Tesis Doctoral

Análisis de ejecuciones parciales de Software Model Checkers

Castaño, Rodrigo

2018-03-21

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Análisis de ejecuciones parciales de Software Model Checkers

Tesis presentada para optar al título de
Doctor de la Universidad de Buenos Aires
en el área Ciencias de la Computación

Lic. Rodrigo Castaño

Director de tesis: Dr. Víctor A. Braberman
Consejero de estudios: Dr. Diego Garbervetsky

Buenos Aires, 2018
Fecha de defensa: 21 de marzo de 2018

ANÁLISIS DE EJECUCIONES PARCIALES DE SOFTWARE MODEL CHECKERS

Las herramientas de análisis estático de código han mostrado un progreso significativo en la última década. Una de las formulaciones del problema a resolver es el de descubrir si una propiedad es satisfecha necesariamente por un determinado sistema bajo análisis o si, por el contrario, existen ejecuciones de dicho sistema que violan la propiedad deseada. Dicha enunciación del problema es conocida como Software Model Checking y el mismo es indecidible en el caso caso general.

Habitualmente las herramientas de Software Model Checking buscan llegar a estar en condiciones de producir un resultado afirmativo, confirmando que la propiedad es satisfecha, o negativo, que habitualmente incluye un contraejemplo que viola la propiedad en cuestión. Sin embargo, en muchos casos, las herramientas se ven obligadas a producir un tercer resultado que indica que no se pudo demostrar la propiedad pero tampoco generar un contraejemplo. Además de los límites teóricos mencionados, en la práctica el problema resulta intratable para una gran cantidad de instancias de relevancia debido a que insume una elevada cantidad de recursos y de tiempo, incluso para casos en los que se alcanzaría finalmente una solución.

En estos casos la gran mayoría de las herramientas indican al usuario únicamente que el intento de verificación no alcanzó un resultado concluyente, sin ninguna aclaración adicional. Este trabajo se centra en proveer al usuario información adicional en dichos intentos de verificación. Con ese objetivo en mente, proponemos distintas formas de presentar e interpretar la información que se puede extraer en esos casos, teniendo en cuenta distintos posibles grados de familiaridad con las técnicas de verificación subyacente por parte del usuario.

En particular, nos centramos en una amplia familia de técnicas de verifi-

cación y presentamos varias vistas del progreso realizado por la herramienta previo a interrumpir el intento de verificación, acompañadas de su correspondiente caracterización formal. Adicionalmente, adaptamos la noción de cobertura, más frecuentemente utilizada en testing, a la familia de técnicas analizada. En ambos casos brindamos algoritmos que generan automáticamente tanto las vistas propuestas como sub-aproximaciones de la métrica de cobertura.

Las técnicas propuestas son evaluadas sobre instancias de referencias ampliamente utilizadas tanto para determinar la practicalidad en cuanto al tiempo de ejecución requerido como para analizar la interpretabilidad de los resultados generados.

La experimentación realizada confirma que es factible extraer información a partir de resultados inconcluyentes e interpretar dichos resultados revelando información no trivial.

Palabras claves: Model Checking, Análisis Automático de Programas, Cobertura, Verificación Formal.

ANALYSIS OF PARTIAL SOFTWARE MODEL CHECKING RESULTS

Static analysis tools have shown significant progress in the past decade. The problem these tools tackle can be formulated as deciding whether a property always holds for a specific system-under-verification or, on the contrary, certain execution in fact violates the desired safety property. Such problem definition is known as Software Model Checking and is, in the general case, undecidable.

Software Model Checkers usually attempt to either prove the property holds or, when it does not, to produce a counterexample that constitutes a violation. However, in many cases, tools produce neither and instead they are forced to produce a third kind of result, indicating the attempt failed to produce a conclusive result, i.e. the property could not be proved but a counterexample was not found either. Taking the undecidability results aside, in practice the problem remains intractable for a large number of industrial instances due to the immense resources required to solve them, even when a conclusive result would, at last, be produced.

In all of these cases most tools would indicate, without any further clarifications, that the resource limits were reached and the result was inconclusive. This work aims to provide users with additional information in these cases. With that goal, we propose a number of approaches to presenting and interpreting the information that can be extracted from an inconclusive verification attempt. Moreover, we take into account the different possible degrees of expertise that a user could have with the underlying verification techniques and attempt to make our output understandable to all users.

Concretely, we focus on a broad family of verification techniques and present a number of views of the progress achieved during verification. We provide, in each case, a formal characterization. Furthermore, we adapt the

notion of coverage, more commonly used in testing, to the family of verification techniques discussed. In both cases, we present algorithms to automatically compute both the views and the coverage metric proposed without requiring additional user input.

The ideas proposed are evaluated on standard benchmark instances, both to assess the practicality in terms of performance and also to determine the understandability of the results generated.

Our experiments confirm that it is possible to extract information from inconclusive verification results and gather non-trivial insights from the results.

Keywords: Model Checking, Program Analysis, Coverage, Formal Verification.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Verification can take many forms and is one of the many activities devoted to providing quality and safety assurances. The goal of verification is to prove that a certain system or component satisfies a property of interest.

Verification comprises a diverse set of techniques, including model checking [69], theorem proving [67], simulation [62] and symbolic execution [21], among other. In particular, software model checking [59] encompasses tools and techniques that combine some of the former and others and enable automated analysis of complex software systems.

Software model checkers have fundamental limitations since the problem is undecidable [42]. Moreover, the shortcomings are not only theoretical, these tools handle immense state spaces [82] in their internal representations, frequently leading to prohibitively large resource requirements or impractically long execution times.

Verification has been making significant progress in recent years and current software model checkers can tackle relevant industrial instances in specific domains and use cases, such as bug finding in Windows device drivers [3] or sound verification of avionics software [35]. Unfortunately, analyzing some systems can take hours of computation after which the execution could be interrupted due to a limitation of the underlying technique [12], a predefined exploration bound [19] or simply reaching a time or memory limit. In these cases, most current software model checkers inform only that the result was inconclusive, without providing neither hints as to what was analyzed nor providing concrete safety assurances.

Software testing, when the expected test results can be provided, is the most widespread verification technique in industry. Interestingly, testing lacks soundness guarantees but is still used to provide safety and quality assurances [85, 43, 72].

In this dissertation we discuss and extend the existing representations of inconclusive verification outcomes, with the goal of presenting them directly to users.

By studying and classifying existing work according to three different dimensions, we identify desired properties of future approaches. With those insights as guiding principles we propose different views of the work performed during an inconclusive verification attempt and formally characterize the properties of each of those views.

As a way to inspect what has been analyzed and whether any system executions were proved to satisfy the property of interest, we put forth Model Checker Execution Reports. Execution Reports present the user with sets of system executions that summarize the progress of the verification tool, e.g. a set of system executions which have been analyzed and proved to satisfy the property of interest or, conversely, a set of system executions that were not captured by the abstraction employed for verification. The insights obtained from execution reports usually lead to follow-up questions about the inconclusive results which we also address with a number of extensions.

The benefit of execution reports, the wealth of information they provide, can also be a shortcoming because it is only possible to take full advantage of these reports by performing a manual inspection. To overcome this shortcoming, we adapt the notion of coverage, as understood and broadly used in the context of testing, and redefine it for a family of verification techniques as a measure of progress.

Contributions

The contributions of this thesis can be summarized as follows:

- A survey of existing literature
- The definition of Execution Reports
- The definition of Coverage in the context of verification

- A parametric (with respect to the underlying verification technique) algorithm to compute Execution Reports
- An under-approximate algorithm to estimate Verification Coverage for a broad family of techniques
- Implementations of our algorithms on top of CPAchecker, an open source state-of-the-art Software Model Checker
- A qualitative evaluation of the understandability of Execution Reports by means of manually inspecting the output generated from inconclusive verification results of complex benchmark instances
- Extensions to improve the usability of execution reports
- A number of possible avenues for future research enabled by execution reports and its extensions

Road map

Chapter 2 comprises a survey of existing work. This chapter encompasses a framework with which to classify existing work. The analysis of the strengths and shortcomings of the related literature motivates the rest of the dissertation.

Chapter 3 introduces Model Checker Execution Reports, trace-based summary of the work performed by a software model checker during an inconclusive verification attempt. These summaries provide formal guarantees and are applicable to a large family of techniques. The chapter includes an evaluation of the practicality of the approach using standard benchmark instances.

Chapter 4 constitutes a follow-up of Chapter 3. The shortcomings of execution reports are revisited and a number of approaches are suggested to overcome the possible limitations. The techniques range from highly efficient post-processing techniques of the output to heavy weight analyses encoded as verification tasks. Due to the breadth of the approaches discussed, the depth

of this chapter is much more shallow. Finally, a number of promising lines for future research directly motivated from these extensions are discussed.

Chapter 5 presents a coverage metric for partial software model checks. The chapter includes a formal definition that aims to preserve the intuitive notion of coverage, as understood in testing, and an algorithm to compute under-approximations of the desired measure. To assess the performance of the algorithm as well as the coverage numbers achieved by different techniques, we also discuss the results of an empirical evaluation using standard benchmark instances.

Chapter 6 consists of a few closing remarks. In particular, we discuss the connections between our contributions and explain how they pave the way for future research in the general line of work that we pursued.

RESUMEN: INTRODUCCIÓN

La verificación es una entre varias tareas destinadas a fortalecer la calidad de un sistema y ofrecer garantías de seguridad. El objetivo de utilizar técnicas de verificación es el de demostrar que un sistema o componente satisface una propiedad de interés.

Existen diversas técnicas de verificación, incluyendo model checking, theorem proving, simulación y ejecución simbólica, entre otras. En particular software model checking incluye varias técnicas y herramientas que combinan varias técnicas de verificación para el análisis de sistemas de software de mayor complejidad.

Las herramientas de software model checking tienen limitaciones fundamentales debido a que el problema es indecidible. Lo que es más, las limitaciones no son únicamente teóricas sino que, siendo que estas herramientas habitualmente trabajan con espacios de estados enormes en sus representaciones internas, esto puede llevar a que tengan requerimientos de recursos prohibitivamente altos o que exhiban tiempos de ejecución excesivos.

El progreso realizado durante los últimos años en el área de verificación ha llevado a la disciplina a varios éxitos en aplicaciones industriales, incluyendo encontrar errores en drivers de Windows y permitir verificar software utilizado en aviación. Desafortunadamente, en muchos casos, analizar un sistema aún lleva horas de cómputo tras lo cual la ejecución puede interrumpirse por una multitud de razones, incluyendo límites de tiempo o recursos pre establecidos o limitaciones de las herramientas subyacentes. En estos casos, la mayoria de los software model checkers sólo informan que el resultado fue inconcluyente, sin proveer indicios respecto de qué fue analizado ni proveer garantías en relación al correcto funcionamiento del sistema.

Esta tesis se enfoca en discutir y extender representaciones de resultados de verificación inconcluyentes, a la vez que proponemos nuevas técnicas.

El resto de la tesis inicia con una reseña de trabajo relacionado que motivará el resto de nuestras contribuciones.

Más adelante presentaremos model checker execution reports, reportes basados en trazas, es decir secuencias de instrucciones, que capturan los comportamientos analizados durante un intento de verificación.

El siguiente capítulo incluye extensiones a la versión básica de los model checker execution reports. Los reportes producidos tienen algunas limitaciones en su versión básica. Discutimos estas limitaciones y proponemos una variedad de técnicas adicionales para superarlas. Adicionalmente sugerimos varias líneas de investigación que derivan de las técnicas propuestas.

Posteriormente, presentamos una métrica de cobertura para ejecuciones parciales de software model checkers. Este capítulo incluye una definición formal que apunta a preservar las nociones intuitivas de cobertura en el contexto de testing. Además, incluímos un algoritmo que permite computar sub-aproximaciones de la medida de cobertura propuesta.

Por último, en el capítulo final concluímos la tesis con una discusión conectando las distintas contribuciones y explicando cómo allanan el camino para extensiones de la línea de trabajo que hemos empujado.

2. LANDSCAPE OF PARTIAL VERIFICATION RESULTS

We will provide a structured overview of related work throughout this chapter. The core contributions of this chapter will be a broad, although possibly not exhaustive, survey of existing work and a conceptual framework for classifying work on this area under the light of attempting to assess the progress or shortcomings of an inconclusive outcome. We will conclude the chapter by discussing certain aspects worth improving that motivated us to pursue the line of research we develop throughout the rest of the dissertation.

The work we will discuss throughout this chapter will be related to inconclusive verification results and we will exclude approaches solely based on testing. Testing is inconclusive in nature and its literature has naturally developed under that premise. In contrast, other verification techniques are sound under certain assumptions. For those techniques, inconclusive outcomes pose interesting challenges and have been overlooked in many cases.

Analysis of Existing Work

Given the breadth of verification techniques, we will use three dimensions to classify each approach and we will start by briefly describing each.

First, and to structure our presentation, we will go over different possible types of output.

Secondly, we will classify existing output representations according to two other dimensions, breadth of its target audience and whether on-line monitoring is supported, that is, whether the output can be produced at arbitrary moments during the execution of the underlying verification technique.

When classifying by breadth of its audience, we will distinguish four possible degrees: marginal audience, technique experts, verification community and software development community.

The first degree will include many machine-readable representations which are not originally meant to be consumed by a human.

The second degree, encompassing the audience of experts in a particular technique, will comprise output that can be of use to an expert in the particular underlying verification technique but hardly so for someone with a background in verification but not specifically knowledgeable in that particular technique. One example of this would be exposing parts of the internal representation used by the verification tool at the moment the execution is interrupted.

The third degree, which includes the verification community at large, will contain representations which can be understood by, at least, a fraction of the community which is not necessarily familiar with the underlying verification tool, thus widening the underlying technique's possible audience. One such example would be the result of post-processing an internal representation such that it corresponds to more widely understood concepts.

Finally, the last degree corresponds to output that can be integrated into the software development process as it exists today. One such example is producing a mutation score.

The classification in these degrees will be highly subjective but provides a framework for analysis that can be adjusted to specific settings. In particular, each classification will be explained, such that it can be adjusted to specific settings where the applicability might increase or decrease.

Regarding how restrictive each approach is, we will classify each work according to whether the approach poses specific restrictions on the termination conditions leading to the inconclusive outcome. We make this distinction because some approaches to analyzing inconclusive verification results are only applicable to complete explorations of a predefined unsound state space. Among such techniques we can consider most variants of bounded model checking. Supporting a broader set of techniques and termination conditions, we believe, drastically simplifies the adoption for non-experts. Using bounded techniques as described typically involves acquiring in-depth a pri-

ori knowledge of how to specify tractable state space bounds for an instance. Targeting techniques that do not have these restrictions makes it possible to experiment without making such a significant upfront investment.

It is important to note that throughout the following discussion we will consider use-cases which some of these approaches were not originally designed for. As we anticipated, our main focus will be the use case of assessing the progress or shortcomings of an inconclusive verification outcome.

Types of output

Predicate-based

The idea of representing an inconclusive verification outcome using predicates has been proposed in a number of different ways.

One such example consists, precisely, on annotating the code using predicates over program variables at specific locations [30]. The annotations have been used to document [30] and analyze the impact [29] of unsound assumptions incorporated during verification. More specifically, the annotations comprise a single language extension, a variant of `assume` statements, widely used in verification. Another application of explicit annotations of assumptions [30] is in bounded abstract interpretation [32]. The idea is to record the abstract domain elements at each program location at the moment the analysis reaches a certain bound and is interrupted.

Generating preconditions constitutes a particular case of predicate-based output. To the best of our knowledge this has not been implemented but the idea of encoding inferred preconditions as partial verification results has been proposed [86] as a possible use case for the annotations of unsound assumptions [30].

A custom framework consisting of a combination of over-approximating and under-approximating analyses has been used to provide users with an over-approximation of the failing input space of a program [47].

Angelic verification [38] involves generating environments under which a

program is guaranteed to run safely. Angelic environments, expressed as predicates, need to satisfy a number of requirements but, since they are unsoundly inferred, they constitute part of the output, such that it can be inspected by users.

Quality-based

The ultimate goal when inspecting or evaluating an inconclusive verification outcome is, precisely, to assess its quality. Although an objective quality measure seems elusive, a number of proxies exist.

Mutation score is one of the measures normally used, mostly in testing, as a proxy for quality. Under a set of assumptions, the mutation score of a test suite seeks to approximate the effectiveness of such test suite in detecting faults that a programmer could normally introduce by mistake. The approach to compute such approximation is to create mutants by seeding artificial faults that aim to emulate common programming mistakes and counting how many of those mutants are detected by the test suite.

Although the adoption and study of mutation analysis in testing dates back decades, its adoption within the broader realm of verification, in particular for inconclusive verification attempts, is far more recent. One interesting application of mutations to verification is to guide incremental verification efforts. The approach, coined falsification-driven verification [50], comprises a number of tools and techniques around the basic methodology of creating a closed program using a bounded model of its environment and analyzing the closed program using bounded model checking. One key insight of this work is that it is possible to compute a mutation score by mutating the program-under-analysis and using the bounded model checker with the same bound on each of the mutants.

An entirely unrelated approach proposes to produce a correctness score as one of many ways of combining static and dynamic techniques [80] and displaying the results within a development environment. The score ranges

from -1 to 1 as evidence mounts either suggesting the presence of faults or the correctness of the system, respectively. Although the authors assign an equal score of zero to all inconclusive results from static techniques, the idea could possibly be extended preserving the output proposed.

The probability of a system exhibiting a failure can also be considered a measure of quality. An approach providing such probabilities exists for partial model checking attempts of probabilistic models [73]. That work [73] tackles Markov chains and requires careful modeling of the usage profile of the system. That is, the probabilities of failure are produced relative to a certain probability distribution that models the likelihood of certain operations being triggered or certain input values entered. Similarly, a probability measure can be computed for incomplete runs of symbolic executions [44]. This approach [44] also produces probabilities that are relative to a specific to a usage profile. Additionally, the authors [44] also introduce the concept of confidence. Informally, reliability and confidence dissociate the likelihood of reaching an error from the probability of reaching a condition still unexplored.

One way of defining a proxy of quality of a partial software model check is to measure the coverage with respect to a state space. Such a coverage estimation has been proposed [77] for explicit state model checkers.

Structure-based

Although the use of structural adequacy criteria in testing, such as statement or branch coverage, has been controversial, its ubiquitous adoption can hardly be ignored.

The use of structural coverage metrics is, however, not so widespread in other forms of verification. Existing work on using bounded model checking results as part of assurances cases [37] runs into, precisely, the issue of arguing that a certain bound manages to satisfy a certain structural coverage adequacy criterion. Curiously the approach is similar to the one used to perform mutation analysis [50] in that it relies on repeating the exploration by using

the same bound.

Trace-based

Listing the traces, that is, sequences of statements, that have been analyzed during a partial verification attempt is one possible way of conveying an inconclusive result. Although not explicitly the intention of the authors, the instantiation of conditional model checking [11, 12] exposes the concrete executions that have been analyzed. However, their output can actually contain spurious traces, that is, sequences of statements that do not correspond to actual system executions.

Other types of feedback

A few works considering inconclusive verification outcomes provide other types of feedback. One such example is the idea of an integrated environment for verification [27]. The most related contribution within that work is that of highlighting parts of the specification which are causing timeouts. That is, for example, a certain postcondition might be causing the underlying prover to reach the predefined time limit. This serves as a hint for the user to focus on that particular postcondition and the parts of the code that could be related to it.

An extension [41] to Alloy [56] has been proposed, that consists of highlighting the parts of the specification that are *hard* or *problematic*, in the authors' words. Alloy is a specification language and its corresponding solver, which translated high-level specifications into propositional logic using user-defined parameters as bounds. The language is particularly well-suited for structural properties of data structures. The extension works by exposing, in terms of the high-level specification, the most active variables and clauses at the propositional level.

Support for on-line monitoring

In order for an approach to be compatible with on-line monitoring, it should be compatible with arbitrary termination conditions. For example, if an analysis can only be applied after normal termination of the underlying verification technique, on-line monitoring would not be possible since generating the output would not be possible during the execution. In contrast, approaches that support arbitrary termination conditions enable on-line monitoring, since it is possible to interrupt the execution, which is equivalent to setting a time limit, and generate a derived representation at any time. As such, the discussion about support for on-line monitoring will be framed in terms of how restrictive is each technique with respect to the termination conditions supported for the underlying verification technique.

Some of the approaches discussed only tackle bounded model checking. In particular, generating a mutation score [50] or a code coverage metric [37] has only been discussed in the context of bounded model checking. We consider these two works quite restrictive in the termination conditions required because they only work when bounded model checking terminates successfully. That is, both approaches deal with exhaustively explored pre-defined bounds. As such, providing on-line information would not be possible.

The idea of highlighting which parts of a specification are taking longer than their alloted time limits to be verified [27] can be applied at any moment during execution.

Angelic verification [38] leads, in many cases, to inconclusive outcomes. The output in those cases are the angelic specifications. The specifications are produced only after normal termination, which falls on the restrictive end of the spectrum. The rationale for considering this restriction rather strong is that, once again, the properties of the inconclusive result are known in advance and only one termination condition is supported: when execution concludes normally.

Many other representations offer more flexible support for different ter-

mination conditions. The work discussed on language extensions to annotate unsound assumptions [30], for example, offers limited support since the use-case scenarios presented by the authors are those in which the unsound assumptions are known a priori. That is, the limitations of the inconclusive result are known before the verification attempt is performed. In contrast, the application of these annotations to bounded abstract interpretation [32] shows far superior support for arbitrary termination conditions. For example, this approach supports arbitrary interruptions and time limits, which make the inconclusive results substantially more diverse.

The implementation presented as a proof-of-concept of conditional model checking [12] also offers advanced support. In this case, the underlying tool [14] and techniques are based on abstract reachability trees. This work supports arbitrary interruptions, including time and resource bounds.

The works on bounding the probability of reaching an error condition [73, 44] both support stopping the underlying verification technique at any point. That is, these two approaches offer full support for arbitrary termination conditions.

The idea of generating preconditions from inconclusive verification outcomes [86], to the best of our knowledge, has not been implemented. A combination of bounded abstract interpretation [32] with a post-processing step transforming the output into preconditions [86] could potentially support arbitrary interruptions.

The framework for computing comprehensive failure characterizations [47] alternates over and under approximating analyses to produce its output. It seems possible to extract intermediate results before the alternation converges, but it is unclear how long, on average, each iteration takes. As such, defining the support for on-line monitoring in this case requires further study.

The existing implementation capable of producing a correctness score [80] defaults to a score of zero for all inconclusive results. As such, we consider this implementation only compatible with normal termination, lacking support for

on-line monitoring.

The Alloy extension [41] exposing the most active propositional variable and clauses can generate output at any point during execution.

A coverage estimation [77] for explicit state model checkers can be computed at any point during execution. As such, this output can support on-line monitoring.

Breadth of Audience

Some of the existing representations of inconclusive verification outcomes were designed to be consumed by another technique. Such is the case, for example, of the proof-of-concept implementation of conditional model checking [12], which generates machine-readable assumption automata. The size of these assumption automata is usually unmanageable for a human user and extracting any insights directly from the raw representation poses many challenges, as we will discuss in the next chapter.

Another representation with a severely limited audience, or at least applied in the scenarios discussed in its original presentation, is that of explicit annotations [30] of unsound assumptions. The audience for the annotations, as presented in their first paper [30], is marginal because it does not provide any particularly new insights with respect to the usual documentation of the underlying techniques. For example, verification systems which model integers as unbounded always make that unsound assumption, adding the annotation will not reveal any new information to the user without further analysis. It is important to emphasize that, at this point, we are only considering the audience for the specific use-case of inspecting the output, which is only briefly mentioned in the original paper. We can, instead, consider the main use-case promoted by the authors, that of complementing the partial verification by testing the unverified state space. The output in this situation would combine the predicate annotations and some form of a test suite. If the test suite continued to be inconclusive, that is, every test passed, assessing the progress

or shortcomings of the verification attempt would be challenging. That is, if such a scheme were to be adopted, when does a user consider the combined verification and testing efforts to be sufficient if no errors are discovered? It is unclear how this technique could be integrated into existing quality assurance efforts because existing metrics, widely used in testing, are not immediately applicable. As such, the combined output of predicates and some form of a test suite would, at best, be appealing to an audience already familiar with verification.

The application of these annotations [30] for bounded abstract interpretation [32] also have a rather limited audience. The technicalities of the approach, involving a combination of two variants of `assume` statements, make these annotations, at best, only accessible to users informed on abstract interpretation. It is, once again, important to emphasize that manually inspecting these annotations is not one of the intended use cases of this output and, as such, our observations in no way attempt to devalue these contributions. We merely consider the proposed approached and the associated output under a different light.

The possible application of these annotations [30] for encoding inferred preconditions has been proposed [86]. However, to the best of our knowledge this approach has not been implemented yet. This output would be accessible to the majority of the verification community, but hardly targeting the wider software development audience since it is not clear what these preconditions would look like, how they would be used or what the gain in practice from encoding them using the special notation instead of as preconditions would be.

A coverage estimation [77] for explicit state model checkers, in principle, can only be adequately interpreted by users familiar with these tools. As such, the audience is limited to experts or, at most, a fraction of the verification community.

Exposing the most actively explored parts of an Alloy specification [41]

can be useful for a user already experienced with this type of specifications and some understanding of the translation to propositional logic. That is, the output would be useful mostly for an expert audience or, at the very least, a user with some knowledge of verification.

The idea of using ranges over input variables has been proposed for bounding the failing input space of a program [47]. The choice of only using ranges as the output representation makes it, arguably, more amenable to users than unrestricted predicates over several theories. However, the output still appeals the verification community mainly, not the software development community at large.

Highlighting parts of a specification that are causing timeouts [27] gives a user familiar with Dafny [65] valuable hints as to what could be causing difficulties in the verification process. Users without previous knowledge on SMT-based [4] interactive verification will most likely struggle to extract insights. Therefore, we consider the target audience of this output limited to an expert audience.

A correctness score [80], intended to summarize the results of several tools, is suitable for an audience already well versed in verification.

A bound on the probability of reaching an error condition [73, 44] can be consumed by a broad audience. However, correctly interpreting the probability with respect to the usage profile provided can be not only unintuitive but also misleading. Moreover, understanding how sensitive the result can be with respect to the usage profile provided can even exceed the expertise of member of the verification community. As such, the audience for this type of output could only reach a portion of the verification community.

Another type of output accessible to the verification community is that of angelic verification [38]. The output, when the outcome is inconclusive, consists of a proposed angelic environment, encoded as a set of predicates assumed to hold for the input of the program and for external modules. The output could be hard to interpret for a software engineer without a background

in formal methods, but the representation is conceptually straightforward for users within the verification community.

Representations targeting the community of software developers at large also exist. One of the examples discussed proposes to produce a mutation score [50] can be used as is by the significant group of software developers already using mutation analysis [1]. Other such representations are code coverage metrics [37], which are massively used by software developers.

Conclusions and Motivation for our Line of Research

Figure 2.1 loosely illustrates the distribution of existing output representations. The axes depicted correspond to the broadest plausible target audience, in the X axis, and the degree of support for arbitrary interruptions, in the Y axis.

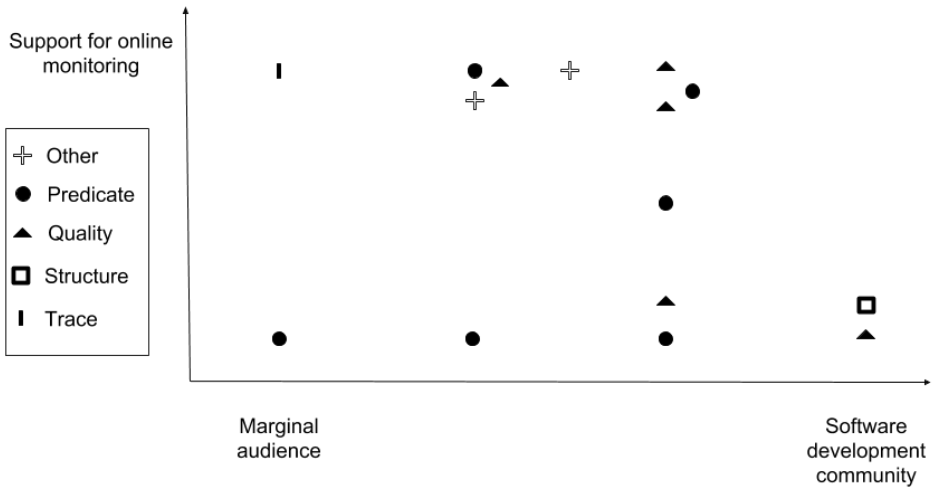

Fig. 2.1: Distribution of partial verification results

The different predicate-based representations considered cover the spec-

trum of a marginal audience, experts and in some cases even the verification community at large. Moreover, on-line monitoring with some of these approaches could also potentially be implemented.

Quality-based representations cover a wider audience, with one output representation reaching the software development community at large. The limitation, in this case, being that on-line monitoring is not available.

We only encountered one structure-based representation in the existing literature. This representation is accessible to the software development community but, again, cannot be produced on demand during the execution of the underlying verification technique.

The only trace-based representation available is not easily interpretable by users without significant knowledge of the underlying techniques involved. However, a trace-based output can be produced on-line at any moment during the verification attempt.

Finally, other types of representations exist reaching as far as a portion of the verification community.

To sum up, existing representations can potentially reach the wider software development audience in its quality-based and structure-based forms, comprising widely used metrics such as mutation and code coverage scores. However, to the best of our knowledge, in these cases arbitrary interruptions are not supported and existing approaches are inherently incompatible with on-line monitoring.

Figure 2.1 highlights the lower density of representations reaching the software development community at large and, more evidently, the lack of existing works combining these two desirable characteristics.

Recent work focusing on inconclusive outcomes of verification attempts shows an increased interest in this area. The characterization of existing work we provide also suggests a healthy dispersion across the two usability dimensions we put forth.

However, there seems to exist an area yet to be explored in the simul-

taneous combination of both usability dimensions. That is, approaches that provide different degrees of user-awareness while also enabling on-line monitoring of the underlying verification techniques have not been published in the related literature.

We believe the combination of both features could drastically reduce the barrier of entry for inexperienced developers while also providing interesting feedback to the verification community.

Our research falls in this spectrum. Model checker execution reports, discussed in Chapter 3, which provide user-aware trace-based representations of inconclusive verification outcomes. We propose some extensions to the basic approach on Chapter 4. These extension further improve the usability of execution reports and we comment on a number of use-cases involving the software development community at large that are likely enabled with minor modifications.

We move further in the direction of state-of-the-practice awareness by providing a structural coverage metric for software model checking that is compatible, in terms of its guarantees, to the analogous metrics in the context of software testing. Furthermore, this output can be produced without user interaction, in contrast with the approaches introduced in Chapters 3 and 4 Our contributions with respect to coverage measures for partial software model checks are discussed in Chapter 5

With these contributions, we combine the two dimensions of usability by supporting arbitrary interruptions and providing output with increasingly larger possible audiences and varying degrees of detail.

RESUMEN: PANORAMA DE RESULTADOS PARCIALES DE VERIFICACIÓN

Este capítulo consiste en una reseña estructurada del trabajo relacionado con proveer representaciones de resultados parciales de técnicas de verificación. Las contribuciones centrales de este capítulo son, por un lado una amplia resña del trabajo relacionado y, por otro, un encuadre para clasificar trabajo en este área de investigación bajo la perspectiva de intentar cuantificar y comunicar el progreso o limitaciones de un intento de verificación inconcluso. El capítulo concluye con una discusión de aspectos a mejorar que nos motivaron inicialmente a empujar la línea de investigación desarrollada en detalle a lo largo de la tesis.

La reseña estará enfocada en trabajos relacionados con intentos de verificación no concluyentes, es decir, intentos que por alguna razón fueron interrumpidos previo a determinar si la propiedad de interés se cumple. No consideraremos en esta reseña el trabajo relacionado con testing, ya que es inconcluyente por naturaleza y la literatura de testing se ha desarrollado naturalmente bajo esa premisa. En cambio, otras técnicas de verificación son correctas bajo ciertas hipótesis. En esos casos, determinar la calidad y las garantías provistas por resultados inconcluyentes implica desafíos interesantes que han sido ignorados en muchos casos.

3. MODEL CHECKER EXECUTION REPORTS

Introduction

Software model checking [59] constitutes an undecidable problem and, as such, even an ideal tool will in some cases fail to give a conclusive answer. In practice, undecidability is not the only issue. The vast state spaces can lead to the complete exhaustion of system resources or impractically long execution times.

Software model checking has been making steady progress during the past decade and today's state-of-the-art software model checkers can handle specific industrial problems particularly well. For instance SDV [3] is highly successful in finding bugs in Windows device drivers.

Unfortunately, some instances take hours of computation, only to inform the user that no counterexample was found within the allotted time or memory limit. A user facing this situation is confronted with several high-level questions about what the verification attempt actually achieved. Should she retry with a longer time limit? How much longer? Is the tool making any progress on the instance? Maybe she should try another technique?

Our goal is to extend and complement existing work on partial verification by providing a different way for users to observe the work performed by the software model checker. An important step towards our goal is to be able to answer much simpler inquiries about incomplete verification attempts.

We believe answering the following informal questions would be valuable for a user after an inconclusive verification attempt:

- Can partial safety assurances about the system be extracted from an incomplete verification attempt? For instance, a user that receives a report showing that a whole class of relevant behaviors has been exhaustively checked may use this as part of a dependability case.

- Can behavior that was not analyzed be explored by a user? For instance, a user that can observe that relevant classes of behavior have not even been looked at by the checker, let alone verified, may decide that what seemed like a sufficiently thorough analysis is not such (e.g., an inexperienced user would benefit from knowing that a tool based on BMC with a fixed bound can sometimes give up without ever exploring anything beyond an initialization loop[63]). Moreover, a more experienced user attempting full verification may decide that a drastic change in the verification strategy is needed (e.g., another tool, abstracting the system-under-verification, etc.).

By incomplete verification attempt we refer to when a software model checker fails to confirm any counterexample as feasible and also fails to prove the instance safe.

In this chapter we explore answering the first question using the notion of *safe cone*. A *safe cone* is a finite trace for which any extension has been analyzed in the incomplete verification attempt. A *minimal feasible safe cone* represents compactly a set of traces that have been successfully verified by the checker. For the second question we build on the notion of a *frontier*. A *frontier trace* is a feasible finite trace that was analyzed by the checker but that none of its feasible extensions were. Frontier traces represent compactly classes of traces that were not explored by the checker.

Our hypothesis is that execution reports that under-approximate the set of minimal feasible *safe cones* and the set of maximal feasible *frontier traces* can be computed in reasonable time (with respect to the cost of verification) and can provide non-trivial feedback on incomplete verification attempts.

We will start by illustrating with examples the information we wish to extract and defining a possible formalization of the idealized properties it should have. Subsequently, we define and discuss Execution Reports, an under-approximation of the ideal output. Afterwards, we instantiate these concepts for the family of techniques based on Abstract Reachability Trees

(ART) [53], and discuss a proof-of-concept implementation built as an extension of CPAchecker [13]. We also include an empirical evaluation of our implementation.

We conclude this chapter with a discussion of related work and how our approach compares to existing techniques followed by a few concluding remarks.

Motivation: What has and has not been analyzed?

We frame our work in the context of a verification attempt being interrupted before its completion.

Many techniques perform verification in a way that the state space is incrementally explored. This is the case for software model checking techniques and implementations like BMC [19], lazy predicate abstraction [53], inlining and unrolling based-techniques like Corral [64], DSE [21] and, in general, ART-based implementations of various techniques, including explicit value analysis [16] and CEGAR variants of some of the previous techniques among other.

In these techniques and implementations one can understand that incremental exploration leads to an incremental but silent increase of *analyzed* traces, i.e. sequences of statements, as depicted in the examples that follow. Moreover, certain statement traces will reach a portion of the behavior space that has been fully verified, implicitly defining a *safe cone* containing all possible ways of extending such traces.

Understanding that partial explorations provide safety assurances, we are particularly interested in the *frontiers* defined by minimal *safe cone* traces and maximal *analyzed* statement traces.

We illustrate these concepts with a verification attempt of the instance in Figure 3.1 with bounded model checking.

Example 1 (Analyzed behaviors in BMC)**.** *The code snippet in Figure 3.1 corresponds to a parametric test harness exercising the method* `min`*. The harness*

```
1  int nondet();
2  int min(int a[], int n) {
3      int res = a[0];
4      for (int i=0;i<n;++i)
5        if (a[i] < res) res = a[i];
6      return res;
7  }
8  void init_vector(int a[], int n) {
9      int i = 0;
10     for (i=0;i<n;++i) {
11       a[i] = nondet();
12     }
13 }
14 void test_min(int large) {
15     int n;
16     if (large) {
17         n = 20;
18     } else {
19         n = 1;
20     }
21     int a[20];
22     init_vector(a, n);
23     int min_elem = min(a, n);
24     assert(min_elem <= a[0]);
25 }
```

Fig. 3.1: Harness for method `min`

input as well as the result of method `nondet` *are interpreted by the verification technique as non-deterministic. Our verification attempt, in this example, is not interrupted due to reaching a resource limit but instead due to using a bounded model checker [19] with a bound on the number of loop iterations set to 3.*

Using this configuration, the tool would perform an exhaustive exploration but would disregard any executions involving the fourth loop iteration of the `for`*-loop in* `init_vector`*.*

Any sequence of statements reaching line 19 (`n = 1;`*) will necessarily satisfy the safety property, since the incomplete verification attempt would not find any assertion failures and the loop within* `init_vector` *can be exhaustively analyzed. That means the sequence of statements consisting of lines 15 (*`int n;`*), 16 (*`if (large)`*) and 19 (*`n = 1;`*) defines a* safe cone*, because every continuation of the statement trace is also safe. The former trace is also minimal in the sense that the trace resulting from removing the last statement, line 19, is not a* safe cone*.*

Moreover, any execution that carries out line 10 (`for (i=0; i<n; ++i)`*) at most 4 times (3 full iterations and 1 bound check) is also analyzed by the incomplete verification attempt. In contrast, any sequence containing line 10 at least 5 times is ignored by BMC and therefore will not be examined. Therefore, traces containing line 10 at least 5 times are not elements of analyzed. That is, the trace composed of lines 15, 16, 17, 21, 22, 9 and then 4 repetitions of lines 10 (*`for (i=0;i<n;++i)`*) and 11 (*`a[i] = nondet();`*) is a maximal analyzed trace.* △

We will now revisit these concepts from an entirely different technique: lazy predicate abstraction [53].

Lazy predicate abstraction, the algorithm used by BLAST [53], consists of two alternated phases. The first phase generates, on-the-fly, a reachability

tree whose nodes correspond to vertices of the Control Flow Automaton[1] of the program. This process goes on until exhaustively exploring the tree or until reaching a node that corresponds to an assertion failure. Each node is associated with a predicate, initially *true*, that must hold for any path reaching that node and helps prune the successors that are not reachable. If, and when, a node that represents an error is reached, the second phase deals with analyzing the potential counterexample to determine whether the path reaching the error node is feasible. If the latter phase determines the potential counterexample is infeasible, the reachability tree is refined by strengthening the predicates associated to the appropriate nodes so that the path reaching the error node is pruned. Lastly, when a counterexample is produced, it can be checked with a more precise analysis to ensure its feasibility. However, if this additional check fails, the search is abandoned.

Example 2 (Analyzed behaviors in lazy predicate abstraction)**.** *The code presented in Figure 3.2, reproduced from the paper presenting Conditional Model Checking [11, 12], contains a non-linear safety property. As explained, the construction of the reachability tree will reach the error node and the second phase would attempt to verify the feasibility of the path leading to it. The feasibility check is usually implemented by creating a verification condition to be checked by an underlying SMT solver and, therefore, inherits the latter's limitations. In particular, SMT solvers usually cannot handle non-linear arithmetic and, instead, model multiplication as uninterpreted functions. Concretely, the SMT solver would not be able to prove the path infeasible. However, the subsequent counterexample feasibility check would prove the path is actually infeasible, causing the exploration to stop.*

Given the failure to analyze the assertion, the following is a maximal analyzed trace: lines 3 (`int p = nondet();`*), 4 (*`if(p)`*), 9 (*`int x = 5;`*), 10 (*`int y = 6;`*) and 11 (*`int r = x * y;`*).*

[1] A Control Flow Automaton, similar to a Control Flow Graph, captures the control flow of the program, where nodes correspond to locations and edges are labeled with statements.

```
1  int nondet();
2  int main() {
3    int p = nondet();
4    if (p) {
5        int i;
6        for (i = 0; i < 1000000; i++);
7        assert(i >= 1000000);
8    } else {
9        int x = 5;
10       int y = 6;
11       int r = x * y;
12       assert(r >= x);
13   }
14   return 0;
15 }
```

Fig. 3.2: Non-linear arithmetic. Reproduced from CMC papers [11, 12].

On the other hand, the then *branch of the* `if`*-statement can be successfully verified with this technique, since tracking the predicate* `i < 1000000` *suffices to prove the assertion always holds when execution leaves the* `for`*-loop.*

The successful verification of the then *branch would make the trace composed of lines 3 (*`int p = nondet();`*), 4 (*`if (p)`*) and 5 (*`int i;`*) a* safe cone. △

Execution Reports (ERs)

We now formally define execution reports as an under-approximation of the set of *safe cones* and *frontier* traces. These definitions rely on two predicates (*analyzed*? and $isSafeCone$?) whose definition depends greatly on the underlying verification technique used in the incomplete verification attempt. Consequently, in this section we simply provide properties that predicates *analyzed*? and $isSafeCone$? ought to satisfy. In the next section we ground

the definition of these predicates for Abstract Reachability Tree based verification techniques [53].

Examples 1 and 2 illustrate how the notions of *analyzed* and *safe cone* apply to diverse techniques. We will capture these notions as predicates over traces in the following definitions.

Let the alphabet Σ contain all statements in a program, making $\pi \in \Sigma^*$ a sequence of statements.

Property 1. *The predicate* $analyzed? : \Sigma^* \to Bool$ *satisfies the following property, where* $\cdot$ *stands for concatenation:*

$\forall \pi, \pi' \in \Sigma^*. \neg analyzed?(\pi) \to \neg analyzed?(\pi \cdot \pi')$

Property 1 aims to formalize the notion of incrementality that we implicitly used throughout the examples. Note that this property is logically equivalent to its contrapositive, that is, $analyzed?(\pi \cdot \pi') \to analyzed?(\pi)$, as expected of an incremental exploration.

Property 2. *The predicate* $isSafeCone? : \Sigma^* \to Bool$ *satisfies the following property, where* $\cdot$ *stands for concatenation:*

$\forall \pi, \pi' \in \Sigma^*. isSafeCone?(\pi) \to isSafeCone?(\pi \cdot \pi')$

Property 2 reflects our notion of *safe cone* as a trace reaching a fully analyzed portion of the behavior space. Any trace extension will necessarily also be a *safe cone*.

Property 3. *The predicate* $isSafeCone? : \Sigma^* \to Bool$ *satisfies the following property:*

$\forall \pi \in \Sigma^*. isSafeCone?(\pi) \to analyzed?(\pi)$

Property 3 formalizes the connection between the two predicates, in particular how $isSafeCone?(\pi)$ subsumes $analyzed?(\pi)$.

Definition 1. *Given a trace* $\pi \in \Sigma^*$ *and a program* $\mathcal{P}$*, we introduce the following predicate:*

$feasible_{\mathcal{P}}(\pi)$ *holds* iff *there exists a concrete execution of the program* $\mathcal{P}$ *that executes* π*.*

Given that we will always refer to a single program at a time, the system-under-analysis, we will omit the subscript.

Property 4. *The predicate analyzed*? : $\Sigma^* \rightarrow$ *Bool satisfies the following property, where* φ *is a boolean predicate that captures the safety property of interest:*

$\forall \pi \in \Sigma^*.\ analyzed?(\pi) \wedge\ feasible(\pi) \rightarrow \varphi(\pi)$

Property 4 is at the core of interpreting *analyzed* traces as providing safety assurances. This property also holds for $isSafeCone?$ due to Property 3. The predicate $feasible(\pi)$ in the antecedent places the focus of safety assurances on feasible traces, that is, traces that correspond to actual behaviors of the system-under-analysis.

Recall that we provide properties that constrain the predicates *analyzed*? and $isSafeCone?$ but not concrete definitions of these predicates as specific definitions depend on the underlying verification technique. We now define the set of *safe cones* and *frontier traces* of an incomplete verification attempt.

Definition 2. *The set* $SafeCone$ *is defined as follows:*

$SafeCone = \{\pi \cdot s | \pi \in \Sigma^*, s \in \Sigma, \neg isSafeCone?(\pi) \wedge feasible(\pi \cdot s) \wedge isSafeCone?(\pi \cdot s)\}$

Definition 3. *The set* $Frontier$ *is defined as follows:*

$Frontier = \{\pi \cdot s | \pi \in \Sigma^*, s \in \Sigma,\ analyzed?(\pi) \wedge feasible(\pi \cdot s) \wedge \neg analyzed?(\pi \cdot s)\}$

Definitions 2 and 3 are related to Properties 2 and 1 respectively, since the incrementality of the analysis is key to the search for maximal *analyzed* traces, as in the set $Frontier$, and minimal *safe cone* traces, as in $SafeCone$.

The set $SafeCone$ can provide safety assurances about the system, as in Example 2, where the *then* branch of the `if`-statement has been fully verified.

Conversely, $Frontier$ can suggest shortcomings in the incomplete verification attempt. For instance, in Example 1, the existence of a trace $\pi \in Frontier$

that did not even go past the initialization loop suggests an important part of the test harness was not sufficiently analyzed.

The conclusions obtained from inspecting both sets can be useful to assess the progress achieved throughout the incomplete verification attempt.

We anticipated the intuitive notion captured by these definitions in Examples 1 and 2, but there is one important consideration that we omitted so far and now included in the definitions: feasibility.

Feasibility is relevant because, by definition, infeasible traces do not correspond to system behaviors. Without feasibility guarantees, interpreting each trace would require careful analysis, because it could mislead a user into either increasing or decreasing her confidence in the system-under-analysis.

Definition 4. *An* execution report *is a tuple* (S, F) *where* $S \subseteq SafeCone$ *and* $F \subseteq Frontier$.

Definition 4 defines *execution reports* as under-approximations of the sets $SafeCone$ and $Frontier$, allowing empty sets as valid *execution reports*.

The sets $SafeCone$ and $Frontier$ can grow quickly, making it extremely inefficient to compute the full sets. Furthermore, some of the traces can be redundant, in a sense, if they only differ in a few statements, making it sensible to under-approximate.

Ideally, it would be desirable to characterize these under-approximation. However, we opted in this work for a notion of *completeness* with respect to statements in the code that does not fully characterize the sets S and F but does not allow, in the general case, empty sets as valid *execution reports*: For both sets, $SafeCone$ and $Frontier$, we require that if a trace $\pi \in SafeCone$ (resp. $Frontier$) ending in a specific location l exists, then there exists $\pi' \in S$ (resp. F) and π' also ends in l. This completeness guarantee does not force extremely large sets of paths to be reported and loosely resembles a notion of coverage. Our algorithm to generate *execution reports* guarantees this completeness criterion.

Reports for ART-based implementations

Throughout this section we will explain how we generate Execution Reports for Abstract Reachability Tree (ART)-based [53] techniques.

ARTs constitute a relevant intermediate data structure used in verification. The variety of techniques implemented using ARTs makes them ideal for our proof-of-concept implementation. ART-based implementations comprise a wide range of dissimilar techniques, encompassing lazy abstraction [53], BMC [19], explicit value analysis [16] and CEGAR variants of some of the previous [16], among other.

In order to explain how we generate Execution Reports for ART-based techniques, we first define *analyzed*? and $isSafeCone$? for these techniques in the following subsection.

We will use Assumption Automata, an existing machine-readable abstract representation of ARTs, for our implementation. Subsequently, wewill briefly explain Assumption Automata and their two states most relevant to us, `TRUE` and `FALSE`, which we will use to obtain *safe cones* and $frontier$ traces, respectively.

Finally, we will explain how we compute Execution Reports using an Assumption Automaton, produced by an earlier incomplete verification attempt, and the system-under-verification as input, as depicted in Figure 3.3.

ARTs as intermediate data structures

We have discussed how the concepts of *analyzed* and *safe cone*, captured by predicates *analyzed*? and $isSafeCone$? respectively, apply to example techniques. In this sub-section we will instantiate these concepts to Abstract Reachability Trees (ARTs) [53].

An ART is a tree whose nodes correspond to vertices of the CFA of a program and each node is associated to an element of an abstract domain. In the case of BLAST, that abstract domain is a lattice of predicates.

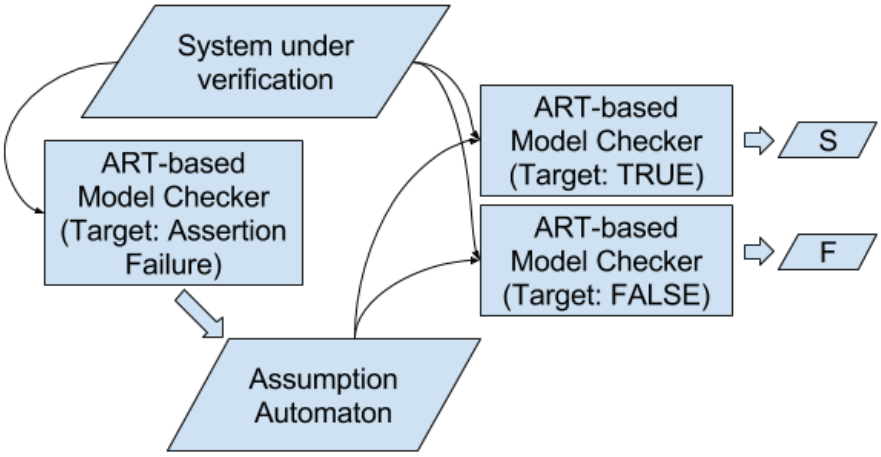

Fig. 3.3: Architecture of ER generation.

ART-based algorithms consist of two phases, the first one comprises an incremental generation of the ART and the second phase involves a more thorough counterexample check.

For the purpose of stating which traces can be considered *analyzed* we can ignore the abstract domain elements associated to each node. We can then think of an ART as a tuple $(G, W, q_0, covered)$ where $G = (S, \Sigma, \delta)$ is a graph, $W \subseteq S$ represents a wait list, $q_0 \in S$ is the initial state, $covered : S \to S$ captures subsumption between nodes, and $\delta : S \times \Sigma \to S$ is a transition function. The graph G captures the structure of the partially built ART. W is the wait list that contains elements to be analyzed in order to continue the construction of the ART. The function *covered* is necessary to represent subsumption between nodes but is not total. We say that a node e is covered by e' *iff* $covered(e) = e'$. Analogously, we consider that a node e is not covered *iff* $covered(e)$ is undefined.

To make the definitions easier to read, we will assume the following invariant holds for ARTs relative to *covered*:

Property 5. *$\forall e \in S$. such that $\exists e' \in S.covered(e') = e$ then $covered(e)$ is not defined.*

Informally, this means that no node is covered by another covered node.

In ART-based algorithms, a node e covered by e' need not be further analyzed because any error state found from e will also be found from e'. However, the sub-tree rooted in e cannot be considered exhaustively built unless that is also the case for the sub-tree rooted in e'. This observation is crucial to define what can be regarded as *analyzed* or *safe cone*.

We will extend δ as $\delta' : (S \cup \{None\}) \times \Sigma \to S \cup \{None\}$, with $None \notin S$ to make it total and resolve the *covered* function transparently as follows:

$$\delta'(q,s) = \begin{cases} \delta(q,s) & \text{if } \delta(q,s) \text{ is defined and} \\ & covered(\delta(q,s)) \text{ is not} \\ covered(\delta(q,s)) & \text{if both } \delta(q,s) \text{ and} \\ & covered(\delta(q,s)) \text{ are defined} \\ None & \text{otherwise} \end{cases}$$

Due to Property 5, $\delta'(q,\pi)$ is never a covered node.

Moreover, we will adapt δ' to traces with $\hat{\delta}' : S \cup \{None\} \times \Sigma^* \to S \cup \{None\}$ as follows, where $\cdot$ stands for concatenation:

Given $q \in S \cup \{None\}$, $s \in \Sigma$ and $\pi \in \Sigma^*$:

$$\hat{\delta}'(q,s) = \delta'(q,s)$$
$$\hat{\delta}'(q, s \cdot \pi) = \hat{\delta}'(\delta'(q,s), \pi)$$

We also define the auxiliary predicate $isPrefix : \Sigma^* \times \Sigma^* \to Bool$, where $\cdot$ stands for concatenation, as follows:

$$isPrefix(\pi', \pi) \textit{ iff } \exists \pi'' \in \Sigma^*. \pi' \cdot \pi'' = \pi$$

We will consider that $analyzed?(\pi)$ holds for a trace π *iff* no prefix of π reaches a node in W from the initial node. That is:

$$analyzed?(\pi) \textit{ iff } \neg \exists \pi' \in \Sigma^*. isPrefix(\pi', \pi) \wedge \hat{\delta}'(q_0, \pi') \in W$$

Informally, we consider π *analyzed* when no prefix of π reaches one of the states pending exploration, i.e. those in W. The predicate *analyzed*?(π) is clearly monotonic, satisfying Property 1: $\neg$*analyzed*?(π) means there exists a prefix that reaches W, therefore any extension $\pi \cdot \pi'$ will also share that prefix.

Similarly, we will consider that $isSafeCone?(\pi)$ holds for a trace π *iff* any prefix of π leads, from the initial node, to a sub-tree already exhaustively built. Intuitively, a sub-tree is exhaustively built when none of its nodes are in W, the set containing states pending exploration. More precisely:

$$isSafeCone?(\pi) \textit{ iff}$$
$$\exists \pi' \in \Sigma^*.isPrefix(\pi', \pi) \wedge \forall \pi'' \in \Sigma^*.\hat{\delta}'(q_0, \pi' \cdot \pi'') \notin W$$

Analogously, the predicate $isSafeCone?$ is monotonic, satisfying Property 2: $isSafeCone?(\pi)$ means there exists a prefix from which W is unreachable and consequently any extension $\pi \cdot \pi'$ will also share that prefix.

Assumption Automata

We now briefly explain Assumption Automata [11] because our implementation takes an Assumption Automaton as part of its input instead of ARTs.

An Assumption Automaton is essentially an abstraction of an ART where the elements of the abstract domain associated to each node are removed. Additionally, the structure is compressed by collapsing sub-trees which have been entirely verified into a single node `TRUE` and every node covered by another node is merged with the latter. Finally, nodes in the wait list are only connected to a single node `FALSE`. An Assumption Automaton encodes the progress achieved throughout an earlier incomplete verification attempt in a machine-readable format [9], which is the fundamental reason why we use it.

The definition of predicate *analyzed*?(π) can also be stated in terms of Assumption Automata. The predicate holds *iff* no prefix of π reaches `FALSE`. Once again, understanding an Assumption Automaton as a graph, the predi-

cate is defined as follows:

$$analyzed?(\pi) \textit{ iff}$$
$$\neg\exists \pi' \in \Sigma^* \text{ such that } isPrefix(\pi', \pi) \wedge \hat{\delta}'(q_0, \pi') = \texttt{FALSE}$$

Similarly, $isSafeCone?(\pi)$ holds iff a prefix of π reaches the node `TRUE`. That is:

$$isSafeCone?(\pi) \textit{ iff}$$
$$\exists \pi' \in \Sigma^* \text{ such that } isPrefix(\pi', \pi) \wedge \hat{\delta}'(q_0, \pi') = \texttt{TRUE}$$

It is worth noting that, even though one Assumption Automaton can correspond to several ARTs, applying the predicates $isSafeCone?$ and $analyzed?$ to the former or to any of the latter will yield the same result.

Generating reports for CPAchecker

We built a proof-of-concept implementation, consisting of two verification tasks, capable of generating execution reports for ART-based techniques on top of CPAchecker.

The input for our implementation is an Assumption Automaton and the original system-under-analysis. Our output are the sets S and F, composed of the counterexamples, i.e. traces, generated by two independent verification tasks, as shown in Figure 3.3.

We resort to the conceptual framework of Configurable Software Verification [13] to formalize how our algorithm is parametric, allowing different reachability analyses to be used. It is worth mentioning that the algorithm used to generate an Execution Report is in no way related to or restricted by the technique used for the original verification attempt, as long as the latter generates an Assumption Automaton.

Informally, we augment an existing ART-based algorithm by adding an Assumption Automaton state to the abstract domain element associated with each node. That is, in Figure 3.3, the process that produces F will attempt to

find feasible traces that reach the state `FALSE` in the Assumption Automaton, whereas `TRUE` will be the target state in the case of the process generating S.

We already explained the basics of ART-based algorithms, but some more detail is necessary to define our extension. The framework of Configurable Software Verification allows us to define a Configurable Program Analysis (CPA) $\mathbb{P} = (D_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \texttt{merge}_{\mathbb{P}}, \texttt{stop}_{\mathbb{P}})$ in terms of an abstract domain $D_{\mathbb{P}}$, a transfer relation $\rightsquigarrow_{\mathbb{P}}$, a merge operator $\texttt{merge}_{\mathbb{P}}$, and a termination check $\texttt{stop}_{\mathbb{P}}$.

Moreover, we can define a CPA as a composition of other CPA. CPA composition formalizes how we can plug our set-specific analysis, e.g. $SafeCone$ or $Frontier$, into existing reachability analyses, such as explicit value analysis or predicate abstraction.

A CPA, either simple or composite, can be analyzed with one of the several variants [13, 18] of the basic algorithm used in Configurable Software Verification, which we reproduced in Algorithm 1.

Algorithm 1 differs only slightly from the one in the original presentation of Configurable Software Verification [13]. We added lines 13 and 14, because we want to stop the exploration and return as soon as a target state is found. We also consider the sets `waitlist` and `reached` as inputs, making the core analysis more amenable to extensions [16, 18], such as CEGAR [34] or finding multiple counterexamples, which requires a similar approach.

The check performed in line 13, `isTargetState(`e'`)`, as mentioned, will verify whether `TRUE` (respectively `FALSE`) is part of the composite state e'. `TRUE` captures the portions of the state space which have been exhaustively verified, whereas `FALSE` corresponds to nodes in the ART pending analysis at the time the verification attempt was interrupted. In other words, the check evaluates whether any path π leading to e' satisfies $isSafeCone?(\pi)$ (resp. $\neg analyzed?(\pi)$). Any prefix of π will necessarily satisfy the negation, otherwise it would have triggered the generation of a counterexample. Therefore, if the check `isTargetState(`e'`)` is positive, as long as $feasible(\pi)$ holds, $\pi \in SafeCone$ (resp. $\pi \in Frontier$) since Property 2 (resp. Property 3) is

input : A configurable program analysis $\mathbb{P} = (D, \rightsquigarrow, \texttt{merge}, \texttt{stop})$, a set `waitlist` of elements of E, denoting the set of elements of the semi-lattice of D, a set `reached` of reachable abstract states.

output: An updated `reached` and `waitlist`.

```
1  while waitlist ≠ ∅ do
2      pop e from waitlist
3      for each e′ with e ⇝ e′ do
4          for each e″ ∈ reached do
5              // Combine with existing abstract state
6              e_new := merge(e′, e″)
7              if e_new ≠ e″ then
8                  waitlist := (waitlist ∪{e_new}) \ {e″}
9                  reached := (reached ∪{e_new}) \ {e″}
10         if ¬stop(e′, reached) then
11             waitlist := waitlist ∪ {e′}
12             reached := reached ∪ {e′}
13             if isTargetState(e′) then
14                 return (reached, waitlist)
15     return (reached, ∅)
```

Algorithm 1: Basic algorithm (from Configurable Software Verification [13, 18])

satisfied by any such trace π.

If such a trace π is found, phase two collects π if its feasibility is confirmed. Regardless, Algorithm 1 is executed from where it left off, since the full internal representation, the sets `reached` and `waitlist`, was returned at the end of the previous call. This alternation between phase one and two produces a number of traces which will constitute S (resp. F) in the *execution report*.

In order to preserve the properties of ER, we require the underlying analysis

to be precise, that is, it does not produce spurious counterexamples. Precise variants of different analyses, such as predicate abstraction and explicit value analysis, have been expressed as CPA [18, 16]. The actual algorithms for these techniques are based on Algorithm 1 and contain additional modifications but in both cases the composition with other CPA is still supported.

Let's now define our CPA $\mathbb{P}(A) = (D_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \texttt{merge}_{\mathbb{P}}, \texttt{stop}_{\mathbb{P}})$ with $D_{\mathbb{P}}$ based on the flat lattice for the set of all the states in the Assumption Automaton A taken as input. For the transfer relation, $e \overset{stm}{\rightsquigarrow}_{\mathbb{P}} e'$ if there exists a transition labeled stm from state e to state e' of the Assumption Automaton A taken as input and $e \overset{stm}{\rightsquigarrow}_{\mathbb{P}} \bot$ otherwise. Finally, $\texttt{merge}_{\mathbb{P}} = \texttt{merge}^{sep}$ and $\texttt{stop}_{\mathbb{P}} = \texttt{stop}^{sep}$, where $\texttt{merge}^{sep}(e, e') = e'$ and $\texttt{stop}^{sep}(e, R) = (\exists e' \in R : e \sqsubseteq e')$. The operator $\texttt{merge}_{\mathbb{P}}$ and $\texttt{stop}_{\mathbb{P}}$ affect the precision and performance of the analysis [13]. The proposed definitions for these operators aim to increase the former possibly at the cost of the latter. In any case, the analysis can be made entirely precise by checking the feasibility before reporting any counterexample. Therefore, we define these operators for completeness but other options are entirely possible and might be desirable for performance.

We are now ready to define a composite program analysis $\mathcal{C} = (\mathbb{A}, \mathbb{P}, \rightsquigarrow_{\times}, \texttt{merge}_{\times}, \texttt{stop}_{\times})$, where $\mathbb{P}$ is the one just defined and $\mathbb{A}$ is an arbitrary analysis. We will use $\texttt{merge}_{\times} = \texttt{merge}^{sep}$, $\texttt{stop}_{\times} = \texttt{stop}^{sep}$ and define the transfer relation $\rightsquigarrow_{\times}$ such that $(l, r) \overset{g}{\rightsquigarrow}_{\times} (l', r')$ iff $l \overset{g}{\rightsquigarrow}_{\mathbb{A}} l'$ and $r \overset{g}{\rightsquigarrow}_{\mathbb{P}} r'$.

In order to satisfy the completeness guarantees with respect to locations in the program that we put forth in sub-section 3.2.1, we only require that the underlying analysis does not merge different location states. Neither of the analyses we tried, lazy predicate analysis nor explicit value analysis, do.

It is worth emphasizing that the predicates *analyzed*? and $isSafeCone$? only play a conceptual role to make sure our implementation preserves the informal semantics of $frontier$ traces and *safe cones* discussed in Section 3.2. We merely mapped the intuitions to the specific case of ART-based algorithms and computed the sets S and F directly from the Assumption Automaton.

	Safe: explicit from explicit-AA			Safe: predicate from explicit-AA		
	# traces	Full?	CPU time	# traces	Full?	CPU time
gigaset.BUG.c	-		927.44s	1	✓	417.08s
farsync.BUG.c	-		926.65s	1	✓	280.36s
loop.BUG.c	-		901.08s	-		135.17s
synclink_gt.BUG.c	1	✓	883.54s	2	✓	534.85s
ppp_generic.BUG.c	1	✓	894.03s	1	✓	855.15s
lirc_imon.BUG.c	-		901.05s	-		911.77s
token_ring.14.BUG.c	19	✓	25.93s	-		902.45s
transmitter.16.BUG.c	7	✓	16.43s	-		901.97s
toy.c	-		970.09s	0	✓	893.52s
		4/9 Full			5/9 Full	

Tab. 3.1: Complete results for S from an Assumption Automaton generated using Explicit Value analysis

Evaluation

This section aims to evaluate if our approach is capable of generating informative ERs within a reasonable time (with respect to the time budget invested in the original verification attempt). We analyze the performance of our proof-of-concept implementation with a set of standard benchmark instances. We also discuss the insights that we extracted from the output on this set of benchmarks.

All the files necessary to reproduce the experiments are available on-line: `https://github.com/rcastano/cpachecker-1/tree/submission_ase`

	Safe: explicit from predicate-AA			Safe: predicate from predicate-AA		
	# traces	Full?	CPU time	# traces	Full?	CPU time
toy1_BUG.c	0	✓	14.15s	0	✓	52.47s
token_ring.06.c	0	✓	48.40s	0	✓	272.59s
pipeline.c	0	✓	7.66s	0	✓	10.93s
token_ring.09.BUG.c	-		901.05s	-		901.07s
token_ring.04.c	0	✓	20.60s	0	✓	78.64s
token_ring.05.c	0	✓	50.49s	0	✓	299.75s
token_ring.08.c	0	✓	367.83s	-		901.09s
token_ring.14.BUG.c	-		963.61s	-		901.07s
mem_slave_tlm.3.c	0	✓	8.72s	0	✓	27.59s
token_ring.07.c	0	✓	108.09s	-		901.08s
mem_slave_tlm.5.c	0	✓	15.80s	0	✓	145.01s
mem_slave_tlm.4.c	0	✓	9.00s	0	✓	27.56s
kundu.c	0	✓	16.90s	0	✓	135.16s
pktcdvd.BUG.c	-		900.97s	5	✓	166.12s
toy.c	0	✓	13.91s	0	✓	97.82s
token_ring.03.c	1	✓	16.43s	0	✓	68.03s
		13/16 Full			12/16 Full	

Tab. 3.2: Complete results for S from an Assumption Automaton generated using Predicate analysis

	Frontier: explicit from predicate-AA			Frontier: predicate from predicate-AA		
	# traces	Full?	CPU time	# traces	Full?	CPU time
toy1_BUG.c	10	✓	20.44s	23	✓	133.11s
token_ring.06.c	153	✓	83.51s	-		910.96s
pipeline.c	1	✓	8.93s	1	✓	19.71s
token_ring.09.BUG.c	-		1000.85s	-		901.10s
token_ring.04.c	29	✓	37.28s	136	✓	527.54s
token_ring.05.c	85	✓	78.29s	-		900.90s
token_ring.08.c	871	✓	628.35s	-		905.49s
token_ring.14.BUG.c	-		1000.78s	-		939.53s
mem_slave_tlm.3.c	5	✓	14.97s	5	✓	25.82s
token_ring.07.c	337	✓	188.40s	-		910.68s
mem_slave_tlm.5.c	14	✓	45.17s	18	✓	302.88s
mem_slave_tlm.4.c	6	✓	16.07s	6	✓	27.05s
kundu.c	15	✓	32.30s	-		292.07s
pktcdvd.BUG.c	-		901.07s	39	✓	274.03s
toy.c	10	✓	19.93s	23	✓	137.26s
token_ring.03.c	2	✓	17.29s	3	✓	61.84s
		13/16 Full			9/16 Full	

Tab. 3.3: Complete results for F from an Assumption Automaton generated using Predicate analysis

	Frontier: explicit from explicit-AA			Frontier: predicate from explicit-AA		
	# traces	Full?	CPU time	# traces	Full?	CPU time
gigaset.BUG.c	-		930.02s	-		917.01s
farsync.BUG.c	-		918.63s	13	✓	98.98s
loop.BUG.c	-		937.49s	4	✓	153.23s
synclink_gt.BUG.c	-		917.86s	-		1000.97s
ppp_generic.BUG.c	-		901.01s	30	✓	94.69s
lirc_imon.BUG.c	-		901.05s	-		912.70s
token_ring.14.BUG.c	4	✓	12.65s	4	✓	16.58s
transmitter.16.BUG.c	10	✓	19.59s	10	✓	22.80s
toy.c	-		901.02s	-		931.81s
		2/9 Full			5/9 Full	

Tab. 3.4: Complete results for F from an Assumption Automaton generated using Explicit Value analysis

Performance

We evaluated our algorithms using the families SystemC and DeviceDrivers of the SV-COMP [7] set of benchmarks, which were used previously to evaluate CMC. Our experiments consisted of two different phases: 1) a verification attempt with a predefined time limit of 900 seconds; 2) production of ERs for the instances which ended the first phase with inconclusive results.

We used an Ubuntu 16.04 system equipped with an Intel® Core™ i7-3770 CPU clocked at 3.40GHz with 16GB of DDR3 memory for the experiments in a system without a swap partition running. We used BenchExec [17] to run the experiments and only allowed access to a single CPU core and 12GB of RAM.

We ran the verification phase over all the SV-COMP instances included in the reproduction package[2] of the original CMC presentation [12]. These bench-

[2] Available on-line: `https://www.sosy-lab.org/~dbeyer/cpa-cmc/`

Verification ER	explicit predicate	predicate predicate	explicit explicit	predicate explicit
F	55.56 %	50 %	22.22 %	75 %
S	22.22 %	75 %	22.22 %	81.25 %

Tab. 3.5: % of instances producing a full ER within 50% of the original verification time (900 s)

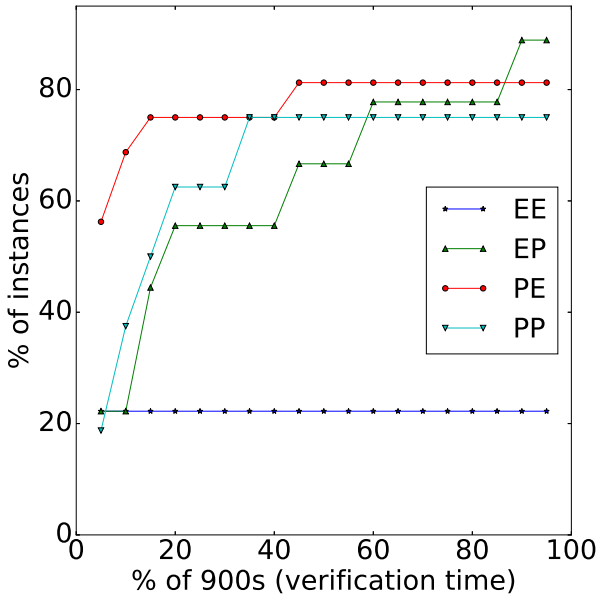

Fig. 3.4: Time until first element of S or F is produced

marks include instances from the sets SystemC and DeviceDrivers, comprising a total of 68 instances. We used explicit value analysis (`EV`) and predicate abstraction (`PA`), leaving 9 and 16 instances, respectively, with inconclusive results.

In the second phase, for each of these executions interrupted due to reaching the time limit, we generate the corresponding ER. To do so, we also experimented with both techniques, leading to 4 combinations of techniques, namely, `EV-EV`, `EV-PA`, `PA-EV` and `PA-PA`, where `EV-PA` denotes using `EV` for the first phase and `PA` for ER generation. For each combination, we considered two configurations, one that finds a single trace for each of S and F, to assess how early the second phase can produce output, and another one where we configure CPAchecker to continue listing counterexamples until exhausting its abstract state space, to determine the performance when generating the components to completion.[3]

Tables 3.1 to 3.4 include the full results of the second phase when running to completion. The number of traces is omitted for instances that did not yield conclusive results (for instance due to reaching the time limit or limitations of the analysis), as indicated with the dash. The column labeled "Full?" indicates with a check mark (✓) when the component (either S or F) was generated to completion within the time limit.

We will mostly focus on the number of full components generated, that is, the number of instances for which the component, either S or F, is generated to completion within the allotted time, as indicated by the fraction below the second column for each combination of techniques.

Tables 3.2 and 3.3 show that both generating the component S and the component F to completion seems possible for a significant number of instances when the Assumption Automaton was produced by lazy predicate abstraction. Furthermore, using either lazy predicate abstraction again (right) or explicit

[3] It is worth noting that a full ER component will satisfy the completeness guarantee with respect to locations in the program but will most likely not contain all valid traces.

value (left) for the generation of S shows similar performance results, with a slightly better performance of the latter.

In the case of F, there seems to be a greater difference in performance when using explicit value analysis (left) with respect to using lazy predicate abstraction (right) again for ER generation.

Table 3.3 suggests that using a different technique to generate the F component than the one used for verification could yield greater performance. This pattern also shows up in Table 3.4, where using lazy predicate abstraction (right) for the ER generation phase outperformed using explicit value analysis for an Assumption Automaton generated using explicit value analysis.

Tables 3.1 and 3.4 show a lower number of instances for which the components were produced to completion within the allotted time, compared to the case when the Assumption Automaton was produced after an initial verification attempt using lazy predicate abstraction. This could be explained by the size of Assumption Automata produced when using explicit value analysis, which is usually significantly larger than those produced with predicate analysis.

We consider these results encouraging in general, taking into account that the instances considered are those for which the initial verification attempt already exceeded the time limit.

Table 3.5 aggregates these results and shows the percentage of instances for which a full ER was generated within 50% (450 s) of the original verification time (900 s). This table essentially measures how often the ideal condition holds, that is: both S and F are produced to completion within a fraction of the original verification time. While the first, second and last column, in our opinion, suggest that standard techniques can achieve good performance and generate full ERs for widely dissimilar verification techniques, the third column shows that producing full ERs can, in some cases, demand significant computation, relative to the original verification time. The latter finding comes as no surprise, since our approach to generating ERs constitutes a verification

task in itself. Nevertheless, it seems clear to us that consistently generating full ERs requires further work.

However, we are also interested in understanding how fast, relative to the original verification time, our implementation starts generating output. Figure 3.4 shows the percentage of instances for which an element of either F or S is produced. Figure 3.4 suggests that a fraction of the original verification time can be sufficient to start generating traces.

We consider these performance results encouraging for a proof-of-concept implementation and discuss some promising ideas for further improvement in Section 3.6.

Number of traces

We mentioned a completeness criterion with respect to the statements of the system-under-verification, but this only provides a lower bound in the number of traces. Tables 3.1 to 3.4 show the number of traces that each component (S and F) contains.

The numbers in Table 3.2 prompted us to investigate the cause for the lack of *safe cones*. The underlying reason seems to be that reverse post-order is used as the traversal strategy for lazy predicate abstraction. This strategy heavily deprioritizes nodes with no successors in the CFA. In particular, a final return statement is not analyzed unless no other option exists. This exploration, unless additional specific configuration options are used, causes the analysis to seldom produce *safe cones*.

This detailed inspection prompted by Table 3.2 allowed us to gain insights on the inner workings of the technique. Moreover, wider availability of Execution Reports could provide an incentive for tool implementers to make implementation decisions that lead to richer intermediate results, which would be reflected in the corresponding Execution Reports.

Tables 3.2 and 3.4 show that in many cases the number of traces produced remains manageable: within the few dozen and in many cases less than 5.

We consider these number encouraging, since little effort on behalf of the user would be required to inspect the traces and assess their usefulness.

However, Table 3.3 shows that for a few cases, the number of traces produced can exceed the hundreds. In these cases, depending on the particular instance, it could be possible to group traces or inspect only a few, in both cases still gaining non-trivial information about the incomplete verification attempt with minimal effort.

Discussion

This section aims to shed light on the insights that can be extracted from ERs in practice. With this goal, we manually inspected the Execution Reports produced for the families SystemC and DeviceDrivers of the SV-COMP benchmark and also for combined instances used to validate CMC [12] and gained some anecdotal insights worth discussing.

The following features appeared repeatedly throughout the benchmark instances. Extracting insights from the ERs required knowledge about the instances. However, a brief inspection of the code, guided by the output of our tool, yielded the necessary information.

In all instances where the *Frontier* component was finished we identified seemingly relevant behaviors that had not been analyzed. More precisely, both SystemC and DeviceDrivers instances consist of a main while-loop with a non-deterministic termination condition and a set of methods from which one is chosen also non-deterministically and subsequently called. In all of the instances the *Frontier* component contained traces that did not even reach the end of the first loop iteration. We exemplified this sort of output in Example 1, where the initialization loop could not be entirely analyzed under certain non-deterministic choices. These insights contrast starkly with a coverage metric provided by CPAchecker. Contrary to the indications of insufficient analysis reflected in the *execution report*, the coverage metric provided by CPAchecker reported over 85% line coverage for 11 out of the 16 instances when applying

```
1  void main () {
2  int x = nondet();
3  if (x) main0 ();
4  if (!x) main1 ();
5  }
```

Fig. 3.5: Combined instance

lazy abstraction. This type of feedback could also be used to better understand the effects of either resource bounds or iteration bounds on the progress of the exploration.

Generating insight from an element $e \in S$ requires not only knowledge of the instance but also understanding how e constrains the execution from that point on. For some instances, further research is needed to properly evaluate the relevance of each element $e \in S$. However, for the combined instances, it was fairly easy to interpret ERs and they showed a relevant set of behaviors were safe. These instances have the structure shown in Figure 3.5, where method `main0` corresponds to the main method of a SystemC instance and `main1` corresponds to the main method of a DeviceDrivers instance. Much like the setting we showed in Example 2, where one branch of an `if`-statement was easier to verify than the other, in this case `main0` is significantly less challenging, for a specific technique, than `main1`. In these examples, S would contain the trace `int x = nondet();`, `if(x)`, `main0()`. The trace only constrains the value of `x`, which has to be positive, but `x` is a local variable, therefore `main0` is absolutely unconstrained and has been completely verified.

Related Work

Conditional model checking [12] (CMC) is an approach where model checkers are extended to produce results even when the verification run could not be completed successfully. The output, in its general form, is a condition under which the program can be safely run. CPAchecker [13] instantiates CMC by generating an Assumption Automaton. We use this implementation

to produce the ERs. ERs introduce the notions of *frontier* and *safe cone* to characterize the state space denoted by structures like Assumption Automata. The lack of any feasibility guarantees in Assumption Automata has significant consequences. A user might be misled by the size of the Assumption Automaton since a vast Automaton might correspond to a minuscule number of concrete feasible executions and also the other way around. We overcome this limitation by formally characterizing the properties of our output and adding explicit feasibility guarantees.

A slightly different approach [30, 31] to providing feedback of partial verification results consists of a language extension to be used to annotate assumptions made during verification. This extension can be used to annotate the code and explicitly state conditions under which the program is guaranteed to run safely. These annotations are especially well-suited for local assumptions, sometimes used during manual verification, and for uniform assumptions, such as the absence of integer overflows, which are not affected by the context in which they occur. However, these annotations are not well-suited to state assumptions made by some techniques, such as those based in unrolling loops [30] or those tackled by Assumption Automata, and as such, they are incomparable to our approach, which can provide value in these cases. One of the use cases of these annotations is to complement the verification efforts and produce a small test suite. This idea of testing to complement earlier verification efforts was later replicated [36] using Assumption Automata as input.

A recent extension of the Dafny IDE [65, 28] provides, as one of its many features, hints about parts of a specification that might cause timeouts. However, given the modular nature of the tool and the sort of specifications shown as examples, the approach tackles a different problem than ours and the feature might not be applicable in our setting.

There is also previous work on quantifying partial model checker explorations based on usage profiles [73]. These estimations are based on abstract models of behavior and heavily depend on the provided usage profile. A sim-

ilar approach [44] works applying symbolic execution over the source code of a system, without the need for an abstract model, but in this case, the implementation requires finite domains for all input variables, as well as a usage profile for each of them. Reliability as they define it can be hard to interpret and, once again, could be extremely sensitive to the usage profile provided. Both techniques can be used in conjunction with ours providing different value.

The modeling language Alloy [55] enables its users to specify structural properties. An extension of the Alloy Analyzer [41] highlights parts of the specification that are "problematic" or "hard", in the authors' own words, by monitoring the activity of variables and clauses in the underlying SAT solver. The output is inherently heuristic, in contrast, our technique and proposed implementation provide strong guarantees backed by a formal definition of the semantics of the output generated. The ideas behind the Alloy Analyzer extension could also be applied in conjunction with our techniques to provide additional information.

Our work is heavily influenced by the ideas and tool support of witness validation [9, 6], which we leverage as a machine-readable representation of exploration progress. However, we aim at enabling a richer manual interaction whereas that line of work also attempts to increase tool automation and reduce the need for manual inspection.

Summary

Software model checking is already capable of handling industrial instances and produce valuable results. However, some instances still remain intractable for full verification. Our work provides users with a different way to observe the progress achieved during an incomplete verification attempt by producing an execution report (ER).

We formulated the concepts of *analyzed* and *safe cone* traces and formalized the notion of ERs. We also discussed a proof-of-concept implementation to generate ERs and subsequently evaluated it both qualitatively and quanti-

tatively with benchmark instances.

One line of research we plan to follow involves exploring ways to abstract the traces included in the execution reports for easier visualization and understanding. In this setting, it could be useful to define both existential and universal semantics for abstract traces in the sense that some property applies to the some or every concretization of these abstractions. For instance, guaranteeing that every concretization has been analyzed, could be useful and would pose interesting challenges. In a similar line, to better assess the relevance of an element $\pi \in S$, we need to devise meaningful views and metrics of the possible continuations of e, e.g. statement coverage metrics of the cone defined by π.

We would also like to analyze how our technique performs for a specific verification technique and varying time limits.

It would be desirable to conduct a user study to assess the effectiveness of our approach once more competing output representations become available.

We intend to look into alternative usages of our output. For example, elements of an *execution report* can be leveraged as additional input to choose the right algorithm [81] to proceed after an incomplete verification attempt.

As mentioned in Section 3.2, the partition of system behaviors into the subsets proposed is also applicable to verification techniques beyond those already implemented using ARTs: Corral, DSE, techniques based on inlining or loop unrolling among other. Several other verification systems already produce specification violation witnesses [6] in a machine-readable format closely related to that of Assumption Automata. Producing Assumption Automata for incomplete executions of many of these software model checkers seems possible, making our *execution reports* immediately available to them. Making the necessary changes to these tools and evaluating our approach could bring new insights and challenges worth looking into.

We also consider looking into performance improvements in the ER generation phase, for example using techniques that trade off soundness for efficiency.

More precisely, instead of using a verification tool for ER generation, we could use simulation or graph exploration techniques that are not necessarily exhaustive.

RESUMEN: MODEL CHECKER EXECUTION REPORTS

En este capítulo presentamos un enfoque basado en trazas, o secuencias de instrucciones, para presentar el trabajo realizado durante la ejecución de un software model checker. El contexto en el que trabajamos es cuando estas herramientas culminan la ejecución sin haber alcanzado un resultado concluyente. Por ejemplo, entre las posibles causas de interrupción se encuentran haber alcanzado un límite de tiempo, haber superado los recursos disponibles y simplemente la existencia de una limitación en la técnica de verificación que no permite continuar la ejecución.

Las siguientes dos potenciales preguntas a ser formuladas, en lenguaje natural, por un usuario de estas herramientas guían nuestra representación:

En primer lugar, ¿es posible ofrecer garantías sobre el correcto funcionamiento del sistema a partir de una ejecución inconcluyente de un software model checker?

En segundo lugar, ¿es posible identificar comportamientos del sistema que no hayan sido analizados por el software model checker?

Las respuestas a estas dos preguntas ofrecen información complementaria, la primera aumentando la confianza en el sistema y la segunda permitiendo focalizar intentos subsecuentes de analizarlo.

Nuestra propuesta consiste, justamente, en generar dos tipos de trazas, apuntando a ofrecer información que ayude a responder esas dos preguntas. El primer tipo de trazas corresponde a aquellas que alcanzan regiones exploradas completamente durante el intento de verificación inconcluyente. Es decir, estas trazas ofrecen garantías respecto del correcto funcionamiento del sistema.

En cambio, el segundo tipo de trazas consiste en trazas que alcanza al límite de lo explorado por el software model checker al momento de interrumpirse la ejecución.

La hipótesis que validamos a lo largo de este capítulo es que es posible

extraer y presentar información no trivial de una ejecución de un software model checker en los reportes que definimos.

4. EXTENSIONS OF MODEL CHECKER EXECUTION REPORTS

Introduction

Model Checker Execution Reports [23] provide a novel way to present the work performed by a software model checker when the outcome of the verification attempt is inconclusive.

The long term aim of the work is to provide useful information to *a user* and, as such, understandability and usability are important goals. Throughout this chapter, we aim to discuss the shortcomings of the early iterations of Model Checker Execution Reports and present a number of extensions that partially tackle these limitations.

The contributions covered in this chapter comprise two main cores. Firstly, we show a number of questions that are brought up by inspecting model checker execution reports, thereby providing anecdotal evidence supporting the possibility of extracting insights from them. Secondly, we propose approaches to answer the follow-up questions naturally arising from the inspection of model checker execution reports.

The guiding research question of this chapter is what insights can model checker execution reports provide, if any. We will approach this inquiry by using standard benchmark instances as case studies and showing how the proposed approaches enable us to systematically gain understanding of the inconclusive verification attempts under different scenarios.

k-Unsoundness

The almost absolute lack of visibility of inconclusive results for most inconclusive verification outcomes is at the core of the motivation for our line of research. However, certain techniques do provide an intuitive mental model [60]

of the depth of the exploration. For example, some variants of bounded model checking analyze all program behaviors up to a bound in the number of loop iterations. These techniques lead, in many cases, to inconclusive outcomes. For instance, if the system-under-analysis contains code paths exceeding the iteration bound, it would usually not be possible to prove the program safe using such a model checker.

The specific way in which the state space is bounded makes it easier for a user, given a concrete execution, to determine whether such a behavior was analyzed or exceeded the bounds of the exploration.

We put forth the notion of k-unsoundness as a way to compare other inconclusive outcomes to those of a bounded model checker.

Definition 5. *We define that a partial software model check of a program $\mathcal{P}$ is k-unsound if and only if there exists $\pi \in \Sigma^*$, where Σ is the alphabet of statements, such that π is executable in $\mathcal{P}$, π has not been analyzed, that is $\neg analyzed?(\pi)$, and π executes at most k iterations for each of the loops it traverses.*

For Definition 5 we resort to the predicate *analyzed*?, which we presented as a conceptual basis for model checker execution reports. The definition can be refined into the following property of model checker execution reports.

Property 6. *A model checker execution report composed of the sets $S, F \subseteq \Sigma^*$ such that $F \neq \emptyset$ implies that the corresponding inconclusive verification outcome is k-unsound for some $k \in \mathbb{N}$.*

We will see that Property 6 holds by the definition of execution reports and k-unsoundness, a non-empty set F, containing all frontier traces, in an execution report, implies that the inconclusive outcome of the partial software model check is k-unsound for some k. Any trace $\pi \in F$, the frontier component of an execution report, satisfies the first two conditions in Definition 5. The first condition in Definition 5 is satisfied because π is executable in the system-under-analysis $\mathcal{P}$. The second condition is satisfied because π reaches the edge

of the internal representation used for verification. In the case of ART-based techniques, the predicate *analyzed*? can be understood as equivalent to an assumption automaton and π would be a path to a `FALSE` state, signifying that $\neg analyzed?(\pi)$ holds, that is, π was not captured by the analysis. Finally, since π is a finite trace, there is some bound k such that none of the loops in $\mathcal{P}$ are executed more than k times.

It is important to note that since execution reports do not necessarily contain all traces in $Frontier$, it is not possible to deduct, from an execution report, that an inconclusive outcome is not k-unsound, for any k.

Property 6 draws a connection between execution reports and k-unsoundness. The result leads to an efficient reuse of execution reports for providing k-unsoundness information to a user. Calculating the value of k implies counting, for each trace $\pi \in F$, the maximum number of iterations any loop is executed, which we can denote k_π. The result is that the inconclusive results is k-unsound with $k = \max(k_\pi)$. Performing this calculation has a lineal time complexity on the total number of statements of all traces in F, making it particularly efficient once the execution report has already been computed.

Insights from k-Unsoundness in Benchmark Instances

We generated model checker execution reports generated for Chapter 3 and inspected them as a preliminary empirical evaluation of k-unsoundness in practice.

Using Property 6, we used the traces in set F of each execution reports to prove k-unsoundness of the underlying partial software model check.

The observed values of k were rather discouraging: for the majority of instances the partial software model checks were 2-unsound. This observation held both for partial software model checks performed using lazy predicate abstraction and using value analysis.

This preliminary empirical evaluation, in our opinion, provides a novel way to quantify the potential shallowness of missed defects even after considerable

computation time (15 m).

Limitations

One of the main shortcomings of the notion of k-unsoundness is that it only provides evidence of deficiency in the exploration and never of its strength. It is entirely possible that a k-unsound outcome, although missing at least one system execution with less than k loop iterations, also explored other behaviors that far exceed k loop iterations.

Another limitation is that k-unsoundness can be coarse grained. For example, all loops are considered equal in terms of Definition 5. That is, if a trace finishes within the first k iterations of an initialization loop with a fairly simple loop guard, it leads to the same conclusion than if a trace also finishes within the first k iterations but traverses several conditional statements, making the path far harder to analyze.

Another shortcoming is that the understandability and usability of the concept of k-unsoundness is necessarily limited to the intuitions provided by bounds in bounded model checking.

One such limitation is that although it is straightforward to understand whether a specific execution exceeds a bound on the number of loop iterations, the opposite connection can be more convoluted. That is, gaining an intuitive understanding of which system executions traverse a loop less than a predefined number of iterations can be more involved.

One reason for the difficulty in mapping a bound to concrete executions is because the loop guard might not be expressed in terms that correspond to the number of loop iterations. For example, a linear search returning the first occurrence of the desired element.

Another obstacle in mapping a bound to a set of concrete executions would be the lack of a detailed understanding of the implementation by the user. In that case, certain parts of the implementation could contain loops that are ignored by the user. One such example could be the initialization of an

internal data structure.

In both of these cases, the existence of a k-unsoundness witness would be of little use, since the bound itself would be meaningless to the user.

Abstract Traces

The initial evaluation of model checker execution reports suggested that the number of traces produced were, in general, manageable, within a few tens of traces. However, for some cases, the number was several times larger. Moreover, traces can also be long in terms of number of statements.

We attempt to mitigate these two issues by introducing abstract traces. Informally, by choosing a subset of *relevant statements*, a trace can be summarized by showing a view that only contains the statements of the trace which are also *relevant* for the user.

The following example illustrates how, by choosing a subset of statements, the output can by drastically compacted and more easily interpreted.

Example 3. *Figure 4.1 corresponds to a harness that repeatedly calls function* `f` *in a loop that never terminates. Let us consider a software model checking attempt using bounded model checking and a loop iteration bound of 5. This technique, for this instance in particular, is interesting in that it is easy to understand the effect of the bound in the depth of the exploration and the system behaviors analyzed. We will show throughout this example how the execution report reflects the exploration in contrast to the intuitive understanding.*

The execution report for this inconclusive attempt contains 2^4 *different system executions in its frontier component: four iterations, the fifth is not reached, two branches in each iteration.*

For a preliminary assessment, the concrete paths taken within method `f()` *may not be quite relevant. Instead, the number of calls to* `f()` *more succinctly captures the actual depth of the exploration.*

If we simplify the traces by only considering the statements corresponding to calls to `f()`*, the 16 different system executions are all projected into a single*

```
1  int f(int x, int y) {
2    if (x) {
3      y = 2*y + 1;
4    } else {
5      y = 2*y;
6    }
7    return y;
8  }
9  int main() {
10   int y = 1;
11   while (1) {
12     y = f(nondet(), y);
13     assert(property(y));
14   }
15 }
```

Fig. 4.1: Large execution report due to branching.

abstract *trace: one with 4 consecutive calls to method* `f()`. △

Intuitively, abstract traces summarize system executions defined as sequences of statements of a program. More specifically, a system execution is projected into a subset of relevant statements (such as method calls).

The following definition captures this intuition leaving plenty of room for different types of abstractions:

Definition 6. *Given a set* Σ *of statements,* $\pi \in \Sigma^*$*. An abstract trace is the result of applying a total function* $f : \Sigma^* \to \Sigma^*$ *to* π*, such that* $|f(\pi)| \leq |\pi|$*, where* $|\pi|$ *is the number of statements in* π*.*

One example of an abstraction function is to project a sequence of statements over a sub-set of the statements, as shown below.

Definition 7. *A projection is defined as follows:*

$$\epsilon|_A = \epsilon \qquad (a.\pi)|_A = \begin{cases} a.\pi|_A & \text{if } a \in A \\ \pi|_A & \text{otherwise} \end{cases}$$

Given a set $A \subseteq \Sigma$, $\pi \in \Sigma^*$, we define a projection abstraction $\alpha_A(\pi)$ as $\alpha_A(\pi) = \pi|_A$.

In the case study that follows we will use the projection abstraction, α_A, instantiating A in the set of method calls, which will only show method calls, composed with the following abstraction function which filters statements executed within a stack depth exceeding a predefined threshold.

Definition 8. *Given a set Σ of statements, a set $C \subseteq \Sigma$ containing all statements corresponding to method calls, $R \subseteq \Sigma$ such that $R \cap C = \emptyset$, containing all statements corresponding to function exits, $\pi \in \Sigma^*$, we define a stack depth limit abstraction $stackDepth_{C,R}(\pi, d)$ as follows:*

$$stackDepth_{C,R}(\epsilon, d) = \epsilon$$

$$stackDepth_{C,R}(a.\pi, d) = \begin{cases} a.stackDepth_{C,R}(\pi, d-1) & \text{if } a \in C \wedge d > 0 \\ stackDepth_{C,R}(\pi, d-1) & \text{if } a \in C \wedge d \leq 0 \\ a.stackDepth_{C,R}(\pi, d+1) & \text{if } a \in R \wedge d > 0 \\ stackDepth_{C,R}(\pi, d+1) & \text{if } a \in R \wedge d \leq 0 \\ a.stackDepth_{C,R}(\pi, d) & \text{if } a \notin R \wedge a \notin C \wedge d > 0 \\ stackDepth_{C,R}(\pi, d) & \text{otherwise} \end{cases}$$

Example 3, as we mentioned before, showcases how abstract traces can summarize a rather verbose execution report. However, the abstract trace presented in that example can be interpreted in at least two ways: as existential, meaning there **exists** a concrete execution *in the execution report* that is captured by the abstract trace, or universal, meaning that **every** concrete execution captured by the abstract trace satisfies some property, such as having been analyzed.

We will proceed to discuss existential and universal abstract traces and reproduce the motivating instances and scenarios that we encountered while performing manual inspections of the execution reports.

Existential Abstract Traces

The output of model checker execution reports was defined in terms of system executions. As such, the length and number of traces was sometimes problematic when attempting to manually inspect the output.

To alleviate the issue, we propose to provide the user with a summarized view of the execution reports that, instead of containing the sets of traces S and F, as suggested in Definition 4 of the previous chapter, contains sets of abstract traces interpreted as existential evidence, as defined below:

Definition 9. *Given a set of frontier (safe) traces F (S) and an abstraction function f, the set of existential abstract frontier (safe) traces $F_{f,\exists}$ ($S_{f,\exists}$) is defined as:*

$\pi' \in F_{f,\exists}(S_{f,\exists})$ *iff* $\exists \pi \in F(S)$ *such that* $f(\pi) = \pi'$

Definition 9 formalizes the intuitive interpretation we adopt when interpreting abstract traces as existential.

To illustrate this definition, let us consider the instance `token_ring.04.c` from the SV-COMP benchmark as a case study. To incrementally analyze this instance, we will compose the two abstraction functions defined above into $f = \alpha_C \circ stackDepth_{C,R}$. The presentation of the example will follow the incremental inspection of the execution report and will be naturally guided by the insights extracted.

Inspecting frontier traces using existential abstract traces

The instance we will use in this example, `token_ring.04.c`, has 364 lines of code. One of the execution reports corresponding to a verification attempt for this instance yields 136 frontier traces in total, as shown in Table 3.3.

The total number of traces seems intimidating for manual inspection. However, a preliminary assessment of a few traces using the already existing visualization provided with CPAchecker seems to suggest that each trace is a slight variation of one another. For example, two different traces sometimes

share most of the statements with the exception of a few at the end and a different branch taken within an `if`-block nested deep in the call stack.

Intuitively, small differences in the traces might not make a significant difference in our assessment of the interrupted verification attempt. Moreover, it would be useful to capture the whole set of *similar* traces in some way. This is where abstract traces play a role.

By only showing method calls performed within the entry method of the program, we can drastically reduce the number of traces. In fact, only one trace remains, with only two method calls: `init_model` followed by `start_simulation`. This abstract trace should be interpreted *existentially*, that is, there *exists* at least one trace in the set of frontier traces that, when abstracted, corresponds to the abstract trace.

Without additional knowledge of the instance, this information is hard to interpret, therefore we reproduce the `main` method in Figure 4.2.

```
1 int main(void)
2 {
3   init_model();
4   start_simulation();
5   return 0;
6 }
```

Fig. 4.2: Entry method of `token_ring.04.c`.

With this information, and recalling the existential interpretation of abstract traces in this context, it is hard to assess whether the verification attempt will provide safety assurances. In particular, it would be necessary to understand what the method `start_simulation` involves and what parts of its possible behaviors have been exercised by the concrete traces in the execution report.

One possibility is to increase the stack depth that we used to generate the abstract trace, thereby increasing the level of detail. In that case, with a stack

depth of 2, once again we generate a single abstract trace:

```
1  init_model()
2  start_simulation()
3 -> update_channels() // 2nd level
4    init_threads()
5    fire_delta_events()
6    activate_threads()
7    reset_delta_events()
8    eval()
```

Fig. 4.3: Abstract trace generated with a stack depth of 2.

Once again, to put the abstract trace in context, we reproduce the method `start_simulation` in Figure 4.4, with the statements from the abstract trace (Figure 4.3) underlined.

A quick overview of the code snippet shows that the method contains a single `while` loop that iterates until some condition depending on the output of method `stop_simulation` holds. The abstract trace, with its statements underlined in Figure 4.4, indicates that the verification attempt failed to reach the end of the first iteration of the loop.

In principle, not reaching the end of the first loop iteration would not be a good sign, but given that this instance corresponds to automatically generated C code from a SystemC model, it is possible that the while loop can only be executed once.

Therefore, we need to increase the stack depth once again, this time to 3. At this point, producing abstract traces yields no reduction in the number of traces, that is, the number of abstract traces is exactly the same as the number of traces in the execution report. Since the abstract traces provide too much detail, preventing different traces from being grouped together, we now need to take a step in the opposite direction and make the abstraction coarser.

```
1  void start_simulation(void)
2  { int kernel_st ; int tmp ; int tmp___0 ; kernel_st = 0;
3    update_channels();
4    init_threads();
5    fire_delta_events();
6    activate_threads();
7    reset_delta_events();
8    while (1) {
9      while_6_continue:
10     kernel_st = 1;
11     eval();
12     kernel_st = 2;
13     update_channels();
14     kernel_st = 3;
15     fire_delta_events();
16     activate_threads();
17     reset_delta_events();
18     tmp = exists_runnable_thread();
19     if (tmp == 0) {
20       kernel_st = 4;
21       fire_time_events();
22       activate_threads();
23       reset_time_events();
24     }
25     tmp___0 = stop_simulation();
26     if (tmp___0) {
27       goto while_6_break;
28     }
29   }
30   while_6_break: /* CIL Label */ ;
31   return;
32 }
```

Fig. 4.4: Method `start_simulation`. Abstract trace underlined.

By comparing a few of the resulting abstract traces, what seems to occur is that they all share a common prefix and diverge towards the end, within the call to method `eval`. This hypothesis prompts us to inspect `eval` in detail.

A quick inspection of the code reproduced in Figure 4.5, corresponding to method `eval`, suffices to realize three relevant facts about the method. First, `eval` contains only one loop. Second, every iteration of the loop starts with a call to `exists_runnable_thread`. And third, methods inside the loop are apparently called with a high degree of non-determinism. To group traces, we could abstract away the method calls performed inside the `eval` loop and preserve only the call to `exists_runnable_thread`. That way, applying the abstraction function to traces with the same number of loop iterations would yield the same abstract trace.

More specifically, we remove the methods `master` and `transmit1` to `transmit4` from the set of method calls to be used in the abstraction function (as defined in Definition 7). Removing these methods makes the abstraction coarser and applying the abstraction function yields 5 unique abstract traces, ranging from 2 to 6 loop iterations within `eval`, as illustrated in Figure 4.6.

At this stage we have a detailed understanding of the depth of the verification attempt. However, interpreting this information requires knowledge about the specific use-case.

If the expectation for this system had been to perform full formal verification, then the information we gathered seems to suggest that the inconclusive verification attempt under analysis might not be thorough enough to increase our confidence in the correctness of the system. More specifically, for example, we know that our inconclusive verification attempt fails to provide any safety guarantees for actual system behaviors that can be exercised with as little as two iterations of the `eval` loop, as witnessed by the abstract trace shown in Figure 4.6.

On the other hand, if the expectation had been to complement other unsound verification efforts, such as an existing test suite or a previous verifica-

```
1  void eval(void) {
2    while (exists_runnable_thread()) {
3      if (m_st == 0 && nondet()) {
4        m_st = 1;
5        master();
6      }
7      if (t1_st == 0 && nondet()) {
8        t1_st = 1;
9        transmit1();
10     }
11     if (t2_st == 0 && nondet()) {
12       t2_st = 1;
13       transmit2();
14     }
15     if (t3_st == 0 && nondet()) {
16       t3_st = 1;
17       transmit3();
18     }
19     if (t4_st == 0 && nondet()) {
20       t4_st = 1;
21       transmit4();
22     }
23   }
24 }
```

Fig. 4.5: Method `eval`.

```
 init_model()
 start_simulation()
-> update_channels() // 2nd level
   init_threads()
   fire_delta_events()
   activate_threads()
->   is_master_triggered() // 3rd level
     is_transmit1_triggered()
     is_transmit2_triggered()
     is_transmit3_triggered()
     is_transmit4_triggered()
   reset_delta_events()
   eval()
->   exists_runnable_thread()
     exists_runnable_thread()
```

Fig. 4.6: Abstract trace generated with a stack depth of 3, ignoring `master` and `transmit1` to `transmit4` showing two loop iterations inside `eval`.

tion attempt using a bounded technique, the existence of traces reaching as deep as 7 iterations into the loop in `eval` could provide evidence of value.

Implementation and Performance

The generation of abstract traces can be implemented as a post-processing step once the model checker execution report of interest has been generated.

Our implementation is optimized for the abstraction considered in the example above but it would be possible to extend the implementation with other abstraction functions. We provide a command-line user interface that exposes the abstraction functions through a number of operations. To support abstraction function composition, the interface keeps track of a set of current traces, which is updated with the result of the different operations. The functionality will become clearer as we discuss the different commands.

One detail about the abstraction functions $stackDepth_{C,R}(\pi, d)$ and α_A is that they are parametric on the sets C, R and A. However, it is possible to initialize C, R and A to sensible defaults to reduce manual effort. Method calls and return statements are specifically label by CPAchecker in its output, therefore computing C and R requires no additional interaction involving the user. The set A in our case is also a subset of C, the set of method call statements, and can be initialized with C so as to reduce manual effort.

As such, we expose the abstraction function $stackDepth_{C,R}(\pi, d)$ through the command `collapse_depth d`, which applies $stackDepth_{C,R}$ to each trace in the current set (possibly the result of previously applying another abstraction function), initializing its second parameter to `d`.

Analogously, `collapse_non_methods` applies α_C to each trace, where C is the set of method call statements.

In the example shown above, we excluded 5 methods from the set C to be able to group more traces together. We achieve this functionality with `collapse_method m`, which applies $\alpha_{C \setminus \{c | c \text{ is a method call to method } \texttt{m}\}}$ to each of the traces in the current set. It is important to note that applying `collapse_method` twice, with methods `m1` and `m2`, will remove calls to both methods, since the second operation will take as input the result of the first one.

Finally, applying these abstraction functions might result in a number of identical abstract traces. We provide the command `group_traces` based on the pseudo-code in Algorithm 2.

input : A set of traces (sequences of statements) T, an abstraction function f.

output: A set of traces (sequences of statements)

```
1 AT := ∅
2 for each π ∈ T do
3 |   if f(π) ∉ AT then
4 |   |   AT := AT ∪ {f(π)}
```

Algorithm 2: Generation of abstract traces

Algorithm 2 iterates over a set of traces T and applies the abstraction

function f to each of them. Since two traces $\pi, \pi' \in T$ could correspond to the same abstract trace in AT, that is $f(\pi) = f(\pi')$, we add abstract traces to the result in line 4, only after checking in line 3 that it is not equal to any abstract trace previously generated.

The complexity of the algorithm corresponds to $|T|$ times the complexity of executing line 3, since line 4 can be performed in constant time by representing AT using a linked list. We implement the check in line 3 by applying f and then performing a string equality check against each element in AT, therefore, the complexity of this check is $cost_f(maxLength) + |AT| \times maxLength$, where $cost_f$ is an upper bound to the time complexity of f as a function of the size of its input and $maxLength = max_{\pi' \in T}|\pi'|$. An upper bound to the complexity in terms of T is, therefore, $O(|T|^2 \times maxLength + |T| \times cost_f(maxLength))$.

We also implemented the abstraction function we used to analyze the example, $\alpha_C \circ stackDepth_{C,R}$, with a linear time complexity. The running time is negligible with respect to the execution report generation time. In our experience exploring SV-COMP benchmark instances, the set of abstract traces is generated instantaneously, as expected.

Experience with Benchmark Instances

We used existential abstract traces to incrementally explore the execution reports of SV-COMP [7] instance. The example discussed above reflects the methodology we used.

By inspecting the abstract traces using incrementally higher stack depths we were pointed to very few and highly specific sections of the code. Since we did not have previous knowledge about the instances, this served the purpose of directing our learning process, making it highly focused and efficient.

In all cases, we were able to get a clear picture of the depth of the execution. This includes both techniques, that is, when the original verification technique was explicit value analysis and also when using lazy predicate abstraction.

Our experience analyzing SV-COMP instances, although anecdotal due to

its exploratory nature, provides encouraging preliminary evidence related to our guiding research question: whether model checker execution reports are understandable.

Universal Abstract Traces

The example above illustrates a detailed analysis of the depth of an inconclusive verification attempt using lazy predicate abstraction. As we mentioned, the quality assessment of the inconclusive verification result will necessarily depend on the intended use. In particular, while the **existence** of *shallow* traces in the frontier can be convincing evidence of insufficient exploration, the existential interpretation of abstract traces seems severely limited in its capacity to provide positive evidence highlighting the value of the inconclusive results and its associated safety assurances. This section will start with an example illustrating some shortcomings of the interpretation of abstract traces that we have adopted so far, motivating an alternative use of abstract traces: interpreting abstract traces universally instead of existentially. We will continue our presentation by introducing the notion of relative soundness which we will illustrate with an example. Subsequently, we will proceed to present the formal core of this section, a detailed account of how to compute relative soundness for the abstraction functions we have been using. Moreover, we will discuss why our approach soundly computes the notion of relative soundness.

Let us start by revisiting Example 3. The code under verification corresponds to a harness that called function `f()` inside a loop, as shown in Figure 4.1. The execution report, after applying the abstraction function, only showed a single abstract trace, composed of 4 consecutive calls to method `f()`. Since model checker execution reports in principle offer no completeness guarantees, the absence of shorter traces within the set F, traces known to reach the exploration frontier, does not imply their non-existence. Therefore, had the user not had a clear understanding of the underlying verification technique that reached the inconclusive outcome, he or she could ask the following ques-

tion: have all behaviors composed of less than 4 calls to `f()` been analyzed?

The question just posed is, in fact, closely related to k-unsoundness. By Definition 5, a partial software model check is k-unsound when a trace was not analyzed but it would have been checked if the underlying verification technique had been bounded model checking with a bound of k loop iterations.

Therefore, in the example we are discussing, the partial software model check has analyzed all behaviors composed of less than 4 calls to `f()` if and only if the partial software model check is **not** k-unsound.

The fundamental link between the example and k-unsoundness is, precisely, that the underlying verification technique used was bounded model checking. The question, although posed for a specific example, underscores a shortcoming of model checker execution reports. Assuming we defined k-soundness such that it held if and only if k-unsoundness did not, we are unable to prove the property of k-soundness for a particular partial software model check. Furthermore, this relative soundness criterion might be too coarse for techniques other than bounded model checking. To overcome this limitation, we will put forth the definition of relative soundness. We will define that a partial software model check is sound relative to an abstract trace when all possible concretizations of the abstract trace have been analyzed by the partial software model check.

Definition 10 (Relative Soundness)**.** *Given a program $\mathcal{P}$ where Σ is its set of statements, an abstract trace α, an abstraction function f and a partial model check captured by the predicate analyzed?, the partial software model check is sound relative to α and f if and only if $\forall \pi \in \Sigma^*. isPrefix(f(\pi), \alpha) \wedge feasible_{\mathcal{P}}(\pi) \Rightarrow analyzed?(\pi)$. The predicate $isPrefix(\pi, \pi')$ holds if and only if π is a prefix of π'. The predicate $feasible_{\mathcal{P}}(\pi)$ holds if and only if the sequence of statements π is an actual execution of the program $\mathcal{P}$.*

Definition 10 allows us to compare partial software model checks with respect to abstract traces, by considering all of the abstract trace's executable concretizations.

Let us consider a benchmark instance and explore what insights we can extract using universal traces.

```
1  void insert(int key, int v[], int i)
2  {
3    while((i>=0) && (v[i]>key)) {
4       v[i+1] = v[i];
5       i = i - 1;
6    }
7    v[i+1] = key;
8  }
9  void check(int v[], int j)
10 {
11   int k;
12   for (k=1;k<j;k++)
13     assert(v[k-1]<=v[k]);
14 }
15 int main()
16 {
17    unsigned int SIZE=nondet_uint();
18    int i, j, key;
19    int v[SIZE];
20    for (j=1;j<SIZE;j++) {
21       key = v[j];
22       i = j - 1;
23       insert(key, v, i);
24       check(v, j);
25   }
26   return 0;
27 }
```

Fig. 4.7: Insertion sort with added check.

Example 4. *We will examine the instance* `insertion_sort_true-unreach-call_-`

`true-termination.i`. *As the name suggests, it is an implementation of insertion sort with a final check at the end of the execution that verifies that elements are sorted. We slightly modified the code: we check after every insertion that the appropriate portion of the array remains sorted, as shown in Figure 4.7. The instance as a whole can be understood as a function* `insert` *that we are interested in analyzing and the harness that provides a closed environment and exercises the function.*

We attempted an initial verification attempt with a time limit of 60 s *using value analysis as the underlying technique.*

We generated an execution report and, to inspect the frontier traces, we used the same abstraction functions discussed above, which produce abstract traces containing only method calls performed within the `main` *method and not deeper within the call stack.*

Only 2 abstract traces remain, which are identical except for a final call to `check` *in one of them. The abstract traces are shown in Figure 4.8, with the final* `check` *statement of the second abstract trace included as a comment.*

The abstract traces shown in Figure 4.8 correspond to 10 iterations of the main loop of the harness. The execution report might lead us to hypothesize that all behaviors captured by less than 10 iterations of the main loop have been analyzed. However, as we commented before, since execution reports do not offer completeness guarantees, we cannot soundly infer anything from the lack of shorter frontier traces.

To overcome this limitation, we resort to the notion of relative soundness, as posed in Definition 10. We will discuss how to effectively check relative soundness after this example, but assuming such a procedure exists, we move forward and confirm that, indeed, the partial software model check is sound relative to the abstract trace.

Of course, the verification attempt remains inconclusive, but the guarantees at this point subsume non-trivial testing efforts. For example, testing for corner cases such as inserting in arrays with zero elements, inserting in arrays

with repeated elements and inserting an element already contained within the array are all behaviors that have been analyzed during the partial model check.

In particular, any test using an input array with less than 10 elements and checking for the same safety property will necessarily be redundant.

One important observation about this insight is that it is fundamentally different to a bounded model checking attempt with depth 10. One reason why is that the user can be oblivious to the specific implementation details of method `insert` *and the harness itself.*

For this particular instance, the notion of relative soundness allows us to significantly increase our confidence in the system with respect to the property being verified. △

Algorithm

Given that we leverage CPAchecker as our underlying infrastructure, we will resort to the conceptual framework of configurable software verification [13] to formalize our approach, as we did for model checker execution reports.

In this case, similarly, we augment the abstract domain by adding yet another component to a tuple combining different elements. In the case of execution reports, we used a tuple where some of the components were defined by the underlying analysis techniques and one component was an assumption automaton node. For example, in the case of predicate abstraction, several components are combined, including the location in the code, the call stack and a predicate, which we augmented with an assumption automaton node.

Our approach to computing relative soundness consists of storing information relative to the abstract trace generated leading up to a specific node in the abstract reachability tree. In order to use the configurable software verification framework, we restrict the types of abstraction function we attempt to handle. We only tackle abstraction functions that can be defined in terms of a transducer. By imposing this restriction, we can compose the abstraction function transducer derived from f, with an automaton derived

from an abstract trace α, with f and α as referred to in Definition 10. This way, the approach we used with assumption automata can be easily adapted to this setting, since the composition of the abstract trace automaton and the abstraction function transducer is itself an automaton.

We will start the presentation by first including all the formal definitions. Then we will instantiate those definitions with the specific abstraction functions we use and exemplify some of the definitions. Although we discuss the definitions for the abstraction functions we use the approach is potentially applicable to other abstraction functions as well.

We will reproduce some of the definitions already presented in Chapter 3 and taken from the configurable software verification framework [13]. A configurable program analysis (CPA) $\mathbb{P} = (D_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \texttt{merge}_{\mathbb{P}}, \texttt{stop}_{\mathbb{P}})$ is defined by an abstract domain $D_{\mathbb{P}}$, a transfer relation $\rightsquigarrow_{\mathbb{P}}$, a merge operator $\texttt{merge}_{\mathbb{P}}$, and a termination check $\texttt{stop}_{\mathbb{P}}$.

Such a CPA $\mathbb{P}$ will be executed with Algorithm 1 from Chapter 3.

Our CPA $\mathbb{P}(A, B) = (D_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \texttt{merge}_{\mathbb{P}}, \texttt{stop}_{\mathbb{P}})$ will be parametric on assumption automaton A and bound B represented as an automaton as well. $D_{\mathbb{P}}$ will be based on the flat lattice for the set of all tuples (a, b) in the cross product between the set of states of A and the set of states of B. For the transfer relation, $(a, b) \overset{stm}{\rightsquigarrow}_{\mathbb{P}} (a', b')$ if there exists a transition labeled stm from state a to state a' of the assumption automaton A and also there exists a transition labeled stm from state b to state b' of the automaton B. Otherwise, $(a, b) \overset{stm}{\rightsquigarrow}_{\mathbb{P}} \bot$.

Finally, $\texttt{merge}_{\mathbb{P}} = \texttt{merge}^{sep}$ and $\texttt{stop}_{\mathbb{P}} = \texttt{stop}^{sep}$, where $\texttt{merge}^{sep}(e, e') = e'$ and $\texttt{stop}^{sep}(e, R) = (\exists e' \in R : e \sqsubseteq e')$. The proposed definition for these last operators can be tweaked to improve performance at the cost of precision. We did not experiment with other definitions, but they are certainly possible and examples of the trade-offs are discussed in the original presentation of the configurable software verification framework [13].

The main aspect of the definition that requires further attention, before we

explain how it leads to computing relative soundness, is how we generate an automaton B that bounds the exploration space from an abstraction function and an abstract trace.

In order to ensure the soundness of our analysis, a few additional properties about automaton B will be required. Our analysis can be considered sound if and only if the non-existence of a counterexample within the bound imposed by automaton B implies the partial software model check was relatively sound with respect to abstraction function f and abstract trace α.

Definition 11 reproduces the usual definition of automata, without requiring its set of states to be finite.

Definition 11. *A deterministic automaton is a tuple* (Q, Σ, I, δ) *such that:*

- Q *is a set of nodes*
- Σ *is the alphabet of the automaton*
- $I \subseteq Q$ *is a set of initial nodes*
- $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ *is the transition function*

Moreover, we reproduce the standard definition for its extended transition function:

Definition 12. *Given a transition function* $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$*, the extended transition function* $\hat{\delta}$ *is the smallest set such that:*

- $\delta \subseteq \hat{\delta}$
- $\delta(q, \epsilon) = q$ *for all* $q \in Q$
- $\hat{\delta}(q, \pi) = q'$ *and* $\delta(q', s) = q''$ *imply* $\hat{\delta}(q, \pi \cdot s) = q''$

In terms of the automaton B and the abstract domain derived from it, to guarantee the soundness of our approach, we will require the following property holds, which captures the intuition that B, as a bound, does not exclude concretizations of α.

Property 7. *Given a system execution π, an abstraction function f, an abstract trace α and an automaton $B = (Q, \Sigma, \delta, q_0, F)$, if the following property holds for B, then our approach to computing relative soundness is in itself sound:*

$isPrefix(f(\pi), \alpha) \implies \hat{\delta}(q_0, \pi)$ *is defined*

Given our definition of transition relation $\rightsquigarrow_{\mathbb{P}}$, the analysis will only be pruned by B if no successor is found for some state $b \in Q$ and some symbol $stm \in \Sigma$. Furthermore, assumption automata never prune the abstract reachability tree. Therefore, Property 7 guarantees that any trace π that satisfies $isPrefix(f(\pi), \alpha)$ necessarily remains within the bound defined by B.

Since Definition 10, capturing relative soundness, requires the existence of a trace satisfying a number of conditions, including $isPrefix(f(\pi), \alpha)$, for a software model check to be unsound relative to f and α, as long as all such traces are within the bounds imposed, the analysis is sound. As such, Property 7 implies our approach is sound.

We will now focus on how we encode the abstraction functions we use as transducers, how we generate an automaton that captures the abstract trace and how the composition satisfies Property 7.

As we anticipated, we encode the abstraction function as a transducer and the abstract trace as an automaton. Therefore, the composition of the transducer and the automaton results in the desired automaton B.

The specific abstraction function we use, only showing method calls within less than a predetermined stack depth, can be encoded as a transducer in a straightforward way.

Let us recall the abstraction functions we are using. The first function to be applied, $stackDepth_{C,R}$, takes as input the sequence of statements, π, and the desired stack depth, d. The result of the function, $stackDepth_{C,R}(\pi, d)$, will consist of the subsequence composed of all statements at most d levels deep in the call stack.

The function can then be represented as a transducer with one state for

each possible depth in the call stack, as depicted in Figure 4.9. The initial parameter d will determine a single initial state and $d-1$ subsequent nodes which copy input symbols to the output. The nodes copying symbols are labeled 1 to d in Figure 4.9. In contrast, nodes labeled 0 or negative discard the input symbols, since these would be statements executed deeper than d levels within the call stack.

More formally, we will reproduce the standard definition of a transducer, without the usual restriction of finiteness imposed over the set of states.

Definition 13. *A transducer is a tuple* $(Q, \Sigma, \Gamma, I, \delta)$ *such that:*

- Q *is a set of nodes*
- Σ *is the input alphabet of the transducer*
- Γ *is the output alphabet of the transducer*
- $I \subseteq Q$ *is a set of initial nodes*
- $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times Q$ *is the transition relation, with* ϵ *being the empty string*

The definition of the extended transition relation for transducer is analogous to the one for automata, reproduced in Definition 12.

Using Definition 13, the transducer described above for the abstraction function will be parametric on sets $C, R \subseteq \Sigma$ and defined as $T_d(C, R) = (Q_d, \Sigma, \Sigma, I_d, \delta_d)$ where sets C and R corresponds to method calls and return statements respectively and Σ is the set of all statements. The set of states Q_d, as anticipated, will contain d nodes spanning q_1 to q_d which will copy input symbols to the output in their outgoing transitions and all nodes q_i with $i \leq 0$ which discard the input symbols. As such, $Q_d = \{q_i | i \in \mathbb{Z}.\, i \leq d\}$. The set of initial states I_d contains a single element, q_d. Finally the transition relation is defined as follows, closely resembling Definition 8.

For states $q_i \in Q$, statement $stm \in \Sigma$:

- $(q_i, stm, stm, q_{i-1}) \in \delta_d$ if and only if $stm \in C \wedge i > 0$
- $(q_i, stm, \epsilon, q_{i-1}) \in \delta_d$ if and only if $stm \in C \wedge i \leq 0$
- $(q_i, stm, stm, q_{i+1}) \in \delta_d$ if and only if $stm \in R \wedge d > i > 0$
- $(q_i, stm, \epsilon, q_{i+1}) \in \delta_d$ if and only if $stm \in R \wedge d > i \leq 0$
- $(q_i, stm, stm, q_i) \in \delta_d$ if and only if $stm \notin R \wedge stm \notin C \wedge i > 0$
- $(q_i, stm, \epsilon, q_i) \in \delta_d$ if and only if $stm \notin R \wedge stm \notin C \wedge i \leq 0$
- $(q_i, \epsilon, \epsilon, q_i) \in \delta_d$

One caveat of this definition is that the set Q_d is clearly infinite. However, the conditions accompanying each transition are essentially predicates over variable i, since the sets C and R remain constant throughout the analysis. For our implementation, since the language provided by CPAchecker to specify automata, based on the BLAST query language [8], supports integer variables, this can be expressed with a single state, parametric on the value of the variable. In general, the set C can be determined syntactically whereas the set R can be computed from the control flow graph of each method. For our implementation these two sets are already precomputed by CPAchecker and available during the analysis.

The transducer for abstraction function α_C is even simpler. A single node suffices and two types of transitions exist. For symbols contained in set C the transitions reproduce the input symbol in the output tape. For all other symbols, the transitions simply discard the input symbols.

More formally, the transducer for α_C will be $T_{\alpha_C} = (Q_{\alpha_C}, \Sigma, C, I_{\alpha_C}, \delta_{\alpha_C})$. The set of states Q_{α_C} will contain a single node q_0. The set of initial states I_{α_C} contains the only node in Q_{α_C}, q_0. Finally the transition relation is defined as follows.

For state $q_0 \in Q$, statement $stm \in \Sigma$:

- $(q_0, stm, stm, q_0) \in \delta_{\alpha_C}$ if and only if $stm \in C$

- $(q_0, stm, \epsilon, q_0) \in \delta_{\alpha_C}$ if and only if $stm \notin C$
- $(q_0, \epsilon, \epsilon, q_0) \in \delta_{\alpha_C}$

Finally, the automaton capturing the abstract trace serves the purpose of bounding the search space, which was the motivation of relative soundness, as determined in Definition 10. In order to enforce the bound, the automaton will display a `backtrack` state, as shown in Figure 4.10. If the `backtrack` state is reached during the abstract reachability tree construction, that branch of the tree will be ignored, because the definition of $\rightsquigarrow_{\mathbb{P}}$ for $\mathbb{P}(A, B)$ will only allow a transition to $\bot$ when one of the components has no successors, and the algorithm will continue expanding other nodes.

Let us now see what the automaton would look like for a small example. The code snippet in Figure 4.11 shows a harness exercising methods `m1` and `m2`. Let us also consider the abstract trace $\pi = [\texttt{m1()}, \texttt{m1()}, \texttt{m1()}]$ and the abstraction function $\alpha_{\{\texttt{m1()},\texttt{m2()}\}}$, which only shows method calls to `m1` or `m2`.

The automaton for the abstract trace is depicted in Figure 4.10. Using the abstract trace π, behaviors involving the execution of `m2()` in the first loop iteration will not be analyzed. Since the automaton state will be part of the composite abstract domain used in the abstract reachability tree construction, any path traversing the call to `m2()` will lead to the `backtrack` state. The `backtrack` state, in turn, has no outgoing edges. Since one of the components of the composite abstract domain element has no outgoing edges the abstract reachability tree construction algorithm will trim that branch without analyzing it, due to the definition of $\rightsquigarrow_{\mathbb{P}}$ for $\mathbb{P}(A, B)$.

To formalize this, we will resort to Definition 11, capturing the components of an automaton.

The abstract trace π can be encoded as the automaton $A = (Q_a, \Sigma, I_a, \delta_a)$, using Definition 11. The set of states will be $Q_a = \{\texttt{backtrack}\} \cup \{q_i | 0 \leq i \leq |\pi|\}$, where $|\pi|$ denotes the length of the trace π. The set of initial states will contain a single state q_0, therefore $I = \{q_0\}$.

Definition 14. *The transition function will be defined as follows, where π_i denotes the i-th symbol in π, counting from 0:*

$$\begin{aligned}\delta(q_i, stm) &= \begin{cases} q_{i+1} & \text{if } i < |\pi| \wedge stm = \pi_i \\ \texttt{backtrack} & \text{otherwise} \end{cases} \\ \delta(q_i, \epsilon) &= q_i \end{aligned}$$

Finally, $\delta(\texttt{backtrack}, stm)$ is undefined for all symbols $stm \in \Sigma$ and $\delta(\texttt{backtrack}, \epsilon)$ is undefined too.

Now that we have defined the transducers and the automaton for the abstract trace, we need to define a composition between them.

Definition 15. *The result of the composition of transducers $A = (Q_A, \Sigma_A, \Gamma_A, I_A, \delta_A)$ and $B = (Q_B, \Sigma_B, \Gamma_B, I_B, \delta_B)$, with $\Gamma_A \subseteq \Sigma_B$, is a transducer $C = (Q_C, \Sigma_A, \Gamma_B, I_C, \delta_B)$. The set of states Q_C is defined as the product of Q_A and Q_B. That is $Q_C = Q_A \times Q_B$. The set of initial states, analogously, corresponds to pairs of states in the corresponding initial sets, that is, $I_C = \{(q_a, q_b) | q_a \in I_A \wedge q_b \in I_B\}$. Finally, the transition relation is defined such that $((q_a, q_b), stm_{in}, stm_{out}, (q'_a, q'_b)) \in \delta_C$ if and only if there exist $s \in (\Gamma_A \cup \{\epsilon\}) \subseteq (\Sigma_B \cup \{\epsilon\})$, such that $(q_a, stm_{in}, s, q'_a) \in \delta_A$ and $(q_b, s, stm_{out}, q'_b) \in \delta_B$.*

Analogously, we proceed to define the composition of an automaton and a transducer, resulting in an automaton.

Definition 16. *The result of the composition of a transducer $T = (Q_T, \Sigma_T, \Gamma_T, I_T, \delta_T)$ and an automaton $A = (Q_A, \Sigma_A, I_A, \delta_A)$, with $\Gamma_T \subseteq \Sigma_A$, is an automaton $C = (Q_C, \Sigma_T, I_C, \delta_C)$. The set of states Q_C is defined as the product of Q_T and Q_A. That is $Q_C = Q_T \times Q_A$. The set of initial states, analogously, corresponds to pairs of states in the corresponding initial sets, that is, $I_C = \{(q_t, q_a) | q_t \in I_T \wedge q_a \in I_A\}$. Finally, the transition relation is defined such that $((q_t, q_a), stm_{in}, (q'_t, q'_a)) \in \delta_C$ if and only if there exists $s \in (\Gamma_T \cup \{\epsilon\}) \subseteq (\Sigma_A \cup \{\epsilon\})$, such that $(q_t, stm_{in}, s, q'_t) \in \delta_T$ and $(q_a, s, q'_a) \in \delta_A$.*

Having encoded the abstraction functions as transducers, the abstract trace as an automaton and defined the composition of both, we can show that

Property 7 holds for the composition. Property 7 always holds when the extended transition function for the composition, $\hat{\delta}$, is defined. We will first see that the extended transition function is always defined for the transducers.

From Definition 6, we know abstraction functions are total. As such, if a transducer $T = (Q_T, \Sigma_T, \Gamma_T, I_T, \delta_T)$ encodes an abstraction function, the extended transition relation $\hat{\delta}_T$ will necessarily satisfy that for all $\pi \in \Sigma_T^*$ there exist $\alpha \in (\Gamma_T \cup \{\epsilon\})^*, q \in Q_T, q_0 \in I_T$ such that $(q_0, \pi, \alpha, q) \in \hat{\delta}_T$. In particular, assuming we encoded function f in the transducer correctly, $\alpha = f(\pi)$.

This holds for the composition as well. Given $T_A = (Q_A, \Sigma_A, \Gamma_A, I_A, \delta_A)$ and $T_B = (Q_B, \Sigma_B, \Gamma_B, I_B, \delta_B)$, capturing abstraction functions f_A and f_B respectively, such that $\Gamma_A \subseteq \Sigma_B$, the composition is $T_C = (Q_C, \Sigma_A, \Gamma_B, I_C, \delta_C)$. Since we know that:

1. for all $\pi \in \Sigma_A^*$ we know that $(q_{A_0}, \pi, f_A(\pi), q_a) \in \delta_A$ for some $q_{A_0} \in I_A, q_a \in Q_A$ with $f_A(\pi) \in (\Gamma_A \cup \{\epsilon\}) \subseteq (\Sigma_B \cup \{\epsilon\})$
2. for all $\pi' \in \Sigma_B^*$ we know that $(q_{B_0}, \pi', f_B(\pi'), q_b) \in \delta_B$ for some $q_{B_0} \in I_B, q_b \in Q_B$
3. instantiating π' in item 2 as $\pi' = f_A(\pi)$ with $\pi \in \Sigma_A^*$ we have that $(q_{B_0}, f_A(\pi), f_B(f_A(\pi)), q_b) \in \delta_B$ for some $q_{B_0} \in I_B, q_b \in Q_B$
4. from item 3 and using Definition 15 we know that $(q_0, \pi, f_B(f_A(\pi), q) \in \delta_C)$ for some $q_0 \in I_C, q \in Q_C$.

Having established that the extended transition function is always defined for the transducers, let us now focus on the automaton $A = (Q_A, \Sigma_A, q_0, \sigma_A)$ derived from the abstract trace α. Since $\hat{\delta}_A(q_0, \pi)$ is only undefined when the `backtrack` state is reached, Property 7 could only be violated for traces reaching that state. Let us assume that the property could be violated. Then, necessarily there exists $\pi \in \Sigma_A^*$ such that $isPrefix(\pi, \alpha) \wedge \hat{\delta}_A(q_0, \pi)$ is undefined.

In terms of the automaton derived from α, this would mean that there exist $\pi' \in \Sigma_A^*$ such that $isPrefix(\pi', \alpha) \wedge \hat{\delta}_A(q_0, \pi') =$ `backtrack`. But the

following property also holds, $\hat{\delta}_A(q_0, \pi')$ is defined $\wedge\, \hat{\delta}_A(q_0, \pi') = q_i \leftrightarrow \pi' = isPrefix(\pi', \alpha) \wedge |\pi'| = i$.

We will prove this by induction on the length of π'. The base case is straightforward: $|\pi'| = 0$, then the property holds, since $isPrefix(\pi', \alpha)$ and $\hat{\delta}_A)q_0, \epsilon) = q_0$ both also hold. Now let us consider the inductive case, in which $|\pi'| = i > 0$ and the inductive hypothesis holds for $0 \leq j < i$. Without loss of generality, $\pi' = \pi \cdot s$ for some $\pi \in \Sigma_A^*, s \in \Sigma_A$. Let us consider both possible cases.

- Case $isPrefix(\pi', \alpha)$:

  Knowing that $|\pi'| = i$, we need to show that $\hat{\delta}_A(\pi', \alpha)$ is defined $\wedge$ $\hat{\delta}_A(\pi', \alpha) = q_i$. Since $isPrefix(\pi', \alpha)$ holds, then $isPrefix(\pi, \alpha)$ must also holds, because π is itself a prefix of π'. Moreover, since both $isPrefix(\pi, \alpha)$ and $|\pi| = i - 1$ the inductive hypothesis implies that $\hat{\delta}_A(q_0, \pi) = q_{i-1}$ holds. Using Definition 14, $\hat{\delta}_A(q_0, \pi')$ is defined $\wedge$ $\hat{\delta}_A(q_0, \pi') = q_i$, therefore the property holds also in the inductive case when $isPrefix(\pi', \alpha)$ holds.

- Case $isPrefix(\pi', \alpha)$ does not hold:
  Then, considering $\pi' = \pi \cdot s$, two options arise, depending on whether $isPrefix(\pi, \alpha)$ holds.

  In both cases, we want to show that either $\hat{\delta}_A(q_0, \pi')$ is undefined or $\hat{\delta}_A(q_0, \pi') \neq q_i$.

  1. Case $isPrefix(\pi, \alpha)$ holds:
     Since $|\pi| = i-1$ and the inductive hypothesis holds, then $\hat{\delta}_A(q_0, \pi) = q_{i-1}$. Combining these assumptions with Definition 14, we can conclude that $\hat{\delta}_A(q_0, \pi') = \texttt{backtrack} \neq q_i$, since $s \neq \pi'_{i-1}$ because otherwise π' would be a prefix of α too.
  2. Case $isPrefix(\pi, \alpha)$ does not hold:
     In this case, due to the inductive hypothesis, we must consider two

cases. First option is that $\hat{\delta}_A(q_0, \pi)$ is not defined, in which case $\hat{\delta}_A(q_0, \pi')$ is not defined either, as we wanted to show. Finally, the last option is that $\hat{\delta}_A(q_0, \pi) \neq q_i$. In this case, either $\hat{\delta}_A(q_0, \pi) =$ `backtrack`, in which case $\hat{\delta}_A(q_0, \pi')$ is undefined, or $\hat{\delta}_A(q_0, \pi) = q_j$ with $j \neq i - 1$. If the latter case holds, using Definition 14, either $\hat{\delta}_A(q_0, \pi') =$ `backtrack` or $\hat{\delta}_A(q_0, \pi') = q_{j+1}$ with $j + 1 \neq i$. In both cases, $\hat{\delta}_A(q_0, \pi') \neq q_i$, which is the desired property.

Use cases enabled by universal traces

Although the example discussed above highlights the possibility of using universal traces to more easily convey safety assurances, a number of other possible use cases exist.

Let us recall the original motivation behind the notion of relative soundness, presented in Definition 10, which was to provide a succinct, yet fine grained representation of the bounds of a partial software model check.

A trivial use case of universal traces is to decouple the verification harness from the bounded state space that is tractable by a particular tool. There are two advantages to this. First, it makes verification harnesses simpler by removing accidental bounds imposed by the limitations of underlying tools. This way, assumptions about the environment in which a system is executed remain isolated from unsound assumptions made as a trade-off for performance. Second, it enables users to use different verification techniques and tweak the bounds as appropriate to each tool.

An additional advantage of these bounds, due to the use of abstract traces, is that they can be robust with respect to changes in the implementation. This robustness makes them suitable to calculate mutation scores, which has yet to be tackled for anything other inherently bounded techniques [50]. The key enabling feature, in this case, is the ability to reproduce an exploration for slightly different variants of the code, which other representations, such as assumption automata, did not allow.

Sound verification has been used to generalize testing [70, 78], but those approaches are not applicable to inconclusive verification outcomes. Well understood, through mutation analysis, and robust, with respect to minor syntactic differences, bounds are well-suited for ongoing partial verification efforts. That is, using verification to complement regression testing after each change in the code. The key enabling factor, once again, is the robustness to syntactic changes.

The encoding of these bounds in terms of an abstract domain makes it compatible with a wide range of techniques encompassing, at the very least, the whole family of those based on abstract reachability trees. As such, it is possible to compare the performance of different verification techniques for a particular instance and bound. This could be particularly useful in the context of partial regression verification, where having prompt feedback significantly increases usability.

Lastly, by understanding the failure detection capabilities of partial software model checks, unsound optimizations [66] can be considered far more broadly, since their impact can be assessed and conveyed on a case-by-case basis. For example, analyses frequently assume the absence of integer overflow to improve efficiency and the impact of such optimizations in the failure detecting capabilities is far from clear. The empirical analysis of violations of this type of unsound assumptions [29] has shown that such violations exist in a diverse set of applications, which naturally calls for methods and techniques to quantify the effect of such assumptions in the failure detection capability of partial software model checks. In our example analyzing the method `insert`, assuming the absence of integer overflow may yield a significant performance improvement without affecting the failure detection capability, since within the behaviors analyzed integer overflow is indeed impossible. In other cases, the situation might be the opposite and the trade-off disproportionately costly.

Related Work

There is substantial existing work proposing abstractions and techniques for dealing with a diverse set of behaviors in the context of software development and also, more specifically, in verification. For example, parameterized unit tests [78] can serve as summaries to symbolic execution techniques, capturing a set of test cases. Program query languages [68, 48] have been used to specify patterns that match certain sequences of events, capturing a possibly large number of such sequences. Code navigation and querying tools [57] provide support for exploring and understanding code.

The term *soundiness* [66] was coined to encompass the usual compromise of soundness when handling specific complex languages features. Researchers in favor of soundiness, as opposed to actual soundness, highlight the need to make these compromises more visible. The publication presenting the notion of soundiness [66], however, suggests a different approach to increasing visibility: to measure how prevalent these ignored language features are within a specific benchmark. One example following this line of research is the work by Christakis et al. [29]. This work [29] consists of an evaluation of the impact of deliberate unsoundness in static program analyzers. The authors both put forth a methodology to evaluate the impact of unsoundness in practice and also apply it to applications from different domains.

Our use of abstract traces in some way resembles program slicing [84, 79]. However, both the underlying techniques, applications and interpretations of the slices are entirely unrelated to our work.

Although our work in this chapter aims to deal with groups of traces, there is also work on handling single counterexamples [52, 51, 58, 25] produced by model checkers. The two lines of work, although similar in motivation, tackle fundamentally different issues. In many cases, the work on explaining or augmenting single counterexamples can complement our work dealing with groups of traces.

Summary

Throughout this chapter we explored two extensions of model checker execution reports that can significantly improve their usability. Both extension are based on using abstract traces instead of concrete ones. But the use of different interpretations for the abstract traces enables two entirely different interactions yielding complementary insights. With the aid of the concepts of k-unsoundness and relative soundness we showed how to provide evidence both of shallowness and of increased confidence in a system, respectively.

Further research and evaluation are necessary to fully understand the impact and applicability of the contributions presented. However, the anecdotal evidence presented, using standard benchmark instances and verification techniques, seems to suggest the approach is indeed applicable. Furthermore, a number of possible use cases are discussed, opening promising avenues for future research.

One common characteristic of execution reports, discussed in Chapter 3, and the extensions discussed in this chapter is that both enable richer interaction between the user and the verification techniques. An evident drawback of richer user interaction is the necessary additional manual effort by the users of the verification tool.

The following chapter we will take a complementary approach and provide a standard structural coverage measure which would provide value without requiring additional user interaction.

```
1  insert(key, v, i)
2  check(v, j)
3  insert(key, v, i)
4  check(v, j)
5  insert(key, v, i)
6  check(v, j)
7  insert(key, v, i)
8  check(v, j)
9  insert(key, v, i)
10 check(v, j)
11 insert(key, v, i)
12 check(v, j)
13 insert(key, v, i)
14 check(v, j)
15 insert(key, v, i)
16 check(v, j)
17 insert(key, v, i)
18 check(v, j)
19 insert(key, v, i)
20 check(v, j)
21 insert(key, v, i)
22 /* First abstract trace ends here. */
23 /* check(v, j) Second abstract trace contains this statement. */
```

Fig. 4.8: Abstract traces generated from `insert` partial software model check.

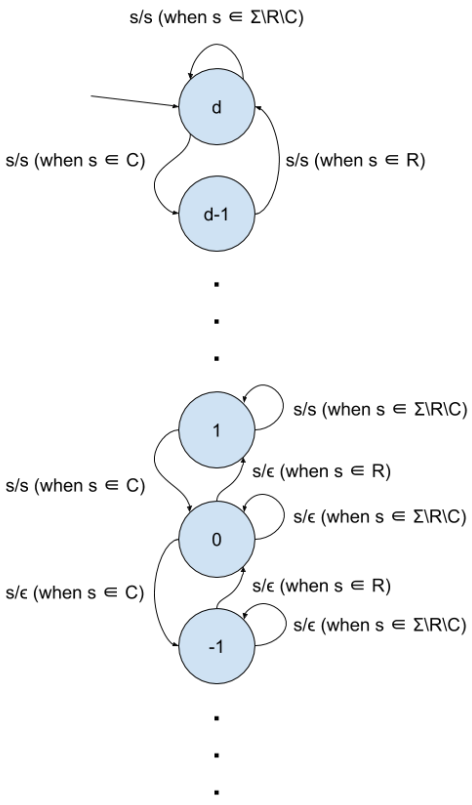

Fig. 4.9: Depiction of $stackDepth_{C,R}$ as a transducer.

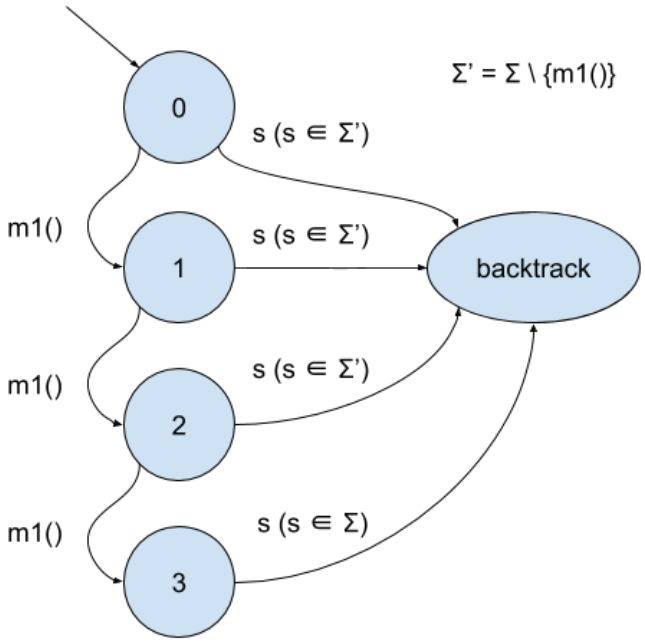

Fig. 4.10: Depiction of automaton corresponding to abstract trace [`m1()`, `m1()`, `m1()`].

```
1 int main() {
2   while (1) {
3     if (nondet()) {
4       m1();
5     } else {
6       m2();
7     }
8   }
9 }
```

Fig. 4.11: Example harness exercising `m1` and `m2`.

RESUMEN: EXTENSIONES DE MODEL CHECKER EXECUTION REPORTS

Los reportes presentados en el capítulo anterior proveen una manera novedosa de presentar el trabajo realizado durante la ejecución de un software model checker.

El objetivo a largo plazo de nuestro trabajo es presentar a los usuarios información valiosa. Por lo tanto, es importante que la presentación que ofrezcamos sea entendible y usable. Durante este capítulo, discutiremos algunas de las limitaciones más salientes de los model checker execution reports en su versión básica. Tomaremos esta discusión como puntapié inicial para presentar varias limitaciones que apuntan a superar estos obstáculos.

Las contribuciones de este capítulo están comprendidas por dos núcleos. En primer lugar, mostraremos una serie de preguntas que surgen de inspeccionar los reportes generados en el capítulo anterior. Estas preguntas aportan evidencia anecdótica de la utilidad de los reportes. Por otro lado, presentaremos enfoques que permiten responder estas preguntas naturalmente surgidas de la inspección de los reportes.

En esta línea, presentaremos los conceptos de k-incompletitud y de completitud relativa. Ambos permiten comparar el trabajo realizado por un software model checker durante una ejecución inconcluyente en términos de las limitaciones que tiene y las garantías que ofrece, respectivamente.

Para definir y ejemplificar estos conceptos, será relevante la capacidad de agrupar distintas trazas, para lo cual presentamos el concepto de trazas abstractas.

El capítulo estará guíado por la inspección manual de distintos ejemplos extraídos de instancias utilizadas en benchmarks estándar. Sin embargo, se incluye además una discusión respecto de la correctitud de nuestro enfoque para computar completitud relativa, que hace un uso novedoso del marco

conceptual de configurable software verification.

Por último, el capítulo concluye con una discusión de una variedad de posibles temas de investigación a futuro y aplicaciones de las extensiones propuestas.

5. COVERAGE OF PARTIAL SOFTWARE MODEL CHECKS

Introduction

Software Model Checkers[59] have shown outstanding performance improvements in recent times. Moreover, for specific use cases, software model checking techniques have shown to be highly effective, leading to a number of high-profile success stories [3, 22, 71].

Software model checkers often produce inconclusive results, that is, they neither find errors nor completely prove the absence of a specific bug within the allotted time or resource bounds. In these cases, most tools fail to provide any safety assurances, making the time spent on model checking entirely useless. Surely, however, the model checking effort must count for something. *How can partial software model checks be used to increase the confidence an engineer has on the system-under-analysis?*

Testing is the most widely adopted verification method used today in software engineering [5]. Testing is a partial verification effort in the sense that it does not cover the entire state space of the system under analysis. Test coverage in its different flavors is a widely adopted and discussed [85, 72, 43] metric that aims to describe to what extent the system is exercised. Although there are no definitive results regarding the relation between coverage, test suite quality and error finding capability(e.g., [54, 49]), coverage metrics are commonly used in industry [5] as a measure of the effort undergone in systematically testing software [49]. These metrics are of mandatory use in industrial standards for critical systems (e.g. DO-178B/C DO-278(A) for Commercial/Defense Avionics and Ground Systems, IEC 61508 for Industrial Controls, ISO 26262 for Automotive). Similar to a test suite, a partial software model check can help increase the confidence on a system if the extent to which

the model checker has explored the system can be meaningfully measured.

In this chapter, we propose measuring the coverage of a partial software model check as a metric that can provide some insight to developers regarding the degree to which the system has been analyzed. We believe that using metrics that are commonplace in testing may allow better understanding by an engineer of what has been accomplished by a software model checker and may even allow complementing testing results.

Moreover, in contrast with model checker execution reports and the extensions discussed in Chapter 4, the metric can be generated without additional user interaction and interpreted with minimal effort, given the already widespread use of structural coverage metrics in the context of testing.

The *main contribution of this work is a novel perspective on quantifying progress of partial software model checks.* Specifically, in this chapter we ask what it means for a code element to be covered by an incomplete SMC. If it is possible to compute the coverage (or a good lower bound) within reasonable time, and if incomplete SMCs actually cover a significant portion of the software under analysis.

The chapter is organized as follows. We first *recast the definition of coverage to the context of partial software model checks*, discussing some of the challenges in pinning down a definition that resembles the deeply rooted semantics of test coverage. We illustrate these difficulties with examples involving two different software model checking techniques. Subsequently we *propose a coverage definition for a broad family of software model checkers: those based on Abstract Reachability Trees.* Abstract Reachability Trees (ARTs) are an intermediate verification data structure that stores reachability information related to paths in a control flow automaton. ARTs are at the core of tools like Blast [10] and CPAchecker [14]. Moreover, ARTs allow significant flexibility, as proved by the techniques implemented on them, with examples as diverse as bounded model checking (BMC) [19], lazy predicate abstraction [53], value analysis [16], IC3-based techniques [33] and CEGAR variants of the former.

```
1 int nondet();
2 #define false 0;
3 int main() {
4   for (i = 0; i < 1000000; i++);
5   assert(false);
6   return 0;
7 }
```

Fig. 5.1: Long running loop.

In addition, we *discuss a general approach to computing an under-approximation of coverage*. Finally, we present an evaluation of a proof-of-concept implementation on instances from the SV-COMP [7] benchmark.

Recasting Coverage to Partial Model Checks

Application of the usual coverage metrics used in testing to partial software model checks is not immediate. Code coverage in testing is defined in terms of the parts of the system exercised throughout the execution of at least one test case. More specifically, statement coverage is defined as [87]:

> $C_{Statement}$ of T for P is the fraction of statements of program P executed by at least one test case in T.

$$C_{Statement} = \frac{\text{number of executed statements}}{\text{number of statements}}$$

One of the difficulties of adapting this definition is that, in the context of testing, test cases are expected to run to completion but that notion cannot always be directly translated to partial software model checks. More precisely, software model checking techniques frequently employ abstractions to represent the portion of the system that has been explored that do not necessarily capture any execution in its entirety.

Example 5 (Loop unrolling)**.** *Let us consider the example in Figure 5.1 and an exploration using a software model checking technique that explores paths in the Control Flow Automaton (CFA) of the program while keeping track of the actual values of variables, i.e. value analysis.*

The for*-loop runs 1000000 times increasing the value of* i *to that number. In this case, with a short enough time limit, the algorithm would keep unwinding the loop until timing out. This situation leads to an explored state space that contains no complete executions of the code. The analysis would be oblivious to the assertion in any partial software model check that does not fully unwind the loop.*

If we considered the exercised statements as covered, a relatively high coverage would be misleading, when the trivially false *assertion would be reached by the only possible execution of the code.* △

When defining a notion of software model check coverage, one might be tempted to quantify progress by directly measuring the internal representation of the technique as it is. In fact, this is the implementation of the coverage measure reported by CPAchecker[14]. However, the abstractions commonly used can capture spurious behaviors due to over-approximation. Consequently, it could be misleading to quantify progress this way.

Example 6 (Unreachable code)**.** *The code snippet in Figure 5.2 contains unreachable code. The condition in the* if *statement never holds, that is, the* then *branch can never be executed, therefore the assertion in line 11 always holds.*

In this case, a software model check using lazy predicate analysis would initially deal with an abstract representation which would not be precise enough to prove the then *branch unreachable. This first attempt would find a spurious counterexample and attempt to refine the abstraction. However, abstraction refinement, i.e. finding the right predicate to prove the* then *branch unreachable, would fail due to the non-linear condition.*

In this case, the unreachable portion of the state space corresponds to dead

```
1  int nondet();
2  int main() {
3      int x = nondet();
4      int reached_dead_code = 0;
5      int z = 1;
6      if (x*x < 0) {
7          reached_dead_code = 1;
8      } else {
9          z = 1;
10     }
11     assert(!reached_dead_code);
12     return 0;
13 }
```

Fig. 5.2: Dead code.

code, but more subtle cases can, and often do, arise in practice.

Therefore, unless unreachable portions of the state space are ignored, the coverage metric can be arbitrarily inflated with respect to its intuitive interpretation. △

The examples above show that to produce a definition of coverage, consistent with that of [87] but applied to partial software model checks (instead of testing) depends greatly on the data structures and algorithms used by the specific software model checking technique. In the next section we explain how the notion of coverage can be defined for a family of software model checkers based on Abstract Reachability Trees.

Coverage definition for Abstract Reachability Trees

In this section, we will present a definition of coverage for Abstract Reachability Tree [53] (ART)-based techniques. We will start by explaining our decision to tackle ARTs to implement a proof of concept. And, in order to make the

presentation self-contained, we will also provide the necessary background.

Why ARTs?

ARTs are a widely used data structure in software model checking. A number of dissimilar techniques have been implemented using ARTs, including lazy predicate abstraction [53], BMC [19], value analysis [16], IC3-based techniques [33] and CEGAR variants of the former, among other. As such, targeting ARTs allows us to handle a wide variety of techniques, at least in their ART implementation.

Moreover, Conditional Model Checking [11, 12] (CMC) has been instantiated for ART-based techniques and implemented within CPAchecker [14], providing valuable infrastructure. CMC proposes to augment software model checkers by either returning a counterexample to the property of interest or returning a condition ψ under which the model checking attempt proved the program safe to execute.

Background

An ART is a tree with each node corresponding to a single node in the control flow automaton (CFA) of the system-under-verification, essentially unfolding the CFA. ARTs are versatile and can be used by such a wide variety of techniques because each node is associated to an element of an arbitrary abstract domain. The abstract domain element captures a condition that must hold for any path that reaches the corresponding node, implicitly defining a reachable region.

ART-based algorithms then work by incrementally building an ART from its root and keeping a wait list of nodes whose successors have not yet been added to the tree. It is important to note that in the presence of branching statements, such as `if`-statements or loops, infeasible edges can be added to the tree. Whenever a target node, that is, a property violation, is added to the tree, a counterexample is produced and checked for feasibility before reporting

the assertion failure to the user. If spurious, the exploration continues right after refining the abstraction by strengthening the abstract domain element associated to one of the nodes in the path to the target state such that the latter is left outside the reachable region and therefore removed from the ART. The nodes in the wait list can be understood as the boundary of the exploration at a given iteration of the ART construction algorithm.

The instantiation of CMC for ART-based techniques represents ψ in terms of Assumption Automata over the alphabet of statements. That is, ψ in the case of ARTs can be understood as a predicate over sequences of statements.

An Assumption Automaton captures the structure of the ART at the moment when the software model check was interrupted: with the exception of a distinguished state, each state in the Assumption Automaton corresponds to a single state in the Control Flow Automaton of the system-under-verification and transitions are labeled with statements. Lastly, the distinguished state `FALSE` captures the unexplored state space. As expected, the state `FALSE` is added as a successors to the states corresponding to nodes in the wait list of the ART. In Assumption Automata the abstract domain elements are discarded.

Consequently, $\psi(\pi)$ holds *iff* π does not constitute a path from the initial state to the distinguished state `FALSE` in the Assumption Automaton.

Definition

Our definition aims to tackle the issues discussed and exemplified in Section 5.2 by adapting the notion of running a test to completion to the context of partial software model checking.

Definition 17. *Given a predicate ψ that corresponds to the output of a conditional model checker, a predicate φ capturing a safety property of interest and a set $\mathcal{T}$ of terminating executions of the system-under-verification, that is, executions reaching the system exit, a statement s is considered to be covered by the exploration when the following holds:*

$\exists\, t \in \mathcal{T}$, *such that* $\varphi(t) \wedge isExercisedWithinAnalysis_{\psi}(s, t)$

Moreover, in the context of ART-based analyses, the predicate $isExercised\text{-}WithinAnalysis_{\psi}(s,t)$ *holds iff:*

$$\exists\ \pi \in \Sigma^*, \textit{ such that } isPrefix(\pi,t) \wedge s \in \pi \wedge \psi(\pi)$$

where $isPrefix(\pi,t) = \exists \pi' \in \Sigma^*$ *such that* $\pi \cdot \pi' = t$ *and* $\cdot$ *stands for concatenation.*

Intuitively, Definition 17 states that a statement s is considered covered when some terminating execution (t) satisfying the safety property ($\varphi(t)$) contains an analyzed ($\psi(p)$) prefix ($isPrefix(p,t)$) that exercises s ($s \in p$).

Having adapted the notion of a particular statement being covered in the context of partial software model checking, we can also put forth a definition of statement coverage:

$C_{Statement}(\psi)$ for P is the fraction of statements of program P covered by the exploration captured by ψ.

$$C_{Statement}(\psi) = \frac{\text{number of statements covered by } \psi}{\text{number of statements}}$$

Computing coverage

We propose to compute partial software model check coverage by producing safe terminating executions of the system-under-verification that serve as witnesses of specific statements being covered. In terms of Definition 17, we generate elements $t \in \mathcal{T}$ and see which statements of t have been exercised during the partial software model check.

In order generate those executions, we encode, as a safety property, the negation of the conditions under which a set of statements is covered.

By feeding the modified safety property to an off-the-shelf software model checker, the counterexamples produced confirm that one of the statements included in the specification is covered whereas, if the property holds, then none of the statements are covered. Concretely, our algorithm takes as input

an Assumption Automaton and outputs the software model check coverage.

We depict the architecture of our approach in Figure 5.3. Algorithm 3 provides greater detail on the inner workings of the components of *Coverage Measurement*, as illustrated in Figure 5.3.

As shown in Algorithm 3, we iteratively augment the set `covered` until either the verification task yields no counterexample (line 11) or all statements have been covered (loop head, line 4). When the former occurs, statements in `remaining`, the complement of `covered`, are not covered by the exploration captured in the Assumption Automaton.

input : An Assumption Automaton: `AA`, A system `S`

output: A number indicating statement coverage.

```
1  allStatements := statements(S)
2  covered := ∅
3  remaining := allStatements
4  while remaining ≠ ∅ do
5  |  specification := terminatingExecutionSpecCovering(remaining,
   |   AA)
6  |  result := verify(S, specification)
7  |  if result.unknown then
8  |  |  warning("Producing under-approximation.")
9  |  |  break
10 |  else if ¬ result.foundCounterexamples then
11 |  |  break
12 |  else
13 |  |  cex := result.counterexamples
14 |  |  covered := covered ∪ exercisedWithinAnalysis(cex, AA)
15 |  |  remaining := remaining \ covered
16 return covered/allStatements
```

Algorithm 3: Computing Coverage

As mentioned earlier, intuitively, each iteration produces a safe terminating

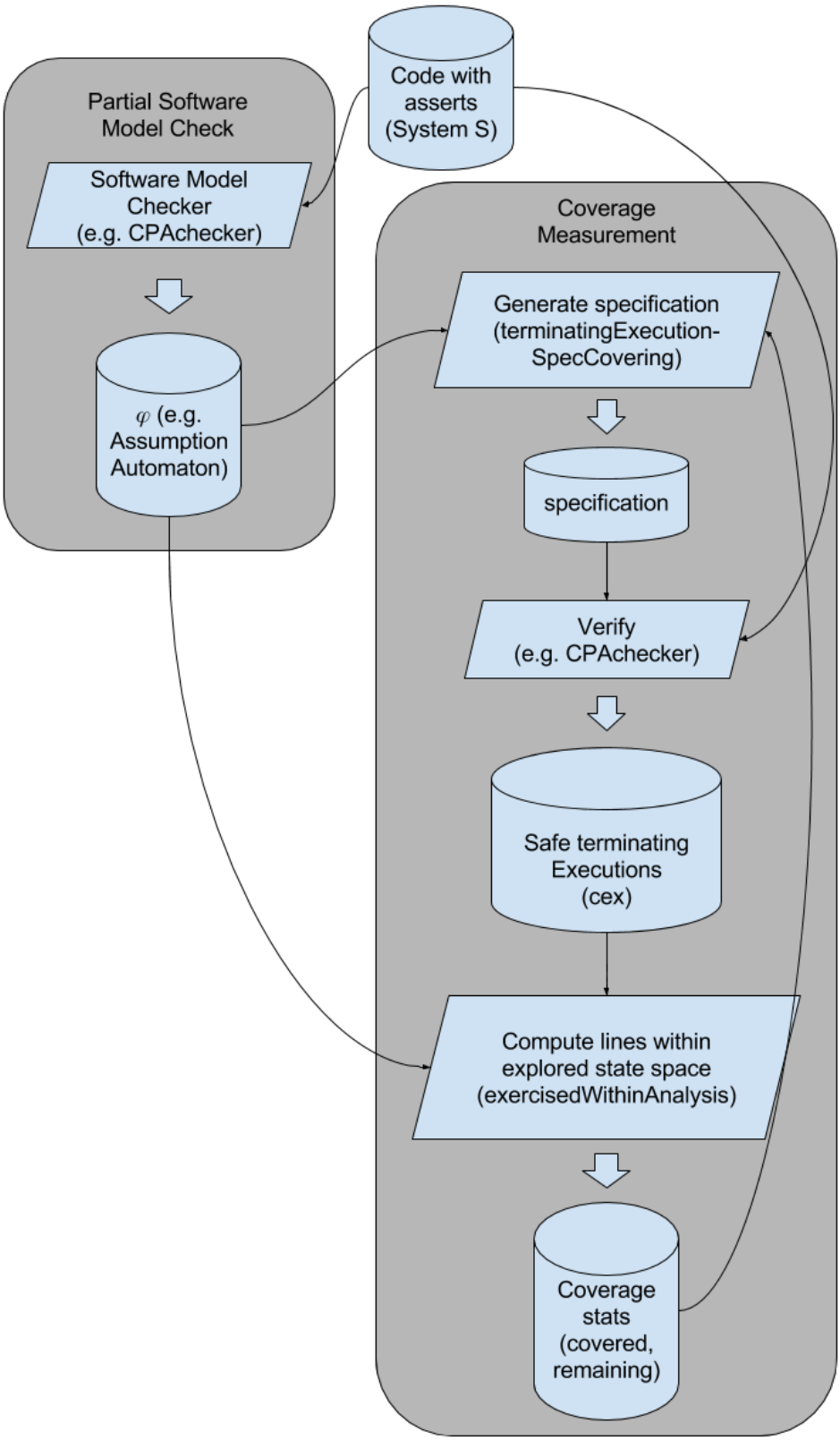

Fig. 5.3: Coverage measurement architecture.

execution of the system-under-verification that serves as a witness of a subset of statements being covered. Implicitly, the set of executions produced can be understood as a witness test suite.

Our algorithm yields the exact statement coverage as long as the procedure *verify* always returns a conclusive result. In practice, this is not always the case, as anticipated in line 8, therefore many times the algorithm produces an under-approximation. It is worth noting that if an exact result is not desired, it is possible to add a time budget or a limit in the number of iterations of the main loop and the result is guaranteed to be an under-approximation.

The specifics of how to build the corresponding specification, that is, the actual implementation of method *terminatingExecutionSpecCovering*, depend on the desired analysis technique and chosen tool. However, if this algorithm is used to produce an under-approximation, the only strict requirement for this method is that it should generate safe terminating executions. This can be expressed in most tools by adding false assertions right before the final return statements and inspecting the executions generated to ensure they are not safety violations.

Lastly, *exercisedWithinAnalysis* instantiates the predicate *isExercisedWithin-Analysis*, as stated in Definition 17.

The function *exercisedWithinAnalysis* is implemented by interpreting the counterexample as a path in the Assumption Automaton. The automaton contains a distinguished state `FALSE`, which captures the unexplored state space, and given the counterexample, i.e. a sequence of statements capturing a system execution, it is possible to follow, from an initial state, the transitions corresponding to each statement of the execution. The statements corresponding to transitions before reaching `FALSE` will be included in the result of the procedure.

That is, we are implicitly using the following property:

Property 8. *Given a sequence of statements* `cex`*, the following holds for the*

result of exercisedWithinAnalysis*:*

$$s \in exercisedWithinAnalysis(\texttt{cex}, \texttt{AA}) \text{ iff}$$
$$isExercisedWithinAnalysis_{\psi}(s, \texttt{cex})$$

Evaluation

We conducted a round of initial experiments with the exact algorithm, but the time necessary to compute the coverage measure was, in many cases, a multiple of the original model checking time. This prompted us to develop the under-approximation algorithm discussed. In this section we discuss the preliminary empirical evaluation we performed.

Our research questions for this evaluation are the following:

- **RQ 1**: Is it possible to compute a sufficiently good coverage under-approximation of a partial SMC efficiently?
- **RQ 2**: Do partial software model checks cover a reasonable proportion of the system under analysis?

To answer the research questions we require instances in which a ART-based software model checker fails to successfully complete its analysis. We used CPAchecker with a time limit of 900 s, which is standard in verification competitions [7]. The instances we used are those previously chosen to evaluate CMC, belonging to the families SystemC and DeviceDrivers of the SV-COMP [7] set of benchmark instances. The software model checks were run on an Ubuntu 16.04 system equipped with an Intel® Core™ i7-3770 CPU clocked at 3.40GHz with 16GB of DDR3 memory in a system without a swap partition running. We used BenchExec [17] to limit the resources to a single core and 12 GB of RAM.

For the instances for which CPAchecker did not produce a conclusive result, we computed under-approximations of the statement coverage. We used the approach described in Section 5.4 . The specific software model checking

technique used to generate terminating executions (see line 6 in Algorithm 3) is a flavor of ART-based value analysis [16] with a traversal geared towards quickly reaching the end of the program: a combination of depth-first-search with postorder (CFA nodes with a lower postorder index are selected first).

RQ 1: Is it possible to compute a sufficiently good coverage under-approximation of a partial SMC efficiently?

We considered a reasonable efficiency threshold for the computation of coverage under-approximation, the same execution time allotted to the inconclusive software model checking attempt, executing over the same hardware and operating system. We consider that longer running times could be deemed impractical by potential users. Thus, we set a time limit of 900 s for the computation of coverage under-approximations.

To assess how good the under-approximations are, ideally they should be compared against the true coverage of the partial software model checks. However, this data is not available as it is very expensive to compute (if at all possible). Consequently, we collected the coverage information that CPAchecker generates by default. This coverage information is an over-approximation of the real coverage as it corresponds to the number of statements for which a node in the ART existed at the moment when the execution was interrupted. Because ART nodes can be unreachable, the result reported by CPAchecker constitutes an over-approximation of the real coverage. Thus, the distance between the under-approximation produced by our approach and that of CPAchecker is an upper bound to the distance between our under-approximation and the real coverage.

Tables 5.1 and 5.2 contain the results of our experiments. Both tables comprise the instances that failed to produce a conclusive result within 900 s: in the case of Table 5.1, the partial software model check used value analysis, whereas Table 5.2 corresponds to using lazy predicate abstraction instead. Both techniques can be configured in a number of different ways, we used the

same configuration used for the evaluation of CMC[1]. The tables report for each instance, the name of the instance, the total number of lines excluding comments and blanks, the statement coverage under-approximation computed with our algorithm, the number of terminating executions generated to compute the under-approximation and the total CPU time used (which includes up to an additional 100 second for CPAchecker shutdown). Note that the two tables contain different instances as value analysis and lazy predicate abstraction have different strengths and weaknesses and consequently succeed and fail differently when attempting complete model checks of instances.

An important note is that a side effect of measuring coverage of a partial software model check is that it requires further exercising the system under analysis to produce terminating executions. When doing so, it is possible to find executions violating the property of interest. In other words, when computing coverage, bugs may be found. Indeed, this occurred in our experiments: for some of the instances, while attempting to produce terminating executions, the software model checker found assertion violations instead, as reported in column "Bug?" in Tables 5.1 and 5.2.

Although it is interesting to observe that the extension of a partial model check to terminating executions can enhance the bug detection capability of the partial check, this is not central to our contribution. However, this is relevant to the interpretation of the results, since our under-approximation algorithm stops when an assertion failure is reached. For this reason all rows in Tables 5.1 and 5.2 that indicate that a bug was found have execution times significantly lower than 900s and also report low coverage (even coverage 0).

Summarizing, the tables report for each instance the following information:

- # lines: Total number of lines excluding comments and blank lines.
- Over A.: Over-approximation of coverage reported by CPAchecker.
- Under A.: Statement coverage under-approximation we computed.

[1] CMC evaluation is available online: `https://www.sosy-lab.org/~dbeyer/cpa-cmc/`

Tab. 5.1: Complete results from explicit value analysis AA

			Baseline (with **Heuristic**)			
	# lines	Over A.	Under A.	# exec.	Bug?	CPU time
token_ring.14.BUG.c	833	594	570 (**-**)	1 (**-**)	- (**-**)	28.68s (**906.10s**)
toy.c	-	-	- (**-**)	- (**-**)	- (**-**)	979.53s (**971.91s**)
transmitter.16.BUG.c	900	658	- (N/A)	- (**1**)	- (✓)	921.54s (**22.85s**)
farsync.BUG.c	5700	1944	- (N/A)	- (**7**)	- (✓)	905.54s (**217.85s**)
gigaset.BUG.c	16188	1609	1077 (**970**)	9 (**9**)	- (**-**)	55.91s (**902.60s**)
lirc_imon.BUG.c	2166	875	- (**611**)	- (**6**)	- (**-**)	907.52s (**905.77s**)
loop.BUG.c	-	-	- (**-**)	- (**-**)	- (**-**)	902.31s (**912.45s**)
ppp_generic.BUG.c	6969	1849	- (**848**)	- (**2**)	- (**-**)	923.36s (**924.99s**)
synclink_gt.BUG.c	11780	1112	- (N/A)	- (**1**)	- (✓)	910.66s (**13.68s**)

- Under as % of Over: Statement coverage under-approximation expressed as a percentage of the over-approximation provided by CPAchecker.
- # exec.: Number of executions generated, and subsequently used, to compute the under-approximation.
- Bug?: Check mark (✓) indicates an assertion violation was found while attempting to find a terminating execution.
- CPU time: Total CPU time used.

As shown in Table 5.2, our approach generated an under-approximation for 14 out of 16 incomplete lazy predicate analysis explorations within the same time provided to the predicate analysis check. Table 5.1 shows similar results, producing an under-approximation for 7 out of 9 instances. This suggests that computing under-approximations for partial software model checks of different nature may be viable.

When comparing how close the coverage under-approximations are to the

Tab. 5.2: Complete results from lazy predicate abstraction AA

			Baseline (with **Heuristic**)			
	# lines	Over A.	Under A.	# exec.	Bug?	CPU time
kundu.c	266	256	250 (**252**)	1 (**2**)	- (**-**)	15.16s (**878.60s**)
mem_slave_tlm.3.c	796	550	490 (**490**)	1 (**2**)	- (**-**)	20.46s (**856.79s**)
mem_slave_tlm.4.c	801	553	495 (**495**)	1 (**2**)	- (**-**)	26.74s (**856.69s**)
mem_slave_tlm.5.c	806	753	500 (**500**)	1 (**2**)	- (**-**)	32.46s (**864.80s**)
pipeline.c	388	267	0 (**0**)	0 (**0**)	- (**-**)	87.67s (**867.81s**)
token_ring.03.c	306	261	227 (**253**)	1 (**5**)	- (**-**)	17.94s (**854.81s**)
token_ring.04.c	364	307	265 (**303**)	1 (**6**)	- (**-**)	60.42s (**861.16s**)
token_ring.05.c	422	361	303 (**353**)	1 (**7**)	- (**-**)	297.92s (**863.45s**)
token_ring.06.c	480	415	341 (**403**)	1 (**8**)	- (**-**)	439.57s (**876.84s**)
token_ring.07.c	538	469	- (**443**)	- (**8**)	- (**-**)	905.58s (**918.29s**)
token_ring.08.c	596	523	- (**421**)	- (**2**)	- (**-**)	909.00s (**906.12s**)
token_ring.09.BUG.c	660	582	456 (**458**)	1 (**1**)	- (**-**)	54.39s (**913.39s**)
token_ring.14.BUG.c	833	743	570 (**-**)	1 (**-**)	- (**-**)	190.27s (**910.16s**)
toy.c	-	-	- (**-**)	- (**-**)	- (**-**)	967.02s (**973.23s**)
toy1_BUG.c	316	295	N/A (N/A)	10 (**1**)	✓(✓)	34.86s (**14.19s**)
pktcdvd.BUG.c	6925	4660	- (**857**)	- (**3**)	- (**-**)	903.28s (**855.53s**)

real coverage metrics, the difference between the over approximation provided by CPAchecker and the computed under-approximation provides an upper bound. We look at the two tables separately as the results are rather dissimilar.

For the case of value analysis (Table 5.1), we must first consider the extreme cases: For two instances (`transmitter.16.BUG.c` and `synclink_gt.BUG.c`) the algorithm produced a trivial under-approximation of 0. This is because the tool found an assertion failure and stopped within 5% of the allotted time. For four of the remaining cases in Table 5.1, if we consider the over-approximation as the actual coverage, the under-approximations computed are, worst case, between 40% and 50% of the real coverage. Finally, for one instance, `lirc_imon.BUG.c`, the under-approximation is, at worst, 87% of the actual coverage value.

For the predicate abstraction case (Table 5.2), for 5 out of the 14 cases for which an under-approximation was computed, worst case is that the approximation is only 20% less than the actual value. The difference was slightly larger for 6 of the remaining instances, ranging from 20% to 45% less than the actual value. These results leave 3 instances with under-approximations significantly smaller than the upper bound provided by CPAchecker. In one such case, `pipeline.c`, the program exit is unreachable according to CPAchecker, hence there can be no terminating tests and coverage (under its usual semantics) is necessarily 0. For this last case, it is interesting to note how misleading the over-approximation reported by CPAchecker is. An additional case, `toy1_BUG.c`, corresponds to a bug that was found. And finally, for `pktcdvd.BUG.c`, our algorithm stops due to an `UNKNOWN` result from CPAchecker. This result indicates that CPAchecker failed to remove a spurious counterexample from its search space. It is worth noting that for this instance less than 10% of the allotted processing time was used. Thus, for predicate abstraction, the computation of under-approximation in this experiment was fairly robust, providing feedback for 14 out of 16 instances, finding errors in one and reporting reasonably close under-approximations for 12, with only one instance having a

possibly poor under-approximation. Moreover, it would be possible to make the implementation handle cases such as `pktcdvd.BUG.c` better by providing a fallback mechanism when the configuration we used by default cannot produce a terminating execution.

The results suggest that for both types of partial software model checks, the computation of an under-approximation can provide a reasonable bound for coverage, which in turn suggest that **RQ 1** may be answerable positively. However, and as expected, the degree to which the computation approximates the real coverage seems to vary depending on the underlying software model checking technique used in the original partial check. Our results would seem to show that for the cases analyzed, the approach worked better for the predicate abstraction than for value analysis. However, it is difficult to compare the two as there are few instances for which both techniques fail and because they have fundamentally different exploration approaches that make the ART grow in different ways. Indeed, for value analysis, the assumption automata were significantly larger than for predicate abstraction. We believe that the size of the assumption automata may have increased the difficulty of verifying feasibility of extension traces which may in turn hinder improving the under-approximation.

RQ 2: Do SMCs cover a reasonable proportion of the system under analysis?

In **RQ 1** we aim to understand if it is possible to approximate the coverage of a partial software model check. In this question we are now interested in gaining insight as to whether partial software model checks actually manage to cover a reasonable portion of the system under analysis. Two alternatives to answering this question are to compare the coverage achieved by partial model checking against manually produced test suites or against automatically generated test suites. We opted for the latter as we could not find third-party manually produced test suites and automatic test case generation also allows

Tab. 5.3: Complete results from KLEE test suite generation

	# lines	Test Suite Coverage	Verification Coverage	Klee OOM?
token_ring.14.BUG.c	833	299	-	
toy.c	314	-	-	OOM
transmitter.16.BUG.c	900	-	0	OOM
farsync.BUG.c	5700	78	779	
gigaset.BUG.c	16188	55	796	
lirc_imon.BUG.c	2166	100	765	
loop.BUG.c	4373	79	707	OOM
ppp_generic.BUG.c	6969	112	848	OOM
synclink_gt.BUG.c	11780	72	0	
kundu.c	266	-	239	OOM
mem_slave_tlm.3.c	796	382	324	
mem_slave_tlm.4.c	801	357	325	OOM
mem_slave_tlm.5.c	806	-	465	OOM
pipeline.c	388	-	0	OOM
token_ring.03.c	306	210	220	
token_ring.04.c	364	249	260	
token_ring.05.c	422	288	308	
token_ring.06.c	480	327	336	
token_ring.07.c	538	328	336	OOM
token_ring.08.c	596	335	361	OOM
token_ring.09.BUG.c	660	321	392	OOM
toy1_BUG.c	316	-	0	OOM
pktcdvd.BUG.c	6925	104	912	

a comparison based on verification effort (i.e., CPU time).

Thus, we resort to generating a test suite with comparable resource budgets and comparing the coverage achieved by the test suite and by the partial software model check. It is crucial to emphasize that our approach is not meant to replace testing and we do not envision our techniques or its possible use cases overlapping nor competing within the area of test suite generation. Software model checking and testing, including test suite generation, are complementary and every company or institution adopting software model checking also heavily relies on testing. However, the comparison does allow us to move away from absolute considerations regarding how high or low the coverage of partial software model checks are.

For the purpose of analyzing **RQ 2**, we are assuming that our under-approximation correctly reflects the coverage achieved during the partial software model check. Whether this holds has been discussed as part of **RQ 1**. This entails a conservative comparison in which the worst possible partial software model check coverage is contrasted with the real coverage of an automatically generated test suite.

To automatically generate a test suite we used KLEE [20], a state-of-the-art symbolic virtual machine that has been successfully used to generate high coverage test suites for system programs, finding bugs and vulnerabilities. Moreover, the test suites generated by KLEE outperformed existing test suites in terms of line coverage and also found several bugs that had eluded code reviews and manual testing for years [20]. The symbolic nature of KLEE also makes it suitable to handle the non-determinism present in the benchmark instances at hand. We used the same hardware and Z3 [39] as the solver backend but doubled the time limit (to 1800 s) to account for the time spent both in the partial software model check and computing a coverage approximation. Given the number of out-of-memory errors, it is worth noting that we allowed up to 13 GB for KLEE, whereas we used 12 GB for the other experiments.

In 15 out of 24 instances the partial software model check coverage under-

approximation exceeded the coverage of a test suite generated using KLEE. In 4 of the remaining cases both KLEE and our algorithm tied with respect to coverage: KLEE did not produce any tests and our implementation either failed to produce a result or found an assertion failure and reported 0 coverage. For two instances KLEE produced a test suite with higher coverage than our under-approximation, but the latter remained within a similar range (within 20% of that of the test suite). Lastly, for two instances, `token_ring.14.BUG.c` and `synclink_gt.BUG.c`, KLEE achieved significantly higher coverage. However, our algorithm returned prematurely for `synclink_gt.BUG.c`, after finding an assertion failure.

In some cases, 6 in total, the partial software model check coverage was several factors of that achieved by the test suite. The instances where partial software model check coverage more significantly exceeds KLEE's are the largest ones in terms of number of lines. The exception is `synclink_gt.BUG.c`, where, as we mentioned, our algorithm returned prematurely after finding an assertion failure. This can be explained by the order in which KLEE schedules its state exploration. In contrast to our implementation, which combines depth-first-search with postorder, they alternate two heuristics, one which favors shorter executions and another one which favors reaching statements not yet visited. These two can end up favoring a broader but shallower search, which takes longer to find terminating executions when the level of nondeterminism is high and the paths to the final return statement are long.

The evaluation seems to suggest that partial software model checks can achieve reasonable coverage, providing some evidence to answering **RQ 2** affirmatively.

Discussion

Variants of the algorithm

Algorithm 3 most closely captures our current implementation. However, we will discuss a number of possible variants of interest.

One possibility, for instance, is to generate terminating executions independent from a specific partial software model check. This would be interesting in a number of scenarios.

First of all, if the system-under-verification will be analyzed repeatedly with different techniques, pre-computing a single set of safe terminating executions could yield significant performance gains in this case, at the cost of possibly lower-quality under-approximations.

Another reason to dissociate execution generation from the partial software model check would be to replace the back-end. Crafting a specification that captures a condition that refers to the partial software model check can sometimes be impractical. Our implementation leverages existing CPAchecker features that allow using Assumption Automata as part of specifications, but translating those specifications for other tools to use might require expensive code transformations to encode the automaton within the code. In this case, the implementer would gain in terms of ease of implementation, and possibly performance improvements, at the expense of lower quality under-approximations for a given number of terminating executions.

Moreover, other approaches to generating safe terminating executions might not take a specification as input at all. For instance, in principle, it would be possible to leverage automatic test suite generation or logging of the running system to fulfill the goal of gathering system executions.

Tackling other techniques

In our definition of partial software model check coverage, Definition 17, we instantiate the predicate $isExercisedWithinAnalysis_{\psi}$ in the context of Abstract Reachability Trees.

However, we believe this definition can be applicable to other entirely unrelated techniques. Moreover, the generic approach to computing partial software model check coverage under-approximations, depicted in Figure 5.3, decouples the generation of executions and the subsequent computation of

which statements of those executions were exercised during the partial software model check.

Therefore, computing a partial software model check coverage under-approximation for an entirely different technique equates to being able to instantiate the predicate $isExercisedWithinAnalysis_{\psi}$ to the corresponding context.

Throughout this section we aim to discuss how $isExercisedWithinAnalysis_{\psi}$ can be adapted to relevant unrelated techniques and representations of partial software model checks.

Dafny

Dafny is both a language and a verifier. The language is designed as an imperative, class-based language but has built-in support for specifications, such as preconditions, postconditions and termination metrics. Moreover, the language includes other verification constructs, such as ghost variables and user defined mathematical functions.

Dafny, as a verifier, performs a modular analysis by translating a Dafny program into Boogie 2, an intermediate verification language. The translation is sound, that is, the correctness of the Boogie program implies the correctness of the original Dafny program.

Dafny unifies specification and implementation in one language, providing useful features for assisted verification.

We will consider an extension of Dafny [30] which annotates the code with assumptions made during a partial software model check. The extension consists, essentially, on extending the traditional `assume` statements such that the new variant captures unsound assumptions incorporated during verification. Moreover, `assert` is also extended such that the analysis can record assertions verified only relative to certain unsound assumptions.

This extension is amenable to modular analyses, therefore it would be important to support this use case. Within this Dafny extension, $isExercisedWithinAnalysis_{\psi}(s, \texttt{cex})$

holds as long as all of the following hold:

- s is a statement within a method `foo` already analyzed (some methods might be scheduled to be analyzed later, for example)
- s is the last statement of a prefix of `cex` that satisfies all assumptions within `foo`, including `assumed` statements, regular `assume` statements and preconditions.

Corral

We can now discuss Corral [64], a whole-program analyzer of Boogie [40] code, also used as a back-end to Smack [75], which supports C code. Corral is a bounded reachability solver that works by iteratively generating under and over-approximations of the system-under-analysis. The system is under-approximated by assuming no method calls are reached and, conversely, over-approximated by replacing method calls with a sound over approximation of the effect of such call. If an error is found in the under-approximation, it must also exist in the original system. Conversely, if the over-approximation is proved safe, then the system is also safe. However, if an error is only found in the over-approximation, the method calls involved in the counterexample are inlined. This way, Corral iteratively inlines code on-demand, up to a certain bound, avoiding the performance penalty of inlining every call upfront.

In Corral, $isExercisedWithinAnalysis_{\psi}(s, \texttt{cex})$ holds as long as s is a statement within a prefix of `cex` that does not reach a method call.

CBMC[61] is a bounded model checker that works by unrolling loops up to a given depth. CBMC then turns the modified system, and the assertions encoding a safety property, into a propositional satisfiability formula and uses an off-the-shelf SAT solver to decide if an assertion violation exists.

Similarly to Corral, $isExercisedWithinAnalysis_{\psi}(s, \texttt{cex})$ holds for every statement s within a prefix of `cex` that does not exceed the loop unrolling bound.

Threats to validity

Benchmark and instances

The set of instances used to conduct our empirical evaluation could have a significant impact in the results we obtain.

We mitigate this threat using standard benchmark instances selected to evaluate previous published work. The benchmarks we used are standard within the verification community and used in the annual Competition on Software Verification, SV-COMP [7].

Within the different benchmarks used in SV-COMP we used two families of instances which had been previously chosen to evaluate Conditional Model Checking [11, 12], thereby also mitigating the possibility of selection bias.

Moreover, to the best of our knowledge, there is nothing specific about the instances that would invalidate the results of our experiments.

One important remark regarding the selection of time limits for the coverage computation is that the computation of the under-approximations is incremental, hence should the time limit be reduced or expanded the under-approximations will improve or worsen gradually, consequently we do no believe the particular choice of time limit is an important threat.

A more extensive experiment was not possible due to limited computational resources, making an evaluation with the full set of SV-COMP benchmarks exceedingly onerous.

Test case generation tool

In the context of **RQ 2** we compare partial software model check coverage numbers with test suite coverage from a test suite automatically generated using KLEE. It is possible that other tools could significantly outperform KLEE for these particular instances. However, KLEE is a well maintained state-of-the-art tool with support for symbolic values, which makes its particularly suitable for handling verification instances containing non-determinism. Also,

any threats to validity derived from the usage of KLEE during the evaluation are limited to **RQ 2**.

ART-based techniques

Our empirical evaluation is limited to the proof-of-concept implementation and the verification techniques used. In particular, the choice of lazy predicate analysis and value analysis for software model checking could jeopardize the generality of our results to other techniques.

We mitigate this threat by choosing significantly different techniques. The contrast between lazy predicate abstraction and value analysis can, for example, be observed in the very few number of instances for which they both reach the time limit (only 2).

More broadly, however, it would be possible that the ideas we put forth are not applicable to techniques outside the realm of ARTs. We discuss how to adapt some of our contributions to other techniques but validating the research questions in a broader setting would significantly exceed the scope of this work.

Feasibility versus Executability

We obtain safe terminating execution using a software model checker that guarantees feasibility, but not necessarily executability in a specific architecture.

We mitigate this threat by performing an additional bit-precise feasibility check on every execution generated.

Although we did not implement the following for our current proof-of-concept implementation, it would be possible to enforce executability by generating inputs for the system and confirming executability using the actual compiled binary.

Related Work

Many works have proposed to use model checking to generate test suites to either cover model or code elements [46, 83, 76, 45]. Note that our goal is completely different from those: measure the strength of a partial software model check.

The most related previous work defined a measure of verification coverage [2] consisting of the number of statements proven safe at a certain point. The definition has some subtleties, but a statement which cannot throw any of the predefined exceptions of interest is considered safe. The semantics, in this case, is fundamentally different than that of test suite coverage making the metrics incomparable, but might also not be entirely intuitive or easy to interpret, as the authors themselves acknowledge, since many statements can be proved safe syntactically, regardless of the context. In order to compute verification coverage, they propose to adapt a standard imprecise verification algorithm such that it computes a coverage metric besides producing the warnings of potential errors. Their empirical evaluation suggests that the approach incurs in a significant performance penalty: the running time of the algorithm increases by an order of magnitude when adapted in this way. Lastly, it is not straightforward how to compute the coverage measure for a partial verification attempt using a standard technique, as we intend in our setting.

The possibility of using bounded model checking results in assurance cases was discussed [37] and illustrated with a number of safety critical case studies using CBMC [61] to perform verification. The approach consists of bounding the environment that interacts with the system and performing full verification under that assumption. Successful verification is still inconclusive, because the environment is bounded, but the authors claim that the result can still be used as part of an assurance case, using coverage as evidence of adequacy, and possibly complementing model checking with testing. However, there are some key differences with our work: code coverage in [37] is meant to pinpoint potential deficit of environment restrictions used for the bounded

model check phase. In particular, coverage is computed by the test suite generated by CBMC when using the same environment and configuration used for model checking (which under-approximates the actual software model check coverage as we define it). Thus, the bounds imposed on the environment need to deterministically define the state space explored. Last but not least, the bounded model check phase is not a general partial software model check as defined is this chapter since it is assumed to run to completion given those environment restrictions.

Existing work discussing functional coverage [74] briefly mentions two possible high-level approaches to quantify the verification coverage of *complete* verification attempts of a portion of a system. Our work, in contrast, discusses the topic at a much lower level, having a single *partial* verification, in particular software model checks, as our object of study.

Loop coverage has been discussed previously [63], although tangentially, in relation to bounded-exploration tools. These tools can miss trivial defects when the bound prevents the exploration from crossing a long-running loop. Loop coverage is not defined in that work and only its intuitive meaning is used. The authors use defect recall as a proxy for loop coverage. In contrast, we put forth a detailed definition and discuss at length how to compute an under-approximation.

CPAchecker reports a coverage metric which, to the best of our knowledge, has not been published nor defined formally. The number reported, as currently implemented, can report unreachable code as covered and corresponds to an over-approximation of our coverage metric. We discussed the problems of this semantics in Section 5.2.

Our work is heavily influenced by and builds upon earlier work on partial verification results, such as the notion of Conditional Model Checking [12], reports based on sequences of statements [24] and other complementary representations [30] of partial verification results. Conditional Model Checking allows different techniques to exchange partial software model checks. Simi-

larly, `assumed` annotations in the code [30], different from standard `assume` statements, were proposed to document unsound assumptions made during model checking. These annotations allow subsequent software model checks to scope the verification attempt assuming certain predicates to hold. Also, these annotations can be cleverly used to produce a test suite [30, 31] that complements a partial software model check. The idea of producing a test suite complementing the explored state space was also reproduced using Assumption Automata [36]. Reports based on sequences of statements [24] provide a window to observe the progress achieved in a partial software model check, allowing users to inspect sets of executions which have been proven safe and, conversely, executions which necessarily escape the explored state space. In contrast to the coverage measure we propose, analyzing these reports involves significant manual inspection. These approaches [24, 36, 31, 30, 12], however, neither aim to quantify coverage nor provide any alternative metric of progress or exploration.

Summary and Future Work

Software model checkers have dramatically improved within the last few years and state-of-the-art tools are capable of tackling an increasing number of industrial instances. However, a large number of systems remain intractable for full verification. This chapter is a step towards answering the question of how an incomplete software model check can be used to increase the confidence software engineers have on their software under analysis. Specifically, in this work, we attempt to extract valuable information from partial software model checks by producing a coverage metric. We defined partial software model check coverage in a manner that is consistent with that of coverage in the context of testing and discussed how to produce an under-approximation of that metric. Our empirical evaluation suggests that acceptable under-approximations can be computed efficiently. Moreover, software model checking techniques seem to cover a significant proportion of the system-under-analysis.

Regarding future work, to some extent, in this chapter we propose to interpret partial software model checks as the execution of (tacit) test cases that were not necessarily run to completion. In fact, we show how to report statement coverage as a proxy of strength of a partial software model check if it were interpreted and extended as a test campaign. In this line of thinking, one can imagine several approaches to better characterize the strength of those test suites underlying the partial software model check. Obviously, many other coverage metrics could be reported: from entry point coverage to MC/DC. Moreover, given that the test suite quality seems correlated to size [49], one can imagine actually reporting a number of test cases that could be extracted, in a lightweight manner, from the partial software model check. We conjecture this number is likely to be high or even infinite given the typical symbolic nature of software model checkers and their ability to detect that a given state has already been visited. When a potentially infinite number of test cases can be deduced from a partial SMC, it is likely that new means to report sets of test cases will be required, ones that describe the diversity of the tests beyond coverage.

RESUMEN: COBERTURA ESTRUCTURAL EN EJECUCIONES PARCIALES DE SOFTWARE MODEL CHECKERS

Tomando una dirección diferente a la de los capítulos anteriores, en este capítulo proponemos una presentación mucho más resumida del trabajo realizado por un software model checker.

Inspirados en el casi universal uso de métricas de cobertura de código en el día a día del desarrollo de software para cuantificar la calidad del testing, nos proponemos generar métricas de cobertura estructural que sean compatibles con la interpretación habitual de dichos indicadores.

Adicionalmente, parte de la contribución de este capítulo es explorar distintas dificultades en formular y computar medidas de cobertura que sean consistentes con la definición habitual, utilizada en testing.

A diferencia de los enfoques discutidos en los capítulos anteriores, la métrica que proponemos en este capítulo puede ser generada y consumida por un usuario sin interacción adicional.

6. CONCLUSIONS

In Chapter 2 we presented an overview of the landscape of partial verification results. The analysis of the characteristics of each approach showed that existing work either produced output that was hardly intelligible to inexperienced users or, instead, placed major restrictions on the termination conditions supported, thereby making on-line monitoring of a verification attempt impossible.

In Chapter 3 we presented a novel approach to conveying the progress and shortcomings of a partial software model check for a broad family of verification techniques. The output, although ad-hoc, is entirely intuitive to the software development community at large.

In Chapter 4, we present a number of extensions to the basic model checker execution reports presented in Chapter 3. These extensions address some of the shortcoming of execution reports and further increase their understandability and usability.

Finally, Chapter 5 includes a thorough discussion of the obstacles in defining straightforward structural coverage measures for partial software model checks. Following, we put forth a novel approach to quantifying coverage for partial software model checks that is consistent with the deeply-rooted understanding of structural coverage in the context of testing.

The present work, as a whole, constitutes a broad assessment of the viability of leveraging partial software model checks to produce human-readable output.

Future Work and Outlook

We believe it is important to provide information on inconclusive software model checks performed with techniques other than the ones we tackled. Our

work provides a baseline and a conceptual underpinning to future efforts in this direction.

We have presented several potential use cases for our extensions of execution reports that still need to be evaluated. These seem to be promising avenues for future research, especially in terms of integrating partial software model checks into standard software development work-flows.

Although software model checkers often make unsound assumptions, the effect of such assumptions on the analysis output is assumed to be acceptable. We strongly believe that techniques that can intuitively convey safety assurances even when dealing with inconclusive results can make much more aggressive assumptions. Therefore, future research into combining human-readable partial verification results and aggressive unsound optimizations might yield significant gains.

The main motivation behind this work is leveraging the countless hours of computation invested in software model checking that ultimately lead to inconclusive outcomes. Recent research [15] has highlighted the value of software model checking for bug finding in comparison to testing. However, a necessary step towards greater adoption of software model checking is to quantify and convey progress in a straightforward way. We believe our work solidifies the basis for several promising and diverse approaches, including but surely not limited to the ones we briefly mention here.

RESUMEN: CONCLUSIONES

En este capítulo hacemos un breve repaso de las contribuciones tratadas a lo largo de la tesis y cómo se conectan entre sí.

Para concluir la tesis, incluímos nuestra visión de la perspectiva de posible trabajo a futuro relacionado con la temática tratada. Enfatizamos la importancia de extender nuestro enfoque a otras técnicas y la posibilidad de utilizar nuestras contribuciones como marco de referencia.

Por otro lado, reiteramos que las líneas de posible trabajo a futuro que presentamos en el capítulo 4 constituyen opciones prometedoras, especialmente respecto de integrar esfuerzos de verificación con prácticas estándar en desarrollo de software.

Por último, finalizamos el análisis contextualizando nuestro trabajo como un paso más hacia la adopción de herramientas de software model checking.

BIBLIOGRAPHY

[1] Allen T Acree, Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Mutation analysis. Technical report, GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, 1979.

[2] Stephan Arlt, Cindy Rubio-González, Philipp Rümmer, Martin Schäf, and Natarajan Shankar. The gradual verifier. In *NASA Formal Methods Symposium*, pages 313–327. Springer, 2014.

[3] Thomas Ball, Vladimir Levin, and Sriram K Rajamani. A decade of software model checking with slam. 54(7):68–76, 2011.

[4] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[5] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering, 2007. FOSE '07*, pages 85–103, May 2007.

[6] Dirk Beyer. Software verification and verifiable witnesses. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–416. Springer, 2015.

[7] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In Chechik and Raskin [26], pages 887–904.

[8] Dirk Beyer, Adam J Chlipala, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The blast query language for software verification. In *Static Analysis*, pages 2–18. Springer, 2004.

[9] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 721–733. ACM, 2015.

[10] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. 9(5-6):505–525, 2007.

[11] Dirk Beyer, Thomas A Henzinger, M Erkan Keremoglu, and Philipp Wendler. Conditional model checking. 2011.

[12] Dirk Beyer, Thomas A Henzinger, M Erkan Keremoglu, and Philipp Wendler. Conditional model checking: a technique to pass information between verifiers. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 57. ACM, 2012.

[13] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification*, pages 504–518. Springer, 2007.

[14] Dirk Beyer and M Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification*, pages 184–190. Springer, 2011.

[15] Dirk Beyer and Thomas Lemberger. Software verification: Testing vs. model checking. In *Haifa Verification Conference*, pages 99–114. Springer, 2017.

[16] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on cegar and interpolation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 146–162. Springer, 2013.

[17] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Benchmarking and

resource measurement. In *Model Checking Software*, pages 160–178. Springer, 2015.

[18] Dirk Beyer and Philipp Wendler. Algorithms for software model checking: Predicate abstraction vs. impact. page 106.

[19] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 317–320. ACM, 1999.

[20] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[21] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. 56(2):82–90, 2013.

[22] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. 15:3–11, 2015.

[23] Rodrigo Castaño, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. Model checker execution reports. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 200–205. IEEE Press, 2017.

[24] Rodrigo Castaño, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. Model checker execution reports. 2016.

[25] Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 73–82. ACM, 2004.

[26] Marsha Chechik and Jean-François Raskin, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*. Springer, 2016.

[27] Maria Christakis, K Rustan M Leino, Peter Müller, and Valentin Wüstholz. Integrated environment for diagnosing verification errors. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 424–441. Springer, 2016.

[28] Maria Christakis, K. Rustan M. Leino, Peter Müller, and Valentin Wüstholz. Integrated environment for diagnosing verification errors. In Chechik and Raskin [26], pages 424–441.

[29] Maria Christakis, Peter Müller, Valentin Wüstholz, et al. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *VMCAI*, pages 336–354. Springer, 2015.

[30] Maria Christakis, Peter Müller, and Valentin Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM 2012: Formal Methods*, pages 132–146. Springer, 2012.

[31] Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In Laura K. Dillon, Willem Visser, and Laurie Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 144–155. ACM, 2016.

[32] Maria Christakis and Valentin Wüstholz. Bounded abstract interpretation. In *International Static Analysis Symposium*, pages 105–125. Springer, 2016.

[33] Alessandro Cimatti and Alberto Griggio. Software model checking via ic3. In *International Conference on Computer Aided Verification*, pages 277–293. Springer, 2012.

[34] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. 50(5):752–794, 2003.

[35] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrÉe analyzer. In *Esop*, volume 5, pages 21–30. Springer, 2005.

[36] Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. Just test what you cannot verify! In *Fundamental Approaches to Software Engineering*, pages 100–114. Springer, 2015.

[37] Carmen Cârlan, Daniel Ratiu, and Bernhard Schätz. On using results of code-level bounded model checking in assurance cases. In *International Conference on Computer Safety, Reliability, and Security*, pages 30–42. Springer, 2016.

[38] Ankush Das, Shuvendu K Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In *International Conference on Computer Aided Verification*, pages 324–342. Springer, 2015.

[39] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. pages 337–340, 2008.

[40] Robert DeLine and K Rustan M Leino. Boogiepl: A typed procedural language for checking object-oriented programs. 2005.

[41] Nicolás D'Ippolito, Marcelo F Frias, Juan P Galeotti, Esteban Lanzarotti, and Sergio Mera. Alloy+ hotcore: A fast approximation to unsat core. In *Abstract State Machines, Alloy, B and Z*, pages 160–173. Springer, 2010.

[42] Javier Esparza. Decidability of model checking for infinite-state concurrent systems. 34(2):85–107, 1997.

[43] Motor Industry Software Reliability Association et.al. *MISRA-C: 2004: guidelines for the use of the C language in critical systems*. MIRA, 2008.

[44] Antonio Filieri, Corina S Păsăreanu, and Willem Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 622–631. IEEE Press, 2013.

[45] Gordon Fraser, Franz Wotawa, and Paul E Ammann. Testing with model checkers: a survey. 19(3):215–261, 2009.

[46] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *ACM SIGSOFT Software Engineering Notes*, volume 24, pages 146–162. Springer-Verlag, 1999.

[47] Mitchell J Gerrard and Matthew B Dwyer. Comprehensive failure characterization. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 365–376. IEEE, 2017.

[48] Simon F Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. *ACM SIGPLAN Notices*, 40(10):385–402, 2005.

[49] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 72–82, New York, NY, USA, 2014. ACM.

[50] Alex Groce, Iftekhar Ahmed, Carlos Jensen, and Paul E McKenney. How verified is my code? falsification-driven verification (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 737–748. IEEE, 2015.

[51] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In *International Conference on Computer Aided Verification*, pages 453–456. Springer, 2004.

[52] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *Model Checking Software*, pages 121–136. Springer, 2003.

[53] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. 37(1):58–70, 2002.

[54] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.

[55] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[56] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 62–73. ACM, 2001.

[57] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187. ACM, 2003.

[58] Eunkyoung Jee, Seungjae Jeon, Sungdeok Cha, Kwangyong Koh, Junbeom Yoo, Geeyong Park, Poonghyun Seong, et al. Fbdverifier: Interactive and visual analysis of counter-example in formal verification of function block diagram. *Journal of Research and Practice in Information Technology*, 42(3):171, 2010.

[59] Ranjit Jhala and Rupak Majumdar. Software model checking. 41(4):21, 2009.

[60] David E Kieras and Susan Bovair. The role of a mental model in learning to operate a device. *Cognitive science*, 8(3):255–273, 1984.

[61] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *TACAS*, pages 389–391, 2014.

[62] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *Computer aided verification*, pages 585–591. Springer, 2011.

[63] Akash Lal and Shaz Qadeer. Powering the static driver verifier using corral. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 202–212. ACM, 2014.

[64] Akash Lal, Shaz Qadeer, and Shuvendu K Lahiri. A solver for reachability modulo theories. In *International Conference on Computer Aided Verification*, pages 427–443. Springer, 2012.

[65] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010.

[66] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[67] Donald W Loveland. *Automated Theorem Proving: a logical basis*. Elsevier, 2016.

[68] Michael Martin, Benjamin Livshits, and Monica S Lam. Finding application errors and security flaws using pql: a program query language. In *ACM SIGPLAN Notices*, volume 40, pages 365–383. ACM, 2005.

[69] Kenneth L McMillan. *Model checking*. John Wiley and Sons Ltd., 2003.

[70] Florian Merz, Carsten Sinz, Hendrik Post, Thomas Gorges, and Thomas Kropf. Bridging the gap between test cases and requirements by abstract testing. *Innovations in Systems and Software Engineering*, 11(4):233–242, 2015.

[71] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. 58(4):66–73, 2015.

[72] Alan Page, Ken Johnston, and Bj Rollison. *How we test software at Microsoft*. Microsoft Press, 2008.

[73] Esteban Pavese, Víctor Braberman, and Sebastian Uchitel. My model checker died!: how well did it do? In *Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*, pages 33–40. ACM, 2010.

[74] Andrew Piziali. *Functional verification coverage measurement and analysis*. Springer Science & Business Media, 2007.

[75] Zvonimir Rakamarić and Michael Emmi. Smack: Decoupling source language details from verifier implementations. In *International Conference on Computer Aided Verification*, pages 106–113. Springer, 2014.

[76] Sanjai Rayadurgam and Mats Per Erik Heimdahl. Coverage based test-case generation using model checkers. In *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the*, pages 83–91. IEEE, 2001.

[77] Ali Taleghani and Joanne M Atlee. State-space coverage estimation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 459–467. IEEE Computer Society, 2009.

[78] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In

ACM SIGSOFT Software Engineering Notes, volume 30, pages 253–262. ACM, 2005.

[79] Frank Tip. *A survey of program slicing techniques.* Centrum voor Wiskunde en Informatica, 1994.

[80] Julian Tschannen, Carlo Furia, Martin Nordio, and Bertrand Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. *Software Engineering and Formal Methods*, pages 382–398, 2011.

[81] Varun Tulsian, Aditya Kanade, Rahul Kumar, Akash Lal, and Aditya V Nori. Mux: algorithm selection for software model checkers. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 132–141. ACM, 2014.

[82] Antti Valmari. The state explosion problem. pages 429–528, 1998.

[83] Willem Visser, Corina S Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. 29(4):97–107, 2004.

[84] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[85] James A Whittaker, Jason Arbon, and Jeff Carollo. *How Google tests software.* Addison-Wesley, 2012.

[86] Valentin T Wüstholz. *Partial verification results.* PhD thesis, ETH Zurich, 2015.

[87] Michal Young and Mauro Pezze. Software testing and analysis: Process, principles and techniques. 2005.