Inductive Validity Cores

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Elaheh Ghassabani

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Micael W. Whalen and Mats P. E. Heimdahl

Sept., 2018

Acknowledgements

This thesis is a milestone in four joyful years of work with amazing UMN Critical Systems Group (CriSys). My experience at UMN has been nothing short of spectacular. Since my first day on August 20th, 2014 I have felt at home at UMN with meeting my warm-hearted and caring advisors and friends. I have been given unique opportunities and taken advantage of them. This includes being part of the cutting-edge projects while serving as a graduate research assistant at CriSys, attending a lot of top-tier conferences, meeting with the best researchers in my field, and on top of all, learning from many distinguished and knowledgeable professors at UMN. I would like to extend thanks to the many people who so generously contributed to the work presented in this thesis.

Special mention goes to my enthusiastic supervisor, Michael Whalen. My PhD has been an amazing experience and I thank Michael, not only for his tremendous academic support, but also for giving me so many wonderful opportunities. I would like to express my sincere gratitude to him for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge.

Similar, profound gratitude goes to Mats Heimdahl, who has been a truly supportive co-advisor. I am particularly indebted to him for his constant faith in my work and for his support. I have very fond memories of my time with Michael and Mats, who have been like my family all these years. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my graduate studies.

Besides my advisors, I would like to thank the rest of my thesis committee: Prof. Stephen McCamant, Prof. Marc Riedel, and Prof. Eric Van Wyk, for their insightful comments and encouragement to widen my research from various perspectives.

I thank my fellow labmates for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last four years. Also I thank the wonderful people in other organizations I had the opportunity to work with; in particular, I am grateful to Dr. Andrew Gacek and Jaroslav Bendik.

Last but not least, I would like to thank my loving family: my parents and my sisters for supporting me spiritually throughout writing this thesis and my my life in general.

Abstract

Symbolic model checkers can construct proofs of properties over very complex models. However, the results reported by the tool when a proof succeeds do not generally provide much insight to the user. It is often useful for users to have traceability information related to the proof: which portions of the model were necessary to construct it. This traceability information can be used to diagnose a variety of modeling problems such as overconstrained axioms and underconstrained properties, and can also be used to measure *completeness* of a set of requirements over a model.

We propose the notion of *inductive validity cores* (IVCs), which are intended to trace a property to a *minimal* set of model elements necessary for proof. Such cores are not unique, and algorithms for efficiently producing both single IVC and all IVCs are presented. IVCs can be used for several interesting analyses, including regression analysis for testing/proof, determination of the minimum (as opposed to minimal) number of model elements necessary for proof, the diversity examination of model elements leading to proof, and analyzing fault tolerance.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software has become an integral part of our daily life and is being used in various environments (such as homes, hospitals, factories) and application areas (such as medical devices, aircraft flight control, weapons, and nuclear systems) where failure could lead to loss of life, financial loss, or environmental damage. It is vital to verify the soundness and safety of such critical applications. *Formal verification*, the process of mathematically proving or disproving the correctness of a system with respect to certain requirements or properties, is increasingly applied to critical systems to ensure they work in all cases.

One of the most successful and powerful methods for formal verification is symbolic model checking. Symbolic model checking using induction-based techniques such as IC3/PDR [1] and k-induction [2] can often determine whether safety properties hold of complex finite or infinite-state systems. Model checking tools are attractive both because they are automated, requiring little or no interaction with the user, and if the answer to a correctness query is negative, they provide a counterexample to the satisfaction of the property. These counterexamples can be used both to illustrate subtle errors in complex hardware and software designs [3–5] and to support automated test case generation [6, 7].

In the event that a property is proved, however, it is not always clear what level of assurance should be invested in the result. It is well known that issues such as vacuity [8] can cause verification to succeed despite errors in a property specification or in the model. Even for non-vacuous specifications, it is possible to over-constrain the specification of the *environment* in the model such that the implementation will not work in the actual operating environment. Given that these kinds of analyses are performed for safety- and security-critical software, these issues can lead to overconfidence in the behavior of the fielded system. In such cases, a system or subsystem component will not exhibit the expected behavior in its intended operating environment, which may bring about catastrophic losses. Achieving a proof in verification of safety requirements is not enough: careful scrutiny of the property of interest, the model, and the assumptions (axioms) used during proof must be performed. For these reasons, the level of feedback provided by the tools to the user in the event of a proof is important.

Inductive Validity Cores (IVCs) offer an explanation as to why a property is satisfied by a model in a formal and human-understandable way. Informally, if a model is viewed as a conjunction of constraints, a minimal IVC (MIVC) is a set of constraints that is sufficient to construct a proof such that if any constraint is removed, the property is no longer valid. IVCs and MIVCs can be used for several purposes, including performing traceability between specification and design elements, assessing model coverage, and explaining unsatisfiable test obligations when using model checkers for test case generation.

1.1 Objectives and Significance

In this thesis, we are specifically concerned with the scenarios where a model checker establishes the correctness proof of a given property. When it comes to verification, if the answer to a correctness query is positive, most tools provide no further information. The *objective of this dissertation* is to provide traceability information that explains a proof, in much the same way that a counterexample explains a negative result. Such an explanation should be both formal and human-understandable. This research will add to the usability of the symbolic model checkers by equipping the tools with a mechanism to show why a proved property is valid.

Reasoning about the proofs is not a new idea: UNSAT cores [9] provide the same kind of information for individual SAT or SMT queries, and this approach has been lifted to bounded analysis for Alloy in [10]. What we propose is a generic and efficient mechanism for extracting supporting information, similar to an UNSAT core, from the proofs of safety properties using inductive techniques such as PDR and k-induction. Because many properties are not themselves inductive, these proof techniques introduce lemmas as part of the solving process in order to strengthen the properties and make them inductive. Our approach, which we call *inductive validity cores* (IVCs), allows efficient, accurate, and precise extraction of model elements necessary even in the presence of such auxiliary lemmas. The idea lifts UNSAT cores [9] to the level of sequential model checking algorithms using induction. Informally, if a model is viewed as a conjunction of constraints, a minimal IVC (MIVC) is a set of constraints that is sufficient to construct a proof such that if any constraint is removed, the property is no longer valid.

The IVC idea facilitates several useful system analyses/engineering tasks. Specifically, it is useful when the validity of a safety requirement has been established by the model checker. In this case, IVCs provide usable information both formal and human-understandable that explains why the requirement is satisfied. Such information is valuable in analyzing safety-critical systems and can be used for many purposes in the software verification process, including at least the following:

Vacuity detection: The idea of syntactic vacuity detection (checking whether all subformulae within a property are necessary for its satisfaction) has been well studied [8]. However, even if a property is not syntactically vacuous, it may not require substantial portions of the model. This in turn may indicate that either a.) the model is incorrectly constructed or b.) the property is weaker than expected. We have seen several examples of this mis-specification in our verification work, especially when variables computed by the model are used as part of antecedents to implications of the a property specification.

Traceability: Certification standards for safety-critical systems (e.g., [11, 12]) usually require *traceability matrices* that map high-level requirements to lower-level requirements and (eventually) leaf-level requirements to code or models. Current traceability approaches involve either manual mappings between requirements and code/models [13] or a heuristic approach involving natural language processing [14]. Both of these approaches tend to be inaccurate. For functional properties that can be proven with inductive model checkers, inductive validity cores can provide accurate traceability matrices with no user effort.

Symbolic Simulation / Test Case Generation: Model checkers are now often used for symbolic simulation and structural-coverage-based test case generation [6, 15]. For either of these purposes, the model checker is supposed to produce a witness trace for a given coverage obligation using a "trap property" which is expected to be falsifiable. In systems of sufficient size, there is often "dead code" that cannot ever be reached. In this case, a proof of non-reachability is produced, and the IVC provides the reason why this code is unreachable.

Nevertheless, to be useful for these tasks, the generation process must be efficient and the generated IVC must be accurate and precise (that is, sound and close to minimal). The requirement for accuracy is obvious; otherwise the "minimal" set of model elements is no longer sufficient to produce a proof, so it no longer meets our IVC definition. Minimality is important because (for traceability) we do not want unnecessary model elements in the trace matrix, and (for completeness) it may give us a false level of confidence that we have enough requirements.

In addition, we are also interested in *diversity*: how many different IVCs can be computed for a given property and model? Requirements engineers often talk about "the traceability matrix" or "the satisfaction argument". If proofs are regularly diverse, then there are potentially many equally valid traceability matrices, and this may lead to changes in traceability research. It is often the case that there are multiple MIVCs for a given property. In this case, computing a single IVC provides, at best, an incomplete picture of the traceability information associated with the proof. Depending on the model and property to be analyzed, there is often substantial diversity between the IVCs used for proof, and there can also be a substantive difference in the size of a *minimal* IVC and a *minimum* IVC, which is the (not necessarily unique) smallest MIVC. If *all* MIVCs can be found, then several additional analyses can be performed:

Coverage Analysis: Closely related to vacuity detection is the idea of *completeness checking*, e.g., are all atoms in the model necessary for at least one of the properties proven about the model? Several different notions of completeness checking have been proposed [16, 17], but these are very expensive to compute, and in some cases, provide an overly strict answer (e.g., checking can only be performed on non-vacuous models for [17]). MIVCs can be used to define coverage metrics by examining the percentage of model elements required for a proof. However, since MIVCs are not unique, there are multiple, equally legitimate coverage scores possible. Having *all* MIVCs allows one to define additional metrics: coverage of MAY elements, coverage of MUST elements, as well as policies for the existing MIVC metric: e.g., choose the smallest MIVC.

Optimizing Logic Synthesis: synthesis tools can benefit from MIVCs in the process of transforming an abstract behavior into a design implementation. A practical way of calculating all MIVCs allows us to find a minimum set of design elements (optimal implementation) for a certain behavior. Such optimizations can be performed at different levels of synthesis.

Impact Analysis: Given all MIVCs, it is possible to determine which requirements may be falsified by changes to the model. This analysis allows for selective regression verification of tests and proofs: if there are alternate proof paths that do not require the modified portions of the model, then the requirement does not need to be re-verified.

Robustness Analysis: It is possible to partition the model elements into MUST and MAY sets based on whether they are in every MIVC or only some MIVCs, respectively. This may allow insight into the relative importance of different model elements for the property. For example, if the MUST set is empty, then the requirement has been implemented in multiple ways, such as would be expected in a fault-tolerant system.

The Requirements Engineering community is keenly interested in approaches to manage requirements traceability. In most cases, it is assumed that there is a single "golden" set of trace links that describes how requirements are implemented in software [18–20]. With computing a *single minimal IVC*, we are able to automatically establish one accurate traceability matrix. However, if there are *multiple* MIVCs, then it is possible that there are several equally valid sets of trace links. Examining the diversity of *all* MIVCs could lead to changes in how traceability is performed for critical systems.

1.1.1 Use in Research & Systems Development

Three of the important concerns in *certification* of critical systems are: conformance, traceability, and adequacy. Conformance involves determining whether a system meets its requirements: formal verification tools have excellent support for conformance. However, most formal verification tools do not provide support for traceability and adequacy. IVCs could be a mechanism by which formal verification tools address these concerns.

For example, airborne software must undergo a rigorous software development process to ensure its airworthiness. This process is governed by DO-178C: Software Considerations in Airborne Systems and Equipment Certification and when formal methods tools are used, DO-333: Formal Methods Supplement to DO-178C and DO-278A [11]. DO178C currently uses a variety of metrics to determine adequacy of requirements, but much of the effort involves code-level testing. Test suites are derived from requirements and used to test the software and measured using different structural coverage test metrics. If code-level test suites do not achieve full coverage, then an analysis is performed to determine whether there are missing requirements and test cases. The kind of structural coverage required (e.g., statement, branch, MCDC) for adequate testing is driven by the criticality of the software in question.

With the idea of IVCs, we propose a set of proof-based coverage metrics suitable for analyzing requirements competentness. Then, we will have the utility of the proposed approach evaluated by an industrial partner.

1.2 Contributions

Inductive validity cores have potential software engineering uses in several phases of the development cycle. However, efficient and effective generation strategies must be proposed to achieve these benefits. The contributions of the work are as follows:

- *Efficient techniques for extracting inductive validity cores from inductive proofs of safety properties over sequential models involving lemmas:* The thesis provides a formalization of techniques for computing inductive validity cores, and efficient algorithms for computing approximately minimal IVCs from proofs. *Efficient* in this context means that the computation time required is a small fraction of the time required to compute the original proof.

- *Efficient algorithms for computing all minimal IVCs from inductive proofs of safety properties over sequential models involving lemmas:* depending on the model and the property specification, the property of interest may be satisfied through different proof paths, which could results in multiple distinct IVCs. This thesis formalizes techniques for producing all inductive validity cores. It explores methods that are sound and reasonably efficient for computing all IVCs. It is not possible to guarantee completeness due to decidability issues, but we present algorithms that are complete for decideable problems and that will report possibly incomplete results to the user in situations in which a complete solution may not be possible.

- *A family of coverage metrics for formal verification based on* minimal *Inductive Validity Cores (MIVCs) that evaluate requirements adequacy:* we present a new approach to coverage analysis in formal verification which is much more efficient than previously proposed mutation-based analyses. Our goal is to provide a set of metrics that offer a range of levels of rigor that can be tailored to the criticality of the software. We discuss the relationship between proof-based metrics and mutation-based metrics, including a proof of equivalence between non-deterministic mutation coverage and one of our proposed proof-based metrics.

- *A new notion of proof-based auto-traceability based on IVCs:* requirements traceability is the primary application of IVCs. Currently, this task is performed manually without any formal analysis, which takes a lot of effort and yet is not accurate. With IVCs, we present the notion of complete traceability, by which requirement traceability can be performed automatically and accurately driven from the proofs of the properties.

- *A study of the relationship between inductive validity cores and bounded validity cores (BVCs).* IVCs are derived from inductive proofs. In some cases, proving safety properties over complex systems is often very expensive or infeasible. In these cases, engineers have to rely on bounded proofs. Bounded validity cores explain bounded proofs in the same way that inductive validity cores explain inductive proofs. By definition, such cores are smaller than inductive cores as they explain partial proofs. However, there are important relationships to be studied between bounded and inductive cores: how quickly do bounded cores converge to IVC sets? Can bounded cores be used as a notion of completeness or traceability?

- *Implementation of all the techniques:* the correctness of the techniques is proved/ discussed formally, while their efficacy is evaluated via substantial experiments. To this end, we have implemented all our methods in an open source model checker. The implementation and experimental results are publicly available. To this end, we have chosen an industrial model checker called `JKind` [21], which verifies safety properties of infinite-state synchronous systems. It accepts Lustre programs [22] as input. In JKind, verification is supported by multiple "proof engines" that execute in parallel, including k-induction, property directed reachability (PDR), and lemma generation engines that attempt to prove multiple properties in parallel. To implement the engines, JKind emits SMT problems using the theories of linear integer and real arithmetic. `JKind` supports the `Z3`, `Yices`, `MathSAT`, `SMTInterpol`, and `CVC4` SMT solvers as back-ends. We have extended `JKind` with new engines that implement our IVC generation algorithms.

- *An initial examination of how IVCs can be used to meet certification objectives:* critical software systems must undergo a rigorous software development process to ensure their correctness. This process is usually governed by an standard such as DO-178C [11]. We would like to examine the usefulness of the IVCs in providing satisfaction arguments that formally show how a system meets the certification objectives.

1.3 Chapters

This thesis is organized in 7 chapters. Chapter 2 mentions some formal notations and background and broadly discusses related work. Chapter 3 describes the notion of IVC, minimal IVC, all minimal IVCs, and BVCs while providing some algorithms for each of them. The correctness of the algorithms are formally established in this chapter. In Chapter 4, we describe the implementation of the proposed techniques and algorithms. In Chapter 5, we evaluate our techniques through a set of substantial experiments. In this chapter we introduced some uses of the IVCs. Chapter 6 shows how IVCs could be used in different areas. Finally, Chapter 7 concludes this thesis and outlines future research directions.

Chapter 2

Preliminaries and Related Work

Inductive validity cores aim to bridge the gap between verification techniques and the user insight into the results provided by the tools. The goal behind this idea is to have expressive verification results that help the engineers to evaluate the quality of a system or specification.

This chapter, first, provides some formal background on symbolic model checking, which is the underlying method for IVC generation. Broadly, IVCs can be compared with several existing methods such as invariant minimization, minimal unsatisfiable subformula, and slicing. We compare IVCs with these techniques in this chapter. Several major uses of IVCs are in requirements traceability, checking adequacy and vacuity. This chapter also discusses existing approaches in the literature used for these purposes.

2.1 Symbolic Model Checking

The idea of inductive validity cores is applicable to the context of symbolic model checking using inductive proof methods. After proving the correctness of a given property, we extract a minimal portion of the system (model) necessary for the proof of the property, which is what we call IVCs. Correctness can be expressed in terms of *safety* and *liveness* properties. Safety properties state that nothing bad ever happens, while liveness properties specifying that something good eventually happens.

IVCs determine why a *safety* property is satisfied by the system. Since this information is obtained from the inductive proofs, we call it *inductive* validity core. With minimal IVCs, we are able to abstract away the part of the system irrelevant to the proof of the property. This section mentions some background on symbolic model checking.

Given a state space U, a transition system (I, T) consists of an initial state predicate $I : U \to \mathit{bool}$ and a transition step predicate $T : U \times U \to \mathit{bool}$. We define the notion of reachability for (I, T) as the smallest predicate $R : U \to \mathit{bool}$ which satisfies the following formulas:

$$\forall u.\ I(u) \Rightarrow R(u)$$
$$\forall u, u'.\ R(u) \wedge T(u, u') \Rightarrow R(u')$$

A safety property $P : U \to \mathit{bool}$ is a state predicate. A safety property P holds on a transition system (I, T) if it holds on all reachable states, i.e., $\forall u.\ R(u) \Rightarrow P(u)$, written as $R \Rightarrow P$ for short. When this is the case, we write $(I, T) \vdash P$.

For an arbitrary transition system (I, T), computing reachability can be very expensive or even impossible. Thus, we need a more effective way of checking if a safety property P is satisfied by the system. The key idea is to over-approximate reachability. If we can find an over-approximation that implies the property, then the property must hold. Otherwise, the approximation needs to be refined.

A good first approximation for reachability is the property itself. That is, we can check if the following formulas hold:

$$\forall u.\ I(u) \Rightarrow P(u) \tag{2.1}$$
$$\forall u, u'.\ P(u) \wedge T(u, u') \Rightarrow P(u') \tag{2.2}$$

If both formulas hold then P is *inductive* and holds over the system. If (2.1) fails to hold, then P is violated by an initial state of the system. If (2.2) fails to hold, then P is too much of an over-approximation and needs to be refined.

One way to refine our over-approximation is to add additional lemmas to the property of interest. For example, given another property $L : U \to \mathit{bool}$ we can consider the extended property $P'(u) = P(u) \wedge L(u)$, written as $P' = P \wedge L$ for short. If P' holds on the system, then P must hold as well. The hope is that the addition of L

$$\begin{gathered}
I(u_0) \Rightarrow P(u_0) \\
\vdots \\
I(u_0) \wedge T(u_0, u_1) \wedge \cdots \wedge T(u_{k-2}, u_{k-1}) \Rightarrow P(u_{k-1}) \\
P(u_0) \wedge T(u_0, u_1) \wedge \cdots \wedge P(u_{k-1}) \wedge T(u_{k-1}, u_k) \Rightarrow P(u_k)
\end{gathered}$$

Figure 2.1: k-induction formulas: k base cases and one inductive step

makes formula (2.2) provable because the antecedent is more constrained. However, the consequent of (2.2) is also more constrained, so the lemma L may require additional lemmas of its own. Finding and proving these lemmas is the means by which property directed reachability (PDR) strengthens and proves a safety property [1].

Another way to refine our over-approximation is to use use *k-induction* which unrolls the property over k steps of the transition system. For example, 1-induction consists of formulas (2.1) and (2.2) above, whereas 2-induction consists of the following formulas:

$$\begin{gathered}
\forall u.\ I(u) \Rightarrow P(u) \\
\forall u, u'.\ I(u) \wedge T(u, u') \Rightarrow P(u') \\
\forall u, u', u''.\ P(u) \wedge T(u, u') \wedge P(u') \wedge T(u', u'') \Rightarrow P(u'')
\end{gathered}$$

That is, there are two base step checks and one inductive step check. In general, for an arbitrary k, k-induction consists of k base step checks and one inductive step check as shown in Figure 3.2 (the universal quantifiers on u_i have been elided for space). We say that a property is k-inductive if it satisfies the k-induction constraints for the given value of k. The hope is that the additional formulas in the antecedent of the inductive step make it provable. In practice, inductive model checkers often use a combination of the above techniques. Thus, a typical conclusion is of the form "P with lemmas $L_1, \ldots, L_n$ is k-inductive".

Unbounded model checking is performed through inductive proof methods such as k-induction [2] and IC3/PDR [1]. The PDR is currently the dominant unbounded model checking technique. In the past few years, several variations of this algorithm have been published [23–26]. The original idea in PDR is to compute a safe inductive invariant by strengthening the property using inductive couter-examples without unrolling the transition relation, while a classical implementation of k-induction tries to find an inductive invariant through k-step unrolling of a transition relation. Symbolic model checkers usually employ these proof methods, using an SMT/SAT solver in the backend. For example, `JKind` [21] is an SMT-based model checker for safety properties that uses parallel cooperating engines including k-induction, PDR, and template-based invariant generation.

Another form of symbolic evaluation is performed through bounded model checking. The goal of bounded model checking is to decide if a given program reaches an error within at most k unfolding of the transition relation. Although bounded model checkers do not provide a full proof of correctness, they are useful to discover bugs. For example, `CBMC` [27] checks array bounds (buffer overflows), absence of null de-references, and assertions. The `Alloy analyzer` [28] is another bounded model checker that checks temporal formulas specified using LTL. This tool also has a core extraction capability based on UNSAT cores. `JBMC` [29] is a Bounded Model Checker for Java programs that checks runtime exceptions and user-definded assertions. `LLBMC` is another bounded model checker for finding bugs in C/C++ programs, mainly intended for checking low-level system code. By exploiting the UNSAT core generation mechanism, we will be able to determine bounded validity cores using these tools[1].

As you can see, there are many efficient algorithms and tools for checking safety properties due to their prevalence in practical applications. However, system specifications still contains liveness requirements. Informally, safety properties demonstrate that the system preserves some invariant throughout its execution, while liveness properties demonstrate that eventually the system meets some goal. Formally, liveness properties can be distinguished from safety properties in that they require an infinite counterexample, an infinite path demonstrating the goal was not achieved.

When it comes to checking such requirements, there are techniques that exploit existing safety verification tools to check liveness. One approach is to reduce a liveness problem to a safety problem [30], where a suitable counterexample-detection logic is used by duplicating state elements, $U_{copy} = \{u_c | u \in U\}$. It non-deterministically samples

[1]See 2.3 and 3.2.

the design state and tries to find a valid counterexample scenario with finding a state repetition loop during which the behavior of the liveness and fairness conditions are observed.

Although converting liveness to safety makes it possible to use existing safety verification algorithms, such translations may be impractical because they substantially increase the problem size and complexity. Another technique is bounded liveness checking, k-liveness, [30], which proves the absence of a liveness failure up to a certain bound; i.e., for a property $GF\ P$, instead of proving $\neg P$ cannot happen infinitely often, it tries to prove $\neg P$ does not occur consecutively for k steps. k-liveness checks liveness as a sequence of safety checks increasing k incrementally. An improved version of k-liveness is to perform a sequence of safety queries as necessary to find a large-enough bound to avoid spurious failures by counting the maximum number of times that $\neg P$ can occur [31]. Adapting any of these methods for liveness checking makes it possible to tackle the problem with existing inductive algorithms for safety verification.

We do not discuss liveness further within this thesis, but note that either of these reduction techniques can be transparently used with the IVC techniques in this thesis to perform validity core reasoning for liveness properties.

2.1.1 Compositional Reasoning

Complex systems are usually composed from libraries of components. The specification of such systems are decomposed into properties of each individual component. Then, compositional verification is employed to ensure the correctness of the top level properties while integrating components [32]. Previously, Murugesan et al. demonstrated a model-based approach to system construction in which compositional proofs are used to to establish satisfaction arguments [3]. To cope with the complexity of modeling and scalability of verification of large systems, they proposed an approach in which systems can be decomposed into subsystems, modeled individually and verified compositionally. The decomposition of a system into subsystems induces the need to decompose the requirements of the system "flowed down" to each subsystem that is then modeled and verified.

Given an architectural model of the system (decomposition of system into components) in which each component (including the system) is endowed with its own set of requirements, they used a tool suite called AGREE [32] – a reasoning framework based on assume-guarantee reasoning – to compositionally verify whether system level requirements are established as a logical consequence of the component level requirements and the system level assumptions. Using AGREE they were able to verify large and complex systems efficiently. AGREE partitions the task of verification along the architectural lines of the system. Stating from the leaf level, it systematically verifies if the parent level requirements hold as a logical consequence of its immediately child component requirements in the given architecture.

To verify the requirements, `AGREE` uses the `JKind` [21] model checker. The underlying SMT solver in `JKind` automatically constructs proofs to establish satisfaction of requirements in the model. A proof can be visualized as a derivation tree where the leaves of the tree are axioms – elements of the model such as components requirements, interface connections, system assumptions – and each interior node represents the application of an inference rule that leads to proving the system requirement. If the solver encounters a violation of a requirement while constructing the proof, it reports it along with a counterexample - a concrete path of execution that explains the violation. On the other hand, when the proof is successfully constructed, the tool reports that the requirement is satisfied. There are other tools that perform similar verifications; for example `Kind 2` [33] also supports (assume-guarantee) contracts and `NuXmv` has a tool called `OCRA` [34] that supports the specification and analysis of component-based architectures.

An evidence in this context is nothing but an explanation of which parts of the model (the component requirements and system assumptions) the model checker used to prove the system level requirement. Since the solvers typically abstract away the proof it creates, with IVCs, we develop a technique to query the solver to excavate the axioms that were used as part of the proof. The IVC helps explain how the solver reported the satisfaction of the requirement, that is comparable to the counterexample explains the negative result.

2.1.2 Commercial Model Checkers

Several commercial tools produce *proof-cores* [35, 36], which we believe to be similar to IVCs/MIVCs, but are not presented at a level of formality to perform a precise comparison. However, to the best of our knowledge, none of these tools offer to calculate *all* proof-cores. Besides, the proof-core provided by these tools is usually used for internal analyses the tool performs such as coverage measurement. Therefore, the cores are not intended to be returned to the user in a clear way representing the actual design elements or a portion of the model. Moreover, these tools usually skip the minimization process, so their computed cores are not minimal.

In general, solutions provided by the commercial tools are quite underspecified: no formal description of the proof-core notion or algorithms are provided. In addition, no implementations or experimental results are provided, so we are not able to benchmark our techniques against those tools. However, our work can also be useful towards the support of this capability in future editions of these tools.

2.2 Slicing

Program Slicing is a well-known decomposition technique that maintains a set of program statements relevant to the computation of a selected function, called a slicing criterion. Generally speaking, given a slicing criterion, a slice is defined as any subset of the program which maintains the original effect of the program on the criterion [37], which is called an executable slice [38]. Slicing has many applications including optimizing program models for the purpose of verification using model checking [38–40].

Slicing is usually performed based on reachability analysis in program dependence graphs (PDGs). PDG nodes and edges show program states and dependence[2], respectively. PDGs are specifically useful in *static slicing*, where a slice is independent of the inputs, and maintains the program effects on the criterion correctly for all possible executions. Alternatively, *dynamic slicing* executes a path through the program, computing the statements which have an impact on the criterion for that specific execution [38]. Dynamic slicing is very useful in debugging, while static slicing is more attractive as an aid to verification.

For a given slicing criterion, static slices can be constructed from a backward or forward analysis. A backward slice of a program with respect to a program point p and set of program variables V include all the program statements that may affect the value of variables in V at p [41]. Consider the program in Figure 2.2 (a). A backward slice

[2]data dependence or control dependence

```
1. read (n)
2. i := 1
3. a := 0
4. b := 1
5. while i <= n do
6.    a := a + i
7.    b := b - i
8.    i := i + 1
9.  ptint (a)
10. print (b)
```

(a)

```
1. read (n)
2. i := 1

4. b := 1
5. while i <= n do

7.    b := b - i
8.    i := i + 1

10. print (b)

backward slice for
p = 10 and V = {b}
```

(b)

```


3. a := 0


6.    a := a + i


9.  ptint (a)


forward slice for
p = 3 and V = {a}
```

(c)

Figure 2.2: Example of backward and forward static slicing

for this code snippet over $V = \{b\}$ at the end of the program is shown in Figure 2.2 (b).

A forward slice of a program with respect to a program point p and set of program variables V consists of all program statements that may be affected by the value of variables in V at location p [41] (e.g. Figure 2.2 (c)).

In summary, in the backward approach, the statements of the program that does not have any effect on the criterion are sliced away. However, the forward approach slices away those statements not affected by the criterion. Our work can be viewed as a more accurate form of backwards static slicing starting from a requirement [42]. Slicing can determine the cone of influence (COI) for a given property, while IVCs are a subset of COI.

In fact, to start the verification process and IVC computation, we fisrt perform *backwards slicing* from the formula that defines the property of interest of the model. This step is to speed up the verification process. Then, IVCs are computed from the proof of the property over the sliced model. The slice produced is smaller and more accurate than a static slice of the formula [43], but guaranteed to be a sound slice for the formula for all program executions, unlike dynamic slicing [44]. Predicate-based slicing has been used [45] to try to minimize the size of a dynamic slice. Our approach may have utility for some concerns of program slicing (such as model understanding) by constructing simple "requirements" of a model and using the tool to find the relevant portions of the model.

2.3 UNSAT cores and Minimal UNSAT Subformulae

Satisfiability (SAT) Modulo Theories (SMT) solvers are powerful tools to decide the satisfiability of formulas with respect to background theories expressed in classical first order logic with equality. When using SAT solvers, instead of SMT solvers, the language of the solver is Boolean logic and the problem must be encoded into propositional logic.

A Boolean formula is satisfiable if there is a consistent assignment of values $true$ or $false$ to its variables in such a way that the formula evaluates to $true$. If the formula evaluates to $false$ for all possible variable assignments, then it is unsatisfiable. For example, the formula $a \wedge \neg a$ is unsatisfiable because there is no possible assignment for a to make the formula evaluate to $true$. On the other hand the formula $a \vee b$ is satisfiable when at least either a or b is assigned to $true$. An assignment that makes the formula *true* is called a model. Solvers return a satisfiable model in case the formula is satisfiable: $a = true$ is one possible model for our example.

For a given unsatisfiable problem instance, solvers try to generate a proof of unsatisfiability. It is usually more useful to have a minimum proof of unsatisfiability. Such a proof is dependent on identifying a sub-set of clauses that make the problem unsatisfiable. Solvers are usually capable of reporting such sub-sets in the proof, which is known as *UNSAT core*. However, the generated unsat core is not guaranteed to be minimal.

Every propositional logic formula can be transformed into an equivalent conjunctive normal form (CNF) using the laws of Boolean algebra. A formula is in CNF if it is a conjunction of clauses (or a single clause). Each clause is a disjunction of positive or negative literals (i.e., a variable or the negation of a variable). So each CNF formula can be formulated as a finite set of clauses (or constraints). Then we assume there is a function CheckSat(F) which determines if F is satisfiable or not. When F is unsatisfiable, we assume we have a function UnsatCore() which returns a minimal subset of the constraints such that the formula is satisfiable without them.

Algorithm 1: SAT domain single MUS extraction algorithm

```
input  : an unsatisfiable set of constraints $C$
output: MUS $M$
1 $M \leftarrow C$
2 for $c \in M$ do
3     if CheckSat($M \setminus c$) = UNSAT then
4         $M \leftarrow M \setminus \{c\}$
5 return $M$
```

Definition 1. *Minimal Unsatisfiable Subformulas(MUSes):* Let C be a finite set of constraints. $U \subseteq C$ be its unsatisfiable subset. A constraint $c \in U$ is an unsatisfiable core for U if $U \setminus \{c\}$ is satisfiable. A set of all unsatisfiable cores of U constitute an MUS for C. Note that such a set is not necessarily unique, and C can have several distinct MUSes.

Our work builds on top of a substantial foundation building MUSes from UNSAT cores [46]. Recent algorithms can handle very large problems, but computing MUSes is still a resource-intensive task. While some work is aimed at providing a set of minimal unsatisfiable formulae, minimality is usually defined such that given a set of clauses M, removing any member of M makes it satisfiable [47]. The step of producing minimal invariants for proofs has been investigated in depth by Ivrii et al. [48].

In recent years, a number of efficient algorithms for extracting minimal UNSAT subformulae (MUSes) have been proposed [49], most of which are focused on computing a single MUS [50–54]. A general algorithm for extracting a single MUS is shown in Algorithm 1.

As mentioned, an unsatisfiable problem can have several distinct MUSes. Although the problem of finding all MUSes is even harder than finding one MUS, there is some strong research in the literature focusing on this problem. For example, Recent work by Liffiton et al. [55] proposed an efficient algorithm to generate MUSes, called MARCO. Another work by Bendik et al. [56] tries to address this problem in the domains where minimization process is rather expensive. These algorithms can be used in our work in order to develop a new algorithm for computing all minimal IVCs. This will require changing the underlying mechanisms that are used to construct candidate solutions and also changing the structure of the proof of correctness. The technique used in MARCO can be adapted to our work for computing all minimal IVCs. Algorithm 2 is an abstract version of MARCO. This algorithm uses a symbolic representation of the power set of C, $\mathcal{P}(C)$, cleverly exploiting isomorphism between finite power sets and Boolean algebras. A brute-force approach to calculate all MUSes is basically explore all subsets of C and determine if they are MUS or not. In the power set exploration process, subsets whose satisfiability is not known yet are called *unexplored*; i.e., initially, all subsets in $\mathcal{P}(C)$ are unexplored, and at the end, satisfiability of every subset is known; i.e., all are *explored*. In MARCO, subsets of $C = \{c_1, c_2, \ldots, c_n\}$ are encoded using a set of Boolean variables (literals) $A = \{a_1, a_2, \ldots, a_n\}$ such that every constraint $c_i \in C$ is assigned to a Boolean variable $a_i \in A$. Let assume we have a function $Lit : C \rightarrow A$ that returns a corresponding literal of $c_i \in C$.

Then the algorithm maintains a Boolean formula from literals of A, called *map*, to represent the set of unexplored subsets of C. *map* is initially *true*. MARCO iteratively explores $\mathcal{P}(C)$ to find each MUS. In each iteration, an unexplored subset $C' \subseteq C$ is non-deterministically selected by finding a model of *map*. If C' is satisfiable, it is grown to a maximal satisfiable subset (MSS); Set $S \in C$ is MSS if $\forall c \in C \setminus \{S\}, (S \cup \{c\})$ is unsatisfiable. All subsets of a maximal satisfiable subset are also satisfiable. So, in this case, *map* is updated in a way to block those subsets from future computation by marking them as *explored* (line 5). On the other hand, if C' is `UNSAT`, it is shrunk to a MUS. When an MUS is found then all of its supersets are guaranteed to be unsatisfiable. So, *map* is updated in a way to mark all those supersets as *explored* in order to block them from future exploration (line 9). The grow/shrink procedure can be carried out by any algorithm for a single MSS/MUS extraction which makes MARCO applicable to arbitrary constraint satisfaction domain.[3]

UNSAT cores and MUSes are used for many different activities within formal verification. Gupta et al. [57] and McMillan and Amla [58] introduced the use of unsatisfiable cores in proof-based abstraction engines. Their goal is to shrink the abstraction size by omitting the parts of the design that are irrelevant to the proof of the property under verification. However this work is for finite systems in the domain of SAT solving, and

[3]In Algorithm 2 we abstracted these procedures. The shrink is basically the loop in Algorithm 1. A similar approach can be taken to perform the grow procedure: constraints are added one by one in a loop to check for unsatisfiability. We will explain these procedures further in Chapter 3.

Algorithm 2: MARCO algorithm for computing all MUSes

```
   input  : an unsatisfiable set of constraints $C$
   output: set of all MUSes $AM$
1  $AM \leftarrow map \leftarrow \top$
2  while CheckSat($map$) = SAT do
3      $C' \leftarrow$ find an unexplored subset of $C$ using a model of $map$ if
        CheckSat($C'$) = SAT then
4          $M \leftarrow$ grow ($C'$)
5          $map \leftarrow map \wedge (\bigvee_{c_i \notin M} Lit(c_i))$
6      else
7          $M \leftarrow$ shrink ($C'$)
8          $AM \leftarrow AM \cup M$
9          $map \leftarrow map \wedge (\bigvee_{c_i \in M} \neg Lit(c_i))$
10 return $AM$
```

the abstractions built are not intended to be returned to the user. We design our algorithms for IVC computation for infinite systems with the support of the state of the art of the SMT solvers. In addition, for IVC computation, the goal is to provide meaningful results to the user.

2.4 Proof/Lemma Minimization

The IVC idea shares many similarities with approaches for computing minimal invariant sets for inductive proofs (such as is performed for inductive proof certificates [48, 59]). A proof certificate is an artifact embodying a proof of the claim that can then be validated by a trusted checker. Given a safety property P, an formula Q is a k inductive strengthening of P if $P \rightarrow Q$, and Q is k inductive. Formula Q is a certificate for property P if Q is a k inductive strengthening of P. Certificates need to be concise and efficient to check by an independent tool or method. In particular, checking a certificate should not take more time than proving the original property. Mebsout and Tinelli presented a method for extracting and verifying proof certificates [59] implemented in `Kind 2` model checker. `Kind 2` performs verification by running different engines concurrently. Specifically, it employs number of auxiliary invariant generation engines to discover and pass along auxiliary invariants that might be helpful in proving a property of interest. Then these invariants are considered as safety certificates for the property. To simplify the certificates, they either reduce k or the size/complexity of the certificate formula. After obtaining a k inductive invariant Q, `Kind 2` starts with reducing k before simplifying the formula. It will replay the inductive step for Q for values $i < k$, following one of three different strategies:

- forward enumeration: all values of $1 \leq i \leq k' < k$ are checked, and k' is the first where k' inductiveness holds.
- binary search: the interval $[1, k]$ is divided into subintervals $[1, k']$ and $[k' + 1, k]$ of similar size. Then, the first or the second interval is recursively considered depending on whether Q is k' inductive or not.
- backward enumeration: all values of i from k down to 1 are checked, and it stops when k' inductiveness does not hold anymore.

To simplify the certificate, Q is converted into a set of subformulae. Iteratively, each subformula is removed from a set and it is checked if Q is still k inductive or not. In this case subformulae not needed to prove Q are pruned away. Finally, to verify a certificate, it needs to be proved that Q is a k inductive strengthening of the original property.

Our IVC algorithm also needs to find a minimal lemma set. However, there is a substantive difference: to find a minimal set of constraints, it is usually necessary to find new proofs involving *new lemmas* not used in the original proof, which accounts for the expense of finding an accurate minimal IVC. This process will be explained in Chapter 3.

2.5 Traceability

Requirements traceability can be defined as

> *"the ability to describe and follow the life of a requirement, in both forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)."* [60].

Traceability has been of great interest in research and practice for several decades. Intuitively, it concerns establishing relationships, called *trace links*, between the requirements and one or more artifacts of the system. Among the several different development artifacts and the relationships that can be established from/to the requirements, being able to establish trace links from requirements to artifacts that realize or *satisfy* those requirements—particularly to entities within those artifacts called *target artifacts* [61]—has been enormously useful in practice. For instance, it helps analyze the impact of changes in one artifact on the other, assess the quality of the system, aids in creating assurance arguments for the system, etc. In this work, we focus our attention on a subset of requirements traceability that we call *Requirements Satisfaction Traceability.*

Instead of just recording the trace links from each requirement to the target artifacts, *Satisfaction Arguments* [62] offer a semantically rich way to establish them. Originally proposed by Zave and Jackson [62], a satisfaction argument demonstrates how the behaviors of the system and its environment together satisfy the requirements. From a traceability perspective, these arguments help establish trace links (the *satisfied by* relationship) between the requirements and those parts of the system and environment (the target artifacts) that were necessary to satisfy the requirements; We call those target artifacts a *set of support* for that requirement. This set of support is the same as a minimal inductive validity core obtained from the correctness proof of the requirement.

2.6 Requirements Adequacy

Determining adequacy of properties has also been extensively studied. Certification standards such as DO-178C [11] require that requirements-derived tests achieve some level of structural coverage (MC/DC, decision, statement) depending on the criticality level of the software, in order to approximate adequacy. If coverage is not achieved, then additional requirements and tests are added until coverage is achieved. Recent work by Murugesan [63] and Schuller [64] attempted to combine test coverage metrics with requirements to determine completeness. Chockler [16] defined the first adequacy (completeness) metrics directly on formal properties based on mutation coverage. Later work by Kupferman et al. [65] defines completeness as an extension of vacuity to elements in the model. We present an alternative approach that uses the proof directly, which we expect to be considerably less expensive to compute.

2.6.1 Coverage and Mutations

Different notions of coverage have been defined in software testing. However, in formal verification, it is not immediately obvious how to define and compute coverage.

Coverage in verification was introduced in [66, 67]. Hoskote et al. [66] suggested a state-based metric in model checking based on FSM mutations, which are small atomic changes to the design. Then, the method for measuring coverage is to model check a given property for each mutant design. Later in [16], Chockler et al. provided corresponding notions of metrics used in simulation-based verification for formal verification. In fact, they improved the same idea of mutation-based coverage where each mutation is generated to check if a specific design element is necessary for the proof of the property. However, the proposed algorithm is both computationally expensive: each mutant model must be separately analyzed, which can easily lead to tens of thousands of verification problems on models of moderate size. Note that most of the mutation-based metrics, including [17, 68], are focused on finite state systems and hardware systems. In general, specification completeness can be defined with regard to the notion of coverage. In fact, the way that coverage is formalized plays a key part in the strength/ effectiveness of a method for the assessment of completeness.

The goal of a coverage metric is usually to assign a numeric score that describes how well properties cover the design. The majority of the work on coverage metrics has focused on *mutations*, which are "atomic" changes to the design, where the set of possible mutations depends on the notation that is used. A mutant is "killed" if one of the properties that is satisfied by the original design is violated by the mutated design [16, 17, 65, 68, 69]. There are many different kinds of mutations that have been proposed, primarily focused on checking sequential bit-level hardware designs. For these designs, *state-based* mutations flip the value of one of the bits in the state. There are several variations depending on whether the flip is performed on a single state within a Kripke structure [66], or in the description of the signal in the transition relation of the circuit [68]. *Logic-based* mutations fix the value of a bit to constant zero or one, and can be used to determine whether properties can find stuck-at faults. *Syntactic* mutations [16] remove states in a control flow graph representation of hardware. Similarly, for software, it is possible to apply any of the "standard" source code mutation operators used for software testing [70] towards requirements coverage analysis. Some examples of software mutations are [71]:

1. Replace an integer constant C by one of $\{0, 1, -1, C+1, C-1\}$,
2. Replace an arithmetic, relational, logical, bitwise logical, increment/decrement, or arithmetic-assignment operator by another operator from the same class,
3. Negate the decision in an `if` or `while` statement,
4. Delete a statement.

Mutation-based approaches are often impractically expensive to compute; even for small models, there are many possible mutations and we deal with too many verification problems. The number of single-mutation programs is roughly the product of the leaf elements of the program abstract syntax tree (AST) and the size of the chosen set of mutations, which can lead to an impractical number of verification problems.

Mutations for hardware are discussed in [17, 65, 69]. A more recent work in [69] performs coverage analysis through interpolation [72]. This work is also based on design-dependent mutations [16], where a design is considered as a net-list with nodes of types { AND, INV, REG, INPUT}. Each mutant design changes the type of a single node to INPUT. To decrease the cost of computation, coverage analysis is performed at several stages; first, all the nodes that do not appear in the resolution proof of a given property are marked as *not-covered*, and the rest of the nodes are marked as *unknown*. Then, for the unknown nodes, the basic mutation check is performed: if a corresponding mutant design violates the property, it will be considered as *covered*. Otherwise, the algorithm tries to drive an inductive invariant to prove that the node is not covered. Finally, an interpolant-based model checking is applied to the nodes that are still unknown.

Most of the mutation-based coverage techniques can be put into the category of falsity coverage, where we mutate the design and see if the property is still valid or not. In this way, we understand if that mutant was necessary (covered) for (by) that property. It is important to note that some mutations yield a subset of the state space of the system to be explored; in this case, any universal property that was true of the original system must, by definition, be true of the mutated system. This is where falsity coverage is not effective, and the notion of *vacuity coverage* comes into play. Falsity coverage asks weather the mutant FSM still satisfies the property, while vacuity coverage checks if the mutant FSM satisfies the property vacuously. In vacuity coverage, first, we makes sure the property is non-vacuous. Then, we mutate the design. If the property is vacuous afterwards, it means that the mutant was necessary for that property.

A similar notion to IVCs outlined in a patent [35], which sketches a family of *proof core*-based metrics for use in hardware verification. While the approach described by the patent is general, it is quite underspecified and it is not possible to compare their approach and ours. In addition, in commercial hardware verification frameworks do different forms of coverage analysis: Cadence JasperGold [36] does some form of proof core coverage and Synopsys VC Formal [73] does a mutation-based coverage approach. These coverage measurement approaches may be similar to the metrics we introduce but are not described in sufficient depth to be compared.

A different approach to measure coverage involves checking whether each output signal is fully constrained by the specification [74–76]. For example, in [75], authors propose a design-independent coverage analysis where missing properties are identified by unconstrained output signals. Given a property list and a specific computed signal s (usually drawn from the circuit outputs), if there is a trace with a point in time when s is not constrained to be a single value by the set of properties and the input trace, then the property set is incomplete. Alternately, given two traces that differ only in the value of signal s at a particular time step, if both traces satisfy property P, then s is not covered by P. The work in [77] refines this notion of coverage by providing a numeric score for each incompletely covered signal s. Such metrics are very rigorous but can lead to overspecification: the specification must completely define the input/output function of the implementation.

Another technique to measure requirements completeness is to employ several surrogate models; for example, Zowghi and Gervasi [78] use refinement to show *relative completeness* with respect to a *domain* model, which describes the behavior of the real world, irrespective of change induced by software. In their model, each iteration of refinement of requirements and domain models must be sufficient to prove the requirements of the previous iteration. However, this idea has two problems: first it provides no notion of absolute completeness, and second, it requires construction of a domain model, which is often difficult and/or expensive to construct.

Outside the context of formal verification, many authors have theorised and empirically validated conceptual model completeness, which are mostly dependent on a subjective judgement [78–83].

2.7 Vacuity Detection

Another potential use of our work is for "semantic" vacuity detection. A standard definition of vacuity is syntactic and defined as follows [65]: *A system K satisfies a formula ϕ vacuously iff $K \vdash \phi$ and there is some subformula ψ of ϕ such that ψ does not affect ϕ in K.* Vacuity has been extensively studied [65, 84–88] considering a range of different temporal logics and definitions of "affect". On the other hand, our work can be used to consider a broader definition of vacuity. Even if all subformulae are required (the property is not syntactically vacuous), it may not require substantial portions of the model, and so may be provable for vacuous reasons. The problem is exacerbated when the modeling and property language are the same (as in JKind), because whether a subformula is considered part of the model or part of the property, from the perspective of checking tools, can be unclear.

Torlak et al. in [10] finds MUSes of Alloy specifications, and considers semantic vacuity. Alloy models are only analyzed up to certain size bounds, however, and in general are unable to prove properties for arbitrary models. Also, because we are extracting information from proofs, it is possible to use IVCs for additional purposes (proof explanation and completeness checking).

2.8 Safety Standards

Due to the complexity of computer systems and our reliance on them, it is of the utmost importance that the development of these systems proceeds in a way that minimizes development errors. There are a several safety standards that focus on safety critical components, including DO-178C [11], MOD-0053 [89], and ISO 26262 [90]. Production of a functional safety case is usually a requirement for compliance with a specific standard, which brings opportunities and challenges to safety practitioners and researchers. In this section we briefly describes the objectives of these standards and how IVCs can be related to this area.

Software Considerations in airborne systems and equipment are usually regulated by certification: the DO-178C standard [11]. There are a couple of key components in DO-178C that are related to our purpose; first is to ensure the low-level requirements are in compliance with the high-level safety requirements. That is, each refinement must be shown not to introduce functionality not present in the artifact from which it was derived (adequacy). Another component of DO-178C is coverage analysis at the two levels: requirements-based analysis and structural analysis. After requirements-based testing, which ensures the software in the target computer will satisfy the high-level requirements, the purpose of coverage analysis in DO-178C is to determine how well this type of testing verified the implementation of the software requirements. Then, the structural coverage analysis is to determine which code structure was not exercised by the requirements-based test procedure. DO178C uses a variety of metrics to determine adequacy of requirements, but much of the effort involves code-level testing. Test suites are derived from requirements and used to test the software, then measured using different structural coverage test metrics. If code-level test suites do not achieve full coverage, then an analysis is performed to determine whether there are missing requirements and test cases. The kind of structural coverage required (e.g., statement, branch, MCDC) for adequate testing is driven by the criticality of the software in question. Traceability is another explicitly defined component of DO-178C; that is, low-level requirements must be traceable to the high-level requirements that they refine. Further, two other traceability objectives in DO-178C are (1) traceability of high-level requirements to system requirements and (2) traceability of software design to high-level requirements, which specifically require applicants to demonstrate bi-directional traceability.

MOD-0053 [89] is a defense standard that provides safety management requirements for defense systems, which are designed to be applied in different phases of the development process of MOD[4] projects. A key component of this standard is a *Safety Case*, which demonstrates how safety will be achieved and maintained. To summarize a Safety

[4]Ministry of Defence

Case and document safety management activities, Safety Case Reports should be provided. These reports describe a structured argument that the system is acceptably safe. Hazard analysis is another important component of MOD-0053, for which enough evidence must be provided to show all hazards are identified and properly managed. Therefore, traceability in this standard is a key component defined at two levels: (1) traceability between each safety requirement and the source of that requirement, and (2) traceability of the safety risk/hazards management to hazards and accidents. Requirements traceability in MOD-0053 should be established in both ways: (1) to trace each requirement to the part of the code which implements it, and (2) to trace from any part of the code, back through the software design/specification, to the requirement. Currently, traceability checks are performed via traceability matrices built manually, however, MOD-0053 recommends that traceability be provided between the formal arguments and the software requirement, which will help to check if all the requirements have been verified, and to ensure that the implications of changes to requirements can be assessed. In addition to traceability, this standard requires an assessment of the veracity and completeness of the software Safety Case. We believe that many of these traceability and adequacy checks can be automated using IVCs.

ISO 26262 [90] is a common standard used in the automotive industry whether an automotive system is acceptably safe. This standard provides guidance in different steps of the product development process to manage functional safety of a system at the hardware and software levels. One important component of ISO 26262 is Automotive Safety Integrity Levels (ASILs), by which each component is assigned to an acceptable risk level determined at the beginning of the development process. The goal is to analyze the system functionalities with respect to possible hazards. Each requirement is assigned a class of criticality from A to D, where D has the most safety critical processes and strictest testing regulations. In ISO 26262, qualification of software components demand testing not only under normal operating conditions, but also in the presence of faults so to determine how system reacts to abnormal inputs. ISO 26262 has other important components like *test tool qualification*, which are not closely related to the context of verification.

In order to meet the objectives of the safety standards, developers have to put a lot of manual effort into providing acceptable evidence such as assurance cases, traceability matrices, and requirements adequacy. We claim that some of these safety analyses can be automated with the IVC notion, and we would like to study how we can achieve that. Since IVCs are derived from the formal proofs, they will make much more accurate safety evidence than those created manually.

Chapter 3

Validity Cores

So far we have explained the idea of inductive validity cores and provided an informal explanation of their definition. This chapter provides a formal definition of IVCs and provides algorithms for computing them efficiently. Techniques for generating IVCs discussing the correctness of the techniques formally. We will provide detailed algorithms and illustrative examples for computing the proposed validity cores notions.

Let us start with a very simple system from the avionics domain to illustrate our approach. An Altitude Switch (ASW) is a hypothetical device that turns power on to another subsystem, the Device of Interest (DOI), when the aircraft descends below a threshold altitude, and turns the power off again after the aircraft ascends over the threshold plus some hysteresis factor. An implementation of an ASW containing two altimeters written in the Lustre language (simplified and adapted from Heimdahl *et al.* [91]) is shown in Figure 3.1. If the system is not inhibited, and either altimeter is below the constant `THRESHOLD`, then it turns on the DOI; else, if the system is inhibited or both altimeters are above the threshold plus the hysteresis factor `T_HYST`, then the DOI is turned off, and if neither condition holds, then in the initial computation it is false and thereafter retains its previous value. The notation `(false -> pre(doi_on))` in equation (7) describes an initialized register in Lustre: in the first step, the expression is `false`, and thereafter it is the previous value of `doi_on`. The input variable `inhibit` determines whether or not the system is inhibited.

A simple property `on_p` states that if both altimeters are under the threshold, then the DOI is turned on:

```
const THRESHOLD = 10000;
const T_HYST = THRESHOLD + 100;

node asw (alt1, alt2: int; inhibit: bool)
returns (doi_on: bool);
    var
        a1_below, a2_below, a1_above, a2_above,
        below, above_hyst, d1, d2: bool;
    let
(1)     a1_below = (alt1 < THRESHOLD);
(2)     a2_below = (alt2 < THRESHOLD);
(3)     a1_above = (alt1 >= T_HYST);
(4)     a2_above = (alt2 >= T_HYST);
(5)     below = a1_below or a2_below;
(6)     above_hyst = a1_above and a2_above;
(7)     doi_on = if (below and not inhibit)
                   then true else d1;
(8)     d1 = if (inhibit or above_hyst)
               then false else d2;
(9)     d2 = (false -> pre(doi_on));
    tel;
```

Figure 3.1: Altitude Switch Model

```
on_p = ((alt1 < THRESHOLD) and (alt2 < THRESHOLD))
             and not inhibit => doi_on = true;
```

This property can easily be proved over the model using a k-induction based verifier such as `JKind` [21]. If we perform a backwards static slice over the model starting from `on_p`, the entire model is returned. However, it is possible to prove the property with a minimal inductive validity core containing the equations assigning {`a1_below`, `one_below`, `doi_on`, `on_p`}. We can assign arbitrary values to variables outside the subset and the properties are still provable. Note that for this model there is a symmetric IVC: {`a2_below`, `one_below`, `doi_on`, `on_p`}.

Given a transition system that satisfies a safety property P, we want to know which parts of the system are necessary for satisfying the safety property. One possible way of asking this is, "What is the most general version of this transition system that still satisfies the property?" The answer is disappointing. The most general system is $I(u) = P(u)$ and $T(u, u') = P(u')$, i.e., you start in any state satisfying the property and can transition to any state that still satisfies the property. This answer gives no insight into the original system because it has no connection to the original system. In this section we formalize the notion of *inductive validity core* (IVC) which looks at generalizing the original transition system while preserving a safety property.

We assume the transition relation has the structure of a top-level conjunction. Given $T(u, u') = T_1(u, u') \wedge \cdots \wedge T_n(u, u')$ we will write $T = \bigwedge_{i=1..n} T_i$ for short. By further abuse of notation, T is identified with the set of its top-level conjuncts. Thus, $T_i \in T$ means that T_i is a top-level conjunct of T, and $S \subseteq T$ means all top-level conjuncts of S are top-level conjuncts of T. When a top-level conjunct T_i is removed from T, we write $T \setminus \{T_i\}$. Such a transition system can easily encode our example model, where each equation defines a conjunct within T that we will denote by the variable assigned; so, $T = \{$ `a1_below, a2_below, a1_above, a2_above, below, above_hyst, doi_on, d1, d2` $\}$.

Definition 2. *Inductive Validity Core (IVC):* Let (I, T) be a transition system and let P be a safety property with $(I, T) \vdash P$. We say $S \subseteq T$ for $(I, T) \vdash P$ is an Inductive Validity Core, denoted by $IVC(P, S)$, iff $(I, S) \vdash P$. When I, T, and P can be inferred from context we will simply say S is an inductive validity core.

Definition 3. *Minimal Inductive Validity Core (MIVC):* $S \subseteq T$ is a minimal Inductive Validity Core, denoted by $MIVC(P, S)$, iff $\quad IVC(P, S) \wedge \forall T_i \in S.\ (I, S \setminus \{T_i\}) \nvdash P$.

Note that, given $(I, T) \vdash P$, P always has at least one *MIVC*, and it may also have many distinct *MIVC*s corresponding to different proof paths. To capture the latter, the *all MIVCs (AIVC)* relation has been introduced in [92].

Definition 4. *All MIVCs (AIVC):* Given $(I, T) \vdash P$, $AIVC(P)$ is the set of all *MIVC*s for P:

$$AIVC(P) \equiv \{\ S \mid S \subseteq T \wedge MIVC(P, S)\}$$

Inductive validity cores have the following monotonicity property.

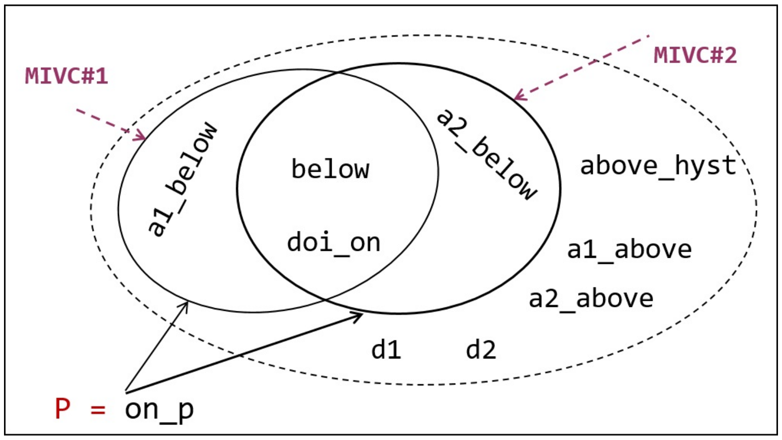


Figure 3.2: Graphical representation of *MIVC*s for the model in Figure 3.1 with $P =$ (on_p)

Lemma 1. Let (I, T) be a transition system and let P be a safety property with $(I, T) \vdash P$. Let $S_1 \subseteq S_2 \subseteq T$. If S_1 is an inductive validity core for $(I, T) \vdash P$ then S_2 is an inductive validity core for $(I, T) \vdash P$.

Proof. From $S_1 \subseteq S_2$ we have $S_2 \Rightarrow S_1$. Thus the reachable states of (I, S_2) are a subset of the reachable states of (I, S_1). □

Figure 3.2 illustrates these notions by a graphical representation of minimal IVCs for property $P =$ (on_p) in the ASW example. As shown in the picture, this property has two distinct *MIVC*s, which means the model satisfies P in two different ways: {{a1_below, below, doi_on}, {a2_below, below, doi_on}}, This is because in the implementation, the DOI is turned on when either of the altimeters is below the threshold, while our property states that they both must be below. Note that there is a subset of model elements, {a1_above, a2_above, above_hyst, d1, d2}, that does not show up in $AIVC(P)$. Elements in such a subset do not affect the satisfaction of P. For comparison, note that a backwards static slice starting from on_p will include the entire model.

Generally, an IVC computation technique aims to determine, for any subset $S \subseteq T$, whether P is provable by S. Then, a minimal subset that satisfies P is seen as a minimal proof explanation called a minimal Inductive Validity Core.

The notion of validity cores can also be adapted for *bounded* model checking to

Algorithm 3: `IVC_BF`: Brute-force algorithm for computing a minimal IVC

```
input  : $(I,T) \vdash P$
output: $MIVC$ for $(I,T) \vdash P$
1 $S \leftarrow T$
2 for $x \in S$ do
3     if $(I, S \setminus \{x\}) \vdash P$ then
4         $S \leftarrow S \setminus \{x\}$
5 return $S$
```

quantify how much of models have been explored by bounded analysis. We coin the term *Bounded Validity Core (BVC)* for this idea.

Definition 5. *Bounded Validity Core (BVC):* Let (I,T) be a transition system and let P be a safety property that is valid *up to* a given bound k, denoted by $(I,T) \vdash_k P$. We say $S \subseteq T$ for $(I,T) \vdash P$ is a Bounded Validity Core at depth k, denoted by $BVC_k(P,S)$, iff $(I,S) \vdash_k P$. When I, T, and P can be inferred from context we will simply say S is a bounded validity core at depth k.

3.1 Algorithms for computing an inductive validity core

Lemma 1 gives us a simple, brute-force algorithm for computing a minimal inductive validity core, Algorithm 3 (`IVC_BF`). The resulting set of this algorithm is obviously an inductive validity core for $(I,T) \vdash P$. The following lemma shows that it is also minimal.

Lemma 2. The result of Algorithm 3 is a minimal inductive validity core for $(I,T) \vdash P$.

Proof. Let the result be S'. Suppose towards contradiction that S' is not minimal. Then there is an inductive validity core M with $M \subset S'$. Take $x \in S' \setminus M$. Since $x \in S'$ it must be that during the algorithm $(I, S \setminus \{x\}) \vdash P$ is not true for some set S where $S' \subseteq S$. We have $M \subset S' \subseteq S$ and $x \notin M$, thus $M \subseteq S \setminus \{x\}$. Since M is an inductive validity core, Lemma 1 says that $S \setminus \{x\}$ is an inductive validity core, and so $(I, S \setminus \{x\}) \vdash P$. This is a contradiction, thus S' must be minimal. $\square$

Algorithm 4: `IVC_UC`: Efficient algorithm for computing a nearly minimal inductive validity core from UNSAT cores

```
    input  : $P$ with invariants $Q$ is $k$-inductive for $(I, T)$
    output: $IVC$ for $(I, T) \vdash P$
1   $k \leftarrow \textsc{MinimizeK}(T, P \wedge Q)$
2   $R \leftarrow \textsc{ReduceInvariants}_k(T, Q, P)$
3   return $\textsc{MinimizeIvc}_k(I, T, R)$
```

This algorithm has two problems. First, checking if a safety property holds is undecidable in general, thus, the algorithm may never terminate even when the safety property is provable over the original transition system. Second, this algorithm is very inefficient since it attempts to re-prove the property multiple times.

The key to a more efficient algorithm is to make better use of the information that comes out of model checking. In addition to knowing that P holds on a system (I, T), suppose we also know something stronger: P with the invariant set Q is k-inductive for (I, T). This gives us the broad structure of a proof for P which allows us to reconstruct the proof over a modified transition system. However, we must be careful since this proof structure may be more than is actually needed to establish P. In particular, Q may contain unneeded invariants which could cause the inductive validity core for $P \wedge Q$ to be larger than the inductive validity core for P. Thus before computing the inductive validity core we first try to reduce the set of invariants to be as small as possible. This operation is expensive when k is large so as a first step we minimize k. This is the motivation behind Algorithm `IVC_UC` (4).

To describe the details of Algorithm 4 we define queries for the base and inductive steps of k-induction (Figure 3.3). Note, in $\textsc{IndQuery}(T, Q, P)$ we separate the assumptions made on each step, Q, from the property we try to show on the last step, P. We use this separation when reducing the set of invariants.

We assume that our queries are checked by an SMT solver. That is, we assume we have a function $\textsc{CheckSat}(F)$ which determines if F, an existentially quantified formula, is satisfiable or not. In order to efficiently manipulate our queries, we assume the ability to create *activation literals* which are simply distinguished Boolean variables. The call $\textsc{CheckSat}(A, F)$ holds the activation literals in A true while checking F. When F is unsatisfiable, we assume we have a function $\textsc{UnsatCore}()$ which returns

$$\begin{aligned}
&\textsc{BaseQuery}_1(I,T,P) \equiv \forall s_0.\ I(s_0) \Rightarrow P(s_0)\\
&\textsc{BaseQuery}_{k+1}(I,T,P) \equiv \textsc{BaseQuery}_k(I,T,P) \wedge\\
&\quad (\forall s_0,\dots,s_k.\ I(s_0) \wedge T(s_0,s_1) \wedge \cdots \wedge T(s_{k-1},s_k) \Rightarrow P(s_k))\\
&\textsc{IndQuery}_k(T,Q,P) \equiv (\forall s_0,\dots,s_k.\\
&\quad Q(s_0) \wedge T(s_0,s_1) \wedge \cdots \wedge Q(s_{k-1}) \wedge T(s_{k-1},s_k) \Rightarrow P(s_k))\\
&\textsc{FullQuery}_k(I,T,P) \equiv\\
&\quad \textsc{BaseQuery}_k(I,T,P) \wedge \textsc{IndQuery}_k(T,P,P)
\end{aligned}$$

Figure 3.3: k-induction queries

Algorithm 5: MinimizeK(T,P)

```
1 $k' \leftarrow 1$
2 while CheckSat($\neg\textsc{IndQuery}_{k'}(T,P,P)$) = SAT do
3     $k' \leftarrow k' + 1$
4 return $k'$
```

a minimal subset of the activation literals such that the formula is unsatisfiable with those activation literals held true. In practice, SMT solvers often return a non-minimal set, but we can minimize the set via repeated calls to CheckSat. We assume both CheckSat and UnsatCore are always terminating.

The function MinimizeK(T,P) is defined in Algorithm 5. This function assumes that P is k-inductive for (I,T). It returns the smallest k' such that P is k'-inductive for (I,T). We start checking at $k'=1$ since smaller values of k' are much quicker to check than larger ones. The checking must eventually terminate since P is k-inductive. We also only check the inductive query since we know the base query will be true for all $k' \leq k$. Although we describe each query in Algorithm 5 separately, in practice they can be done incrementally to improve efficiency.

The function ReduceInvariants$_k(T,\{Q_1,\dots,Q_n\},P)$ is defined in Algorithm 6. This function assumes that $P \wedge Q_1 \wedge \cdots \wedge Q_n$ is k-inductive for (I,T). It returns a set $R \subseteq \{P,Q_1,\dots,Q_n\}$ such that R is k-inductive for (I,T) and $P \in R$. Like MinimizeK, this function only checks the inductive query since each element of R is an invariant and therefore will always pass the base query. A significant complication for reducing

Algorithm 6: $\textsc{ReduceInvariants}_k(T, \{Q_1, \ldots, Q_n\}, P)$

```
1  $R \leftarrow \{P\}$
2  Create activation literals $A = \{a_1, \ldots, a_n\}$
3  $C \leftarrow (a_1 \Rightarrow Q_1) \wedge \cdots \wedge (a_n \Rightarrow Q_n)$
4  **while** $true$ **do**
5      $\textsc{CheckSat}(A, \neg\textsc{IndQuery}_k(T, C, R))$
6      **if** $\textsc{UnsatCore}() = \emptyset$ **then**
7          **return** $R$
8      **for** $a_i \in \textsc{UnsatCore}()$ **do**
9          $R \leftarrow R \cup \{Q_i\}$
10         $C \leftarrow C \setminus \{a_i \Rightarrow Q_i\}$
```

invariants is that some invariants may mutually require each other, even though none of them are needed to prove P. Thus, in Algorithm 6 we find a minimal set of invariants needed to prove P, then we find a minimal set of invariants to prove those invariants, and so on. We terminate when no more invariants are needed to prove the properties in R. Algorithm 6 is guaranteed to terminate since R gets larger in every iteration of the outer loop and it is bounded above by $\{P, Q_1, \ldots, Q_n\}$. As with Algorithm 5, we describe each query in Algorithm 6 separately, though in practice large parts of the queries can be re-used to improve efficiency.

This iterative lemma determination does not guarantee a minimal result. For example, we may find P requires just Q_1, that Q_1 requires just Q_2, and that Q_2 does not require any other invariants. This gives the result $\{P, Q_1, Q_2\}$, but it may be that Q_2 alone is enough to prove P thus the original result is not minimal. Also note, we do not care about the result of CheckSat, only the UnsatCore that comes out of it. Since $P \wedge Q_1 \wedge \cdots \wedge Q_n$ is k-inductive, we know the CheckSat call will always return `UNSAT`.

The function $\textsc{MinimizeIvc}_k(I, \{T_1, \ldots, T_n\}, P)$ is defined in Algorithm 7. This function assumes that P is k-inductive for (I, T). It returns a minimal inductive validity core $R \subseteq \{T_1, \ldots, T_n\}$ such that P is k-inductive for (I, R). It is trivially terminating. Since Algorithms 5, 6, and 7 are terminating, Algorithm 4 is always terminating.

Our full inductive validity core algorithm in Algorithm 4 does not guarantee a minimal inductive validity core. One reason is that ReduceInvariants does not guarantee a minimal set of invariants. A larger reason is that we only consider the invariants that

Algorithm 7: $\textsc{MinimizeIvc}_k(I, \{T_1, \ldots, T_n\}, P)$

```
1 Create activation literals $A = \{a_1, \ldots, a_n\}$
2 $T \leftarrow (a_1 \Rightarrow T_1) \wedge \cdots \wedge (a_n \Rightarrow T_n)$
3 $\textsc{CheckSat}(A, \neg\textsc{FullQuery}_k(I, T, P))$
4 $R \leftarrow \emptyset$
5 for $a_i \in \textsc{UnsatCore}()$ do
6     $R \leftarrow R \cup \{T_i\}$
7 return $R$
```

the algorithm is given at the outset. It is possible that there are other invariants which could lead to a smaller inductive validity core, but we do not search for them. The following theorem shows that minimality checking is at least as hard as model checking and therefore undecidable in many settings.

Theorem 1. Determining if an IVC is minimal is as hard as model checking.

Proof. Consider an arbitrary model checking problem $(I, T) \vdash^? P$ where P is not a tautology. We will construct an IVC for a related model checking problem which will be minimal if and only if $(I, T) \nvdash P$. Let x and y be fresh variables. Construct a transition system with initial predicate $I \wedge \neg x$ and transition predicate $(x' \Rightarrow y') \wedge ((y' \Rightarrow P') \wedge T)$. The constructed system clearly satisfies the property $x \Rightarrow P$. Thus $S = \{x' \Rightarrow y', (y' \Rightarrow P') \wedge T\}$ is an IVC. S is minimal if and only if neither $\{x' \Rightarrow y'\}$ nor $\{(y' \Rightarrow P') \wedge T\}$ is an IVC. Since x and y are fresh and P is not a tautology, $\{x' \Rightarrow y'\}$ is not an IVC. Since x and y are fresh, $\{(y' \Rightarrow P') \wedge T\}$ is an IVC for the property $x \Rightarrow P$ if and only if $(I, T) \vdash P$. Therefore, S is minimal if and only if $(I, T) \nvdash P$. $\square$

When minimality is important, we can combine `IVC_BF` and `IVC_UC` into a single algorithm which aims to more efficiently determine minimality. The hybrid algorithm, `IVC_UCBF` (Algorithm 8), consists of running `IVC_UC` to generate an initial nearly minimal IVC which is then run through `IVC_BF` to attempt to achieve minimality. For sequential model checking problems with infinite theories, neither `IVC_BF` or `IVC_UCBF` is guaranteed to terminate, so our implementations simply time out after a set threshold. If no proof can be found within a certain time threshold after removing a model element, we declare that element necessary to the proof. In Chapter 5 we show that in practice our algorithm is nearly minimal and much more efficient than the brute-force approach.

Algorithm 8: An abstract representation of `IVC_UCBF`

```
   input  : $(I, T) \vdash P$
   output: Minimal IVC for $(I, T) \vdash P$
1  $S \leftarrow$ IVC_UC($(I, T) \vdash P$)
2  for $x \in S$ do
3      if $(I, S \setminus \{x\}) \vdash P$ then
4          $S \leftarrow S \setminus \{x\}$
5  return $S$
```

Algorithm 9: `BVC`: Algorithm for computing a bounded validity core

```
   input  : $P \vdash_{k'} (I, T)$ and bound $k \leq k'$
   output: $BVC$ for $(I, T) \vdash P$ at depth $k$
1  Create activation literals $A = \{a_1, \ldots, a_n\}$
2  $T \leftarrow (a_1 \Rightarrow T_1) \wedge \cdots \wedge (a_n \Rightarrow T_n)$
3  $\textsc{CheckSat}(A, \neg\textsc{BaseQuery}_k(I, T, P))$
4  $R \leftarrow \emptyset$
5  for $a_i \in \textsc{UnsatCore}()$ do
6      $R \leftarrow R \cup \{T_i\}$
7  return $R$
```

Note that as part of our IVC generation process we generate a k-inductive strengthening that is usually smaller and more efficiently computed than the work of Mebsout and Tinelli, as discussed in Section 2.4.

3.2 Algorithms for computing bounded validity cores

One straightforward technique for extracting BVCs is based on the conventional bounded model checking. We described k-induction in Chapter 2. Bounded model checking is the base-case in for a given k. To obtain validity cores from the base-case query, we need to unroll transition relation step by step (for $i = 0$ to $i \leq k$). Assuming the property of interest is valid, at least up to bound k, in each step the result of $\textsc{BaseQuery}_k(I, T, P)$ (described in Algorithm 3.3 using the formulas from Figure 9) will be `UNSAT`. In the same way with IVCs, we can map the unsat-cores to the model elements, which will result in BVCs.

3.3 Algorithm for computing all minimal inductive validity cores

The traceability information provided by MIVCs can be used to perform a variety of engineering analysis such as coverage analysis, robustness analysis, and vacuity detection. The more MIVCs are identified, the more precisely such analyses can be performed. A full enumeration of all MIVCs is an interesting problem that can be approached from two perspectives:

1. Offline approach: obtaining all minimal IVCs is guaranteed at the end of the computation. In this approach we can make use of the fast approximate IVC generation algorithm. However, intermediate results are not guaranteed to be minimal. This approach is useful in problems for which the algorithm can terminate in a reasonable time.

2. Online approach: identifying all MIVCs one by one, which allows the computation to stop at any time. In this approach, we need to come up with an efficient way to obtain as many MIVCs as possible. This approach is useful when the offline approach is very resource consuming and expensive and we need to obtain part of the existing MIVCs (if not all of them); e.g. for properties with a few hundreds of MIVCs, or large and complex models.

3.3.1 Offline Algorithm for all MIVCs

Now, we turn to the problem of finding *all* minimal IVCs. By definition, this approach allows us to find the *minimum* IVC in terms of number of elements (which may not be unique), and it allows us to explore the *diversity* of proofs for a particular property.

Considering the definition of a *MIVC*, a brute-force technique for enumerating all *MIVC*s would be the same as exploring the power set of T (denoted by $\mathcal{P}(T)$). Basically, the algorithm needs to explore the provability of a given property by any subset of T, which would be computationally expensive. Our approach is an adaptation of the the work of MARCO for generating all minimal unsatisfiable subsets (MUSes) in [55], and only needs to explore a (small) portion of $\mathcal{P}(T)$ in order to compute $AIVC$. In fact, it can be viewed as an instantiation of the MARCO proof schema for the richer theory of sequential model checking. We begin by introducing several additional notions and definitions, most of which are analogous or equivalent to those in [55].

Definition 6. *Maximal Inadequate Set (MIS):* $S \subset T$ for $(I, T) \vdash P$ is a Maximal Inadequate Set (*MIS*) iff $(I, S) \nvdash P$ and $\forall T_i \in T \setminus S.\ (I, S \cup \{T_i\}) \vdash P$.

Given $(I, T) \vdash P$, for every $S \in \mathcal{P}(T)$, we have either $(I, S) \vdash P$ or $(I, S) \nvdash P$. In the former case, we say S is **adequate** for P; in the latter, we say that S is **inadequate** for the proof of P. Note that every IVC is an adequate set for P, and every *MIS* is an inadequate set.

Corollary 1. For $(I, T) \vdash P$, if a given subset S is inadequate, then all of its subsets are inadequate as well:

$$\forall S_1 \subseteq S_2 \subseteq T.\ (I, S_2) \nvdash P \Rightarrow (I, S_1) \nvdash P$$

Proof. Immediate from Lemma 1. □

The basic idea behind an algorithm for computing $AIVC(P)$ is the same as exploration of $\mathcal{P}(T)$, with two major performance improvements. First, Lemma 1 and Corollary 1 are used to block large portions of $\mathcal{P}(T)$ from consideration. For example, if a set $S \in \mathcal{P}(T)$ is found to be inadequate, then all subsets of S are also inadequate and do not need to be explicitly considered. Second, if a set $S \in \mathcal{P}(T)$ is found to be adequate, then a fast algorithm (such as `IVC_UC` from [93]) is used to find a smaller $S' \subseteq S$ which is still adequate. This feeds into the first optimization since now all supersets of S' rather than S are blocked from future consideration.

To guide our algorithm, we now introduce a way of exploring $\mathcal{P}(T)$ which allows us to eliminate all subsets or supersets of any given set. We use a Boolean expression called *map*, which is in conjunctive normal form (CNF) and built gradually as the algorithm proceeds. Satisfying assignments for *map* correspond to elements of $\mathcal{P}(T)$. For each $S \in \mathcal{P}(T)$ that the algorithm determines to be adequate or inadequate, a corresponding clause is added to *map* which blocks S and all supersets or subsets, respectively, from consideration. When a clause is added to *map*, the corresponding $S \in \mathcal{P}(T)$ is called *explored.* The supersets or subsets of S which are blocked from consideration are called *excluded.* The remaining elements of $\mathcal{P}(T)$ are *unexplored.*

More precisely, given T with n top-level conjuncts, we define an ordered set of activation literals $\mathcal{A} = \{a_1, \ldots, a_n\}$, where each a_i has type Boolean. We assume the function $\textsc{ActLit} : T \rightarrow \mathcal{A}$ is a bijection assigning every $T_i \in T$ to an $a_i \in \mathcal{A}$ and vice versa. Then, a *map* for $AIVC(P)$ is a CNF formula built over the elements of $\mathcal{A}$ such that:

- Initially *map* is $\top$ since all of $\mathcal{P}(T)$ is unexplored.
- When *map* is satisfiable, a model of it is a set $M \in \mathcal{P}(\mathcal{A})$ consisting of those $a \in \mathcal{A}$ which are assigned *true*.
- Every model M of *map* corresponds to a set $S \in \mathcal{P}(T)$ such that $S = \bigcup_{a_i \in M} \textsc{ActLit}^{-1}(a_i)$ and $M = \bigcup_{T_i \in S} \textsc{ActLit}(T_i)$.
- For every explored set $S \in \mathcal{P}(T)$:
  - if S is adequate for P, then *map* contains a clause $\bigvee_{T_i \in S} \neg\textsc{ActLit}(T_i)$. This clause blocks all supersets of S from future consideration which is consistent with Lemma 1.
  - if S is inadequate for P, then *map* contains a clause $\bigvee_{T_i \in (T \setminus S)} \textsc{ActLit}(T_i)$. This clause blocks all subsets of S from future consideration which is consistent with Corollary 1.

Lemma 3. When *map* is satisfiable with model M, set $S = \bigcup_{a_i \in M} \textsc{ActLit}^{-1}(a_i)$ is not equal to any adequate or inadequate explored set, nor a subset (superset) of any inadequate (adequate) explored set in $\mathcal{P}(T)$.

Proof. Proof by contradiction. Case 1: Suppose there is an adequate set $Ex \subseteq S$ that has been already explored. Therefore, according to the definition, *map* contains a clause $C = \bigvee_{T_i \in Ex} \neg\textsc{ActLit}(T_i)$, and since $Ex \subseteq S$, it is impossible for the model $M = \bigcup_{T_i \in Ex} \textsc{ActLit}(T_i)$ to satisfy C; hence, the assumption is false.

Case 2: Suppose there is an inadequate set Ex such that $S \subseteq Ex$ and Ex has been already explored. Therefore, according to the definition, *map* contains a clause $C = \bigvee_{T_i \in (T \setminus S)} \textsc{ActLit}(T_i)$, and since $S \subseteq Ex$, it is impossible for the model $M = \bigcup_{T_i \in S} \textsc{ActLit}(T_i)$ to satisfy C; so, the assumption is false.

From Case 1 and Case 2, there is no model of *map* whose corresponding set in $\mathcal{P}(T)$ is a non-strict subset (superset) of any inadequate (adequate) explored set. $\square$

Lemma 4. For $(I, T) \vdash P$, *map* is satisfiable iff at least one $S \in AIVC(P)$ or one *MIS* of T is unexplored.

Proof. Let *map* be satisfiable with a model M, and let $S = \bigcup_{a_i \in M} \textsc{ActLit}^{-1}(a_i)$ be the corresponding set of $\mathcal{P}(T)$. If S is adequate, then it contains a *MIVC*. That *MIVC* must not be explored since otherwise S would have been blocked from consideration. The *MIVC* must not be excluded since it is not a strict superset of any adequate set (by minimality) nor a subset of any inadequate set (by Corollary 1). Thus the *MIVC* must be unexplored. The case where S is inadequate is symmetric.

In the other direction, let $S \subseteq T$ be an unexplored *MIVC*. Then consider the model $M = \bigcup_{T_i \in S} \textsc{ActLit}(T_i)$. We will show that each clause of *map* is satisfied by M. There are two types of clauses to consider. A clause $\bigvee_{T_i \in S'} \neg\textsc{ActLit}(T_i)$ is in *map* only if S' is adequate. M would falsify this clause only if $S' \subseteq S$ which is impossible by minimality of S. A clause $\bigvee_{T_i \in (T \setminus S')} \textsc{ActLit}(T_i)$ is in *map* only if S' is inadequate. M would falsify this clause only if $S \subseteq S'$ which is imposssible by Corollary 1. Thus M is a model for *map*. The case for an unexplored *MIS* is symmetric. $\square$

Corollary 2. For $(I, T) \vdash P$, *map* is unsatisfiable iff every $S \in \mathcal{P}(T)$ has been explored or excluded.

Proof. Immediate from the definition of *map* and Lemma 4. $\square$

Algorithm 10 shows the process of capturing all *MIVC*s, which are kept in set A, along with a warning flag, explained below. In line 2, we create the set of activation literals used by function $\textsc{ActLit}$. Line 3 initializes *map* with $\top$ over the set of literals we have. The main loop of state exploration starts at line 4 and continues until *map* becomes `UNSAT` which means all the *MIVC*s have been found. We assume we have a function $\textsc{CheckSat}$ that determines if an existentially quantified formula is satisfiable or not. As long as *map* is satisfiable, the algorithm computes a *maximal* `SAT` model for it (line 5). In this context, a maximal SAT model is a model with as many *true* assignment as possible without violating a clause; this problem is equivalent to the

MaxSAT problem, which has been well studied in the literature [94,95].[1] So, we assume there is a method by which we are able to have a maximal model of *map*. Line 6 extracts a set $M \in \mathcal{P}(\mathcal{A})$ of literals assigned to *true* in the model. Then, we need to obtain the corresponding set of S in $\mathcal{P}(T)$, which is done with function $\textsc{ActLit}^{-1}$ in line 7.

We also assume there is a function CheckAdq that checks whether or not P is provable by a given subset of T. Note that from Theorem 1, finding a minimal IVC is (in general) undecidable if the original checking problem is undecidable. Thus, for undecidable model checking problems, CheckAdq can return Unknown (after a user-defined timeout) as well as Adequate or Inadequate. For a given set S, if our implementation is unable to prove the property, we conservatively assume that the property is falsifiable and set a warning flag w to the user that the results may be approximate. if S is adequate, a *MIVC* is computed by GetIVC and added to set A (lines 10-11).[2] In this case *map* is constrained by a new clause in a way described before and shown in line 12. However, in the case that S is inadequate or unknown, *map* is constrained by the corresponding literals from $T \setminus S$ in line 14. Finally, if S is unknown, the warning flag w is set to true, as the results may be approximate (lines 15-16).

Theorem 2. Algorithm 10 will terminate.

Proof. We assume that CheckAdq has a finite timeout, so all operations within the loop require finite time. Each iteration of the while loop in Algorithm 10 blocks at least one element of $\mathcal{P}(T)$ which was not previously blocked. Since $\mathcal{P}(T)$ is finite, the algorithm terminates. □

Theorem 3. If no approximation warning is returned (w is False), Algorithm 10 enumerates all *MIS*es and *MIVC*s.

Proof. By Theorem 2 the algorithm terminates. This means *map* is eventually unsatisfiable. If w = False then all model checking problems are solved definitively

[1]MaxSAT is defined as the problem of satisfying as many (weighted) clauses as possible in a SAT instance. For N variables, similar to the MaxSAT problem, each clauses is weighted at $N + 1$ and extra unit-weight clauses are added forcing each variable to 1.

[2]Note that CheckAdq can be any method that verifies a safety property, such as K-induction, and the GetIVC function can be any function that returns an (approximately) minimal IVC, such as the `IVC_UC` or `IVC_UCBF` algorithms. The only requirement is that it follows the definition of an inductive validity core, that is: $S' \leftarrow \textsc{GetIVC}(P, S)$ implies that $S' \subseteq S$ and $(I, S') \vdash P$.

Algorithm 10: Algorithm `All_IVCs` for computing $AIVC$

```
input  : $(I, T) \vdash P$
output: $AIVC(P)$, Approximation warning flag $w$
1  $A \leftarrow \varnothing$; $w \leftarrow$ FALSE
2  Create activation literals $\{a_1, \ldots, a_n\}$
3  $map \leftarrow \top$
4  while CHECKSAT($map$) = SAT do
5      $model \leftarrow$ build a maximal model of $map$
6      $M \leftarrow$ extract the set of variables assigned $true$ in $model$
7      $S \leftarrow \bigcup_{a_i \in M} \textsc{ActLit}^{-1}(a_i)$
8      $res \leftarrow \textsc{CheckAdq}(P, S)$
9      if $res$ = ADEQUATE then
10         $S' \leftarrow \textsc{GetIVC}(P, S)$
11         $A \leftarrow A \cup \{S'\}$
12         $map \leftarrow map \wedge (\bigvee_{T_i \in S'} \neg \textsc{ActLit}(T_i))$
13     else
14         $map \leftarrow map \wedge (\bigvee_{T_i \in (T \setminus S)} \textsc{ActLit}(T_i))$
15         if $res$ = UNKNOWN then
16             $w \leftarrow$ TRUE
17 return $A, w$
```

(no UNKNOWN results), so by Lemma 4, all *MIS*es and *MIVC*s are either explored or excluded. However, by maximality and Lemma 1, an *MIS* can never be excluded. Similarily, by minimality and Corollary 1, a *MIVC* can never be excluded. Thus all *MIS*es and *MIVC*s are explored and are elements of A by the end of the algorithm.

□

Note that none of the proofs above require that GETIVC returns a minimal IVC. As shown in Chapter 5, it is computationally cheap to find an approximately minimal *IVC* using the algorithm `IVC_UC`; however, using the better, usually minimal *IVC* using the `IVC_UCBF` algorithm is computationally expensive. For efficiency reasons, it is much better to use the approximate `IVC_UC` algorithm to compute the set of all *MIVC*s. The `IVC_UCBF` algorithm attempts to repeatedly prove the property by brute-force removing elements (BF = "brute force"), so does much of the work of Algorithm 10 in a way that is not effective towards finding other IVCs. The overhead of the `IVC_UC` algorithm is on average 31% over the baseline proof, as opposed to 2276% for the `IVC_UCBF` algorithm. In addition, the average increase in size of IVCs returned by `IVC_UC` is approximately 8% of the `IVC_UCBF` algorithm.

On the other hand, if GetIVC does not return minimal adequate sets, at the end of the process, set A may contain both $MIVC$s and some supersets of $MIVC$s. To make sure that the algorithm only returns the minimal adequate sets ($MIVC$s), all we need is to remove any supersets of other sets in A. We can do this "on the fly" by changing line 11 to the following: $A \leftarrow A \cup \{S'\} \setminus \{S \mid S \in A \wedge S' \subset S\}$. Obviously, the closer to minimal the results of GetIVC are, the fewer iterations are required for Algorithm 1 to terminate. Each non-minimal adequate set returned by GetIVC will induce an additional iteration for Algorithm 1.

3.3.2 Online Algorithm for all MIVCs

This section describes collaborative work with Masaryk University (Bendik et al. [96]). We present an online approach for enumerating all MIVCs. With an offline technique, described in Section 3.3.1, minimality of the IVCs is guaranteed when the algorithm terminates. However, if the termination requires a lot of resources, we may prefer to calculate as many minimal IVCs as possible in a given time. An online enumeration technique can be useful for such cases. The online approach finds minimal IVCs iteratively where minimality is guaranteed at the end of each iteration.

As explained, inadequate sets are duals to inductive validity cores. Each $U \subseteq T$ is either inadequate set or an inductive validity core. In order to unify the notation, we use notation *inadequate* and *adequate*. Note that especially minimal inductive validity cores can be thus called minimal adequate sets.

The monotonicity allows to determine status of multiple subsets of T while using only a single check for adequacy. For example, if a set $U \subseteq T$ is determined to be adequate, than all of its supersets are adequate and do not need to be explicitly checked. Let $Sup(U)$ and $Sub(U)$ denote the set of all supersets and subsets of U, respectively.

Every algorithm for computing MIVCs has to determine status (i.e adequate or inadequate) of every subset of T. We introduce the notion of *explored* vs *unexplored* subsets in Section 3.3.2. Moreover, we distinguish *maximal* unexplored subsets:

- U_{max} is a *maximal unexplored subset* of T iff $U_{max} \subseteq T$, U_{max} is unexplored, and

Algorithm 11: A näive shrinking algorithm

```
input  : $(I, U) \vdash P$
output: MIVC for $(I, U) \vdash P$
1 for $T_i \in U$ do
2     if $(I, U \setminus \{T_i\}) \vdash P$ then $U \leftarrow U \setminus \{T_i\}$
3 return $U$
```

each of its proper supersets is explored.

A straightforward way to find a (so far unexplored) MIVC of T is to find an unexplored adequate subset $U \subseteq T$ and turn U into an MIVC by a process called *shrinking*. A shrinking procedure iteratively attempts to remove elements from the set that is being shrunk, checking each new set for adequacy and keeping only changes that leave the set adequate. A näive example is shown in Algorithm 11.

Section 3.3.2 proposed an algorithm for MIVC enumeration which is based on the MUS enumeration algorithm MARCO [97]. The algorithm iteratively chooses maximal unexplored subsets and tests them for adequacy. Each maximal subset that is found to be adequate is then shrunk into a MIVC. This algorithm enumerates MIVCs in an online manner with a relatively steady rate of the enumeration. However, an evaluation of the algorithm shown that it is rather slow since the shrinking procedure can be extremely time consuming as each check for adequacy is in fact a model checking problem.

In this section, we propose a novel algorithm for online MIVC enumeration using an improved shrinking procedure. Moreover, the algorithm uses a procedure *grow*, which is a dual of the shrinking procedure. The algorithm also maintains the set *Unexplored* of unexplored subsets.

We can effectively use the set *Unexplored* for speeding up the shrinking procedure. When testing the set $U \setminus \{T_i\}$ (see line 2 in Algorithm 11) we first check whether $U \setminus \{T_i\}$ is still unexplored. If $U \setminus \{T_i\}$ is already explored, then its status is already known and no test for adequacy is needed.

Shrink Procedure

In the following observation, we specify which explored subsets can be used to speed up the shrinking procedure.

Algorithm 12: Approximate grow

```
input : $(I,T) \vdash P$
input : inadequate $U \subset T$ for $(I,T) \vdash P$
input : set $Unexplored$ of unexplored subsets of $T$
output: approximately maximal inadequate set for $(I,T) \vdash P$
1 $M \leftarrow$ a maximal $M \in Unexplored$ such that $M \supseteq U$
2 while $(I,M) \vdash P$ do
3     $M_{IVC} \leftarrow$ IVC_UC($(I,M),P$)        // gets approximately minimal IVC
4     $T_i \leftarrow$ choose $T_i \in (M_{IVC} \setminus U)$
5     $M \leftarrow M \setminus \{T_i\}$
6 return $M$
```

Observation 1.. Let U_1, U_2 be subsets of T such that U_1 is explored, U_2 is unexplored, and $U_1 \subset U_2$. Then U_1 is inadequate for $(I,T) \vdash P$.
Symetrically, if U_1, U_2 are subsets of T such that U_2 is explored, U_1 is unexplored, and $U_1 \subset U_2$. Then U_2 is adequate for $(I,T) \vdash P$.

Proof. If U_1 is adequate, then all of its supersets are necessarily adequate. Thus, if U_1 is determined to be adequate, then not just U_1 but also all of its supersets becomes explored. Since U_1 is explored and U_2 is unexplored, then U_1 is necessarily an inadequate subset of T. □

In other words, during the shrinking procedure, we are guaranteed that whenever we find an explored set, this set is inadequate. Thus, as a further optimization in our algorithm we try to identify as many inadequate sets as possible before starting the shrinking procedure. The search for inadequate sets is done with the help of the grow procedure.

Grow Procedure

Recall that if a set is determined to be inadequate then all of its subsets are necessarily also inadequate. Therefore, the larger is the set that is determined to be inadequate, the more inadequate sets are explored. To identify inadequate sets as quickly as possible we search for maximal inadequate sets (MISes).

In order to find a MIS, we can find an inadequate set $U \subset T$ and use a process called *grow* which turns U to a MIS for $(I,T) \vdash P$. The grow procedure iteratively attempts

Algorithm 13: Solving algorithm

```
Function Solve($I$,$U$,$P$):
    $res \leftarrow$ CheckAdq($I, U, P$)
    if $res$ = UNKNOWN then
        $approximateWarning \leftarrow true$                // a global variable
    return ($res$ = ADEQUATE)
```

to add elements from $T \setminus U$ to U, checking each new set for adequacy and keeping only changes that leave the set inadequate. Same as in the case of shrink procedure, we can use the set *Explored* to avoid checking sets whose status is already known. However, such grow procedure might still perform too many checks for adequacy and thus be very inefficient.

Instead, we propose to use a different approach. Algorithm 12 shows a procedure that, given an inadequate set U for $(I, T) \vdash P$, finds an *approximately* maximal inadequate set. It first finds some maximal unexplored set M such that $M \supseteq U$ and checks it for adequacy. If M is inadequate, then it is necessarily a MIS (this is a straightforward consequence of Observation 1.) Otherwise, if M is adequate then it is iteratively reduced until an inadequate set is found.

In particular, whenever M is found to be adequate, the approximative algorithm `IVC_UC` is used to find an approximately minimal IVC M_{IVC} of M. M_{IVC} succinctly explains M's adequacy. In order to turn M into an inadequate set, it is reduced by one element from $M_{IVC} \setminus U$ and checked for adequacy. If M is still adequate then the approximate growing procedure continues with a next iteration. Otherwise, if M is inadequate, the procedure finishes.

Given an unexplored inadequate set U for $(I, T) \vdash P$ and a set *Unexplored* of unexplored subsets of T, Algorithm 12 returns an *unexplored* inadequate subset M of T.

Proof. Let us denote initial M as M_{init}. Since $M_{init} \supseteq U$ and M is recursively reduced only by elements that are not contained in U, then in every iteration holds that $U \subseteq M \subseteq M_{init}$. Since both U, M_{init} are unexplored, then M is necessarily also unexplored. □

Solve Procedure

Determining whether a particular subset of elements $U \subset T$ can prove a property of interest P is as hard as model checking (1). Thus, in the general case, determining whether a set of model elements is an MIVC may not be possible for model checking problems that are in general undecidable, such as those involving infinite theories. We assume there is a function `CheckAdq` that checks whether or not P is provable for some (I, U). `CheckAdq` can return UNKNOWN (after a user-defined timeout) as well as ADEQUATE or INADEQUATE. For a given set U, if our implementation is unable to prove the property, we conservatively assume that the property is falsifiable and set a global warning flag *approximateWarning* to the user that the results produced may be approximate.

Complete Algorithm

In this section, we describe, how to combine the shrink and grow methods to form an efficient online MIVC enumeration algorithm. We call the algorithm Grow-Shrink algorithm. Since knowledge of (approximately) maximal inadequate subsets can be exploited to speed up the shrinking procedure, it might be tempting to first find all MISes. However, this is in general intractable since there can be up to exponentially many MISes (w.r.t. the size of T). Instead, we propose to alternate both the shrinking and growing procedures. Note that during shrinking, we might determine some subsets to be inadequate. Such subsets can be subsequently used as *seeds* for growing. Dually, adequate subsets that are explored during growing can be later used as *seeds* for the shrinking procedure.

The pseudocode of our algorithm is shown in Algorithm 14. The computation of the algorithm starts with an initialisation procedure `Init` which creates a global variable *Unexplored* for maintaining the unexplored subsets and a global shrinking queue *shrinkingQueue* for storing seeds for the shrinking procedure. Then the main procedure `FindMIVCs` of our algorithm is called.

Procedure `FindMIVCs` works iteratively. In each iteration, the procedure picks a maximal unexplored subset U_{max} and checks it for adequacy. If U_{max} is inadequate, then U_{max} and all of its subsets are marked as explored. Otherwise, if U_{max} is adequate, then the algorithm `IVC_UC` [93] is used to reduce U_{max} into an approximately minimal IVC, and subsequently the procedure `Shrink` is used to shrink it into a MIVC.

Procedure `Shrink` works as described in Section 3. However, besides shrinking the given set into a MIVC, the procedure has also another purpose. Every inadequate set that is found during the shrinking is stored in a queue *growingQueue*. At the end of the procedure, all of these inadequate sets are grown into approximately maximal inadequate sets using the procedure `Grow`.

Procedure `Grow` turns a given inadequate set V into an approximately maximal inadequate set M as described in Section 3. The resultant set and all of its subsets are marked as explored. Moreover, every adequate set found during the growing is marked as explored and enqueued into *shrinkingQueue*. The queue *shrinkingQueue* is dequeued at the end of each iteration of the main procedure `FindMIVCs` and the sets that were stored in the queue are shrunk to MIVCs.

We need to ensure that each result of the shrinking procedure is a *fresh* MIVC, i.e., that each MIVC is produced only once. We shrink two kinds of inadequate sets in our algorithm: those that result from the inadequate maximal unexplored subsets, and those that are stored in *shrinkingQueue*. In the former case, we always shrunk an unexplored subset U_{IVC} which guarantees that the resultant MIVC U_{MIVC} is unexplored and thus fresh (if U_{MIVC} is already explored, then U_{IVC} would be necessarily also explored). However, in the latter case, all the sets stored in *shrinkingQueue* are already explored. To guarantee that shrinking of the sets from *shrinkingQueue* result only in fresh MIVCs, we maintain the following invariants of the queue:

I1) For each already produced MIVC M holds that there is no U in the queue such that $M \subseteq U$.

I2) There are no two U, V in the queue such that $U \subseteq V$.

To ensure that the invariants hold, we use the procedure `UpdateShrinkingQueue` which given an adequate set U removes from *shrinkingQueue* all supersets of U. We call the procedure every time a MIVC is found and every time a set is added to the queue.

Correctness: The algorithm produces only the MIVCs found by the shrinking procedure and all of them are *fresh*, i.e., produced only once. Only subsets whose status is known are removed from the set *Unexplored*, thus no MIVC is excluded from the computation. The algorithm terminates and all MIVCs are found since the size of *Unexplored* is reduced after every iteration.

Symbolic Representation of Unexplored Subsets

Since there are exponentially many subsets of T, it is intractable to represent the set *Unexplored* explicitly. Instead, we use a symbolic representation that is based on a well known isomorphism between finite power sets and Boolean algebras. We encode $T = \{T_1, T_2, \ldots, T_n\}$ by using a set of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$. Each valuation of X then corresponds to a subset of T. This allows us to represent the set of unexplored subsets *Unexplored* using a Boolean formula $f_{\mathit{Unexplored}}$ such that each model of $f_{\mathit{Unexplored}}$ corresponds to an element of *Unexplored*. The formula is maintained as follows:

- Initially, $f_{\mathit{Unexplored}} = \mathit{True}$ since all of $\mathcal{P}(T)$ are unexplored.
- To remove an adequate set $U \subseteq T$ and all its supersets from the set *Unexplored* we add to $f_{\mathit{Unexplored}}$ the clause $\bigvee_{i:T_i \in U} \neg x_i$.
- To remove an inadequate set $U \subseteq T$ and all its subsets from the set *Unexplored* we add to $f_{\mathit{Unexplored}}$ the clause $\bigvee_{i:T_i \notin U} x_i$.

In order to get an element of *Unexplored*, we ask a SAT solver for a model of $f_{\mathit{Unexplored}}$. In particular, to get a maximal unexplored subset, we ask a SAT solver for a *maximal model* of $f_{\mathit{Unexplored}}$. To get a maximal unexplored superset of $U \subseteq T$, we fix the truth assignment to the Boolean variables that correspond to elements in U to *True* and ask for a maximal model of $f_{\mathit{Unexplored}}$.

Let us illustrate the symbolic representation on $T = \{T_1, T_2, T_3\}$. If all subsets of T are unexplored then $f_{\mathit{Unexplored}} = \mathit{True}$. If $\{T_1, T_3\}$ is classified as an MIVC and $\{T_1, T_2\}$ as a inadequate set, then $f_{\mathit{Unexplored}}$ is updated to $\mathit{True} \wedge (\neg x_1 \vee \neg x_3) \wedge (x_3)$.

3.3.3 Illustration

Offline AIVC

To illustrate the `All_IVCs` algorithm we use the example presented in Chapter 3 with $P =$ (`on_p`) . For better description, we view T as an ordered set of its top-level conjuncts; i.e., $T = \{$ `a1_below, a2_below, a1_above, a2_above, below, above_hyst, doi_on, d1, d2` $\}$. The algorithm starts with creating activation literals for each $T_i \in T$. Let the ordered set of Boolean variables $\{a_1, \ldots, a_9\}$ be the corresponding literals to the elements of T (e.g. ActLit(`a1_below`) $= a_1$ and ActLit(`d2`) $= a_9$). Then, line 3 initializes *map* with $\top$.

In the first iteration of the `while` loop, since *map* is empty, it is satisfiable, and a model for it can be any subset of literals. So obviously, the first maximal model of *map* contains all the literals, which means, in line 6, $M = \{a_1, \ldots, a_9\}$, and in line 7, $S = T$. Since S is adequate for P, the GetIVC module is called in line 10. Suppose the returned *MIVC* by this function is $S' = \{$`a1_below`, `below`, `doi_on`$\}$; this set is added to A in line 11, and thus it comes to adding a new clause to *map* (line 12), which makes $map = (\neg a_1 \vee \neg a_5 \vee \neg a_7)$. As discussed, this constraint marks all the supersets of S' as blocked and prunes them off the search space.

For the second iteration, *map* is still satisfiable, so the algorithm gets to find a maximal model of it in line 5. Suppose this time, the maximal model makes $M = \{a_1, \ldots, a_4, a_6, \ldots, a_9\}$, then $S = T \setminus \{$`below`$\}$ will be another inadequate set that makes *map* become $map \leftarrow map \wedge a_5$ in line 14.

Suppose, in the third iteration, the maximal model leads to $M = \{a_2, \ldots, a_9\}$ and $S = T \setminus \{$`a1_below`$\}$ in lines 6 and 7. Since this S is adequate for P, GetIVC computes a new *MIVC* in line 10. Let the new *MIVC* be $S' = \{$`a2_below`, `below`, `doi_on`$\}$; after adding this set to A, it is time to constrain *map* by a new clause in line 11, which results in $map \leftarrow map \wedge (\neg a_2 \vee \neg a_5 \vee \neg a_7)$.

After these iterations, *map* is still satisfiable, and the maximal model is $S = T \setminus \{$`a1_below`, `a2_below`$\}$ in line 7. In this case, S is inadequate, so we update *map* as $map \leftarrow map \wedge (a_1 \vee a_2)$ (line 14). After adding this new clause to *map*, all the subsets of $T \setminus \{$`a1_below`, `a2_below`$\}$ will be blocked. The algorithm continues similar to the fourth iteration leading to S (in line 7) and *map* (in line 14) to be as $S = T \setminus \{$`doi_on`$\}$ and

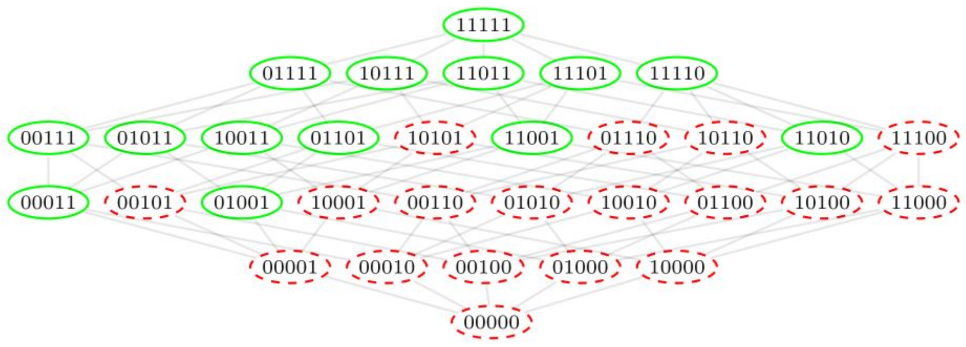


Figure 3.4: The power set from the example execution of our algorithm.

$map \leftarrow map \wedge a_7$.

Finally, after the fifth iteration, map becomes `UNSAT` and the algorithm terminates. Note that MISes and IVCs may be discovered in different orders from what explained here. The order by which sets are explored is quite dependent on the maximal model returned in line 5 as well as the $MIVC$returned in line 10 because there could be several distinct maximal models (MISes) and $MIVC$s. For this example with a $|T| = 9$ and $|\mathcal{P}(T)| = 2^9$, a brute force approach of power set exploration needs to look into 512 cases. However, the `All_IVCs` algorithm only explored 5 cases to cover the entire power set.

Online AIVC

The following example explains the execution of our algorithm on a simple instance where the transition step predicate T is given as a conjunction of five sub-predicates $\{T_1, T_2, T_3, T_4, T_5\}$. For the sake of simplicity, we do not exactly state what are the predicates and what is the safety property of interest. Instead, Figure 3.4 illustrates the power set of $\{T_1, T_2, T_3, T_4, T_5\}$ together with an information about adequacy of individual subsets. The subsets with solid green border are the adequate subsets, and the subsets with dashed red border are the inadequate ones. To save space, we encode subsets as bitvectors, for example the subset $\{T_1, T_2, T_4\}$ is written as 11010. There are three MIVCs in this example: 00011, 01001, and 11010.

We illustrate the first iteration of the main procedure `FindMIVCs` of our algorithm. Initially, all subsets are unexplored, i.e., $f_{Unexplored} = True$ and the queue *shrinkingQueue* is empty. The procedure starts by finding a maximal unexplored subset and checking it for adequacy. In our case, $U_{max} = 11111$ is the only maximal unexplored subset and it is determined to be adequate. Thus, the algorithm `IVC_UC` is used to compute an approximately minimal IVC $U_{IVC} = 01101$ which is then shrunk to a MIVC 01001.

During the shrinking, sets 00101, 01001, and 01000 are subsequently checked for adequacy and determined to be inadequate, adequate, and inadequate, respectively. The set 01001 is the resultant MIVC, thus the formula $f_{Unexplored}$ is updated to $f_{Unexplored} = True \wedge (\neg x_2 \vee \neg x_5)$. The other two sets, 00101 and 01000, are enqueued to the *growingQueue* and grown at the end of the procedure.

We first grow the set 00101. Initially, the procedure `Grow` picks $M = 10111$ as the maximal unexplored superset of 00101, and checks it for adequacy. It is adequate and thus, an approximately minimal IVC $M_{IVC} = 00011$ is computed, enqueued to *shrinkingQueue*, and formula $f_{Unexplored}$ is updated to $f_{Unexplored} = True \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5)$. Then, M is (based on M_{IVC}) reduced to $M = 10101$ and checked for adequacy. It is found to be inadequate, thus formula $f_{Unexplored}$ is updated to $f_{Unexplored} = True \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_2 \vee x_4)$, and the procedure terminates.

The growing of the set 01000 results into an approximately maximal inadequate subset 01110. Moreover, an approximately minimal IVC 11110 is found during the growing and enqueued into *shrinkingQueue*. The formula $f_{Unexplored}$ is updated to $f_{Unexplored} = True \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee x_5)$.

After the second grow, the procedure `Shrink` terminates and the main procedure `FindMIVCs` continues. The queue *shrinkingQueue* contains two sets: 00011, 11110, thus the procedure now shrinks them. During shrinking the set 00011, the algorithm would attempt to check the sets 00001 and 00010 for adequacy, however since both these are already explored, the set 00011 is identified to be a MIVC without performing any adequacy checks. The procedure `FindMIVCs` would now shrink also the set 11110, thus empty the queue *shrinkingQueue*, and continue with a next iteration.

Algorithm 14: The Grow-Shrink algorithm

```
1  Function Init($(I, T) \vdash P$):
2      $Unexplored \leftarrow \mathcal{P}(T)$                          // a global variable
3      $shrinkingQueue \leftarrow$ empty queue                         // a global variable
4      $approximateWarning \leftarrow$ false                           // a global variable
5      FindMIVCs()

1  Function FindMIVCs():
2      while $Unexplored \neq \emptyset$ do
3          $U_{max} \leftarrow$ a maximal set $\in Unexplored$
4          if Solve($I, U_{max}, P$) then
5              $U_{IVC} \leftarrow$ IVC_UC($(I, U_{max}), P$)
6              Shrink($U_{IVC}$)
7          else
8              $Unexplored \leftarrow Unexplored \setminus Sub(U_{max})$
9          while shrinkingQueue is not empty do
10             $U \leftarrow$ Dequeue($shrinkingQueue$)
11             Shrink($U$)

1  Function Shrink($U$):
2      $growingQueue \leftarrow$ empty queue
3      for $T_i \in U$ do
4          if $U \setminus \{T_i\} \in Unexplored$ then
5              if Solve($I, U \setminus \{T_i\}, P$) then $U \leftarrow U \setminus \{T_i\}$
6              else Enqueue($growingQueue, U \setminus \{T_i\}$)
7      output $U$                                                      // Output Minimal IVC
8      UpdateShrinkingQueue($U$)
9      $Unexplored \leftarrow Unexplored \setminus Sup(U)$
10     while growingQueue is not empty do
11         $V \leftarrow$ Dequeue($growingQueue$)
12         Grow($V$)

1  Function Grow($V$):
2      $M \leftarrow$ a maximal set $\in Unexplored$ such that $M \supseteq V$
3      while Solve($I, M, P$) do
4          $M_{IVC} \leftarrow$ IVC_UC($(I, M), P$)
5          UpdateShrinkingQueue($M_{IVC}$)
6          Enqueue($shrinkingQueue, M_{IVC}$)
7          $Unexplored \leftarrow Unexplored \setminus Sup(M_{IVC})$
8          $T_i \leftarrow$ choose $T_i \in (M_{IVC} \setminus V)$
9          $M \leftarrow M \setminus \{T_i\}$
10     $Unexplored \leftarrow Unexplored \setminus Sub(M)$

1  Function UpdateShrinkingQueue($U$):
2      for $V \in shrinkingQueue$ do
3          if $U \subseteq V$ then remove $V$ from $shrinkingQueue$
```

Chapter 4

Implementation in JKind

This chapter explains the implementation of IVC techniques. Part of this chapter is from a collaborative paper with Gacek et al. [98] and [21].

All of the inductive validity core algorithms presented in Chapter 3 have been implemented in `JKind` model checker [21]. `JKind` is an open-source industrial infinite-state inductive model checker for safety properties. Models and properties in `JKind` are specified in `Lustre` [99], a synchronous data-flow language, using the theories of linear real and integer arithmetic. `JKind` uses SMT-solvers to prove and falsify multiple properties in parallel.

`JKind` is one of a number of similar infinite-state inductive model checkers including Kind 2, `Kind`, NuXmv, and Zustre. These tools each offer multi-engine solvers that utilize both k-induction and some variant of IC3/PDR. In a recent comparison [33], `JKind` was the second most capable solver in terms of the number of problems solved (behind Kind 2) and had competitive performance across a large benchmark suite. The most noticeable bottleneck in `JKind` is the start-up time for the Java Virtual Machine (JVM). This cost is insignificant for larger models but causes decreased performance for benchmarks consisting of many very small models. There are also other tools such as `ESBMC-DepthK` [100], `VVT` [101] `CPAchecker`, [102], `CPROVER` [103] attempting to prove C programs using similar techniques (k-induction, invariant generation, and PDR).

An important characteristic of `JKind` is that is it designed to be integrated directly into user-facing applications. Written in Java, `JKind` runs on all major platforms and is easily compiled into other Java applications. `JKind` bundles the Java-based

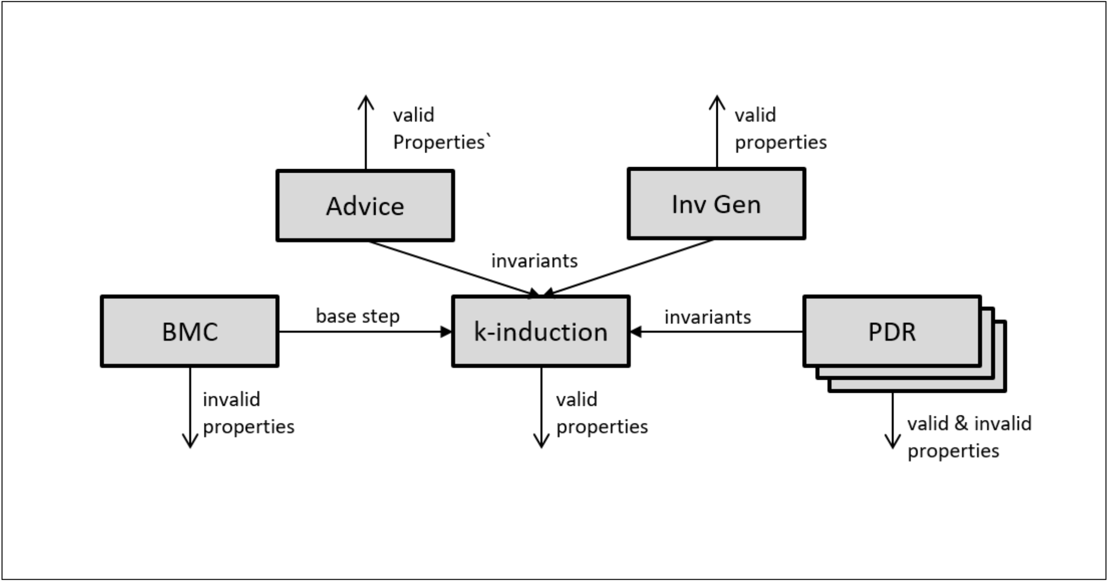


Figure 4.1: `JKind` engine architecture

`SMTInterpol` solver [104] and has no external dependencies. However, it can optionally call `Z3` [105], `Yices` [106], `MathSAT` [107], and `CVC4` [108] if they are available.

Another distinguishing characteristic of `JKind` is its focus on the usability of results. For a falsified property, `JKind` provides options for simplifying the counterexample in order to highlight the root cause of the failure. New integration of IVCs in `JKind`, allows it to provide traceability between the property and individual model elements for a proven property. These additional usability aspects have been found in industrial applications to be at least as important as the primary results [98].

In the rest of this chapter, we describe `JKind` architecture, functionality, and applications. Then, we provide implementation details about the integration of IVCs in this tool.

4.1 JKind Functionality and Main Features

`JKind` is structured as several parallel engines that coordinate to prove properties, mimicking the design of `Kind` and `Kind` 2 [33,109]. Some engines are directly responsible for proving properties, others aid that effort by generating invariants, and still others are reserved for post-processing of proof or counterexample results. Each engine can be enabled or disabled separately based on the user's needs. The architecture of `JKind` allows any engine to broadcast information to the other engines (for example, lemmas, frames, proofs/counterexamples) allowing straightforward integration of new functionality.

The solving engines in `JKind` are:

- **Bounded Model Checking (BMC).** The BMC engine performs a standard iterative unrolling of the transition relation to find counterexamples and to serve as the base case of k-induction. The BMC engine guarantees that any counterexample it finds is minimal in length.
- **k-induction.** The k-induction engine performs the inductive step of k-induction, possibly using invariants generated by other engines. **Invariant Generation.** The invariant generation engine uses a template-based invariant generation technique [110] using its own k-induction loop.
- **Property Directed Reachability (PDR).** The PDR engine performs property directed reachability [111] using the implicit abstraction technique [112]. Unlike BMC and k-induction, each property is handled separately by a different PDR sub-engine. Invariants generated as a side-product of PDR are shared with the k-induction process.

Invariant sharing between the solvers (shown in Figure 4.1) is an important part of the architecture. In our internal benchmarking, we have found that implicit abstraction PDR performs best when operating over a single property at a time and without use of lemmas generated by other approaches. On the other hand, the invariants generated by PDR and template lemma generation often allow k-induction, which operates on all properties in parallel, to substantially reduce the verification time required for models with large numbers of properties.

4.1.1 Post Processing and Re-verification

A significant part of the research and development effort for `JKind` has focused on post-processing results for presentation and repeated verification of models under development.

Smoothing

To aid in counterexample understanding and in creating structural coverage tests that can be more easily explained, `JKind` provides an optional post-processing step to minimize the number of changes to input variables, *smoothing* the counterexample. The smoothing engine uses a `MaxSat` query over the original BMC-style unrolling of the transition relation combined with weighted assertions that each input variable does not change on each step. The `MaxSat` query therefore tries (as best as possible) to hold all inputs constant while still falsifying the original property. This engine is only available with SMT-solvers that support `MaxSat` such as `Yices` and `Z3`.

Advice

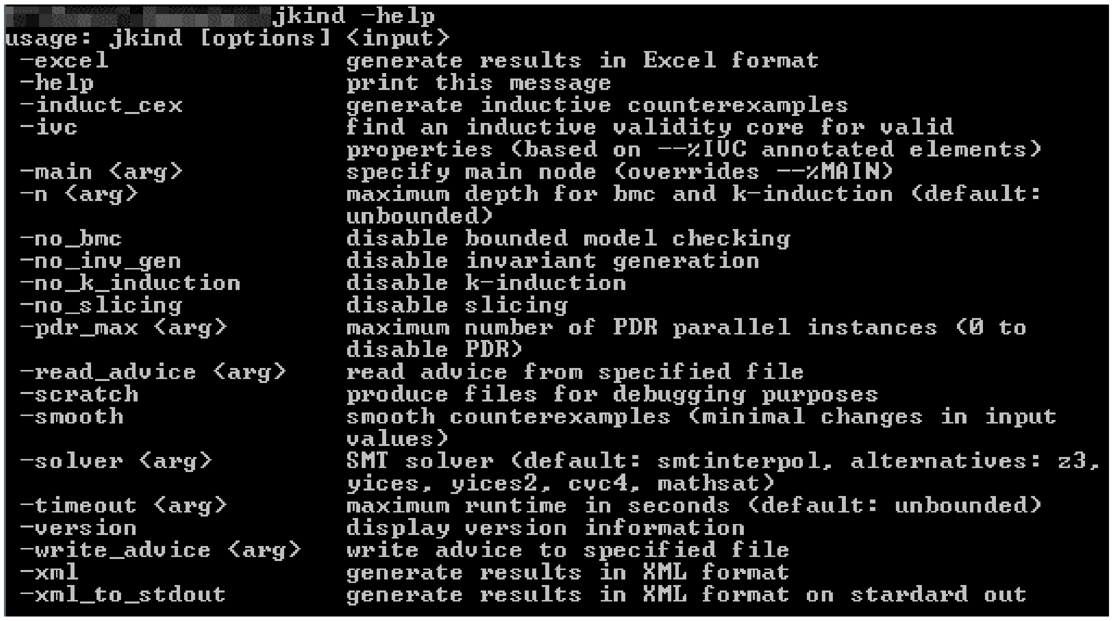
The advice engine saves and re-uses the invariants that were used by `JKind` to prove the properties of a model. Prior to analysis, `JKind` performs model slicing and flattening to generate a flat transition-relation model. Internally, invariants are stored as a set of proven formulas (in the `Lustre` syntax) over the variables in the flattened model. An *advice* file is simply the emitted set of these invariant formulas. When a model is loaded, the formulas are loaded into memory; formulas that are no longer syntactically or type correct are discarded, and the remaining set of formulas are submitted as an initial set of possible invariants to be proved via k-induction: if they are proved, they are passed along to other engines; if falsified, they are discarded. Names constructed between multiple runs of `JKind` are stable, so if a model is unchanged, it can be trivially re-proved using the invariants and k-induction with $k = 1$. In the case that a model is a small delta of a previously-proved model, it is often the case that most of the invariants can be re-proved, leading to reduced verification times.

4.2 Tool Installation

The prerequisite for installing and using JKind is to have Java installed and on the PATH variable for the OS. Installing JKind is accomplished by the following steps:

1. Download the zip containing the latest release from:

   https://github.com/agacek/jkind/releases



```
[redacted]jkind -help
usage: jkind [options] <input>
 -excel                 generate results in Excel format
 -help                  print this message
 -induct_cex            generate inductive counterexamples
 -ivc                   find an inductive validity core for valid
                        properties (based on --%IVC annotated elements)
 -main <arg>            specify main node (overrides --%MAIN)
 -n <arg>               maximum depth for bmc and k-induction (default:
                        unbounded)
 -no_bmc                disable bounded model checking
 -no_inv_gen            disable invariant generation
 -no_k_induction        disable k-induction
 -no_slicing            disable slicing
 -pdr_max <arg>         maximum number of PDR parallel instances (0 to
                        disable PDR)
 -read_advice <arg>     read advice from specified file
 -scratch               produce files for debugging purposes
 -smooth                smooth counterexamples (minimal changes in input
                        values)
 -solver <arg>          SMT solver (default: smtinterpol, alternatives: z3,
                        yices, yices2, cvc4, mathsat)
 -timeout <arg>         maximum runtime in seconds (default: unbounded)
 -version               display version information
 -write_advice <arg>    write advice to specified file
 -xml                   generate results in XML format
 -xml_to_stdout         generate results in XML format on stardard out
```

Figure 4.2: Successful installation of JKind

As of the writing of this document, that is version 4.0.[1]

2. Unzip the contents of that zip to the directory of your choice.

3. Add the folder from step 2 to the PATH variable for your OS. The installation can be tested by executing jkind help from the command line.

Successful installation will provide the following result in Figure 4.2.

To add a solver, users must perform these steps:

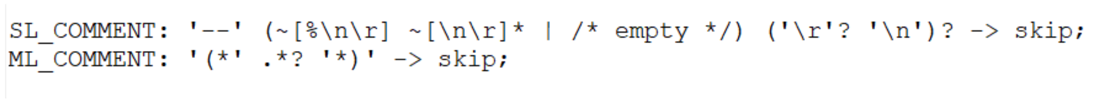
1. Download the solver of choice.

2. Unzip the contents of the solver into a directory.

3. Create an environment variable called <SOLVER>_HOME that points to the directory used the previous step. This is:

   - CVC4_HOME for CVC4
   - Z3_HOME for Z3

[1]All IVCs techniques are not yet part of the official release, but they can be obtained from [113]



```
SL_COMMENT: '--' (~[%\n\r] ~[\n\r]* | /* empty */) ('\r'? '\n')? -> skip;
ML_COMMENT: '(*' .*? '*)' -> skip;
```

Figure 4.3: `Jkind` comment grammar elements

- MATHSAT_HOME for `MathSAT`
- YICES_HOME for `Yices 1`
- YICES2_HOME for `Yices 2`

4. `JKind` will automatically look for the "bin" directory under <SOLVER>_HOME when the solver is activated.

4.3 JKind Lustre Grammar and Command Line Arguments

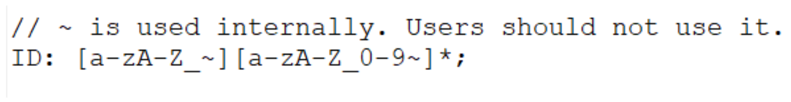
`JKind` accepts input from the synchronous dataflow language Lustre used for programming real-time systems. A dataflow language consists of a set of equations that assign variables where a variable can be computed as soon as its data dependencies have been computed. Such a language is a completely functional model without side effects, suitable for formal verification and program transformation. Like all functional language, a data flow language is naturally parallel where the only constraints on parallelism are due to the data-dependencies between variables.

Dataflow models can be either synchronous or asynchronous. In a synchronous dataflow model, the model is recomputed in a sequence of time instants. In an asynchronous model, the outputs of the system are continually recomputed depending on the inputs to the system.

4.3.1 Syntax Overview

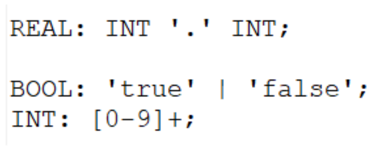
`JKind` Lustre grammar is built in `ANTLR` and can be referenced at *http://github.com/agacek/jkind/blob/master/jkind-common/src/jkind/lustre/parsing/Lustre.gz*.

`JKind` supports single line and multi-line comment styles. The grammar elements for comments are shown in Figure 4.3.



```
// ~ is used internally. Users should not use it.
ID: [a-zA-Z_~][a-zA-Z_0-9~]*;
```

Figure 4.4: `Jkind` identifier grammar



```
REAL: INT '.' INT;

BOOL: 'true' | 'false';
INT: [0-9]+;
```

Figure 4.5: `Jkind` literal grammar

The grammar for identifiers in `Jkind` are shown in Figure 4.4. Valid identifers will begin with a upper or lowercase letter, followed by 0 or more letters, numbers, or underscore characters.

The grammar for `Jkind` literals are shown in Figure 4.5.

A Lustre program is a collection of constant, type, node, and function declarations. The grammar elements for these declarations are shown in Figure 4.6.

`JKind` performs different analyses such as realizability, IVCs, and safety verification, etc. based on several LUSTRE annotations shown in Figure 4.7. The entry point of `JKind` analysis is a node. For the programs with more than one node, the node annotated with the −−%MAIN declaration is considered as the entry point, otherwise `JKind` will use the last node in the file as the main node. To do any meaningful analysis, a Lustre program must have at least one node defined. The remaining elements (constants, type definitions, and functions) can be used to specify a Lustre program for analysis. Properties are Lustre equations identified with −−%PROPERTY annotation. `JKind` runs the verification engines over equations listed in this annotation. We have added one more annotation for IVC analyses, −−%IVC, which will be described in the next section.

Figure 4.8 shows the grammar elements for the Lustre types that `JKind` supports. The grammar snippet in Figure 4.9 shows the Lustre expressions elements `JKind` supports. ID expressions are used to reference variables, enumeration values, and record field elements by name. Literal expressions are used to express boolean literals of true/-false, integer literals 0 to Infinity, and real literals 0.0 to Infinity. For more information

```
program: (typedef | constant | node)* EOF;

typedef: 'type' ID '=' topLevelType ';';

constant: 'const' ID (':' type)? '=' expr ';';

node:
  'node' ID '(' input=varDeclList? ')'
  'returns' '(' output=varDeclList? ')' ';'
  ('var' local=varDeclList ';')?
  'let'
    (equation | property | assertion | main | realizabilityInputs | ivc)*
  'tel' ';'?
;

varDeclList: varDeclGroup (';' varDeclGroup)*;

varDeclGroup: eID (',' eID)* ':' type;
```

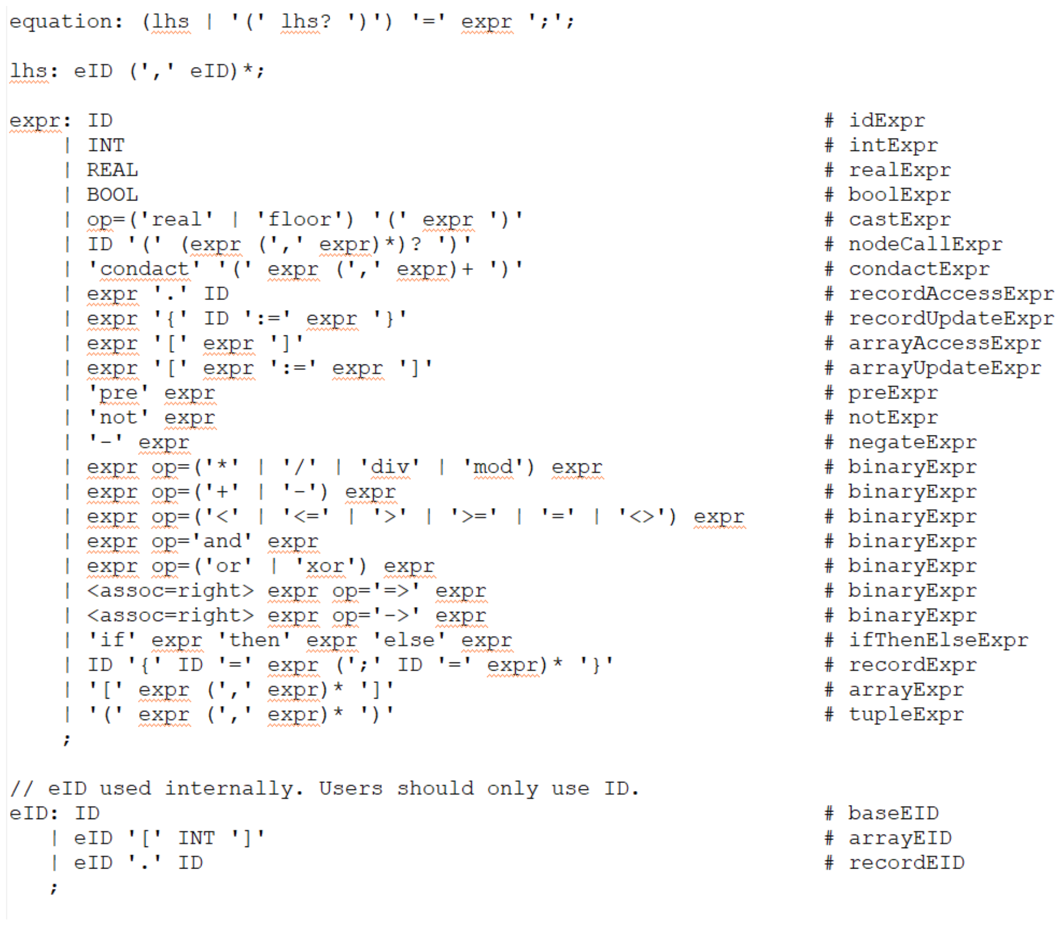
Figure 4.6: Lustre program declarations

```
property: '--%PROPERTY' eID ';';

realizabilityInputs: '--%REALIZABLE' (ID (',' ID)*)? ';';

ivc: '--%IVC' (eID (',' eID)*)? ';';

main: '--%MAIN' ';'?;

assertion: 'assert' expr ';';
```

Figure 4.7: Lustre annotations used by `JKind`

```
topLevelType: type                                        # plainType
    | 'struct' '{' (ID ':' type) (';' ID ':' type)* '}'   # recordType
    | 'enum' '{' ID (',' ID)* '}'                         # enumType
    ;

type: 'int'                                               # intType
    | 'subrange' '[' bound ',' bound ']' 'of' 'int'       # subrangeType
    | 'bool'                                              # boolType
    | 'real'                                              # realType
    | type '[' INT ']'                                    # arrayType
    | ID                                                  # userType
    ;
```

Figure 4.8: `JKind` Lustre type grammar



```
equation: (lhs | '(' lhs? ')') '=' expr ';';

lhs: eID (',' eID)*;

expr: ID                                                       # idExpr
    | INT                                                      # intExpr
    | REAL                                                     # realExpr
    | BOOL                                                     # boolExpr
    | op=('real' | 'floor') '(' expr ')'                       # castExpr
    | ID '(' (expr (',' expr)*)? ')'                           # nodeCallExpr
    | 'condact' '(' expr (',' expr)+ ')'                       # condactExpr
    | expr '.' ID                                              # recordAccessExpr
    | expr '{' ID ':=' expr '}'                                # recordUpdateExpr
    | expr '[' expr ']'                                        # arrayAccessExpr
    | expr '[' expr ':=' expr ']'                              # arrayUpdateExpr
    | 'pre' expr                                               # preExpr
    | 'not' expr                                               # notExpr
    | '-' expr                                                 # negateExpr
    | expr op=('*' | '/' | 'div' | 'mod') expr                 # binaryExpr
    | expr op=('+' | '-') expr                                 # binaryExpr
    | expr op=('<' | '<=' | '>' | '>=' | '=' | '<>') expr      # binaryExpr
    | expr op='and' expr                                       # binaryExpr
    | expr op=('or' | 'xor') expr                              # binaryExpr
    | <assoc=right> expr op='=>' expr                          # binaryExpr
    | <assoc=right> expr op='->' expr                          # binaryExpr
    | 'if' expr 'then' expr 'else' expr                        # ifThenElseExpr
    | ID '{' ID '=' expr (';' ID '=' expr)* '}'                # recordExpr
    | '[' expr (',' expr)* ']'                                 # arrayExpr
    | '(' expr (',' expr)* ')'                                 # tupleExpr
    ;

// eID used internally. Users should only use ID.
eID: ID                                                        # baseEID
   | eID '[' INT ']'                                           # arrayEID
   | eID '.' ID                                                # recordEID
   ;
```

Figure 4.9: JKind Expression and Equation grammar

on other expressions, see JKind User Guide [21].

4.3.2 Command Line

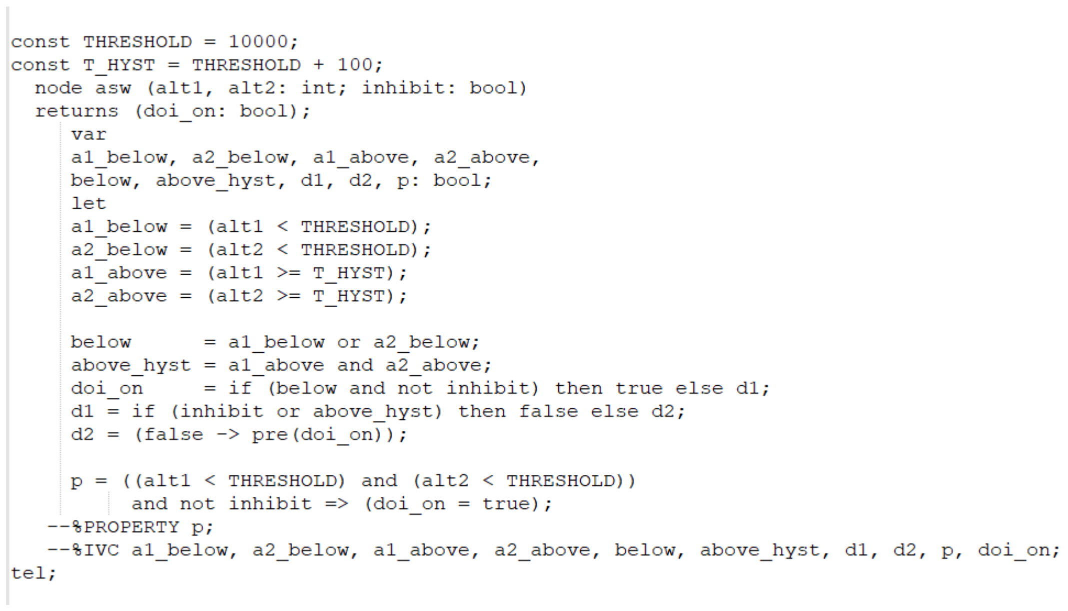
Different JKind functionalities explained so far are accessible from the tool command line. Table 4.1 summarizes major options in JKind.

4.4 Integration of IVCs into JKind

For a proven property, an inductive validity core is a subset of Lustre equations from the input model for which the property still holds. JKind performs IVC analyses based

Table 4.1: Major options in JKind command line

Command	Description
$-solver$ <arg>	SMT solver (default: smtinterpol, alternatives: z3, yices, yices2, cvc4, mathsat)
$-n$ <arg>	maximum depth for BMC and k-induction
$-timeout$ <arg>	maximum runtime in seconds (default: unbounded)
$-no\_bmc$	disable bounded model checking
$-no\_k\_induction$	disable k-induction
$-pdr\_max$ <arg>	maximum number of PDR parallel instances: if arg is 0, PDR will be disabled
$-no\_inv\_gen$	disable invariant generation
$-no\_slicing$	disable slicing
$-ivc$	find an inductive validity core for valid properties (based on $--\%IVC$ annotated elements in the LUSTRE input file)
$-all\_assigned$	mark all equations of the input file as $--\%IVC$
$-all\_ivcs$	find all inductive validity cores for valid properties
$-all\_ivcs\_algv$ <arg>	algorithm to be used for finding all IVCs (1: offline generation, 2: online generation)
$-bvc$	generate BVC at depth $-n$ within given $-timeout$
$-use\_unsat\_core$ <arg>	call UCBF algorithm on an input xml file containing the result provided by the use of $-ivc$ and $-xml$ options
$-xml$	generate results in XML format
$-scratch$	produce files for debugging purposes
$-write\_advice$ <arg>	write advice to specified file
$-read\_advice$ <arg>	read advice from specified file
$-smooth$	smooth counterexamples (minimal changes in input values)
$-interval$	generalize counterexamples using interval
$-help$	print all the command line options



```
const THRESHOLD = 10000;
const T_HYST = THRESHOLD + 100;
  node asw (alt1, alt2: int; inhibit: bool)
  returns (doi_on: bool);
     var
     a1_below, a2_below, a1_above, a2_above,
     below, above_hyst, d1, d2, p: bool;
     let
     a1_below = (alt1 < THRESHOLD);
     a2_below = (alt2 < THRESHOLD);
     a1_above = (alt1 >= T_HYST);
     a2_above = (alt2 >= T_HYST);

     below      = a1_below or a2_below;
     above_hyst = a1_above and a2_above;
     doi_on     = if (below and not inhibit) then true else d1;
     d1 = if (inhibit or above_hyst) then false else d2;
     d2 = (false -> pre(doi_on));

     p = ((alt1 < THRESHOLD) and (alt2 < THRESHOLD))
          and not inhibit => (doi_on = true);
   --%PROPERTY p;
   --%IVC a1_below, a2_below, a1_above, a2_above, below, above_hyst, d1, d2, p, doi_on;
tel;
```

Figure 4.10: An example of a LUSTRE program

on the annotation shown at line 41 of Figure 4.7. Using this annotation, we can mark program variables as potential IVC elements. By doing so, we are asking the tool to check which of these variables are necessary for the proof of each property. Such annotation affects the analyses, and a conservative approach is to bring all variables in this annotation, which can be done with $-all\_assigned$ `JKind` option. Figure 4.10 is the LUSTRE file for the example presented in Chapter 3 with the complete annotations ready to pass to `JKind` for IVC analyses. As you can see, all the program equations are annotated as potential IVC elements. With the current annotation, `JKind` finds 2 MIVCs {{`a1_below, below, doi_on`}, {`a2_below, below, doi_on`}}. However, if we change annotation, we may get different results. For example, if we remove `a1_below` and `doi_on` from the annotation, the All IVCs analysis will come up with only one MIVC containing {`below`}. If we only remove `a1_below` from the annotation, we will get only one MIVC of the form of {`below, doi_on`}.

To perform IVC analyses, we have extended `JKind` with a set of IVC generation engines. When a property is proved and IVC generation is enabled, our IVC engine executes one of the `IVC_UC`, `IVC_BF`, `IVC_UCBF` algorithms [93] to generate an (approximately) minimal IVC. This IVC generation engine is used to implement the GETIVC procedure in Algorithm 10. For generating all minimal IVCs, we have added another engine to `JKind` that implements our online and offline algorithms.

As previously discussed, one issue that needs to be handled in any implementation of the `IVC_UC`, `IVC_UCBF`, and `All_IVCs` algorithms involves the undecidability of the model checking problem; in each iteration of Algorithms 3 and 10 from Chapter 3, we attempt to prove the property with a subset S of the original model. Although we know that $(I, T) \vdash P$, from Theorem 1, the problem of determining whether or not any $S \subset T$ is adequate is undecidable. Therefore, we have to set timeouts for the model checking algorithm for each iteration of the `All_IVCs` procedure. In our implementation, we measure the time required to prove the property over the original model (*proof-time*), and the time required to calculate the first (approximate) IVC using `IVC_UC` (*`IVC_UC`-time*). The timeout we set for each iteration of the `IVC_UCBF` and `All_IVCs` algorithms is (30 sec + 5 $\times$ (*proof-time* + *`IVC_UC`-time*)).

If the `while` loop times out for S in line 8 of Algorithm 10, we treat S as an *inadequate* set to ensure that all results support a proof. In this case, line 13 will prune off S and all its subsets from the search space. Since the timeout is used by both the brute-force algorithm and the `All_IVCs` algorithm, minimality is only guaranteed if there are no timeouts. If a timeout occurs during computation, we report a warning to the user that our results are not guaranteed to be minimal. It is important to note that by increasing the timeout, it is possible that in some cases smaller IVCs can be generated, but the general problem will remain due to the undecidability of the model checking problem.

To implement `BVC` algorithm, a new tool is buit on top of `JKind`. In the implementation, basically, the `JKind` BMC engine has been hacked to emit the bounded cores after each step of transition relation unrolling. The user needs to provide a desired bound and timeout for this calculation.

The additional IVC engines integrated in `JKind` are shown in Figure 4.11. Once IVC generation is activated, through $-ivc$ options, an approximately minimal IVC is generated. If `JKind` has been called with $-all\_ivcs$, the IVC_UC engine passes the first approximate MIVC to the All_IVCs engine, where one of the all IVC generation

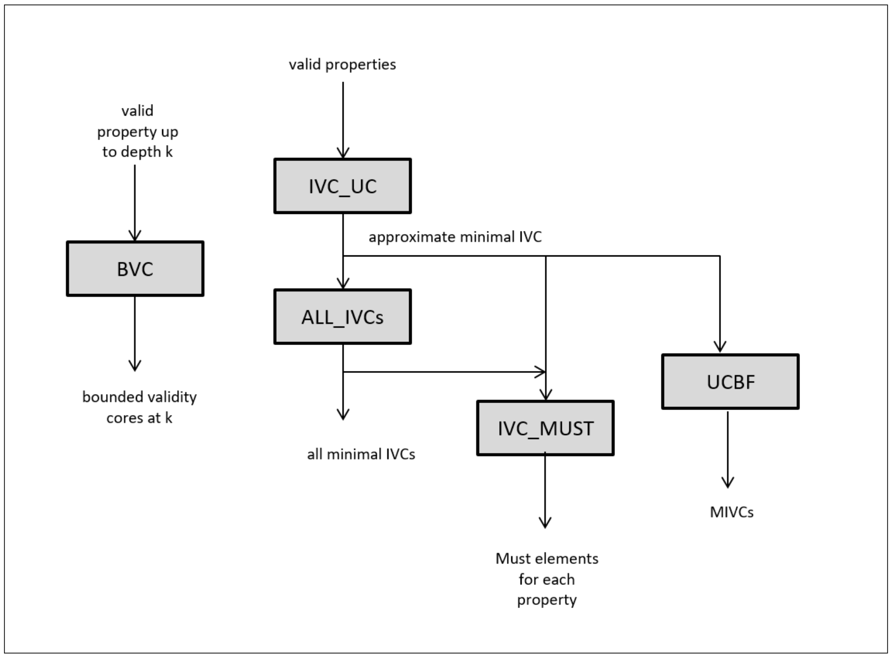
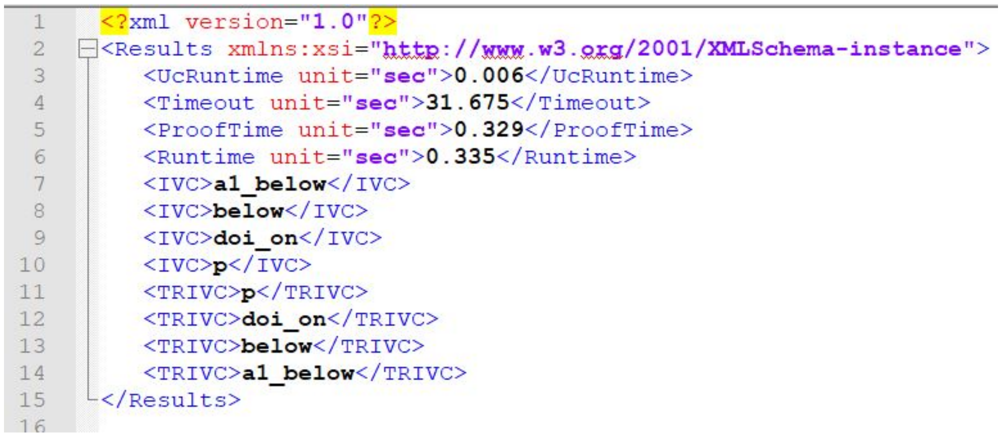


Figure 4.11: `JKind` IVC engines

algorithms will run depending on the provided configuration.
There are three stand-alone tools implemented in `JKind`:

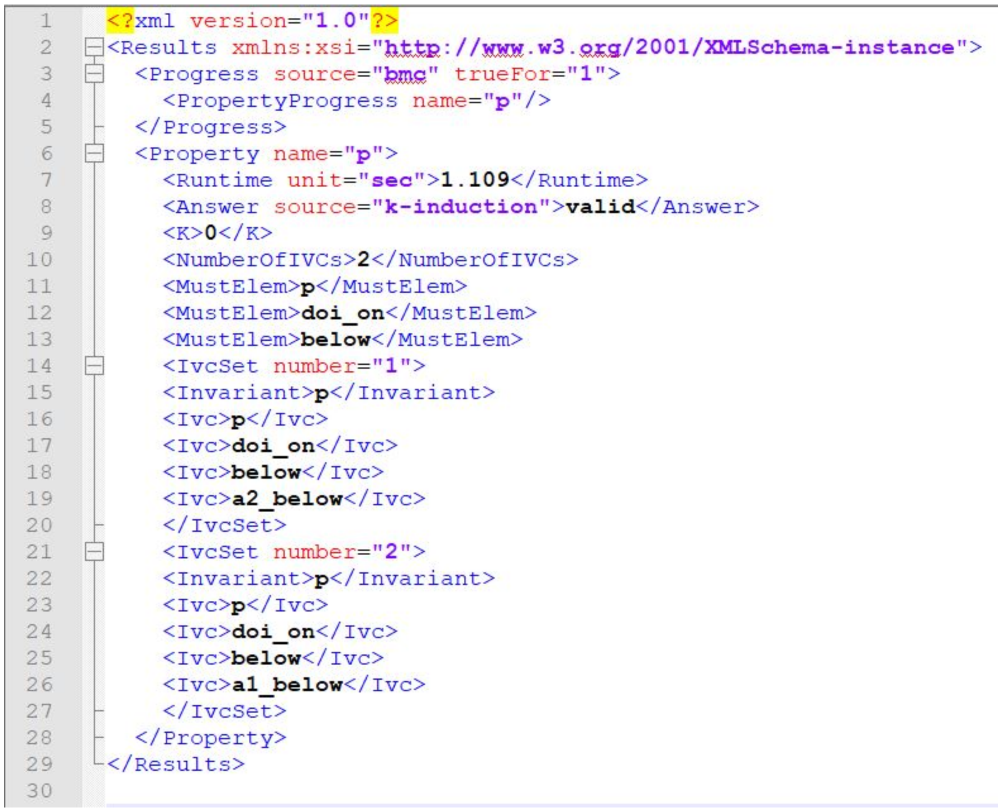
- BVC: this engine starts bounded model checking for a given bound within a specified timout. At the end of the process, it returns bounded validity cores.
- IVC_MUST: $MUST$ computation comes for free after $-all\_ivcs$ engine finds all the minimal MIVCs. The by-product of having all MIVCs are the $MUST$ elements, the intersection of all MIVCs. However, $MUST$ elements can be computed from one single (approximate) minimal IVC, as explained in Chapter 6. The IVC_MUST engine can be called individually and as input it takes the output of IVC_UC engine.
- UCBF: this is another stand-alone engine that takes the output of IVC_UC engine and minimize the approximate MIVC to guarantee minimality.

To wrap up this section, we attach the outputs of the IVC engines for our running example (Figure 4.12- 4.16).



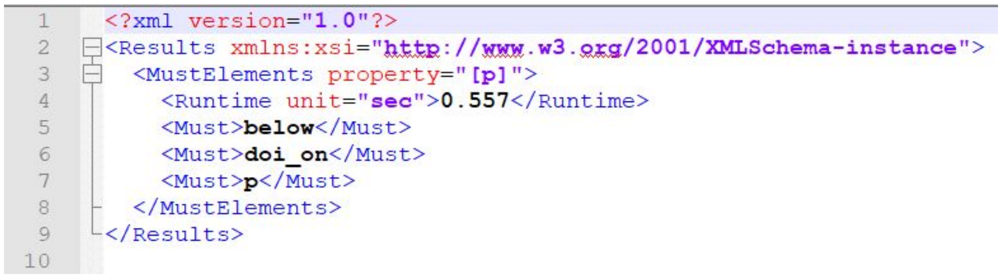
```xml
<?xml version="1.0"?>
<Results xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <UcRuntime unit="sec">0.006</UcRuntime>
    <Timeout unit="sec">31.675</Timeout>
    <ProofTime unit="sec">0.329</ProofTime>
    <Runtime unit="sec">0.335</Runtime>
    <IVC>a1_below</IVC>
    <IVC>below</IVC>
    <IVC>doi_on</IVC>
    <IVC>p</IVC>
    <TRIVC>p</TRIVC>
    <TRIVC>doi_on</TRIVC>
    <TRIVC>below</TRIVC>
    <TRIVC>a1_below</TRIVC>
</Results>
```

Figure 4.12: Sample output of IVC_UC engine



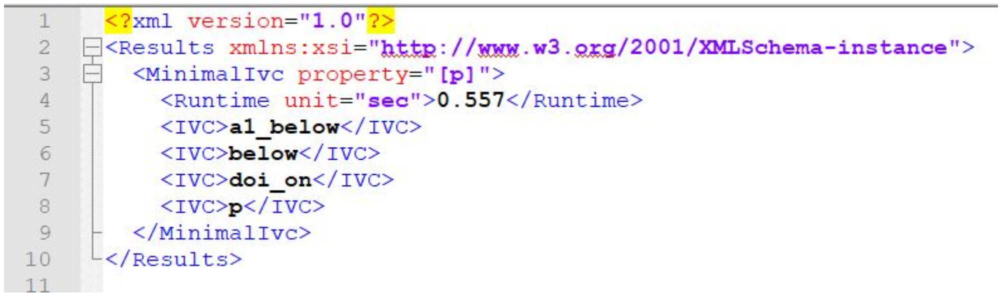
```xml
<?xml version="1.0"?>
<Results xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Progress source="bmc" trueFor="1">
    <PropertyProgress name="p"/>
  </Progress>
  <Property name="p">
    <Runtime unit="sec">1.109</Runtime>
    <Answer source="k-induction">valid</Answer>
    <K>0</K>
    <NumberOfIVCs>2</NumberOfIVCs>
    <MustElem>p</MustElem>
    <MustElem>doi_on</MustElem>
    <MustElem>below</MustElem>
    <IvcSet number="1">
    <Invariant>p</Invariant>
    <Ivc>p</Ivc>
    <Ivc>doi_on</Ivc>
    <Ivc>below</Ivc>
    <Ivc>a2_below</Ivc>
    </IvcSet>
    <IvcSet number="2">
    <Invariant>p</Invariant>
    <Ivc>p</Ivc>
    <Ivc>doi_on</Ivc>
    <Ivc>below</Ivc>
    <Ivc>a1_below</Ivc>
    </IvcSet>
  </Property>
</Results>
```

Figure 4.13: Sample output of ALL_IVCs engine



```
<?xml version="1.0"?>
<Results xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <MustElements property="[p]">
    <Runtime unit="sec">0.557</Runtime>
    <Must>below</Must>
    <Must>doi_on</Must>
    <Must>p</Must>
  </MustElements>
</Results>
```

Figure 4.14: Sample output of IVC_MUST engine



```
<?xml version="1.0"?>
<Results xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <MinimalIvc property="[p]">
    <Runtime unit="sec">0.557</Runtime>
    <IVC>a1_below</IVC>
    <IVC>below</IVC>
    <IVC>doi_on</IVC>
    <IVC>p</IVC>
  </MinimalIvc>
</Results>
```

Figure 4.15: Sample output of UCBF engine

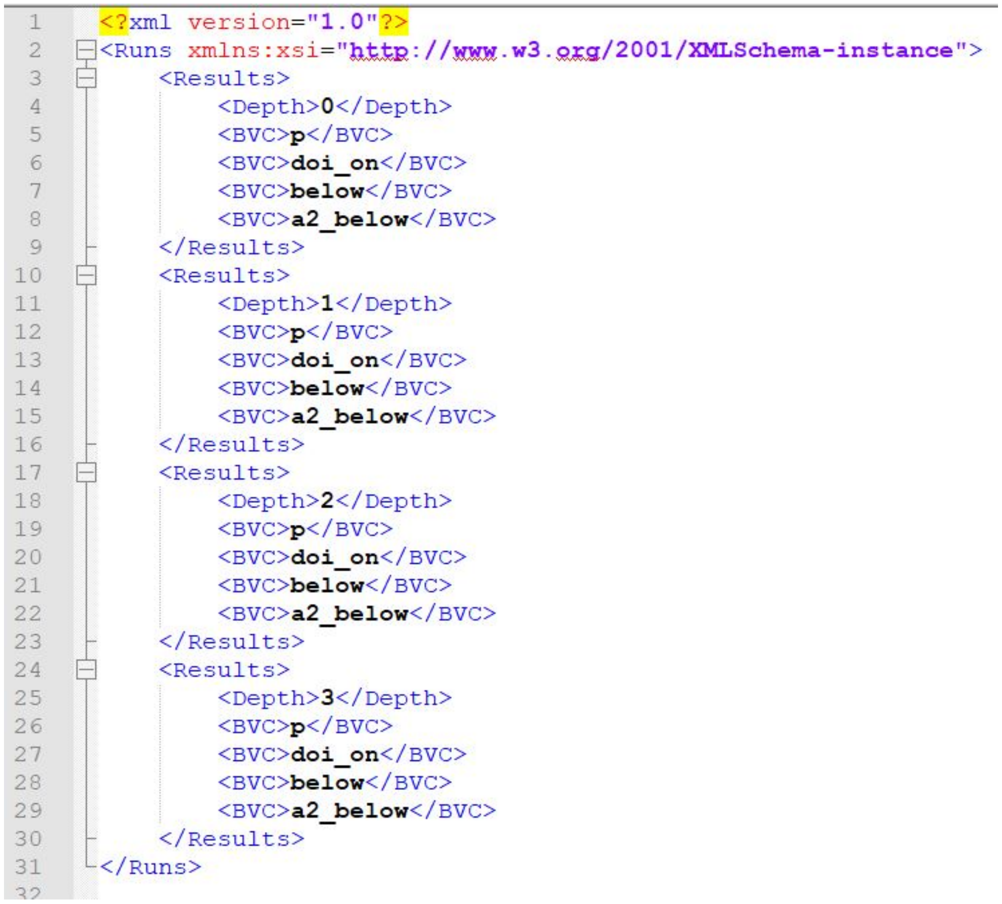
4.5 Integration of JKind and IVCs into Other Tools

`JKind` is the back-end for a variety of user-facing applications. In this section, we briefly highlight a few and how they employ the features discussed previously. `JKind` is developed in Java which makes it multi-platform and very easy to integrate into other Java applications. Moreover, it comes with `JKindApi` package which contains utilities for creating `Lustre` specifications, calling `JKind`, processing `JKind` results, graphically displaying real-time results, and nicely formatting counterexamples. Many of the applications in this section make heavy use of `JKindApi`.

4.5.1 Assume Guarantee Reasoning Environment

The *Assume Guarantee Reasoning Environment* (`AGREE`) [3, 114, 115] is an open-source compositional verification tool that proves properties of hierarchically-composed models in the Architectural Analysis and Design Language (AADL) language.

`JKind` is used as the default model checker for the Assume Guarantee Reasoning Environment (`AGREE`) [114]. `AGREE` refers to both an embedded language annex in the



```
<?xml version="1.0"?>
<Runs xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Results>
        <Depth>0</Depth>
        <BVC>p</BVC>
        <BVC>doi_on</BVC>
        <BVC>below</BVC>
        <BVC>a2_below</BVC>
    </Results>
    <Results>
        <Depth>1</Depth>
        <BVC>p</BVC>
        <BVC>doi_on</BVC>
        <BVC>below</BVC>
        <BVC>a2_below</BVC>
    </Results>
    <Results>
        <Depth>2</Depth>
        <BVC>p</BVC>
        <BVC>doi_on</BVC>
        <BVC>below</BVC>
        <BVC>a2_below</BVC>
    </Results>
    <Results>
        <Depth>3</Depth>
        <BVC>p</BVC>
        <BVC>doi_on</BVC>
        <BVC>below</BVC>
        <BVC>a2_below</BVC>
    </Results>
</Runs>
```

Figure 4.16: Sample output of BVC engine for bound 4

Architectural Analysis and Design Language (AADL) and to a plugin for the OSATE AADL Integrated Development Environment. The `AGREE` annex annotates the AADL model with formal requirements, and the plugin reasons about these requirements. The purpose of `AGREE` is to model behavioral requirements of an embedded system using formal assume guarantee contracts. The plugin generates Lustre specifications that are checked by `JKind`. `AGREE` makes use of multiple `JKind` features including smoothing and interval generalization to present clear counterexamples, IVC to show requirements traceability, and counterexample generation to check the consistency of an AADL component's contract. `AGREE` also uses `JKind` for test-case generation from component contracts.

4.5.2 Specification and Analysis of Requirements

The *Specification and Analysis of Requirements* (`SpeAR`) tool is an open source tool for prototyping and analysis of requirements [116]. Starting from a set of formalized requirements, `SpeAR` uses `JKind` to determine whether or not the requirements meet certain *properties*. It uses IVCs to create a traceability matrix between requirements and properties, highlighting unused requirements, over-constrained properties, and other common problems. `SpeAR` also uses `JKind` with smoothing for test case generation using the Unique First Cause criteria [117]. `SpeAR` captures requirements in a way that is backed by the formal semantics of `Lustre`, which enables them to be analyzed using model checking to ensure they are correct and consistent.

`SpeAR` uses `JKind` to prove properties over requirements, and uses IVC to create a traceability matrix between requirements and properties. This quickly highlights unused requirements, over-constrained properties, and other common problems. `SpeAR` also uses `JKind` for test case generation using the Unique First Cause criteria [117] by creating *trap properties*. Each trap property is expected to be falsifiable, but in such a way that the counterexample has exactly the desired properties for a given test case. `SpeAR` uses smoothing in `JKind` to ensure the resulting test cases are simple and understandable.

4.5.3 Static IMPerative AnaLyzer

The *Static IMPerative AnaLyzer* (`SIMPAL`) is a tool for performing compositional reasoning over software [118]. `SIMPAL` is based on `Limp`, a `Lustre`-like imperative language with extensions for control flow elements, global variables, and a syntax for specifying preconditions, postconditions, and global variable interactions of preexisting components. `SIMPAL` translates `Limp` programs to an equivalent `Lustre` representation which is passed to the `JKind` to perform assume-guarantee reasoning, reachability, and viability analyses. The feedback from these analyses is used to refine the program to ensure that the software functions as intended.

`JKind` is also used by two proprietary tools used by product areas within Rockwell Collins. The first is a *Mode Transition Table* verification tool used for the complex state machines which manage flight modes of an aircraft. `JKind` is used to check properties and generate tests for mode and transition coverage from `Lustre` models generated from the state machines. IVCs are used to establish traceability, i.e., which transitions are covered by which properties. The second is a *Crew Alerting System* MC/DC test-case generation tool for a proprietary DSL used for messages and alerts to airplane pilots. Smoothing is very important in this context as test cases need to be run on the actual hardware where timing is not precisely controllable. Thus, test cases with a minimum of changes to the inputs are ideal.

Chapter 5

Experiments

We would like to evaluate the different algorithms presented in Chapter 3. The use of `JKind` allows additional dimensions to our investigation: as mentioned before, it supports two different inductive algorithms: k-induction and PDR, and a "fastest" mode, that runs both algorithms in parallel. Also, `JKind` supports multiple back-end SMT solvers including `Z3` [105], `Yices` [106], `MathSAT` [107], and `SMTInterpol` [104].

This chapter is organized in two parts. First we evaluate IVCs and their relationship to model checking algorithms. There are two dominant model checking algorithms in modern solvers: PDR and k-induction, and we would like to determine (1) whether the choice of inductive algorithm affects the size of the IVCs calculated by `IVC_UC`, (2) whether different solvers are more or less efficient at producing approximate minimal IVCs (`IVC_UC`), and (3) whether running different solvers/algorithms leads to a diversity of solutions obtained by `IVC_UC`.

Second, using the `IVC_UC` algorithm as a baseline, we evaluate the performance the IVC algorithms presented in Chapter 3 in Section 5.2. Broadly speaking, we are interested in the following aspects: (4) The runtime efficiency of all IVC algorithms, (5) The efficacy (in terms of minimality) of the `IVC_UC` algorithm, (6) factors impacting performance of computing all IVCs. In terms of the sixth question, experimental data was used to construct the online algorithm; an accurate accounting of the relative cost of SAT and UNSAT problems is necessary to create an efficient algorithm for this problem.

Finally, we examine the relationship between inductive validity cores and bounded validity cores in Section 5.4. There are many cases where systems become to large or complex to find an inductive proof; in such cases it is still possible to perform a bounded proof (in terms of the number of steps). Bounded validity cores are always underapproximations of some IVC, so can be used to witness traceablity and adequacy relationships between properties and implementation elements. Several questions are of interest: (1) how similar are BVCs of different depths to IVCs? (2) how quickly do they tend to converge to a specific size?, and (3) is the converged set the same as one of the MIVCs? This initial experiment is designed to lay the groundwork for future explorations of the relationship between these two concepts.

The initial experiments allow us to choose an optimal configuration for `JKind` based on which we can run our major experiments. The major experiments are conducted to evaluate the efficiency and efficacy the IVC algorithms presented in Chapter 3. In summary, we are interested in the following aspects:

- Evaluating the performance of `IVC_UC` algorithm and `All_IVCs`,
- Evaluating the efficacy (minimality) of the `IVC_UC` algorithm outcome,
- Effective factors on the performance of finding all minimal IVCs,
- Evaluating the online approach for generating all minimal IVCs,
- Studying validity cores in bounded model checking.

For this purpose, we organize the major experiments in different categories; Section 5.2 is about the evaluation of our two key proposed algorithms: `IVC_UC` and `All_IVCs`. Then in Section 5.3, we examine our approach for calculating all minimal IVCs in the online manner as described in 3.3.2. Finally in Section 5.4, we investigate a couple of interesting research questions related to bounded validity cores.

5.1 On the Relationship between Model Checking Algorithms and IVCs

In this section we study the effect of different model checking algorithms/solvers on the performance and accuracy of the `IVC_UC` algorithm. We started from a suite of 700 Lustre models developed as a benchmark suite by Hagen and Tinelli [119]. We augmented this suite with 81 additional models from recent verification projects including avionics and medical devices [3, 115]. Most of the benchmark models from [119] are small (with 6-40 equations) and contain a range of hardware benchmarks and software problems involving counters. The additional models are much larger: with over 300 up to 10000 equations. We added the new benchmarks to better check the scalability for the tools, especially with respect to the brute force algorithm. Each benchmark model has a single property to analyze. For our purposes, we are only interested in models with a *valid* property (though it is perhaps worth noting that there is no additional computation—and thus no overhead—using the `JKind` IVC options for *invalid* properties). In our benchmark set, 295 models yield counterexamples, and 10 additional models are neither provable nor yield counterexamples in our test configuration (see next paragraph for configuration information). The benchmark suite therefore contains 476 models with valid properties, which we use as our initial test subjects.

For each test model, we computed `IVC_UC` in 12+1 configurations: the twelve configurations were the cross product of all solvers {`Z3`, `Yices`, `MathSAT`, `SMTInterpol`} and inductive algorithms {k-induction, PDR, fastest}, and the remaining (+1) configuration was an instance of `IVC_BF` run on `Yices`, which is the default solver in JKind. In addition, for each of the 12 configurations, we ran an instance of `JKind` without IVC to examine overhead. The initial experiments were run on an Intel(R) i5-2430M, 2.40GHz, 4GB memory machine, with a 1 hour timeout for each analysis on any model. The data gathered for each configuration of each model included the time required to check the model without IVC, with IVC, and also the set of elements in the computed IVC.[1]

Note that not all analysis problems were solvable with all algorithms: for all solvers, k-induction (without IVC) was unable to solve 172 of the examples. When comparing minimality of different solving algorithms, we only considered cases where both algorithms provided a solution.

For this study, we are interested in the following research questions:

- **RQ1:** Does the choice of SMT solver affect the performance of `IVC_UC`?
- **RQ2:** Does the choice of SMT solver or algorithm used to produce a proof (k-induction or PDR) matter in terms of the minimality of the IVCs generated by

[1]The benchmarks, all raw experimental results, and computed data are available on [120].

Table 5.1: IVC_UC runtime with different solvers

runtime (sec)	min	max	mean	stdev
Z3	0.005	2.335	0.192	0.355
Yices	0.014	13.297	0.589	1.473
SMTInterpol	0.029	19.254	1.396	2.991
MathSAT	0.011	86.421	3.071	10.403

IVC_UC?

- **RQ3:** Do different solvers and algorithms lead to different minimal cores for IVC_UC?

RQ1

First, we examine the performance overhead of the `IVC_UC` algorithm over the time necessary to find a proof using inductive model checking. To examine this question, we use the default *fastest* option of JKind which terminates when either the k-induction or PDR algorithm finds a proof. To measure the performance overhead of the `IVC_UC` algorithm, we execute it over the proof generated by the *fastest* option.

Since the `IVC_UC` algorithm uses the UNSAT core facilities of the underlying SMT solver, the performance is dependent on the efficiency of this part of the solver. Looking at Tables 5.1 and 5.2, it is possible to examine both the computation time for analysis using the four solvers under evaluation and the overhead imposed by the `IVC_UC` algorithm. Figure 5.1 allows a visualization of the runtime for the `IVC_UC` algorithm running different solvers. The lines are sorted individually based on the running time of the `IVC_UC` for each model. The data suggests that `Yices` (the default solver in `JKind`) and `Z3` are the most performant solvers both in terms of computation time and overhead.

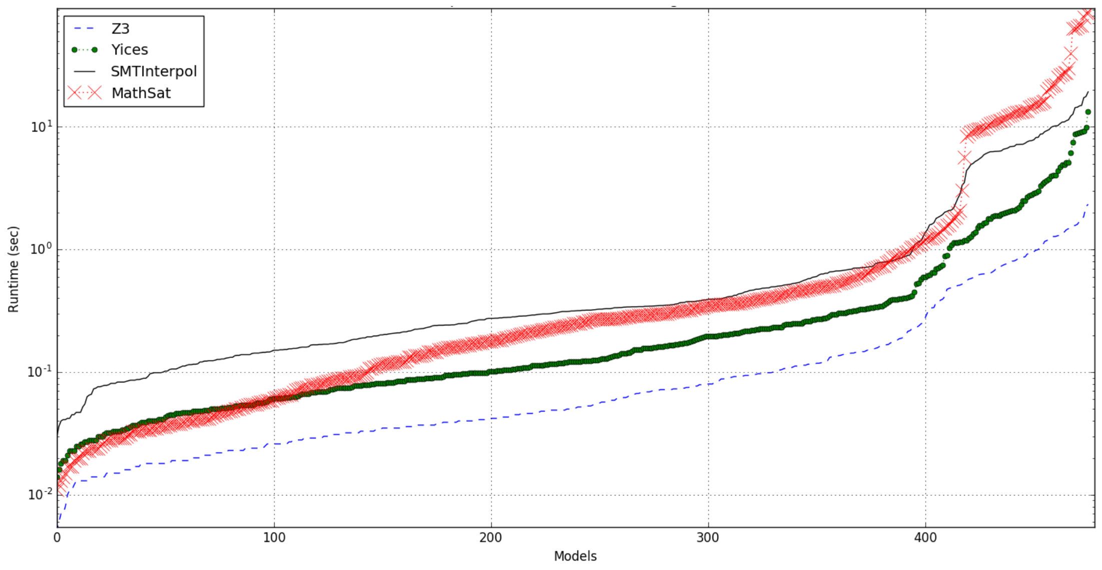


Figure 5.1: `IVC_UC` performance on different solvers

Table 5.2: Overhead of `IVC_UC` computations using different solvers

solver	min	max	mean	stdev
Z3	0.73%	84.13%	17.38%	16.92%
Yices	0.17%	351.47%	52.20%	54.50%
SMTInterpol	1.46%	175.75%	46.81%	37.35%
MathSAT	0.78%	955.52%	80.21%	112.92%

RQ2

As described in Chapter 3, the `IVC_UC` algorithm is not guaranteed to produce minimal cores due in part to the role of invariants used in producing a proof; as k-induction and PDR use substantially different invariant generation algorithms, it is likely that the set of necessary invariants for proofs are dissimilar, and that this would in turn affect the number of model elements required for the proof. It is possible that one or the other algorithm is more likely to yield smaller invariant sets. In addition, differences in the choice of the UNSAT core algorithms in the different solvers could affect the size of the generated core. However, our algorithm already performs a minimization step on UNSAT cores, and thus the only differences would be due to one algorithm leading to a different minimal core than another.

As mentioned, k-induction is unable to solve all of the analysis problems; therefore we include only models that are solvable using *both* k-induction and PDR by *all solvers*, 304 models in all. Examining the aggregate data in Table 5.4, we can see the sizes of cores produced by different algorithms and solvers.

RQ3

In this section, we examine the issue of diversity: do different solvers and algorithms lead to *different* minimal cores? This is both a function of the models and the solution algorithms: for certain models, there is only one possible minimal IVC set, whereas other models might have many. Given that there are multiple solutions, the interesting question is whether using different solvers and algorithms will lead to different solutions. The reason diversity is considered is that it has substantial relevance to some of the uses of the tool, e.g., for constructing multiple traceability matrices from proofs (see Section 6.1). Note that our exploration in this experiment is not exhaustive, but only exploratory, based on the IVCs returned by different algorithms and tools; we leave exhaustive exploration of IVCs for future work.

To measure diversity of the generated IVCs, we use Jaccard distance:

Definition 7. *Jaccard distance:* $d_J(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|}$,
$0 \leq d_J(A,B) \leq 1$

Table 5.3: Pairwise Jaccard distances among all models

min	max	mean	stdev
0.0	0.878	0.026	0.059

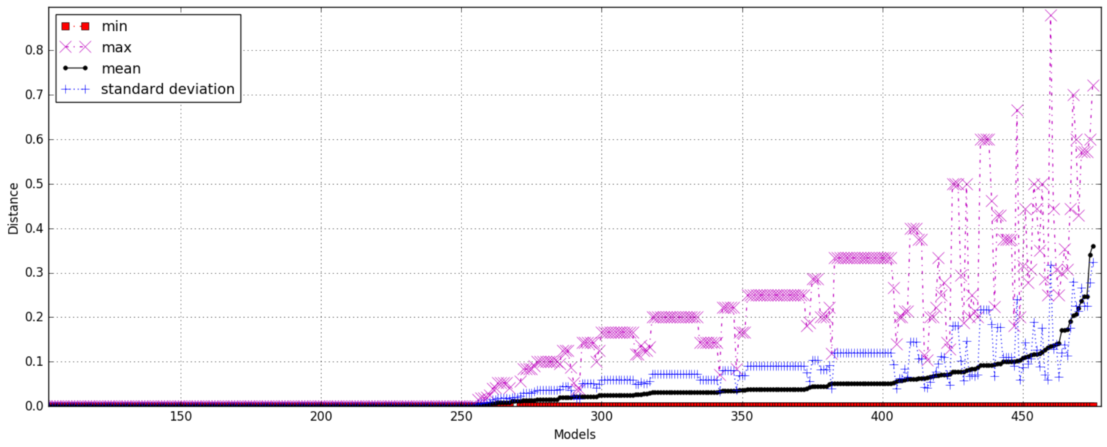


Figure 5.2: Pairwise Jaccard distance between IVCs

Jaccard distance is a standard metric for comparing finite sets (assuming that both sets are non-empty) by comparing the size of the intersection of two sets over its union. For each model in the benchmark, the experiments generated 13 potentially different IVCs. Therefore, we obtained $\binom{13}{2} = 78$ combinations of pairwise distances per model. Then, minimum, maximum, average, and standard deviation of the distances were calculated (Figure 5.2), by which, again, we calculated these four measures among all models. As seen in Table 5.3, on average, the Jaccard distance between different solutions is small, but the maximum is close to 1, which indicates that even for our exploratory analysis, there are models for which the tools yield substantially diverse solutions. The diversity between solutions is represented graphically in Figure 5.2, where for each model, we present the min, max, and mean pairwise Jaccard distance of the solutions produced by algorithm `IVC_UC` for each model, sorted by the mean distance.

Table 5.4: Aggregate IVC sizes produced by `IVC_UC` using different inductive algorithms and solvers

solver	PDR	k-induction	**total**
Z3	2378	2379	4757
Yices	2384	2376	4760
MathSAT	2375	2369	4744
SMTInterpol	2378	2368	4746
total	9515	9492	

5.2 On the efficiency and efficacy of different "offline" IVC algorithms

We would like to evaluate the cost of computing one single IVC using the brute-force algorithm (`IVC_UCBF`) and the UNSAT core-based algorithm (`IVC_UC`). Then, we are interested in examining the *efficacy* and *efficiency* of generating all minimal IVCs, as compared to algorithms for computing a *single approximately minimal* IVC (`IVC_UC` algorithm), and a *minimal IVC* (`IVC_UCBF` algorithm). We would also like to know how performance is affected by the size of models and number of minimal IVCs. Next, we are also interested in examining the minimality of the cores found by `IVC_UCBF` vs `IVC_UC`. Finally we would like to determine whether the `All_IVCs` algorithm generates *smaller* cores than are generated by the `IVC_UCBF` algorithm. Therefore, we investigate the following research questions:

- **RQ1:** How expensive is it to compute IVCs? For this question we examine the cost of the `IVC_UC` and `IVC_UCBF` algorithms that find a single approximately minimal and guaranteed minimal IVC,respectively, as well as the `All_IVCs` algorithm for determining all minimal IVCs.
- **RQ2:** How is the performance of the `All_IVCs` algorithm affected by the baseline proof time and the number of IVCs that can be found for a property?

- **RQ3:** How close to minimal are the IVCs computed by `IVC_UC` as opposed to the (guaranteed minimal) `IVC_UCBF` and the *minimum* IVC computed by `All_IVCs`? How do the sizes of IVCs compare to static slices of the model?

5.2.1 Experimental Setup

To perform the experiments, we augmented the 476 benchmarks in Section 5.1 with more complex and larger models. We have collected a set of benchmarks containing 660 Lustre models, including all of the benchmark models yielding a valid result (530 in total) from [59, 119] and 130 industrial models yielding valid results derived from an infusion pump system [3] and other sources [59, 121]. The benchmark includes 2 models from the NASA Quad-redundant Flight Control System (QFCS) [121]: the Flight Control System (FCS) with 5259 Lustre equations and the Flight Control Computer (FCC) with 10969 equations.

We selected only benchmark problems consisting of a Lustre model with properties that `JKind` could prove with a 3-hour timeout. In initial evaluation (Section 5.1), we showed that the `IVC_UC` algorithm using the `Z3` and `Yices` SMT solvers adds a modest performance penalty to the time required for inductive proofs compared to others. Moreover, we concluded that neither PDR nor k-induction yields a smaller inductive validity core in general. And, the choice of underlying SMT solver does not substantially affect the size of the inductive validity cores. Therefore We choose `Z3` for the rest of our experiments. For each test model, we computed `All_IVCs`, `IVC_UC`, and `IVC_UCBF` algorithms in a configuration with the `Z3` solver and the "fastest" mode of `JKind` (which involves running the k-induction and PDR engines in parallel and terminating when a solution is found). The experiments were run on an Intel(R) i5-4690, 3.50GHz, 16 GB memory machine running Linux, and are available online [120]. [2]

5.2.2 Experimental Results

In this section, we examine our experimental results to address the research questions defined in the experiment.

[2]We will use this configuration and machine for the rest of the experiments presented in this chapter.

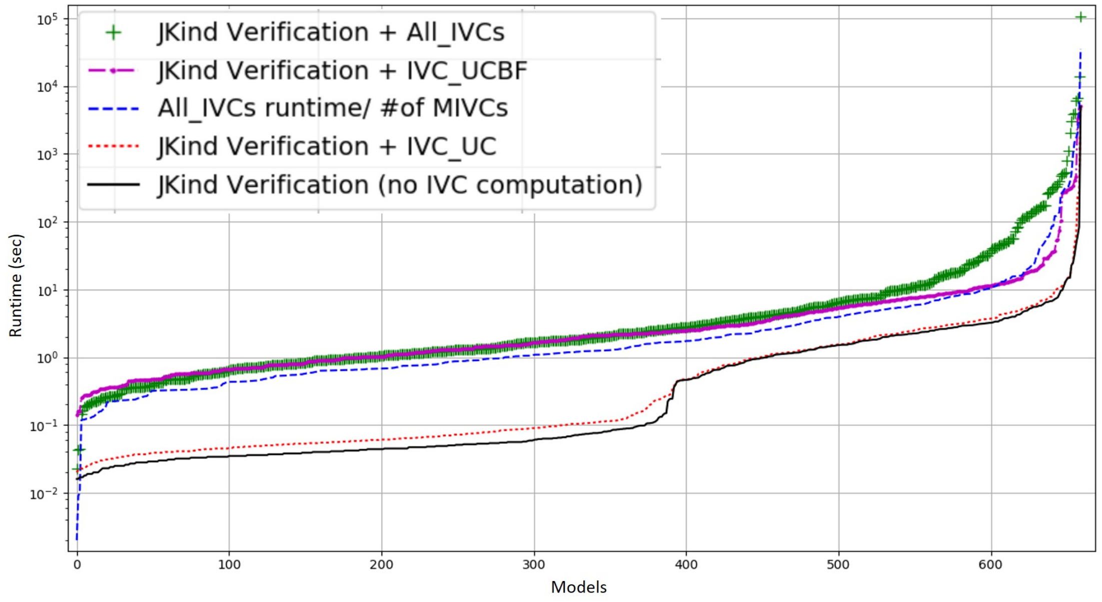


Figure 5.3: Runtime of `All_IVCs`, `IVC_UCBF`, and `IVC_UC` algorithms

RQ1

To address RQ1, we measured the performance overhead of the various IVC algorithms against the baseline time necessary to find a proof using inductive model checking. Figure 5.3 provides an overview of the overhead of the `All_IVCs` algorithm in comparison with the `IVC_UC` and `IVC_UCBF` algorithms. In the figure, each curve is sorted along the x-axis according to the time required for the algorithm to terminate for each analysis problem. Table 5.5 provides a summary of the computation time and the overhead of different algorithms. The `IVC_UC` algorithm imposes a 31% overhead to the baseline proof time, whereas both the `IVC_UCBF` and `All_IVCs` algorithms add a substantial time penalty: `IVC_UCBF` and `All_IVCs` add a (mean) 2276% and 9588% overhead, respectively, to the proof time. For small models, much of this penalty is due to starting many instances of the SMT solver; if we examine models that require $\geq 6s$ of analysis time, the mean overhead of `All_IVCs` over the baseline analysis drops from 9588% to 2514%.

Table 5.5: Runtime of different computations

runtime (sec)	min	max	mean	stdev
proof-time	0.016	4961	1.250	192.97
`All_IVCs`	0.002	105460	229.16	4148.85
`IVC_UCBF`	0.139	5016.6	28.449	294.92
`IVC_UC`	0.001	188.241	0.730	10.019

RQ2

For this research question, we examine how the proof time of the original model and the number of MIVCs associated with the property affects the analysis time of the `All_IVCs` algorithm. Figure 5.4 provides an overview of this data. The data in Figure 5.4 is sorted twice along the x-axis: the major axis is the number of MIVCs that exist for the model, and the minor axis is the analysis time of the baseline model. In this graph, we can visualize how both factors affect the performance of the `All_IVCs` algorithm. Note that there are two scales for the y-axis: the scale on the left is a logarithmic scale of performance in terms of the run time; the scale on the right is a linear scale based on the number of minimal IVCs discovered.

Figure 5.4 shows two distinct trends. First, for models whose baseline proofs are inexpensive and that only have a single MIVC, the `All_IVCs` is roughly equivalent in performance to the `IVC_UCBF`. However, as proofs become more expensive for a single MIVC, the `All_IVCs` begins to underperform the `IVC_UCBF`, this is the case for the properties with one MIVC. In the cases where several MIVCs are found, the performance of the `All_IVCs` is driven to a large degree by the number of MIVCs that exist : the more MIVCs associated with a property, the higher the expense of `All_IVCs` as compared to the `IVC_UCBF` algorithm.

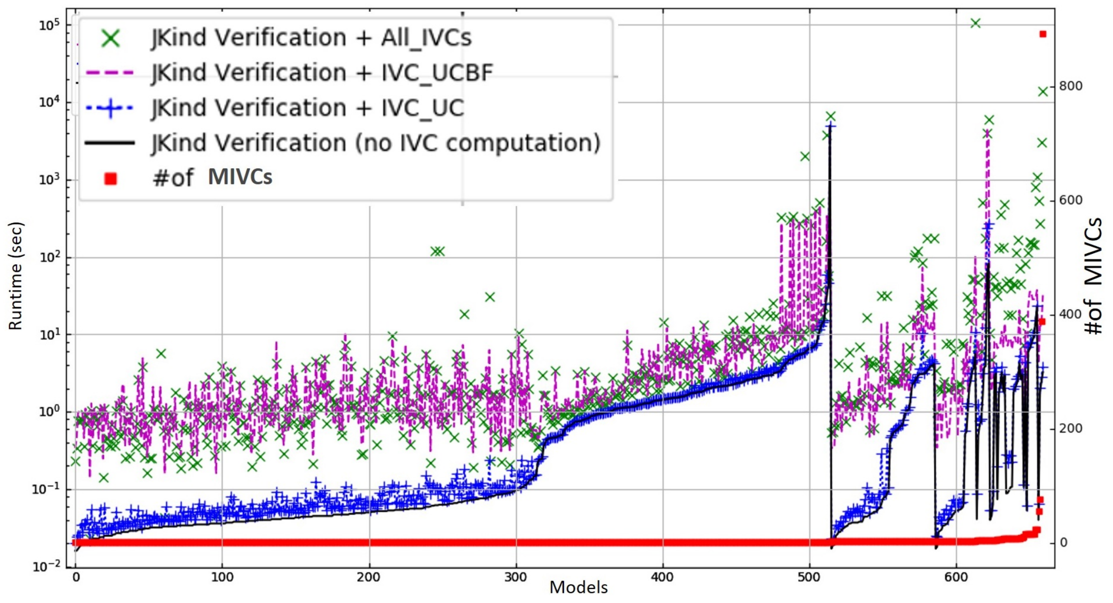


Figure 5.4: Runtime of different computations along with the number of MIVCs

RQ3

For this research question, we analyzed the minimality of the produced IVCs by each algorithm. Results are shown in Figure 5.5. Since 515 of the models had only one MIVC, for these models, the size of the minimum model produced by the `All_IVCs` algorithm should be the same as the `IVC_UCBF` algorithm. For the remainder, even when multiple MIVCs were produced, in only 47 cases did the `All_IVCs` produce smaller minimal IVCs. For these 47 models, the smallest MIVC was 44% the size of the MIVC produced by `IVC_UCBF`, and in the most dramatic case, the number of elements shrank from 62 to 35.

The final question asks how well the approach compares to *backwards static slicing* [42], since slicing also reduces the set of model elements necessary to construct a proof. We start the slice from the equation defining the property of interest, and use the usual approach [122] that performs an iterative backward traversal from the variables used within an equation to their defining equations. We expect the IVC mechanism to be more precise, because the slice overapproximates the set of equations necessary for *any* proof. This claim is demonstrated in Figure 5.5; slices are (mean) 574% larger than the IVCs produced by our `IVC_UC` algorithm and 597% larger than those produced by

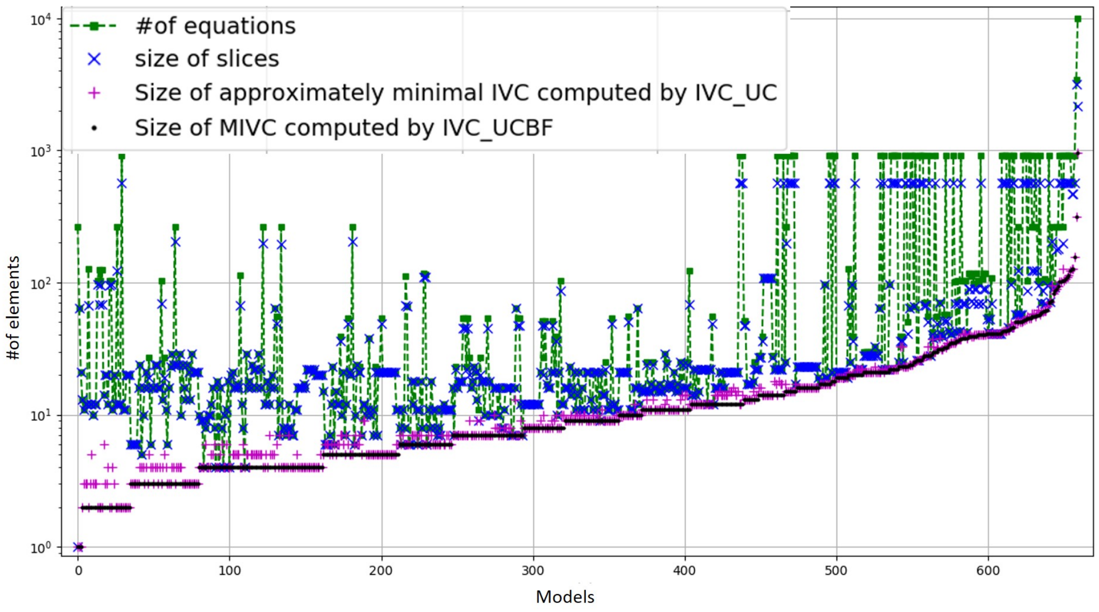


Figure 5.5: Size of the IVC sets produced by different algorithms

IVC_UCBF algorithm.

5.3 On the relative performance of offline and online algorithms for all IVCs.

We are interested in examining the performance of algorithms to compute minimal IVCs. We examine **Grow-Shrink**, the algorithm presented in Section 3.3.2, and the two state-of-the-art algorithms: **Offline MARCO** (All_IVCsSection 3.3.1), and **Online MARCO** (Section 3.3.2) that performs a shrink step prior to returning a solution to ensure minimality. We investigate the following research questions:

- **RQ1:** For the large models where the complete MIVC enumeration is intractable, how many MIVCs are found within the given time limit?
- **RQ2:** For the tractable models, i.e., models in which all MIVCs are found, how much time is required to complete the enumeration of MIVCs?
- **RQ3:** Finally, we are interested in how many solver calls are necessary for the

enumeration. What is the (average) number of solver calls with result adequate/inadequate required by evaluated online algorithms to produce individual MIVCs?

5.3.1 Experimental Setup

We start from a benchmark suite that is a superset of the benchmarks used in the previous experiments. This suite contains 660 models, and includes all models that yield a valid result (530 in total) from previous Lustre model checking papers [59, 119] and 130 industrial models yielding valid results derived from an infusion pump system [3] and other sources [59, 121]. As this paper is concerned with analysis problems involving multiple MIVCs, we include only models that had more than 4 MIVCs (46 models in total). To consider problems with many IVCs, we took those models and mutated them, constructing 20 mutants for each model. The mutants varied both in the number and in the size of individual MIVCs. We added the mutants that still yielded valid results and have more than 5 MIVCs (384 in total) back to the benchmark suite. Thus, the final suite contains 430 Lustre models. The original benchmarks and our augmented benchmark are available online[3].

For each test model, we configured `JKind` to use the `Z3` solver and the "fastest" mode of `JKind` (which involves running the k-induction and PDR engines in parallel and terminating when a solution is found). The experiments were run on a 3.50GHz Intel(R) i5-4690 processor 16 GB memory machine running Linux with a 30 minute timeout. All experimental data is available online[4].

5.3.2 Experimental Results

In this section, we examine the experimental results to address the research questions.

RQ1 and RQ2

Data related to the first two research questions are shown in Figures 5.6 and 5.7. Figure 5.6 describes the number of MIVCs found be the two online algorithms in the intractable benchmarks, i.e., the benchmarks where the algorithms did not complete

[3]`https://github.com/elaghs/benchmarks`

[4]`https://github.com/jar-ben/online-mivc-enumeration`

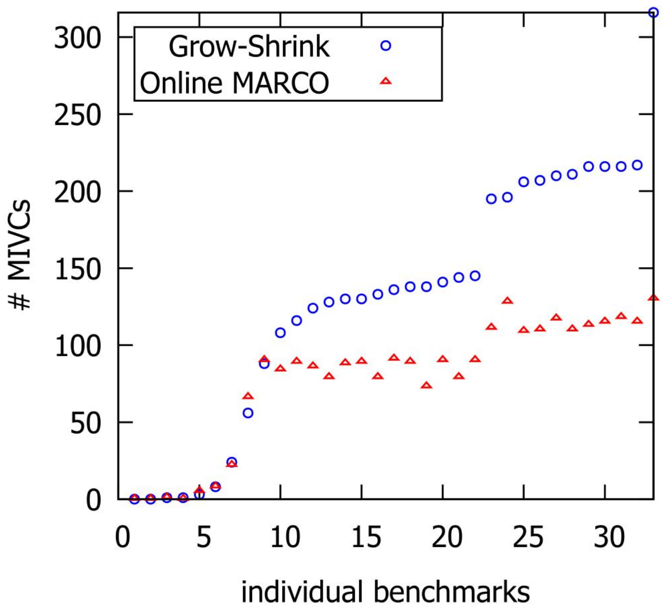


Figure 5.6: Number of MIVCs produced by online algorithms.

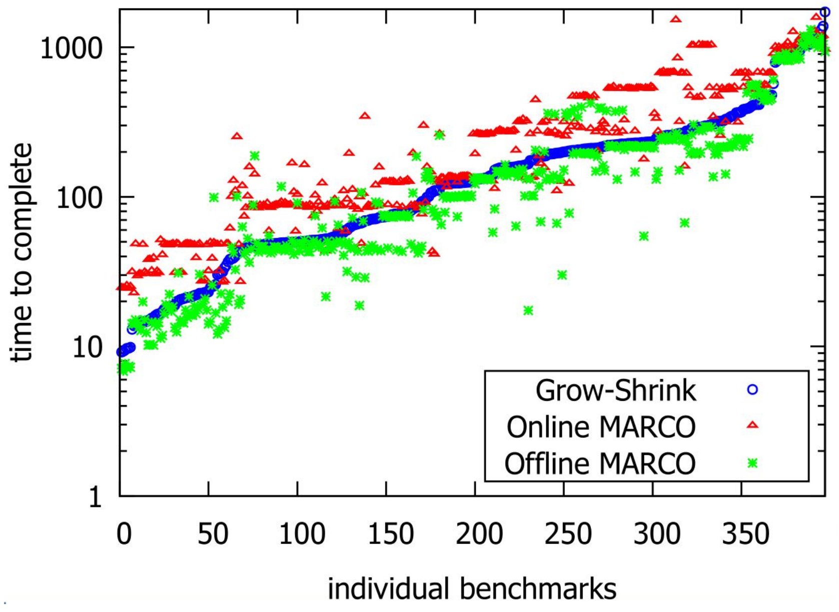


Figure 5.7: Runtime for tractable benchmarks for all algorithms in a log scale.

the computation within the time limit. There are 33 such benchmarks. The Grow-Shrink substantially outperforms Online MARCO in the majority of the benchmarks, finding an average of 55% additional MIVCs.

Figure 5.7 describes the time for each algorithm needed to complete the computation in the case of 397 tractable benchmarks.

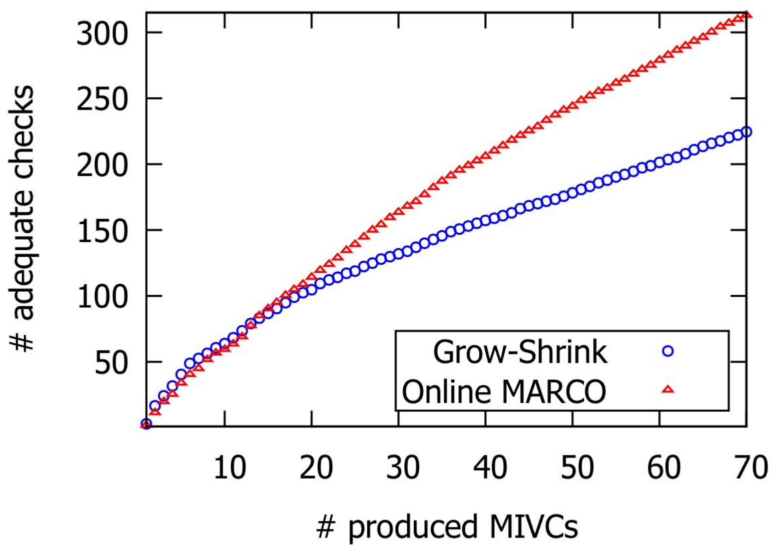
Grow-Shrink is on average only 1.08 times slower than Offline MARCO, yet as previously discussed, has the advantage of returning guaranteed MIVCs, rather than approximate MIVCs. It is on average 1.50 times faster than Online MARCO.

RQ3

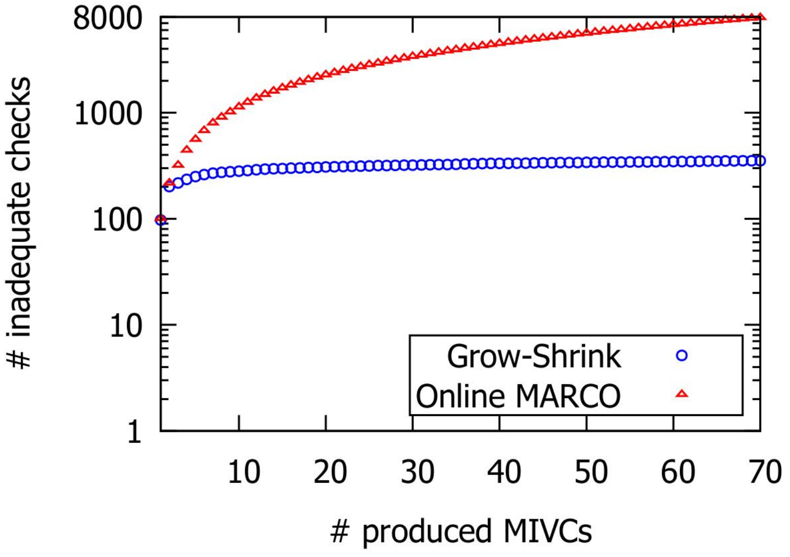
For RQ3, we examined the number of required calls to the solver per MIVC. For this question, we used the 33 models that contained a large number of MIVCs (>70) in order to show the solver efficiency as the number of MIVCs increased. A point with coordinates (x, y) states that the algorithm needed to perform y solver calls (on average) in order to produce the first x MIVCs. We grouped the calls in terms of the number of calls that returned *adequate* vs. *inadequate* results. It is evidenced by the results in Figure 5.8, the new algorithm improves upon Online MARCO as the number of MIVCs becomes larger.

The improvement in the number of *inadequate* calls is due the novel shrinking and growing procedures. Each (approximately) maximal inadequate subset found by the growing procedure allows to save (up to exponentially) many inadequate calls during subsequent executions of the shrinking procedure. Indeed, the Grow-Shrink algorithm performed on average only 353 inadequate calls to output the first 70 MIVCs, whereas the online MARCO needed to perform 7775 calls to output the same number of MIVCs.

The improvement in the number of *adequate* calls is not so significant as in the case of inadequate calls. Yet, since the adequate calls are usually much more time consuming than inadequate ones, even a slight saving in the number of adequate calls might significantly speed up the whole computation. The Grow-Shrink algorithm saves adequate calls due to the usage of the shrinking queue and due to the invariants that are maintained by the queue. In particular, shall two comparable sets appear in the queue, only the smaller is left. Thus, the algorithm avoids shrinking of relatively large sets and saves some adequate calls.



(a) Checks with result "adequate"



(b) Checks with result "inadequate"

Figure 5.8: Average number of performed adequacy checks required to produce individual MIVCs. Note that Figure (b) is in a log scale.

5.4 On the relationship of BVCs and MIVCs

We would like to to observe and study how validity cores evolves over unrolling the transition relation. It is interesting to see how quickly validity cores from a bounded proof converges to an actual minimal IVC.

Note that the purpose of our experiments on BVCs is mostly to point out some research directions. Further studies may even can make use of BVCs in verification problems. It can be the case that a valid property is hard or impossible for an inductive model checker to prove. In such cases, looking at the history of BVC runs may give us some confidence about the correctness of the property. The experimental results show that when we reach one of the actual MIVCs, the BVC algorithm then constantly generates the same cores as depth of exploration increases. That is to say, when the BVC runs begin generating stable cores that do not change as depth changes, it may imply we might have already seen all the reachable states, and implicitly known they are safe. Although this hypothesis is by no means guaranteed to hold, it may be worth further investigations.

5.4.1 Experimental Setup

We perform our experiments on the same benchmark suite with 660 models introduced in Section 5.2.1. The experiment is conducted with a maximum depth of 10 and one hour timeout; i.e., for each model, if unrolling to depth 10 takes more than one hour, the `BVC` algorithm will terminate. We capture BVC_k for $0 \leq k \leq 10$, then compare each BVC of depth k to see how they change during unrolling. Then, the final bounded validity cores obtained from at the maximum[5] reachable depth in one hour, denoted by BVC_{max} , are considered as our final cores. These cores are compared with all the MIVCs gathered in Section 5.2.2 to see if they match up with any of the actual minimal IVCs.

Research questions we would like to answer in this study are as follows:

- **RQ1:** How many of the final BVCs match one of the $MIVC$s?
- **RQ2:** How do BVCs evolve as the analysis depth changes?

[5]Maximum depth in this experiment is 10. For most of the models, it is possible to reach this depth in less than an hour.

- **RQ3:** Is there a relationship between size and structure of models and the size of BVCs and the rate at which they converge with a $MIVC$?

RQ1

The result of the experiments show that BVC_{max} is the same as one of the MIVCs for 474 models out of 660. For 27 of the models, BVC_{max} was not subset of any MIVCs (had additional elements, also none of the MIVCs was a subset of the BVC_{max})[6]. However, BVC_{max} was a subset of one of the MIVCs in 159 of the models.

We performed the experiments with `Z3` and `Yices` solvers. UNSAT core generation in `Z3` is faster than `Yices` in the current implementation of `JKind`. Using `Z3`, 12 of the models did not reach depth 10 in one hour. With `Yices`, 18 of the models timed out. An interesting fact is that we had models that did not reach BVC_{10}, but their BVC_{max} was the same as one of the MIVCs. For example, one of the models containing 571 design elements only reached to BVC_2, but BVC_2 was the same as one of the MIVCs. The BVC size for that model at different depths is as follows:[7]

$|BVC_0| = 6$, $|BVC_1| = 11$, $|BVC_2| = 128$

There are interesting case studies where from the initial depth, the BVC was the same as one of the MIVCs. For example, in our benchmark we have a model with 27 design elements[8], for which $|BVC_i| = 5$, $i \leq 0 \leq 10$, and BVC_0 is the same as its only one MIVC.

RQ2

Our experimental results show that among 474 models for which BVC_{max} is the same as one of the MIVCs, the size of the BVCs were (nonstrictly) increasing 99.9% of the time:

$$0 \leq i \leq max, |BVC_i| \leq |BVC_{i+1}|$$

[6]We will explain the reason in **RQ2**

[7]This particular model is named "steam_boiler_no_arr1.lus" in our benchmarks. You can see the results and model in our experimental directories [120].

[8]File "car_all_e8_856_e2_585.lus" in our benchmark directory.

In other words, for only 12 of these models, the above relation did not hold. It is expected that bounded cores in each unrolling step (nonstrictly) increase as in each step more states are being reached and the cores required for the proof of the property is more likely to expand. We run the experiments over those 12 models with different solvers (once with `Z3` and once with `Yices`). The result of BVC runs for these models (on `Yices`) is shown in Table 5.6.

It is interesting to see why those 12 models show different behavior. By looking at different case studies, we have found three main reasons that explain this anomaly. The first explanation is that when a model has several distinct MIVCs, the bounded core could change during unrolling. However, the set of 12 models contain models that have only a single MIVC, so this cannot be the entire reason. Another explanation for such models is that MIVCs obtained from `All_IVCs` contained timeout loops; therefore, we do not have the exact minimal IVCs for those cases (for example model#6 in Table 5.6).

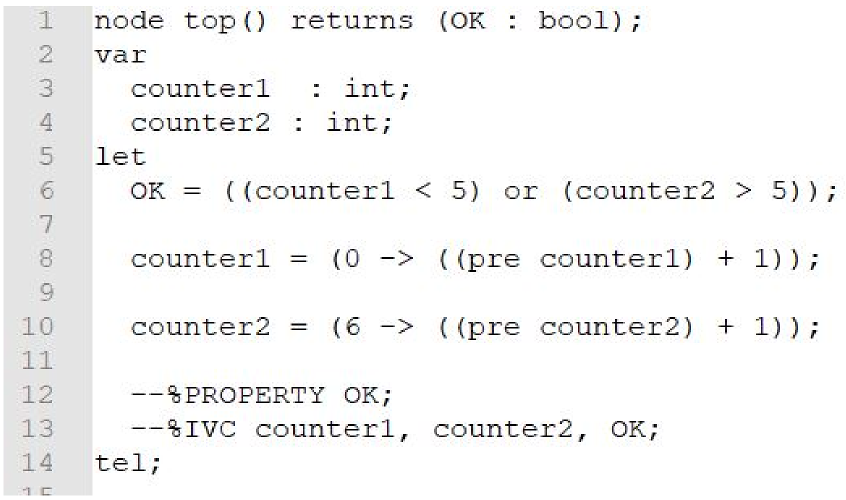
The third reason why the size of BVCs is not always increasing is more interesting. This reason explains why in some cases BVC_{max} is not the subset of any of the MIVCs. This only has to do with the depth of bounded model checking. In some problems, when we are at the earlier steps of unrolling transition relation (i.e., lower depths), the property can be satisfied in different ways. In other words, a property may have multiple BVCs at depth k, but as we advance towards the deeper bounds leading to a proof, the validity cores converge to a smaller subset. For example, consider the toy example in Figure 5.9. It shows a simple model containing two counters. The first counter (`counter1`) has the initial value of 0, and the second counter (`counter2`) starts off from 6. The property (`OK`) is either `counter1` is less than 5 or (`counter2`) is greater than 5. This property has only one MIVC, which is {`counter2, OK`}. However, before depth 5, this property can be satisfied into ways with {`counter1, OK`} and {`counter2, OK`}. You can see the output of the BVC engine for this example in Figure 5.10.

Let us take a look at one of the models that is small enough to display its results in a reasonable amount of space. This model is named *ex3_e8_381_e7_224* in our benchmarks (Figure 5.11). It only has one MIVC ({`V19_late, V64_incr, V63_diff, V65_PC, OK`}), and `JKind` is able to prove its property in less than a second. In addition, for this model, there is no timeout issue in the inner loops of `All_IVCs` algorithm. Using `Yices` in our experiments, BVC_{max} is not the subset of the only MIVC that this model has. However

Table 5.6: BVC runs for the models with non-increasing behavior where BVC_{max} is the same as one of the MIVCs.

$\|BVC_i\|$ / $i =$	0	1	2	3	4	5	6	7	8	9	model size	#of MIVCs
model#1	2	9	34	36	28	28	28	28	28	28	70	1
model#2	5	15	11	11	11	11	11	11	11	11	123	1
model#3	8	9	13	33	28	40	38	41	41	41	57	7
model#4	2	5	8	10	12	10	10	10	10	10	64	9
model#5 (`Yices`)	9	24	84	84	82	82	82	82	82	82	96	1
model#5 (`Z3`)	9	24	82	82	82	82	82	82	82	82	96	1
model#6	5	6	5	7	5	7	5	7	5	7	7	1
model#7	5	6	6	5	5	6	6	5	5	6	6	1
model#8 (`Yices`)	9	12	14	28	37	36	36	36	36	36	103	1
model#8 (`Z3`)	9	12	14	28	37	37	37	37	37	37	103	1
model#9 (`Yices`)	2	6	10	4	4	4	4	4	4	4	64	1
model#9 (`Z3`)	2	4	4	4	4	4	4	4	4	4	64	1
model#10	2	6	8	11	7	7	7	7	7	7	64	1
model#11	4	13	32	47	61	54	54	54	54	54	103	8
model#12 (`Yices`)	8	8	21	29	39	38	38	40	41	41	57	6
model#12 (`Z3`)	8	17	21	29	32	38	38	32	32	32	57	6

Actual model names in the benchmark, respectively: fast_1_e8_751.lus, microwave05.lus, DRAGON_12.lus, Display_Control-Gaurantee0, fast_2_e8_976.lus, twisted_counters.lus, two_counters.lus, cruise_controller_04.lus, Display_Control-Gaurantee2.lus, Display_Control-Gaurantee1.lus, cruise_controller_24.lus, DRAGON_13.lus



```
node top() returns (OK : bool);
var
  counter1  : int;
  counter2 : int;
let
  OK = ((counter1 < 5) or (counter2 > 5));

  counter1 = (0 -> ((pre counter1) + 1));

  counter2 = (6 -> ((pre counter2) + 1));

  --%PROPERTY OK;
  --%IVC counter1, counter2, OK;
tel;
```

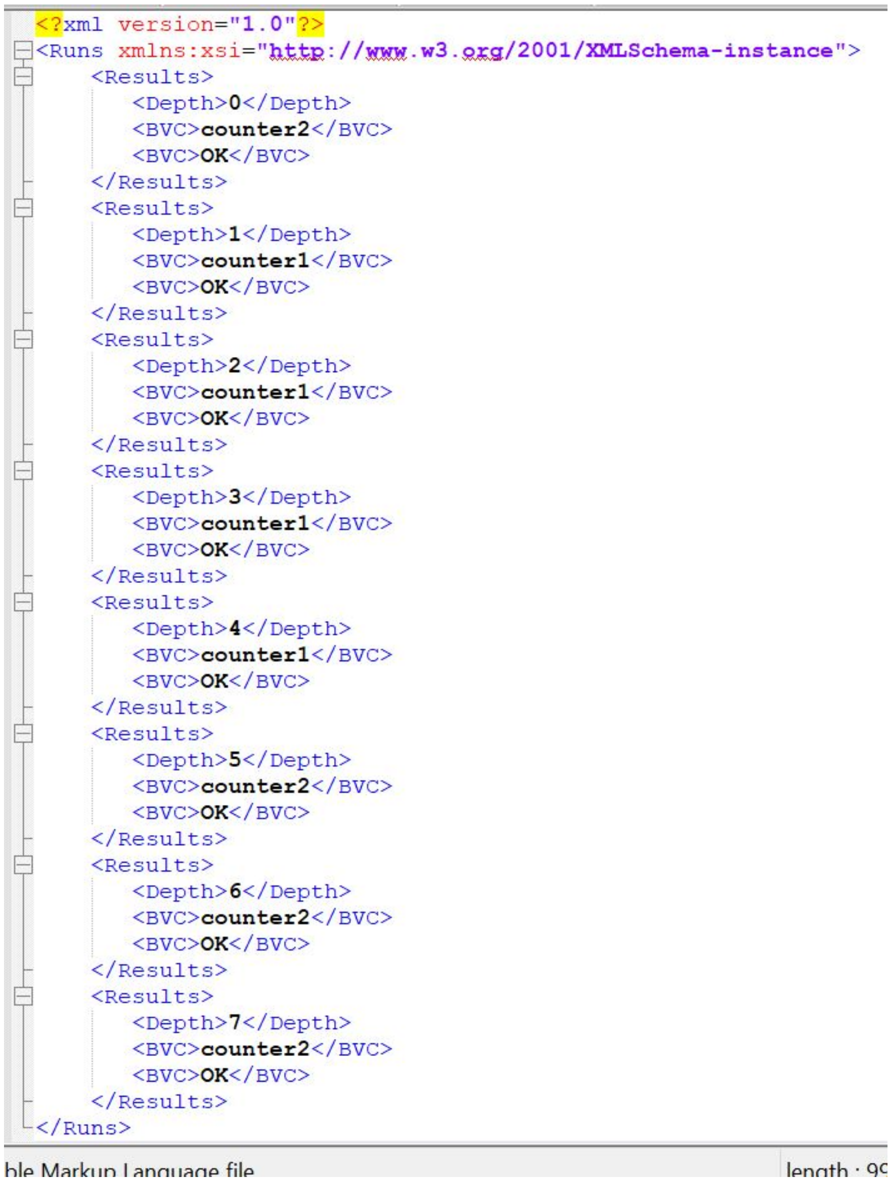
Figure 5.9: A toy example that shows multiple BVCs at earlier depths for a property with single MIVC

if we had just increased the depth by 1, from BVC_{10} on, the BVC would have become the same as the MIVC. For this model, up to depth 10, we have two ways of satisfying the property (we have two bounded validity cores {`V19_late`, `V64_incr`, `V63_diff`, `V65_PC`, `OK`} and {`V20_early`, `V64_incr`, `V63_diff`, `V65_PC`, `OK`}), but after depth 10, the property is satisfied with only one validity core ({`V19_late`, `V64_incr`, `V63_diff`, `V65_PC`, `OK`}). Figure 5.12 shows the output of the `JKind` BVC engine over this model up to depth 15.

RQ3

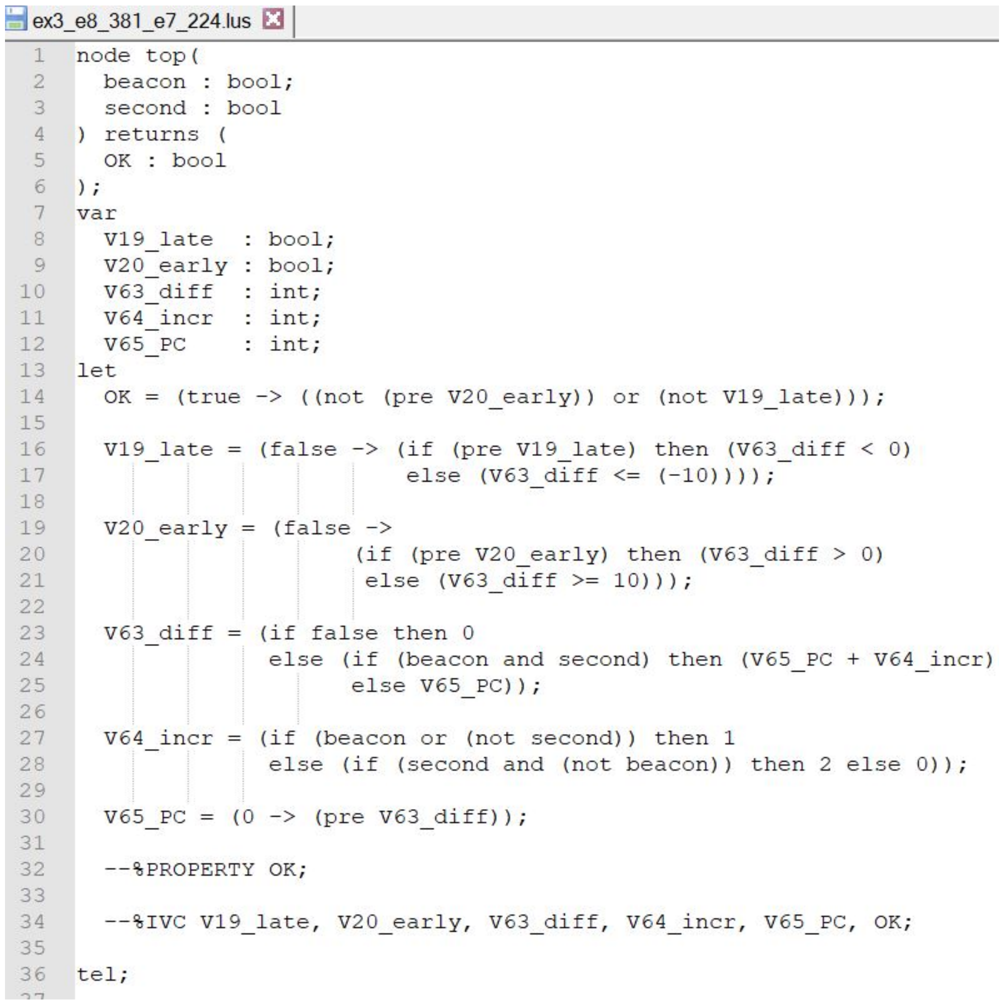
In order to show how quickly BVCs change and converge to an actual MIVC, we chose BVC_0, BVC_3, and BVC_{max} runs and plot the size of the cores. Mostly for models with less than 200 design elements, size of BVCs did not change much from depth 3 to 9. For the larger models there is some difference between the size of BVC_3 and BVC_{max} (Figure 5.13).

We calculated the difference of BVC_{max} of each model with its MIVCs. Part of the results is described in **RQ1**. If BVC_{max} is the subset of one of the MIVCs, we calculated the difference between those two, and if not, BVC_{max} is compared with one



```
<?xml version="1.0"?>
<Runs xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Results>
        <Depth>0</Depth>
        <BVC>counter2</BVC>
        <BVC>OK</BVC>
    </Results>
    <Results>
        <Depth>1</Depth>
        <BVC>counter1</BVC>
        <BVC>OK</BVC>
    </Results>
    <Results>
        <Depth>2</Depth>
        <BVC>counter1</BVC>
        <BVC>OK</BVC>
    </Results>
    <Results>
        <Depth>3</Depth>
        <BVC>counter1</BVC>
        <BVC>OK</BVC>
    </Results>
    <Results>
        <Depth>4</Depth>
        <BVC>counter1</BVC>
        <BVC>OK</BVC>
    </Results>
    <Results>
        <Depth>5</Depth>
        <BVC>counter2</BVC>
        <BVC>OK</BVC>
    </Results>
    <Results>
        <Depth>6</Depth>
        <BVC>counter2</BVC>
        <BVC>OK</BVC>
    </Results>
    <Results>
        <Depth>7</Depth>
        <BVC>counter2</BVC>
        <BVC>OK</BVC>
    </Results>
</Runs>
```

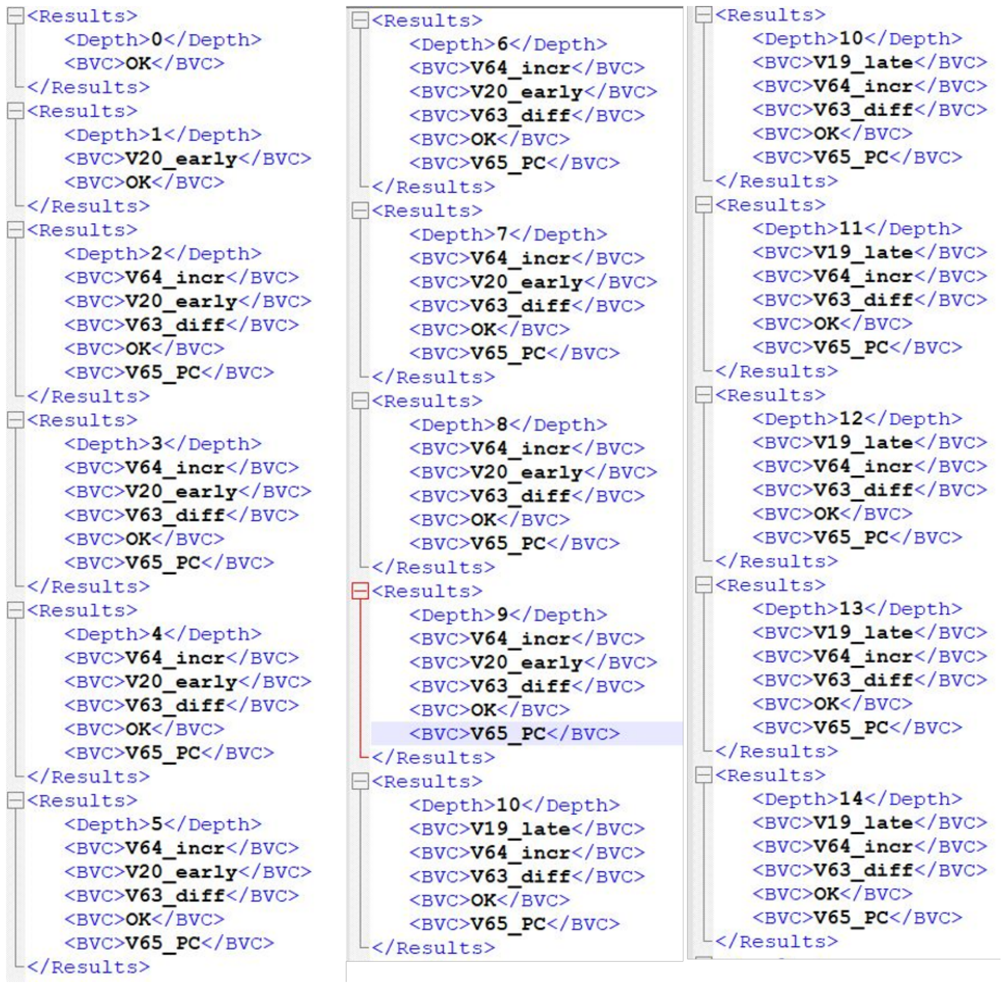
Figure 5.10: BVCs for the property in Figure 5.9



ex3_e8_381_e7_224.lus

```
node top(
  beacon : bool;
  second : bool
) returns (
  OK : bool
);
var
  V19_late  : bool;
  V20_early : bool;
  V63_diff  : int;
  V64_incr  : int;
  V65_PC    : int;
let
  OK = (true -> ((not (pre V20_early)) or (not V19_late)));

  V19_late = (false -> (if (pre V19_late) then (V63_diff < 0)
                        else (V63_diff <= (-10))));

  V20_early = (false ->
                    (if (pre V20_early) then (V63_diff > 0)
                     else (V63_diff >= 10)));

  V63_diff = (if false then 0
              else (if (beacon and second) then (V65_PC + V64_incr)
                    else V65_PC));

  V64_incr = (if (beacon or (not second)) then 1
              else (if (second and (not beacon)) then 2 else 0));

  V65_PC = (0 -> (pre V63_diff));

  --%PROPERTY OK;

  --%IVC V19_late, V20_early, V63_diff, V64_incr, V65_PC, OK;

tel;
```

Figure 5.11: Model ex3_e8_381_e7_224 as a case study



```
<Results>
    <Depth>0</Depth>
    <BVC>OK</BVC>
</Results>
<Results>
    <Depth>1</Depth>
    <BVC>V20_early</BVC>
    <BVC>OK</BVC>
</Results>
<Results>
    <Depth>2</Depth>
    <BVC>V64_incr</BVC>
    <BVC>V20_early</BVC>
    <BVC>V63_diff</BVC>
    <BVC>OK</BVC>
    <BVC>V65_PC</BVC>
</Results>
<Results>
    <Depth>3</Depth>
    <BVC>V64_incr</BVC>
    <BVC>V20_early</BVC>
    <BVC>V63_diff</BVC>
    <BVC>OK</BVC>
    <BVC>V65_PC</BVC>
</Results>
<Results>
    <Depth>4</Depth>
    <BVC>V64_incr</BVC>
    <BVC>V20_early</BVC>
    <BVC>V63_diff</BVC>
    <BVC>OK</BVC>
    <BVC>V65_PC</BVC>
</Results>
<Results>
    <Depth>5</Depth>
    <BVC>V64_incr</BVC>
    <BVC>V20_early</BVC>
    <BVC>V63_diff</BVC>
    <BVC>OK</BVC>
    <BVC>V65_PC</BVC>
</Results>
```

```
<Results>
    <Depth>6</Depth>
    <BVC>V64_incr</BVC>
    <BVC>V20_early</BVC>
    <BVC>V63_diff</BVC>
    <BVC>OK</BVC>
    <BVC>V65_PC</BVC>
</Results>
<Results>
    <Depth>7</Depth>
    <BVC>V64_incr</BVC>
    <BVC>V20_early</BVC>
    <BVC>V63_diff</BVC>
    <BVC>OK</BVC>
    <BVC>V65_PC</BVC>
</Results>
<Results>
    <Depth>8</Depth>
    <BVC>V64_incr</BVC>
    <BVC>V20_early</BVC>
    <BVC>V63_diff</BVC>
    <BVC>OK</BVC>
    <BVC>V65_PC</BVC>
</Results>
<Results>
    <Depth>9</Depth>
    <BVC>V64_incr</BVC>
    <BVC>V20_early</BVC>
    <BVC>V63_diff</BVC>
    <BVC>OK</BVC>
    <BVC>V65_PC</BVC>
</Results>
<Results>
    <Depth>10</Depth>
    <BVC>V19_late</BVC>
    <BVC>V64_incr</BVC>
    <BVC>V63_diff</BVC>
    <BVC>OK</BVC>
    <BVC>V65_PC</BVC>
</Results>
```

```
<Results>
    <Depth>10</Depth>
    <BVC>V19_late</BVC>
    <BVC>V64_incr</BVC>
    <BVC>V63_diff</BVC>
    <BVC>OK</BVC>
    <BVC>V65_PC</BVC>
</Results>
<Results>
    <Depth>11</Depth>
    <BVC>V19_late</BVC>
    <BVC>V64_incr</BVC>
    <BVC>V63_diff</BVC>
    <BVC>OK</BVC>
    <BVC>V65_PC</BVC>
</Results>
<Results>
    <Depth>12</Depth>
    <BVC>V19_late</BVC>
    <BVC>V64_incr</BVC>
    <BVC>V63_diff</BVC>
    <BVC>OK</BVC>
    <BVC>V65_PC</BVC>
</Results>
<Results>
    <Depth>13</Depth>
    <BVC>V19_late</BVC>
    <BVC>V64_incr</BVC>
    <BVC>V63_diff</BVC>
    <BVC>OK</BVC>
    <BVC>V65_PC</BVC>
</Results>
<Results>
    <Depth>14</Depth>
    <BVC>V19_late</BVC>
    <BVC>V64_incr</BVC>
    <BVC>V63_diff</BVC>
    <BVC>OK</BVC>
    <BVC>V65_PC</BVC>
</Results>
```

Figure 5.12: BVC runs for model ex3_e8_381_e7_224

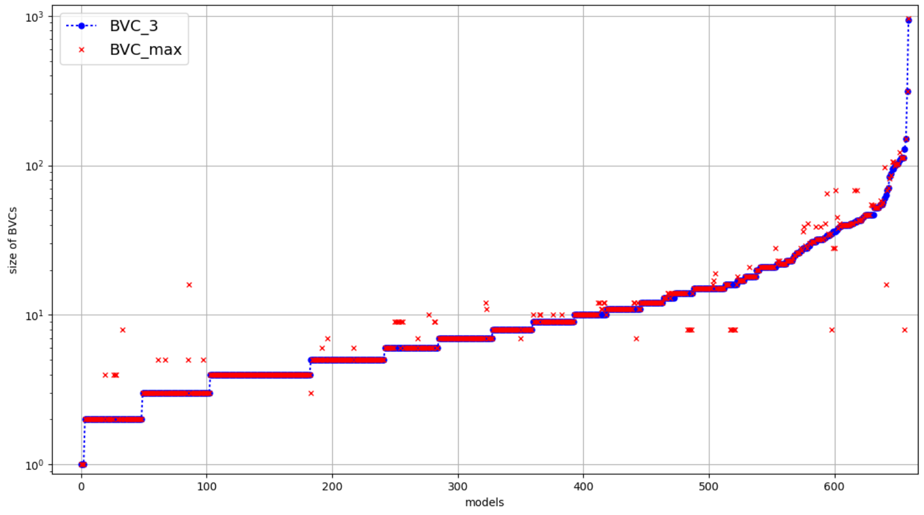


Figure 5.13: Size of BVCs at depth 3 and max

of the MIVCs of the model, selected randomly. Note that it is possible for BVC_{max} to be a subset of more than one of the MIVCs. In our calculation, we randomly selected the first MIVC containing BVC_{max}. Figure 5.14 shows the size of the models versus the size differences of MIVCs and BVCs in logarithmic scale.

Discussion

Experimental results show that in many cases, bounded validity cores can be as accurate as actual minimal IVCs. The abnormal behavior in some cases showed that we cannot make a strong claim about the relationships between BVCs and IVCs. One observation is that at deeper bounds, we can have more accurate bounded validity cores. The more accurate bounded cores are, the more useful information we have to evaluate completeness and adequacy of proofs.

It may be possible to make use of other techniques to evaluate the accuracy of the bounded cores. For example, in case of non-increasing BVCs, we may build different abstractions for the model using different BVCs, and try to prove the property over abstracted models. If property fails over one abstraction, we can easily rule out the bounded core used for that abstraction. For example, in Figure 5.9, if we terminate the

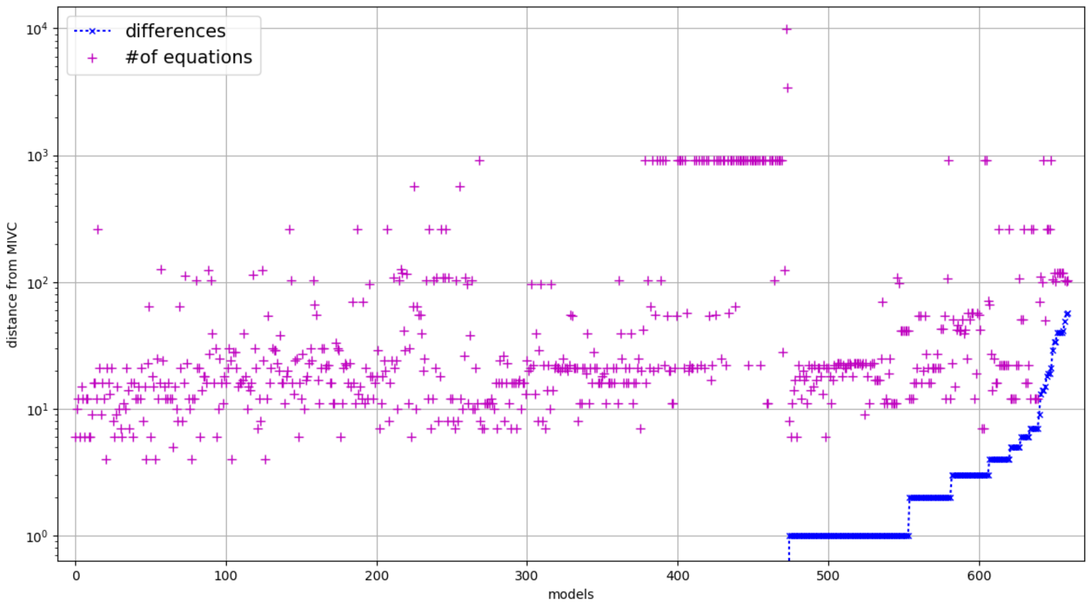
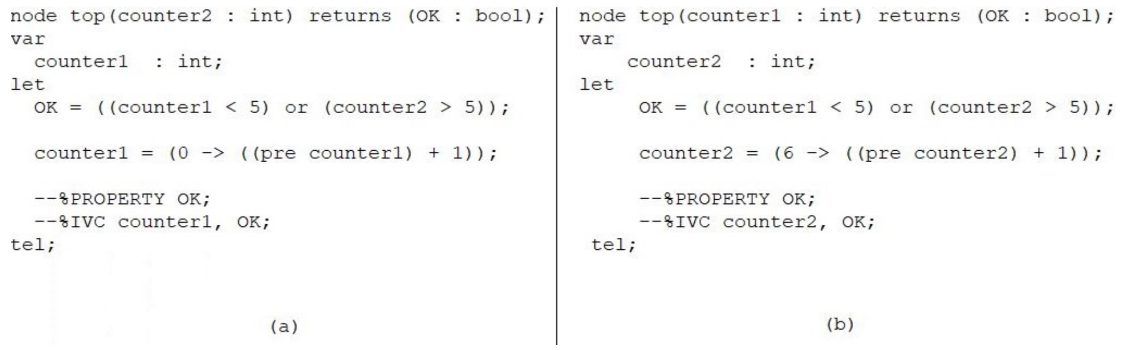


Figure 5.14: Difference between BVC_{max} and MIVCs vs the model size

BVC generation at depth 4, we need to decide which of {`counter1, OK`} and {`counter2, OK`} is necessary for an inductive proof (i.e., is the subset of our MIVC). In order to do so, if we build a new abstract model using BVC {`counter1, OK`} (Figure 5.15 (a)), and try to re-prove the property, property will be violated, which tells us `counter2` was necessary for the proof of our property. In the same way, if we build an abstract model using BVC {`counter2, OK`} (Figure 5.15 (b)) and try to re-prove the property, we will see the property is still valid, which means `counter1` is not in MIVC of the property.

Another observation is that if we calculate *all* BVCs at a given depth for a valid property, there should be at least one BVC, which is the subset of one of the MIVCs. As we advance towards deeper bounds when BVCs stay the same, one may even conclude that BVC has already converged to an actual minimal IVC. However, this conclusion is not always accurate. It is possible that final MIVC is a superset of multiple BVCs at different bounds. Consider the toy example in Figure 5.9. If the initial value of `counter2` changes from 6 to 3, the MIVC of the property will be {`counter1, counter2, OK`} because the correctness of `OK` is dependent on both counters. However, if we obtain the BVCs for this new model (Figure 5.16), you will see that BVC up to depth 5 is



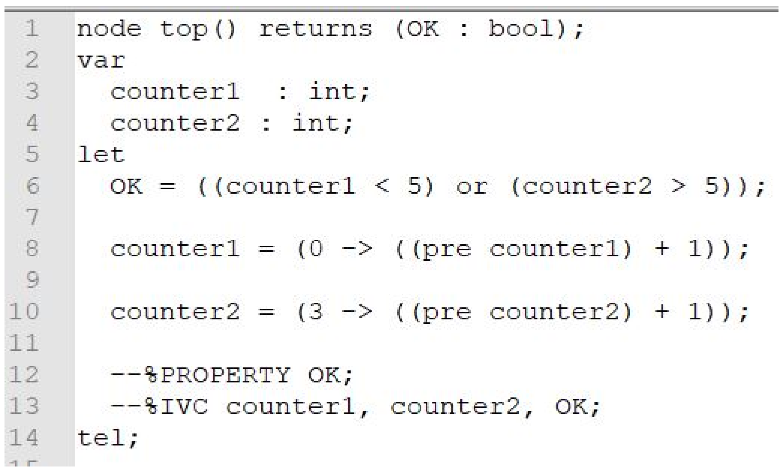
```
node top(counter2 : int) returns (OK : bool);
var
  counter1  : int;
let
  OK = ((counter1 < 5) or (counter2 > 5));

  counter1 = (0 -> ((pre counter1) + 1));

  --%PROPERTY OK;
  --%IVC counter1, OK;
tel;
```

(a)

```
node top(counter1 : int) returns (OK : bool);
var
    counter2  : int;
let
     OK = ((counter1 < 5) or (counter2 > 5));

     counter2 = (6 -> ((pre counter2) + 1));

     --%PROPERTY OK;
     --%IVC counter2, OK;
 tel;
```

(b)

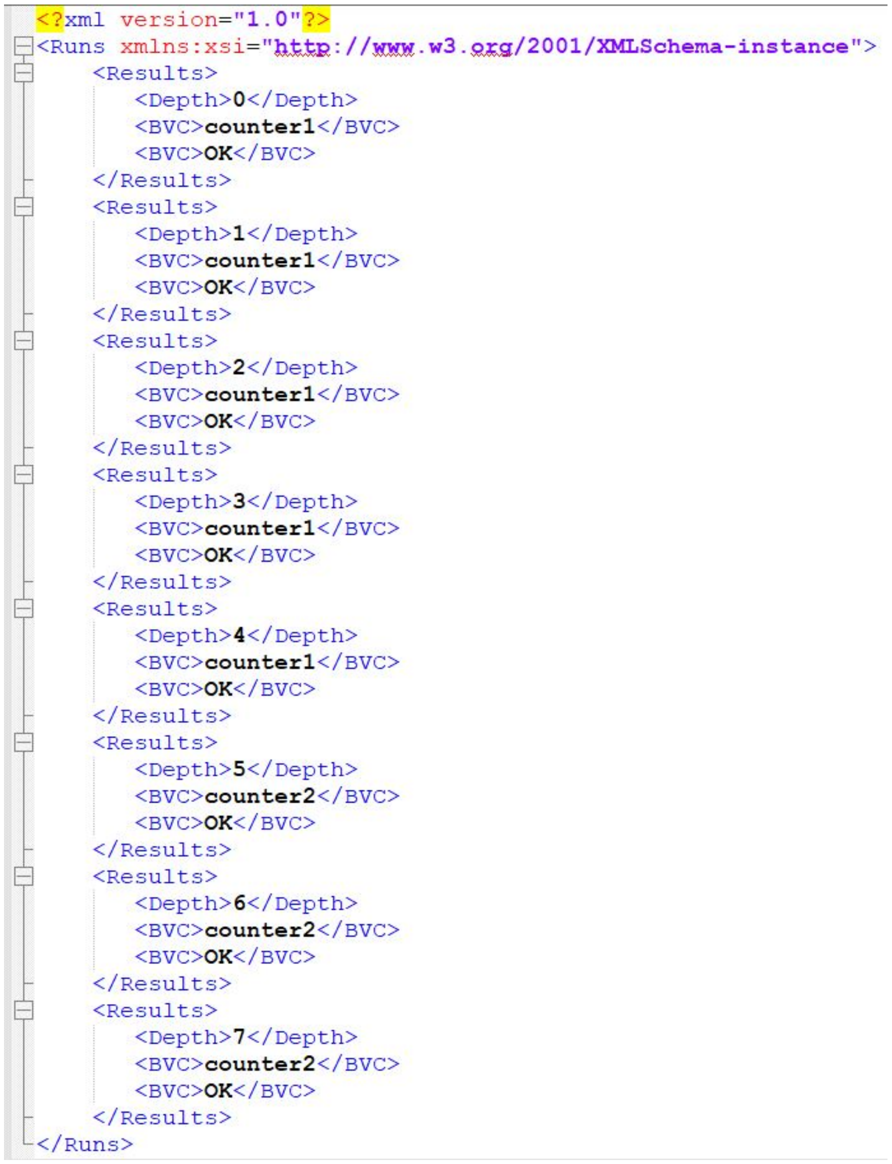
Figure 5.15: Building abstraction using BVCs



```
node top() returns (OK : bool);
var
  counter1  : int;
  counter2 : int;
let
  OK = ((counter1 < 5) or (counter2 > 5));

  counter1 = (0 -> ((pre counter1) + 1));

  counter2 = (3 -> ((pre counter2) + 1));

  --%PROPERTY OK;
  --%IVC counter1, counter2, OK;
tel;
```

Figure 5.16: A toy example where MIVC is the superset of multiple unique BVCs

{`counter1, OK`}, thereafter becomes {`counter2, OK`} and will not change (Figure 5.17).



```
<?xml version="1.0"?>
<Runs xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Results>
       <Depth>0</Depth>
       <BVC>counter1</BVC>
       <BVC>OK</BVC>
    </Results>
    <Results>
       <Depth>1</Depth>
       <BVC>counter1</BVC>
       <BVC>OK</BVC>
    </Results>
    <Results>
       <Depth>2</Depth>
       <BVC>counter1</BVC>
       <BVC>OK</BVC>
    </Results>
    <Results>
       <Depth>3</Depth>
       <BVC>counter1</BVC>
       <BVC>OK</BVC>
    </Results>
    <Results>
       <Depth>4</Depth>
       <BVC>counter1</BVC>
       <BVC>OK</BVC>
    </Results>
    <Results>
       <Depth>5</Depth>
       <BVC>counter2</BVC>
       <BVC>OK</BVC>
    </Results>
    <Results>
       <Depth>6</Depth>
       <BVC>counter2</BVC>
       <BVC>OK</BVC>
    </Results>
    <Results>
       <Depth>7</Depth>
       <BVC>counter2</BVC>
       <BVC>OK</BVC>
    </Results>
</Runs>
```

Figure 5.17: BVCs for the property in Figure 5.16

Chapter 6

Applications

As described in Section 1, IVCs can facilitate several engineering tasks in different phases of a system development process. This chapter explains some of the IVC applications we have explored.

6.1 Automatic Proof-Based Traceability

Requirements traceability can be defined as

> *"the ability to describe and follow the life of a requirement, in both forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)."* [60].

Traceability is concerned with establishing relationships, called *trace links*, between the requirements and one or more artifacts (design elements) of the system. Among the several different development artifacts and the relationships that be can established from/to the requirements, being able to establish trace links from requirements to artifacts that realize or *satisfy* those requirements—particularly to entities within those artifacts called *target artifacts* [61]—has been enormously useful in practice. For instance, it helps analyze the impact of changes in one artifact on the other, assess the quality of the system, aid in creating assurance arguments for the system, etc.

There is substantial interest within the Requirements Engineering research community towards automating the construction and maintenance of traceability links [19, 20, 123]. There are many kinds of trace links that may have to do with functional correctness, performance, architectural qualities, user understanding, and many other criteria. We focus our attention to this subset of requirement traceability called *Satisfaction Arguments* that are used to determine the portions of a design or model that are necessary to satisfy a requirement. IVCs automatically provide such arguments accurately and with no human effort. In addition, we can automatically generate expected artifact types, such as traceability matrices for these kind of relationships (see Figures 6.3 and 6.4).

It is also the case, when computing all IVCs, that we can provide additional insight. As far as we are aware, none of the existing Satisfaction Argument literature discusses the issue that there are often multiple satisfaction arguments between a requirement and its implementation. Given all IVCs, it is possible to perform more accurate impact analysis and define multiple notions of requirements adequacy, as we will see in the following sections.

6.2 Coverage Analysis and Requirements Completeness

For critical systems, it has been argued that formal methods should be applied to gain higher assurance than is possible with testing [5,124,125]. For these approaches, testing may still be performed, but the verification effort is primarily focused on performing proofs. Unfortunately, proof-based approaches tend not to answer the question as to whether implementations have *additional functionality* that is not covered by requirements. Testing, despite its faults, can measure *structural coverage* to find untested functionality and can find some errors by *serendipity*, in which problems not directly related to the requirement under test are exposed. Therefore, in formal verification approaches, it is even more important that requirements be complete.

In general, specification completeness can be defined with regard to the notion of coverage. In fact, the way that coverage is formalized plays a key part in the strength-/effectiveness of a method for the assessment of completeness. The goal of a coverage metric is usually to assign a numeric score that describes how well properties cover the design. Relatively recently, techniques have been devised for analyzing completeness of requirements against formal implementation models, specified as transition systems or Kripke structures [68, 74–76]. These models are agnostic to the abstraction level of the implementation: implementations can be lower-level requirements, software architectures, or concrete implementations of system behavior. The mechanism used is based on *mutation* and *proof*: is it possible to prove that the requirements still hold of the system after mutating the model in some way? If so, then the requirements are incomplete with respect to that model element. Mutations are "atomic" changes to the design, where the set of possible mutations depends on the notation that is used. A mutant is "killed" if one of the properties that is satisfied by the original design is violated by the mutated design [16, 17, 65, 68, 69]. There are many different kinds of mutations that have been proposed, primarily focused on checking sequential bit-level hardware designs. For these designs, *State-based* mutations flip the value of one of the bits in the state. There are several variations depending on whether the flip is performed on a single state within a Kripke structure [66], or in the description of the signal in the transition relation of the circuit [68]. *Logic-based* mutations fix the value of a bit to constant zero or one, and can be used to determine whether properties can find stuck-at faults. *Syntactic* mutations [16] remove states in a control flow graph representation of hardware. Similarly, for software, it is possible to apply any of the "standard" source code mutation operators used for software testing [70] towards requirements coverage analysis. Some examples of software mutations are [71]:

1. Replace an integer constant C by one of $\{0, 1, -1, C + 1, C - 1\}$,
2. Replace an arithmetic, relational, logical, bitwise logical, increment/decrement, or arithmetic-assignment operator by another operator from the same class,
3. Negate the decision in an `if` or `while` statement,
4. Delete a statement.

We assume each element $T_i \in T$ has a set of possible mutations associated with it. Depending on the modeling formalism used, this may be the value of a gate or signal or an expression within a statement in a program. We will further assume the existence of a mutation function f_m that, given a model element, will return a finite set of mutations for that element. We can then define the set of mutant models M as follows:

$$M = \{(T \setminus \{T_i\}) \cup \{m\} \mid T_i \in T, m \in f_m(T_i)\}$$

and then define the mutation score for property P in the standard way:

Definition 8. *Generalized mutation coverage.*

$$\textsc{Mutant-Cov} = \frac{|\{m \mid m \in M \ \wedge \ (I, m) \nvdash P\}|}{|M|}$$

Note that without loss of generality, we consider a single property P, which can be viewed as the conjunction of all the properties of the model.

The state of the art of mutation-based coverage can be found in Chockler *et al.* [69], where a design is considered as a net-list with nodes of types {`AND, INVERTER, REGISTER, INPUT`}. Each mutant design changes the type of a single node to `INPUT`. When property ϕ satisfied by the original net-list fails on the mutant design, it is said that a mutant is discovered for ϕ. Then, the coverage metric for ϕ is defined as the fraction of the discovered mutants, based on which the coverage of a set of properties is measured as the fraction of mutants discovered by at least one property. To decrease the cost of computation, coverage analysis is performed at several stages; first, all the nodes that do not appear in the resolution proof of a given property are marked as *not-covered*, and the rest of the nodes are marked as *unknown*. Then, for the unknown nodes, the basic mutation check is performed: if a corresponding mutant design violates the property, it will be considered as *covered*.

Unfortunately, previous completeness metrics can *underapproximate* which portions of a program are necessary to fulfill the requirements. That is, if we construct a model consisting of only the required model elements as determined by the analysis, it is often no longer possible to prove the requirement. Thus the feedback provided to the developer may be somewhat misleading. In addition, the mutation-based analyses tend to be very computationally expensive. For example, for model checkers, state of the art techniques have runtimes of (in the best case) several times more than is required for proof [69]. We propose a new approach for measuring property completeness based on proof rather than mutation.

Definition 9. *IVC coverage (*IVC-Cov*):*
Given $S \in AIVC(P)$, T_i is covered by P via S *iff* $T_i \in S$.

We call Definition 9 a *proof-preserving* metric because, with a set of the model elements marked as covered by IVC-Cov, P is provable. Other notions, as will be discussed, may yield subsets of the model that are insufficient to reconstruct the proof of the property. The coverage score for IVC-Cov can be calculated with:

$$\frac{|S|}{|T|}$$

Because P may have multiple *MIVC*s, IVC-Cov metric can lead to various scores that belong to the following set:

$$\{\ \frac{|S|}{|T|} \mid S \in AIVC(P)\ \}$$

Note that if an *MIVC* contains all model elements (i.e., the model is *completely covered*), then there is only one possible *MIVC*, so in this case there is no diversity of scores.

Using the notions of MAY and $MUST$, we can introduce additional coverage metrics.

Definition 10. *(*May-Cov*):* $T_i \in T$ is covered by P *iff* $T_i \in$ May-Cov(P), where May-Cov$(P) = \{T_i \mid \exists S \in AIVC(P)\ .\ T_i \in S\}$.

Definition 11. *(*Must-Cov*):* $T_i \in T$ is covered by P *iff* $T_i \in$ Must-Cov(P), where Must-Cov$(P) = \{T_i \mid \forall S \in AIVC(P)\ .\ T_i \in S\}$.

The May-Cov notion aims to deal with the fact that a property P may have several distinct *MIVC*s. In such cases, IVC-Cov only looks at an arbitrary *MIVC* that may contain a subset of $MAY(P)$, which means, depending on which *MIVC* it considers, every time it may report a different part of $MAY(P)$ as uncovered. However, May-Cov resolves this issue reporting the entire set of $MAY(P)$ as covered, which also leads to higher coverage scores. Must-Cov takes the opposite view, considering a model element as covered only if it affects all the proofs of P. Algorithm 15 is also an efficient way of computing the *must* set of a given property using `IVC_UC`. A different algorithm for

Algorithm 15: `IVC_MUST`: an algorithm to compute $MUST(P)$ for a given P

```
input  : $(I, T) \vdash P$
output: Must set for $(I, T) \vdash P$
1 $S \leftarrow$ IVC_UC($(I, T) \vdash P$)
2 $M \leftarrow \varnothing$
3 for $x \in S$ do
4     if $(I, T \setminus \{x\}) \nvdash P$ then
5         $M = M \cup \{x\}$
6 return $M$
```

computing $MUST(P)$ is to first compute $AIVC(P)$ and then take the intersection of all sets in $AIVC(P)$.

It is still possible to build more relaxed coverage metrics in which coverage is captured by looking at individual properties, rather than their conjunction. We can, for example, describe a metric in which any element used by an *MIVC* for any property is considered covered. The next definition, Model-Cov, formalizes this notion.

Definition 12. *(*Model-Cov*):* Given a set of properties Δ over T, $T_i \in T$ is covered *iff* $T_i \in$ Model-Cov(T), where Model-Cov$(T) = \{T_i \mid \exists P \in \Delta,\ S \in AIVC(P).\ T_i \in S\}$.

Based on the categorization of elements, we will state some relationships about *MIVC*s so to compare different proof-based metrics proposed earlier.

Lemma 5. If $MAY(P) \neq \varnothing$, then P is not provable by $MUST(P)$.

Proof. $MAY(P) \neq \varnothing \Rightarrow \exists T_i \in MAY(P).\ T_i \in \bigcup AIVC(P) \wedge T_i \notin MUST(P)$, which implies $\exists S \in AIVC(P).\ T_i \in S$. Considering the fact that S is minimal and $MUST(P) \subset S$ (since $T_i \in S \wedge T_i \notin MUST(P)$), $\nexists S' \subset S.\ (I, S') \vdash P$, which means $(I, MUST(P)) \nvdash P$. □

Now we focus on the relationship between non-deterministic mutation-based coverage and proof-based metrics. In Chockler et. al. [69], each mutant design changes the type of a single node to an input node . Given a suitable encoding of the netlist, assigning a "fresh" input is an isomorphic operation to simply removing a T_i from T. The mapping is as follows: the net-list becomes a conjunction of equations, where each vertex becomes a variable $v_i \in U$, and where each non-input vertex becomes an assignment equation $T_i \in T$. For example, given an AND-vertex v_i with three input edges from other vertexes $\{v_a, v_b, v_c\}$, we would define an equation $T_i \in T$ of the form $(v_i = (v_a \wedge v_b \wedge v_c))$.

Given this encoding, we can reframe the non-deterministic coverage proposed in [69] as follows:

Definition 13. *Nondeterministic coverage (alternate specification) (*NONDET-COV**) [69].* $T_i \in T$ is covered by property P *iff* $T_i \in$ NONDET-COV*(P), where NONDET-COV*$(P) = \{T_i \mid (I,T) \vdash P \wedge (I, T \setminus \{T_i\}) \nvdash P\}$.

Given this definition, it becomes straightforward to define some additional properties.

Lemma 6. $T_i \in$ NONDET-COV*$(P) \Leftrightarrow T_i \in$ MUST-COV(P).

Proof. $T_i \in$ NONDET-COV*(P) means that $(I, T \setminus \{T_i\}) \nvdash P$ then $\forall S \subset T.\ T_i \notin S \Rightarrow (I, S) \nvdash P$. Therefore, since $(I,T) \vdash P$, $T_i \in \bigcap AIVC(P)$, which means $T_i \in MUST(P)$. On the other hand, let $T_i \in MUST(P)$; then $\forall S \in AIVC(P).\ T_i \in S$. By definition, any proof of P is a superset of some minimal IVC in $AIVC(P)$. Thus, any subset S of T leading to proof contains T_i. Therefore, $T \setminus \{T_i\}$ does not lead to a proof.

□

In light of Lemma 6, the NONDET-COV* coverage score of specification P can be also calculated by

$$\frac{|MUST(P)|}{|T|}$$

Corollary 3. NONDET-COV* is not proof-preserving.

Proof. Immediate from Lemma 5 and Lemma 6 □

Corollary 4. IVC-COV is proof-preserving.

Proof. Immediate from Definition 3 and Definition 9 □

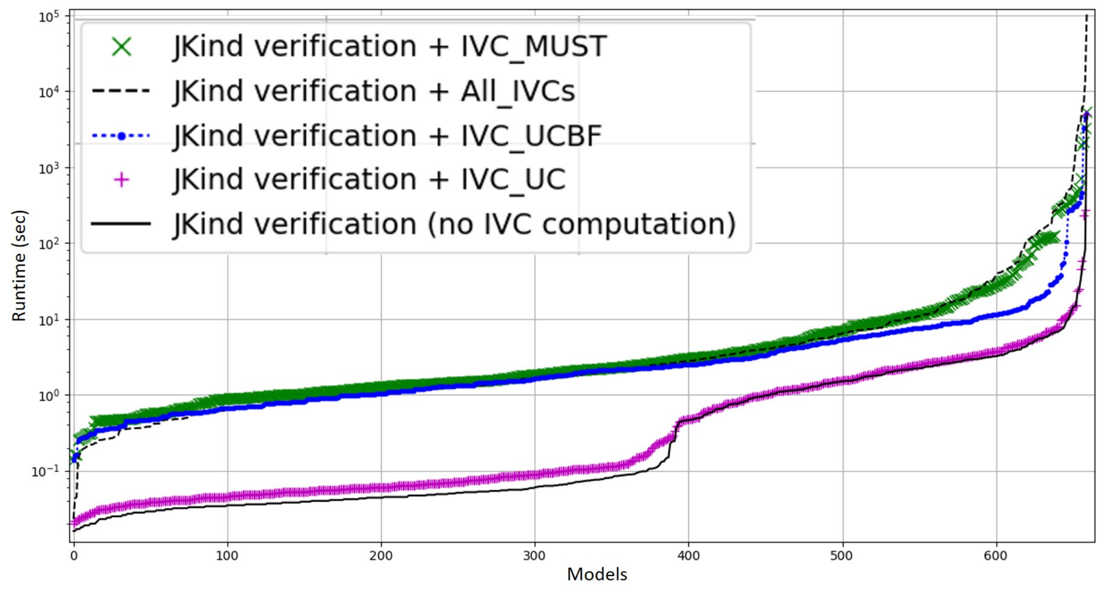


Figure 6.1: Runtime of different analyses

To conclude this section, we should mention that one can define many more proof-based coverage metrics based on the *MIVC*/$AIVC$ idea. Metrics that make use of the $AIVC$ relation are computationally more expensive to compute than IVC-Cov although they result in higher coverage scores.

Figure 6.1 allows a visualization of the runtime of different coverage analyses in comparison with the proof time[1], which indicates the overhead induced by each algorithm. As can be seen, it is computationally cheap to find an approximately minimal IVC using the algorithm `IVC_UC`; however, finding a *guaranteed* minimal IVC using the `IVC_UCBF` algorithm is computationally expensive. The overhead of the `IVC_UC` algorithm is on average 31% over the baseline proof, as opposed to 2276% for the `IVC_UCBF` algorithm. Therefore, in order to compute IVC-Cov, it is much more efficient to use `IVC_UC` rather than the `IVC_UCBF` algorithm. In terms of comparing cost of coverage computation from IVC-Cov and Must-Cov, the Must-Cov computation imposes an average 4183% runtime overhead on the verification time.

When a coverage metric brings about lower coverage scores on average, it is said

[1]Note that this graph is similar to the performance analysis in Chapter 5, except it introduces `IVC_MUST`

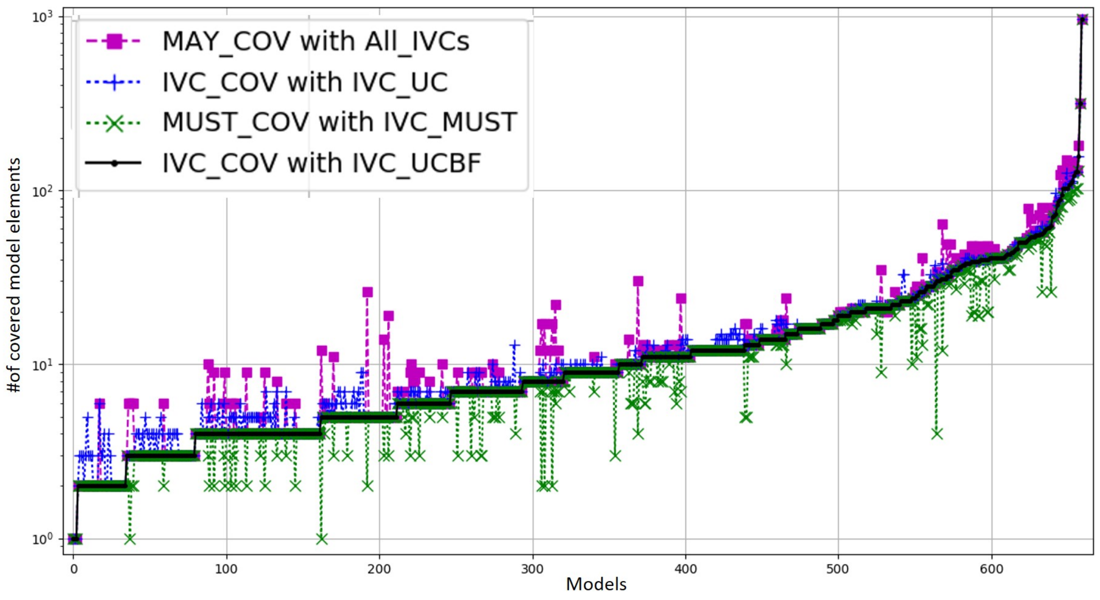


Figure 6.2: Size of the set of covered elements by different algorithms

that the metric is harder to satisfy. To study this aspect of the proposed metrics, we first calculated the size of the output sets generated by each algorithm: on average, the ratio of the size of the sets generated by `IVC_UC` to the size of the ones obtained from `IVC_UCBF` is 1.08, while this ratio for `IVC_MUST` to `IVC_UCBF` is 0.93, which shows `IVC_MUST` is harder to satisfy, and also is not proof-preserving.

Figure 6.2 is a visualization of the size of the set of covered elements by different algorithms. Models over the x-axis are sorted based on the size of the minimal IVCs obtained from the `IVC_UCBF` algorithm. The graph shows the degree of under-approximation of a minimal proof set by `IVC_MUST` as well as the degree of over-approximation by `IVC_UC`. Moreover, the size of sets computed by `IVC_UC` is very close to the size of the ones obtained from `IVC_UCBF`, especially for larger models. For our benchmark suite, in the industrial models, the size of sets obtained from `IVC_UCBF` and `IVC_UC` is more or less the same, which may indicate that the `IVC_UC` is likely to find MIVCs in realistic problems. The average increase in size of IVCs returned by `IVC_UC` is approximately 8% of the `IVC_UCBF` algorithm. Since the overhead of producing `IVC_UC` is only approximately 31% more expensive than the baseline analysis, this test may be efficient enough to run as a standard part of the model checking process.

Table 6.1: Coverage score of different algorithms

score	min	max	mean	stddev
IVC-Cov with IVC_UC	0.002	1.0	0.475	0.302
IVC-Cov with IVC_UCBF	0.002	1.0	0.445	0.291
Must-Cov	0.002	1.0	0.417	0.291
May-Cov	0.002	1.0	0.476	0.301

Table 6.1 describes the aggregate of the coverage scores returned by the analyses. Across all benchmarks, the min and max coverage scores are the same, and as expected, the average number of elements required is smallest for the IVC_MUST algorithm and largest the for IVC_UC algorithm.

The proposed coverage metrics can be ranked in terms of their scores as follows:

$$\text{Nondet-Cov}^{*} \leq \text{IVC-Cov} \leq \text{May-Cov} \leq \text{Model-Cov}$$

IVC-Cov and Nondet-Cov* are equivalent when all elements within the model are covered: if all model elements are MUST elements, then there can only be one *MIVC*, and this *MIVC* uses all of the model elements. The equivalence of Must-Cov and Nondet-Cov* allows us to compare our algorithms against state-of-the-art mutation based coverage.

To investigate the relationship between provability and different coverage notions, we were interested in the number of models in the benchmark for which IVC_MUST resulted in the sets not equal to an MIVC (i.e. models for which IVC_MUST did not preserve provability). Obviously properties are provable by 100% of the IVCs computed by IVC_UC (and IVC_UCBF). As for the IVC_MUST algorithm, the properties of 290 models in the benchmarks were not provable by the output of IVC_MUST. In practice, for larger models, Must-Cov is more likely not to maintain provability, and since more than half of the models are small, 43% may still not reveal the actual degree to which Must-Cov underapproximates the covered parts of a model. The notion of proof preservation is appealing because it allows a concrete demonstration to the user of the irrelevance of portions of the implementation. The IVC coverage notion also allows, in cases where there are multiple minimal satisfying sets, insight on multiple ways by which the model meets a requirement.

6.3 Discussion

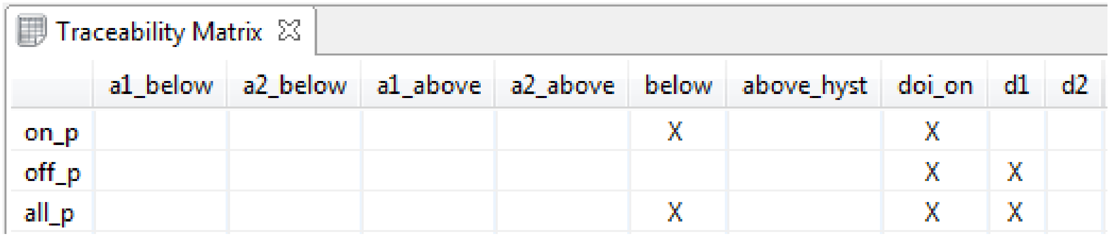
In this section we show how traceability and coverage analyses can be performed using IVCs. The goal of this section is to both illustrate the techniques and discuss the potential pitfalls of the analysis. Here we adapt the same ASW example presented in Section 3. The code is slightly changed so we can discuss how IVCs help to improve the design and specification.

We illustrate the results with traceability matrices produced by the `Spear` requirements specification tool [126]. `JKind` is used as the model checker for the `AADL AGREE` tool suite [32] and also `Spear` [126]. We have extended both tools to add graphical support for displaying adequacy and traceability results. We show screenshots for the `Spear` tool for our running example in Figures 6.3 and 6.4.

The ASW is responsible for turning on and off a device of interest, so we formulate two requirements that describe when the ASW should be *on* and when it should be *off*. The first attempt at formalization (property set 1) is as follows:

```
on_p = (a1_below and a2_below) and not inhibit =>
    doi_on = true;
off_p = (a1_above and a2_above) and inhibit =>
    doi_on = false;
all_p = on_p and off_p;
```

Informally, when both altimeters are below the threshold and not inhibited, then the DOI should be on (`on_p`), and when both altimeters are below the threshold and the ASW is inhibited, then the DOI should be off (`off_p`). For each of the IVC-Cov, May-Cov, and Must-Cov metrics, `all_p` only requires {`below, d1, doi_on`}, as shown in Figure 6.3. This small set of elements is due to a classic specification problem: using computed variables as the antecedents of implications. If these values are


Traceability Matrix

	a1_below	a2_below	a1_above	a2_above	below	above_hyst	doi_on	d1	d2
on_p					X		X		
off_p							X	X	
all_p					X		X	X	

Figure 6.3: Elements covered by the initial property set

computed incorrectly (say, we choose the wrong threshold for `a1_below`), it may cause the property to be valid for incorrect reasons.

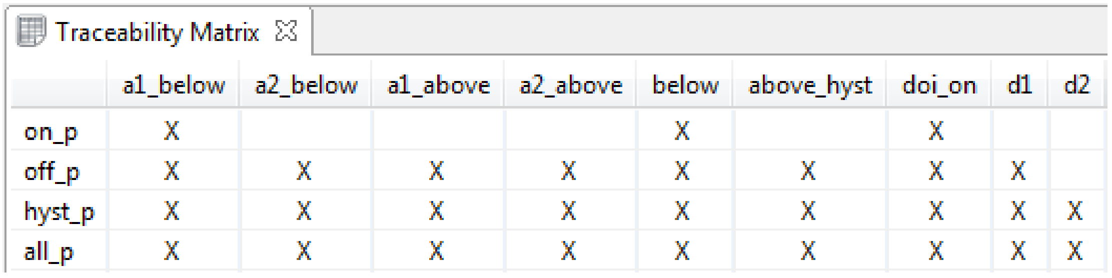
We therefore modify our properties to use inputs and constants as antecedents and derive:

```
on_p = ((alt1 < THRESHOLD) and (alt2 < THRESHOLD))
   and not inhibit => doi_on = true;
off_p = ((alt1 >= T_HYST) and (alt2 >= T_HYST))
   and inhibit => doi_on = false;
```

In this version, distinctions emerge between the metrics. `all_p` has two *MIVC*s: {{`a1_below, below, doi_on, d1`}, {`a2_below, below, doi_on, d1`}}, because of the `on_p` property: in the implementation, the DOI is turned on when either of the altimeters is below the threshold, while our property states that they both must be below. Domain experts determine that the requirement is correctly specified and that our implementation is a reasonable refinement, so there is no need to change the model or the property. The MUST elements are the same as version 1: {`below, doi_on, d1`}, because neither `a1_below` or `a2_below` is required for all proofs. The MAY elements contain both `a1_below` and `a2_below`.

The `above_hyst`, `a1_above`, `a2_above`, and `d2` equations are still missing, meaning that the "above" thresholds are irrelevant to our properties. Examining `off_p`, we realize that we have a specification error; the DOI should be off if either `inhibit` is true or both altimeters are above the threshold. The fix is:

```
off_p = ((alt1 >= T_HYST) and (alt2 >= T_HYST))
   or inhibit => doi_on = false;
```


Traceability Matrix

	a1_below	a2_below	a1_above	a2_above	below	above_hyst	doi_on	d1	d2
on_p	X				X		X		
off_p	X	X	X	X	X	X	X	X	
hyst_p	X	X	X	X	X	X	X	X	X
all_p	X	X	X	X	X	X	X	X	X

Figure 6.4: Elements covered by the final property set

Now the `all_p` requirement proof yields a single *MIVC* that requires all variables except {`d2`}, so *MIVC* = MAY = MUST. Interestingly, the `off_p` proof requires both the lower altimeter thresholds even though the `on_p` proof does not; the reason is that if either of these is false, then `doi_on` will be true. To cover {`d2`}, we realize no property covers the hysteresis case, so an additional property is added for this case:

```
hyst_p = not inhibit and
         (alt1 > THRESHOLD and alt2 > THRESHOLD) and
         (alt1 < T_HYST or alt2 < T_HYST) =>
   (doi_on = false -> doi_on = pre(doi_on))
all_p = on_p and off_p and hyst_p;
```

The final property states that if the antecedent conditions hold, then in the initial state, the `doi_on` variable is assigned false, and in subsequent steps, it retains the same value as it previously had.

As shown in Figure 6.4, the measures again coincide and include all variables, and we appear to have a reasonably complete specification. However, the measures are certainly not foolproof; it turns out that using *only* the hysteresis property `hyst_p` will *also* yield a "complete" result for all of the metrics: to establish its validity, all of the equations that we have defined in the model are required. This is because the partitioning of the transition system (i.e., the equations) is insufficiently *granular* to detect the incompleteness.

6.3.1 Granularity

As we have described in Section 2, a transition relation is considered to be a conjunction of Boolean formulas. The granularity of these formulas substantially affects the analysis results. In the presented example, it was possible to have a "complete" specification of the model involving only the hysteresis property `hyst_p`. The way that the model was structured, in order to determine the validity of the property, all of the equations in the model were required. However, for this property certain subexpressions of the equations were irrelevant, notably the value assigned to the `doi_on` variable in the `then` branches of equations (7) and (8). If we decompose the equations into smaller pieces, e.g., creating separate equations for the `then` and `else` branches, this incompleteness becomes visible and the model is no longer completely covered. It is often the case that splitting a model into more conjuncts, that is, making the model more *granular*, leads to lower coverage of the model.

We have explored granularity within the context of the Lustre language. Lustre provides a nice formalism for discussion because it is top-level conjunctive (as required by our IVC definition), equational, and *referentially transparent* [22]: the behavior of a Lustre program is defined by a system of equations, and any subexpression on the right side of an equation can be extracted and assigned to a fresh variable[2] which is substituted into the original equation without changing the meaning of a program. In this context, we can define a *granular refinement* as an extraction of a subexpression into a new equation assigning a fresh variable.

We call a Lustre model *totally decomposed* if (1) each computed (i.e., non-input) variable is used at most once in the right-hand side of an equation, and (2) each equation is either a single operator (which may be a ternary operator in the case of if-then-else) with leaves all variable expressions, or a constant expression, and (3) each model input is directly assigned to one or more fresh variables and is not used elsewhere in the model. In this case, each instance of a subexpression and each use of an input in the original model is assigned its own variable, so it is maximally factored. Intuitively, if the proofs require each of these elements, it means that there is a reasonably strong claim that the requirements are adequate.

If a set of requirements achieves 100% coverage of a totally decomposed model, then no granular refinement will achieve less than 100% coverage. This is straightforward to show in a proof sketch. Based on the right side of the equation, there are two cases: (1) the entire right-hand side of the equation is extracted (which may be a constant

[2]A fresh variable is a variable with an identifier that has not been used within the program.

or a single operator expression). In this case, by assumption the variable assigned is necessary for proof, so its definition must be necessary for the proof; in this case, the fresh variable must also be necessary for the proof; (2) a leaf expression of the single right-hand side operator is extracted. Since (by definition) the leaf is a variable expression that is used only once, and (by assumption) the variable is necessary for proof, then the fresh variable is also necessary for proof. The resultant model is also a totally-decomposed model, so we can do any number of these extractions.

We have implemented a transformation that splits Lustre models into *totally decomposed* models. In a small initial experiment involving 30 of the original models, we performed our transformation and re-ran the analysis. By changing the granularity of the model, the analysis tools perform significantly slower for proofs, but the ratio of performance between the proof and the `IVC_UC` and `IVC_MUST` algorithms is largely unchanged. However, on some models, the `IVC_MUST` algorithm becomes unacceptably slow (analysis times of tens of hours) and occasionally causes the solver to run out of memory.

The issue of granularity of models is significant, but to the best of our knowledge, is not discussed in previous work. In our approach, we allow the user to choose the level of granularity, but in certain cases, this may lead to misleading answers when checking the adequacy of requirements. This aspect will be a focus of our future work, especially in situations in which the tool determines that a set of requirements is *complete*. We believe that it is possible to substantially optimize the näive preprocessing algorithm that we have presented.

Chapter 7

Conclusions and Future Work

In this thesis, we have introduced the notion of validity cores as an effective means of explaining the proof of a property. We have discussed the applications and uses of this idea. This notion has been formalized, and an efficient technique of extracting a minimal IVC has been provided.

However, a single IVC often does not provide a complete picture of the traceability from a property to a model. We have addressed the problem of extracting *all minimal* IVCs. Obtaining minimal IVCs for a given property is completely relevant to the problem of proving the correctness of that property. Since, in general, provability is undecidable, sometime engineers have to rely on bounded proofs. In order to utilize the IVC ideas in such cases, we have introduced the idea of bounded validity cores (BVCs). We have shown the correctness and completeness of our methods and algorithms.

We have implemented all of the inductive validity core algorithms in the `JKind` [21] industrial model checker, which verifies safety properties of infinite-state synchronous systems. It accepts Lustre programs [22] as input. The translation of Lustre into a symbolic transition system in `JKind` is straightforward and is similar to what is described in [119]. Verification is supported by multiple "proof engines" that execute in parallel, including k-induction, property directed reachability (PDR), and lemma generation engines that attempt to prove multiple properties in parallel. To implement the engines, JKind emits SMT problems using the theories of linear integer and real arithmetic. `JKind` supports the `Z3`, `Yices`, `MathSAT`, `SMTInterpol`, and `CVC4` SMT solvers as back-ends. We extend `JKind` with new engines that implement our IVC generation algorithms. When a property is proved and IVC generation is enabled, an additional parallel engine executes one of the IVC algorithms.

In addition, we have performed a substantial evaluation that shows that the practicality and efficiency of our technique. For this purpose, we have collected a large set of benchmarks from different sources. Our experiments are conducted on a set of benchmarks containing 660 Lustre models, 530 from [59, 119] and 130 industrial models derived from [3] and other sources [59, 121]. This evaluation shows promising results about the IVC techniques.

7.1 Future Research Directions

Our method for computing all MIVCs was inspired by recent work in the domain of satisfiability analysis [55]. One interesting future direction is to devise similar MIVC enumeration algorithms based on other studies on MUSes extraction such as [54]. Another interesting direction for this project is to parallelize the enumeration process: it is certainly possible to ask for multiple distinct maximal models to be solved in parallel. A straight forward parallelization starts with computing one approximate IVC. Then `IVC_MUST` can be executed on one machine while several other engines are running the `All_IVCs` in parallel. The results of `IVC_MUST` shall be dispatched among the engines to expedite the process of exploring the map[1].

It is worthwhile to investigate additional applications of the idea. When performing *compositional verification*, the All-IVCs technique may be able to determine *minimal component sets* within an architecture that can satisfy a given set of requirements, which may be helpful for design-space exploration and synthesis. In addition the approach can be used for robustness analysis; will system satisfy requirement R even if a certain component fails? To answer this question, the All-IVCs analysis can tell us if the requirement is satisfied by different components independently.

An study of the relationship between IVCs of the component-level and system-level properties itself is very interesting and can be useful; for example, in cases that the system property has only bounded proof, we can estimate/obtain its MIVCs without a proof. To do so, we need to map out an algorithm for extracting MIVCs of a system

[1]see 3.3.1

level property from the MIVCs of the component properties.

The granularity issue discussed in 6.3 opens up a new research direction. At this time, we rely on the model structure (granularity level) provided by the designer. There is a notion of "sufficient granularity" that guarantees that if the properties reference all model elements, then no other decomposition will say that it is incomplete.

There is more to be studied over the bounded validity cores. The way we tackled this problem in this thesis was through the conventional bounded model checking technique. With BMC, the state space exploration starts with a set of initial states, and expands over unrolling the transition relation. However, in some other model checking techniques, such as PDR, state space exploration starts with the universal set (*true*), and shrinks over each blocking step. It would be interesting to look at the relationship between bounded cores derived from a bounded-PDR vs BMC. This may help to estimate more accurate BVCs when it comes to unprovable properties. When we have limited resources (time and computing power) to prove a certain property, estimating BVCs may help us to build an abstract model, from which we can prove our property of interest.

The notion of bounded validity core can be applied into testing, where we deal with too many execution paths due to program complexities including loops. It would be interesting to perform coverage analysis with BVCs at the program level, and then compare the results with other existing mutation-based techniques for testing. For example, this can be done for *C* programs, for which there are powerful code model checkers such as `CBMC`. Finally, we are interested in adapting the notion of "all" validity cores for *bounded* model checking.

Investigating the IVC notions in model based development would be another research direction. Model-based development starts with a formal model as an executable specification refined and verified throughout development. In the late stage of the development when it comes to implementation, usually the model is automatically translated into code. For example, `AADL AGREE` tool suite is a powerful tool for this purpose. It starts with AADL modeling language, converted to LUSTRE for verification by `JKind`. Then *C* code is automatically generated from LUSTRE. We already implemented the IVC techniques for LUSTRE and `JKind`. One could implement these techniques in some *C* model checker, and then comparing IVCs at the code and model level might reveal valuable information that could be useful in safety analysis and future research.

Having all MIVCs can be useful in optimizing logic synthesis by discovering a minimum set of design elements (optimal implementation) for a certain behavior. This shall lead to an optimal design implementation after transforming an abstract behavior. Putting this idea into practice would be worthwhile future work.

Our method for extracting approximate MIVCs (`IVC_UC`) is fast and efficient. However there is room for improvement. In the current approach, after obtaining the proof, in order to extract a minimal set of lemmas, we re-prove the property with the k-induction technique, then a minimum k is calculated to build IVC queries (in a k-induction manner). A different approach could be to employ other proof methods such as interpolation or PDR.

Another future study would be to use the IVC approches for fault injection. Fault injection allows to analyze the system behavior in the presence of faults through the use of verification tools. We can investigate the use of IVCs in fault models in order to find the maximum set of faults for which a system is resilient.

References

[1] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 125–134, Austin, TX, 2011. FMCAD Inc.

[2] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, pages 108–125, 2000.

[3] Anitha Murugesan et al. Compositional verification of a medical device system. In *HILT*, 2013.

[4] Ken L. McMillan. A methodology for hardware verification using compositional model checking. Technical Report 1999-01, Cadence Berkeley Labs, Berkeley, CA 94704, 1999.

[5] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.

[6] M. Whalen, G. Gay, D. You, M.P.E. Heimdahl, and M. Staats. Observable modified condition/decision coverage. In *Proceedings of the 2013 Int'l Conf. on Software Engineering*. ACM, May 2013.

[7] D. You, S. Rayadurgam, M.W. Whalen, and M.P.E. Heimdahl. Efficient observability-based test generation by dynamic symbolic execution. In *26th International Symposium on Software Reliability Engineering (ISSRE 2015)*, November 2015.

[8] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *STTT*, 2003.

[9] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *6th International Conference on Theory and Applications of Satisfiability Testing: SAT 2003*, May 2003.

[10] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *Proceedings of the 15th International Symposium on Formal Methods*, FM '08, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] RTCA DO-178C. *Software Considerations in Airborne Systems and Equipment Certification*, 2011.

[12] *Requirements for Safety Related Software in Defence Equipment, Issue 2.* UK Ministry of Defence, 1997.

[13] MathWorks Inc. Simulink Requirements Traceability. http://www.mathworks.com/discovery/requirements-traceability.html, 2016.

[14] Ed Keenan, Adam Czauderna, Greg Leach, Jane Cleland-Huang, Yonghee Shin, Evan Moritz, Malcom Gethers, Denys Poshyvanyk, Jonathan Maletic, Jane Huffman Hayes, Alex Dekhtyar, Daria Manukian, Shervin Hossein, and Derek Hearn. Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1375–1378, Piscataway, NJ, USA, 2012. IEEE Press.

[15] MathWorks Inc. Simulink Design Verifier. http://www.mathworks.com/products/sldesignverifier, 2015.

[16] H. Chockler, O. Kupferman, and M. Vardi. Coverage metrics for formal verification. *Correct hardware design and verification methods*, pages 111–125, 2003.

[17] O. Kupferman, W. Li, and S.A. Seshia. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *Proceedings of the 2008 Int'l Conf. on Formal Methods in Computer-Aided Design*, page 25, 2008.

[18] Center of Excellence for Software Traceability. http://www.coest.org, 2016.

[19] Jane Huffman Hayes, Alex Dekhtyar, and James Osborne. Improving requirements tracing via information retrieval. In *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*, pages 138–147. IEEE, 2003.

[20] Jane Cleland-Huang, Brian Berenbach, Stephen Clark, Raffaella Settimi, and Eli Romanova. Best practices for automated traceability. *Computer*, (6):27–35, 2007.

[21] JKind. `http://loonwerks.com/tools/jkind.html`.

[22] N. Halbwachs et al. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 1991.

[23] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. *SAT*, 7317:157–171, 2012.

[24] Yakir Vizel and Arie Gurfinkel. Interpolating property directed reachability. In *International Conference on Computer Aided Verification*, pages 260–276. Springer, 2014.

[25] Dejan Jovanović and Bruno Dutertre. Property-directed k-induction. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, pages 85–92. FMCAD Inc, 2016.

[26] Arie Gurfinkel and Alexander Ivrii. K-induction without unrolling.

[27] CBMC Bounded Model Checker for C and C++. `http://www.cprover.org/cbmc/`.

[28] Alloy: Bounded model checking for program designs. `http://alloy.mit.edu/alloy/index.html`.

[29] Jbmc: A bounded model checking tool for verifying java bytecode. `https://www.cprover.org/jbmc/`.

[30] Viktor Schuppan and Armin Biere. Liveness checking as safety checking for infinite state spaces. *Electron. Notes Theor. Comput. Sci.*, 149(1):79–96, February 2006.

[31] Koen Claessen and Niklas Sörensson. A liveness checking algorithm that counts. In *Fmcad*, pages 52–59, 2012.

[32] Darren D. Cofer, Andrew Gacek, Steven P. Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. Compositional verification of architectural models. In Alwyn E. Goodloe and Suzette Person, editors, *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)*, volume 7226, pages 126–140, Berlin, Heidelberg, April 2012. Springer-Verlag.

[33] Adrien Champion et al. The Kind 2 model checker. In *CAV*, 2016.

[34] OCRA Tool. `https://ocra.fbk.eu/`.

[35] Z.E. Hanna, P.A.M. Franzen, R.M. Weber, H.A. Farah, and R.K. Ranjan. Formal verification coverage metrics for circuit design properties, May 14 2015. US Patent App. 14/474,280.

[36] Cadence JasperGold Formal Verification Platform. `https://www.cadence.com/`.

[37] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, Ann Arbor, MI, USA, 1979. AAI8007856.

[38] Kelly Androutsopoulos, David Clark, Mark Harman, Jens Krinke, and Laurence Tratt. State-based model slicing: A survey. *ACM Comput. Surv.*, 45(4):53:1–53:36, August 2013.

[39] Ranjit Jhala and Rupak Majumdar. Path slicing. *SIGPLAN Not.*, 40(6):38–47, June 2005.

[40] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh Ranganath, Robby, and Todd Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 73–89, Berlin, Heidelberg, 2006. Springer-Verlag.

[41] Mary Jean Harrold. Software analysis and testing: lecture notes on program slicing. In *Georgia Tech*, 2009.

[42] F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.

[43] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[44] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256, June 1990.

[45] Hon F. Li, Juergen Rilling, and Dhrubajyoti Goswami. Granularity-driven dynamic predicate slicing algorithms for message passing systems. *Automated Software Engineering*, 11(1):63–89, 2004.

[46] Alessandro Cimatti et al. A simple and flexible way of computing small unsatisfiable cores in sat modulo theories. In *SAT'07*.

[47] Anton Belov, Mikoláš Janota, Inês Lynce, and Joao Marques-Silva. On computing minimal equivalent subformulas. In *Principles and Practice of Constraint Programming*, pages 158–174. Springer, 2012.

[48] Alexander Ivrii, Arie Gurfinkel, and Anton Belov. Small inductive safe invariants. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, 2014*, pages 115–122, October 2014.

[49] Mark H Liffiton et al. From MaxSAT to MinUNSAT: Insights and applications. *Ann Arbor*, 2005.

[50] Fahiem Bacchus and George Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *CAV'15*, 2015.

[51] Anton Belov and Joao Marques-Silva. Muser2: An efficient mus extractor. *JSAT journal*, 2012.

[52] Anton Belov et al. Core minimization in sat-based abstraction. In *DATE'13*, 2013.

[53] Anton Belov *et al.* Towards efficient MUS extraction. *AI Communications*, 2012.

[54] Alexander Nadel et al. Accelerated deletion-based extraction of minimal unsatisfiable cores. *JSAT journal*, 2014.

[55] Mark Liffiton et al. Fast, flexible MUS enumeration. *Constraints*, 2016.

[56] Jaroslav Bendík, Nikola Benes, Ivana Cerná, and Jirí Barnat. Tunable Online MUS/MSS Enumeration. In Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen, editors, *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016)*, volume 65 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 50:1–50:13, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[57] Aarti Gupta, Malay Ganai, Zijiang Yang, and Pranav Ashar. Iterative abstraction using sat-based bmc with proof analysis. In *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 416. IEEE Computer Society, 2003.

[58] Kenneth L McMillan and Nina Amla. Automatic abstraction without counterexamples. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 2–17. Springer, 2003.

[59] Alain Mebsout and Cesare Tinelli. Proof certificates for smt-based model checkers for infinite-state systems. In *FMCAD'16*, 2016.

[60] O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 94–101, Apr 1994.

[61] Orlena Gotel, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grünbacher, Alex Dekhtyar, Giuliano Antoniol, Jonathan Maletic, and Patrick Mäder. Traceability fundamentals. In *Software and Systems Traceability*, pages 3–22. Springer, 2012.

[62] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM transactions on Software Engineering & Methodology*, 1997.

[63] Anitha Murugesan, Michael W Whalen, Neha Rungta, Oksana Tkachuk, Suzette Person, Mats PE Heimdahl, and Dongjiang You. Are we there yet? Determining the adequacy of formalized requirements and test suites. In *NASA Formal Methods*, pages 279–294. Springer, 2015.

[64] D. Schuler and A. Zeller. Assessing oracle quality with checked coverage. In *Proceedings of the Fourth IEEE Int'l Conf. on Software Testing, Verification and Validation*, pages 90–99, 2011.

[65] Orna Kupferman. Sanity checks in formal verification. In *Proceedings of the 17th International Conference on Concurrency Theory*, CONCUR'06, pages 37–51, Berlin, Heidelberg, 2006. Springer-Verlag.

[66] Yatin Hoskote, Timothy Kam, Pei-Hsin Ho, and Xudong Zhao. Coverage estimation for symbolic model checking. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 300–305. IEEE, 1999.

[67] Sagi Katz, Orna Grumberg, and Danny Geist. have i written enough properties?-a method of comparison between specification and implementation. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 280–297. Springer, 1999.

[68] Hana Chockler, Orna Kupferman, Robert P Kurshan, and Moshe Y Vardi. A practical approach to coverage in model checking. In *International Conference on Computer Aided Verification*, pages 66–78. Springer, 2001.

[69] Hana Chockler, Daniel Kroening, and Mitra Purandare. Coverage in interpolation-based model checking. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 182–187. IEEE, 2010.

[70] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608 –624, aug. 2006.

[71] Timothy Alan Budd. *Mutation Analysis of Program Test Data*. PhD thesis, New Haven, CT, USA, 1980. AAI8025191.

[72] Kenneth L McMillan. Interpolation and sat-based model checking. In *International Conference on Computer Aided Verification*, pages 1–13. Springer, 2003.

[73] Synopsys VC Formal Platform. `https://www.synopsys.com`.

[74] Sayanlan Das, Ansuman Banerjee, Prasenjit Basu, Pallab Dasgupta, PP Chakrabarti, Chunduri Rama Mohan, and Limor Fix. Formal methods for analyzing the completeness of an assertion suite against a high-level fault model. In *18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*, pages 201–206. IEEE, 2005.

[75] Koen Claessen. A coverage analysis for safety property lists. In *Formal Methods in Computer Aided Design, 2007. FMCAD'07*, pages 139–145. IEEE, 2007.

[76] Daniel Große, Ulrich Kühne, and Rolf Drechsler. Estimating functional coverage in bounded model checking. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1176–1181. EDA Consortium, 2007.

[77] Finn Haedicke, Daniel Große, and Rolf Drechsler. A guiding coverage metric for formal verification. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 617–622. IEEE, 2012.

[78] Didar Zowghi and Vincenzo Gervasi. On the interplay between consistency, completeness, and correctness in requirements evolution. *Information and Software Technology*, 45(14):993 – 1009, 2003.

[79] Rolf Drechsler, Melanie Diepenbeck, Daniel Große, Ulrich Kühne, Hoang M Le, Julia Seiter, Mathias Soeken, and Robert Wille. Completeness-driven development. In *International Conference on Graph Transformation*, pages 38–50. Springer, 2012.

[80] Donald Firesmith. Are your requirements complete? *Journal of Object Technology*, 4(1):27–44, 2005.

[81] Ray-Yaung Chang, Andy Podgurski, and Jiong Yang. Finding what's not there: a new approach to revealing neglected conditions in software. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 163–173. ACM, 2007.

[82] V Katta, C Raspotnig, and P Karpati. Investigating requirements models completeness in a unified process for safety and security'. In *Proceedings of the International Conference on Cybersecurity of SCADA and Industrial Control Systems*, 2013.

[83] Sergio Espana, Nelly Condori-Fernandez, Arturo Gonzalez, and Óscar Pastor. Evaluating the completeness and granularity of functional requirements specifications: a controlled experiment. In *2009 17th IEEE International Requirements Engineering Conference*, pages 161–170. IEEE, 2009.

[84] Arie Gurfinkel and Marsha Chechik. Robust vacuity for branching temporal logic. *ACM Trans. Comput. Logic*, 13(1):1:1–1:32, January 2012.

[85] Hana Chockler and Ofer Strichman. Before and after vacuity. *Formal Methods in System Design*, 34(1):37–58, 2008.

[86] Shoham Ben-David and Orna Kupferman. A framework for ranking vacuity results. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, pages 148–162, 2013.

[87] Hana Chockler and Ofer Strichman. Easier and more informative vacuity checks. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE '07, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society.

[88] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. *Computer Aided Verification: 9th International Conference, CAV'97 Haifa, Israel, June 22–25, 1997 Proceedings*, chapter Efficient detection of vacuity in ACTL formulas, pages 279–290. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[89] MoD Interim Defence Standard. Standard 00-56 issue 4-safety management requirements for defence systems. *Ministry of Defence*, 2007.

[90] ISO26262 ISO. 26262: Road vehicles-functional safety. *International Standard ISO/FDIS*, 26262, 2011.

[91] Mats Heimdahl et al. Deviation analysis via model checking. In *ASE'02*, 2002.

[92] Anitha Murugesan et al. Complete traceability for requirements in satisfaction arguments. In *RE'16 (RE@Next! Track)*, 2016.

[93] Elaheh Ghassabani et al. Efficient generation of inductive validity cores for safety properties. In *FSE'16*, 2016.

[94] Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler sat instances. In *CP'11*, 2011.

[95] Antonio Morgado et al. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 2013.

[96] Michael Whalen Jaroslav Bendk, Elaheh Ghassabani and Ivana ern. Online enumeration of all minimal inductive validity cores. In *16th International Conference on Software Engineering and Formal Methods (SEFM'18)*, 2018.

[97] Mark H. Liffiton, Alessandro Previti, Ammar Malik, and Joao Marques-Silva. Fast, flexible mus enumeration. *Constraints*, 21(2):223–250, 2016.

[98] M. Whalen L. Wagner E. Ghassabani A. Gacek, J. Backes. The JKind model checker. In *CAV2018: Computer Aided Verification*, 2018.

[99] N. Halbwachs et al. The synchronous dataflow programming language lustre. *IEEE*, 1991.

[100] Herbert Rocha et al. Model checking embedded c software using k-induction and invariants. In *SBESC*. 2015.

[101] Dirk Beyer and Matthias Dangl. Smt-based software model checking: An experimental comparison of four algorithms. In *VSTTE*, 2016.

[102] Dirk Beyer et al. Boosting k-induction with continuously-refined invariants. In *CAV*, 2015.

[103] Martin Brain et al. Safety verification and refutation by k-invariants and k-induction. In *SAS*, 2015.

[104] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In *SPIN'12*, 2012.

[105] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS'08*. 2008.

[106] Bruno Dutertre and Leonardo De Moura. The YICES SMT solver. Technical report, SRI, 2006.

[107] Alessandro Cimatti et al. The MathSAT5 SMT solver. In *TACAS'13*, 2013.

[108] Clark Barrett et al. CVC4. In *CAV'11*, 2011.

[109] Temesghen Kahsai and Cesare Tinelli. Pkind: A parallel k-induction based model checker. In *PDMC*, 2011.

[110] Temesghen Kahsai et al. Incremental verification with mode variable invariants in state machines. In *NFM*, 2012.

[111] Niklas Eén et al. Efficient implementation of property directed reachability. In *FMCAD*, 2011.

[112] Alessandro Cimatti et al. IC3 modulo theories via implicit predicate abstraction. In *TACAS*, 2014.

[113] AIVC in JKind. `https://github.com/elaghs/jkind/tree/IVCs`.

[114] Darren Cofer et al. Compositional verification of architectural models. In *NFM*, 2012.

[115] John Backes et al. Requirements analysis of a quad-redundant flight control system. In *NFM*, 2015.

[116] Aaron W. Fifarek et al. Spear v2.0: Formalized past LTL specification and analysis of requirements. In *NFM*, 2017.

[117] Michael W. Whalen et al. Coverage metrics for requirements-based testing. In *ISSTA*, 2006.

[118] Lucas Wagner et al. SIMPAL: A compositional reasoning framework for imperative programs. In *SPIN*, 2017.

[119] G. Hagen and C. Tinelli. Scaling up the formal verification of lustre programs with smt-based techniques. In *FMCAD'08*, 2008.

[120] All IVCs repository. `https://github.com/elaghs/Working/tree/master/all_ivcs/experiments`.

[121] John Backes et al. Requirements analysis of a quad-redundant flight control system. In *NFM 2015*, 2015.

[122] F. Gaucher. Slicing lustre programs. Technical report, VERIMAG, Grenoble, February 2003.

[123] Alexander Egyed and Paul Grunbacher. Automating requirements traceability: Beyond the record & replay paradigm. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 163–171. IEEE, 2002.

[124] John Rushby. Software verification and system assurance. In Dang Van Hung and Padmanabhan Krishnan, editors, *Seventh International Conference on Software Engineering and Formal Methods (SEFM)*, pages 3–10, Hanoi, Vietnam, November 2009. IEEE Computer Society.

[125] David Hardin, D. Randolph Johnson, Lucas Wagner, and Michael W. Whalen. Development of security software: A high-assurance methodology. In *Proceedings of the 11th Int'l Conf. of Formal Engineering Methods (ICFEM 2009)*. Springer Verlag, December 2009.

[126] Erika Hoffman Benjamin Rodes M. Anthony Aiello Jennifer Davis Aaron Fifarek, Lucas Wagner. Spear v2.0: Formalized past ltl specification and analysis of requirements. In *Proceedings of 9th International NASA Formal Methods Symposium*. Springer, 2017.

List of Publications

[1] Elaheh Ghassabani, Michael Whalen, Andrew Gacek, and Mats Heimdahl. Inductive validity cores. *IEEE Transactions on Software Engineering (TSE).*

[2] Michael Whalen Lucas Wagner Elaheh Ghassabani Andrew Gacek, John Backes. The JKind model checker. In *CAV2018: Computer Aided Verification*, 2018.

[3] Jaroslav Bendík, Elaheh Ghassabani, Michael W. Whalen, and Ivana Cerná. Online enumeration of all minimal inductive validity cores. In *SEFM2018: Software Engineering and Formal Methods - 16th International Conference*, 2018.

[4] Elaheh Ghassabani, Andrew Gacek, Michael W Whalen, Mats Heimdahl, and Wagner Lucas. Proof-based coverage metrics for formal verification. In *ASE2017: 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[5] Elaheh Ghassabani, Andrew Gacek, and Michael W Whalen. Efficient generation of all minimal inductive validity cores. In *FMCAD2017: International Conference on Formal Methods in Computer-Aided Design*, 2017.

[6] Elaheh Ghassabani, Andrew Gacek, and Michael W Whalen. Efficient generation of inductive validity cores for safety properties. In *FSE2016: ACM Sigsoft International Symposium on the Foundations of Software Engineering*, pages 314–325. ACM, 2016.

[7] Michael Whalen, Anitha Murugesan, Elaheh Ghassabani, and Mats Heimdahl. Complete traceability for requirements in satisfaction arguments. In *24th International Requirements Engineering Conference (RE@Next! Track)*, pages 359–364. IEEE, 2016.