \wedge v_1') \end{aligned} \tag{29}$$

2.11.3. Symbolic Representation of the Labeling Function

In order to symbolically represent the labeling function L and the set of propositional variables AP, first, we need to define the reverse labeling function L^{-1}, mapping each propositional variable $a \in AP$ to the set of states in which a holds.

Definition 72 (Reverse Labeling)**.** *The reverse labeling function of a transition system $\mathcal{TS}$ is the function $L^{-1} : AP \to 2^S$ such that, for each $a \in AP$,*

$$L^{-1}(a) = \{s \in S \mid a \in L(s)\} \tag{30}$$

Each propositional variable $a \in AP$ is mapped to a set of states by $L^{-1}(a) \subseteq S$. For each propositional variable a, the set of states $L^{-1}(a)$ can be represented as its characteristic formula $\xi_{L^{-1}(a)}$ over V. The labeling function L is therefore symbolically represented as a set of characteristic formulas $\xi_{L^{-1}(a)}$ over V, one for each $a \in AP$. A propositional variable $a \in AP$ is then symbolically represented by a propositional formula encoding the set of states $L^{-1}(a)$ for which a holds.

Definition 73 (Characteristic Formula of Atomic Propositions)**.** *The characteristic formula of an atomic proposition represented by the propositional variable $a \in AP$ is the propositional formula $\xi_a = \xi_{L^{-1}(a)}$.*

Note that ξ_a is satisfied only by those states $s \in L^{-1}(a)$.

Example 8. *Given the hardware design and its transition system described in Example 1, the atomic propositions $\{a_0, a_1\}$ will be encoded as follows by the reverse labeling function L^{-1}:*

$$L^{-1} = \{(a_0, \{s_1, s_3\}), (a_1, \{s_2, s_3\})\} \tag{31}$$

$$a_0 = \xi_{s_1} \vee \xi_{s_3} = (v_0 \wedge \neg v_1) \vee (v_0 \wedge v_1) = v_0 \tag{32}$$

$$a_1 = \xi_{s_2} \vee \xi_{s_3} = (\neg v_0 \wedge v_1) \vee (v_0 \wedge v_1) = v_1 \tag{33}$$

2.11.4. Symbolic Representation of Transition Systems

Given a transition system $\mathcal{TS} \stackrel{\text{def}}{=} \langle S, S_0, R, AP, L \rangle$, we define a symbolic representation of the whole system as follows.

Definition 74 (Symbolic Transition System)**.** *A symbolic transition system encoding $\mathcal{TS}$ is the tuple $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$, where*

- *V is a set of propositional state variables, with $|V| = \lceil log_2 |S| \rceil$, encoding the states S of $\mathcal{TS}$.*

- $\mathcal{I}$ *is a propositional formula over* V *representing the initial states* S_0 *of* $\mathcal{TS}$.
- T *is a propositional formula over* $V \cup V'$ *representing the transition relation* R *of* $\mathcal{TS}$.

All definitions provided in Section 2.8 about reachability on transition systems naturally extends to their symbolic counterparts.

Example 9. *Given the hardware design and its transition system described in Example 1, the overall encoding of the transition system* $\mathcal{TS}$ *will be the tuple* $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ *defined as follows:*

$$V = \{v_0, v_1\} \tag{34}$$

$$\mathcal{I} = \neg v_0 \wedge \neg v_1 \tag{35}$$

$$\begin{aligned} T = (&\neg v_0 \wedge \neg v_1 \wedge v_0' \wedge \neg v_1') \vee \\ &(v_0 \wedge \neg v_1 \wedge \neg v_0' \wedge v_1') \quad \vee \\ &(\neg v_0 \wedge v_1 \wedge \neg v_0' \wedge \neg v_1') \vee \\ &(v_0 \wedge v_1 \wedge \neg v_0' \wedge v_1') \end{aligned} \tag{36}$$

2.11.5. Symbolic Representation of Invariant Properties

Given a transition system $\mathcal{TS} = \langle S, S_0, R, AP, L \rangle$ and an invariant property ψ with invariant condition φ over AP, we define a symbolic representation of the invariant condition as follows.

Definition 75 (Symbolic Invariant Property)**.** *A symbolic invariant property encoding* φ *is the propositional formula* P *over* V *such that*

$$P \stackrel{def}{=} \varphi[a_1 \leftarrow \xi_{a_1}, \ldots, a_n \leftarrow \xi_{a_n}] \tag{37}$$

for each $a_i \in \text{Vars}(\varphi)$*, i.e., the formula obtained by substituting each propositional variable* a *of* φ *with its characteristic formula* ξ_a *over* V.

Example 10. *Given the hardware design and its transition system described in Example 1, the invariant property* ψ *defined in Example 5 will have an invariant condition* φ *encoded as follows:*

$$\varphi = \neg a_0 \vee \neg a_1 \tag{38}$$

$$P = \neg \xi_{a_0} \vee \neg \xi_{a_1} = \neg v_0 \vee \neg v_1 \tag{39}$$

2.12. Symbolic Invariant Verification

As shown in Section 2.10, the invariant verification problem can be reduced to a problem of reachability in a transition system. Reachability in a transition system (or equally invariant verification) can, in turn, be reduced to reachability in a corresponding symbolic transition system, as follows.

Theorem 3 (Symbolic Invariant Verification)**.** *Given a transition system* $\mathcal{TS} = \langle S, S_0, R, AP, L \rangle$ *and an invariant property* ψ *with invariant condition* φ *over* AP*, the reachability problem of* $\neg\varphi$ *in* $\mathcal{TS}$ *can be reduced to the reachability problem of* $\neg P$ *in* $\mathcal{M}$ *where* $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ *is a symbolic transition system encoding* $\mathcal{TS}$ *and* P *a symbolic invariant property encoding* φ.

In this paper, we are interested in algorithms for symbolic invariant verification. Therefore, with abuse of notation, we hereinafter use the terms "transition system", "reachability", "invariant property" and "invariant verification" to denote their symbolic counterparts.

Notation 5 (Target and Bad States)**.** *Given a transition system $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ and an invariant property P, with the notion of* target *we identify set of states corresponding to $\neg P$. States (or sets of states) which are part of the target, or may reach the target, are called* bad *states.*

Given a transition system $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ and an invariant property P, we provide the following useful definitions for invariant verification algorithms.

Definition 76 (Path Formula)**.** *A path formula of length $k = j - i$, starting from timeframe i and reaching timeframe j, is the propositional formula $\Pi(i, j)$ over $V^i \cup \cdots \cup V^j$:*

$$\Pi(i, j) = \bigwedge_{h=i}^{j-1} T(V^h, V^{h+1}) \tag{40}$$

Definition 77 (Initial Path Formula)**.** *An initial path formula of length k is the following propositional formula:*

$$\Pi_0(k) = \mathcal{I}(V^0) \wedge \Pi(0, k) \tag{41}$$

Intuitively, a path formula $\Pi(i, j)$ encodes all paths of length $k = j - i$ starting at timeframe i in $\mathcal{M}$. An initial path formula $\Pi_0(k)$ encodes all paths of length k starting from the initial states in $\mathcal{M}$. An initial path formula $\Pi_0(k)$ can thus be used to represent the set of states that can be reached from the initial states in exactly k steps, in $\mathcal{M}$.

Definition 78 (Bad Cone)**.** *A bad cone of length $k = j - i$ from timeframe i to timeframe j is the propositional formula $Cone(i, j)$ over $V^i \cup \cdots \cup V^j$:*

$$Cone(i, j) = \Pi(i, j) \wedge \bigvee_{h=i}^{j} \neg P(V^h) \tag{42}$$

A bad cone $Cone(i, j)$ encodes all paths starting at timeframe i that can reach the target in at most $k = j - i$ steps and, thus, it represents the set of bad states that are backward reachable in at most k steps from the target.

Using initial path formulas the reachability problem of $\neg P$ in $\mathcal{M}$ can be reduced to Boolean satisfiability as follows.

Theorem 4 (Reduction of Reachability to Satisfiability)**.** *Given a transition system $\mathcal{M} \stackrel{def}{=} \langle V, \mathcal{I}, T \rangle$ and a property P over V, the reachability of $\neg P$ can be reduced to the problem of finding a* bound *$k \geq 0$ such that the propositional formula*

$$\Pi_0(k) \wedge \neg P(V^k) \tag{43}$$

is satisfiable.

Definition 79 (Induction)**.** *Given a transition system $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$, let F and G be propositional formulas over V and let initiation, consecution and relative consecution be the conditions defined as follows:*

$$\mathcal{I} \rightarrow F \qquad (Initiation)$$
$$F \wedge T \rightarrow F' \qquad (Consecution)$$
$$G \wedge F \wedge T \rightarrow F' \qquad (Relative\ Consecution)$$

A formula F is said to be inductive *if it satisfies consecution. A formula F is said to be an* inductive invariant *if it satisfies both initiation and consecution. A formula F is said to be* inductive relative *to another formula G if it satisfies relative consecution. A formula F is said to be an* inductive invariant relative *to another formula G if it satisfies both initiation and relative consecution.*

Definition 80 (Safety). *Given a transition system $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ and an invariant property P over V, a propositional formula F over V is said to be safe with respect to F iff F is stronger than P, i.e., $F \rightarrow P$.*

Lemma 1 (Inductive Invariants as Overapproximations of Reachable States). *Given a transition system $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$, an inductive invariant F of $\mathcal{M}$ is an overapproximation of the set of reachable states of $\mathcal{M}$, i.e., $\mathcal{R}(\mathcal{M}) \rightarrow F$.*

Intuitively, an inductive invariant expresses a quality of reachable states. Note that $\mathcal{R}(\mathcal{M})$ can be seen as the strongest inductive invariant of $\mathcal{M}$.

Definition 81 (Inductive Strengthening). *Given a transition system $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ and a property P over V, an inductive invariant F of $\mathcal{M}$ is called an inductive strengthening of P iff it is safe with respect to P.*

Lemma 2 (Invariant Verification as Search of an Inductive Strengthening). *Given a transition system $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ and a property P over V, if an inductive strengthening F of P can be found in $\mathcal{M}$, then P holds for every reachable state of $\mathcal{M}$ and $\mathcal{M}$ is said to be safe.*

Many state-of-the-art model checking algorithms are based on a search for an inductive strengthening.

Definition 82 (Traces). *Given a transition system $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$, a trace of length k with respect to $\mathcal{M}$ is a sequence $\mathbf{F_k} = (F_0, \ldots, F_k)$ in which each F_i is a propositional formula over V, called a frame, such that the following conditions hold:*

$$F_0 = \mathcal{I} \qquad \text{(Base)}$$
$$F_i \wedge T \rightarrow F'_{i+1} \text{ for } 0 \leq i < k \qquad \text{(Image Approximation)}$$

A trace $\mathbf{F_k}$ may also satisfy one or both of the following additional conditions, with P a property over V:

$$F_i \rightarrow F_{i+1} \text{ for } 0 \leq i < k \qquad \text{(Monotonicity)}$$
$$F_i \rightarrow P \text{ for } 0 \leq i < k \qquad \text{(Safety)}$$

A trace $\mathbf{F_k}$ that satisfies the monotonicity condition is said to be *monotonic*. A trace $\mathbf{F_k}$ that satisfies the safety condition with respect to the transition system P is said to be *safe*. In order to provide a more clear understanding of the various traces natures, Figure 4 provides a graphical representation as an overview.

The following lemmas hold for traces.

Lemma 3 (Overapproximation of Sets of Exactly Reachable States). *Given a transition system $\mathcal{M}$ and a trace $\mathbf{F_k} = (F_0, \ldots, F_k)$ with respect to $\mathcal{M}$, each timeframe F_i of $\mathbf{F_k}$, with $0 \leq i < k$, is an over-approximation of the set of states reachable in exactly i steps in $\mathcal{M}$, i.e., $\mathcal{R}_i^E(\mathcal{M}) \rightarrow F_i$.*

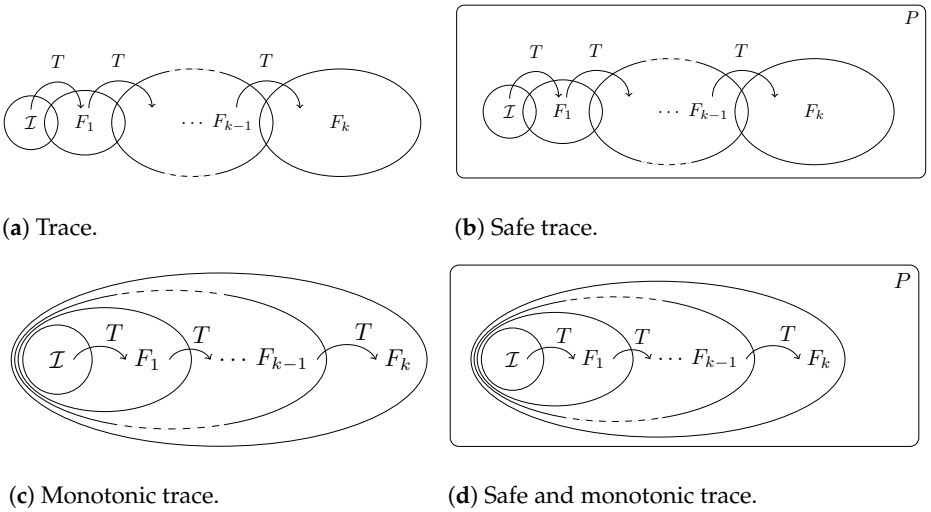
Lemma 4 (Overapproximation of Sets of Reachable States). *Given a transition system $\mathcal{M}$ and a monotonic trace $\mathbf{F_k} = (F_0, \ldots, F_k)$ with respect to $\mathcal{M}$, each timeframe F_i of $\mathbf{F_k}$, with $0 \leq i < k$, is an overapproximation of the set of states reachable in within i steps in $\mathcal{M}$, i.e., $\mathcal{R}_i(\mathcal{M}) \rightarrow F_i$.*

Lemma 5 (Safety up to Trace Bound). *Given a transition system $\mathcal{M}$, a property P over V and a safe and monotonic trace $\mathbf{F_k} = (F_0, \ldots, F_k)$ with respect to P and $\mathcal{M}$, then there does not exist any counterexample to P of length at most $k - 1$ in $\mathcal{M}$.*

Lemma 6 (Inductive Invariants/Strengthening in Traces)**.** *Given a transition system $\mathcal{M}$ and a trace $\mathbf{F_k} = (F_0, \ldots, F_k)$ with respect to $\mathcal{M}$, let us define $F_{0,i} \stackrel{def}{=} \bigvee_{j=0}^{i} F_j$ for each $0 \leq i < k$. If there is $F_{0,i}$ so that $F_{i+1} \rightarrow F_{0,i}$ then $F_{0,i}$ is an inductive invariant of $\mathcal{M}$. If $\mathbf{F_k}$ is also safe with respect to a property P, then $F_{0,i}$ is also an inductive strengthening for P.*

Lemma 7 (Inductive Invariants/Strengthening in Monotonic Traces)**.** *Given a transition system $\mathcal{M}$ and a monotonic trace $\mathbf{F_k} = (F_0, \ldots, F_k)$ with respect to $\mathcal{M}$, if $F_{i+1} \rightarrow F_i$ for some $0 \leq i < k$ then $F_i = F_{i+1}$ is an inductive invariant of $\mathcal{M}$. If $\mathbf{F_k}$ is also safe with respect to a property P, then $F_i = F_{i+1}$ is also an inductive strengthening for P.*

These properties makes traces a very useful data structure to aid the search of an inductive strengthening in $\mathcal{M}$.



(**a**) Trace. (**b**) Safe trace.

(**c**) Monotonic trace. (**d**) Safe and monotonic trace.

Figure 4. Different types of traces with respect to a given $\mathcal{M} \stackrel{\text{def}}{=} \langle V, \mathcal{I}, T \rangle$.

3. Bounded Model Checking

Given a transition system $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ and an invariant property P, bounded model checking (BMC) [38] is an iterative approach whose purpose is to check whether a counterexample to P of length at most k in $\mathcal{M}$ exists or to prove its absence. In order to carry this out, BMC simply performs a SAT check on a formula defined as follows.

Definition 83 (BMC Formula)**.** *A BMC formula of length k for P in $\mathcal{M}$ is the propositional formula* $\text{BMC}(k)$ *over $V^0 \cup \cdots \cup V^k$:*

$$\text{BMC}(k) \stackrel{def}{=} \Pi_0(k) \wedge \bigvee_{i=0}^{k} \neg P(V^i) = \mathcal{I} \wedge Cone(0, k) \tag{44}$$

Intuitively, a BMC formula of length k encodes all initial paths in $\mathcal{M}$ of length at most k capable of reaching a bad state in $\neg P$. If the BMC formula is SAT, then there exists a counterexample to P of length at most k in $\mathcal{M}$. Conversely, no such counterexample exists.

BMC tools iteratively solve BMC formulas of increasing bound, until either a counterexample is found or some maximum bound is reached. Due to the way it operates, BMC is effective in finding counterexamples but it is not able to detect whether P holds in $\mathcal{M}$. In order to surpass such a limitation, specific techniques are required to support unbounded model checking. The ability to check reachability fix-points and/or to find inductive invariants is thus the main difference, and additional complication, between BMC and UMC techniques, which will be introduced in the following sections.

Algorithm 2 reports the BMC procedure up to a certain bound k_{max}. First, the algorithm checks whether the property P is satisfied by the initial states (lines 2–3). If an initial state fails to satisfy P, the procedure terminates by returning FAIL along with a trivial counterexample. Otherwise, the procedure starts iterating over increasing values of k up to k_{max}, checking whether there is at least a path satisfying the BMC formula up to bound k (Definition 83) at each iteration (lines 6–8). If a satisfying path $\boldsymbol{\pi}^{0,k}$ is found, the procedure returns FAIL along with the path as a counterexample. If the procedure does not find any counterexample of length up to k_{max}, it returns a SUCCESS, indicating that P holds up to the given bound.

Algorithm 2 Top-level procedure of the BMC algorithm.

Input: $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ a transition system; P a property over V; k_{max} maximum bound.
Output: $\langle res, cex \rangle$ with $res \in \{\text{SUCCESS}, \text{FAIL}\}$; cex a (possibly empty) initial path representing a counterexample.

1: **procedure** BOUNDEDMODELCHECKING($\mathcal{M}$, P, k_{max})
2: **if** $\exists s_0 \models \mathcal{I}(V) \wedge \neg P(V)$ **then**
3: **return** $\langle \text{FAIL}, (s_0) \rangle$
4: $k \leftarrow 1$
5: **while** $k \leq k_{max}$ **do**
6: $\text{BMC}(k) \leftarrow \Pi_0(k) \wedge \bigvee_{i=0}^{k} \neg P(V^i)$
7: **if** $\exists \boldsymbol{\pi}^{0,k} \models \text{BMC}(k)$ **then**
8: **return** $\langle \text{FAIL}, \boldsymbol{\pi}^{0,k} \rangle$
9: $k \leftarrow k + 1$
10: **return** $\langle \text{SUCCESS}, - \rangle$

As explained above, the search for a counterexample of a given length is inherently incomplete, thus the need for additional techniques in order to overcome such a limitation. Nevertheless, BMC find application in several scenarios, due to its relative ease of application, both in hardware and in software model checking. The minimum requirement is being capable of encoding the transition relation of the instance at hand in a suitable way, e.g., a circuital representation. Furthermore, given a proper encoding, it is possible to combine both hardware and software problems in one, thus making BMC applicable to hybrid scenarios in which HW/SW co-design and co-verification take place. Such scenarios include, but are not limited to, RTL implementations to be checked against golden models, the model checking of properties arising from hardware–software interactions and sequential equivalence checking in between high-level synthesis and hardware description languages.

Example of Bounded Model Checking

Let $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ be a transition system described by the following:

$$V = \{v_0, v_1\} \tag{45}$$

$$\mathcal{I} = \neg v_0 \wedge \neg v_1 \tag{46}$$

$$\begin{aligned} T = (&\neg v_0 \wedge \neg v_1 \wedge v_0^1 \wedge \neg v_1^1) \vee \\ &(v_0 \wedge \neg v_1 \wedge \neg v_0^1 \wedge v_1^1) \vee \\ &(\neg v_0 \wedge v_1 \wedge v_0^1 \wedge v_1^1) \vee \\ &(v_0 \wedge v_1 \wedge \neg v_0^1 \wedge \neg v_1^1) \end{aligned} \tag{47}$$

and P be an invariant property over V:

$$P = \neg v_0 \vee \neg v_1 \tag{48}$$

Figure 5 illustrates the transition systems here described, for reference. The initial state, 00, is marked with the leftmost ingoing edge. The only bad state, 11, is highlighted with a line hatched filling.

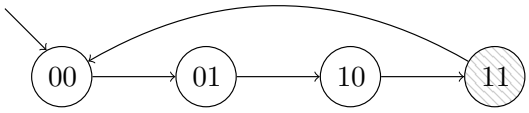


Figure 5. Visualization of states in a transition system $\mathcal{M}$. States are labeled with their corresponding truth-value assignments to state variables V.

An algorithm based on bounded model checking is capable of identifying a counterexample to the target property in four steps, by applying iteratively the BMC formula described in Definition 83.

In the first step, the algorithm would pose the following query:

$$(\neg v_0 \wedge \neg v_1) \wedge (v_0 \wedge v_1) \tag{49}$$

which is UNSAT.

In the second step, the algorithm would pose the following query, in which, for the sake of compactness, we represent with $T(V^i, V^{i+1})$ the aforementioned transition relation, at a given timeframe:

$$\begin{aligned} &(\neg v_0 \wedge \neg v_1) \wedge \\ &T(V^0, V^1) \wedge \\ &(v_0^1 \wedge v_1^1) \end{aligned} \tag{50}$$

which is still UNSAT.

In the last step, the algorithm would pose the following query:

$$\begin{aligned} &(\neg v_0 \wedge \neg v_1) \wedge \\ &T(V^0, V^1) \wedge \\ &T(V^1, V^2) \wedge \\ &T(V^2, V^3) \wedge \\ &(v_0^3 \wedge v_1^3) \end{aligned} \tag{51}$$

which is SAT, hence identifying the counterexample to the property, which does not hold in state 11.

4. Temporal Induction

BMC, introduced in Section 3, is characterized by being an incomplete method: it is capable of finding counterexamples but it is unsuitable to prove correctness. In order to support proofs for unbounded depths, BMC has to be complemented with additional techniques. As an intuition, if exploration is capable of reaching a deep enough bound, such as to have explored all the behavior of the model at hand, such a bound could be considered a so-called completeness threshold. Such a threshold can be used as an upper bound for the length of the counterexamples that have to be considered before stating whether a proof holds.

In the context of model checking, SAT-based techniques are particularly suitable for performing checks of inductions, i.e., whether the transition system of the model under verification satisfies a given inductive invariant. The underlying question we are posing is whether a given property holds in the initial states and in all the states reachable from states in which the property is satisfied. The main limitation of such reasoning is due to the fact that, if the second condition fails, nothing can be said about the property at hand.

On these premises, temporal induction [39], also known as strong induction or $k-$induction, represents a widely used technique in unbounded model checking. Such a technique is a generalization of the notion of inductive invariants, which provide a strengthening factor to the base intuition presented above.

Given a transition system $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ and an invariant formula φ over V, that formula is said to be $k-$invariant if it is true in the first k steps of $\mathcal{M}$.

Definition 84 (k-invariant formula)**.**

$$\Pi_0(k) \rightarrow \bigwedge_{i=0}^{k} \varphi(V^i) \tag{52}$$

A k-invariant formula is a k-inductive invariant if it is $(k-1)$-invariant and is inductive after k steps of $\mathcal{M}$. With respect to simple induction, $k-$induction strengthen the hypothesis in the induction step. We have that the property is assumed to hold in the first $k-1$ steps, starting from 0, and is established in step k.

Whenever we have a k-invariant formula φ such that $\varphi \rightarrow P$, we say that φ is a safe $k-$inductive invariant with respect to $\mathcal{M}$.

Algorithm 3 reports the top-level procedure for a generic k-induction model checking scheme. First, the algorithm checks whether the property P is satisfied by the initial states (lines 3–4). If an initial state fails to satisfy P, the procedure terminates by returning FAIL along with a trivial counterexample (lines 5–6). Otherwise, the procedure starts iterating over increasing values of k, checking whether there is at least a path satisfying the induction step formula up to bound k (line 7), and taking into account only non-looping paths (line 8). If a satisfying path is found, the procedure returns SUCCESS (lines 9–10), otherwise the bound k is increased and the procedure iterates further.

Algorithm 3 Top-level procedure of the k-induction algorithm.

Input: $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ a transition system; P a property over V.
Output: $\langle res, cex \rangle$ with $res \in \{\text{SUCCESS}, \text{FAIL}\}$; cex a (possibly empty) initial path representing a counterexample.

1: **procedure** TEMPORALINDUCTIONMODELCHECKING($\mathcal{M}$, P)
2: $k \leftarrow 0$
3: **while** true **do**
4: $\text{BASE}(k) \leftarrow \Pi_0(k) \wedge \bigwedge_{i=0}^{k} P(V^i)$
5: **if** $\exists \boldsymbol{\pi}^{0,k} \models \text{BASE}(k)$ **then**
6: **return** $\langle \text{FAIL}, \boldsymbol{\pi}^{0,k} \rangle$
7: $\text{STEP}(k) \leftarrow \Pi(0,k) \wedge \bigwedge_{i=0}^{k} P(V^i) \wedge \neg P(V^{k+1})$
8: $\text{UNIQUE}(k) \leftarrow \bigwedge_{0 \leq i < j \leq k} s_i \neq s_j$
9: **if** $\nexists \boldsymbol{\pi}^{0,k} \models \text{STEP}(k) \vee \text{UNIQUE}(k)$ **then**
10: **return** $\langle \text{SUCCESS}, - \rangle$
11: $k \leftarrow k + 1$

Theorem 5. *Given a transition system $\mathcal{M}$, there exists a safe inductive invariant with respect to $\mathcal{M}$ iff there exists a safe $k-$inductive invariant with respect to $\mathcal{M}$.*

Theorem 5 states that k-induction is as complete as 1-induction.

5. Interpolation-Based Model Checking

Craig's interpolation theorem is a fundamental result in mathematical logic taking into account the relationship between model theory and proof theory. Interpolation, despite its heavy theoretical foundation, has found many practical applications in different areas of computer science. Originally, the formulation of the theorem introduced by Craig [40] was given in the context of first-order logic. Variants of the theorem hold for other logical systems as well, including propositional logic, which is the context we will focus on hereinafter, as it is the one typically encountered in the context of model checking.

Theorem 6 (Propositional Craig's Interpolation Theorem)**.** *Given two propositional formulas A and B, if $A \wedge B$ is unsatisfiable then there is a propositional formula I, called interpolant between A and B, such that*

- $A \rightarrow I$ *is valid;*
- $I \wedge B$ *is unsatisfiable;*
- $\mathrm{Vars}(I) \subseteq \mathrm{Vars}(A) \cap \mathrm{Vars}(B)$.

Intuitively, I is an abstraction of A from the standpoint of B. The interpolant summarizes and translates in the common language between the two formulas the reasons why A and B are inconsistent. For ease of notation, we denote with $I = \textsc{Itp}(A, B)$ the procedure that derives a Craig's interpolant starting from a pair of inconsistent formulas A and B.

Interpolants can be derived from refutation proofs of unsatisfiable SAT-solving runs. Given an unsatisfiable formula $A \wedge B$, most modern SAT solvers are capable of generating a refutation proof, as described in Section 2.6, either in resolution-based or clausal form. Given a resolution proof, an interpolant $I = \textsc{Itp}(A, B)$ can be derived in the form of an AND/OR combinational circuit in polynomial time and space with respect to the size of the proof. Different algorithms for generating interpolants from resolution proofs have been proposed, such as the ones by Huang [41], Krajícek [42], Pudlák [43] and McMillan [44]. We refer to those algorithms as *interpolation systems*. Interpolants can also be derived as CNF formulas, instead of circuits, from either resolution proofs [45] or clausal proofs [46]. Although interpolants may be quite large in size, different approaches exist to compact them, also taking into account the relation between the size and strength of the interpolants themselves [47].

In the context of model checking, if A represents a set of reachable states and B represents a set of bad states, then the interpolant $I = \textsc{Itp}(A, B)$ can be deemed as a safe overapproximation of A with respect to B. In turn, such overapproximations can be used to detect a reachability fix-point.

5.1. McMillan's Interpolation Algorithm

McMillan [44] introduced the first complete algorithm for symbolic model checking based on Craig's interpolation. In the literature, such an algorithm is called *standard interpolation* or just ITP for short. The algorithm computes Craig's interpolants to overapproximate reachable states of the system. The interpolants, as mentioned in the previous section, are computed from refutation proofs of unsatisfiable BMC runs.

The algorithm comprises two nested loops. The outer loop is presented in procedure ItpModelChecking (Algorithm 4) whereas the inner loop is presented in procedure ApproxForwardTraversal (Algorithm 5). At each outer loop iteration, ApproxForwardTraversal is invoked to perform an overapproximated forward traversal of the reachable states while preserving safety with respect to a backward unrolling from the target (*bad cone*). ApproxForwardTraversal can be seen as computing a safe monotonic trace. Note that such a trace is not explicitly maintained. Only its final frame is kept at each iteration and used as base for computing the next one.

We describe Algorithm 5 first. The procedure APPROXFORWARDTRAVERSAL operates a forward traversal in which interpolation is used as an overapproximated image operator. At each iteration, the procedure checks a BMC formula of fixed length k, composed of two parts:

- $A \stackrel{\text{def}}{=} R(V^0) \wedge T(V^0, V^1)$
- $B \stackrel{\text{def}}{=} Cone(1,k) = \Pi(1,k) \wedge \bigvee_{i=1}^{k} \neg P(V^i)$

where R is a set of overapproximated forward reachable states. In such a scenario, A represents the image of the set of states at the current traversal step, whereas B represents the set of backward reachable bad states, reachable in at most $k-1$ transitions from the target.

Algorithm 4 Top-level procedure of McMillan's interpolation algorithm. This iterates overapproximated forward traversals of reachable states while keeping safety with respect to a bad cone of increasing depth from the target.

Input: $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ a transition system; P a property over V.
Output: $\langle res, cex \rangle$ with $res \in \{\text{SUCCESS}, \text{FAIL}\}$; cex a (possibly empty) initial path representing a counterexample.

```
1:  procedure ITPMODELCHECKING(M, P)
2:      if ∃s_0 ⊨ I(V) ∧ ¬P(V) then
3:          return ⟨FAIL, (s_0)⟩
4:      k ← 1
5:      while true do
6:          ⟨res, cex⟩ ← APPROXFORWARDTRAVERSAL(M, P, k)
7:          if res is UNREACH then
8:              return ⟨SUCCESS, −⟩
9:          else if res is REACH then
10:             return ⟨FAIL, cex⟩
11:         k ← k + 1
```

Algorithm 5 Inner procedure of McMillan's interpolation algorithm. This operates an overapproximated forward traversal of the reachable state space while keeping safety with respect to a bad cone of fixed depth from the target.

Input: $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ a transition system; P a property over V; k bound of a backward unrolling from the target.
Output: $\langle res, cex \rangle$ with $res \in \{\text{REACH}, \text{UNREACH}, \text{UNDEF}\}$; cex a (possibly empty) initial path representing a counterexample.

```
1:  procedure APPROXFORWARDTRAVERSAL(M, P, k)
2:      R ← I
3:      if ∃π^{0,k} ⊨ R(V^0) ∧ T(V^0, V^1) ∧ Cone(1, k) then
4:          return ⟨REACH, π^{0,k}⟩
5:      while ⊤ do
6:          A ← R(V^0) ∧ T(V^0, V^1)
7:          B ← Cone(1, k)
8:          if ∃π^{0,k} ⊨ A ∧ B then
9:              return ⟨UNDEF, −⟩
10:         else
11:             I ← ITP(A, B)
12:             if ∄s ⊨ I ∧ ¬R then
13:                 return ⟨UNREACH, −⟩
14:             R ← R ∨ I
```

At the first iteration, R is the set of initial states $\mathcal{I}$ (line 2) and therefore is not overapproximated. In that case, the formula $A \wedge B$ is exactly a BMC formula of length k. If such a formula is satisfiable, the algorithm has found a counterexample $\pi^{0,k}$ of length at most k (lines 3–4). Then, both the nested and the top-level procedures terminate, returning

the counterexample found. Otherwise, the algorithm starts traversing the reachable state space, computing overapproximated images by means of interpolation (lines 5–14). After composing A and B (lines 6–7), the algorithm checks whether $A \wedge B$ is satisfiable (line 8). If that is the case, a (possibly spurious) counterexample $\boldsymbol{\pi}^{0,k}$ is found, i.e., a path of length at most k from the overapproximated set R to a bad state in $Cone(1,k)$. Since R is an overapproximation of the reachable states, the algorithm cannot determine whether $\boldsymbol{\pi}^{0,k}$ is a *real counterexample* or a *spurious counterexample*. Therefore, the procedure aborts the current forward traversal and returns the control to the top-level procedure. Otherwise, $A \wedge B$ is UNSAT and an interpolant I can be derived from its refutation proof (line 11). Such an interpolant represents a safe overapproximation of the states reachable from R in one transition with respect to the bad cone $Cone(1,k)$, as illustrated in Figure 6. Therefore, no counterexample can be reached from I in $k-1$ or less transitions, i.e., I is safe with respect to P.

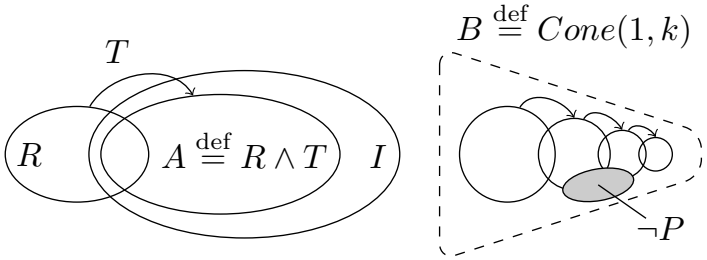


Figure 6. Interpolation as an overapproximated image operator. Interpolant I is an overapproximation of the image of R that does not intersect $Cone(1,k)$.

Since the interpolant is an overapproximation of the image of R, it is treated as a *candidate inductive invariant*. The algorithm checks whether consecution $I \rightarrow R$ is valid (i.e., $\neg R \wedge I$ is unsatisfiable). In that case, R is an inductive invariant for $\mathcal{M}$ and, since R is safe with respect to P, it is also an inductive strengthening for P. Conversely, a new set of overapproximated forward reachable states is computed as $R \vee I$ and the algorithm iterates once again. The result of the nested procedure, i.e., the sequence of R composed at each iteration, can be seen as a safe monotonic trace. Monotonicity stems from having R initialized with $\mathcal{I}$ (line 2) and from the fact that each consecutive R is a disjunction of the previous one and of an overapproximation of the states reachable from the previous (line 14). Safety with respect to P stems from the fact that $\mathcal{I}$ was proved to be safe (Algorithm 4, lines 2–4) and taking into the account that each interpolant I used to construct R does not intersect $Cone(1,k)$. From Lemma 7 follows the detection of an inductive strengthening for P from that trace. The sequence of interpolants, instead, can be seen as a non-monotonic safe trace.

Taking into account Algorithm 4, in the beginning, the initial states $\mathcal{I}$ are checked to be safe (line 2–3). If said check fails, there exists a trivial counterexample consisting of just a single initial state. The procedure can then halt and return the counterexample found. Conversely, the bad cone bound k is initialized to 1 (line 4) and the procedure starts iterating forward overapproximated traversals of reachable states, while keeping safety with respect to a bad cone of increasing depth k from the target (lines 5–11). Increasing the value of k lets the algorithm find real counterexamples and generate more precise overapproximations on subsequent iterations. Since the bad cone from the target unwinds, a few states in the overapproximated images computed at previous iterations might be reached by the unrolling and are therefore excluded from newer images. During each outer loop iteration, the inner procedure starts a forward traversal from scratch from the initial states. As k increases, the algorithm is guaranteed to find a bound k in which the computed interpolants are precise enough to find an inductive strengthening if P holds for the system, or to find a real counterexample otherwise [44].

5.2. Example of Interpolation-Based Model Checking

Let $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ be a transition system described by the following:

$$V = \{v_0, v_1, v_2\} \tag{53}$$

$$\mathcal{I} = \neg v_0 \wedge \neg v_1 \wedge \neg v_2 \tag{54}$$

$$\begin{aligned} T = {} & (\neg v_0 \wedge \neg v_1 \wedge \neg v_2 \wedge v_0' \wedge \neg v_1' \wedge \neg v_2') \vee \\ & (v_0 \wedge \neg v_1 \wedge \neg v_2 \wedge \neg v_0' \wedge \neg v_1' \wedge v_2') \vee \\ & (\neg v_0 \wedge \neg v_1 \wedge v_2 \wedge v_0' \wedge \neg v_1' \wedge v_2') \vee \\ & (v_0 \wedge \neg v_1 \wedge v_2 \wedge \neg v_0' \wedge \neg v_1' \wedge \neg v_2') \vee \\ & (\neg v_0 \wedge v_1 \wedge \neg v_2 \wedge v_0' \wedge v_1' \wedge \neg v_2') \vee \\ & (v_0 \wedge v_1 \wedge \neg v_2 \wedge v_0' \wedge v_1' \wedge v_2') \vee \\ & (v_0 \wedge v_1 \wedge v_2 \wedge v_0' \wedge v_1' \wedge v_2') \vee \\ & (\neg v_0 \wedge v_1 \wedge v_2 \wedge v_0' \wedge v_1' \wedge v_2') \end{aligned} \tag{55}$$

and P be an invariant property over V:

$$P = \neg v_0 \vee \neg v_1 \vee \neg v_2 \tag{56}$$

Figure 7 illustrates different steps of the procedure reported in Algorithms 4 and 5 over $\mathcal{M}$. The ITPMODELCHECKING procedure starts by checking whether or not the initial states $\mathcal{I}$ are safe. In the case of the system at hand, the single initial state 000 satisfies P; therefore, the procedure will continue by invoking APPROXFORWARDTRAVERSAL with $k = 1$. During the first iteration of the traversal procedure, R is set to the initial states $\mathcal{I}$, A represents the image of $\mathcal{I}$ and B corresponds to the negation of the property $\neg P$. Suppose that the interpolant I extracted at line 11 of Algorithm 5 is the one described in Figure 7a. I overapproximates the image of R, comprising both states that are actually reachable from $\mathcal{I}$, like 001, and states that are not reachable from $\mathcal{I}$, like 010 and 011. Figure 7b describes the situation after the current set of reachable states R has been updated by disjointing it with the interpolant I. During the second iteration of the traversal procedure, an intersection between the image of the current set of reachable states A and the bad states B will be found (state 111, as indicated by both the solid and the dash–dotted line enclosing it in Figure 7b). The traversal procedure will then halt reporting an UNDEF result: this is the case in which a spurious counterexample to the property has been found, due to the interpolant I being too loose an overapproximation of the states actually reachable from $\mathcal{I}$, thus including a state (011) that is backward reachable from a bad state but not forward reachable from the initial states. The ITPMODELCHECKING procedure will then resume by increasing k and invoking the traversal procedure once again with an enlarged cone of bad states. Figure 7c describes the situation during the first step of the new traversal, in which R and A are left unchanged whereas B is enlarged to include the pre-image of $\neg P$ (states 011 and 110). Suppose that a new interpolant I is extracted that includes only states 000, 001, 100 and 101; then, after updating the current set of reachable states R and performing a second traversal step, a fix-point will be reached as illustrated in Figure 7d. The overall model checking procedure will end with a SUCCESS result.

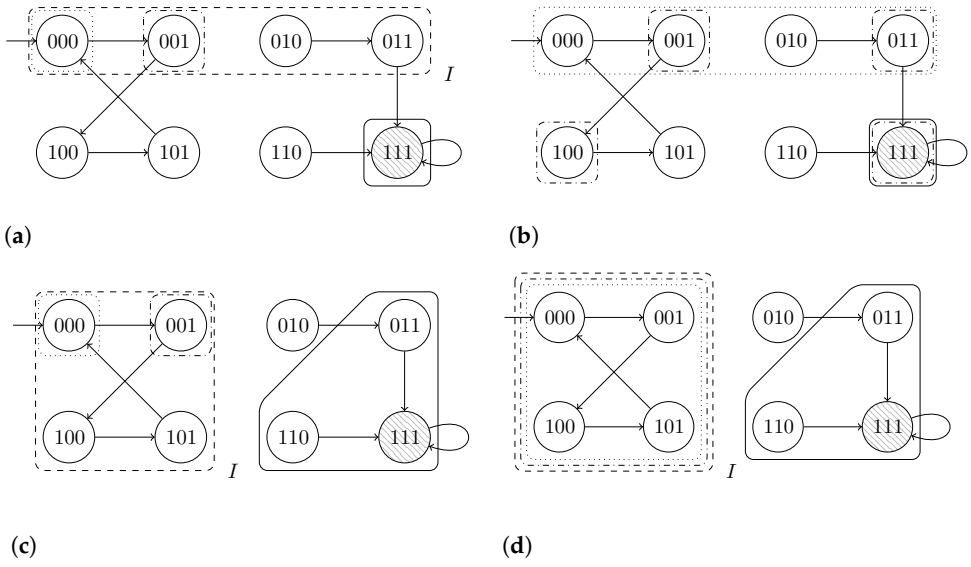


Figure 7. Different steps of McMillan's interpolation-based model checking procedure visualized as groupings of states in a transition system $\mathcal{M}$. States are labeled with their corresponding truth-value assignments to state variables V. States enclosed by solid lines represent the set B of bad states or states that are backward reachable from a bad state. States enclosed by dotted lines represent the current set R of states that are reachable from the initial states. States enclosed by dash–dotted lines represent the set A of states in the image of R. States enclosed by dashed lines represent the interpolant I that overapproximates A. In (**a**) the initial scenario is depicted, where I overapproximates the image of R including both reachable and unreachable states (from $\mathcal{I}$). (**b**) depicts the situation after the current set of reachable states R has been updated by disjointing it with the interpolant I and the subsequent traversal lead to a spurious counterexample. (**c**) depicts a new traversal after overapproximation refinement. (**d**) depicts the convergence fix-point for the algorithm.

6. IC3

IC3 [48] is a SAT-based algorithm for symbolic invariant verification. Given a transition system $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ and an invariant property P over V to be checked, the target of IC3 is finding an inductive strengthening of P for $\mathcal{M}$.

In order to carry this out, IC3 maintains two main data structures: a *trace* and a *proof-obligation queue*.

The *trace* $\mathbf{F_k} = (F_0, \ldots, F_k)$ is both monotonic and safe with respect to the property P. Each frame F_i, with $0 \leq i < k$, is a safe overapproximation of the set of states reachable in at most i steps in $\mathcal{M}$ (see Lemmas 4 and 5). The purpose of the algorithm is to iteratively refine such an $\mathbf{F_k}$ in order to satisfy the condition $F_{i+1} \rightarrow F_i$ for some $0 \leq i < k$, thus finding an inductive strengthening of ψ according to Lemma 6.

IC3 also maintains a second data structure called a *proof-obligation queue*, which is used to collect sets of states in $\mathbf{F_k}$ that may reach a violation of the property in an arbitrary number of steps. Those sets of states are processed by the algorithm according to a given priority and for each of them IC3 either finds a backward path to the initial states or learns a new inductive lemma that can be used to refine $\mathbf{F_k}$ in order to exclude such states from the overapproximation. In case a path to the initial states is found, the algorithm has found a counterexample to P. In the latter case, the algorithm continues its search of an inductive strengthening of P over a tighter approximation of the reachable state sets.

IC3 needs to solve SAT calls at different stages during its run. Such SAT queries are peculiar [49], in the sense that they are very frequent but involve only a single instance of the transition relation T. Performing many *local reachability checks*, IC3 achieves better control over the precision of the computed overapproximations.

Given $\mathbf{F_k}$, each frame F_i is represented by a set of clauses, denoted by clauses (F_i), to enable efficient syntactic checks for frame equality. The base condition of the trace is fulfilled by initializing the first frame with $\mathcal{I}$. Monotonicity is maintained syntactically by enforcing the condition clauses$(F_{i+1}) \subseteq$ clauses(F_i). On the other hand, image approximation and safety are guaranteed explicitly by the algorithm operations.

The main procedure of IC3, described in Algorithm 6, is composed of an initialization phase followed by two nested iterations. During initialization, the algorithm sets up the trace and ensures that the initial states are safe (lines 2–5). If that is not the case, a counterexample of length zero, comprising only the initial state s violating the property, is found and the procedure outputs a failure. During each outer iteration (lines 6–17), the algorithm tries to prove that P is satisfied up to k steps in $\mathcal{M}$, for increasing values of k. Inner iterations of the algorithm refine the trace $\mathbf{F_k}$ computed so far by adding new relative inductive clauses to some of its frames (lines 7–11). The algorithm iterates until either an inductive strengthening of the property is generated (line 16) or a counterexample to the property is found (line 11).

Algorithm 6 Top-level procedure of the IC3 algorithm: IC3 ($\mathcal{M}$, P).

Input: $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ a transition system; P a property over V.
Output: $\langle res, cex \rangle$ with $res \in \{\text{SUCCESS}, \text{FAIL}\}$ and cex a (possibly empty) initial path representing a counterexample.

```
1:  procedure IC3(M, P)
2:      k ← 0
3:      F_0 ← I
4:      if ∃s ⊨ F_0 ∧ ¬P then
5:          return ⟨FAIL, (s)⟩
6:      repeat
7:          while ∃s ⊨ F_k ∧ ¬P do
8:              q ← EXTEND(s)
9:              ⟨res, cex⟩ ← BLOCKCUBE(q, Q, F_k)
10:             if res= FAIL then
11:                 return ⟨FAIL, cex⟩
12:         F_{k+1} ← ∅
13:         k ← k + 1
14:         PROPAGATE(F_k)
15:         if F_i = F_{i+1} for some 0 ≤ i < k then
16:             return ⟨SUCCESS, −⟩
17:     until forever
```

At a given outer iteration k, the algorithm has already computed a monotonic and safe trace. From Lemma 5, it follows that P is satisfied up to $k-1$ steps in $\mathcal{M}$. IC3 then tries to prove that P is satisfied up to k steps as well, by enumerating states of F_k that violate P and trying to block them in F_k.

Definition 85 (Cube Blocking)**.** *Blocking a state (or, more generally, a cube) q in a frame F_k means to prove q unreachable within k steps in $\mathcal{M}$ and, consequently, to refine F_k so that q is excluded from its overapproximation.*

To enumerate each state of F_k that violates P (line 7), the algorithm looks for states s in $F_k \wedge \neg P$. Such states are called *bad states* and can be found as satisfying assignments for the following SAT query:

$$\text{SAT } ?(F_k \wedge \neg P) \tag{57}$$

If a bad state s can be found (i.e., Query (57) is SAT), the algorithm tries to block it in F_k. To increase performance of the algorithm, as suggested in [50], the bad state s found is first extended to a bad cube q by removing, if possible, some of its literals. Figure 8a visualizes

this step. The EXTEND(s) procedure (line 8), not reported here, performs this operation via either ternary simulations [50] or other SAT-based procedures [51]. The resulting cube q is stronger than s and it still violates P; it is thus called a *bad cube*. The algorithm then tries to block the bad cube q in frame k rather than s. It is shown in [50] that extending bad states into bad cubes before blocking them dramatically improves IC3 performance. The bad cube q is blocked in F_k calling the BLOCKCUBE(q, Q, $\boldsymbol{F_k}$) procedure (line 9), described in Algorithm 7.

When no further bad states can be found, the conditions in Definition 82 hold for $k+1$ and IC3 can safely move to the next outer iteration, i.e., trying to prove that P is satisfied up to $k+1$ steps. Before moving to the next iteration, a new empty frame F_{k+1} is created (line 12). Initially, clauses$(F_{k+1}) = \emptyset$, so that $F_{k+1} = \top$. Note that $\top$ is a valid overapproximation to the set of states reachable within $k+1$ steps in $\mathcal{M}$.

After creating a new frame, a phase called *clause propagation* takes place (line 14). During such a phase, IC3 tries to refine every frame F_i, with $0 < i \leq k$, by checking whether some of its clauses can be pushed forward to the following frame. Possibly, clause propagation refines the outermost frame F_k so that an inductive invariant is reached. The propagation phase can lead to two adjacent frames becoming equivalent (see Figure 8g). If that happens, the algorithm has found an inductive strengthening of P for $\mathcal{M}$ (equal to those frames). Therefore, following Lemma 6, property P holds for every reachable state of $\mathcal{M}$ and IC3 returns a successful result (line 16). Procedure PROPAGATE($\boldsymbol{F_k}$), described and discussed later, handles the clause propagation phase.

The purpose of procedure BLOCKCUBE(q, Q, $\boldsymbol{F_k}$) is to refine the trace $\boldsymbol{F_k}$ in order to block a bad cube q in F_k. To preserve *image approximation*, prior to blocking a cube in a certain frame, IC3 has to recursively block predecessors of that cube in the preceding frames. To keep track of the states (or cubes) that must be blocked in certain frames, IC3 uses the formalism of *proof obligations*.

Definition 86 (Proof Obligation)**.** *Given a cube q and a frame F_j, a proof obligation is a couple (q, j) formalizing the fact that q must be blocked in F_j.*

Given a proof obligation (q, j), the cube q can either represent a set of bad states or a set of states that can reach a bad state in some number of transitions. The index j indicates the position in the trace where q must be proved unreachable, or else the property fails.

Definition 87 (Discharging Proof Obligations)**.** *A proof obligation (q, j) is said to be discharged when s becomes blocked in F_j.*

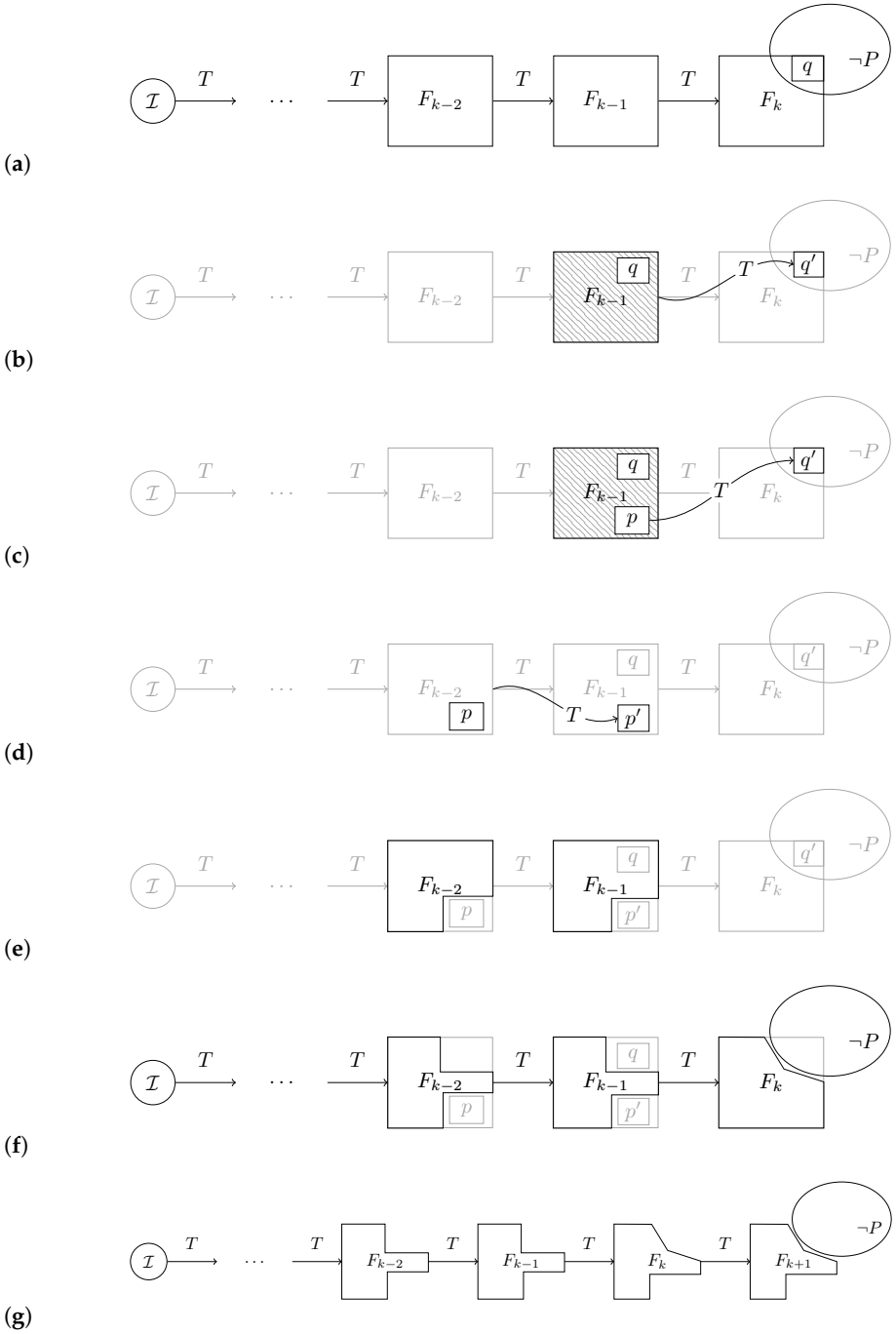


Figure 8. Visual representation of different steps in the IC3 procedure. In (**a**), a bad cube s that violates P is found in F_k and extended to q. (**b**) shows the beginning of the cube blocking procedure for q, in which a clause $\neg q$ is checked for induction relative to F_{k-1}. (**c**) shows the case in which relative induction of $\neg q$ does not hold and thus a predecessor cube s is found to be a CTI and extended into cube p that needs to be blocked in F_{k-1}. (**d**) shows the cube blocking procedure for p, in which a clause $\neg p$ is checked for induction relative to F_{k-2}. In (**e**), $\neg p$ is found to be inductive relative to F_{k-2}; thus, $\neg p$ undergoes inductive generalization and the result is used to refine the trace. In (**f**), after refining the trace, $\neg q$ becomes inductive relative to F_{k-1}; the proof obligation of q can thus be discharged by generalizing $\neg q$ and refining the trace once more. In (**g**), a fix-point is found by detecting $F_k = F_{k+1}$ after propagation.

Algorithm 7 Cube blocking procedure IC3: BLOCKCUBE(q, Q, $\boldsymbol{F_k}$).

Input: q a bad cube in F_k; Q the proof-obligation queue; $\boldsymbol{F_k}$ the trace.
Output: $\langle res, cex \rangle$ with $res \in \{\text{SUCCESS}, \text{FAIL}\}$ and cex a (possibly empty) initial path representing a counterexample.

1: **procedure** BLOCKCUBE(q, Q, $\boldsymbol{F_k}$)
2: Add proof obligation (q, k) to Q
3: **while** Q is not empty **do**
4: Extract proof obligation (q, j) with minimum j from Q
5: **if** $j > k$ or $q \not\models F_j$ **then continue**;
6: **if** $j = 0$ **then return** $\langle \text{FAIL}, \text{RECONSTRUCTCEX}(q) \rangle$
7: **if** $\exists s, s' \models F_{j-1} \wedge \neg q \wedge T \wedge q'$ **then**
8: $p \leftarrow \text{EXTEND}(s)$
9: Add proof obligations $(p, j-1)$ and (q, j) to Q
10: **else**
11: $c \leftarrow \text{GENERALIZE}(j, s, \boldsymbol{F_k})$
12: $F_i \leftarrow F_i \cup c$ for $0 < i \leq j$
13: Add proof obligation $(j+1, c)$ to Q
14: **return** $\langle \text{SUCCESS}, - \rangle$

In order to discharge a proof obligation, new ones may have to be recursively discharged. This can be carried out through a recursive implementation of the cube blocking procedure. However, in practice, handling proof obligations using a priority queue Q proved to be more efficient [50] and it is thus the commonly used approach in most state-of-the-art IC3 implementations. While blocking a cube, proof obligations (q, j) are extracted from Q and discharged for increasing values of j, ensuring that every predecessor of a bad cube q will be blocked in F_j ($j < k$) before q will be blocked in F_k. In the BLOCKCUBE(q, Q, $\boldsymbol{F_k}$) procedure, described in Algorithm 7, the queue of proof obligations is initialized with (q, k), encoding the fact that q must be blocked in F_k (line 2). Then, proof obligations are iteratively extracted from the queue and discharged (lines 3–14).

Prior to discharging a proof obligation (q, j), IC3 checks whether that proof obligation still needs to be discharged. It is in fact possible for an enqueued proof obligation to become discharged as a result of some previous proof-obligation discharging. To perform this check, the algorithm tests whether q is still included in F_j (line 5). This can be performed by posing the following SAT query:

$$SAT?(F_j \wedge q) \tag{58}$$

If q is in F_j (i.e., Query (58) is SAT), the proof obligation (q, j) still needs to be discharged. Otherwise, q has already been blocked in F_j and the procedure can move on to the next iteration.

If the proof obligation (q, j) still needs to be discharged, then IC3 checks whether F_j is the initial frame (line 6). If so, the states represented by q are initial states that can reach a violation of property P. Thus, a counterexample to P can be constructed by following the chain of proof obligations that led to $(q, 0)$. This is performed by means of the procedure RECONSTRUCTCEX(q), not described here. In that case, the procedure terminates with a failure and returns the counterexample found.

To discharge a proof obligation (q, j), i.e., to block a cube q in F_j, IC3 tries to derive a clause c such that $c \subseteq \neg q$ and c is inductive relative to F_{j-1}. Figure 8b visualizes this step. The *initiation* condition of induction ($I \rightarrow \neg q$) holds by construction, whereas the algorithm must check whether or not the *relative consecution* condition ($F_{j-1} \wedge \neg q \wedge T \rightarrow \neg q'$) holds. This can be performed by proving that the following SAT query is unsatisfiable (line 7):

$$SAT?(F_{j-1} \wedge \neg q \wedge T \wedge q') \tag{59}$$

If relative consecution holds (i.e., Query (59) is UNSAT), then the clause $\neg q$ is inductive relative to F_{j-1} and can be used to refine F_j, ruling out q (lines 10–13). To pursue a

stronger refinement of F_j, the inductive clause found undergoes a process called *inductive generalization* (line 11) prior to being added to the frame. Inductive generalization is carried out by the GENERALIZE$(j, q, \boldsymbol{F_k})$ procedure, described in Algorithm 8, which tries to minimize the number of literals in a clause $c = \neg q$ while maintaining its inductiveness relative to F_{j-1}, in order to preserve monotonicity.

The resulting clause is added not only to F_j, but also to every frame F_i, $0 < i < j$ (line 12). Doing so discharges the proof obligation (q, j) ruling out q from every F_i with $0 < i \leq j$. Since the sets F_i with $i > j$ are larger than F_j, q may still be present in one of them and $(q, j+1)$ may become a new proof obligation. To address this issue, Algorithm 7 adds $(q, j+1)$ to the proof-obligations queue (line 13). Figure 8e,f visualize this step.

Otherwise, if the relative consecution does not hold (see Figure 8c), there is a predecessor s of q in $F_{j-1} \wedge \neg q$. Such predecessors are called *counterexamples to the inductiveness* (CTIs). To preserve image approximation, before blocking a cube q in a frame F_j, every CTI of q must be blocked in F_{j-1} (Figure 8d). Therefore, the CTI s is first extended into a cube p (line 8), and then both proof obligations $(p, j-1)$ and (q, j) are added to the queue (line 9).

The GENERALIZE$(j, q, \boldsymbol{F_k})$ procedure (Algorithm 8) performs inductive generalization, which is a fundamental step of the algorithm. During inductive generalization, given a clause $\neg q$ relative inductive to F_{j-1}, IC3 tries to compute a clause c subset of $\neg q$ such that c is still inductive relative to F_{j-1}. Acting this way, and using c to refine the frames, the algorithm can block not only the original bad cube q but potentially also other states, which in turn may lead to faster convergence. Inductive generalization works by dropping literals from the input clause while maintaining relative inductiveness with respect to F_{j-1}. The procedure computes a minimal inductive sub-clause, i.e., a sub-clause that is inductive relative to F_{j-1} and no further literals can be dropped without forgoing inductiveness [52]. In the general case, though, finding a minimal inductive sub-clause is often inefficient [48] and, thus, most implementations of IC3 rely on an approximate version of such a procedure. Approximate inductive generalization is significantly less expensive, yet still able to drop a reasonable number of literals.

Algorithm 8 Iterative inductive generalization procedure: GENERALIZE $(j, s, \boldsymbol{F_k})$

Input: j a frame index; q a cube such that $F_{j-1} \wedge \neg q \wedge T \rightarrow \neg q'$; $\boldsymbol{F_k}$ the trace.
Output: c a clause such that $c \subseteq \neg q$ and $F_{j-1} \wedge c \wedge T \rightarrow c'$.

```
1: procedure GENERALIZE(j, q, F_k)
2:     c ← ¬q
3:     for all literals l in c do
4:         t ← c \ l
5:         if ∄s, s' ⊨ F_{j-1} ∧ t ∧ T ∧ ¬t' then
6:             if ∄s ⊨ I ∧ ¬t then
7:                 c ← t
8:     return c
```

In the GENERALIZE $(j, q, \boldsymbol{F_k})$ procedure, a clause c initialized with $\neg q$ (line 2) represents the current inductive sub-clause. For each literal of c, the candidate clause t is obtained by discarding that literal from c (line 4). Dropping literals from a relative inductive clause can violate both initiation and relative consecution conditions of induction. The candidate clause t must thus be checked for inductiveness relative to F_{j-1}. IC3 checks whether the relative consecution condition $(F_{j-1} \wedge t \wedge T \rightarrow t')$ keeps holding for t by posing the following SAT query:

$$SAT?(F_{j-1} \wedge t \wedge T \wedge \neg t') \tag{60}$$

If the relative consecution condition holds (i.e, Query (60) is UNSAT), the algorithm needs to prove the initiation condition of induction $(\mathcal{I} \rightarrow t)$. This check (line 6) can be performed either syntactically or semantically. If $\mathcal{I}$ can be described as a cube, it is enough to check whether at least one of the literals of $\mathcal{I}$ appears in t with opposite polarity. In such

a case, $\neg t$ does not intersect $\mathcal{I}$ and the initiation condition holds. Otherwise, the initiation of induction must be checked explicitly by posing the following SAT query:

$$SAT?(\mathcal{I} \wedge \neg t) \tag{61}$$

If Query (61) is UNSAT, the initiation condition holds for t. If both the relative consecution and the initiation conditions still hold for the candidate clause t, the current inductive sub-clause c can be updated with t (line 7).

The PROPAGATE ($\boldsymbol{F_k}$) procedure (Algorithm 9) manages the propagation phase. For every clause c of each frame F_j, with $0 \leq j < k-1$, it checks whether c can be pushed forward to F_{j+1}. This is performed by checking whether c is inductive relative to F_j (line 4). The procedure must check whether relative consecution $F_j \wedge c \wedge T \rightarrow c'$ holds, by answering the following SAT query:

$$SAT?(F_j \wedge c \wedge T \wedge \neg c') \tag{62}$$

If relative consecution holds (i.e., Query (62) is UNSAT), then c is relative inductive to F_j and can be pushed forward to F_{i+1}. Otherwise, c cannot be pushed forward and the algorithm moves to the next iteration.

Algorithm 9 Clause propagation procedure: PROPAGATE ($\boldsymbol{F_k}$).

Input: $\boldsymbol{F_k}$: the current trace.

1: **procedure** PROPAGATE($\boldsymbol{F_k}$)
2: **for** $j = 0$ to $k-1$ **do**
3: **for all** $c \in F_j$ **do**
4: **if** $\exists s, s' \models F_j \wedge c \wedge T \wedge \neg c'$ **then**
5: $F_j \leftarrow F_j \setminus \{c\}$
6: $F_{j+1} \leftarrow F_{j+1} \cup \{c\}$

Property Directed Reachability

Property Directed Reachability (PDR) is a variant implementation of IC3 proposed by Een et al. in [50]. PDR differs from the original IC3 implementation by Bradley [48] as follows:

- The trace is represented as sets of blocked cubes rather than learned clauses. Furthermore, to avoid duplication, PDR only stores a cube in the last frame where it holds. PDR also adds a special frame F_∞ which will hold cubes that have been proved unreachable from the initial states by any number of transitions.
- PDR uses a slightly modified trace semantics with respect to IC3, by not requiring the last frame of the trace to entail the property. Een et al. show that this behavior can be emulated by PDR by preprocessing the input system using a one-step target enlargement of P. This results in a small performance gain, simplifies the implementation and has the extra benefit that F_∞ is a proper invariant, which can be used to strengthen other proof engines, or exploited for synthesis. Furthermore, in principle, the same target-enlargement idea can be generalized and applied for any number $k \geq 1$ of steps.
- PDR proposes the use of ternary simulation as a method to shrink proof obligations before blocking them (implemented as the EXTEND procedure in Algorithm 7). In [50], ternary simulation is shown to have a big impact on performance.

7. IGR

IGR [53], *Interpolation with Guided Refinement*, is a model checking algorithm than can be seen as a variant of standard interpolation. It incorporates, within an interpolation scheme, explicit trace computation and refinement, images and cones simplification under observability don't care and guided cone unwinding/rewinding.

The main purpose is to improve the standard interpolation in order to support the incremental computation of reachable states sets and dynamic tuning of the backward unrolling from the target. Incremental data structures are used to enable the reuse of previ-

ously computed overapproximations and make interpolant-based algorithms better suited for the verification of multiple properties or for integration with abstraction/refinement approaches. Furthermore, maintaining overapproximated reachable state information in an incremental data structure allows the algorithm to dynamically adjust the bound of the BMC formulas checked during each traversal step, for better controlling the precision of the computed image overapproximations.

In order to better describe the proposed variation, it is necessary to introduce first the foundations on which it is built upon, in order to better characterize then the overall algorithm.

7.1. Incremental State Sets in Interpolation

In order to enable the reuse of previously computed interpolants, in IGR, a trace of overapproximations to reachable states is maintained and incrementally refined. Since interpolants are safe image overapproximations with respect to the property under verification, the trace maintained is safe as well.

At each i-th iteration of standard ITP's inner loop, from the refutation proof of a BMC formula, an overapproximation of the states reachable in i steps in the system is computed by extracting an interpolant. Such an interpolant is then discarded at the end of the iteration. Furthermore, when a spurious counterexample is found, the current forward traversal is interrupted, the backward cone from the target is unwound by one step and the forward traversal of reachable states restarts once again from the initial states. One of the focus point in IGR is to keep track of the overapproximations computed during each run of ApproxForwardTraversal, in order to enable their reuse in further iterations of the outer loop. It is thus necessary to extend the standard interpolation algorithm to keep track of a *trace* of reachable states. As the bound k of the cone increases, stronger overapproximations are computed at each traversed time frame and used to refine the trace.

A trace $\mathbf{F_k} = (F_0, \ldots, F_k)$ is used in order to keep track of previously computed overapproximations. From Lemma 3, it follows that each timeframe F_i of $\mathbf{F_k}$, with $0 \leq i < k$, is an overapproximation of the set of states that are reachable in exactly i steps. The trace is constructed so that it is safe with respect to P.

The proposed ITP variant, informally called IncrItpModelChecking, is sketched in Algorithms 10 and 11. The differences between the variant and standard interpolation are the following:

- The trace is initialized with $F_0 = \mathcal{I}$ before starting the first forward traversal (Algorithm 10, line 5).
- Each time the forward traversal reaches the end of the current trace, a new frame F_{i+1} is instantiated equal to $\top$ and added to the trace (Algorithm 11, lines 8–10).
- Every time a new interpolant, overapproximating states reachable in $i+1$ steps, is computed, the corresponding frame F_{i+1} in the trace is refined (Algorithm 11, lines 17).

Refinement is a strengthening step carried out by conjoining the previous set with a new term.

Algorithm 10 Top-level procedure of the proposed ITP variant that keeps track of the computed interpolants using a trace.

Input: $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ a transition system; P a property over V.
Output: $\langle res, cex \rangle$ with $res \in \{\text{SUCCESS}, \text{FAIL}\}$; cex a (possibly empty) initial path representing a counterexample.

1: **procedure** INCRITPMODELCHECKING($\mathcal{M}$, P)
2: **if** $\exists s_0 : s_0 \models \mathcal{I}(V) \wedge \neg P(V)$ **then**
3: **return** $\langle \text{FAIL}, (s_0) \rangle$
4: $\mathbf{F_k}[0] \leftarrow \mathcal{I}$
5: $k \leftarrow 1$
6: **while** true **do**
7: $\langle res, cex \rangle \leftarrow$ INCRAPPROXFORWARDTRAVERSAL($\mathcal{M}, P, \mathbf{F_k}, k$)
8: **if** *res* is UNREACH **then**
9: **return** $\langle \text{SUCCESS}, - \rangle$
10: **else if** *res* is REACH **then**
11: **return** $\langle \text{FAIL}, cex \rangle$
12: $k \leftarrow k + 1$

Algorithm 11 Inner procedure that keeps track of the computed interpolants using a trace.

Input: $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ a transition system; P a property over V; $\mathbf{F_k}$ a trace; k bound of a backward unrolling from the target.
Output: $\langle res, cex \rangle$ with $res \in \{\text{REACH}, \text{UNREACH}, \text{UNDEF}\}$; cex a (possibly empty) initial path representing a counterexample.

1: **procedure** INCRAPPROXFORWARDTRAVERSAL($\mathcal{M}$, P, $\mathbf{F_k}$, k)
2: $R \leftarrow F_0$
3: **if** $\exists \boldsymbol{\pi}^{0,k} \models F_0(V^0) \wedge T(V^0, V^1) \wedge Cone(1, k)$ **then**
4: **return** $\langle \text{REACH}, \boldsymbol{\pi}^{0,k} \rangle$
5: $i \leftarrow 0$
6: **while** $\top$ **do**
7: **if** $i = |\mathbf{F_k}|$ **then**
8: $\mathbf{F_k}[i+1] \leftarrow \top$
9: $A \leftarrow F_i(V^0) \wedge T(V^0, V^1)$
10: $B \leftarrow Cone(1, k)$
11: **if** $\exists \boldsymbol{\pi}^{0,k} \models A \wedge B$ **then**
12: **return** $\langle \text{UNDEF}, - \rangle$
13: **else**
14: $I \leftarrow \text{ITP}(A, B)$
15: $F_{i+1} \leftarrow F_{i+1} \wedge I$
16: **if** $\nexists s \models F_{i+1} \wedge \neg R$ **then**
17: **return** $\langle \text{UNREACH}, - \rangle$
18: $R \leftarrow R \vee F_{i+1}$
19: $i \leftarrow i + 1$

7.2. Frames and Cone Simplification

In order to keep cones and overapproximations of reachable states contained in size, the algorithms exploit an optimization that aims at simplifying the representation of frames and/or bad cones through ad hoc redundancy removal, exploiting the general notion of redundancy removal under *observability don't cares*. In order to understand the approach, let us denote *simplification under a care set* as the function SIMPLIFY(F, C), where F is a formula over V to be simplified and C is another formula over V to be used as a care set for the simplification of F. The care set is defined with respect to a reference formula G over $V \cup W$ in which F appears as a subformula, as follows.

Definition 88 (Care Set and Don't Care Set)**.** *Given a propositional formula F over V and another formula G over $V \cup W$ such that F is a subformula of G, we define the care set C_F^G of F with respect to G as the set of assignments over V under which the value of F affects the value of G. A care set for F with respect to G can be represented by the formula:*

$$C_F^G = G \oplus G[F \leftarrow \neg F] \tag{63}$$

The complement of C_F^G is called the don't care set *of F with respect to G. It represents the set of assignments over V under which the value of F does not affect the value of G.*

Simplification of a formula F under care set C_F^G with respect to a reference formula G can entail the application of any number of equivalence-preserving or strengthening transformations over F, as long as the following constraint is preserved:

$$G \equiv G[F \leftarrow \textsc{Simplify}(F, C_F^G)] \tag{64}$$

Computation of care sets and don't care sets can be costly [54]. Considering a conjunction $F = A \wedge B$ as a reference formula, the following lemma describes two straightforward ways to obtain care sets for either one of the conjoined formulas. We focus on B, since the same dual reasoning applies to A.

Lemma 8 (Care Sets for Conjunctive Reference Formulas)**.** *Given A and B, two propositional formulas over V, and given $F = A \wedge B$, then A is a care set for B with respect to F. Given C, a propositional formula over V, such that $A \rightarrow C$, then C is a care set for B with respect to F.*

Figure 9a,b illustrate the simplification of a formula B with respect to a reference formula $F = A \wedge B$ under care sets as defined by Lemma 8, in terms of sets of assignments. Given two propositional formulas $A = a$ and $B = (a \vee b)$, the knowledge that any assignment satisfying their conjunction $F = A \wedge B$ must be a satisfying assignment of either one can be used to simplify the other. Using A as a care set for B, since $F = a \wedge (a \vee b)$ is satisfied only by assignments satisfying A (i.e., assignments μ such that $\mu(a) = \top$) it is possible to simplify B through the injection of a constant $\top$ prior to conjoining it to A, obtaining $\top \vee b \equiv b$. The resulting formula after simplification $A \wedge \textsc{Simplify}(B, A) = a \wedge b$ is syntactically more compact than the equivalent $A \wedge B = a \wedge (a \vee b)$.

(**a**) (**b**)

Figure 9. Examples of simplification under a care set. (**a**) Simplification of B using A as care set. The Simplify procedure simplifies B without affecting its conjunction F with A. (**b**) Simplification of B using C as care set, with $A \rightarrow C$. The Simplify procedure simplifies B without affecting its conjunction F with A.

Despite the fact that many redundancy removal techniques could be used to perform simplification under a care set, e.g., latch correspondence, signal correspondence and equivalence to constants, most of them are too expensive to be performed at each forward traversal iteration within an ITP scheme. For the sake of having a fast operator, simplify is limited to the removal of equivalences between state variables, i.e., *latch correspondences*. Given a target formula B and a care set A, the Simplify operator identifies pairs of state variables (v_1, v_2) such that $A \rightarrow (v_1 \leftrightarrow v_2)$ and then B is simplified as $B[v_1 \leftarrow v_2]$.

Simplification under a care set can be used on frames during refinement steps (Algorithm 11, line 17), or to simplify the bad cone $Cone(1,k)$ prior to checking its intersection with the image of F_i (Algorithm 11, line 13). During forward overapproximated traversal, when checking whether the image of the current set of reachable states $F_i \wedge T$ intersects with the bad cone $Cone(1,k)$, it is possible to simplify $Cone(1,k)$ using any overapproximation of the states reachable in the next k transitions that is already in the trace. Each frame F_j in $\mathbf{F_k}$, with $i < j < i+k$, can be used as a care set to simplify $Cone(1,k)$. With TRACESIMPLIFY($Cone(1,k)$, $\mathbf{F_k}$, i, k) we identify the function applying SIMPLIFY($Cone(1,k), F_j$) for each $i < j < i+k$, i.e., the function applying latch correspondences substitution at each intermediate transition relation boundaries in $Cone(1,k)$. In this way, reachability information computed during previous iterations of the algorithm are used to simplify the formula before injecting it into the SAT solver.

7.3. Cone Unwinding and Rewinding

In standard interpolation, when finding a spurious counterexample, the current forward traversal is restarted from the initial states after the bad cone has been expanded by one step. IGR, instead, dynamically unwinds or rewinds the cone during forward traversal, in order to guarantee the refinement of some previously computed overapproximation of reachable states. The depth of the cone is therefore guided by the frames in the trace so that it can lead to a strengthening refinement for some of them.

Supposing that, at a given point during the execution of INCRITPMODELCHECKING, the algorithm has computed a trace $\mathbf{F_k}$, two approaches can be pursued with the goal of potentially expanding and/or refining $\mathbf{F_k}$:

Cone Unwinding. When the forward traversal hits the cone, it starts a new traversal at an intermediate step in order to guarantee the refinement of the trace.

Cone Rewinding. When the forward traversal hits the cone, it continues the traversal with iteratively smaller cones in order to refine and expand the trace.

Overall, guided cone unwinding/rewinding allows IGR to dynamically tune the unrolling from the target and therefore to have better control over the precision of the computed overapproximations (interpolants). In this respect, standard interpolation is too rigid, as overapproximations are always strengthened expanding the cone by one and restarting the traversal from scratch.

7.3.1. Cone Unwinding

At a given iteration i of the forward traversal, given k the bound of the cone, if the formula

$$F_i(V^0) \wedge T(V^0, V^1) \wedge Cone(1,k) \tag{65}$$

is SAT, then a possibly spurious counterexample is found. In such a scenario, standard interpolation would unwind the bad cone by one step and then start a new traversal from the initial states. When step i is then reached once again in the traversal, the overapproximation F_i could have been strengthened enough to exclude the previously found spurious counterexample. If that is not the case, the algorithm restarts the traversal again, incrementing k until either the spurious counterexample is excluded from the overapproximation or the counterexample is confirmed to be a concrete one. With cone unwinding, the cone is unwound of the minimum depth necessary to strengthen a frame F_j, with $0 < j \leq i$, in order to eliminate the spurious counterexample directly.

Upon encountering a spurious counterexample, the BMC-like problem described in Equation (65) is SAT. Therefore, the algorithm starts an iterative process checking BMC problems of fixed bound in which the cone is unwound by one step and the algorithm moves to the previous frame. Starting from frame F_i and cone $Cone(1,k)$, at each j-th iteration it considers the BMC-like problem:

$$F_{i-j}(V^0) \wedge T(V^0, V^1) \wedge Cone(1,k+j) \tag{66}$$

with $0 < j \leq i$. At each iteration it swaps an overapproximated image in F_{i-j} with an exact image in $Cone(1, k+j)$. If the BMC-like problem in Equation (66) is SAT at a given j, then F_{i-j} is not strong enough to refute the counterexample and the process iterates. Eventually, either it finds UNSAT thus refuting the spurious counterexample, or it reaches the initial states thus confirming the counterexample as a concrete one. In fact, for $j = i$ we have exactly $\text{BMC}(i+k)$. Assuming that it can find a j, with $0 < j < i$, such that Equation (66) is UNSAT, then it can restart the forward traversal from F_{i-j} with $Cone(1, k+j)$. This is guaranteed to generate an interpolant and therefore refine the current trace.

7.3.2. Cone Rewinding

When, at a given step i, the forward traversal hits the cone of depth k, rather than starting a new traversal with an unwound cone, the algorithm can optionally continue the current traversal with a sequence of iterations, called *refinement sequence*. During refinement, each iteration uses a cone of decreasing depth for which the safety of the current reachable state overapproximation is guaranteed.

Assume that, at a given step i, $F_i(V^0) \wedge T(V^0, V^1) \wedge Cone(1, k)$ is SAT. Since F_i was proved to be safe with respect to $Cone(1, k)$ at iteration $i-1$, there does not exist any counterexample of length at most k starting from i, and hence it is guaranteed that

$$F_{i+j}(V^0) \wedge T(V^0, V^1) \wedge Cone(1, k-j-1) \tag{67}$$

is UNSAT for each $0 \leq j < k-1$. A new interpolant I_j can be generated from each of such calls to be used to refine F_{i+j+1}. Alternatively, one could compute an interpolation sequence directly from the BMC call (67) and use it to refine the trace.

To summarize, once a cone of depth k is hit at step i of the forward traversal, the algorithm can compute a sequence of k interpolants, each representing an overapproximation of states reachable in $i+j+1$ transitions, with $0 \leq j < k-1$, that can be used to refine, or generate, the frames $(F_{i+1}, \ldots, F_{i+k})$ of the trace. The main purpose of such a refinement sequence is to let future traversals operate over more precise overapproximations of the reachable states.

On such premises, the overall IGR algorithm can now be presented.

The top-level procedure of IGR is reported in Algorithm 12. The algorithm first checks the safety of the initial states (lines 2–3) and initializes the trace $\mathbf{F_k}$ (line 4). Indexes i_{hit} and k_{hit} are also initialized (line 5–6) and they are used to keep track of the traversal step and cone depth at which a cone was hit during the previous traversal, if any. Past initialization, the outer loop starts iterating overapproximated forward traversals (lines 7–13). Procedure UNWIND is invoked first to seek the best frame at which to start the next traversal (line 8), to perform cone unwinding (as described in Section 7.3.1) starting from the step i_{hit} and cone depth k_{hit}, until either a concrete counterexample or a frame that could be refined by computing a new interpolant is found. In the first case, UNWIND returns a REACH result and a counterexample. The algorithm, thus, terminates with FAIL producing the counterexample as output (lines 9–10). In the second case, UNWIND returns an UNDEF result, a step i and a cone depth k to be used for the next traversal. Note that, during the first iteration, i_{hit} and k_{hit} are initialized to zero and one, respectively; therefore, UNWIND simply checks whether the initial states can reach the target in one transition. If no concrete counterexample is found, the algorithm starts a new forward overapproximated traversal by calling the IGRAPPROXFORWARDTRAVERSAL routine (line 11). Upon termination of such a procedure, if the result is UNREACH then an inductive invariant has been found during traversal and thus the algorithm terminates with SUCCESS (lines 12–13). Conversely, the cone was hit and the traversal procedure returns an UNDEF result together with the step i_{hit} and cone depth k_{hit} at which that occurred. In such a scenario, it means that a spurious counterexample was found during traversal and the algorithm then iterates to perform a new traversal.

On these premises, we can summarize the overall scheme of IGRMODELCHECKING as follows:

- Iteratively choose a starting frame F_i and a cone $Cone(1, k)$ to be unwound with guidance throughout the overapproximated trace $\mathbf{F_k}$.
- Initiate a forward traversal from F_i with $Cone(1, k)$ aiming at refining $\mathbf{F_k}$ and filtering out the latest spurious counterexample found within $F_{i_{hit}}$.

The two sub-tasks are performed by UNWIND and IGRAPPROXFORWARDTRAVERSAL, respectively.

Algorithm 12 Top-level procedure of IGR.

Input: $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ a transition system; P a property over V.
Output: $\langle res, cex \rangle$ with $res \in \{\text{SUCCESS}, \text{FAIL}\}$; cex a (possibly empty) initial path representing a counterexample.

1: **procedure** IGRMODELCHECKING($\mathcal{M}$, P)
2: **if** $\exists s_0 : s_0 \models \mathcal{I}(V) \wedge \neg P(V)$ **then**
3: **return** $\langle \text{FAIL}, (s_0) \rangle$
4: $\mathbf{F_k}[0] \leftarrow \mathcal{I}$
5: $i_{hit} \leftarrow 0$
6: $k_{hit} \leftarrow 1$
7: **while** true **do**
8: $\langle res, cex, i, k \rangle \leftarrow \text{UNWIND}(\mathcal{M}, P, \mathbf{F_k}, i_{hit}, k_{hit})$
9: **if** res is REACH **then**
10: **return** $\langle \text{FAIL}, cex \rangle$
11: $\langle res, i_{hit}, k_{hit} \rangle \leftarrow \text{IGRAPPROXFORWARDTRAVERSAL}(\mathcal{M}, P, \mathbf{F_k}, i, k)$
12: **if** res is UNREACH **then**
13: **return** $\langle \text{SUCCESS}, - \rangle$

Procedure IGRAPPROXFORWARDTRAVERSAL, described in Algorithm 13, performs an overapproximated forward traversal of reachable states. The procedure starts from a given frame in the trace while preserving safety with respect to a cone of a given depth. It first computes the current overapproximated set of reachable states at step i by disjoining the first i frames (line 2). During each iteration of the traversal loop (lines 6–29), the algorithm then proceeds in two different ways taking into account whether or not *cone rewinding* (see Section 7.3.2) has been triggered. If it has not been triggered, the algorithm performs a traversal step using a cone of bound k. Conversely, the procedure decreases the bound of the cone at each iteration to perform rewinding (lines 9–10). The procedure then performs cone simplification (line 12), as described in Section 7.2, and checks whether the current set of overapproximated reachable states R hits the cone (line 13), during each traversal step. In case of a hit, the algorithm saves the current step and cone bound in i_{hit} and k_{hit}, respectively, and triggers a refinement sequence (lines 14–16). Conversely, a new overapproximated image is computed through interpolation (lines 18) and the current frame F_i is refined and simplified (line 19) as described in Section 7.2. The algorithm then checks whether the overapproximation is an inductive invariant, returning UNREACH if that is the case (lines 20–21). If no inductive invariant has been found, the new set of overapproximated forward reachable states to be used for the next iteration is computed as $R \vee F_{i+1}$ (line 22). Each time the forward traversal reaches the end of the current trace, a new frame F_{i+1} is instantiated equal to $\top$ and added to the trace (lines 7–8). At the end of each iteration, if a given depth threshold D has been reached, the algorithm forces rewinding (line 24–27). When rewinding is triggered, either after finding a spurious counterexample or by force, the algorithm continues the traversal decreasing the cone bound at each iteration. When the cone has been completely rewound, the algorithm terminates returning UNDEF alongside the step and cone bound at which either a spurious counterexample was found (lines 15–16) or rewinding has been forced (lines 26–27).

The aforementioned D threshold heuristically controls activation of cone rewinding. Whenever $D = 0$, rewinding is always active, so the approach obtains a minimal refinement while mimicking the effect of interpolation sequences [55]. High values of D keep the k

value constant until a hit, mimicking a scheme which would result far closer to standard interpolation. Empirically, it occurs that small values tend to be usually better at small sequential depths, as they can enact more, relatively inexpensive, refinement steps.

Algorithm 13 Inner procedure that keeps track of the computed interpolants using a trace.

Input: $\mathcal{M} = \langle V, \mathcal{I}, T \rangle$ a transition system; P a property over V; $\mathbf{F_k}$ a trace; i start step of the traversal; k bound of a backward unrolling from the target.

Output: $\langle res, i_{hit}, k_{hit} \rangle$ with $res \in \{\text{UNREACH}, \text{UNDEF}\}$; i_{hit} the step (if any) at which the cone is hit during traversal; k_{hit} the depth of the cone hit.

1: **procedure** IGRAPPROXFORWARDTRAVERSAL($\mathcal{M}$, P, $\mathbf{F_k}$, k)
2: $R \leftarrow \bigvee_{j=0}^{i} F_j$
3: $rewind \leftarrow \bot$
4: $i_{hit} \leftarrow i$
5: $k_{hit} \leftarrow k$
6: **while** $\top$ **do**
7: **if** $i = |\mathbf{F_k}|$ **then**
8: $\mathbf{F_k}[i+1] \leftarrow \top$
9: **if** $rewind \wedge k > 0$ **then**
10: $k \leftarrow k - 1$
11: $A \leftarrow F_i(V^0) \wedge T(V^0, V^1)$
12: $B \leftarrow \text{TRACESIMPLIFY}(Cone(1,k), \mathbf{F_k}, i+1, k)$
13: **if** $\exists \boldsymbol{\pi}^{0,k} \models A \wedge B$ **then**
14: $rewind \leftarrow \top$
15: $i_{hit} \leftarrow i$
16: $k_{hit} \leftarrow k$
17: **else**
18: $I \leftarrow \text{ITP}(A, B)$
19: $F_{i+1} \leftarrow \text{SIMPLIFY}(F_{i+1}, I) \wedge I$
20: **if** $\nexists s \models F_{i+1} \wedge \neg R$ **then**
21: **return** $\langle \text{UNREACH}, -, - \rangle$
22: $R \leftarrow R \vee F_{i+1}$
23: $i \leftarrow i + 1$
24: **if** $\neg rewind \wedge i > D$ **then**
25: $rewind \leftarrow \top$
26: $i_{hit} \leftarrow i$
27: $k_{hit} \leftarrow k$
28: **else if** $rewind \wedge k = 0$ **then**
29: **return** $\langle \text{UNDEF}, i_{hit}, k_{hit} \rangle$

At each iteration, the UNWIND procedure described in Algorithm 14 computes i and k, starting from i_{hit} and k_{hit}, relative to the previous spurious counterexample. Following the strategy described in Section 7.3.1, the cone bound k is extended while i is decremented (lines 5–6), until either the algorithm encounters an UNSAT BMC check or the initial states are reached (line 4). In the former case, UNWIND managed to find a frame at which starting a traversal is expected to lead to a $\mathbf{F_k}$ refinement and to filtering out the last spurious counterexample found. The procedure then returns an UNDEF result alongside the corresponding values for i and k (line 9). In the latter case, the procedure has detected an actual counterexample as a side effect. The procedure then returns a REACH result alongside the counterexample (line 8).

Algorithm 14 Unwinding procedure of IGR.

Input: $\mathcal{M} \stackrel{\text{def}}{=} \langle V, \mathcal{I}, T \rangle$ a transition system; P a property over V; i_{hit} the step at which the cone was hit during traversal; k_{hit} the depth of the cone hit.

Output: $\langle res, cex, i, k \rangle$ with $res \in \{\text{UNDEF}, \text{REACH}\}$; cex a (possibly empty) initial path representing a counterexample; i step at which resume forward traversal from; k cone depth to use in the resumed traversal.

1: **procedure** UNWIND($\mathcal{M}$, P, i_{hit}, k_{hit})
2: $i \leftarrow i_{hit}$
3: $k \leftarrow k_{hit}$
4: **while** $i \geq 0 \wedge \exists \boldsymbol{\pi}^{0,k} \models F_i(V^0) \wedge T(V^0, V^1) \wedge Cone(1, k)$ **do**
5: $i \leftarrow i - 1$
6: $k \leftarrow k + 1$
7: **if** $i < 0$ **then**
8: **return** $\langle \text{REACH}, \boldsymbol{\pi}^{0,k}, -, - \rangle$
9: **return** $\langle \text{UNDEF}, -, i, k \rangle$

8. Comparative Analysis and Experimental Evaluation

In order to compare and evaluate the various model checking algorithms, we here present the data gathered taking into account our own implementation of said techniques on top of the PdTRAV tool [56], a state-of-the-art model checking academic tool.

The benchmark set considered was derived from past HWMCC suites [14,57], starting from the 2014 edition until the most recent one. The set includes hardware- as well as software-derived verification problems, mostly stemming from industrial-level verification instances (such as IBM and intel benchmarks).

Experiments were run on an Intel Core i7-1370, with 16 CPUs running at 3.4 GHz hosting a Ubuntu 22.04 LTS Linux distribution. All the experiment were run taking into account a time limit of 3600 s and a memory limit of 32 GB.

Experimental results are reported in Tables 1 and 2, which share the same structure. Both tables highlights solves, split between SAT and UNSAT instances (Result), for each main engine (Engine) of our verification suite. For each group of results, we provide the average number of memory elements (avg_l), input variables (avg_i) and gates (avg_g) of the benchmarks belonging to each set, to provide some insight into the size of the instances at hand. As summary statistics, we also provide average solving time, both in term of CPU time (avg_t) and wall-clock time (avg_r), and memory requirements (avg_m) for each group. Table 2 provides a subset of the results of Table 1, filtering out easier instances, i.e., instances that require 60 seconds or less to be solved.

Table 1. Results on the full benchmark set of the number of instances solved by different engines.

Engine	Result	Solutions	avg_i	avg_l	avg_g	avg_t [s]	avg_r [s]	avg_m [MB]
BDD	SAT	12	16.25	307.33	9458.17	1014.53	175.71	1518.29
	UNS	91	83.81	222.68	2723.90	1959.57	353.31	1921.69
BMC	SAT	185	5811.45	16,011.02	201,773.13	1409.58	330.53	3314.58
COM	SAT	1	122.00	381.00	2526.00	0.59	0.69	58.60
	UNS	121	6500.72	24,608.60	215,393.50	28.80	29.16	462.13
IGR	UNS	32	1367.50	1463.75	42,964.44	3359.04	657.87	4796.34
ITP	UNS	86	2728.55	6079.29	74,085.99	3294.79	670.03	3923.90
LMS	UNS	19	47.26	584.32	3687.63	731.97	124.60	1464.49
PDR	SAT	43	579.37	1188.88	18,866.86	496.82	95.26	1482.05
	UNS	233	558.95	3599.55	36,061.80	462.56	97.19	1792.21
RED	UNS	2	203.00	271.00	4840.00	1.54	2.64	113.35
SIM	SAT	36	1011.22	4596.92	45,924.17	224.19	78.73	1742.41
SYN	UNS	91	596.62	819.20	14,008.29	10.36	7.69	395.29

Table 2. Results on the filtered benchmark set of the number of instances solved by different engines, discarding easy-to-solve instances.

Engine	Result	Solutions	avg_i	avg_l	avg_g	avg_t [s]	avg_r [s]	avg_m [MB]
BDD	SAT	10	18.30	334.50	10,029.30	1217.21	210.41	1698.32
	UNS	46	77.20	315.65	2711.20	3869.45	695.41	2972.38
BMC	SAT	157	6580.46	17,892.90	229,695.41	1657.67	387.19	3698.95
COM	UNS	17	651.53	8537.94	71,562.18	151.05	148.73	258.09
IGR	UNS	32	1367.50	1463.75	42,964.44	3359.04	657.87	4796.34
ITP	UNS	81	2850.21	6407.47	78,112.98	3495.62	710.60	4096.33
LMS	UNS	14	48.29	674.14	4114.29	986.12	166.85	1645.11
PDR	SAT	14	1216.57	1452.29	37,274.07	1493.70	274.34	2255.86
	UNS	98	1060.56	6139.47	63,953.65	1075.83	220.33	2591.29
SIM	SAT	24	853.25	5563.62	52,609.83	323.82	112.88	2034.62
SYN	UNS	5	4104.20	2679.60	76,255.80	150.76	103.28	840.24

In the tables one can find both SAT-based model checking techniques, which are the focus of this paper, such as IC3 in its PDR variant, BMC, induction-based approaches exploiting lemmas (LMS), interpolation and IGR, as well as some solutions obtained through auxiliary techniques, such as BDD-based reachability, logic synthesis, simulation and combinatorial techniques to circuit manipulation and checking, which go past the scope of the current work. It is worth remembering that the portfolio-based approach awards the solution for a specific instance to the first engine capable of solving the corresponding verification task, for the sake of efficiency. In such a scenario, after (at least) one engine manages to solve a problem, all the concurrent running task are brought to a halt, in order to spare resources.

As mentioned before, there is not currently a definitive winning strategy when it come to bit-level hardware model checking, so portfolio approaches are the de facto standard in the field. Since its introduction, IC3/PDR has proven itself time and time again as one of the most proficient strategies in the state of the art, as also shown by the results here provided. Nevertheless, its contribution need to be complemented with interpolation-based strategies in order to provide better coverage. This is far more evident when we shift the focus to harder-to-solve instances, in which the subset of benchmarks solved by interpolation-based strategies stays basically the same, whereas the number significantly decreases for IC3. IC3 proves itself a high-performing engine, with a significant edge on easier and mid-tier instances. Interpolation-based techniques, in turn, complement it providing orthogonal approaches. The same consideration holds for BMC-based approaches, that focus on falsification with the aim of finding an actual counterexample to the property under verification. Their targets stay basically unchanged, with a scope on instances with deeper unrollings of the corresponding transition relation, which other engines may struggle to tackle.

When taking into account BDD-based approaches, which are not directly the focus of this paper, one can take into account both similarities and differences with their SAT-based counterparts. On the one hand, they both try to address problems encoded in Boolean form but where SAT checkers just need to focus on finding a single satisfying assignment for the formula under verification, BDD-based ones need to encode a function describing all the satisfying assignments. Once such an encoding is available, BDDs are capable of performing tasks which are not supported by SAT-based solutions, such as focusing on optimality given a cost function, provide counting, or supporting variable quantification, at the cost of being able to build such a data structure, though. In general SAT checkers relying on some variant of the DPLL algorithm outperform BDD-based solutions; nevertheless, there are niche scenarios in which BDDs are still the winning strategies, such as specific instances of equivalence and parity checking.

Complementary to all of the techniques referenced above, we have logic synthesis, simulation and combinatorial-based approaches which tackle verification problems from a more circuital and structural point of view. They tend to be applicable to smaller instances

or instances with very specific patterns and a large number of equivalent elements, which can be collapsed in a few equivalence classes and/or abstracted away from the actual model under verification.

Practical Considerations Concerning Model Checking Algorithms

The algorithms we have described, falling into the SAT-based and symbolic model checking category, are usually quite resilient to the state space explosion problem, which can be problematic when dealing with large designs, comprising millions, or more, variables and constraints. In such a scenario, though, the trade-off has to be taken into account, which lies in the inherent time complexity, as SAT-based problems are known to be NP-complete.

One the one hand, we have that worst-case complexity indeed falls into exponential time; on the other hand, we have that, exploiting the proper modeling of systems, representation of data and simplification steps, SAT-based model checking algorithms are applicable to real-world scenarios.

One of the major issues with tuning resources for running verification instances is that it is almost impossible, in the general case, to foresee what would be the appropriate amount of time and memory to dedicate to a given task. The experimental evaluation proposed above follows the same setup of model checking competitions to provide comparable results with a well-defined baseline, but at the same time such a setup is nothing more than an executive choice on the matter.

It is not uncommon for industrial-level instances and more complex benchmarks to have to account for timeouts in the order of days, or more, in order to hopefully obtain usable results. Let us also point out once more how there is absolutely no guarantee that any of the engines may actually be able to solve a given instance, regardless of the time spent on the problem. On these premises, there are situations in which the aim is to go *as deep as possible* in the model exploration, trying to verify as much of the behavior as possible, without any claim of completeness, such as in pure BMC-based runs, where engines are left to run until resource depletion, either in terms of time or memory.

Because of this, orthogonal to the research and development of new techniques, a vast amount of research is targeted at improving the existing base of knowledge in order to make current techniques better performing and more applicable. Such an effort may be directed, among other things, at improving formalism and encoding styles and rules for the models [58], improving SAT-solver management strategies [49] or introducing different strategies for task management or exploiting partial results in a collaborative fashion among engines in order to increase their chance of success [36,37].

A far more in-depth analysis of practical solutions and techniques to improve the applicability and scalability of SAT-based verification is presented in [59].

9. Conclusions

This paper provides an in-depth overview of the most common SAT-based algorithms usually applied in bit-level hardware model checking scenarios. In the last two decades, SAT-based model checking brought about a leap in the performance and scalability of symbolic model checking algorithms. SAT-based model checking tools have managed to almost supplant BDD-based ones in both industry and academia. Model checking algorithms based on SAT solving, however, still suffer from non-negligible scalability issues when confronted with the complexity of many industrial-scale designs. Improving the performance and scalability of such algorithms is a very challenging, yet very active, research path.

This paper strives to provide both a thorough overview of the background required to understand the topics at hand, and a complete description of the aforementioned algorithms and their intricacies. Based on the current state of the art, a selection of the most relevant methods has been identified and described, taking into account both bounded and unbounded model checking, with a focus on safety properties. For the sake of generality, this work provides a description of each algorithm from a theoretical standpoint, trying

to avoid details which would turn out to be too implementation oriented in favour of the soundness and comprehensiveness of their description and discussion.

As mentioned in the introductory section, as of today, there is no a single winning strategy applicable to the broad range of verification problems one may face. It has thus become almost mandatory to tackle verification tasks using a portfolio of techniques, running in a concurrent/collaborative fashion, in order to maximize the chance of success. Furthermore, additional factors, such as model pre-processing and simplification, property management, engine orchestration, etc., play a role as important as the actual model checking algorithm of choice in determining the success of the verification procedure. This is the main reason why the current research path in this field usually reaches across several different directions, targeting different aspects of the verification chain, rather than focusing on a single specific step. Nevertheless, it is absolutely crucial to develop a firm and sound understanding of the available techniques in order to properly orchestrate verification efforts.

For the future, the current path of research is aimed at supporting hierarchical verification for top-down design, embedding support for verification in the whole design chain, starting from specification languages. Currently, there is a need for being able to write a verifiable high-level design, that can then be refined and implemented, while guaranteeing that it is still consistent with its abstraction. The main aim is for properties that have been verified at a given step of the design chain to hold at any other successive level of refinement. Such a desideratum leads to hybrid and top-down/bottom-up design methodologies, with emphasis on verification as grounding for correctness, as well as introducing the need for some sort of certificate, to guarantee the correctness of each step, transformation and refinement involved in the process. Alongside hierarchical verification, the integration of model checking within design becomes even more pressing, where model checkers represent auxiliary tools within the design chain, to support and validate correctness. Despite its potential usefulness, the adoption of model checking in industrial scenarios is still an ongoing process, where funding and methodology changes are the main limiting factors, in conjunction with the absence of a definitively best approach at tackling verification problems. Because of this, we can foresee that for the coming years portfolio-based approaches, exploiting divide-and-conquer strategies among different verification engines, will still be the most consistent way to address such tasks.

Author Contributions: Conceptualization, P.E.C., G.C., M.P. and P.P.; methodology, M.P. and P.P.; software, G.C.; validation, M.P. and P.P.; formal analysis, P.E.C. and G.C.; investigation, G.C., M.P. and P.P.; resources, P.E.C. and G.C.; data curation, M.P. and P.P.; writing—original draft preparation, M.P. and P.P.; writing—review and editing, P.E.C., G.C., M.P. and P.P.; visualization, M.P. and P.P.; supervision, P.E.C. and G.C.; project administration, P.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Acknowledgments: This work was supported in part by Partenariato Esteso-RESTART "RESearch and innovation on future Telecommunications systems and networks, to make Italy more smART"-PE00000001.

Conflicts of Interest: The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Abbreviations

The following abbreviations are used in this manuscript:

BMC	Bounded Model Checking
CNF	Conjunctive Normal Form
CTI	Counterexample To Induction
HW	Hardware
HWMCC	Hardware Model Checking Competition
ITP	Interpolant
IC3	Incremental Construction of Inductive Clauses for Indubitable Correctness
IGR	Interpolation with Guided Refinement
MC	Model Checking
PDR	Property Directed Reachability
RTL	Register Transfer Level
UMC	Unbounded Model Checking
WFF	Well-Formed Formula

References

1. Foster, H. 2022 Wilson Research Group FPGA Functional Verification Trends. Available online: https://www.innofour.com/static/default/files/documents/pdf/fpga-trend-report_2022-wilson-research-verification-study_hfoster.pdf (accessed on 6 June 2024).
2. Rashinkar, P.; Paterson, P.; Singh, L. *System-on-a-Chip Verification: Methodology and Techniques*; Springer: Berlin/Heidelberg, Germany, 2013.
3. Kaufmann, M.; Moore, J.S. Some key research problems in automated theorem proving for hardware and software verification. *Math. J. Span. R. Acad. Sci.* **2004**, *98*, 181–195.
4. Boyer, R.S.; Moore, J.S. *A Computational Logic Handbook*, 1st ed.; Academic Press Professional, Inc.: San Diego, CA, USA, 1988.
5. Harrison, J. Theorem Proving for Verification (Invited Tutorial). In Proceedings of the 20th International Conference on Computer Aided Verification, CAV, Princeton, NJ, USA, 7–14 July 2008; Volume 5123, pp. 11–18. [CrossRef]
6. Kuehlmann, A.; Somenzi, F.; Hsu, C.J.; Bustan, D. Equivalence Checking. In *Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology*; CRC Press: Boca Raton, FL, USA, 2016; pp. 77–108. [CrossRef]
7. Clarke, E.M.; Emerson, E.A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*; Kozen, D., Ed.; Springer: Berlin/Heidelberg, Germany, 1981; Volume 131, pp. 52–71. [CrossRef]
8. Clarke, E.M.; Grumberg, O.; Peled, D.A.; Veith, H. *Model Checking*, 2nd ed.; MIT Press: Cambridge, MA, USA, 2018.
9. Bryant, R.E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* **1986**, *35*, 677–691. [CrossRef]
10. Mishchenko, A. ABC: A System for Sequential Synthesis and Verification. Available online: http://people.eecs.berkeley.edu/~alanmi/abc/ (accessed on 6 June 2024).
11. Cabodi, G.; Nocco, S.; Quer, S. PdTRAV: Politecnico di Torino Reachability Analysis & Verification. Available online: https://github.com/polito-fmgroup/pdtools (accessed on 6 June 2024)
12. Griggio, A.; Roveri, M.; Tonetta, S. Certifying proofs for SAT-based model checking. *Form. Methods Syst. Des.* **2021**, *57*, 178–210. [CrossRef]
13. Yu, E.; Froleyks, N.; Biere, A.; Heljanko, K. Towards Compositional Hardware Model Checking Certification. In Proceedings of the Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, 24–27 October 2023; pp. 1–11. [CrossRef]
14. Biere, A.; Jussila, T. The Model Checking Competition. Available online: http://fmv.jku.at/hwmcc (accessed on 8 June 2024).
15. Biere, A. Tutorial on World-Level Model Checking. In Proceedings of the 2020 Formal Methods in Computer Aided Design (FMCAD), Online, 21–24 September 2020; p. 1. [CrossRef]
16. Palena, M. Exploiting Boolean Satisfiability Solvers for High Performance Bit-Level Model Checking. Ph.D. Thesis, Politecnico di Torino, Turin, Italy, 2017.
17. Pasini, P. Improving Bit-Level Model Checking Algorithms for Scalability through Circuit-Based Reasoning. Ph.D. Thesis, Politecnico di Torino, Turin, Italy, 2017.
18. Clarke, E.M.; Henzinger, T.A.; Veith, H.; Bloem, R. *Handbook of Model Checking*, 1st ed.; Springer: Berlin/Heidelberg, Germany, 2018.
19. Biere, A.; Heule, M.; van Maaren, H.; Walsh, T. (Eds.) *Handbook of Satisfiability*, 2nd ed.; Frontiers in Artificial Intelligence and Applications; IOS Press: Amsterdam, The Netherlands, 2021; Volume 336. [CrossRef]
20. Seshia, S.A.; Hu, S.; Li, W.; Zhu, Q. Design Automation of Cyber-Physical Systems: Challenges, Advances, and Opportunities. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2017**, *36*, 1421–1434. [CrossRef]
21. Barwise, J. (Ed.) *Handbook of Mathematical Logic*; Elsevier: Amsterdam, The Netherlands, 1977.
22. Hunter, G. *Metalogic: An Introduction to the Metatheory of Standard First Order Logic*, 1st ed.; Macmillan Student Editions; University of California Press: Berkeley, CA, USA, 1973.
23. Hopcroft, J.E.; Motwani, R.; Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*, 3rd ed.; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2006.

24. Cook, S.A. The Complexity of Theorem-proving Procedures. In *Third Annual ACM Symposium on Theory of Computing*; Harrison, M.A., Banerji, R.B., Ullman, J.D., Eds.; ACM: New York, NY, USA, 1971; pp. 151–158. [CrossRef]
25. Davis, M.; Logemann, G.; Loveland, D. A Machine Program for Theorem-proving. *Commun. ACM* **1962**, *5*, 394–397. [CrossRef]
26. Marques Silva, J.P.; Sakallah, K.A. Grasp—A New Search Algorithm for Satisfiability. In Proceedings of the The Best of ICCAD: 20 Years of Excellence in Computer-Aided Design, San Jose, CA, USA, 10–14 November 1996; pp. 73–89. [CrossRef]
27. Huang, J. The Effect of Restarts on the Efficiency of Clause Learning. In Proceedings of the 20th International Joint Conference on Artifical Intelligence, IJCAI, Hyderabad, India, 6–12 January 2007; pp. 2318–2323.
28. Biere, A. Preprocessing and Inprocessing Techniques in SAT. In Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing, HVC, Haifa, Israel, 6–8 December 2011; p. 1. [CrossRef]
29. Beame, P.; Kautz, H.; Sabharwal, A. Towards Understanding and Harnessing the Potential of Clause Learning. *J. Artif. Intell. Res.* **2004**, *22*, 319–351. [CrossRef]
30. Goldberg, E.; Novikov, Y. Verification of Proofs of Unsatisfiability for CNF Formulas. In Proceedings of the 2003 Design, Automation and Test in Europe Conference, DATE, Munich, Germany, 3–7 March 2003; p. 10886. [CrossRef]
31. Zhang, L.; Malik, S. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In Proceedings of the 2003 Design, Automation and Test in Europe Conference, DATE, Munich, Germany, 3–7 March 2003; pp. 10880–10885. [CrossRef]
32. Wetzler, N.; Heule, M.J.H.; Hunt, W.A. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In Proceedings of the Theory and Applications of Satisfiability Testing—SAT 2014: 17th International Conference, Vienna, Austria, 14–17 July 2014; Volume 8561, pp. 422–429. [CrossRef]
33. Kripke, S.A. A completeness theorem in modal logic. *J. Symb. Log.* **1959**, *24*, 1–14. [CrossRef]
34. Pnueli, A. The Temporal Logic of Programs. In Proceedings of the 18th Annual Symposium on Foundations of Computer Science, FOCS, Providence, RI, USA, 31 October–2 November 1977; pp. 46–57. [CrossRef]
35. Clarke, E.M.; Emerson, E.A.; Sistla, A.P. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *Acm Trans. Program. Lang. Syst.* **1986**, *8*, 244–263. [CrossRef]
36. Cabodi, G.; Camurati, P.E.; Loiacono, C.; Palena, M.; Pasini, P.; Patti, D.; Quer, S. To split or to group: From divide-and-conquer to sub-task sharing for verifying multiple properties in model checking. *Int. J. Softw. Tools Technol. Transf.* **2018**, *20*, 313–325. [CrossRef]
37. Dureja, R.; Baumgartner, J.; Kanzelman, R.; Williams, M.; Rozier, K.Y. Accelerating Parallel Verification via Complementary Property Partitioning and Strategy Exploration. In Proceedings of the 2020 Formal Methods in Computer Aided Design (FMCAD), Online, 21–24 September 2020; pp. 16–25. [CrossRef]
38. Biere, A.; Cimatti, A.; Clarke, E.; Zhu, Y. Symbolic Model Checking without BDDs. In Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS, Amsterdam, The Netherlands, 22–28 March 1999; Volume 1579, pp. 193–207. [CrossRef]
39. Sheeran, M.; Singh, S.; Stålmarck, G. Checking Safety Properties Using Induction and a SAT-Solver. In Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design, FMCAD, Austin, TX, USA, 1–3 November 2000; Volume 1954, pp. 127–144. [CrossRef]
40. Craig, W. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *J. Symb. Log.* **1957**, *22*, 269–285. [CrossRef]
41. Huang, G. Constructing Craig Interpolation Formulas. In Proceedings of the First Annual International Conference on Computing and Combinatorics, COCOON, Xi'an, China, 24–26 August 1995; Volume 959, pp. 181–190. [CrossRef]
42. Krajícek, J. Interpolation Theorems, Lower Bounds for Proof Systems, and Independence Results for Bounded Arithmetic. *J. Symb. Log.* **1997**, *62*, 457–486. [CrossRef]
43. Pudlák, P. Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations. *J. Symb. Log.* **1997**, *62*, 981–998. [CrossRef]
44. McMillan, K.L. Interpolation and SAT-Based Model Checking. In Proceedings of the 15th International Conference on Computer Aided Verification, CAV, Boulder, CO, USA, 8–12 July 2003; Volume 2725, pp. 1–13. [CrossRef]
45. Vizel, Y.; Nadel, A.; Ryvchin, V. Efficient generation of small interpolants in CNF. *Form. Methods Syst. Des.* **2015**, *47*, 51–74. [CrossRef]
46. Gurfinkel, A.; Vizel, Y. DRUPing for Interpolants. In Proceedings of the 14th International Conference on Formal Methods in Computer-Aided Design, FMCAD, Lausanne, Switzerland, 21–24 October 2014; pp. 99–106. [CrossRef]
47. Cabodi, G.; Camurati, P.E.; Palena, M.; Pasini, P.; Vendraminetto, D. Reducing Interpolant Circuit Size Through SAT-Based Weakening. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2020**, *39*, 1524–1531. [CrossRef]
48. Bradley, A.R. SAT-Based Model Checking without Unrolling. In Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI, Austin, TX, USA, 23–25 January 2011; Volume 6538, pp. 70–87. [CrossRef]
49. Cabodi, G.; Camurati, P.E.; Mishchenko, A.; Palena, M.; Pasini, P. SAT solver management strategies in IC3: An experimental approach. *Form. Methods Syst. Des.* **2017**, *50*, 39–74. [CrossRef]
50. Eén, N.; Mishchenko, A.; Brayton, R. Efficient Implementation of Property Directed Reachability. In Proceedings of the 11th International Conference on Formal Methods in Computer-Aided Design, FMCAD, Austin, TX, USA, 30 October–2 November 2011; pp. 125–134.

51. Chockler, H.; Ivrii, A.; Matsliah, A.; Moran, S.; Nevo, Z. Incremental Formal Verification of Hardware. In Proceedings of the 11th International Conference on Formal Methods in Computer-Aided Design, FMCAD, Austin, TX, USA, 30 October–2 November 2011; pp. 135–143.
52. Bradley, A.R.; Manna, Z. Checking Safety by Inductive Generalization of Counterexamples to Induction. In Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design, FMCAD, Austin, TX, USA, 11–14 November 2007; pp. 173–180. [CrossRef]
53. Cabodi, G.; Camurati, P.E.; Palena, M.; Pasini, P. Interpolation with guided refinement: Revisiting incrementality in SAT-based unbounded model checking. *Form. Methods Syst. Des.* **2022**, *60*, 117–146. [CrossRef]
54. Mishchenko, A.; Brayton, R.K. SAT-Based Complete Don't-Care Computation for Network Optimization. In Proceedings of the 2005 Design, Automation and Test in Europe Conference, DATE, Munich, Germany, 7–11 March 2005; pp. 412–417. [CrossRef]
55. Cabodi, G.; Nocco, S.; Quer, S. Interpolation sequences revisited. In Proceedings of the Design, Automation and Test in Europe, DATE 2011, Grenoble, France, 14–18 March 2011; pp. 316–322. [CrossRef]
56. Cabodi, G.; Nocco, S.; Quer, S. Benchmarking a model checker for algorithmic improvements and tuning for performance. *Form. Methods Syst. Des.* **2011**, *39*, 205–227. [CrossRef]
57. Cabodi, G.; Loiacono, C.; Palena, M.; Pasini, P.; Patti, D.; Quer, S.; Vendraminetto, D.; Biere, A.; Heljanko, K.; Baumgartner, J. Hardware Model Checking Competition 2014: An Analysis and Comparison of Solvers and Benchmarks. *J. Satisf. Boolean Model. Comput.* **2016**, *9*, 135–172. [CrossRef]
58. Tseitin, G.S. On the Complexity of Derivation in Propositional Calculus. In *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*; Springer: Berlin/Heidelberg, Germany, 1983; pp. 466–483. [CrossRef]
59. Ganai, M.K.; Gupta, A. *SAT-Based Scalable Formal Verification Solutions*; Series on Integrated Circuits and Systems; Springer: Berlin/Heidelberg, Germany, 2007. [CrossRef]