TRANSLATION VALIDATION FOR COMPILATION VERIFICATION

BY

THEODOROS KASAMPALIS

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor Vikram Adve, Chair
Professor Grigore Rosu
Research Professor Elsa Gunter
Professor John Regehr, University of Utah

ABSTRACT

Modern optimizing compilers such as LLVM and GCC are huge and complex, and mature releases routinely have uncaught bugs. Beyond harm to software development, the lack of formal correctness guarantees for the compilation process seriously limits the guarantees other software systems can provide, since the compiler that generates the final executable cannot be trusted. These circumstances have motivated broad interest in *compilation verification*: providing a formal guarantee that a compilation of a program is correct.

Translation Validation is a commonly used compilation verification technique that aims to prove correctness of a single instance of compilation, by considering only the specific input and output programs and treating the compiler mostly as a black box. Translation Validation techniques are well-suited to the compilation verification problem because they can be composed to validate a sequence of compilation steps, they can easily retrofit to existing compilers, and they can be maintained independently from the compiler itself by a separate team of formal method experts.

The basic components of a Translation Validation system are (1) a formal notion of program equivalence, (2) a verification condition generator that generates a relation between program points and variables in the input and output programs, (3) a proof system that accepts the verification conditions, generates a machine-checkable equivalence proof, and checks the proof for correctness.

Ideally such a system is completely agnostic to the specifics of transformation from the input to the output as well as independent of the input/output languages. This allows the same system to be reused across the many transformation and translation passes found in modern compilers. However, this is not true in the state of the art: most existing systems are custom-tailored for a particular sequence of transformations, and moreover, specialized for a specific, common intermediate language for the input and output programs.

The overall goal of this work is to show that it is possible to develop a (mostly) language-independent, transformation-agnostic translation validation system with support for different input/output languages for an optimizing, production-quality compiler. In this thesis, we present such a system as well as the theoretical and practical advances needed to arrive to it.

First, we present a formal framework for program equivalence checking that is transformation-agnostic and language-independent. This framework can serve as-is as the proof system for any number of Translation Validation systems targeting different transformation and/or

translation phases within an existing compiler. The basis of the framework is a rigorous formalization, namely cut-bisimulation, for weak bisimulation variants that serves as a generalization of the various (sometimes ad-hoc) notions of program equivalence found in the literature. We develop a program equivalence checking algorithm that proves two programs equivalent by reducing a proposed relation between corresponding program states to a cut-bisimulation relation. We implement this algorithm in KEQ, a new tool for checking program equivalence that accepts the operational semantics of the input and output languages as parameters, and is independent of the transformation used to generate the output. This is the first program equivalence checking tool known to the authors that is language-parametric instead of containing hard-coded language semantics as is the norm in the literature.

Then, we use KEQ as the equivalence checker for two different Translation Validation systems targeting two phases of the LLVM compiler: the Instruction Selection phase and the Register Allocation phase. The two systems share the same notion of equivalence (cut-bisimulation), the same proof system (KEQ), as well as the semantic definitions for the input/output languages (LLVM IR and x86-64 based Machine IR), which are separate artifacts and not hardcoded into the logic of the systems. The only components that are transformation-specific are the two verification condition generators. The Instruction Selection one requires minimal support from the compiler in the form of compiler-generated hints, while the Register Allocation one is employing a novel inference algorithm for register allocation and related optimizations. These systems were evaluated on the GCC SPEC 2006 benchmark, where they correctly validated 4331 / 4732 (91.52%) and 4574 / 4732 (96.67%) functions with supported features respectively.

To my father, Georgios, my mother, Dimitra, my brother Vassilis, and my sister, Amalia.
To my fiancée, Maria.

ACKNOWLEDGMENTS

The completion of this dissertation and the successful conclusion of my Ph.D. would not have been possible without the help and support of many people to which I am most grateful. I would like first to thank my advisor, Professor Vikram Adve, for giving me the opportunity to pursue a Ph.D., and for being my mentor, an always reliable source of support, advice, and training. Vikram has always been able to steer me out of research dead-ends as well as challenge me to further improve my work when things were finally going according to plan. Even more importantly, Vikram has been there to support me psychologically during my graduate studies, always being helpful and understanding for things ranging from adjustment difficulties during my first months abroad to anxiety due to personal hardships.

I would like next to thank Professor Grigore Rosu for acting as a second advisor to me, giving me encouragement, feedback and direction for my research. Beyond providing technical feedback, Grigore has always been enthusiastic about this work and has helped me see how it can have real-world impact. His stance towards the work has kept me inspired and motivated as well as helped me overcome those unavoidable moments of doubt.

I would also like to thank all members of my Ph.D. committee for their valuable insights and feedback towards improving this work.

There were several fellow students that helped me in various ways during my graduate studies. I would like first to thank Daejun Park for collaborating with me on this work. This work would not have been completed without him. Both his technical contributions and his insights and ideas are an invaluable and inseparable part of this work. I would also like to thank Zhengyao Lin and Sandeep Dasgupta which have contributed to different aspects of the work. Collaborating with both of them has been a highlight of my graduate years. Finally, I would like to thank Nathan Dautenhahn (a student at the time, and an Assistant Professor as of the time of this writing) that helped me immensely during my first years as a graduate student. Nathan helped me understand what it takes to become a successful researcher both theoretically, through discussion and providing of resources, and in practice, through a collaboration that was a shaping experience for my following research years, even though it did not become part of this dissertation.

I would like to thank my dear friends and fellow students, Hassan Eslami and Shadi Abdollahian Noghabi. As fellow students, Hassan and Shadi were co-travellers on my graduate studies journey, and as close friends, they made this journey much more enjoyable and easy for me: they provided me with advice, help, and fun distractions along the whole way.

I would like to thank my fiancée, Maria, who has been with me since our undergraduate years and through our graduate studies, through countless happy moments, as well as several hard and sad ones. I cannot imagine my life without her and I cannot think I would be able to complete this Ph.D. without her unconditional love and support.

Most of all, I would like to thank my family. The hard work, sacrifice, and never-ending support of my parents are and will always be solely responsible for all of my achievements, including my doctoral graduation. The love, encouragement, and the happy childhood memories I have from my brother and sister have always been a driving and motivating force. I thank and love you all.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Compilation Verification and Why it is Necessary 1
1.2 Challenges for Compilation Verification 2
1.3 Summary of the State of the Art 3
1.4 Our Goal for Compilation Verification 6
1.5 Overview of Our Approach: Modular Black-Box Translation Validation . . . 7
1.6 Contributions 9
1.7 Dissertation Outline 11

CHAPTER 2 RELATED WORK 12
2.1 Translation Validation 12
2.2 Verified Compilers 18
2.3 $\mathbb{K}$ Framework and Other Formal Semantic Definition Frameworks 19

CHAPTER 3 CUT-BISIMULATION AND KEQ 21
3.1 Background on Bisimulation Theory 21
3.2 Towards Formalizing Program Equivalence 23
3.3 Cut-Bisimulation 26
3.4 Comparison to Stuttering Bisimulation 31
3.5 KEQ, A Language-Parametric Program Equivalence Checker 36
3.6 Preliminary Evaluation of KEQ 43
3.7 Equivalence Proof Examples 45

CHAPTER 4 TRANSLATION VALIDATION FOR INSTRUCTION SELECTION 49
4.1 Introduction and Motivating Example 49
4.2 Translation Validation for LLVM Instruction Selection 51
4.3 Evaluation on Compilation of Real-World Code 61

CHAPTER 5 TRANSLATION VALIDATION FOR REGISTER ALLOCATION . . 66
5.1 Introduction 66
5.2 The Reusable Translation Validation Components for Register Allocation . . 67
5.3 Black Box Synchronization Point Generator for Register Allocation 70
5.4 Evaluation of the TV System for LLVM's Register Allocation 81

CHAPTER 6 LESSONS LEARNED AND FUTURE WORK 86
6.1 Lessons about Designing A Translation Validation System for Modern Compilers 86
6.2 Translation Validation for the Whole LLVM to x86-64 Compilation Path . . 89
6.3 Towards Transformation-Agnostic Inference of Synchronization Points 90
6.4 Using Translation Validation to Discover Compiler Bugs 90

CHAPTER 7 CONCLUSION 92

REFERENCES 94

CHAPTER 1: INTRODUCTION

1.1 COMPILATION VERIFICATION AND WHY IT IS NECESSARY

Compilers are programs that transform an input program written in a source programming language to an output program written in a (potentially different) target programming language, a process known as *compilation*. The most common application for compilers is the translation of programs written by humans in a high-level programming language (e.g C/C++, Java, Haskell, etc.) to a hardware instruction set (e.g. x86-64, ARM, PowerPC, etc.) native to the target machine that executes said programs. The most general definition of compilation also includes program transformations that preserve the input language to the output, such as code refactoring, optimization transformations, and others.

A compilation process is expected to preserve the semantics of the input program in the the output program: existing behaviors in the input should not be altered in the output and no new behaviors should be added in the output. In other words, the input and output programs should be *equivalent* (for some well-defined notion of program equivalence). Despite its importance for the correctness of the compilation process, the input-output program equivalence is only a best-effort goal for most of today's modern compilers. Indeed, *most modern compilers do not provide any formal correctness guarantee for the compilation process.*

On the other hand, modern optimizing compilers such as LLVM [1] and GCC [2] have evolved into intricate systems with huge code bases and, consequently, uncaught bugs that make it into mature releases [3]. The reality of compilation bugs combined with lack of any formal correctness guarantee for the compilation process hinders software development and limits the guarantees other software systems can provide. Following are two examples indicative of the necessity of formal correctness guarantees for the compilation process.

Most iOS applications and all of the watchOS and tvOS applications are shipped by developers to the Apple Store as LLVM bitcode [4, 5] and they are compiled to machine code on Apple's servers, thus allowing the possibility of unintended behaviors introduced to the binary due to compilation errors. This has raised concerns among developers that the machine code may not be identical to what they tested before shipping and, even worse, that it may contain new, unexpected bugs that could not be caught during testing on the developer's site. [6]. A correctness guarantee for the compiler on the Apple server's site would alleviate these concerns.

As another example, the seL4 operating system microkernel [7] is verified on the source

code level to have various security and functionality properties. In an effort to guarantee the same properties for the microkernel ARM binary, the seL4 developers engineered a custom verification solution that completely removes the compiler from the trust base [8]. That would not be necessary if the compiler came with a formal correctness guarantee.

These circumstances have motivated broad interest in *compilation verification*: providing a formal guarantee that a compilation of a program is correct. The benefits from compilation verification are numerous and here we list the most important ones:

- Guaranteed correct compilation allows for formal guarantees for the input program to be transferred to the output program. This is especially helpful when the output program is written in a hard to analyze language such as the various instruction set architectures.

- Compilation verification allows for safe decoupling of the development and testing of applications from the actual compilation to a native binary. Developers can test and ship their application in a target-agnostic bytecode language such as the LLVM Intermediate Representation (IR) [9], while the user compiles the application locally optimizing for their native hardware and/or other custom specifications (energy consumption, binary size, etc.)

- A verified compilation system can be used to aid the debugging efforts for compilers as it will always report wrong compilations. Such reports can be used by compiler developers to fix underlying compiler bugs.

Note that compilation verification is subtly but importantly different than *compiler verification*. The latter refers to formally verifying parts of or the whole compiler for correctness. Compilation verification is a more general term that refers to any approach that provides formal guarantees for the correctness of the compilation process, including but not limited to using compiler verification.

1.2 CHALLENGES FOR COMPILATION VERIFICATION

The design of a compilation verification system for an optimizing, production-quality compiler is challenging because it should allow for a system that scales with the size of the target compiler as well as the size of the input application, is developed independently by formal verification experts while following the development of the target compiler, and has a low enough complexity to be trusted as opposed to the target compiler itself. In more detail, the main challenges for a compilation verification system are as follows:

- Existing modern compilers such as GCC [2] and LLVM [1] are composed of a host of different subsystems across hundreds of thousands of lines of code that transform the compiled code into many intermediate stages and languages before producing a final output targeting a number of different hardware instruction sets. A compilation verification system for such a compiler should be able to scale to the size of the code base and handle the multitude of code transformations as well as the various intermediate and final languages found in the compiler. This means that the verification system should employ *reusable components across different transformation and/or translation stages*, wherever possible.

- Similarly, *compilation verification should scale with respect to the input program size*. The verification should be modular so that compiling large input programs can be decomposed into verification of individual functions separately, for example.

- Moreover, modern compilers are actively developed on a daily basis and a compilation verification system should be able to keep up with the evolution of the corresponding compiler. On the other hand, compiler engineers are not verification experts and should not be expected to maintain the verification system along with the compiler development. This means that *the verification system should be developed separately from the compiler itself, and ideally, treat the compiler as black box.*

- A central component of any compilation verification system is the formal semantics of the language(s) involved in the compilation process. This semantics is the basis for any formal proof of equivalence. Of course such semantics are also hardcoded into the logic of the compiler and many compilation bugs are due to erroneous codification of the semantics into the compiler code. A compilation verification system should not similarly incorporate hardcoded semantics, but instead keep the formal semantics definition clearly separated and easily accessible by its users. Doing so increases the trust on the verification system, since users can access and more importantly test the formal semantics, that is the framework upon which the equivalence proof is built. Consequently, *the verification system should be decoupled from any language semantic definition(s) and ideally it should be completely parametric to it (them).*

1.3 SUMMARY OF THE STATE OF THE ART

In this section, we give a brief summary of the state of the art for compilation verification. We discuss the main approaches to the problem and their advantages and shortcomings

with the goal of identifying techniques appropriate for our goal and areas that need to be improved. A more detailed review of the related research can be found in Chapter 2. There are two different approaches to compilation verification: Translation Validation and Verified Compilers.

Verified Compilers A ideal approach to compilation verification is verified compilers: compilers with formally verified source code, as in CompCert [10], a verified compiler for C, and CakeML [11], a verified compiler for Standard ML. Verified compilers come with a formal proof of correctness for their implementation, meaning that they are proven to not miscompile their input. Verified compilers are attractive because they give a development-time guarantee of correctness, unlike Translation Validation systems that, as we will see, can only provide compile-time guarantees and suffer from potential false-positives.

The main limitation with the verified compiler approach is that it is at odds with the development life cycle of an existing compiler (e.g., LLVM and GCC) as opposed to a compiler designed from scratch for verification. First, retroactive full formal verification of the current code base of an existing, production-quality compiler, such as LLVM, is far beyond the state of the art. Second, the maintenance of the various proofs while the compiler code base evolves cannot be done by the compiler engineers. Instead, it requires active involvement of and coordination with verification engineers in a way that is not practical for these projects: Patches that update code in a way that makes existing proofs about said code outdated (i.e. failing) need to be reviewed by verification engineers to fix the failing proofs. That, in turn, may require further changes by compiler engineers in the patch that caused the proof failures.

In fact, all existing verified compilers have been built from the ground up with the goal of verification in mind (see CompCert [10], CakeML [11], and Chipala *et al.* [12] among others). These compilers are developed by formal methods experts and their development life cycle includes formal correctness proofs for any patch of new implementation, meaning that compiler engineering is tightly coupled with corresponding proof engineering. Although this requirement can sometimes hinder compiler development, formal correctness has higher priority for a verified compiler. For example, the register allocation phase of CompCert had to use a sub-optimal algorithm for one of its transformations (spilling) because the correctness proof for the preferable algorithm would be "a daunting task" [13]. A production-quality optimizing compiler that prioritizes performance could not adhere to such standards.

To sum up, the approach of verified compilers makes the compiler development and the compilation verification inseparable, and as a result, it does not suffice as a practical solution for compilation verification for existing compilers.

Translation Validation (TV) [14] It is clear from the discussion above that full static verification of the compiler source code itself has only been shown to work for compilers designed specially from the ground up for formal proofs, and requires extensive proof engineering by formal methods experts. On the other hand, compilation verification systems based on Translation Validation are focusing on validating the correctness of a single instance of compilation. The system accepts a pair of input and output programs and potentially some compiler-generated information about the transformation that took place (e.g. the correspondence between input and output variable names) and either validates the transformation, providing a formal certificate of correctness, or rejects it. Typically, a TV system for compilation verification is sound with regard to validation, meaning that a validated transformation is guaranteed to be correct (i.e. no false-negatives in validation). Such systems are incomplete, meaning they can reject a correct transformation (i.e. generate a false-positive).

Translation Validation techniques are attractive for compilation verification because they can easily be retrofitted to existing compilers. By operating solely on the input/output programs of the compilation and utilizing only zero to limited support from the compiler for transformation-specific information, a TV system for compilation verification treats the compiler largely as a black box and can be developed independently from it. Moreover, multiple TV systems addressing different transformation and/or translation phases in the compiler can be composed to validate the whole compilation path, thus scaling with the complexity of a production-quality compiler.

The ideal TV system would have several key characteristics, many of which have been demonstrated for realistic production compilers. First, a TV system should be scalable and powerful enough to handle large, production-quality optimizing compilers compiling large, real-world application programs (see first and second challenge in Section 1.2). Second, a TV system should require few or no changes to the compiler functionality itself, so that there are no compromises to the quality or debuggability of generated code, or to compilation times. Third, a TV system should require relatively few extensions to the compiler to generate the verification conditions that guide the formal correctness proof, and importantly, it should be possible to generate the verification conditions with only standard compiler techniques and not sophisticated formal verification techniques, since the skills to develop and maintain such techniques are almost always lacking in real-world compiler teams (see third challenge in Section 1.2). Fourth, TV techniques should be possible to use in a modular fashion, validating separate (sequences of) transformations, which can be then composed to provide a correctness guarantee for the whole compilation sequence (again see first challenge in Section 1.2). Together, these characteristics would ensure that the TV approach is very well suited for verification of existing compilers.

There is a rich literature of successful TV systems for compilation verification, for example [15] for GCC and [16] for LLVM, among others. The main limitation of these systems is that each of them is custom-tailored for a particular sequence of transformations, and moreover, specialized for a specific, common intermediate language for the input and output programs. The fixed language makes it difficult to use the approach for modern compilers, which typically use different intermediate languages for different stages (e.g,. LLVM uses at least three different languages). For example, Necula's work on GCC [15] is limited to the lowest-level IR form, Register Transfer Language (RTL), and does not apply to the bulk of the optimizations which are done on the higher-level GIMPLE representation [17]. Moreover, none of these previous systems would be able to verify a key phase like Instruction Selection in LLVM, which converts between two different IRs. The best effort has been to translate both input and output programs to a third, common internal representation as a preliminary step [8], which introduces two new unverified language translators in order to verify the original translator.

In short, existing TV systems for verified compilation successfully address the separation from the compiler challenge, but fall short of being modular and decoupled from the input/output language semantics. Due to these significant weaknesses, existing TV systems cannot at this point be regarded as a satisfactory general solution to the problem of compilation verification.

1.4 OUR GOAL FOR COMPILATION VERIFICATION

In this work, we attack the problem of *compilation verification*. We do that not only theoretically, as an instance of program equivalence, but from a practical standpoint as well: We aim for a solution suitable for optimizing, production-quality compilers such as LLVM. Our solution should overcome all the aforementioned challenges to be practical. Specifically, we propose a compilation verification system design based on translation validation that is (a) modular with as many reusable transformation-agnostic and language-independent components as possible, (b) separate from the compiler, and (c) decoupled from any semantic language definitions which are separate artifacts.

In short, the thesis of this work is to show that it is possible to develop a (mostly) language-independent, transformation-agnostic translation validation system with support for different input/output languages for an optimizing, production-quality compiler.

In this dissertation, we present such a system as well as the theoretical and practical advances needed to arrive to it. The system targets the LLVM compiler backend for the x86-64 Instruction Set Architecture (ISA) [18], specifically the Instruction Selection phase [19]

that translates LLVM IR to x86-64 assembly as well as the Register Allocation [20] phase that transforms the code so that it uses only the physical register file for x86-64 by eliminating references to virtual registers (i.e. temporary names generated during previous compilation stages). These are intricate, major components of the LLVM compiler backend that each span above 100,000 lines of code and operate on two different languages: the LLVM IR and the x86-64 machine IR [21], a low-level intermediate representation parameterized by the opcodes and operand types of the x86-64 ISA. The semantic definitions of these languages are given as independent artifacts, while the translation validation system consists of a proof system module that is reused across transformations and two transformation-specific verification condition generators.

1.5 OVERVIEW OF OUR APPROACH: MODULAR BLACK-BOX TRANSLATION VALIDATION

In this work, we focus on a compilation verification approach based on *Translation Validation* [14], since it is proven to be better suited to existing compilers, as discussed in the previous section. Translation Validation aims to prove correctness of a single compilation run, by considering only a specific pair of input and output programs. A TV system treats the compiler largely as a black box, can be developed independently from compiler, can focus on a specific (sequence of) transformation(s), and can be easily combined with other TV systems to validate the whole compilation path.

These properties make TV techniques practical for compilation verification and are the reason we focus on TV for achieving the goals of this work. For the rest of this section, we discuss how to utilize the strengths of TV to arrive to a system that is modular with as many reusable across the compilation path components as possible. There are three essential components of a TV system:

1. A formal notion of program equivalence.

2. A verification condition (VC) generator that generates a sufficient set of obligations to be discharged in order to prove equivalence. Verification conditions relate program points and variables in the input and output programs.

3. A proof system that accepts the verification conditions, generates a machine-checkable equivalence proof, and checks the proof for correctness.

The key insight underlying our work is that two of the three TV system components mentioned above can be generalized to be *transformation-agnostic and language-independent*:

the formal notion of equivalence, and the proof system. The only component that needs to depend on specific transformations is the VC generator, and that is conceptually the simplest because it requires little formal methods expertise, making our approach suitable for real-world compiler teams which typically lack such expertise. Together, these make our approach far more practical than previous solutions.

More specifically, we design a program equivalence checker, KEQ, that can be used *unchanged* for different transformation passes and input/output language pairs. KEQ accepts operational semantics definitions of the input and output languages as parameters, as well as the VC for a transformation sequence. The operational semantics of each IR must be defined once, and KEQ can then be reused across all the transformations found in the compilation path. Moreover, the input and output languages can be completely different, as long as programs can be related using the verification condition.

In this work, we showcase the power of these properties by using KEQ in a prototype TV system for the Instruction Selection phase of LLVM, a sophisticated phase that translates LLVM IR [9] to Machine IR [21] representing the x86-64 instruction set. We are also using KEQ *unchanged* in another prototype TV system for the Register Allocation phase of LLVM, with a VC generator that treats the allocator completely as a black box (i.e, has no knowledge of the allocation algorithm), and we plan to apply it to LLVM-to-LLVM transformations in future.

We provide a strong theoretical foundation for KEQ and its correctness. First, we present a formalization for program equivalence which we call cut-bisimulation and is a variant of weak bisimulation [22]. Cut-bisimulation is weak enough to enable proofs for realistic compiler transformations, and yet expressive enough to subsume most of the equivalence properties that have been used in existing TV systems. In fact, previous work has used the intuition behind the cut-bisimulation formalism for applications on program equivalence, but none of them have formalized it as a general framework suitable for program equivalence. Cut-bisimulation can express equivalence of programs in two different languages, as long as a verification condition can relate program states in the two languages. We then use cut-bisimulation to define an equivalence checking algorithm that forms the theoretical basis for KEQ.

Given those, only the VC generator needs to be designed per transformation (or sequence of transformations). Such generators need to provide a candidate relation between input and output program states that the proof system can verify to be indeed a cut bisimulation relation. A VC generator typically needs information about the effects of the specific transformation(s) on the input. This information can either be provided by the compiler or inferred, and both approaches have been explored in the literature [15, 23]. Compiler hints

are more accurate while requiring some modification in the compiler code, while inference algorithms can generate more false positives but treat the compiler as a complete black box. We add a small number of compiler hints in the Instruction Selection phase of LLVM for our VC generator for this phase. On the other hand, we use inference techniques in the VC generator for the Register Allocation phase, thus making the TV system for Register Allocation mostly [1] black-box.

In short, this work presents the first TV system with a language-independent and transformation-agnostic proof system. The system requires the minimum amount of customization per transformation and (intermediate) language: a semantic definition of every language found in the compilation path and one or more VC generators that use transformation-specific information. Note that each semantic definition need only to be defined once for all transformations that use the corresponding language as input and/or output.

1.6 CONTRIBUTIONS

The contributions of this work can be broken down to two categories: Contributions in the area of program equivalence checking, and contributions in the area of compilation verification.

Contributions in Formal Program Equivalence Checking The main contribution here is a framework for program equivalence checking[2] that is transformation-agnostic and language-independent. This framework can serve as-is as the proof system for any number of TV systems targeting different transformation and/or translation phases within an existing compiler. Specifically:

1. A rigorous formalization, namely *cut-bisimulation*, for weak bisimulation variants that have been traditionally used in different TV systems. Cut-bisimulation is essentially a (strong) bisimulation over a set of "cut" states. The cut is a subset of the program states that satisfies certain well-defined properties. We also present a proof that stuttering bisimulation can be reduced to cut-bisimulation. As such, cut-bisimualtion serves as a generalization of the various (sometimes ad-hoc) notions of equivalence found in the literature.

[1] At this point, the only information required from the compiler is the number of arguments in each callsite.

[2] Our framework supports checking for program refinement in addition to program equivalence. For refinement, the notion of bisimulation is replaced by the notion of simulation. In the following text, wherever a claim is made about equivalence/bisimulation, a corresponding claim is also true for refinement/simulation.

2. A *program equivalence checking algorithm along with a correctness proof* based on cut-bisimulation. The algorithm proves two programs equivalent by reducing a proposed relation between corresponding program states to a cut-bisimulation relation.

3. KEQ, *a new tool for checking program equivalence that accepts the operational semantics of the input and output languages as parameters*, and is independent of the transformation used to generate the output. This is the first program equivalence checking tool known to the authors that is language-parametric instead of containing hard-coded language semantics as is the norm in the literature. KEQ implements the cut-bisimulation based equivalence algorithm employing symbolic execution driven by the given semantic definitions. KEQ is implemented as a tool within the $\mathbb{K}$ Framework [24].

Contributions in Compilation Verification This work presents the first TV system for Compilation Verification that has a language-independent and transformation-agnostic proof system and requires the minimum amount of customization per transformation and (intermediate) language: a semantic definition of every language found in the compilation path and one or more verification condition generators that use transformation-specific information. Specifically:

1. *A prototype of a Translation Validation system for the Instruction Selection pass of the LLVM compiler infrastructure*, able to automatically prove equivalence for translations from LLVM IR when compiling to the x86-64 instruction set. This is a mature, sophisticated translation phase of a production compiler. Moreover this is a transformation that uses different input and output languages, and as such, avoids the problem in previous systems of defining translators to a third, common language, e.g. in [8, 25]. This prototype employs KEQ for its equivalence proof system and a custom verification condition generator for Instruction Selection. The latter relies on a minimal hint generator added to the LLVM compiler.

2. *A black-box inference algorithm for verification conditions required for Register Allocation.* The algorithm supports important optimizations such as spilling, live-range splitting, register coalescing, and rematerialization, all of which can be found in production-quality compilers such as LLVM and GCC. The algorithm automatically discovers the correspondence between virtual registers in the input and physical registers and/or spill memory locations in the output programs. The inference algorithm is the base of *a prototype Translation Validation system for the Register Allocation phase of the*

LLVM compiler. The system is able to prove equivalence for programs transformed by register allocation, along with a set of related optimizations: spilling, live range splitting, register coalescing, and rematerialization. The system is implemented with minimal effort by reusing the exact same equivalence proof system based on KEQ, that is utilized by the TV prototype for Instruction Selection. The only new component is the verification condition generator, which implements the aforementioned inference algorithm and requires no additional compiler-generated information.

3. We evaluate the Instruction Selection TV prototype on 4732 functions of the GCC SPEC 2006 [26] benchmark with supported features. We correctly validate the translation of 91.52% of the supported functions in GCC, i.e., 4331 / 4732 functions. We evaluate the Register Allocation TV prototype on 2815 functions of the GCC SPEC 2006 benchmark with supported features. We correctly validate the translation of 96.67% of the supported functions in GCC, i.e., 4574 / 4732 functions.

4. *Formal semantic definitions for the LLVM IR and the x86-64 based Machine IR*. These definitions are implemented in $\mathbb{K}$ and are used to parameterize KEQ for our TV systems for Instruction Selection and Register Allocation. These definitions are independent artifacts that are useful beyond their TV application here. Thanks to the $\mathbb{K}$ Framework paradigm, these definitions can be used to generate various tools for their respective languages, such as parsers, interpreters, proof checkers, etc.

1.7 DISSERTATION OUTLINE

The remainder of this dissertation is organized as follows. In Chapter 2, we review the current state of the art for compilation verification based both on TV and verified compilers. We also give a brief survey of the most common bisimulation variants in the literature. Chapter 3 presents our rigorous Cut-Bisimilation formalization for program equivalence proofs, as well as KEQ and its equivalence checking algorithm. Chapter 4 presents the TV system prototype for Instruction Selection and Chapter 5 presents the TV system prototype for Register Allocation along with the VC inference algorithm. Finally, Chapter 7 presents the main conclusions and take-aways from our work and Chapter 6 gives directions for future work.

CHAPTER 2: RELATED WORK

In this Chapter we discuss related work for program equivalence and compilation verification. We first look into translation validation and the state of the art for systems that employ it to verify the compilation process. Then we discuss the alternative approach of verified compilers. Finally, we give a brief presentation of the $\mathbb{K}$ framework and language semantic definitions in $\mathbb{K}$.

2.1 TRANSLATION VALIDATION

Translation Validation as a method of verifying the correctness of a compilation first proposed by Samet [27] and reformulated by Pnueli *et al.* [14]. Following is an incomplete list of the various applications of Translation Validation on compilation verification aimed to provide a high-level overview of the state of the art. Translation validation has been used to prove correctness of specific compiler optimization passes [15, 16, 23, 28, 29, 30, 31], discover compiler bugs [32], and to prove correctness of end-to-end compilation [8, 14, 25]. VOC-64 [28] for the SGI Pro-64 compiler, Necula *et al.* [15] for the GNU C compiler, Peggy [29] for the Soot Java bytecode optimizer, LLVM-MD [16] and Namjoshi *et al.* [23] for LLVM, are all tools that perform translation validation for specific optimization passes in their respective compiler. Sewell *et al.* [8] presents a translation validation approach for the compilation of the seL4 kernel from C to binary. Dahiya *et al.* [25] presents a translation validation system for a collection of C compilers (GCC, `clang`, ICC, and CompCert). Hawblitzel *et al.* [32] uses translation validation to determine whether assembly code produced by different versions of the CLR JIT compiler are semantically equivalent and thus report miscompilations when there are differences. PEC [30] applies TV techniques in pairs of partially specified programs, where such a pair describes a general optimization on all the corresponding concrete programs. DDEC [31, 33] is an equivalence checker for x86 loops that uses data collected from test runs rather than inference or hints to construct a simulation relation.

2.1.1 Formal Notion of Program Equivalence and Cut-Bisimulation

The proof of program equivalence in the majority of these translation validation tools [8, 14, 15, 28, 30, 31, 32] is based on generating sets of verification conditions, the satisfiability of which is enough to prove equivalence. The verification conditions are produced as a combination of invariants that have to be inferred and a refinement requirement that is defined in a

slightly different way in the context of each work. All these various refinement requirements attempt to capture some weak notion of simulation suitable for compiler translation validation. In Chapter 3 we claim that cut-bisimulation is exactly the appropriate bisimulation variant for practical use in this field, and that in fact, any of these refinement requirements can be expressed as a cut-simulation proof requirement. For instance, the equivalence proof rule used to generate the refinement requirement in VOC-64 [28] is reminiscent of our notion of cut-similarity, but is expressed using syntactic devices (such as basic blocks and paths in the control flow graph) that unnecessarily restrict its generality and distance it from classic bisimulation theory. Although we do not claim that the intuition behind cut-bisimulation is entirely novel, we do claim that cut-bisimulation formally captures the essential properties required for compiler translation validation, and moreover, enables proof systems to be parameterized by the operational semantics of the input and output languages.

Namjoshi *et al.* [23] uses a variant of stuttering-bisimulation (see Section 3.1) with ranking functions, first introduced in [34]. Informally, the ranking function returns an integer rank for each pair in the relation which represents how many times one of the transition systems is allowed to stutter while the other advances before the former has to advance in order for the systems to reach another pair of related states. This variant requires matching single transitions only, similarly to strong bisimulation and unlike classic stuttering bisimulation, where a single transition may have to be matched with a finite but unbounded number of transitions, thus leading to large number of generated proof requirements. Cut-bisimulation shares the same property of matching single transitions only and is more appealing for proof automation, since the proof generator does not need to produce ranking functions in addition to the synchronization points.

Hur *et al.* [35] presents the relation transition systems (RTS) as a technique for program equivalence proofs suitable for ML-like languages, that combine features such as higher-order functions, recursive types, abstract types, and mutable references. Bisimulation is used as part of the RTS equivalence proof technique. Our notion of cut-bisimulation is orthogonal to RTS and it can be the notion of bisimulation of choice within an RTS equivalence proof. More specifically, our notion of acceptability relation $\mathcal{A}$ is similar to the global knowledge relation used in bisimulation proofs within the RTS proof. However, whereas a global knowledge relation contains a subset relation (named local knowledge) that should be proven to consist only of equivalent pairs, an acceptability relation is assumed from the start to only contain equivalent pairs: this is unavoidable when we want to do an inter-language equivalence proof, since the knowledge of what states are considered equivalent is indispensable for even to define what it means for programs written in different languages to be equivalent. The authors argue that RTS is a promising technique for inter-language proofs that involve ML-

like languages (although they leave the claim as future work), and we believe that the notion of cut-bisimulation can indeed help towards enabling RTS-style inter-language equivalence proofs.

2.1.2 Graph-Based Notions of Equivalence

LLVM-MD [16] and Peggy [29] move away from simulation proofs, and instead use graph isomorphism techniques to prove equivalence. Both tools operate on a graph-based representation of the LLVM IR. Program transformations can then be reduced to a series of simple trusted graph transformations. Peggy applies the set of trusted transformations while exploring various transformed forms of the input in order to discover optimized versions without employing any complex optimizations. LLVM-MD attempts to transform both the input and output program value-graphs to isomorphic graphs, thus proving equivalence. In both cases, the set of trusted graph transformations is crucial to the effectiveness of the approach. These transformations are inspired by the effects of actual compiler optimizations on the code.

Another example of graph isomorphism employed for program equivalence is found in Dasgupta *et al.* [36]. The work presents a translation validation approach for decompilers. It uses dataflow graph isomorphism to prove equivalence between original LLVM IR code and its counterpart that has been lifted to LLVM IR from a compiled binary with a decompiler and then passed through a canonicalizer. The canonicalizer is expensive, because it uses auto-tuning to find sufficient LLVM sequences, and not guaranteed to work because it may not find an LLVM sequence that results in isomorphic dataflow graphs.

2.1.3 Hints vs Heuristics for Verification Condition Generation

Our proposed algorithm takes as input a relation between program points in the input and output languages. To generate this relation, our implemented prototype for the LLVM Instruction Selection phase uses compiler-generated hints, similar to the witnesses introduced in [23]. Other works discuss various heuristics that can be used instead. In particular Necula *et al.* [15] describes an inference algorithm to generate both a relation between program points and the accompanying constraints between program variables and memory locations for two functions when any number of compiler transformations have been applied to the original function to produce the transformed function. The algorithm uses transfer functions, generated by symbolic execution, to describe the effect of each basic block. It then scans the two programs using said transfer functions and collecting constraints that make the symbolic

input states of corresponding blocks equal. Working towards a language independent proof generator, it is possible that one can derive a language independent version of this inference algorithm by implementing to be parametric in the language semantics in a fashion similar to KEQ. A similar inference algorithm is presented by Dahiya *et al.* [25], which is also enhanced with back-tracking to handle cases where there is no clear correspondence between basic blocks and jumps in the two programs. Finally, DDEC [31] uses a combination of static analysis and data-driven inference for constructing simulation relations: Static analysis is used to determine the program locations of synchronization points and the live variables while the constraints between variables are inferred from execution traces.

2.1.4 Language-Independence in Translation Validation Systems

All the previous work on Translation Validation assumes that the input and output programs are either written in or are translated to a common language or representation: GNU RTL [15], LLVM IR [23], value graphs [16, 29], x86 [31], Boogie IR [32, 37], a C-like intermendiate language for PEC [30], and a common representation called Transition Systems [14, 28]. Even the translation validation approach for the seL4 kernel proposed in [8] requires translation of the input C code and decompilation of the output binary to a common graph language used for equivalence checking. Finally, the fully black-box equivalence checker proposed in [25] requires the input C code to be lowered and the output binaries to be lifted to a common intermediate representation, namely transfer function graphs.

On the other hand, our equivalence checking algorithm is parametric to the input and output language semantics, thus generalizing the original approach of Pnueli *et al.* by eliminating the requirement for a common semantic framework. This makes it much easier to validate translations between two different languages (e.g., as in Instruction Selection), because it does not require unchecked translations into a common language. Our program equivalence checker, KEQ, is the first such tool we know of that is truly language-independent.

Our equivalence checking algorithm was inspired from the language-independent proof system for mutual equivalence introduced in [38]. Instead of a proof system, in our work we propose a bisimulation relation and an algorithm based on it and symbolic execution, leading to the first language-independent implementation of a checker for equivalence between programs written in two different languages.

2.1.5 Translation Validation for Register Allocation

Unlike Instruction Selection, the Register Allocation phase of the compilation process

typically involves the same language for input and output and as such it has been the target of various compilation verification systems. Here we discuss translation validation works that handle the Register Allocation phase as well as related optimizations.

Huang *et al.* [39] propose a static analysis approach that guarantees no false alarms and generates informative messages when detecting errors in the output of the allocator. This approach is useful for discovering bugs in the compiler but cannot be used in a setting where formal guarantees for the compilation process are needed, since the underlying static analysis does not guarantee correctness when no errors are reported. Moreover, the analysis is not accompanied by a correctness proof. Such proof does not seem trivial as it would require a formal definition of global value numbering, a complex analysis that is part of this system's static analyser.

On the other hand, Rideau *et al.* [13] present another static analysis approach for Register Allocation that is proven correct in both algorithm and implementation (using Coq), and provides a formal guarantee of correctness for validated compilations. This system is used in the CompCert verified compiler [10] as a replacement for a verified register allocator, since the latter was constrained to a rather naive spilling heuristic that negatively impacted the quality of the compiled code. The Translation Validation approach allows CompCert to use an untrusted and more aggressive spilling algorithm, while maintaining the correctness guarantee for the compilation result. The static analysis used in this work makes similar assumptions to our inference algorithm (i.e. 1-1 correspondence of basic block and non-copying instructions) and manages to validate register allocated code without any symbolic execution. However, the analysis (and its correctness proof) are specific to register allocation, while the KEQ equivalence algorithm (and its correctness proof) can be reused for validation of different transformations[1]. In short, the static analysis in [13] has the advantage that it is a lightweight and fast analysis, but would be impractical to design separately for all the phases of an existing production compiler. Our approach is better suited for existing compilers, by entirely reusing the theoretical results and tools across different transformations.

Nandivada *et al.* [40] proposes RALF, a framework for easy development and evaluation of register allocation algorithms. The framework consists of two languages, MIRA and FORD, and a type system for checking correctness of register allocation. MIRA is an intermediate level language designed to represent programs before register allocation. MIRA programs contain architectural information such as the register file, calling convention, etc., as well as static analysis information such as def-use chains, control flow, etc. FORD is a language for

[1]The register allocation specific part of our system, our inference algorithm and the VC generator, is not proven correct but need not to be trusted: KEQ will reject bogus synchronization points as explained in Subsection 4.2.7.

register allocation directives such as directives for spills, register/variable mappings at different program points etc. Finally, the type system is designed to ensure that a type-correct FORD program preserves the values alive in the underlying MIRA program. The proposed framework is mainly intended for fast implementation and testing of register allocation algorithms, since it allows for easy plugging of said implementations into the GCC compilation path and offers built-in validation of the allocator's output. It cannot however be used for translation validation of an existing register allocator within a production compiler, as it only supports allocators that work with the MIRA/FORD intermediate representations.[2] On the other hand, our proposed design focuses on existing compilers and it can be applied to the register allocator of production compilers such as LLVM without any modification of the allocator's code.

Necula *et al.* [15] proposes a Translation Validation system for the GCC compiler that tackles Register Allocation as one of the many supported transformations. Similar to our approach, the system uses no compiler-generated information but rather employs an intricate inference algorithm to produce equality constraints for each basic block of a function. The inference algorithm uses transfer functions to describe the effect of each basic block, which are generated using symbolic execution of the RTL representation on which the system operates. In comparison, our inference algorithm is more light-weight (only a simple static dataflow analysis, no need for SMT solvers) since we are able to focus our effort on a specific transformation as opposed to a wide set of optimizations. By using a general equivalence checker, KEQ, we are essentially refactoring the demanding workload of such automated equivalence proofs (symbolic execution, SMT solvers) out of the inference logic that drives the verification condition generation and needs to take transformation-specific characteristics into account. This is especially important because the verification condition generator is the only part that must be modified as passes are modified or added to a production compiler, and a simple (yet automatic) generator like ours is far easier for compiler teams to develop and maintain than one that requires sophisticated knowledge of theorem proving techniques.

Finally, Sewell *et al.* [8] and Dahiya *et al.* [25] tackle Register Allocation as one of the many optimizations handled by the respective TV systems. Since these systems are designed to validate end-to-end compilations, they are significantly more complex and heavyweight than our verification condition generator, that specializes on Register Allocation.

[2]Of course, a type-checker using the proposed type system could be implemented for the intermediate representation of a target production compiler to be then used for validation of its register allocator.

2.2 VERIFIED COMPILERS

One approach to the problem of compiler verification is the full formal verification of the compiler, as in CompCert [10], CakeML [11], Jinja [41, 42], and the lambda calculus to typed assembly compiler in [12]. Full formal verification is attractive because it gives an ahead-of-time guarantee of correctness for all input programs, whereas translation validation approaches detect errors only when actually compiling programs and are also susceptible to false alarms.

However, so far this approach has only been used for compilers built from the ground up with the goal of verification in mind. For example, CompCert [10], a verified compiler for C, has been written in the Coq proof assistant's specification language [43]. The approach requires extensive manual effort ("proof engineering"), and much greater expertise in formal methods than is usually available in production compiler teams. Such design decisions and development processes cannot easily be applied retroactively in existing compilers.

Instead of retroactively verifying an existing compiler, verification efforts for such compilers have focused on formal verification of specific compiler transformation passes. Zhao *et al.* presents a framework for formal verification of SSA-based transformations in the LLVM compiler [44]. The framework utilizes the Vellvm semantics of the LLVM IR [45], a formal semantics written in Coq, and it allows formally defined transformations (that come with correctness proofs) to be extracted as LLVM passes that the can be used by the actual compiler. Alive [46] is a framework for formally describing peephole optimizations on the LLVM IR. Describing an existing peephole optimization in Alive allows the compiler engineer to formally verify the correctness of said optimization. Moreover, Alive can export formally proven correct optimizations as C++ code to used by the LLVM peephole optimizer. Finally, Peek [47] is a framework for expressing, verifying, and running assembly-level peephole optimizations in the x86 backend of CompCert.

Beyond verifying existing compiler passes and/or exporting correct implementations for compiler transformations, efforts such as the above have revealed inconsistencies in the hardcoded semantics logic that the compiler uses to apply different transformations. For example, Lee *et al.* [48] discuss such inconsistencies in the semantics of the *undef* and *poison* values of the LLVM IR across optimizations found in the compiler.

2.2.1 Correct-by-Construction Compilers

Going one step further from generating verified compiler transformations for specific compilers, there have been efforts to automatically generate correct-by-construction compiler

transformations in a language-independent way. TRANS [49, 50] is a specification language for compiler transformations, where a transformation is described as rewrites on the Control Flow Graph (CFG) [51] with side conditions expressed as temporal logic formulae [3] over paths of the CFG. The semantics of the TRANS specification language along with semantics for the involved input/output languages provide a framework for formal correctness proofs of transformations specified in TRANS. Manksy *et al.* [54] has implemented the semantics for TRANS using the Isabelle proof assistant's specification language [55]. Moreover, in his thesis [56], Mansky presents PTRANS, a transformation specification language based on TRANS and expanded to support transformations on parallel programs, that has been given an executable semantics in addition to its Isabelle implementation. The executable semantics allows deriving prototype implementations for transformations described in PTRANS automatically from their PTRANS specification.

2.3 $\mathbb{K}$ FRAMEWORK AND OTHER FORMAL SEMANTIC DEFINITION FRAMEWORKS

The $\mathbb{K}$ Framework [24] is a rewriting-based framework with foundations on Reachability Logic [57] for defining executable semantic specifications of programming languages. Given the syntax and semantics of a language, K automatically generates a parser, an interpreter, as well as formal methods analysis tools such as a deductive verifier and a symbolic execution engine. This avoids duplication while improving efficiency and consistency. For example, using the interpreter, one can test the semantics immediately, which significantly increases the efficiency of and confidence in semantics development. The verifier uses the same internal model for verifying programs, and that confidence carries over.

There exists a rich literature on using $\mathbb{K}$ for formalizing existing languages, such as C [58, 59], Java [60] and JavaScript [61], among others. $\mathbb{K}$ has also been used to formally specify EVM [62], the current smart contract language for Ethereum [63]. In fact, the process of formalizing EVM as an executable semantics uncovered various inconsistencies and unspecified behaviors in its original English language specification [64]. In this work, we use $\mathbb{K}$ for our semantic definitions of LLVM IR and x86-64 based Machine IR to easily and automatically obtain symbolic execution engines for the languages. Moreover, we found it natural to implement KEQ as another language semantics parametric tool of the $\mathbb{K}$ framework.

[3] Linear Temporal Logic (LTL) [52] is the most commonly used such logic and is suitable for describing properties on a single path of the CFG. Computation Tree Logic (CTL) [53] supports specifying properties over a set of paths representing a branching tree of computation. CTL is the temporal logic used in TRANS.

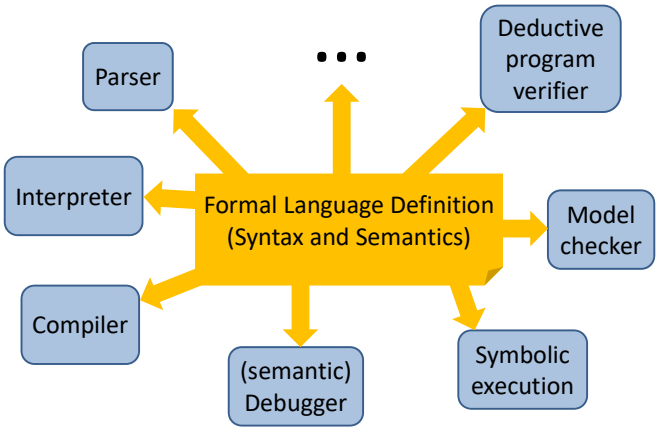


Figure 2.1: The K vision [24]

Beyond $\mathbb{K}$ and the $\mathbb{K}$ Framework, proof assistants such as Isabelle [55] and Coq [43] have been used to formally define language semantics as theories written in the assistant's specification language. The semantics then can be used within any verification effort supported by the proof assistant. For example, a semantic definition of an LLVM IR subset, namely miniLLVM, written as an Isabelle theory, is used along with PTRANS for the various transformation correctness proof included in the work [56]. Both Isabelle and Coq allow for exporting of such semantic definitions into executable form [4], thus making the definitions themselves executable. Some examples of real-world languages given formal, executable semantic definitions using this approach include Vellvm [45], a semantics for LLVM IR in Coq, CoqJVM [65], a semantics for the JVM in Coq, and Jinja [41, 42], a semantics for a large subset of Java and Java threads in Isabelle. The latter work also includes semantics for compilation from Java to JVM bytecode along with a correctness proof for the compilation. Since the compilation semantics is also executable, it has been used to derive a verified compiler for Java.

[4]Typically as OCaml and ML code respectively.

CHAPTER 3: CUT-BISIMULATION AND KEQ

We propose an algorithmic semantics-based approach for proving equivalence of programs written in possibly different languages. We introduce a new notion of bisimulation, named *cut-bisimulation*, that allows the two programs to semantically synchronize at relevant "cut" points, but to evolve independently otherwise. While being analogous, cut-bisimulation is different from stuttering bisimulation. We provide realistic counter-example programs that are cut-bisimilar but not stuttering-bisimilar, which can be easily found in a compiler optimization. Also, we identify a subclass of stuttering bisimulation that can be reduced to cut-bisimulation. Based on cut-bisimulation, we have implemented the first *language-independent* tool for proving program equivalence, parametric in the formal semantics of the source and target languages, built on top of the $\mathbb{K}$ framework. As a preliminary evaluation, we instantiated our tool with an LLVM semantics, and used it to prove equivalence of the aforementioned example programs written in LLVM.

This is joint work with Daejun Park and appears in his thesis [66].

3.1 BACKGROUND ON BISIMULATION THEORY

In this section, we present a brief introduction in bisimulation theory, which is a prerequisite for the work presented in the thesis. For more details on the classic bisimulation and its variants, we refer the reader to [22]. We use the following notations in this Section and the rest of the paper.

Binary Relations If $R \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is a binary relation, then we write $a\ R\ b$ instead of $(a, b) \in R$ and let $R_1 = \{a \mid \exists b\,.\, a\ R\ b\}$ and $R_2 = \{b \mid \exists a\,.\, a\ R\ b\}$ denote the projections $\Pi_i(R)$, where $i \in \{1, 2\}$.

Transition Systems Let $\mathcal{S}$ be a set of states (thought of as all possible configurations/states of a language, over all programs in the language). Let $T = (S, \xi, \rightarrow)$ be an *$\mathcal{S}$-transition system*, or just a *transition system* when $\mathcal{S}$ is understood, that is a triple consisting of: a set of *states* $S \subseteq \mathcal{S}$, an *initial state* $\xi \in S$, and a (possibly nondeterministic) *transition relation* $\rightarrow\ \subseteq S \times S$. Let $next(s)$ denote the set $\{s' \mid s \rightarrow s'\}$. T is *finitely branching* iff $next(s)$ is finite for each $s \in S$. Let $\rightarrow^*$ be the reflexive and transitive closure of $\rightarrow$, and $\rightarrow^+$ be the transitive closure of $\rightarrow$.

Traces A (possibly infinite) *trace* $\tau = s_0 s_1 \cdots s_n \cdots$ is a sequence of states with $s_i \to s_{i+1}$ for all $i \geq 0$. Let $\tau[n]$ be the n^{th} state of τ where the index starts from 0, and let $\mathsf{size}(\tau)$ be the length of τ (∞ when τ is infinite). Let $\mathsf{first}(\tau) = \tau[0]$ be the first state of τ, and let $\mathsf{final}(\tau)$ be the final state of τ when τ is finite. Let $\mathsf{traces}(s)$ be the set of all traces starting with s, also called *s-traces*, and let $\mathsf{traces}(S)$ be $\bigcup_{s \in S} \mathsf{traces}(s)$. A *complete trace* is either an infinite trace, or a finite trace τ where $\mathit{next}(\mathsf{final}(\tau)) = \emptyset$.

Bisimulations One of our major contributions is a new notion of bisimulation, suitable for formalizing the equivalence of programs in different languages. Below we summarize existing variants of bisimulation (from [22]) that we attempted, but failed, to use for our task. Triple $(S, L, \to)$ is a *labeled transition system* (LTS) when L is a set of labels and $\to \subseteq S \times L \times S$ is a *labeled transition relation*; we write $p \to^{\mu} q$ when $(p, \mu, q) \in \to$. Assume two LTS's with the same labels L, and R a binary relation on their respective states. We let p, p_1, p_2, p' range over the states of the first LTS and q, q_1, q_2, q' over the states of the second.

Definition 3.1 (Strong Bisimilarity)**.** Relation R is a *strong simulation* if, whenever $p\ R\ q$, for each $p \to^{\mu} p'$, there exists q' such that $q \to^{\mu} q'$ and $p'\ R\ q'$. Relation R is a *strong bisimulation* if both R and R^{-1} are strong simulations. *Strong bisimilarity* is the union of all strong bisimulations.

Strong bisimulation is too strong for cross-language program equivalence, because different programming languages typically have different computation granularity. Weaker notions of bisimulation are required when non-observable or internal transitions need to be considered. Let ϵ be the label for the internal transitions (the label for internal transitions is typically τ in the literature, but we use τ for traces in this paper).

- Let $\Rightarrow$ be the reflexive and transitive closure of $\to^{\epsilon}$.
- Let $\Rightarrow^{\mu}$ be the composition of $\Rightarrow$, $\to^{\mu}$, and $\Rightarrow$.
- Let $\Rightarrow^{\widehat{\mu}}$ be $\Rightarrow^{\mu}$ if $\mu \neq \epsilon$, and $\Rightarrow$ otherwise.

Definition 3.2 (Weak Bisimilarity)**.** Relation R is a *weak simulation* if, whenever $p\ R\ q$, for each $p \to^{\mu} p'$, there exists q' such that $q \Rightarrow^{\widehat{\mu}} q'$ and $p'\ R\ q'$. A relation R is a *weak bisimulation* if both R and R^{-1} are weak simulations. *Weak bisimilarity* is the union of all weak bisimulations.

Weak bisimulation is therefore not concerned with associating behaviors to ϵ-transitions. As detailed in Section 3.3, it is not trivial to differentiate between observable and internal

transitions when different languages are concerned, and internal transitions can carry computational contents that cannot be ignored. Additionally, ignoring ϵ-transitions may lead to failure in distinguishing branching structures, which led to the development of more variants of bisimulation [22].

Definition 3.3 (Branching, η-, and Delay Bisimilarities)**.** Relation R is a *branching simulation* if, whenever $p\ R\ q$, for each $p{\rightarrow^{\mu}}p'$: either $\mu = \epsilon$ and $p'\ R\ q$; or there exists q_1, q_2, q' such that $q{\Rightarrow}q_1{\rightarrow^{\mu}}q_2{\Rightarrow}q'$ and (1) $p\ R\ q_1$, (2) $p'\ R\ q_2$, and (3) $p'\ R\ q'$. Furthermore, *η-simulation* (and *delay simulation*, resp.) is defined the same way as above, except the requirement (2) (and (3), resp.). A relation R is a *branching*, (η-, and *delay*, resp.) *bisimulation* if both R and R^{-1} are branching, (η-, and delay, resp.) simulations. *Branching*, (η-, and *delay*, resp.) *bisimilarity* is the union of all branching, (η-, and delay, resp.) bisimulations.

In addition to the already mentioned difficulty to differentiate observable from internal transitions, when programs in different languages are concerned, it is counterintuitive to ensure condition (1) in the variants of branching bisimulation above. For example, suppose that $p{\rightarrow^{\mu}}p'$ corresponds to an (observable) 64-bit memory store instruction in one language, which simulates two 32-bit store instructions in the other language. In addition to not being clear which of the two 32-bit store operations should be observable and which should be internal, the condition (1) above requires the inconsistent state in-between the two 32-bit operations to be equivalent to a consistent state of the first program.

3.2 TOWARDS FORMALIZING PROGRAM EQUIVALENCE

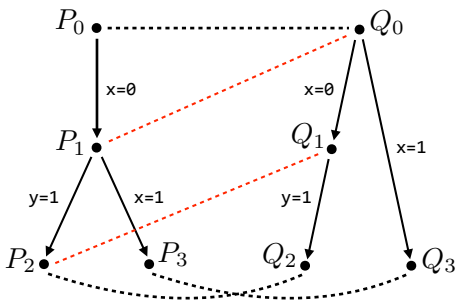
Translation validation ensures the correctness of a compiler by proving equivalence of each instance of compilation (i.e., a pair of source and target programs), instead of verifying the compiler itself. Intuitively, two (possibly non-terminating) programs are "equivalent" when given the same input they reach the same relevant states in the same order. Similarly, a (low-level) program L "refines" another (high-level) program H when program L contains no new behaviors not found in program H.

A simulation-based technique for proving equivalence of transition systems is a well-studied approach that admits a coinductive proof that deals with recursion, non-termination, and nondeterminism of the systems in a uniform and elegant way. In such a technique, the notion of equivalence is formulated as a bisimulation relation between the states of two transition systems,[1] and proving equivalence is reduced to finding a bisimulation relation.

[1]While most of bisimulation variants are originally defined over Kripke structures or labeled transition

```
P:  x = 0; if (*) { y = 1; } else { x = 1; }
Q:  if (*) { x = 0; y = 1; } else { x = 1; }
```

(a)



(b)

Figure 3.1: Program transformation example (as part of partial redundancy elimination), a stuttering bisimulation relation (both black and red dotted lines), and a cut-bisimulation relation (only black dotted lines). The `if(*)` statement denotes a non-deterministic branching operation.

Classic bisimulation variants, however, are often too strong for proving equivalence of programs, especially for the purpose of translation validation of program transformation. Specifically, strong bisimulation does not admit equivalence of programs generated by reordering transformations. On the other hand, stuttering bisimulation can deal with reordering transformations to certain extent (e.g., [23]), but it does not admit program transformations that modify branching structures. For example, consider the simple program transformation example shown in Figure 3.1(a), commonly performed by compilers as part of partial redundancy elimination. The seemingly equivalent two programs are still not strongly bisimilar, mainly because the intermediate states (P_1 and Q_1) are not "similar". Weaker variants, such as stuttering or branching bisimulation [22], could be used to prove their equivalence, since they are flexible to admit the irrelevant intermediate states. Figure 3.1(b) depicts a stuttering bisimulation relation shown as both black and red dotted lines, where the transitions $P_0 \to P_1$ and $Q_1 \to Q_2$ are considered "stuttering" transitions.[2] Note that, however, identifying the stuttering transitions are non-trivial. Indeed, the irrelevant intermediate states P_1 and Q_1 have the potential to stutter with all adjacent states. As such, there exist many candidate stuttering transitions (which are $P_0 \to P_1$, $P_1 \to P_2$, $P_1 \to P_3$, $Q_0 \to Q_1$, and $Q_1 \to Q_2$ in this example) and identifying the appropriate ones among many candidates

systems, we can adapt their definition to transition systems by considering a transition system as a Kripke structure with a single dummy atomic proposition, or a labeled transition system with a single dummy label.

[2]More precisely, it is a stuttering bisimulation over the Kripke structure where the labeling function L satisfies $L(P_0) = L(P_1)$ and $L(Q_1) = L(Q_2)$.

is not straightforward. The problem of identifying stuttering transitions becomes apparent when we consider the witness-based translation validation approach [23], in which the candidate relation is generated by the compiler as a "witness" for the correctness of the transformation, and proving the equivalence is reduced to checking that the generated relation is a (bi)simulation. However, it is not easy for the compiler to identify stuttering transitions which are not directly related to the internal information used in the compiler transformation. Thus, the stuttering transitions should be inferred separately, which incurs additional overhead in proving equivalence.[3]

Symbolic summaries have also been used as another notion of equivalence that is more relaxed than classic bisimulation variants. A symbolic summary of a program is a closed formula that describes the effect of the program on its input state. For example, the two programs in Figure 3.1(b) can be shown equivalent by showing that they have the same symbolic summary: $f(state) = if\ (*)\ then\ state[x : 0, y : 1]\ else\ state[x : 1]$. Symbolic summaries, however, are hard to compute for any program since they require symbolic execution of the program that can lead to exponential explosion (in programs with loops that have symbolic bounds) or even be intractable (in non-terminating programs).

On the other hand, a natural way of reasoning about equivalence of the two loops in Figure 3.1(a) is to employ symbolic execution techniques to summarize the effect of the two loop bodies and then use bisimulation treating the loop bodies as single nodes. In other words, one can prove equivalence of the two loops in two separate steps, where the first step is to prove equivalence of their loop bodies using symbolic summaries, and the second step is to prove equivalence of the loops using a bisimulation-based technique, while assuming that their loop bodies are equivalent. This blending of symbolic summaries and bisimulation proof techniques is practical for real-world program transformations, where symbolic summaries are used for local reasoning and bisimulation is used for global reasoning.

In this work, we present a bisimulation variant, named *cut-bisimulation*, that captures this combination of symbolic summaries and traditional bisimulation. Essentially, cut-bisimulation is a (strong) bisimulation over the "cut" states, a subset of the program states that satisfies certain well-defined conditions (see Section 3.3). The intuition for the cut of a program is that the states in the cut suffice as observation points of the program behavior, that is, nothing relevant can happen which is not witnessed by a cut state. A nice property of the cut is that there exists no infinite path between the cut states, thus symbolic execution between them is tractable. Intuitively, a cut-bisimulation proof amounts to proving same symbolic summaries for control flow between the cut states, and proving a strong bisimula-

[3]The time complexity of the best known algorithm for inferring stuttering bisimulation is $O(m \log n)$ where m is the number of transitions and n is the number of states [67].

tion over the cut states. Moreover, cut-bisimulation allows one to fine-tune the combination of the two proof techniques by adjusting the cut. For example, cut-bisimulation can be tuned to be plain symbolic summary based equivalence by having the cut set to include only the initial and final states, while it can become strong bisimulation by having the cut to include all the states (see Section 3.3). We will also show that cut-bisimulation can be tuned to be a stuttering bisimulation by properly setting the cut set (see Section 3.4). We believe that this generality of cut-bisimulation will allow us to reconcile various notions of program equivalence and their algorithms and techniques in an uniform and general framework.

Based on cut-bisimulation, we have implemented a *language-independent* program equivalence checker, KEQ, on top of the $\mathbb{K}$ framework [24]. Parameterized by formal semantics of source and target languages, KEQ takes as input two programs and a candidate relation between the two, and checks if the candidate relation is indeed a cut-bisimulation (see Section 3.5). As a preliminary evaluation, toward the use of our tool in translation validation of the LLVM compiler, we instantiated our tool with an LLVM semantics, converted the two programs in Figure 3.1 into LLVM bitcode, and proved equivalence of the two LLVM programs using the tool (see Section 3.6).

3.3 CUT-BISIMULATION

As mentioned earlier, cut-bisimulation is essentially a bisimulation over the cut states, where the cut represents a set of relevant states at which two programs can be synchronized (e.g., ones at the beginning of functions, or loop headers, etc.). This enables an intuitive procedure of proving equivalence, where one can check if the two programs indeed synchronize at the cut states (hence we also refer to them as synchronization points throughout this paper). Also, the cut can be adjusted to control the granularity of synchronization points, to represent the observable behaviors of two programs desired to consider when reasoning about their equivalence. This makes it easier to deal with intermediate states that are not relevant in identifying equivalence of programs.

In order for cut bisimulations to correctly capture program equivalence, however, two conditions must be satisfied. First, there must be enough cut states in the two transition systems so that no relevant behavior of one program can pass unsynchronized with a behavior of the other program. This implies, in particular, that each final state must be in the cut. It also implies that each infinite execution must contain infinitely many cut states, because otherwise one of the programs may not terminate while the other terminates.

Second, any two states related by a cut bisimulation must be compatible. Otherwise, one

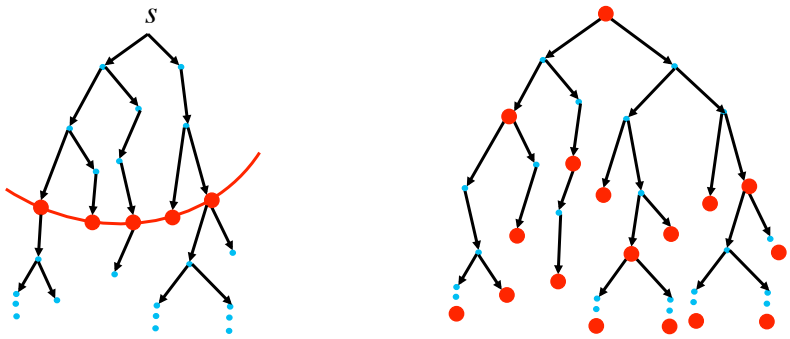


Figure 3.2: Left: a cut C for state s (each complete s-trace intersects C). Right: a cut C for a transition system (C contains the initial state and is a cut for itself, i.e., for each state in C)

can establish a cut bisimulation even for non-equivalent programs.[4] One straightforward such compatibility relation relates two states when their corresponding variables have the "same" value. However, what it precisely means for two values, especially in programs written in different languages, to be the same is not trivial, due to different representations (e.g., big-endian vs little-endian, or 32-bits vs 64-bits), different memory layouts (physically same location may point to different values, or contain garbage that has not been collected yet), etc. Also, state compatibility may require to check if specific memory locations (in the context of embedded systems), environment variables, input/output buffers, files, etc., are also "the same". Moreover, states corresponding to undefined behaviors (e.g., division by zero) may or may not be desired to be compatible, depending upon what kind of equivalence is desired. We found it awkward to encode such complex state compatibility abstractions as labels on transitions, as the existing notions of bisimulation require. Instead, we design the notion of cut-bisimulation to be parameterized by a binary relation on states, which we call an *acceptability* (or *compatibility* or *indistinguishability*) relation.

Now let us formalize our notion of cut-bisimulation.

Definition 3.4 (Cut and Cut Transition System)**.** Let $T = (S, \xi, \rightarrow)$ be a transition system. A set $C \subseteq S$ is a *cut for* $s \in S$, iff for any complete trace $\tau \in \mathsf{traces}(s)$, there exists some strictly positive $k > 0$ such that $\tau[k] \in C$. The set $C \subseteq S$ is a *cut for* T iff $\xi \in C$ and C is a cut for each $s \in C$, in that case T is called a *cut transition system* and is written as a quadruple $(S, \xi, \rightarrow, C)$. See Figure 3.2.

In a cut transition system, any finite complete trace starting with the initial state termi-

[4]For any two terminating programs, for example, there always exists a trivial cut bisimulation where all initial (and final) states are related to each other, respectively, if the compatibility of the states is not considered.

nates in a cut state, and any infinite trace starting with the initial state goes through cut states infinitely often:

Lemma 3.1. Let $T = (S, \xi, \rightarrow, C)$ be a cut transition system. Then for each complete trace $\tau \in \mathsf{traces}(\xi)$ and each $0 < i < \mathsf{size}(\tau)$, there is some $j \geq i$ such that $\tau[j] \in C$.

Proof. Let $\tau \in \mathsf{traces}(\xi)$ be a complete trace. Assume to the contrary that there exists i such that $\forall j \geq i.\ \tau[j] \notin C$. Pick such an i. Then we have two cases. When $\forall k < i.\ \tau[k] \notin C$, we have $\forall k > 0.\ \tau[k] \notin C$, which is a contradiction since C is a cut for $\xi = \tau[0]$. Otherwise, $\exists k < i.\ \tau[k] \in C$, and let k be the largest such number. Then, we have $\forall l > k.\ \tau[l] \notin C$, which is a contradiction since C is a cut for each $s \in C$, thus a cut for $\tau[k] \in C$. QED.

This result is reminiscent of the notion of Büchi acceptance [68]; specifically, if S is finite and $\mathit{next}(s) \neq \emptyset$ for all $s \in S$, then it says that the transition system T regarded as a Büchi automaton with C as final states, accepts all the infinite traces. This analogy was not intended and so far played no role in our technical developments.

Cuts do not need to be minimal in practice, and are not difficult to produce. For example, a typical cut includes all the final states (normally terminating states, error/exception states, etc.) and all the states corresponding to entry points of cyclic constructs in the language (loops, recursive functions, etc.). Such cut states can be easily identified statically using control-flow analysis, or dynamically using a language operational semantics.

Definition 3.5 (Cut-Successor)**.** Let $T = (S, \xi, \rightarrow, C)$ be a cut transition system. A state s' is an (immediate) *cut-successor* of s, written $s \rightsquigarrow s'$, iff there exists a finite trace $ss_1 \cdots s_n s'$ where $s' \in C$ and $n \geq 0$ and $s_i \notin C$ for all $1 \leq i \leq n$.

Definition 3.6 (Cut-Bisimilarity)**.** Let $T_i = (S_i, \xi_i, \rightarrow_i, C_i)$ be two cut transition systems ($i \in \{1, 2\}$). Relation $R \subseteq C_1 \times C_2$ is a *cut-simulation* iff whenever $(s_1, s_2) \in R$, for all s_1' with $s_1 \rightsquigarrow_1 s_1'$ there is some s_2' such that $s_2 \rightsquigarrow_2 s_2'$ and $(s_1', s_2') \in R$. Let $\leq$ be the union of all cut-simulations (also a cut-simulation). Relation R is a *cut-bisimulation* iff both R and R^{-1} are cut-simulations. Let $\sim$ be the union of all cut-bisimulations (also a cut-bisimulation).

Cut-bisimulation generalizes standard (strong) bisimulation [22]. A cut bisimulation on $(S_i, \xi_i, \rightarrow_i, C_i)$ is a bisimulation on $(S_i, \xi_i, \rightarrow_i)$, when $C_i = S_i$. Also, cut-bisimulation becomes bisimulation if we cut-abstract the transition systems:

Definition 3.7 (Cut-Abstract Transition System)**.** Let T be a cut transition system $(S, \xi, \rightarrow, C)$. The *cut-abstract transition system of* T, written $\overline{T}$, is the (standard) transition system $(C, \xi, \rightsquigarrow)$.

Lemma 3.2. Let $T_i = (S_i, \xi_i, \rightarrow_i, C_i)$ be two cut transition systems ($i \in \{1, 2\}$). A relation $R \subseteq C_1 \times C_2$ is a cut-bisimulation on T_1 and T_2, iff R is a (standard) bisimulation on $\overline{T_1}$ and $\overline{T_2}$.

Proof. By identifying $\rightsquigarrow_i$ as the transition relation of $\overline{T_i}$. QED.

Corollary 3.1. Let $T_i = (S_i, \xi_i, \rightarrow_i, C_i)$ be two cut transition systems ($i \in \{1, 2\}$). Let R be a cut-bisimulation, and $(s_1, s_2) \in R$. For any state $s_1' \in C_1$ with $s_1 \rightarrow_1^+ s_1'$, there exists some $s_2' \in C_2$ with $s_2 \rightarrow_2^+ s_2'$ such that $(s_1', s_2') \in R$. The converse also holds.

Proof. By Lemma 3.2 and the bisimulation invariant of reachability. QED.

Now we formalize the equivalence of cut transition systems in the presence of a given acceptability (or compatibility, or indistinguishability) relation $\mathcal{A}$ on states.

Definition 3.8. Let $\mathcal{A} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$, which we call an *acceptability relation.* Let $T_i = (S_i, \xi_i, \rightarrow_i, C_i)$ be two cut transition systems ($i \in \{1, 2\}$). T_2 *cut-simulates* T_1 *(i.e.,* T_1 *cut-refines* T_2*) w.r.t.* $\mathcal{A}$, written $T_1 \leq_{\mathcal{A}} T_2$, iff there exists a cut-simulation $\mathsf{P} \subseteq \mathcal{A}$ such that $\xi_1 \ \mathsf{P} \ \xi_2$. Furthermore, T_1 *and* T_2 *are cut-bisimilar w.r.t.* $\mathcal{A}$, written $T_1 \sim_{\mathcal{A}} T_2$, iff there exists a cut-bisimulation $\mathsf{P} \subseteq \mathcal{A}$ such that $\xi_1 \ \mathsf{P} \ \xi_2$.

Note that if a cut bisimulation P like above exists, then there also exists a largest one; that's because the union of cut bisimulations included in $\mathcal{A}$ is also a cut bisimulation included in $\mathcal{A}$. We let the relation $\sim_{\mathcal{A}}$ denote that largest cut bisimulation, assuming that it exists whenever we use the notation (and similarly for $\leq_{\mathcal{A}}$).

Our thesis is that $\sim_{\mathcal{A}}$ yields the right notion of program equivalence. That is, that two programs are equivalent according to a given state acceptability (or compatibility or indistinguishability) relation $\mathcal{A}$ between the states of the respective programming languages, iff for any input, the cut transition systems T_1 and T_2 corresponding to the two program executions satisfy $T_1 \sim_{\mathcal{A}} T_2$. The following result strengthens our thesis, stating that cut-bisimilar transition systems reach compatible states at cut points, and, furthermore, that they cannot indefinitely avoid the cut points:

Theorem 3.1. If $T_1 \sim_{\mathcal{A}} T_2$ then for each s_1 with $\xi_1 \rightarrow_1^+ s_1$ there exists some s_2 with $\xi_2 \rightarrow_2^+ s_2$, such that: (1) if $s_1 \in C_1$ then $s_1 \sim_{\mathcal{A}} s_2$; and (2) if $s_1 \notin C_1$ then there exists some $s_1' \in C_1$ such that $s_1 \rightarrow^+ s_1'$ and $s_1' \sim_{\mathcal{A}} s_2$. The converse also holds.

Proof. We only need to show the forward direction, since the backward is dual. First we have $\xi_1 \sim \xi_2$ by Definition 3.8 and the fact that $\sim$ is the union of all cut-bisimulations. Let s_1 be a state with $\xi_1 \rightarrow_1^+ s_1$. Then we have two cases:

- When $s_1 \in C_1$. There exists s_2 such that $\xi_2 \to_2^+ s_2$ and $s_1 \sim s_2$ by Corollary 3.1.
- When $s_1 \notin C_1$. There exists s_1' such that $s_1 \to_1^+ s_1'$ and $s_1' \in C_1$ by Lemma 3.1 and the fact that C_1 is a cut for $\xi_1 \in C_1$. Then, there exists s_2 such that $\xi_2 \to_2^+ s_2$ and $s_1' \sim s_2$ by Corollary 3.1.

QED.

3.3.1 Property Preservation

Consider two cut transition systems where one cut-simulates another, but not the other way around. For example, an abstract model cut-simulates its concrete implementation, if implemented correctly, but the inverse may not hold since the model may omit to specify some details, leaving as implementation-dependent, for which the implementation can freely choose any behavior. In this case, it is not trivial to see whether a property of the model is also held in the implementation. Intuitively, the set of all reachable cut-states of the model is a super set of that of the implementation. Thus, if a cut-state is not reachable in the model, then it is also not reachable in the implementation. This implies that safety properties of the model are preserved in the implementation, since a safety property can be represented as "nothing bad happens", i.e., in other words, "a bad state is not reachable". In general, inductive invariants are preserved in the refined system.

Now we formulate the property preservation of cut-simulation. Let $T = (S, \xi, \to, C)$ be a cut transition system. Let P be a predicate over a domain D, and $f : S \to D$ be a state normalization function. Let P_f be a predicate over S, defined by $P_f(s) \stackrel{\texttt{def}}{=} P(f(s))$ for some $s \in S$. The predicate P_f is a cut-inductive invariant of T, if $P_f(\xi)$, and $P_f(s) \wedge s \rightsquigarrow s' \implies P_f(s')$ for any states $s, s' \in S$. A cut-inductive invariant, thus, holds for all reachable cut-states. Also, let $T_i = (S_i, \xi_i, \to_i, C_i)$ be two cut transition systems (for $i \in \{1, 2\}$). Suppose $T_1 \geq T_2$, that is, T_1 cut-simulates T_2, (in other words, T_2 cut-refines T_1). We say $\geq$ is right-total if for all $s_2 \rightsquigarrow_2 s_2'$, there exists $s_1 \rightsquigarrow_1 s_1'$ such that $s_1 \geq s_2$ and $s_1' \geq s_2'$.

Theorem 3.2. Suppose $T_1 \geq T_2$. Suppose $\geq$ is right-total, and $\xi_1 \geq \xi_2$. Suppose P_{f_1} is a cut-inductive invariant of T_1, and $f_1(s_1) = f_2(s_2)$ if $s_1 \geq s_2$. Then, P_{f_2} is a cut-inductive invariant of T_2.

Proof. $P_{f_2}(\xi_2)$ since $\xi_1 \geq \xi_2$ and $P_{f_1}(\xi_1)$. Suppose $P_{f_2}(s_2)$ and $s_2 \rightsquigarrow_2 s_2'$. Since $T_1 \geq T_2$ and $\geq$ is right-total, there exists $s_1 \rightsquigarrow_1 s_1'$ such that $s_1 \geq s_2$ and $s_1' \geq s_2'$. Then, $P_{f_1}(s_1)$ since $f_1(s_1) = f_2(s_2)$. Since P_{f_1} is inductive, $P_{f_1}(s_1')$. Thus, $P_{f_2}(s_2')$ since $f_1(s_1') = f_2(s_2')$. QED.

3.4 COMPARISON TO STUTTERING BISIMULATION

In this section, we identify a subclass of stuttering bisimulation that can be reduced to cut-bisimulation.

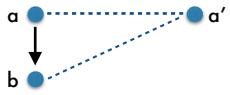


Figure 3.3: Systems that are stuttering-bisimilar, but not cut-bisimilar.

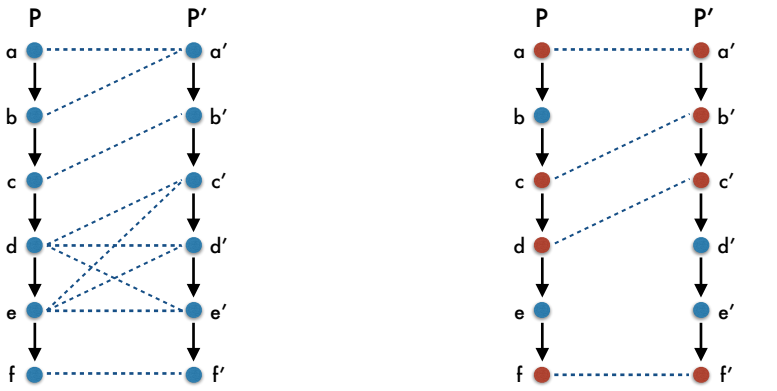
Although cut-bisimulation is analogous to stuttering bisimulation in the sense that both deal with intermediate states that are not relevant in reasoning about equivalence of systems, they are not equivalent. As explained in Section 3.2, there exist systems that are cut-bisimilar but not stuttering-bisimilar, as shown in Figure 3.1. Also, there exist (trivial) systems that are stuttering-bisimilar but not cut-bisimilar, as shown in Figure 3.3. Here, the relation $\{(a, a'), (b, a')\}$ is not a cut-bisimulation, because while a and a' are related and there exists a successor from a, there does not exist a successor from a'.

On the other hand, there still exist many non-trivial systems that are both cut-bisimilar and stuttering-bisimilar, for which a stuttering-bisimulation can be converted to a cut-bisimulation. Let us illustrate the intuition of the reduction. Consider Figure 3.4 that shows a stuttering bisimulation on two deterministic systems P and P', and its reduction to a cut-bisimulation. The stuttering bisimulation R shown in Figure 3.4(a) relates a path fragment with another fragment, where the fragments identify "stuttering" nodes, i.e., nodes b and e in P, and nodes d' and e' in P'. The cut-bisimulation R_c shown in Figure 3.4(b) can be constructed by restricting R on the non-stuttering nodes. In other words, R induces a partition of states, i.e., $\{\{a, b\}, \{c\}, \{d, e\}, \{f\}\}$ for P, and $\{\{a'\}, \{b'\}, \{c', d', e'\}, \{f'\}\}$ for P', from which R_c can be constructed by restricting R on the representatives of the partitions, i.e., $R_c = R \cap (\{a, c, d, f\} \times \{a', b', c', f'\})$. We will explain later how to identify such representatives.

However, not all stuttering bisimulations induce such a state partition, especially when systems involve loops and a stuttering bisimulation takes different stuttering nodes for different loop iterations. In that case, it is not straightforward to reduce a stuttering bisimulation to a cut-bisimulation, if any. In the remaining of this section, we will identify a subclass of stuttering bisimulation, named *stationary stuttering bisimulation*, that can be reduced to cut-bisimulation.

Notations Let AP be a set of atomic propositions, and $(S, \xi, \rightarrow, L)$ be a Kripke structure over AP, i.e., a transition system augmented with a labeling function $L : S \rightarrow 2^{AP}$. The underlying graph structure G_T of the Kripke structure $T = (S, \xi, \rightarrow, L)$ is a directed graph with vertices S and edges $\rightarrow$.

A *partition* ϕ of a finite trace τ is a finite sequence of indexes $p_0, p_1, \cdots, p_n$, where $0 = p_0 <$



(a) Stuttering bisimulation R (b) Cut-bisimulation R_c

Figure 3.4: Conversion of stuttering bisimulation to cut-bisimulation

$p_1 < \cdots < p_n < \mathsf{size}(\tau)$, which denotes a sequence of n sub-traces of τ, $(\tau[p_0], \cdots, \tau[p_1 - 1]), \cdots, (\tau[p_n], \cdots, \tau[p_{n+1} - 1])$, where $p_{n+1} = \mathsf{size}(\tau)$. Let $\phi[k]$ be the k^{th} sub-trace of τ, and $[\![\phi[k]]\!]$ be the set of states in $\phi[k]$, $\{\tau[p_k], \cdots, \tau[p_{k+1} - 1]\}$, for any $0 \leq k \leq n$. A partition of an infinite trace is a infinite sequence of indexes, which is similarly defined. Let $\mathsf{size}(\phi)$ be the length of ϕ (∞ when ϕ is infinite), and $\mathsf{final}(\phi) = \phi[\mathsf{size}(\phi) - 1]$ be the final sub-trace when ϕ is finite. Let $\mathsf{partitions}(\tau)$ be the set of all partitions of τ, and $\mathsf{partitions}(\mathcal{T})$ be $\bigcup_{\tau \in \mathcal{T}} \mathsf{partitions}(\tau)$.

First, let us recall the definition of stuttering bisimulation, proposed in [69].

Definition 3.9 (Stuttering Bisimulation [69])**.** Let $T_i = (S_i, \xi_i, \rightarrow_i, L_i)$ be two Kripke structures ($i \in \{1, 2\}$). Relation $R \subseteq S_1 \times S_2$ is a *stuttering simulation* iff whenever $(s_1, s_2) \in R$, $L_1(s_1) = L_2(s_2)$, and for every trace $\tau_1 \in \mathsf{traces}(s_1)$ there exist a trace $\tau_2 \in \mathsf{traces}(s_2)$, and partitions $\phi_i \in \mathsf{partitions}(\tau_i)$, such that:

$$\mathsf{size}(\phi_1) = \mathsf{size}(\phi_2), \text{ and for any } 0 \leq k < \mathsf{size}(\phi_1),\ [\![\phi_1[k]]\!] \times [\![\phi_2[k]]\!] \subseteq R \tag{3.1}$$

Relation R is a *stuttering bisimulation* iff both R and R^{-1} are stuttering simulations.

Now, let us define a trace partitioning function that induces a state partition, which will be used to identify a subclass of stuttering bisimulation later.

Definition 3.10. Let $T = (S, \xi, \rightarrow, L)$ be a Kripke structure, and τ be a trace over S. Let $M : S \rightarrow \mathbb{N}$ be a finitely partitioning function over S, where $\{s \in S \mid M(s) = k\}$ is finite for all $k \in \mathbb{N}$. An *M-induced partition of τ*, $\phi_M(\tau)$, if it exists, is a *unique* partition of τ, i.e., a sequence of indexes $p_0 = 0, p_1, p_2, \ldots$, which denotes a sequence of sub-traces of τ,

$(\tau[p_0], \cdots, \tau[p_1 - 1]), (\tau[p_1], \cdots, \tau[p_2 - 1]), \cdots$, such that for any $k \geq 0$, $p_k < p_{k+1}$:

$$M(\tau[i]) = M(\tau[j]) \quad \text{for} \quad p_k \leq i, j < p_{k+1}, \tag{3.2}$$

$$\tau[p_k] \neq \tau[i] \quad \text{for} \quad p_k < i < p_{k+1}, \text{ and} \tag{3.3}$$

$$M(\tau[p_{k+1} - 1]) \neq M(\tau[p_{k+1}]) \quad \text{or} \quad \tau[p_k] = \tau[p_{k+1}]. \tag{3.4}$$

Moreover, let G_T be the underlying graph of T. Then, an *M-induced partition of G_T* is a set of components where each component is a weakly-connected component of a subgraph of G_T induced by a set of vertices $\{s \in S \mid M(s) = k\}$ for some $k \in \mathbb{N}$. Each component in the M-induced partition of G_T is called an *M-induced component of G_T*. An *entry vertex* of an M-induced component of G_T is the vertex of the initial state ξ, or a vertex that has incoming edges from outside the component. We say that M is *natural* if:

(i) every M-induced component of G_T has only a single entry vertex, and

(ii) for any finite complete trace τ over S, the final sub-trace of an M-induced partition of τ, if it exists, is a singleton, i.e., $\mathsf{size}(\mathsf{final}(\phi_M(\tau))) = 1$.

Recall that a vertex-induced subgraph may contain multiple weakly-connected components. Also, note that the condition (i) can be satisfied for any natural loop. The condition (ii) can be also easily satisfied by augmenting the original Kripke structure with a dummy next state for each final state.

Now, we identify a subclass of stuttering bisimulation that can be reduced to cut-bisimulation.

Definition 3.11 (Stationary Stuttering Bisimulation)**.** Let $T_i = (S_i, \xi_i, \rightarrow_i, L_i)$ be two Kripke structures ($i \in \{1, 2\}$), and $R \subseteq S_1 \times S_2$ be a stuttering bisimulation. We say that R is *stationary*, if there exist natural, finitely partitioning functions M_i such that whenever $(s_1, s_2) \in R$, for every trace $\tau_1 \in \mathsf{traces}(s_1)$, there exists a trace $\tau_2 \in \mathsf{traces}(s_2)$ such that the M_i-induced partitions $\phi_{M_i}(\tau_i)$ exist and satisfy the equation (3.1), and the converse also holds.

Theorem 3.3. Suppose R is a stationary stuttering bisimulation on two Kripke structures $T_i = (S_i, \xi_i, \rightarrow_i, L_i)$ ($i \in \{1, 2\}$), and R is total, i.e., $\Pi_i(R) = S_i$. Then there exist cuts $C_i \subseteq S_i$ and $R_c \subseteq R$ such that R_c is a cut-bisimulation on two cut-transition systems $T_i' = (S_i, \xi_i, \rightarrow_i, C_i)$.

Proof. Let M_i be the natural, finitely partitioning function for R, and G_{T_i} be the underlying graph of T_i. Let C_i be the union of the entry vertex state of all M_i-induced components of

G_{T_i}. Then, C_i is a cut for T_i by Lemma 3.5 and the fact that an M_i-induced partition exists for any trace over S_i, since R is a total, stationary stuttering bisimulation.

Now, let us prove that $R_c = R \cap (C_1 \times C_2)$ is a cut-bisimulation on T'_i. Let $(s_1, s_2) \in R_c \subseteq R$ be two states related by R_c, and let $\tau_1 \in \mathsf{traces}(s_1)$ be a complete trace starting with s_1. If $\mathsf{size}(\tau_1) = 1$, that is, s_1 is a final state, then it immediately satisfies the condition of cut-bisimulation. Now, suppose that $\mathsf{size}(\tau_1) > 1$. Then, by Lemma 3.3 and Lemma 3.6, there exists a non-singleton partition $\phi_M(\tau_1) = p_0, p_1, \ldots$, and an entry vertex state s'_1 (thus $s'_1 \in C_1$), such that $s_1 \rightsquigarrow s'_1 = \tau_1[p_1]$. Then, by Definition 3.11, there exists another trace $\tau_2 \in \mathsf{traces}(s_2)$ and another non-singleton partition $\phi_M(\tau_2) = q_0, q_1, \ldots$, such that $(s'_1, s'_2) \in R$ where $s'_2 = \tau_2[q_1]$. Then, by Lemma 3.6, s'_2 is an entry vertex state (thus $s'_2 \in C_2$), and $\tau_2[k]$ is not an entry vertex state (thus $\tau_2[k] \notin C_2$) for any $0 < k < q_1$. Thus, $s_2 \rightsquigarrow s'_2$ and $(s'_1, s'_2) \in R_c$, which concludes that R_c is a cut-simulation. Similarly, R_c^{-1} is a cut-simulation, and thus R_c is a cut-bisimulation.

QED.

The lemmas used in the above proof are presented in the following Subsection 3.4.1.

3.4.1 Lemmas for Theorem 3.3

Lemma 3.3. Let $T = (S, \xi, \rightarrow, L)$ be a Kripke structure, G_T be the underlying graph of T, and M be a natural, finitely partitioning function over S. Suppose that there exists an M-induced partition of τ for any trace τ over S. Then we have the following:

1. An M-induced component of G_T is a singleton, if it contains a final state.
2. For any complete trace τ over S, if $\mathsf{size}(\tau) > 1$, then $\mathsf{size}(\phi_M(\tau)) > 1$.

Proof. Let us prove the claim 1. Let W be an M-induced component of G_T. Suppose W contains a final state s_1, but is not a singleton. Then, there exists a (complete) trace $\tau = s_0 s_1$, where s_0 is in W. Then, the M-induced partition of τ, $\phi_M(\tau)$, is a singleton sequence, thus $\mathsf{size}(\mathsf{final}(\phi_M(\tau))) = 2$, which is a contradiction because of the condition (ii) of Definition 3.10.

Let us prove the claim 2. If τ is infinite, we immediately have $\mathsf{size}(\phi_M(\tau)) > 1$. Suppose $\tau = s_0 s_1 \ldots s_n$ is finite, where $n \geq 1$. Assume that $\mathsf{size}(\phi_M(\tau)) = 1$. Then, by the equation (3.2), $M(s_0) = M(s_n)$. On the other hand, since τ is a finite complete trace, s_n is a final state, and $s_0 \neq s_n$. Thus, by the claim 1, s_n is in a singleton M-induced component, which means that s_0 is in another M-induced component, but it is a contradiction because of $M(s_0) = M(s_n)$. QED.

Lemma 3.4. Let $T = (S, \xi, \to, L)$ be a Kripke structure, G_T be the underlying graph of T, and M be a natural, finitely partitioning function over S. Suppose that there exists an M-induced partition of τ for any trace τ over S. Then, for any M-induced component of G_T, if there exists a cycle within the component, the cycle contains the entry vertex of the component, or is not reachable from the entry vertex.

Proof. Suppose that there exists a cycle within the component, and the cycle is reachable from the entry vertex of the component, but does not contain the entry vertex. Then, there exists an infinite trace τ within the component that starts from the entry vertex but never revisits the entry vertex, i.e., $\tau[0] \neq \tau[i]$ for any $i > 0$, and $M(\tau[i]) = M(\tau[j])$ for any $i, j \geq 0$. Thus, there exists no $p_1 > 0$ that satisfies the equation (3.4) for $k = 0$ in Definition 3.10, which is a contradiction because there exists an M-induced partition of τ. QED.

Lemma 3.5. Let $T = (S, \xi, \to, L)$ be a Kripke structure, G_T be the underlying graph of T, and M be a natural, finitely partitioning function over S. Suppose that there exists an M-induced partition of τ for any trace τ over S. Let C be the union of the entry vertex state of all M-induced components of G_T. Then, C is a cut for T.

Proof. First, it is obvious that ξ is in C. Let $s \in C$ be a state, and $\tau \in \mathsf{traces}(s)$ be a complete trace starting with s. Then we have three cases:

- When τ reaches a final state without entering to another M-induced component. In this case, τ is a singleton, i.e., s is a final state, by Lemma 3.3. Thus, we immediately have that C is a cut for s.

- When τ enters into another M-induced component, i.e., there exists $k > 0$ such that $M(s) = M(\tau[k-1]) \neq M(\tau[k])$. In this case, $\tau[k]$ is the entry vertex of the component because of the condition (i) in Definition 3.10. Thus, C is a cut for s, since the entry vertex is in C.

- When τ stays within the current M-induced component without reaching a final state, i.e., for any $k > 0$, $M(s) = M(\tau[k])$. In this case, there exists a cycle within the component that τ visits infinitely. Since the cycle contains s by Lemma 3.4, there exists $k > 0$ such that $\tau[k] = s \in C$, and thus C is a cut for s.

Thus, C is a cut for T by Definition 3.4. QED.

Lemma 3.6. Let $T = (S, \xi, \to, L)$ be a Kripke structure, G_T be the underlying graph of T, and M be a natural, finitely partitioning function over S. Let τ be a trace over S that starts

with an entry vertex of an M-induced component of G_T. Suppose that there exists an M-induced partition of τ, $\phi_M(\tau) = p_0, p_1, \ldots$, where $p_0 = 0$. Then, for any $1 \leq i < \mathsf{size}(\phi_M(\tau))$,

$$\tau[p_i] \text{ is an entry vertex, and} \tag{3.5}$$

$$\tau[k] \text{ is not an entry vertex for any } p_{i-1} < k < p_i. \tag{3.6}$$

Proof. Let us first prove the equation (3.5). Suppose that there exist integers i such that $1 \leq i < \mathsf{size}(\phi_M(\tau))$ and $\tau[p_i]$ is not an entry vertex. Let j be the smallest integer among them. Then, $\tau[p_{j-1}]$ is an entry vertex, since $\tau[p_0]$ is an entry vertex. Since $\tau[p_j]$ is not an entry vertex, $M(\tau[p_j - 1]) = M(\tau[p_j])$. Thus, by the equation (3.4), $\tau[p_{j-1}] = \tau[p_j]$, which is a contradiction because $\tau[p_{j-1}]$ is an entry vertex.

Now let us prove the equation (3.6). Suppose there exists $1 \leq i < \mathsf{size}(\phi_M(\tau))$ and $p_{i-1} < k < p_i$ such that $\tau[k]$ is an entry vertex. By the equation (3.5), $\tau[p_{i-1}]$ is an entry vertex. We have two cases:

- When $\tau[p_{i-1}]$ and $\tau[k]$ are in different M-induced components. Then, we have $M(\tau[p_{i-1}]) \neq M(\tau[k])$, which is a contradiction because of the equation (3.2).
- When $\tau[p_{i-1}]$ and $\tau[k]$ are in the same M-induced component. Then, by the condition (i) of Definition 3.10, $\tau[p_{i-1}] = \tau[k]$, which is a contradiction because of the equation (3.3).

QED.

3.5 KEQ, A LANGUAGE-PARAMETRIC PROGRAM EQUIVALENCE CHECKER

We implemented a language-independent equivalence checking tool on top of the $\mathbb{K}$ framework [24]. $\mathbb{K}$ provides a language for defining operational semantics of programming languages, and a series of generic tools that take a language semantics as input and specialize themselves for that language: concrete execution engine (interpreter), symbolic execution engine, (bounded) model checker, and a deductive program verifier. The main idea underlying $\mathbb{K}$ is that a given language operational semantics is turned into a transition system generator, one for each program, and a suite of existing components provide the capability to work with such transition systems generically, in a language-independent manner. We developed a new such tool, KEQ, which takes *two* language semantics as input and yields a checker that takes two programs as input, one in each language, and a (symbolic) synchronization relation, and checks whether the two programs are indeed equivalent with the synchronization relation as witness.

Note that checking program equivalence in Turing complete languages is equivalent to checking the totality of a Turing machine (whether it terminates on all inputs), which is a known Π_2^0-complete problem [70], so strictly harder than recursively or co-recursively enumerable. It is therefore impossible to find any algorithm that decides equivalence for two given programs. The best we can do is to find techniques and algorithms that work well enough in practice. Definition 3.8 suggests such a technique: find a (witness) relation P and show that it is a cut-bisimulation. While finding such a relation is hard in general, it is relatively easy to check if a given relation, for example one produced by an instrumented compiler, is a cut-bisimulation.

Theoretical Equivalence Checking Algorithm Based on Cut-Bisimulation Our KEQ implementation follows the model of the theoretical Algorithm 3.1. Function main essentially checks whether P is a cut-bisimulation: for each pair $(p_1, p_2) \in \mathsf{P}$, for each p_1' with $p_1 \rightsquigarrow_1 p_1'$, there exists p_2' with $p_2 \rightsquigarrow_2 p_2'$ such that $p_1' \ \mathsf{P} \ p_2'$; and the converse. It first computes the cut-successors of p_i (at line 7), and checks whether each pair of the successors is related in P (line 9). The pairs found to be related in P are marked in black (line 10), while the others remain in red. If all of the successors are in black, it returns true (line 12). Note that the algorithm can also be used for checking whether P is a cut-simulation, by simply considering only N_1 in the line 12, i.e., replacing the if-condition with $\forall n \in N_1.\ n.\mathsf{color} = \mathsf{black}$.

Due to its concrete (as opposed to symbolic) nature, Algorithm 3.1 may not terminate in practice, since P could be infinite. Line 2 assumes that P is recursively enumerable, so iterable. Furthermore, lines 19 and 8 terminate only if T_i is finitely branching. We will explain how to refine Algorithm 3.1 to be practical shortly; for now we can show that it is refutation-complete, in the sense that if it does not terminate then the two programs are equivalent.

Theorem 3.4 (Correctness of Algorithm 3.1)**.** Suppose that cut transition systems $T_i = (S_i, \xi_i, \rightarrow_i, C_i)$ are finitely branching ($i \in \{1, 2\}$) and $\mathsf{P} \subseteq \mathcal{A}$ is recursively enumerable with $(\xi_1, \xi_2) \in \mathsf{P}$. If Algorithm 3.1 does <u>not</u> terminate with false, then $T_1 \sim_{\mathcal{A}} T_2$.

Indeed, suppose that Algorithm 3.1 does not terminate with *false*. Then none of the `check(`p_1`,`p_2`)` calls (line 3) terminates with *false* (line 13). Since the loop at Line 8 always terminates (T_1 and T_2 are finitely branching), it means that all nodes are colored black at line 11. Therefore, for each $(p_1, p_2) \in \mathsf{P}$, each cut-successor of p_1 can be paired in P with a cut-successor of p_2, and vice versa. Then P is a cut-bisimulation (Definition 3.6), that is, $T_1 \sim_{\mathcal{A}} T_2$ (Definition 3.8).

Algorithm 3.1: Equivalence checking algorithm. For checking cut-simulation, replace $N_1 \cup N_2$ with N_1 at line 11. As given, the algorithm works with concrete data and thus is not practical. Replace boxed expressions with their grayed variants to the right for a practical, symbolic algorithm, as implemented in KEQ.

Data: $T_i = (S_i, \xi_i, \to_i, C_i)$; $\mathsf{P} \subseteq C_1 \times C_2$; `// P is r.e.`

1 **Function** `main()`:
2 **foreach** $\boxed{(p_1, p_2) \in \mathsf{P}}$ **do** // $\boxed{\overline{\mathsf{P}}}$
3 **if** `check`$(p_1, p_2) = \mathit{false}$ **then**
4 **return** *false*;
5 **return** *true*;

6 **Function** `check`(p_1, p_2):
7 $N_1 \leftarrow \mathtt{next}_1(p_1)$; $N_2 \leftarrow \mathtt{next}_2(p_2)$;
8 **foreach** $(n_1, n_2) \in N_1 \times N_2$ **do**
9 **if** $\boxed{(n_1, n_2) \in \mathsf{P}}$ **then** // $\boxed{[\![(n_1, n_2)]\!] \subseteq [\![\overline{\mathsf{P}}]\!]}$
10 $n_1.\mathsf{color} \leftarrow \mathsf{black}$; $n_2.\mathsf{color} \leftarrow \mathsf{black}$;
11 **if** $\forall n \in N_1 \cup N_2.\ n.\mathsf{color} = \mathsf{black}$ **then**
12 **return** *true*;
13 **return** *false*

14 `// Returns cut-successors of` n
15 **Function** $\mathtt{next}_i(n)$:
16 $N \leftarrow \{n\}$; $\mathit{Ret} \leftarrow \emptyset$;
17 **while** N *is not empty* **do**
18 choose n from N; $N \leftarrow N \setminus \{n\}$;
19 $N' \leftarrow \{n' \mid n \boxed{\to_i} n'\}$; // $\boxed{\overline{\to_i}}$
20 **foreach** $n' \in N'$ **do**
21 **if** $\boxed{n' \in C_i}$ **then** // $\boxed{[\![n']\!] \subseteq [\![\overline{C_i}]\!]}$
22 $n'.\mathsf{color} \leftarrow \mathsf{red}$;
23 $\mathit{Ret} \leftarrow \mathit{Ret} \cup \{n'\}$;
24 **else**
25 $N \leftarrow N \cup \{n'\}$;
26 **return** Ret;

Proof. By Definition 3.8, we only need to show that P is a cut-bisimulation when `main` does not terminate with returning false. First let us characterize the two sub-functions:

1. `check`(p_1, p_2) terminates; and if it returns true, then for all $p_1 \rightsquigarrow_1 s_1$, there exists s_2 such that $p_2 \rightsquigarrow_2 s_2$ and $(s_1, s_2) \in \mathsf{P}$; and the converse also holds.

2. `next`$_i(n)$ terminates and returns the set of all cut-successors of n, i.e., $\{n' \mid n \rightsquigarrow_i n'\}$.

P is cut-bisimulation by (1), while the claim (1) is straightforward by (2). Now we only need to show the claim (2). To show (2), let us claim the invariant of the while loop as follows. It is easy to show that it is maintained in each iteration.

- For each finite trace $n_1 \cdots n_k n'$ $(k \geq 1)$ such that $n_1 = n$, $n_j \notin C_i$ $(1 < j \leq k)$ and $n' \in C_i$, either holds: $n' \in \mathit{Ret}$ or $\exists m \in [1, k].\ n_m \in N$.

- For each $n' \in N \cup \mathit{Ret}$ such that $n' \neq n$, there exists a finite trace $n n_1 \cdots n_k n'$ $(k \geq 0)$ such that $n_j \notin C_i$ $(1 \leq j \leq k)$.

- $\mathit{Ret} \subseteq C_i$.

Now let us show (2). First, $\mathit{Ret} \subseteq \{n' \mid n \rightsquigarrow_i n'\}$ by the second and the third bullets. Then, $\mathit{Ret} \supseteq \{n' \mid n \rightsquigarrow_i n'\}$ by the first bullet, since N is empty when it terminates. Let us show the termination. Assume `next`$_i(n)$ does not terminate. If it does not terminate, then since T_i is finitely branching, there should exist an infinite trace from n that never comes across one of C_i, which is the contradiction since C_i is a cut for $n \in \mathsf{P}_i \subseteq C_i$. Now let us show that the loop invariant is maintained in each iteration. The second and the third bullets are trivial. Let us show the first bullet. Assume that it holds in some iteration. Pick such a finite trace $n_1 \cdots n_k n'$. We have three cases: $n' \in \mathit{Ret}$, $n_k \in N$, and $n_m \in N$ where $m < k$. For the first and the third cases, it is easy to show the invariant is maintained in the next iteration. In the second case, we have $n' \in \mathsf{next}(n_k)$ by definition of the traces. Then n' is added in Ret in the line 23, since $n' \in C_i$, and we conclude. QED.

Note that if we replace the if-condition of the line 11 with "$\forall n \in N_1.\ n.\mathsf{color} = \mathsf{black}$", then it suffices to show $T_1 \leq_{\mathcal{A}} T_2$, i.e., T_1 refines T_2.

Symbolic Implementation of Algorithm 3.1 Note that Algorithm 3.1 may also terminate with *true*, namely when P is finite. Unfortunately, P is not expected to be finite in practice. For example, P may include all the synchronization points at the beginning of the main loop in a reactive system implementation. Nevertheless, in practice it is often the case that we can over-approximate infinite sets symbolically. For example, we can use a

logical formula φ to describe a symbolic state, which denotes a potentially infinite set $[\![\varphi]\!]$ of concrete states that satisfy it. Then we may be able to describe the sets of states S_i and C_i of the cut transition systems T_i ($i \in \{1, 2\}$) with finite sets $\overline{S_i}$ and $\overline{C_i}$, respectively, of symbolic states. Similarly, symbolic pair (φ, φ') can describe infinite sets $[\![(\varphi, \varphi')]\!]$ of pairs of states in the two transition systems, related through free/symbolic variables that φ and φ' can share. Then we may also be able to describe P as a finite set $\overline{\mathsf{P}}$ of pairs of symbolic states. If all these are possible, then Algorithm 3.1 can be modified by replacing the boxed expressions with their symbolic variants (grayed); n, n', n_1, n_2, p_1, p_2, etc., are symbolic now.

Given an operational semantics of a programming language, $\mathbb{K}$ provides us with an API to calculate symbolic successors of symbolic program configurations. This allows us to conveniently implement the symbolic $\overline{\Rightarrow_i}$ transitions at line 19. Also, $\mathbb{K}$ is fully integrated with the Z3 solver [71], allowing us to implement the set inclusion checks, i.e., $[\![(n_1, n_2)]\!] \subseteq [\![\overline{P}]\!]$ (at line 9) and $[\![n']\!] \subseteq [\![\overline{C_i}]\!]$ (at line 21), by requesting Z3 to solve the implications of the corresponding formulae. (See below.)

It is clear that the symbolic variant of Algorithm 3.1 terminates provided that Z3 terminates. Also, working symbolically allows us to usually eliminate the restriction that T_i must be finitely branching, as infinite branching can often be modeled symbolically (e.g., a random number generator can be modeled as a fresh symbolic variable).

Optimizing SMT Queries Before checking the symbolic set inclusion $[\![(n_1, n_2)]\!] \subseteq [\![\overline{P}]\!]$, we check first the equivalence between the path conditions of the two symbolic states n_1 and n_2, since the SMT query for checking the set inclusion becomes much simpler when the two path conditions are equivalent.[5] Let the path conditions of n_1 and n_2 be φ_1 and φ_2, respectively. Then, we need to prove $\varphi_1 \Rightarrow \varphi_2$ and $\varphi_2 \Rightarrow \varphi_1$ to prove the path condition equivalence. For proving $\varphi_1 \Rightarrow \varphi_2$, we ask Z3 to prove that its negation is unsatisfiable, that is, that $\varphi_1 \wedge \neg\varphi_2$ is unsatisfiable. (Similarly for proving $\varphi_2 \Rightarrow \varphi_1$ as well.) However, we found that Z3 performs poorly for proving the unsatisfiability of $\varphi_1 \wedge \neg\varphi_2$, especially because of the negation applied to φ_2 that involves existential quantifiers.

To improve the performance of Z3 solving, we devised the following optimization that is applicable when the underlying transition systems are deterministic. Suppose that $N_1 = \{n_1, n_1'\}$ and $N_2 = \{n_2, n_2'\}$ (at line 7). Let the path condition of n_1, n_2 and n_2' be φ_1, φ_2 and φ_2', respectively. Since the transition systems are deterministic, we have that $\varphi_2 \vee \varphi_2'$ is tautology and φ_2 is disjoint from φ_2'. Thus, $\varphi_1 \wedge \neg\varphi_2$ is equivalent to $\varphi_1 \wedge \varphi_2'$. Now, for

[5] In case that the two path conditions are not equivalent, we can split the symbolic states with different path conditions and re-run the loop (lines 8–10).

```
int foo(unsigned n) {
  int i = 0;
  while (i < n) {
    i = i + 1;
  }
  return i;
}
```

```
int foo(unsigned n) {
  int i = 0;
  while (i < n) {
    i = i + 2;
  }
  return i;
}
```

Figure 3.5: Program transformation example. Two programs are equivalent provided that `n` is an even natural number.

proving $\varphi_1 \Rightarrow \varphi_2$, we ask Z3 to prove the unsatisfiability of $\varphi_1 \wedge \varphi_2'$ (instead of $\varphi_1 \wedge \neg\varphi_2$). Note that Z3 performs much better for solving the positive form $\varphi_1 \wedge \varphi_2'$ than the original negative form $\varphi_1 \wedge \neg\varphi_2$, even though the two are logically equivalent in theory.

This optimization has been adopted in our TV prototypes, utilizing the fact that both relevant language semantics (LLVM and x86) are deterministic.

Example Application of Algorithm 3.1 We implemented the symbolic variant of Algorithm 3.1 in a tool called KEQ for checking language-independent program equivalence.[6] To illustrate how KEQ works, consider the example in Figure 3.5. At the beginning of the programs, we have the symbolic synchronization point p_{init} which is a triple $(s_{p_{\mathsf{init}}}, s'_{p_{\mathsf{init}}}, \psi_{p_{\mathsf{init}}})$, where

$$s_{p_{\mathsf{init}}} \equiv \mathtt{i} \mapsto i \ * \ \mathtt{n} \mapsto n \quad \text{where} \ \ n \bmod 2 = 0 \tag{3.7}$$

$$s'_{p_{\mathsf{init}}} \equiv \mathtt{i} \mapsto i' \ * \ \mathtt{n} \mapsto n' \quad \text{where} \ \ n' \bmod 2 = 0 \tag{3.8}$$

$$\psi_{p_{\mathsf{init}}} \equiv n = n' \tag{3.9}$$

$s_{p_{\mathsf{init}}}$ and $s'_{p_{\mathsf{init}}}$ are the symbolic state of the first and second program, respectively ($*$ is a separator for map bindings), and $\psi_{p_{\mathsf{init}}}$ is the constraint for $s_{p_{\mathsf{init}}}$ and $s'_{p_{\mathsf{init}}}$ to be related, essentially saying that the inputs of the two programs are the same and they are even. Mathematically, p_{init} denotes the set of infinitely many pairs of states $\{(\mathtt{i} \mapsto i * \mathtt{n} \mapsto n, \ \mathtt{i} \mapsto i' * \mathtt{n} \mapsto n) \mid i, i', n \in \mathbb{N} \ \wedge \ n \text{ is even}\}$. Also, we have a synchronization point p_{loop} at the beginning of each loop iteration (i.e., the loop head), which is a triple $(s_{p_{\mathsf{loop}}}, s'_{p_{\mathsf{loop}}}, \psi_{p_{\mathsf{loop}}})$, where

$$s_{p_{\mathsf{loop}}} \equiv \mathtt{i} \mapsto i \ * \ \mathtt{n} \mapsto n \quad \text{where} \ \ i \bmod 2 = 0 \ \wedge \ n \bmod 2 = 0 \tag{3.10}$$

$$s'_{p_{\mathsf{loop}}} \equiv \mathtt{i} \mapsto i' \ * \ \mathtt{n} \mapsto n' \quad \text{where} \ \ i' \bmod 2 = 0 \ \wedge \ n' \bmod 2 = 0 \tag{3.11}$$

[6] KEQ also supports program refinement, but for simplicity we only discuss equivalence.

$$\psi_{p_{\mathsf{loop}}} \equiv i = i' \ \wedge \ n = n' \tag{3.12}$$

Finally, we have a synchronization point p_{final} at the end of the programs, which is a triple $(s_{p_{\mathsf{final}}}, s'_{p_{\mathsf{final}}}, \psi_{p_{\mathsf{final}}})$, where

$$s_{p_{\mathsf{final}}} \equiv \mathtt{i} \mapsto i \ * \ \mathtt{n} \mapsto n \tag{3.13}$$
$$s'_{p_{\mathsf{final}}} \equiv \mathtt{i} \mapsto i' \ * \ \mathtt{n} \mapsto n' \tag{3.14}$$
$$\psi_{p_{\mathsf{final}}} \equiv i = i' \ \wedge \ n = n' \tag{3.15}$$

Note that p_{final} is relaxed, capturing more states than the reachable states. This is allowed as long as it is admitted by the acceptability relation (in this case, equality between the same variables). Indeed, the more synchronization points are relaxed, the easier it is to automatically generate them (e.g., by instrumenting a compiler). Below we will show this relaxed synchronization point is enough to prove the equivalence.

Next we illustrate how KEQ symbolically runs Algorithm 3.1. Let $\mathsf{P} = \{p_{\mathsf{init}}, p_{\mathsf{loop}}, p_{\mathsf{final}}\}$. First, KEQ picks a point (say p_{init}) from P (line 2 of Algorithm 3.1) and executes the function `check` with it. In `check`, it first symbolically executes each program (lines 7 and 19) until they reach another synchronization point (line 21). In this case they reach states s_1 and s'_1 that are matched by $s_{p_{\mathsf{loop}}}$ and $s'_{p_{\mathsf{loop}}}$ respectively, where:

$$s_1 = \mathtt{i} \mapsto 0 \ * \ \mathtt{n} \mapsto n \quad \text{where} \ \ n \bmod 2 = 0 \tag{3.16}$$
$$s'_1 = \mathtt{i} \mapsto 0 \ * \ \mathtt{n} \mapsto n' \quad \text{where} \ \ n' \bmod 2 = 0 \tag{3.17}$$

KEQ checks if $(s_1, s'_1, \psi_{p_{\mathsf{init}}})$ is matched by p_{loop} (line 9), which is true. Since s_1 is the only pair that reaches p_{loop}, the `check` function returns true (line 12).

Next, suppose KEQ picks p_{loop} (line 2). Symbolic execution starting from p_{loop} yields two pairs of symbolic traces, that reach synchronization points p_{loop} (through the for-loop body) and p_{final} (escaping the for-loop), respectively. Let us consider the first case. We have the pair of states s_2 and s'_2 that are matched by $s_{p_{\mathsf{loop}}}$ and $s'_{p_{\mathsf{loop}}}$ respectively, where:

$$s_2 = \mathtt{i} \mapsto i + 2 \ * \ \mathtt{n} \mapsto n \quad \text{where} \ \ i \bmod 2 = 0 \ \wedge \ n \bmod 2 = 0 \tag{3.18}$$
$$s'_2 = \mathtt{i} \mapsto i' + 2 \ * \ \mathtt{n} \mapsto n' \quad \text{where} \ \ i' \bmod 2 = 0 \ \wedge \ n' \bmod 2 = 0 \tag{3.19}$$

Note that s_2 is resulted from executing the loop twice, since the result of the single loop iteration ($\mathtt{i} \mapsto i + 1 \ * \ \mathtt{n} \mapsto n$) is not matched by $s_{p_{\mathsf{loop}}}$ because $(i + 1) \bmod 2 \neq 0$, that is, it is *not* in the cut (line 21). KEQ checks if $(s_2, s'_2, \psi_{p_{\mathsf{loop}}})$ is matched by p_{loop} (line 9),

```
int cnt(unsigned n) {
  int c = 0;
  int i = 0;
  while (i < n) {
    i = i + 1;
    c = c + 1;
  }
  return c;
}
```

```
int cnt(unsigned n) {
  int c = 0;
  int i = n;
  while (i > 0) {
    i = i - 1;
    c = c + 1;
  }
  return c;
}
```

Figure 3.6: Two equivalent programs with the out-of-order loop iteration.

which is true. (Here KEQ needs to rename the free variables in p_{loop} to avoid the variable capture.) The other case is similar and `check` with p_{loop} eventually returns true. Then, KEQ continues to pick from the remaining synchronization points and execute `check` with each of them (loop at lines 2-4), eventually returning true (line 5).

On the other hand, Figure 3.6 shows an example of equivalent programs with the out-of-order loop. The first program iterates the loop increasing the loop index `i`, while the second program iterates decreasing `i`. The cut-bisimulation is expressive enough to capture this out-of-order execution. The following three synchronization points are sufficient for KEQ to prove the equivalence:

- At the beginning: $\mathtt{n} = \mathtt{n}'$
- At the loop head: $\mathtt{n} = \mathtt{n}'$, $\mathtt{i} + \mathtt{i}' = \mathtt{n}$, and $\mathtt{c} = \mathtt{c}'$
- At the end: $\mathtt{c} = \mathtt{c}'$

where the primed variables refer to the second program's variables. Note that the non-trivial part of the synchronization points is the equality $\mathtt{i} + \mathtt{i}' = \mathtt{n}$, but the compiler can provide this information that must be known to perform such an out-of-order loop transformation.

3.6 PRELIMINARY EVALUATION OF KEQ

As a preliminary evaluation, we revisit the example of Figure 3.1 and prove their equivalence using KEQ. The transformation presented in these programs is called partial redundancy elimination (PRE) and it is a common transformation in compiler literature. Figure 3.7 shows two LLVM programs that correspond to the code shown in Figure 3.1. The non-deterministic choice operator is simulated in LLVM with a call to an external function, `@check`, that returns a boolean value, at line 9 in Figure 3.7(a) and line 8 in Figure 3.7(b).

```
 1: define void @pre1() {
 2: entry:                          ; p0
 3:  br label %while.body
 4:
 5: while.body:                     ; p1, p2
 6:  %y.0 = phi i32 [1, %entry], [%y.1, %if.end]
 7:  %x.0 = phi i32 [1, %entry], [%x.1, %if.end]
 8:  %inc = add i32 %x.0, 1
 9:  %call = call i32 @check()
10:  %tobool = icmp ne i32 %call, 0
11:  br i1 %tobool, label %if.then, label %if.else
12:
13: if.then:
14:  %inc1 = add i32 %y.0, 1
15:  br label %if.end
16:
17: if.else:
18:  %mul = mul i32 %y.0, 2
19:  br label %if.end
20:
21: if.end:
22:  %y.1 = phi i32 [%inc1, %if.then],
                    [%y.0,  %if.else]
23:  %x.1 = phi i32 [%inc,  %if.then],
                    [%mul,  %if.else]
24:  br label %while.body
25:
26: return:
27:  ret void
28: }
29:
30: declare i32 @check()
```

(a) Before PRE

```
 1: define void @pre2() {
 2: entry:                          ; p0
 3:  br label %while.body
 4:
 5: while.body:                     ; p1, p2
 6:  %y.0 = phi i32 [1, %entry], [%y.1, %if.end]
 7:  %x.0 = phi i32 [1, %entry], [%x.1, %if.end]
 8:  %call = call i32 @check()
 9:  %tobool = icmp ne i32 %call, 0
10:  br i1 %tobool, label %if.then, label %if.else
11:
12: if.then:
13:  %inc = add i32 %x.0, 1
14:  %inc1 = add i32 %y.0, 1
15:  br label %if.end
16:
17: if.else:
18:  %mul = mul i32 %y.0, 2
19:  br label %if.end
20:
21: if.end:
22:  %y.1 = phi i32 [%inc1, %if.then],
                    [%y.0,  %if.else]
23:  %x.1 = phi i32 [%inc,  %if.then],
                    [%mul,  %if.else]
24:  br label %while.body
25:
26: return:
27:  ret void
28: }
29:
30: declare i32 @check()
```

(b) After PRE

Figure 3.7: A simple partial redundancy elimination transformation in LLVM. The two LLVM programs mirror the loops shown in Figure 3.1. The `while` loop is diverging (i.e., non-terminating) and the non-deterministic condition is simulated by a call to an external function `@check`. The `add`-operation in line 8 of the original function `pre1` is moved inside the `if.then` block as shown in line 13 of the transformed function `pre2`.

Figure 3.8 summarizes the synchronization points that KEQ needs to prove to establish a cut-bisimulation: one point at the entry of the code ($p0$) and two points at the entry of the loop, one for entry from outside ($p1$) and one for entry through the back edge ($p2$). Once instantiated with an LLVM semantics in $\mathbb{K}$, KEQ took as input the set of synchronization points and successfully proved equivalence of the two LLVM programs. The KEQ proof took 26 seconds in a laptop machine with an Intel Core i7-6500U processor at 2.50GHz and 12GB of memory.

Let us illustrate how KEQ prove that the synchronization points establish a cut-bisimulation. It is trivial to see that all traces starting from $p0$ will reach $p1$ in both programs. Traces

Sync Point	*Previous Block*	*Constraints*
(0)	-	-
(1)	`%entry`	-
(2)	`%if.end`	$\texttt{\%x.1} = \texttt{\%x.1}' \ \wedge \ \texttt{\%y.1} = \texttt{\%y.1}'$

Figure 3.8: Synchronization points for the PRE transformation in Figure 3.7. The primed variable names refer to the variables in Figure 3.7(b).

```
volatile int G;

void f(int X, int A) {
  int a;
  while (1) {
    a = A + X;
    G += a;
  }
}
```

(a) *C program for LICM example*

```
unsigned f(unsigned n) {
  unsigned x = 5;
  unsigned i;

  for (i = 0; i < n; ++i)
    if (x > 5) ++x;

  return x;
}
```

(b) *C program for SCCP example*

Figure 3.9: C programs for the LICM and SCCP equivalence proofs.

starting from $p1$ will reach the entry of the loop through the back edge with `%x.1` = `%x.1`$'$ = 2 and `%y.1` = `%y.1`$'$ = *if-then-else*(`@check()`, 2, 1), where primed variable names refer to the variables in Figure 3.7(b). These values satsify the constraints for synchronization point $p2$, so it will be reached from $p1$. Finally, traces starting from $p2$, where `%x.1` = `%x.1`$'$ = X and `%y.1` = `%y.1`$'$ = Y, will reach the entry of the loop through the back edge with `%x.1` = `%x.1`$'$ = *if-then-else*(`@check()`, $X + 1, 2Y$) and `%y.1` = `%y.1`$'$ = *if-then-else*(`@check()`, $Y + 1, Y$). These values also satisfy the constraints for synchronization point $p2$, so it will be reached from $p2$.

Note that the set of synchronization points in Figure 3.8 is straightforward enough to be automatically inferred. Indeed, Necula *et al.* [15] proposed an inference algorithm of such synchronization points in their their translation validation system for the GNU C compiler transformation. Their system uses an informal notion of program equivalence that is reminiscent of cut-bisimulation, and we believe that it can be employed in our language-independent equivalence checking framework, so that their technique can be easily applied to other languages, which we leave as future work.

3.7 EQUIVALENCE PROOF EXAMPLES

This section presents two more example equivalence proofs using cut-bisimulation for the Loop Invariant Code Motion (LICM) and Sparse Conditional Constant Propagation (SCCP) transformations. Both are standard compiler optimizations found in most of modern production compilers.

3.7.1 Loop Invariant Code Motion (LICM)

The LICM transformation moves code out of a loop when the corresponding computation

```
 1: @G.0 = common global i32 0, align 4
 2:
 3: define void @f(i32 %A.0, i32 %X.0) {
 4: entry:   ; p0
 5:   br label %while.body
 6:
 7: while.body:
 8:   %a.0 = add i32 %A.0, %X.0
 9:   %0 = load volatile i32, i32* @G.0, align 4
10:   %1 = add i32 %0, %a.0
11:   store volatile i32 %1, i32* @G.0, align 4
12:   ; p1
13:   br label %while.body
14:
15: return:
16:   ret void
17: }
```

(a) *Before LICM*

```
 1: @G.1 = common global i32 0, align 4
 2:
 3: define void @f(i32 %A.1, i32 %X.1) {
 4: entry:   ; p0
 5:   %a.1 = add i32 %A.1, %X.1
 6:   br label %while.body
 7:
 8: while.body:
 9:   %0 = load volatile i32, i32* @G.1, align 4
10:   %1 = add i32 %0, %a.1
11:   store volatile i32 %1, i32* @G.1, align 4
12:   ; p1
13:   br label %while.body
14:
15: return:
16:   ret void
17: }
```

(b) *After LICM*

Figure 3.10: A simple LICM transformation. The `add`-operation in line 8 of the original function (a) can be hoisted out of the loop as shown in line 5 of the transformed function (b). Note that the `while`-loop is diverging (i.e., non-terminating). See Figure 3.9(a) for the original C program.

Sync Point	*Constraints*		
(0)	`%A.0 = %A.1,`	`%X.0 = %X.1,`	`[@G.0] = [@G.1]`
(1)	`%A.0 = %A.1,` `[@G.0] = [@G.1],`	`%X.0 = %X.1,` `%a.1 = %A.1 + %X.1`	`%a.0 = %a.1`

Figure 3.11: Synchronization points for the LICM translation in Figure 3.10.

can be proven to be independent of the loop iteration, thus it can be safely executed only once before entering the loop. Additionally, before attempting the transformation, the compiler should prove that the hoisted computation has no side-effects and/or that the original program would execute at least one loop iteration.

Figure 3.10 shows an LLVM IR function containing a loop with the loop invariant code in line 8 on the left and the transformed function after LICM where the loop invariant code has been hoisted out of the loop in line 5 on the right. The corresponding C program is shown in Figure 3.9(a). Intuitively, the two functions synchronize at the entry (point (0)) and at the end of each loop iteration (point (1)). We use non-terminating loops with side-effects to showcase the importance of using bisimulation-based proofs for program equivalence. Although both functions diverge, they should not be declared equivalent when they had a different effect on the value of the volatile global variable `@G`.

The intuition about where the two functions synchronize is reduced to a cut-bsimulation equivalence proof in a straight-forward way. It sufficies to formally describe the above synchronization points, prove that the individual states are indeed a cut for each corresponding function, and finally prove that the given synchronization point relation is indeed a cut-

```
 1: define i32 @f(i32 %N.0) {
 2: entry:     ; p0
 3:   br label %for.cond
 4:
 5: for.cond:
 6:   %x = phi i32 [ 5, %entry ], [ %y, %for.inc ]
 7:   %i = phi i32 [ 0, %entry ], [ %inc.i, %for.inc ]
 8:   %cmp.for = icmp ult i32 %i, %N.0
 9:   br i1 %cmp.for, label %for.body, label %for.end
10:
11: for.body: ; p1
12:   %cmp.if = icmp ugt i32 %x, 5
13:   br i1 %cmp.if, label %if.then, label %if.end
14:
15: if.then:
16:   %inc.x = add i32 %x, 1
17:   br label %if.end
18:
19: if.end:
20:   %y = phi i32 [ %inc.x, %if.then ], [ %x, %for.body ]
21:   br label %for.inc
22:
23: for.inc:
24:   %inc.i = add i32 %i, 1
25:   br label %for.cond
26:
27: for.end:  ; p2
28:   ret i32 %x
29: }
```

(a) *Before SCCP*

```
1: define i32 @f(i32 %N.1) {
2: entry:   ; p0 p1
3:   br label %for.end
4:
5: for.end: ; p2
6:   ret i32 5
7: }
```

(b) *After SCCP*

Figure 3.12: A simple SCCP transformation: the for loop in lines 5-25 of the original function (a) can be removed as dead code, and the constant 5 can be propagated to the return instruction as shown in line 6 of the transformed function (b). See Figure 3.9(b) for the original C program.

bisimulation. The formal descriptions of the synchronization points are given in Figure 3.11. The proof for individual states being a cut is straight-forward. In general, it is acceptable to trust an automated proof generator only to produce synchronization point relations using individual states that define a cut for the corresponding program. Finally, the symbolic version of algorithm 3.1 can be used to prove the relation to be a cut-bisimulation, hence the functions are equivalent.

Notice that for the synchronization point (1), in addition to the equality constraints between the two functions' variables, we need an extra constraint that relates the values of %A.1, %X.1, and %a.1. This constraint will allow us to prove that the cut-successors of any pair of states described by (1) are also a pair of states described by (1). An automated proof generator can generate this constraint using compiler-provided hints or heuristics about the effect of the LICM transformation. This type of constraint generation beyond simple variable equality constraints is typical for equivalence proofs of program transformations.

3.7.2 Sparse Conditional Constant Propagation (SCCP)

The SCCP transformation propagates constant values and removes infeasible branches that are discovered through constant propagation. In the example shown in Figure 3.12 with the corresponding C program shown in Figure 3.9(b), we see that the constant 5 is

Sync Point	*Constraints*
(0)	`%N.0 = %N.1`
(1)	`%N.0 = %N.1, %x = 5`
(2)	`%N.0 = %N.1, %x = 5`

Figure 3.13: Synchronization points for the SCCP translation in Figure 3.12.

assigned to %x in line 6 during the first iteration of the loop in lines 5-25 of the original function. Therefore, the branch in lines 12-13 will be taken to line 19 and the constant 5 will also be assigned to %y in line 6 during the first iteration, hence %x is assigned the constant 5 during the second and all subsequent iterations. By proving that %x is constant (equal to 5), the compiler can remove the infeasible branch in lines 11-17, and afterwards the whole loop in lines 5-25 as dead code. Finally the constant 5 can be propagated to the return instruction in line 28.

The cut-bisimulation equivalence proof follows from the observation that each iteration of the loop in the original function synchronizes with the start state of the transformed function, as well as from the fact that $\%\mathtt{x} = 5$ is an invariant for the execution of the original function. Figure 3.13 shows the formal synchronization points that capture these observations, which can be automatically generated by a proof generator that uses compiler-provided hints or heuristics about the effect of the SCCP transformation. Again, it is straightforward to see that the individual states in the synchronization points constitute a cut for the corresponding functions and using KEQ we can prove that the synchronization point relation is a cut-bisimulation.

CHAPTER 4: TRANSLATION VALIDATION FOR INSTRUCTION SELECTION

4.1 INTRODUCTION AND MOTIVATING EXAMPLE

We present here our prototype Translation Validation (TV) system for the Instruction Selection phase [19] of the LLVM compiler [1]. As discussed in Chapter 2, there is a rich literature of successful TV systems for compilation verification, but each one is assuming a fixed common intermediate language for input and output programs. Thus, none of these previous systems would be able to verify a compilation transformation like Instruction Selection, which converts between two different IRs. The Instruction Selection phase of LLVM is a sophisticated transformation that translates LLVM IR [9] to Machine IR [21] representing the x86-64 instruction set.

In this section, we present a TV system that comprises of modular components that can be reused with minimal effort for the various transformations and languages found in the compiler. Specifically, the key insight underlying our work is that two of the three TV system components (see Section 1.5) can be generalized to be *transformation- and language-independent*: the formal notion of equivalence, and the proof system. We use KEQ as our proof system and Cut-bisimulation as the notion of equivalence, as presented in Chapter 3. An equivalence proof with KEQ involves proving that a given verification condition (VC), provided in the form a set of pairs of relevant program states or *synchronization points*, is a cut-bisimulation for the input and output programs.

For example, consider the simple C code shown in Figure 4.1. Figure 4.2 shows the mid-level internal representation (IR) of this code in the LLVM Compiler Infrastructure (the LLVM IR, or simply, LLVM), as well as the output of the instruction selection (ISel) phase of the compiler when compiling for x86-64. This phase is the primary language translation step beyond the front-end: it translates LLVM IR to a low-level IR called Machine IR representing

```
unsigned arithm_seq_sum(unsigned a0, unsigned d, unsigned n) {
  unsigned s = a0, a = a0, i;
  for (i = 1; i < n; ++i) {
    a = a + d;
    s = s + a;
  }
  return s;
}
```

Figure 4.1: Function to compute the sum of the first `n` elements of an arithmetic sequence with first element `a0` and step `d`.

```
define i32 @arithm_seq_sum(i32 %a0, i32 %d, i32 %n) {
entry:      ; p0
  br label %for.cond

for.cond:   ; p1, p2
  %s.0 = phi i32 [ %a0, %entry ], [ %add1, %for.inc ]
  %a.0 = phi i32 [ %a0, %entry ], [ %add, %for.inc ]
  %i.0 = phi i32 [ 1, %entry ], [ %inc, %for.inc ]
  %cmp = icmp ult i32 %i.0, %n
  br i1 %cmp, label %for.body, label %for.end

for.body:
  %add = add i32 %a.0, %d
  %add1 = add i32 %s.0, %add
  br label %for.inc

for.inc:
  %inc = add i32 %i.0, 1
  br label %for.cond

for.end:  ; p3
  ret i32 %s.0
}
```

(a) *LLVM IR*

```
arithm_seq_sum:
.LBB0:    ; p0
    %vr8_32  = COPY edx
    %vr7_32  = COPY esi
    %vr6_32  = COPY edi
    %vr9_32  = mov  1
    jmp  .LBB1
.LBB1:    ; p1, p2
    %vr0_32  = PHI  %vr6_32, .LBB0, %vr4_32, .LBB3
    %vr1_32  = PHI  %vr6_32, .LBB0, %vr3_32, .LBB3
    %vr2_32  = PHI  %vr9_32, .LBB0, %vr5_32, .LBB3
    %vr10_32 = sub  %vr2_32, %vr8_32
    jae  .LBB4
    jmp  .LBB2
.LBB2:
    %vr3_32  = add  %vr1_32, %vr7_32
    %vr4_32  = add  %vr0_32, %vr3_32
    jmp  .LBB3
.LBB3:
    %vr5_32  = inc  %vr2_32
    jmp .LBB1
.LBB4:
    eax      = COPY %vr0_32
    ; p3
    ret
```

(b) *Virtual x86*

Figure 4.2: The arithmetic sequence sum in LLVM IR and Virtual x86, as produced by Instruction Selection at optimization level O0. Comments in red show the synchronization points generated by our prototype.

a particular target instruction set (ISA). The Machine IR for x86-64 keeps some high-level abstractions such as an unlimited amount of virtual registers and support for SSA virtual registers, along with the x86-64 ISA opcodes; we call this output language "Virtual x86". In Figure 4.3, points $\{p0, p1, p2, p3\}$ are corresponding synchronization points where state comparisons are valid for live values between the LLVM IR and the machine code generated by the ISel phase. Using these points and appropriate LLVM IR and Virtual x86 semantics definitions, KEQ proves that the synchronization point relation is a cut-bisimulation and hence the two programs are equivalent (see Section 3.5 for more details).

The final component of the TV system, the *verification condition generator*, has to take into account the specifics of the transformation in question (either using compiler-generated hints or heuristics inspired by the transformation's effect in input programs). For this reason, it is not clear how to effectively generalize it, although there have been examples in prior work of verification condition generators able to work with a wider range of transformations (see Chapter 2). The verification condition generator for Instruction Selection is implemented a python script that relies on a minimal hint generator added to the LLVM compiler. The hint generation code that we needed to add to LLVM contains less than 500 lines of C++. This code is conceptually simple and requires only standard compiler skills. Most of it simply outputs information that the compiler already computes as part of the instruction selection. For comparison, instruction selection itself uses more than 140,000 lines of code. Using the compiler-generated hints, the generator produces a set of synchronization points that are given to KEQ to check for a bisimulation relation, thus proving equivalence.

Sync Point	*Prev BB* (LLVM)	*Prev BB* (Vx86)	*Equality Constraints*	
$p0$	-	-	`%a.0 = edi,`	`%d = esi,`
			`%n = edx,`	
$p1$	`%entry`	`.LBB0`	`%d = %vr7_32,`	`1 = %vr9_32,`
			`%a.0 = %vr6_32,`	`%n = %vr8_32,`
$p2$	`%for.inc`	`LBB3`	`%add = %vr3_32,`	`%n = %vr8_32,`
			`%add1 = %vr5_32,`	`%d = %vr7_32,`
			`%inc = %vr5_32`	
$p3$ (exit)	-	-	`%s.0 = eax`	

Figure 4.3: Synchronization points for the translation of the arithmetic sequence sum in Figure 4.2. A more detailed explanation of how KEQ uses these points to prove equivalence will be given in Subsection 4.2.9

Finally, in order to use KEQ for the Instruction Selection phase of LLVM, we have developed $\mathbb{K}$ semantic definitions of a subset of the LLVM and Virtual x86 languages which are the input and output languages of this transformation.

In short, this work presents the first translation validation system for compilation verification that can handle a transformation with different input and output languages. This feature is a consequence of our design: a modular system that maximizes reuse of components across the compilation path. Specifically, the semantic definitions as well as the proof system used in the prototype are reused as-is in the Register Allocation prototype presented in Chapter 5, and only the verification condition generator is differs between the two systems.

4.2 TRANSLATION VALIDATION FOR LLVM INSTRUCTION SELECTION

Here we describe the application of the proposed equivalence checking algorithm in a translation validation system for the Instruction Selection phase of LLVM. This phase translates LLVM intermediate representation (IR) into various target instruction sets, and we focus on the x86-64 target for the scope of this work. We chose this particular application because it is a non-trivial component of a widely-used mature compiler that operates with different input and output languages. Moreover, in an LLVM-based compiler (e.g., Clang [72], Swift [73], Julia [74]), this phase is the primary language translation step beyond the front-end: it converts the mid-level IR to the low-level Machine IR. As explained in Chapter 2, none of the existing TV techniques can be directly used to validate instruction selection,

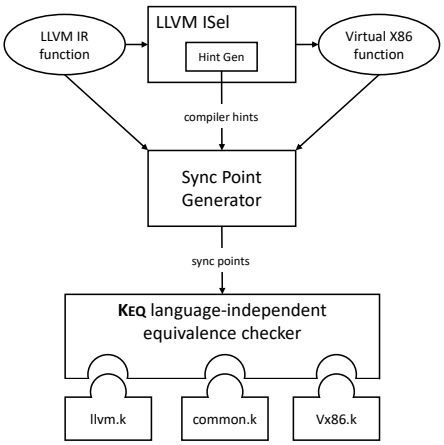


Figure 4.4: Translation validation system for LLVM ISel phase

which translates between two fundamentally different languages.

The various components of the system along with the Instruction Selection (ISel) phase itself are shown in Figure 4.4. ISel translates LLVM IR to Machine IR; when targeting x86-64, the generated machine IR represents a slightly simplified version of the x86-64 instruction set that we call "Virtual x86". Let us discuss the various components of the TV system in more detail.

Verification Condition Generator We enhanced ISel with a hint generator to output information relevant to the specific translation instance. This information along with the input and output programs guides the generation of the synchronization points for the translation instance. These two components, the hint generator and the synchronization points generator, constitute the Verification Condition (VC) generator. A key observation is that the hint generator will be part of the compiler code base and maintained by compiler engineers, so we want its implementation to be possible without any formal methods expertise.

The specific strategy for synchronization point generation employed by our VC generator for ISel is described in more detail in Subsection 4.2.6. In general, the objective of a VC generator for use with KEQ is to provide a set of points that form a cut for the input and output programs and are adequate for a cut-bisimulation proof by KEQ. Determining a strategy for the generation of such points requires understanding of the target transformation's assumptions and effects on the code, as well as formal methods knowledge about bisimulation proofs.

Typically, a successful VC generator will need some information about the effects of the target transformation on the code. For example, our VC generator requires knowledge of the mapping from LLVM IR virtual registers to Virtual x86 virtual and/or physical registers that

was used during the translation of the target input program. Such information can be either automatically obtained by an appropriate inference algorithm or provided by enhancing the compiler with a hint generator. The former approach treats the compiler as a completely black box, while the latter trades off transparency for increased accuracy. In case of a hint generator, we emphasize that this specific component should not require any formal methods expertise to be developed or maintained.

Language Semantics The set of synchronization points is provided to KEQ, which is parameterized by the $\mathbb{K}$ semantic definitions of LLVM IR and Virtual x86. These are discussed in more detail in Subsections 4.2.2 and 4.2.3. In general, one needs a $\mathbb{K}$ semantic definition for the input and output languages involved in the target transformation (which may be identical if the language is preserved).

Acceptability Relation The acceptability relation is a formal way to abstract away the correspondence between program states in different languages: Recall that the cut-bisimulation theory is parameterized by a given acceptability relation, which relates equivalent states across the two programs. Such correspondence may not be trivial between two different languages. However it has no effect on the theory other than the requirement that the states related in the cut-bisimulation relation must also be related in the acceptability relation. In general, the acceptability relation can be arbitrarily complex to define for any two given different languages. In our system, we provide `common.k`, a third semantic definition accepted by KEQ, for formally defining complex acceptability relations. This way, we can at least make the formal definitions for the same language pairs reusable, similar to the language semantic definitions themselves. In the case of LLVM IR and Virtual x86, the acceptability relation is mostly straight-forward. Specifically, in our TV system, the `common.k` module contains various definitions of equivalent (or common) components of the state in the two languages. This serves as a shortcut so that these components need not be repeatedly marked as equivalent in every synchronization point. The most significant such component is the memory model used for the two definitions (see Subsection 4.2.4).

KEQ runs the equivalence checking algorithm presented in Section 3.5 on the given set of synchronization points and outputs a verdict that validates the translation instance or flags it as not validated. In our ISel TV system, we need to trust (beyond the KEQ implementation and the $\mathbb{K}$ semantic definitions) that the given synchronization points cover all entry points of each function of the program, and the synchronization points belong to the acceptability relation.[1] Note that we do *not* need to trust that other relevant points (e.g., exit points,

[1] In principle, the latter can be excluded from the trust base by verifying it in a separate process, but we

loopheads, etc.) are also covered by the set of synchronization points, because otherwise KEQ will fail.

4.2.1 LLVM Instruction Selection Phase

The ISel phase [19] of an LLVM-based compiler is responsible for translating LLVM IR into a selected target's instruction set. This is a non-trivial component of the compiler, implemented in more than 140,000 lines of C++ and TableGen [75] code (*excluding* target-specific code for back-end targets other than x86-64). During ISel, LLVM IR code is first converted into a target-independent direct acyclic graph (DAG) representation called SelectionDAG with one DAG generated per basic block. Next, the actual instruction selection happens by matching patterns of DAG subgraphs to new subgraphs that contain the target opcodes. Finally, the DAG nodes are linearized to produce instruction sequences for each basic block.

Our translation validation prototype has been developed for the version 5.0.2 of the LLVM compiler. There are two different algorithms for instruction selection in LLVM 5.0.2, namely SDISel and FastISel. SDISel is slower and more sophisticated, and is the default for compilation to native; FastISel is faster and used at O0 and in JIT compilation. Our prototype works on the SDISel algorithm with optimization level O0, since this level performs the least amount of extra transformations to the code and focuses mainly on the language translation. Optimizations enabled in higher levels include more aggressive pattern folding and more aggressive constant propagation. These are not conceptually affecting the applicability of our method, but may require more sophisticated hint generation that we currently do not support.

4.2.2 LLVM IR Semantics

A full documentation of the LLVM intermediate representation can be found in [9]. In our LLVM semantic definition, we model the `i1`, `i8`, `i16`, `i32`, and `i64` integer types, composite (arbitrarily nested) array and struct types, the corresponding pointer types, and the `getelementptr` instruction used to compute the address of an element nested within a composite type. We also model integer arithmetic operators, bitwise operators, and the integer/-pointer comparison operators. We model the control flow instructions for unconditional and conditional branches, as well as function calls and returns. Finally, the supported memory operations are loads, stores, and the `alloca` instruction for stack allocation of local variables.

only manually checked it since the acceptability relation is rather straightforward in this case.

The LLVM semantics uses the common memory model described in Subsection 4.2.4 as its memory abstraction. Our memory abstraction does not yet take alignment requirements into consideration, so we do not support programs that assume any kind of variable or load/store alignment.

4.2.3 Virtual x86 Semantics

The output of ISel is LLVM Machine IR, a low-level representation representing the opcodes and operand types of the selected target ISA. More specifically, the LLVM Machine IR is a register-based IR that is parametric to any number of ISA opcodes and physical registers. It also supports a number of higher-level features such as various pseudo-opcodes (such as `COPY`, `PHI`, and others), an unlimited number of virtual registers, a frame abstraction for modeling call stacks, and a jump table abstraction. The Machine IR used in the x86 backend of the LLVM compiler is then specialized by using all the x86 opcodes and the full x86 physical register file [18]. We call this version of the Machine IR "Virtual x86."

Our $\mathbb{K}$ semantic definition of Virtual x86 captures all the extended features except jump tables and also various features of x86-64. We model integer arithmetic operations and integer comparison operations, bitwise operations, the general-purpose physical registers, conditional and unconditional jump instructions as well as the flags and program counter registers, `eflags` and `rip`. The program address space is modeled (similar to the LLVM IR semantics) using the common memory model abstraction described in Subsection 4.2.4. We model a variety of move instructions that copy data between registers and memory.

4.2.4 Common Memory Model

Both the LLVM and Virtual x86 semantics definitions are using a common low-level sequentially consistent memory model. This simplifies the formal definition of equivalent memory configurations between LLVM and Virtual x86 programs (although it is not necessary). The semantic definition of this common memory model is part of `common.k` (see Figure 4.4).

In the following, we describe the various aspects of the common memory model and how they are utilized by the two languages.

Memory Address A memory address in the common memory model is generated by an allocation operation and/or offset addition and subtraction operations on an existing address. It is represented as a quad (P, L, A, S) where P is the 64-bit unsigned integer value of the address, L is an identifier of the (dynamic) memory allocation operation that allocated this

address, A is the base address of the allocation operation, and S is the size of the allocation operation. We keep the information about the allocation operation (identifier, base address, and size) as part of a memory address so that we can perform bounds checks that are needed by the LLVM semantics in order to capture out-of-bounds undefined behaviors. Both the LLVM and Virtual x86 semantics definitions are using the memory address quad as the value for integer and pointer objects, hence not distinguishing between integer and pointer values. Finally, note that symbolic memory addresses can appear during symbolic execution, for example when allocating a new memory location or loading a pointer from a memory location. These symbolic addresses adhere to certain theorems that are part of the memory model semantic definition, for example two address with different allocation identifiers are implied to not overlap.

Memory Contents Since both LLVM and Virtual x86 feature byte-addressable memories, a memory location in the common memory model also contains a single byte. A byte is represented as a triple (B, S, V), where V is the integer value to which this byte belongs, B is the order of the byte with 0 being the least significant, and S is the total size in bytes of the value V. We use this representation to efficiently convert between integer and byte values: For example a 32-bit integer i can be converted to 4 bytes $(0, 4, i)..(3, 4, i)$ and vice versa. Both LLVM and Virtual x86 semantics convert between scalar (and, in case of LLVM, array and struct) values and byte series, when interfacing with the common memory model.

Memory Map The common memory model defines a mapping from memory addresses to bytes that represents the contents of the memory during execution. It also defines a set of freed memory locations that is useful for capturing undefined behaviors due to out of bounds accesses and double free operations.

Memory Access Operations The common memory model provides semantics for memory allocation, deallocation, loading, and storing. All the memory operations use the memory address quad representation for address values and the byte representation for data values. Both LLVM and Virtual x86 semantics contain rules that convert their respective memory operations into the operations of the common memory model.

4.2.5 Execution State Representation

A set of synchronization points is required as an input for the equivalence checking algorithm. Before we discuss details about the generation of synchronization points, we briefly discuss how program states and execution traces are described in the context of a $\mathbb{K}$ semantic definition. A $\mathbb{K}$ configuration is a collection of cells representing various components of the program state (e.g. a cell for the physical register `%rsp`) which are populated with concrete

values (e.g. the address of the top of the stack) to represent a specific state of the program. The specification of the cells and their expected content is part of the $\mathbb{K}$ semantic definition of a language. The rest of the $\mathbb{K}$ semantic definition is a set of rules that describe how to transition from a given configuration to valid next ones according to the language semantics. Hence, program states are described by $\mathbb{K}$ configurations and execution traces are described by sequences of configurations produced by applying rules to a starting configuration. In the following we use the terms configuration and state indiscriminately.

4.2.6 Characterizing Synchronization Points

Each synchronization point is a pair of symbolic states of the input and output programs accompanied by a set of equality constraints over symbolic variables found in the two states. Figure 4.3 shows the synchronization points generated by our TV system for the programs in Figure 4.2. For example, the synchronization point p1 consists of a symbolic state for the LLVM IR program that represents states entering the `for.cond` basic block while coming from the `entry` basic block, and a symbolic state for the Virtual x86 program that represents states entering the `.LBB1` basic block while coming from the `.LBB0` basic block. In addition, the point represents only pairs of states that satisfy the equality constrains `%d` = `%vr7_32`, `%a.0` = `%vr6_32`, `%n` = `%vr8_32`, and `%vr9_32` = 1, where the names of the virtual registers serve also as the names of the symbolic values that they hold in the corresponding symbolic states. In general, each synchronization point describes a potentially infinite number of input and output program state pairs, one pair for each concrete substitution of the symbolic variables of the two states that satisfies both state constraints as well as the equality constraints of the synchronization point.

To prove cut-bisimulation, the synchronization points should be a cut for the programs, i.e., "covering" all possible program executions (see Chapter 3). In the rest of this subsection, we discuss how the synchronization point generator creates such a set of points to be given to KEQ using compiler-provided hints and static analysis results. Please note that this generator is specifically designed for the ISel pass of LLVM. As discussed earlier, different transformations would require different synchronization point generation strategies.

Function Granularity An important design decision is whether input/output function pairs should be treated independently or not. When function pairs are treated independently, the translation of each function is considered a unique instance of the equivalence checking problem. In this case, we can assume that the called functions will be translated correctly, and hence function calls to the same function are equivalent. This is the same assumption that the compiler makes when applying any intra-procedural transformation to a function,

and for this reason treating function pairs independently is a natural choice when doing translation validation of an intra-procedural transformation, such as ISel. This approach has the added benefit that it deals uniformly with cases of missing code and compile-time unknown callees: Every function call, whether the callee is known, unknown, or missing is treated the same.

Function Entry and Exit We generate synchronization points that cover the entry and corresponding exits of each function pair. These are the points $p0$ and $p3$ in Figure 4.3 for the program in Figure 4.2. We can infer the equality constraint for these points from the calling convention.

Loop Entry We also generate synchronization points that cover the entries of corresponding loops in order to cover states that belong to cycles. These are the points $p1$ and $p2$ in Figure 4.3 (one point per predecessor to expedite the symbolic execution of the phi instructions). The relation between loops in the input and output is provided as a compiler-generated hint. The equality constraints for these points relate corresponding live registers in the input and output. The register correspondence is provided as a compiler-generated hint. The liveness information is computed by a static analysis.

Callsites Assuming that calls starting from equivalent states result in returns to equivalent states, it suffices to generate synchronization points before and after corresponding callsites. A synchronization point before a callsite is treated as covering an exiting state (meaning that we do not symbolically execute the call itself in KEQ). The equality constraints for a point before a callsite are inferred from the calling convention. The equality constraints for a point after a callsite relate corresponding live registers in the input and output as well as the return value registers (inferred from the calling convention).

Memory state Finally, all synchronization points should contain constraints that ensure that corresponding memory objects accessible by the functions hold the same contents. Since our prototype uses a common memory model for input and output, this requirement is translated to a simple equality constraint between the whole memory of the input and output.

In summary, the only compiler generated hints required in our approach are pairs of corresponding LLVM and Virtual x86 virtual registers and of corresponding loops. The hint generator records and outputs these for each translation instance. Its implementation is trivial, adding just about 500 lines of C++ code to ISel, and does not require any formal methods expertise.

4.2.7 Trusted Synchronization Points

It is important to note that we only need to trust that the generated synchronization points: (a) cover all the entry program points in the two functions, and (b) describe equivalent states, i.e. states that are related in the acceptability relation. The two above requirements ensure that the generated synchronization points form a cut for the (terminating) programs and are related in acceptability relation. These are indeed the requirements, along with proving that the points form a cut-bisimulation, for an equivalence proof.

There is a clarification we need to make in the above observation. There is an extra requirement for a set of synchronization points to form a cut for a pair of (potentially non-terminating) programs: synchronization points need to cover corresponding exiting points as well as corresponding non-terminating execution paths. Moreover, the latter should be such that we can always reach a synchronization point in a finite number of steps when starting from a synchronization point. However, we do not need to trust that the generated synchronization points satisfy these properties. Indeed our equivalence checking algorithm will fail to terminate if these properties are not satisfied: When the KEQ symbolically explores a path leading to an uncovered pair of corresponding exiting points, it will fail to reach a synchronization point before the programs exit, hence failing to prove cut-bisimulation. Similarly, when starting from a synchronization point in a not properly covered, non-terminating path, the symbolic execution would need to explore an infinite number of steps to reach another point in the path. Thus, if the generated synchronization points do not form a cut for a non-terminating program, KEQ will fail to prove equivalence (it will in fact terminate with a time-out exception).

Other than that, we do not need to trust the placement of the rest of the generated synchronization points. For a pair of terminating programs, the points form a cut as long as the entry and exits of the two programs are covered and we only need to trust that the entry points are covered. For a pair of non-terminating programs, as discussed above, we do not need to trust that the non-terminating paths are covered appropriately.

4.2.8 Characterizing Undefined Behaviors

The ability to handle undefined behaviors is an important aspect of a practical TV system. Our prototype handles undefined behaviors related to memory out-of-bounds accesses as well as signed integer overflow.

We model these undefined behaviors by a set of uniquely marked error states. When such behavior can potentially happen in a symbolic state, our semantics have rules to conditionally

branch into an error state, which captures information about the nature of the reached undefined behavior. Our acceptability relation for LLVM IR and Virtual x86 relates LLVM error states to any Virtual x86 state. On the other hand, Virtual x86 error states are only related with relevant LLVM error states in the acceptability relation, e.g. the out-of-bounds access error state in Virtual x86 is related only to the out-of-bounds access error state in LLVM. This way KEQ automatically reverts to checking refinement in the presence of undefined behaviors.

4.2.9 Example of Validation with KEQ

To illustrate how KEQ works within the Instruction Selection translation validation prototype, consider the running example in Figure 4.2. At the beginning of the programs, we have the symbolic synchronization point $p0$ which is a triple $(s_{p0}, s'_{p0}, \psi_{p0})$, where

$$s_{p0} \equiv \%\texttt{a0} \mapsto a_0 * \%\texttt{d} \mapsto d_0 * \%\texttt{n} \mapsto n_0 \tag{4.1}$$

$$s'_{p0} \equiv \texttt{edi} \mapsto a'_0 * \texttt{esi} \mapsto d'_0 * \texttt{edx} \mapsto n'_0 \tag{4.2}$$

$$\psi_{p0} \equiv a_0 = a'_0 \ \wedge \ d_0 = d'_0 \ \wedge \ n_0 = n'_0 \tag{4.3}$$

are the symbolic state of the LLVM program, the symbolic state of the x86 program ($*$ is a separator for map bindings), and the constraint for s_{p0} and s'_{p0} to be related, essentially saying that the inputs of the two programs are the same. Mathematically, $p0$ denotes the set of infinitely many pairs of states $\{(s_{p0}, s'_{p0}) \mid \psi_{p0}\} = \{(\%\texttt{a0} \mapsto a * \%\texttt{d} \mapsto d * \%\texttt{n} \mapsto n, \ \texttt{edi} \mapsto a * \texttt{esi} \mapsto d * \texttt{edx} \mapsto n) \mid a, d, n \in \mathbb{N}\}$ (an over-approximation including all the pairs of interest). Symbolic synchronization points $p1$, $p2$, and $p3$ are similarly defined (see Figure 4.3).

Next we illustrate how KEQ symbolically runs Algorithm 3.1. Let $\mathsf{P} = \{p0, p1, p2, p3\}$. First, KEQ picks one point (say $p0$) from P (line 2 of Algorithm 3.1) and executes the function `check` with it. In `check`, it first symbolically executes each program (lines 7 and 19) until they reach another synchronization point (line 21). In our case they reach $p1$ with the pair of symbolic states

$$s_{p1} \equiv s_{p0} \tag{4.4}$$

$$s'_{p1} \equiv s'_{p0} * \%\texttt{vr8} \mapsto n'_0 * \%\texttt{vr7} \mapsto d'_0 * \%\texttt{vr6} \mapsto a'_0 * \%\texttt{vr9} \mapsto 1 \tag{4.5}$$

KEQ checks if $\{(s_{p1}, s'_{p1}) \mid \psi_{p0}\}$ is included in $p1$ (line 9), which is true.

Next, suppose KEQ picks $p2$ (line 2). Symbolic execution starting from $p2$ yields two pairs

Result	#Functions
Succeeded	4,331
Failed due to timeout	206
Failed due to out-of-memory	179
Other	16
Total	4,732

Figure 4.5: Translation validation results for GCC benchmark

of symbolic traces, that reach synchronization points $p2$ (through the for-loop body) and $p3$ (escaping the for-loop), respectively. Let us consider the first case. We have the pair of the final states $s_{p2} \equiv \%\texttt{d} \mapsto d_2 * \texttt{add1} \mapsto s_2 + a_2 + d_2 * \%\texttt{n} \mapsto n_2 * \texttt{add} \mapsto a_2 + d_2 * \texttt{inc} \mapsto i_2 + 1$ with the path condition $i_2 < n_2$ (due to `icmp ult`), and $s'_{p2} \equiv \%\texttt{vr7} \mapsto d'_2 * \%\texttt{vr4} \mapsto s'_2 + a'_2 + d'_2 * \%\texttt{vr8} \mapsto n'_2 * \%\texttt{vr3} \mapsto a'_2 + d'_2 * \%\texttt{vr5} \mapsto i'_2 + 1$ with the path condition $i'_2 - n'_2 < 0$ (due to `sub` and `jae`), where we have $\psi_{p2} \equiv a_2 = a'_2 \wedge d_2 = d'_2 \wedge s_2 = s'_2 \wedge i_2 = i'_2 \wedge n_2 = n'_2$. KEQ checks if $\{(s_{p2}, s'_{p2}) \mid \psi_{p2}\}$ is included in $p2$ (line 9), which is true. Regarding the path conditions, KEQ checks if $i_2 < n_2$ and $i'_2 - n'_2 < 0$ are equivalent given ψ_{p2}, which is true.

The other case for $p3$ is similar and `check` with $p2$ returns true. Then, KEQ continues to pick from the remaining synchronization points and execute `check` with each of them (loop at lines 2-4), eventually returning true (line 5).

4.3 EVALUATION ON COMPILATION OF REAL-WORLD CODE

We evaluate the Translation Validation system for Instruction Selection in two ways. First, we apply the system on the compilation of the GCC SPEC 2006 benchmark [26]. Second, we reintroduce several bugs that were found and fixed in the code of the Instruction Selection pass of LLVM and verify that our system does not validate translations that trigger said bugs and thus contain miscompilations.

4.3.1 Application to GCC from SPEC 2006

We applied the Translation Validation prototype to the source code of the GCC SPEC 2006 benchmark, a version of an important piece of software that affects the correctness of many other critical software systems (e.g., the Linux kernel). The GCC source code is comprised of 5572 C functions, which we compiled to LLVM IR using `clang-5.0.2` at optimization level -O0 and translated to Virtual x86 by the ISel pass of LLVM 5.0.2. For each verification run, we allocated 2 Intel Xeon CPU E7-8837 processors at 2.67GHz and

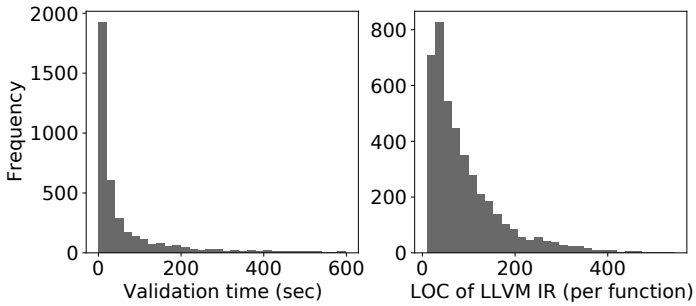


Figure 4.6: Distributions of validation time and code size

12GB of memory, with a timeout of 3 hours.

Out of the 5572 functions, our evaluation considered 4732 functions that are covered by our LLVM and x86 language semantics (Section 4.2.2 & 4.2.3). The remaining functions involve floating point, SIMD, or certain bitwise operations that are not supported by the current semantics. In the following discussion, 4732 will be the denominator of all percentages mentioned.

Figure 4.6 shows distributions of validation time and the code size of the functions. With the above hardware setup, the average time for processing a function in our GCC experiment is 150 seconds and the median is 0.8 seconds. Note that this does not include the time used for K to load the semantics and parse the input proofs.

Out of the 4732 functions, our prototype was able to formally verify the translation of 4331 functions (91.52%). Figure 4.5 categorizes the reasons for failure for the remaining 401 functions. We discuss these categories in more detail below.

Timeout 206 functions (4.35%) failed due to timeout (3 hour limit), and the Z3 solving time was the dominating factor. With the symbolic variant of Algorithm 3.1, KEQ may make multiple Z3 queries in each step containing path conditions, which grow significantly over time, particularly when there is a large number of complicated memory operations and branching conditions. Making things worse, the current integration of Z3 in the $\mathbb{K}$ framework does not use the incremental query solving feature of Z3, and thus solving each query needs to have a cold start even if many of the complex queries share significant sub-queries. We believe that this can be improved by integrating the incremental Z3 query solving to the $\mathbb{K}$ framework, which we leave as future work.

Out of Memory 179 functions (3.78%) failed with an out-of-memory exception. All these failures happened during the parsing of our synchronization point specifications, which were caused by performance issues in the $\mathbb{K}$ builtin parser. The $\mathbb{K}$ parser was designed to be

quite general, equipping a comprehensive parsing ambiguity resolving mechanism, which does not often scale well. This can be improved by using a more compact representation of the synchronization point specification to alleviate the burden on the $\mathbb{K}$ parser, or by using a more recent feature of $\mathbb{K}$ to generate a static parser ahead of time.

Inadequate Synchronization Points The rest of the failures (16 functions) are due to an inaccuracy in our liveness analysis, that resulted in a mismatch of LLVM and Virtual x86 live registers at the beginning of certain basic blocks, i.e. a live register in the x86 block with no live counter-part in the LLVM block. This caused the VC generator to generate an inadequate set of synchronization points for an equivalence proof. A more sophisticated liveness analysis would resolve this issue.

4.3.2 Evaluation with Real LLVM Bugs

We discuss two Instruction Selection bugs that made their way in the LLVM code base. Although the bugs are currently fixed, we were able to reintroduce them in the compiler and we attempt to validate translations that trigger the buggy code with our system. In both cases, the Translation Validation system could not verify the translation.

Write-After-Write Dependency Violation When Translating Store Instructions This bug causes a miscompilation that violates a write-after-write dependency for a memory location when subsequent overlapping stores access said location. The compiler erroneously reorders the two write accesses while attempting to optimize the compilation of the store instructions by merging them into fewer wider stores.

This is a bug for the x86-64 backend and it last appeared in clang 3.7.x (as a regression from older versions) for optimization levels -O2 and -O2 [76]. Figures 4.7 and 4.8 demonstrate the miscompilation. Figure 4.7 shows the LLVM code. The shown function performs 3 2-byte wide stores at offsets 2, 3, 1 of a global byte array. This means that the first two stores both write the byte at offset 3. A straight-forward correct translation to x86-64 is shown in Figure 4.8(a), while Figure 4.8 shows a correct optimized compilation: The third store has been merged into the first store that becomes a 4-byte wide store. This is correct because there is no dependency between the third store and any of the rest and the order of the first and second store has been preserved. On the other hand, Figure 4.8(b) shows the miscompilation due to the bug: This time the first store has been merged into the third thus reversing the write-after-write dependency between the first and second store.

```
@b = external global [8 x i8]

define void @foo() {
entry:
  store i16 0, i16* bitcast (i8* getelementptr inbounds ([8 x i8], [8 x i8]* @b, i64 0, i64 2) to i16*)
  store i16 2, i16* bitcast (i8* getelementptr inbounds ([8 x i8], [8 x i8]* @b, i64 0, i64 3) to i16*)
  store i16 1, i16* bitcast (i8* getelementptr inbounds ([8 x i8], [8 x i8]* @b, i64 0, i64 0) to i16*)
  ret void
}
```

Figure 4.7: LLVM function for the write-after-write dependency violation bug

```
foo:
  movw $0, b+2(%rip)
  movw $2, b+3(%rip)
  movw $1, b(%rip)
  retq
```

(a) *Simple correct translation*

```
foo:
  movw $2, b+3(%rip)
  movl $1, b(%rip)
  retq
```

(b) *Optimized incorrect translation*

```
foo:
  movl $1, b(%rip)
  movw $2, b+3(%rip)
  retq
```

(c) *Optimized correct translation*

Figure 4.8: x86 functions for the write-after-write dependency violation bug

Our system catches the bug, since KEQ cannot prove the candidate synchronization point set is a cut-bisimulation. Indeed, starting from the entry point and assuming that the global memory contents are the same, the symbolic execution of the input and output programs leads to different memory contents for the byte at offset 3, hence not allowing KEQ to prove the constraint for equal memory contents at the exiting synchronization point.

Incorrect Load Narrowing with Non-Power-of-Two Types This bug causes a miscompilation that leads to an out-of-bounds memory access. The compiler erroneously compiles a 4-byte wide load to an 8-byte wide load when attempting to narrow a load that accesses a memory location holding a non-power-of-two bitwidth type.

This is a bug for the x86-64 backend and it was found in clang 2.6.x for optimization levels -O2 and higher [77]. Figure 4.9 and 4.10 demonstrate the miscompilation. Figure 4.9 shows the LLVM code. The shown function loads from a memory location holding a 12-byte (`i96`) integer. It then logically shifts right the lower 8 bytes and stores the remaining 4 bytes, zero-extended as an 8-byte (`i64`) integer to another memory location. A correct translation to x86-64 is shown in Figure 4.10(a): The code first loads the upper 4 bytes of the source location into `eax`, thus zeroing-out the higher 4 bytes of the 64-bit general purpose register `rax`, according to the x86-64 semantics. It then stores `rax` which now holds the correct contents (the higher 4 bytes of the store zero-extended as an 8-byte integer) to the destination memory location. On the other hand, Figure 4.10(b) shows the miscompilation due to the bug: This time the load is 8-byte wide, thus accessing 4 out-of-bounds bytes that may contain garbage or cause a segmentation fault. In the former case, the value stored at the destination is incorrect because the 4 higher bytes are not zeroed-out but rather have random values.

```
@a = external global i96, align 4
@b = external global i64, align 8

define void @foo() {
  %srcval = load i96, i96* @a, align 4
  %tmp96 = lshr i96 %srcval, 64
  %tmp64 = trunc i96 %tmp96 to i64
  store i64 %tmp64, i64* @b, align 8
  ret void
}
```

Figure 4.9: LLVM function for the load narrowing bug

```
foo:
  movl a+8(%rip), %eax
  movq %rax, b(%rip)
  retq
```

(a) *Optimized correct translation*

```
foo:
  movq a+8(%rip), %rax
  movq %rax, b(%rip)
  retq
```

(b) *Optimized incorrect translation*

Figure 4.10: x86 functions for the load narrowing bug

Similar to previous case, our system catches the bug, since KEQ cannot prove the candidate synchronization point set is a cut-bisimulation. Indeed, starting from the entry point and assuming that the global memory contents are the same, the symbolic execution of the output x86 program branches into an out-of-bounds error state in addition to reaching the exiting point. This error state cannot be matched with any state in the input LLVM program, hence not allowing KEQ to prove cut-bisimulation: there is a behavior in the output that is not found in the input[2].

[2]In fact, in this case we cannot even prove that the output simulates the input.

CHAPTER 5: TRANSLATION VALIDATION FOR REGISTER ALLOCATION

5.1 INTRODUCTION

Register allocation is a phase present in every modern optimizing compiler that determines the mapping from program variables to physical registers of the target instruction set architecture. Typically compilers convert the input program to an intermediate language representation that is amenable to analysis and optimization, e.g., GIMPLE [17] in GCC [2] and the LLVM IR [9] in the LLVM compiler infrastructure [1]. These intermediate languages feature an infinite number of temporary variables (sometimes called virtual registers) that need to be mapped to the limited physical register file of the target instruction set architecture in an efficient way that maximizes register usage and minimizes the need to use memory as temporary storage (spilling). This is an NP-complete problem (specifically, it can be reduced to graph coloring [78]) and the Register Allocation phase of modern compilers includes a complex set of transformations that aims to give a best-effort solution: spilling [78, 79], register coalescing [80], live range splitting [81], and rematerialization [82].

Verification of register allocation is an important phase for compilation verification due to both its complexity of implementation and its ubiquity as part of compiler backends. Unfortunately, it has been proven challenging from the point of view of verified compilers. Some of its transformation algorithms have been proven hard to normally prove correct without sacrificing some of their complexity and thus their optimization capabilities. Specifically, a state-of-the-art verified C compiler, CompCert [10], is shown to use a sub-optimal spilling algorithm for the transformation, so that the implementation can be proven correct.

In this Chapter, we present our Translation Validation system for compilation verification of the Register Allocation phase of the LLVM compiler. Similar to the Instruction Selection system, the system accepts the input and output programs and a set of verification conditions, i.e formal descriptions of correspondence between variables in the two programs, generates a machine-checkable proof of equivalence, and verifies correctness by checking said proof.

For Register Allocation, we reuse the whole equivalence proof system used in Instruction Selection (cut-bisimulation based equivalence plus KEQ), while only implementing a transformation-specific verification condition generator. We implement a verification condition generator specific to the Register Allocation phase in a modern optimizing compiler, including advanced optimizations like live range splitting, register coalescing, and rematerialization. Note that KEQ *needed no modifications* to accommodate the Register Allocation

phase. Instead, we were able to focus our efforts on the development of an inference algorithm for the verification condition generator. Our algorithm is *black-box*: it is capable of inferring the correspondence between virtual registers in the input code and physical registers and/or spill memory locations in the output code solely by analyzing the input and output programs, and requiring minimal assistance from the compiler itself [1]. Thus, we arrive to a highly practical compilation verification solution for the complex phase of Register Allocation of a modern optimizing compiler, as well as provide more evidence for the value of a modular TV System with a reusable equivalence checker.

We evaluate our TV System on the GCC benchmark of SPEC 2006 [26]. We are able to successfully validate 4574 out of the 4732 supported GCC functions (96.67%). The main reasons for the failures are discussed in Subsection 5.4.3, mainly inadequacies of our inference algorithm to handle specific edge cases as well as performance reasons related with symbolic execution. We also reintroduced two actual register allocation bugs to LLVM and verified that our system does not validate miscompilations due to said bugs, as shown in Subsection 5.4.4.

5.2 THE REUSABLE TRANSLATION VALIDATION COMPONENTS FOR REGISTER ALLOCATION

In this work, we take advantage of the modular TV system for modern compilers presented in the previous Chapter 4. Specifically, the system is based on a *language- and transformation-independent* equivalence checker, namely KEQ. This checker accepts a pair of programs, the input program and an output program that is the result of a (set of) transformations applied to the input. It is also parameterized by formal semantic definitions for the input and output languages. Finally, KEQ accepts a set of verification conditions in the form of synchronization points: pairs of symbolic state descriptions for the input/output programs that are known to be related along with a set of constrains that equate variables in the two programs.

An example of synchronization points for an input program and its corresponding output after register allocation is shown in Figures 5.1 and 5.2: the locations of those points are shown in the former and the equality constraints in the latter. For example, the synchronization point $p2$ is a pair of symbolic descriptions of the states of the input and output programs when execution is reaching basic block `.BB1` from block `.BB3`. These symbolic

[1] Currently we only require the compiler to provide the number of arguments in a callsite. The inference algorithm is then able to figure out the registers and/or stack slots that are used for argument passing from the calling convention.

```
1  main:
2  .BB0: ;p0
3     COPY %vr2, edi
4     movl %vr4, 0
5
6
7     jmp .BB1
8
9
10 .BB1: ;p1 p2
11    %vr0 = PHI (%vr4, .BB0), (%vr1, .BB3)
12
13    %vr5 = subl %vr0, 19
14
15
16    jg .BB4
17    jmp .BB2
18
19
20 .BB2: ;p3
21
22    %vr8 = addl %vr0, 1
23
24    movsxq %vr9, %vr0
25    movl [frm(0), 4, %vr9], %vr8
26    jmp .BB3
27
28
29 .BB3: ;p4
30
31    %vr1 = addl %vr0, 1
32
33    jmp .BB1
34
35
36 .BB4 : ;p5
37
38    movsxq %vr6, %vr2
39    movl %vr7, [frm(0), 4, %vr6]
40    COPY eax, %vr7
41    ;p6
42    retl eax
```

(a) *Before Register Allocation*

```
1  main:
2  .BB0: ;p0
3
4     movl eax, 0
5     movl [frm(1)], edi
6     movl [frm(2)], eax
7     jmp .BB1
8
9
10 .BB1: ;p1 p2
11    movl eax, [frm(2)]
12    COPY ecx, eax
13    ecx = subl ecx, 19
14    movl [frm(3)], eax
15    movl [frm(4)], ecx
16    jg .BB4
17    jmp .BB2
18
19
20 .BB2: ;p3
21    movl eax, [frm(3)]
22    eax = addl eax, 1
23    movl ecx, [frm(3)]
24    movsxq rdx, ecx
25    movl [frm(0), 4, rdx], eax
26    jmp .BB3
27
28
29 .BB3: ;p4
30    movl eax, [frm(3)]
31    eax = addl eax, 1
32    movl [frm(2)], eax
33    jmp .BB1
34
35
36 .BB4: ;p5
37    movl eax, [frm(1)]
38    movsxq rcx, eax
39    movl eax, [frm(0)], 4, rcx]
40
41    ;p6
42    retl eax
```

(b) *After Register Allocation*

Figure 5.1: A simple array accessing program in Virtual x86 before and after Register Allocation (with the greedy allocator of LLVM). Comments in red show the program points for which synchronization points are generated by our inference algorithm.

state descriptions contain symbolic variables for the live values at this point (e.g. for the value of virtual register `%vr1` in the input program). The point $p2$ also includes a set of equality constraints between such symbolic variables in the input and output. The pairs of concrete states that satisfy the equality constraints are considered to describe execution points that the two programs synchronize.

Given the input and output program along with a proposed set of synchronization points, and parameterized by the input and output language semantics, KEQ attempts to prove that there exists a bisimulation relation [22] between the two programs, the existence of which is enough to prove equivalence. The KEQ equivalence algorithm involves symbolic execution of both programs, and, in its core, it tries to answer a set of reachability queries: starting from each pair of synchronization points (that do not describe exiting states) can we reach, after a finite number of steps in each program, states that are also captured by a pair of synchronization points. If the answer is yes, then the proposed synchronization points are

Sync Point	*Prev BB*	*Equality Constraints*
$p0$	-	`frm(0) = frm(0), %edi = %edi`
$p1$	`.BB0`	`%vr2 = [frm(1)], %vr4 = [frm(2)], frm(0) = frm(0)`
$p2$	`.BB3`	`%vr2 = [frm(1)], %vr1 = [frm(2)], frm(0) = frm(0)`
$p3$	*any*	`%vr2 = [frm(1)], %vr0 = [frm(3)], frm(0) = frm(0)`
$p4$	*any*	`%vr2 = [frm(1)], %vr0 = [frm(3)], frm(0) = frm(0)`
$p5$	*any*	`%vr2 = [frm(1)], frm(0) = frm(0)`
$p6$	*any*	`%eax = %eax`

Figure 5.2: Equality constraints of synchronization points for the translation of the simple array accessing program in Figure 5.1.

in fact a bisimulation relation for the programs, and assuming that the proposed point do describe equivalent states, the programs are equivalent.

KEQ is implemented as a tool within the $\mathbb{K}$ framework [24]. $\mathbb{K}$ is language for defining operational semantics of programming languages. These semantic definitions are used to specialize a series of generic tools into tools for the defined language. KEQ expands on that principle as it can be parameterized by two (possibly different) languages.

5.2.1 KEQ in Instruction Selection and Virtual x86

KEQ has been successfully applied to the Instruction Selection phase of the LLVM compiler [19], that is the phase that translates LLVM IR [9] to Machine IR [21], a generic low level representation that can be parameterized with the specific opcodes of a target instruction set architecture and retains some higher-level features such as an infinite register file, phi instructions and SSA form, function signatures, etc. We call the Machine IR form of the x86-64 instruction set [18] *Virtual x86*. This is the output language for Instruction Selection and also the input and output language for the Register Allocation phase that is addressed in this work.

A feature of Virtual x86 that plays an important role in Register Allocation is *stack frames*. Virtual x86 functions can define any number of such stack frames, one frame for each stack slot in the function's call frame. These frames are typically utilized to allocate local objects with not known compile-time size. After Register Allocation they are also utilized as *spill locations*, stack slots used in addition to physical registers, when the latter are not enough

to store all live values at a given point in the function's code.

5.2.2 KEQ in Register Allocation

We are able to use both KEQ and the Virtual x86 semantic definition presented in 4.2.3 without any modifications for our TV prototype for the Register Allocation phase of the LLVM compiler [20]. The additional component required to arrive to a fully functional TV system is a custom synchronization point generator for Register Allocation. We discuss the generator in detail in Section 5.3. To familiarize the reader with the transformations of Register Allocation, here we give an example of a pair of input and output programs along with the desirable synchronization points that KEQ would use to prove equivalence.

Figure 5.1 shows a simple function that accesses a local array object, stored in `frm(0)` as an array of 20 32-bit integers. The function is shown in Virtual x86 before and after the Register Allocation phase. As expected, the input function uses virtual registers in SSA form. Physical registers are only used for parameter passing (line 3) and return values (lines 40 and 42). The function also uses some of the so-called pseudo-instructions of Virtual x86, namely `COPY` and `PHI`. The output function uses only physical registers and it is not in SSA form any more, the `PHI` instructions have been removed as well. In addition, more frames are used as spill locations.

In order to prove the two programs equivalent, we generate seven synchronization points, $p0$ to $p6$: the program locations for those points are shown in red in the code in Figure 5.1 while their equality constraints are shown in Figure 5.2. Given these synchronization points, KEQ is able to prove equivalence: Given any synchronization point other than the exiting point at $p6$, and after symbolic execution starting from the given point itself, KEQ can reach another synchronization point in a finite number of steps. For example, starting from $p1$ in both programs each program reaches either $p2$ or $p5$ after one loop iteration. Moreover, assuming the equality constraints for $p1$, KEQ can prove that after symbolic execution, either the equality constraints for $p2$ hold true (when assuming the path condition for reaching $p2$) or the equality constraints for $p5$ hold true (when assuming the path condition for reaching $p5$).

5.3 BLACK BOX SYNCHRONIZATION POINT GENERATOR FOR REGISTER ALLOCATION

In order to construct the set of synchronization points and their equality constraints as needed by KEQ, we design a generator that operates on the input and output programs

without any assistance from the compiler itself. In the following we present the synchronization point generator in more detail and discuss the equality constraint inference algorithm that is the basis of the generator.

5.3.1 Synchronization Point Placement

Here we discuss the various program points that are selected as synchronization points from the generator. Note that we break the problem of program equivalence into smaller problems of function equivalence for corresponding functions in the input and the output. By doing so, we require synchronization point sets for function pairs and we invoke KEQ for each function pair independently. We generate synchronization points for a given function pair assuming that calls to corresponding functions with equal arguments have equivalent effects to the caller function states.

As shown in Chapter 3, the synchronization points ought to form a *cut* for each program for the equivalence proof to be valid. A *cut* C in a program P is a set of program points in P, such that, there exists a constant bound, $B_{P,C}$ specific to P and C, so that in any execution of P, the number of consecutive instructions between two successive cut points is at most $B(P, C)$. One simple way to create a cut is to include the program points at every function entry, function exit, and at least one instruction in every cycle in the control flow graph of every function. We can then place synchronization points at corresponding pairs of cut points, i.e., every pair of function entries, pair of exits, and at least one pair of instructions in every cycle in the control flow graph.

For register allocation, we in fact generate more than the minimum points needed for a cut: we generate a synchronization point per start of corresponding basic blocks, plus pairs of return or other exiting instructions. A 1-1 basic block correspondence is trivial to infer since Register Allocation does not modify the control flow graph. The equality constraints for these synchronization points are inferred by the algorithm presented in Subsection 5.3.5. These constraints require that live virtual registers in the input are equal to the corresponding physical registers or spill frames in the output. The equality constraints for return or exit points can be generated based on the calling convention, with no need for inference.

Finally, we generate synchronization points before and after corresponding function calls, effectively treating function calls as exiting points. The equality constraints for these can also be generated based on the calling convention. Specifically for the points after a function call, we need to add equality constraints for the corresponding live registers and/or spill locations. We do that by using the inferred live pairs at the entries of successors of the blocks containing the call and propagating the liveness information backwards until we reach the instructions

after the call in each function.

It is important to note that we only need to trust that the generated synchronization points cover all the entry and exiting program points in the two functions and that the constraints of those points correctly reflect equivalent entry and exiting states. Other than that, we do not need to trust the placement and the constraints of the rest of the generated synchronization points. KEQ will not be able to successfully prove equivalence in presence of bogus synchronization points other than the trusted ones.

5.3.2 Spill Locations in Virtual x86

Before discussing the inference algorithm, we briefly discuss how the algorithm recognizes spill locations and spill/unspill instructions in the output of the Register Allocation phase. As we saw in Subsection 5.2.1, Virtual x86 uses frame identifiers to refer to the various stack slots accessed by the function. These are removed in the Prologue/Epilogue Insertion phase of the LLVM compiler backend, which follows the Register Allocation phase. Spill locations in the output are then such frame identifiers. Since frames in the input program are used for stack-allocated variables (implementing the LLVM `alloca` instruction), they are not reused in the output code. This enables us to use a simple technique to pinpoint the frames that correspond to spill locations: they are the extra frames generated after Register Allocation and not existing in the input function. In the example of Figure 5.1, the frames that correspond to spill locations are `frm(1)`, `frm(2)`, and `frm(3)`, while `frm(0)` appears in both the input and output and correspond to the stack slot holding the local array object.

Given the knowledge of the frames corresponding to spill locations, we identify spill and unspill instructions as memory stores and loads that access frames designated for spill locations. Again in the example of Figure 5.1, instructions in lines 5, 6, 14, 15, and 32 are spills, while instructions in lines 11, 21, 23, 30, and 37 are unspills.

5.3.3 Inference Algorithm Intuition

The desirable inference result for the synchronization point generator is, for each pair of corresponding basic blocks, a set of pairs (R_v, L), where R_v is a live virtual register in the input and L is either a physical register or stack frame in the output. To infer these pairs, we make the observation that the input and output programs for register allocation maintain the same opcode sequence, modulo various `COPY`, `PHI`, and spill/unspill instructions that are inserted or removed by the allocator. (Rematerialization violates this assumption and requires a minor extension to the algorithm, as explained in Section 5.3.7.) Moreover, these

same-opcode instructions have the same immediate (constant) operands and can only differ in their register/frame operands. Finally, the extra instructions are only copying their input to their output. Therefore, by ignoring the copying instructions (which do not necessarily have counterparts in the other program), we can use the correspondence of the rest of the instructions to infer pairs of live registers and/or frames.

The inference algorithm traverses the input and output functions matching such same-opcode instructions and collecting information about register and/or frame pairs by comparing their operands. These pairs must hold the same value for equivalence but they are not necessarily live at the entry of the block. Therefore, the algorithm propagates these pairs backwards to the entry of the block, updating their counterpart registers or frames as they flow through copying instructions and dropping pairs that are overwritten (killed) by non-copying instructions. The algorithm treats copying instructions in a special way: when a copying instruction kills a register/frame by defining it, the algorithm replaces this register/frame with the right-hand side of the copy and continues propagating the pair backwards to the entry of the block. The pair that finally reaches the entry of the block (if any pair reaches at all) is the live pair that is inferred by the algorithm. This way the algorithm takes into account the effect of copying instructions for the liveness analysis, but otherwise ignores them for the inference of corresponding register and/or frame pairs.

5.3.4 Inference Algorithm Assumptions

Based on the discussion above, here we list the assumptions made by our inference algorithm:

- There is a 1-1 correspondence of nodes (basic blocks) and edges in the CFGs of the input and output functions.
- Within corresponding basic blocks, there is the same sequence of instructions except specific copying instructions. Each pair of *matching instructions* has the same opcode and matching immediate and/or symbol operands and may only differ on register/stack frame operands.
- Unmatched *copying instructions* are allowed within otherwise matching instruction sequences. These instructions copy a register or frame operand to another register or frame operand. `PHI` instructions are also considered copying instruction.

These assumptions are satisfied by the LLVM Register Allocation phase, and are general enough for register allocators found in most modern optimizing compilers. Specifically, let

us comment on how spilling and SSA elimination, two transformations that are intrinsic parts of Register Allocation satisfy the above assumptions.

Spilling Register spilling only inserts copying instructions in the form of spills and unspills, i.e. stores and loads to designated stack slots. The inference algorithm is equipped to deal with register spilling as discussed above.

SSA Elimination After register allocation the code cannot be in SSA form since the physical register file is finite. So SSA elimination needs to happen during or before register allocation. The inference algorithm is equipped to deal with SSA elimination since it only removes and adds copying instructions (`PHI` and `COPY` instructions respectively), which are skipped by the algorithm during matching.

5.3.5 Inference Algorithm in Detail

Algorithms 5.1 to 5.3 present in pseudocode form the inference algorithm used to generate the equality constraints for the various synchronization points discussed in Subsection 5.3.1. The algorithm accepts the control flow graphs (CFGs) for the input and output functions and computes *live-in pairs*: pairs of registers and/or frames in the two programs that are live at the entry of corresponding basic blocks, and should hold equal values for equivalence. Note that we have designed the algorithm so that the assertions shown in the pseudocode should not fail for legal inputs, and the assertions are shown here for explanatory purposes by conveying key invariants of the algorithm.

Algorithm 5.1: Inference algorithm for Register Allocation.

```
1 Function main(F_pre, F_post):
2     LiveInPairs ← map();
3     MatchedInstrs ← set();
4     // Infer live-ins
5     for (BB_pre, BB_post) in blocks(F_pre, F_post) do
6         infer(BB_pre.start, BB_pre, BB_post.start, BB_post, LiveInPairs, MatchedInstrs);
7     // Propagate live-ins backwards in CFG
8     propagate(F_pre, F_post, LiveInPairs);
9     return LiveInPairs;
```

The entry point to the algorithm is function `main`, as shown in Algorithm 5.1. The algorithm operates in two phases. First, it infers corresponding register-register and/or

register-frame pairs by matching instructions with the same opcode and it propagates these pairs through copying instructions to the entry of their basic block (alg. 5.1, lines 5 to 6). These propagated pairs are the initial live-in pairs that feed the second phase, where live-in pairs are propagated backwards through predecessor blocks in the control flow graphs (alg. 5.1, line 8). Finally the algorithm returns a mapping, $LiveInPairs$ (line 9), from each basic block to a set of live-in pairs.

Algorithm 5.2: Inference phase.

```
1  Procedure infer(I_pre, BB_pre, I_post, BB_post, LiveInPairs, MatchedInstrs):
2      // Find next pair of non-copying
3      // instructions
4      I_pre ← nextNonCopyingInstr(I_pre, BB_pre);
5      I_post ← nextNonCopyingInstr(I_post, BB_post);
6      // Check for block end
7      if blockEnd(I_pre, BB_pre) then
8          assert blockEnd(I_post, BB_post);
9          return;
10     // Record match
11     assert match(I_pre, I_post);
12     insert(MatchedInstrs, (I_pre, I_post));
13     // Infer live-in pairs by
14     // following matching uses backwards
15     for (U_pre, U_post) in zip(uses(I_pre), uses(I_post)) do
16         L_pre ← findLiveinOrKill(U_pre, I_pre, BB_pre);
17         L_post ← findLiveinOrKill(U_post, I_post, BB_post);
18         if L_pre ∈ BB_pre and L_post ∈ BB_post then
19             // Uses not live,
20             // assert killing definitions have been matched
21             assert (L_pre, L_post) ∈ MatchedInstrs;
22             continue;
23         // Live-in pair found
24         assert L_pre ∉ BB_pre;
25         assert L_post ∉ BB_post;
26         insert(LiveInPairs[(BB_pre, BB_post)], (L_pre, L_post));
27     // Continue inference for next pair
28     // of matching instructions
29     infer(I_pre, BB_pre, I_post, BB_post, LiveInPairs, MatchedInstrs);
```

Inference Phase The inference phase is shown in Algorithm 5.2 for a pair of corresponding basic blocks. This phase scans the two basic blocks to find pairs of *matching instructions*, i.e. instructions with the same opcode and immediate arguments that are not `COPY`, `PHI`, or spill/unspill instructions (referred to as *copying instructions*). To do so, it skips any such copying instructions in the two blocks until it finds the next non-copying instruction pair (alg 5.2, lines 4 to 5). After a match is found in line 11, it is recorded in line 12. Note that the algorithm asserts that successive pairs of non-copying instructions should match. This is the case for every transformation related with Register Allocation, except Rematerialization. For more details on how we handle rematerialized instructions, see Subsection 5.3.7.

Algorithm 5.3: Backwards propagation of a use to find a live-in at the block entry. Returns either a live-in or a killing definition.

```
1  Function findLiveinOrKill(Use, I, BB):
2      I_prev ← previousInstr(I, BB);
3      if blockStart(I_prev, BB) then
4          // Block entry reached,
5          // use is live-in
6          return Use;
7      if not Use in defs(I_prev) then
8          // Def not found,
9          // continue search backwards
10         return findLiveinOrKill(Use, I_prev, BB);
11     if isCopying(I_prev) then
12         // Copying def found,
13         // search backwards with new use
14         return findLiveinOrKill(uses(I_prev), I_prev, BB);
15     // Non-copying def found,
16     // use is killed,
17     // return killing def
18     return I_prev;
```

When an instruction match is found, the algorithm attempts to find live-ins at the beginning of the blocks by matching the input operands (uses) of the two matched instructions (alg 5.2, line 15), and searching back to check that they are not killed by a previous definition within the current block. This is done by invoking the `findLiveinOrKill` function for each of the two matched uses (alg 5.2, lines 16 and 17). These calls return either a pair of live-ins corresponding to the matched uses, in which case the pair is recorded in the mapping (alg 5.2, line 26), or a pair of instructions with killing definition for the matched uses, in

which case the search for live-in pairs continues with the next pair of matched uses (alg 5.2, lines 18 to 22).

The `findLiveinOrKill` function, shown in Algorithm 5.3, recursively checks the previous instruction for a definition that matches the given use (alg 5.3, lines 2 and 7). If a definition is not found in the previous instruction the search continues backwards (alg 5.3, line 10). If the search ever reaches the start of the block, the the current use is a live-in and is returned (alg 5.3, lines 3 to 6). On the other hand, if a definition is found in the previous instruction, then the algorithm checks if the definition instruction is copying or not (alg. 5.3, line 11). In the former case, the search continues with a new use, the right-hand side of the copying instruction (alg. 5.3, line 14). In the latter case, the search stops since the non-copying definition has killed the current use, and the killing definition is returned (alg. 5.3, line 18).

Note that the actual algorithm requires a modification to correctly handle live-ins that have been discovered through a `PHI` copying instruction, as they are only live when the block is reached from a specific, as opposed to any, predecessor. See, for example, the live-in `%vr4` in line 11 of Figure 5.1(a): This live-in is only truly live at the entry of block `.BB1` when the block is reached from predecessor block `.BB0` and not when it is reached from its other predecessor block `.BB3`. We omit this complication in this presentation for clarity, although our prototype synchronization point generator correctly handles this case by recording the required predecessor block for such live-in pairs. Later, in synchronization point generation, we generate one synchronization point per predecessor if any of the inferred live-ins has additional predecessor information attached. That is the case for synchronization points $p1$ and $p2$ in Figure 5.2, both of which correspond to same program point (entry of basic block `.BB1`) but describe different program states where execution has reached `.BB1` from different predecessors. In the example of register `%vr4` above, this is only a live-in for synchronization point $p1$ and not for point $p2$.

Backwards Propagation Phase The backwards propagation phase is shown in Algorithm 5.4. It is a standard Live Variables backwards dataflow problem, but working simultaneously on two matching functions: In every propagation step, live-ins in the entry of each basic block are propagated to the entries of predecessor blocks (alg. 5.4, line 13) as long as the predecessor in question does not contain a definition for the live-in (alg. 5.4, lines 7, 8, and 9 to 11), effectively killing the live-in. Note that the algorithm requires both live-ins in a pair to be either propagated or both killed in the same way (alg. 5.4, line 12). Also note that backwards propagation goes through copying instructions by using `findLiveinOrKill` while propagating the live-ins to the entries of predecessor blocks. Backwards propagation steps are applied until no change is recorded in the latest step (alg. 5.4, lines 2, 14, and 16

Algorithm 5.4: Backwards propagation phase.

```
1  Procedure propagate(F_pre, F_post, LiveInPairs):
2      LiveInsUpdated ← false;
3      // Do one step of propagation
4      for (BB_pre, BB_post) in blocks(F_pre, F_post) do
5          for (BB'_pre, BB'_post) in preds(BB_pre, BB_post) do
6              for (L_pre, L_post) in LiveInPairs[(BB_pre, BB_post)] do
7                  L'_pre ← findLiveinOrKill(L_pre, BB'_pre.last, BB'_pre);
8                  L'_post ← findLiveinOrKill(L_post, BB'_post.last, BB'_post);
9                  if L'_pre ∈ BB'_pre and L'_post ∈ BB'_post then
10                     // Both live-ins killed in predecessor blocks
11                     continue;
12                 assert L'_pre ∉ BB'_pre and L'_post ∉ BB'_post;
13                 insert(LiveInPairs[(BB'_pre, BB'_post)], (L'_pre, L'_post));
14                 LiveInsUpdated ← true;
15     // If changes were made, do another step of propagation
16     if LiveInsUpdated then
17         propagate(F_pre, F_post, LiveInPairs);
```

to 17), thus allowing live-ins to propagate many blocks backwards as long as they remain not killed.

5.3.6 Example Application

As an example, let us see how the inference algorithm will arrive to the equality constraints for synchronization point $p3$ in Figure 5.2. The first non-copying matching instructions in the `.BB2` basic blocks are the `addl` instructions in line 22. From these, we get the pair of uses `(%vr0, eax)`. While propagating the pair backwards, we find the copying unspill instruction in line 21 that defines `eax`, thus we replace it with the right-hand side of the unspill, namely `[frm(3)]`. This happens in line 14 of Algorithm 5.3. Finally we arrive at the live-in pair `(%vr0, [frm(3)])`.

Next matching instructions are the `movsxq` ones in line 24. From these, we get the pair of uses `(%vr0, ecx)`. With a similar backward propagation of this pair through the unspill in line 23, we arrive at the same live-in pair `(%vr0, [frm(3)])` as before. Note that the other pair of operands, `(%vr9, rdx)`, are definitions and so we do not need to propagate them since we only need to determine the live-ins at the synchronization points, and the definitions only act as kills for later uses in determining the live-ins.

The final matching instructions with non-immediate opcodes are the `movl` ones in line 25. From these, we get three pairs of uses: `(frm(0), frm(0))`, `(%vr9, rdx)`, and `(%vr8, eax)`. The first pair propagates backwards to a live-in with no previous definitions in both programs. The second pair reaches a non-copying definition in line 24 for both programs: the `movsxq` instructions are definitions for the uses of `%vr9` and `rdx` that kill both of these uses. Thus, the algorithm does not generate a live-in pair from this pair of uses. Similarly, the final pair of uses also reaches a non-copying definition in line 22 for both programs: the `addl` instructions are definitions that kill those uses.

In order to arrive to the third live-in pair for $p3$, `(%vr2, [frm(1)])`, the algorithm needs to propagate that pair from block `.BB4` during the backwards propagation phase. The pair is inferred as a live-in pair for block `.BB4` in the inference phase and then gets successfully propagated through predecessor blocks `.BB1`, `.BB3`, and finally `.BB2`, since there are no definitions for `%vr2` or `[frm(1)]` in these blocks of the input and output functions, respectively.

5.3.7 Inference and Register Allocation Optimizations

Various optimizations happen as part of the Register Allocation phase in order to increase the quality of the output code, that is to minimize copies and spilling and to maximize the usage of the available register file. Here we discuss the effect of these transformation to the output code and how they affect the effectiveness of our inference algorithm. does not affect the assumptions made by the inference algorithm.

Register Coalescing [80] This transformation aims to minimize unnecessary copies introduced in the code after leaving SSA form. The transformation prioritizes assigning the same physical register to virtual registers that are the uses and definition of a `PHI` instruction, so that to eliminate the copying instructions that would be generated by removing the `PHI` instruction when leaving SSA form. This transformation does not introduce any non-matching, non-copying instructions, thus it does not affect the assumptions made by the inference algorithm.

Live Range Splitting [81] This transformation introduces copying instructions in order to reduce the number of different values that are live in any given point, thus reducing register pressure. The introduced copying instructions break long live ranges of the copied values, so that fewer values are live at the same time in various points of the code. Again this transformation only introduces copying instructions and as such it does not invalidate the assumptions of the inference algorithm.

Algorithm 5.5: Modification in Algorithm 5.2 for handling rematerializations

```
1 // Skip remats
2 while not blockEnd(I_post, BB_post and not match(I_pre, I_post) do
3     assert isRemat(I_post);
4     I_post ← nextNonCopyingInstr(I_post, BB_post);
```

Rematerialization [82] This is a variant of live range splitting, where a copy of an instruction that computes a value is inserted just before a use of that value, instead of reusing the register in which the value was saved earlier. This way the live range of the original register is shortened. This transformation has the potential to add non-copying, non-matching instructions to the output function. The inference algorithm takes its effects into account by skipping non-matching instructions in the output as long as it can recognize them as rematerializations. We do this by adding the code shown in Algorithm 1 between lines 5 and 7 of Algorithm 5.2. In our implementation of the inference algorithm, we have implemented the `isRemat` heuristic according to the possible rematerializable instructions of the X86 backend of the LLVM compiler. These are `mov` and `lea` variants that copy a constant value or compute a constant address (e.g. based on a global symbol plus constant offset) to a register.

5.3.8 Synchronization Point Generation

The synchronization point generator for our prototype TV system for Register Allocation combines all the aforementioned parts into a mostly black-box tool that operates on the input and output Virtual x86 functions and generates a proposed set of synchronization points that can be given to KEQ for equivalence checking. The generator first parses the two functions and constructs their CFGs along with the mapping between their corresponding basic blocks. It also computes which output function memory operations are spills and unspills as described in Subsection 5.3.2 by comparing the stack frame of the input and output functions. It finally constructs def/use information for all instructions in the two programs. To do this for call instructions, the calling convention is used to figure out which registers and/or frames are used for argument passing. At this point, the generator requires the number of the arguments on each callsite to be provided as a compiler hint.

All this information is then used by the inference algorithm presented in Subsection 5.3.5 to obtain a set of live-in register and/or frame pairs for each pair of corresponding basic blocks. The generator uses the live-ins to construct synchronization points at corresponding basic block entries with correct equality constraints. Another piece of information needed

for the synchronization point generation is the sizes of various virtual registers appearing in the input program, the sizes of physical x86 registers are apparent from their names. This is needed to correctly constraint symbolic variables assigned to the values of various live-in registers to their appropriate range. The virtual register size information is inferred by the generator by a separate pass based on the opcodes of the instructions in which they appear as operands. Finally, the generator constructs synchronization points before and after every function call and before every return instruction as discussed in Subsection 5.3.1.

5.4 EVALUATION OF THE TV SYSTEM FOR LLVM'S REGISTER ALLOCATION

We evaluate the Translation Validation prototype in three ways. First, we apply the prototype to some case study programs with interesting features, such as recursion, loop nesting, and potential non-termination. Second, we apply the prototype to a substantial subset of the source code of the GCC SPEC 2006 benchmark [26] to showcase its usefulness in a complex, real-world use case. Last, we reintroduce two register allocator bugs originally found in LLVM 3.5 and LLVM 7.0 and we verify that our prototype not only rejected translations affected by those bugs, but also produced enough information for a compiler developer to diagnose the locations and potential causes of both failures.

5.4.1 Methodology

All the target programs we tested were written in C. For each program, we compiled the source code into LLVM IR using `clang-5.0.2` at optimization level -O0, then translated to Virtual x86 using the default Instruction Selection (ISel) pass of LLVM 5.0.2. (Recall that this pass has previously been tested using the KEQ Translation Validation system for a number of programs (see Section 4.3.) This Virtual x86 forms the input code for our validation experiments. We then applied the default (fast) Register Allocation pass, to obtain the output code for our experiments.

5.4.2 Case Studies

We begin with several small case studies to evaluate the capabilities of the translation validation prototype. The TV prototype successfully proved that the register allocation pass was correct for all these programs, i.e., that the Virtual X86 before register allocation is semantically equivalent to that produced after register allocation. The specific programs we used for these case studies are as follows.

```
unsigned collatz(unsigned n) {
  unsigned c = 1;
  while (n != 1) {
    c = c + 1;
    if (n % 2 != 0) n = n * 3 + 1;
    else n = n / 2;
  }
  return c;
}
```

Figure 5.3: The Collatz conjecture testing function.

Fibonacci This program computes the n^{th} Fibonacci number: It is implemented recursively as a straightforward application of the mathematical definition (this is highly inefficient, but serves to demonstrate applicability to a recursive computation). Checking equivalence at function granularity allows our system to generate synchronization points that will cover all possible execution trace cycles due to recursion, and therefore avoid infinite loops of symbolic execution in the equivalence checking algorithm. Indeed, we handle function calls as exiting points (see Subsection 5.3.1) instead of symbolically executing function calls; those points serve to break cycles that would be created if we instead symbolically executed the recursive function calls.

Ackermann This program computes the value of the Ackermann function for given non-negative integers m and n, as follows:

$$Ack(0, n) = n + 1 \tag{5.1}$$

$$Ack(m > 0, 0) = Ack(m - 1, n) \tag{5.2}$$

$$Ack(m > 0, n > 0) = Ack(m - 1, Ack(m, n - 1)) \tag{5.3}$$

This program is also implemented as a recursive function and is handled similarly to the Fibonacci program.

Collatz Conjecture Test This program (Figure 5.3) tests the Collatz conjecture [83, 84] for a given positive integer. It is unknown whether this loop terminates for arbitrary n because it is an open mathematical problem to prove whether or not the sequence of values of n terminates at 1. This example shows experimentally that KEQ can handle potentially non-terminating programs. Non-termination is orthogonal to program equivalence, and our tool proves that, given the same input, the input and output programs either both terminate and return the same result or both run forever. Moreover, for every input, the proof implies

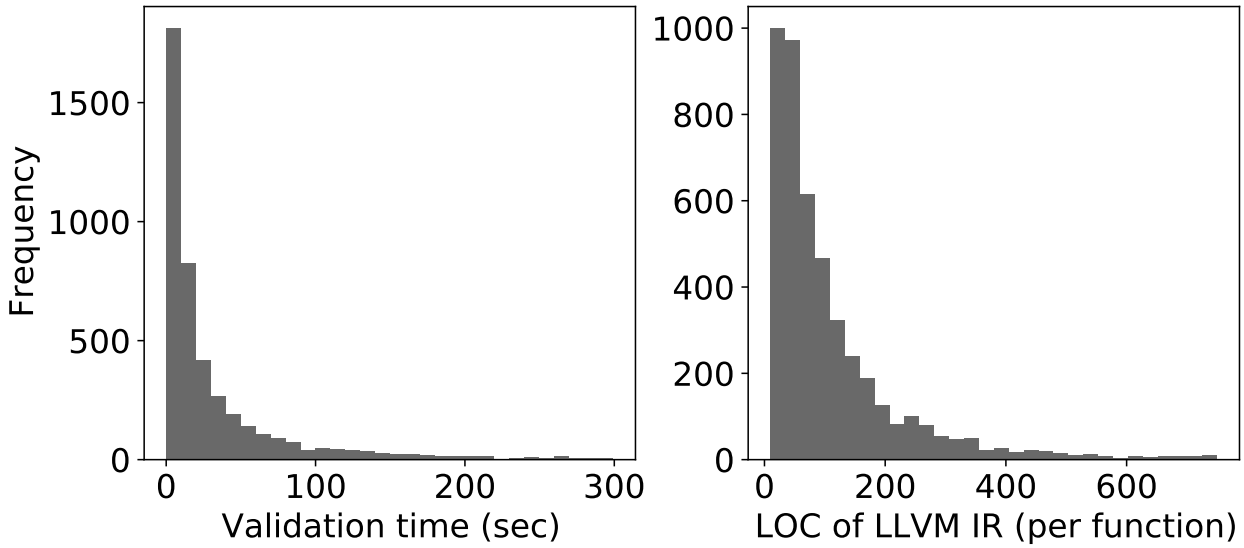


Figure 5.4: Distribution of validation time and LOC of LLVM IR of verified functions

that the input and output programs produce two (finite or infinite) sequences of identical states at the corresponding pairs of synchronization points.

BubbleSort It sorts an array of integers in place using the bubblesort algorithm. The array is stored in memory and passed to the function through a pointer. The proof demonstrates that the sequence of memory states at the corresponding pairs of synchronization points are identical for the input and output programs.

5.4.3 Application on the GCC SPEC 2006 Benchmark

In order to evaluate the Translation Validation prototype for a complex and important application, we applied it to a subset of the source code of the SPEC 2006 GCC benchmark. We compiled the GCC source code to LLVM IR, through instruction selection and register allocation, as described above, then tested our TV prototype on the resulting Virtual X86 output. For each verification run, we allocated 2 Intel Xeon CPU E7-8837 processors at 2.67GHz and 10GB of memory, with a timeout of 120 minutes.

Out of the 5572 functions in the benchmark, our evaluation considered 4732 functions that are covered by the fragment of Virtual x86 language semantics that we support (see Section 4.2.3). The remaining functions are unsupported because they involve floating-point operations, vectors, or dynamic allocas. Out of the 4732 supported functions, our prototype was able to formally verify the translation of 4574 functions (96.67%). Some statistics of the verified functions are shown in Figure 5.4.

The main reasons of failure for the remaining functions are as follows:

Limitations in the Inference Algorithm There were 42 (0.89%) functions that failed due to inadequacies in the synchronization point generator. One of the common issues is the inference of virtual register sizes. As mentioned in Subsection 5.3.5, our algorithm for this inference is based on instruction opcodes that unfortunately could not cannot infer an operand size for some operands of pseudo copy instructions just by looking at the opcode. A better register size inference algorithm is in our immediate future plans. Note that this limitation is completely unrelated to the main ideas in this paper, in particular, the dataflow based verification condition inference technique presented in Section 5.3.

Timeouts and Out-of-Memory 116 (2.5%) functions failed due to timeouts or out-of-memory. The primary cause, which accounts for 96 failures, is that the builtin $\mathbb{K}$ parser often does not scale well since it includes a very general ambiguity resolving mechanism. This is exacerbated by our inference algorithm's approach of generating a synchronization point for each basic block. To alleviate this issue, we can switch to a more compact representation of the synchronization point specification to reduce the work of the $\mathbb{K}$ parser. Other than the parser issue, there were 13 functions that timed out or ran out of memory during symbolic execution and 7 timed out in invocations of the SMT solver, Z3, due to the complexity of the constraint formulas.

5.4.4 Evaluation with Real-World LLVM Bugs

We tested KEQ on two bugs found in the fast register allocator in LLVM-7.0 and -3.5, with the former one actually present in the version of LLVM we are using for the GCC experiment (5.0.2).

The first bug [85], reported and fixed in LLVM-7.0 (but present in earlier versions), happens when an instruction defines (at least) two registers, with a virtual register definition appearing before a physical register definition in the operand list. In such a case, the buggy version of the allocator could potentially use the same physical register for both definitions. An example is shown in Figure 5.5. The instruction `mulxq` multiplies `rdx` with the third operand, and stores the high and low half of the result to the first and second operands, respectively. When the first two operands are the same, they will both contain the high half of the result. So in this case, it's incorrect to assign `rax` to `%vr1`. KEQ detected the problem during symbolic execution of the two versions of code, because the equality constraint for `rax` fails to match between the two versions at the point just before the `retq` instruction.

```
.BB0:                          .BB0:
movabsq %vr0,1                 movabsq rax,1
movabsq rdx,1                  movabsq rdx,1
                               movq [frm(1)],rax
mulxq %vr1,rax,%vr0            mulxq rax,rax,rax
retq rax                       retq rax
```

Before Register Allocation (returns 1) | *After bogus Register Allocation* (returns 0)

Figure 5.5: LLVM Bug 41790 [85]

Besides detecting the bug, this information would help guide a compiler developer debugging the problem by pinpointing the program interval in which the faulty allocation occurs.

The second bug [86], reported and fixed in LLVM-3.5, causes incorrect computation of live ranges of physical registers in the input code when the physical registers appear as implicit definitions. Various x86 instructions have implicit operands that do not appear in the instruction's syntax but are well-defined in its semantics. For example, the unsigned division instruction, `div`, accepts only one explicit operand, the register containing the divisor, and uses `edx` as both an implicit use operand, containing the dividend, and an implicit definition that will hold the result of the division. When the bug is present, the allocator incorrectly assumes that physical registers that have been implicitly defined are available when in fact their implicit definition is live. Our inference algorithm detects the bug during the inference phase of the verification condition generator (see Algorithm 5.2) when it attempts to verify that two matching uses reach matching definitions while being propagated backwards within their basic blocks (line 21). Indeed, the definition found for the input program is the implicit definition, while the definition found for the output of the buggy register allocator is an incorrect definition (after the implicit one) that was introduced because the allocator assumed that the register was free to use. Again, this information not only detects the bug but is also sufficient to tell the developer which operands were involved, and that the missed definition in the input program is an implicit definition.

CHAPTER 6: LESSONS LEARNED AND FUTURE WORK

In this Chapter, we briefly discuss important lessons we learned from this work and propose future work towards the goal of a modular transformation-agnostic TV system for the full LLVM compiler. The following sections discuss three extensions to the existing prototype that aim to either increase its coverage of the compilation path or enhance the reusability of its components across different transformations.

6.1 LESSONS ABOUT DESIGNING A TRANSLATION VALIDATION SYSTEM FOR MODERN COMPILERS

Let us discuss some useful insights we gained by engaging with the problem of Translation Validation for modern compilers.

Hints and Heuristics We have used both approaches for verification condition generation: The Instruction Selection generator is based on compiler-generated hints while the Register Allocation generator employs a novel heuristic. In our experience, the hint-based generation is more reliable and easier to implement than designing and testing heuristics. Although it is appealing to have a (heuristic-based) design that requires zero information from the compiler, we believe that accuracy loss and increased development effort make the trade-off not favorable. The heuristic-based design becomes more attractive for systems designed to handle a large set of transformations in one tool, since collecting hints from all transformations involved and combining them to obtain useful information becomes more complex with the increasing number of transformations. However, our modular design allows for separate handling of transformations without significant increase in development effort thus minimizing the need for heuristics. On the other hand, generation of synchronization points based on compiler-generated hints removes any guesswork out of the design and is more straight-forward to implement. We need of course to stress the fact that compiler-based hints generation should keep all formal methods related work (e.g. synchronization point placement, collection of constraints, etc.) within the verification condition generator and assign only compiler-related work to the hint generator (e.g. information about variable correspondence, branch and other control flow modifications, etc.). This is so compiler engineers, who would maintain the hint generator as part of the compiler code base, are not required to have any formal methods expertise to do so.

Complex Optimizations with Simple Visible Effects Many classic compiler optimizations use intricate algorithms and have complex implementation but their effect on the output is rather simple. Instruction Selection and Register Allocation are two such examples as are peephole optimizations, dead code elimination optimizations, various loop related optimizations and more. The complexity in algorithm and implementation arises from the fact that these optimizations may require complex static analyses to discover optimization opportunities, use hard-coded semantics about the input (and output) language(s), and/or employ various heuristics. However, Translation Validation approaches are oblivious to this internal complexity and are only interested in the effect on the output code. For example, an advanced pointer analysis may be required for a dead code elimination transformation, but a translation validation system may be interested only in whether a branch in the control flow graph was removed as a result and in that case what was the specific alias pair that enabled the code elimination. The details of the pointer analysis used by the compiler need not to be communicated to the translation validation system. On the other hand, the information that the control flow graphs differ in a certain way as well as the alias pair responsible are important for the verification condition generator to pick sufficient synchronization points with sufficient side conditions. This information can be hinted by the compiler (in which case, the system trusts the compiler's pointer analysis implementation) or it can be inferred by an external pointer analysis . Similarly, complex (and typically hard-coded) semantics information is needed by Instruction Selection to efficiently translate patterns of input language instructions, but a translation validation system is only interested in the correspondence between input/output variables and other names. Again the correctness of the translation is to be determined through validation using trusted formal semantics, but the naming correspondence is necessary to inform generation of sufficient verification conditions.

This observation reinforces our thesis for a modular Translation Validation system: First, by minimizing the work needed to address a new optimization, it is now practical to handle optimizations individually thus taking advantage of their simplicity in terms of visible effects. This simplicity is lost when one attempts to handle a large set of optimizations in one setting. Second, we take advantage of said simplicity by having accurate verification condition generators based on hint generators that can be realistically implemented by compiler engineers. Third, we delegate a large factor of compiler complexity, the hard-coded formal semantics for involved languages, to separate, reusable artifacts.

On a similar note, we need to emphasize that the granularity of the translation validation system needs to be sufficient to capture all the information that was available to the compiler at the time it performed the transformation. More specifically, a translation validation system that operates on function-level granularity, i.e. it proves code equivalence

in a function-by-function basis, is likely to have an increased rate of proof failures when it targets an inter-procedural transformation, i.e. a transformation that leverages calling context information to discover transformation opportunities in a given function. For example, an inter-procedural constant propagation transformation may discover opportunities for constant propagation within a function's body by taking into account how the data-flow from the various call sites restricts the possible values of that function's parameters. In this case, a translation validation system that operates on function-level granularity will not be able to prove equivalence because it does not have access to the function's call sites and hence it cannot arrive to the same restrictions under which the transformation is correct. We either need to expand the system's granularity so that the necessary context is part of the code checked for equivalence (that is the function's call sites in this example), or we need the compiler to communicate said restrictions, thus making the underlying static analysis of the compiler part of the trusted code base of our translation validation system. The former approach has the drawback of expanding the amount of code that the system needs to symbolically execute and potentially reducing its scalability. The latter approach increases the trusted code base of the system, but it has similar benefits with the hint-based approach discussed above: increased accuracy and easier implementation. One can even use an external to the compiler static analysis (that may come with its own correctness guarantees) to avoid trusting the compiler for analysis results.

Support from Compilers The work presented here aims to the end goal of having a Translation Validation system that can be plugged in an existing modern compiler and automatically provide formal correctness guarantees to the compiler's users. This goal cannot be achieved without cooperation with the compiler community. We have identified two major areas where compiler support can make a difference.

First, as has been discussed before, compiler engineers should be expected to work in tandem with the formal methods experts to provide needed information of transformation passes. Ideally the two teams should agree on an interface for compiler-generated hints. Such hints should not require any formal methods expertise but are crucial for the verification condition generator because they both decrease the false alarm rate (due to more accurate verification conditions than ones generated by a heuristic) and increase the performance (no need to run potentially expensive heuristic-based inference algorithms).

Second, intermediate representation languages developed for internal compiler use should nevertheless come with a well-documented, preferably formal, semantics definition. This is necessary for any serious effort on compilation verification since it removes any subjectivity of what a program written in such a language means. Without such formal definitions, compiler

verification can adopt only imperfect approaches of either interpreting ambiguity in the semantics in an arbitrary way or being forced to accommodate end-to-end compilation where input/output language semantics are known. The former approach is clearly undesirable, even more so because there may be no single correct interpretation of ambiguous semantics that fits all compiler transformations; see [48] for an example of these in LLVM IR. The latter approach is also far from ideal since there may be significant sacrifice in accuracy when attempting end-to-end efforts sch as this. Moreover, all known to us systems that validate end-to-end compilation employ a custom common intermediate representation and custom-made translators to that representation.

6.2 TRANSLATION VALIDATION FOR THE WHOLE LLVM TO X86-64 COMPILATION PATH

Extending the TV system to accommodate the whole LLVM to x86-64 compilation path of the LLVM compiler amounts to validating the various transformation passes that precede the Instruction Selection phase or follow the Register Allocation phase. Since we are going to reuse cut-bisimulation and KEQ as the equivalence notion and equivalence checker for the rest of the passes, the only part of the system that needs to be extended is the verification condition generators.

There are two trade-offs that need to be evaluated for the design of said generators. First, there is a trade-off between the amount and complexity of compiler-generated hints and inference heuristics. We want to minimize the amount of hints that are required from the compiler but we also do not want to increase the number of false reports due to inadequate heuristics that cannot generate verification conditions strong enough to prove equivalence. Second, there is a trade-off between grouping many optimizations and designing a strong generator that provides adequate verification conditions to prove equivalence. Our modular design that clearly separates the language semantics and the proof system from the generators allows for easier experimentation to reach an optimal design.

Finally, since the output language of the backend is the x86-64 ISA, there is need for a $\mathbb{K}$ definition of x86-64 to appropriately parameterize KEQ for the equivalence proof of the last passes that involve x86-64, rather than the internal x86-64 Machine IR. For this purpose, we could look towards a recently published such definition that is the most complete x86-64 formal definition to date [87].

6.3 TOWARDS TRANSFORMATION-AGNOSTIC INFERENCE OF SYNCHRONIZATION POINTS

Currently the verification condition generator is the only component of the proposed design that is transformation-specific. However there is existing work on generating proof obligations for equivalence checking in a transformation-agnostic but language-aware manner. Necula *et al.* [15] describe a such a proof obligation inference algorithm based on symbolic summaries of the effects of basic blocks of the input and output programs. Churchill *et al.* [33] in their recent work on data-driven equivalence checking develop a semantics-aware algorithm that employs data from execution traces to infer proof obligations. A future direction of this dissertation would be the evaluation of the applicability of these algorithms for the proof generators of the our proposed design for translation validation systems. Also, we could explore the feasibility of extending such algorithms to be parametric to the semantics definitions of the input and output languages in a fashion similar to KEQ, thus working towards language-independent verification condition generators that can be used for passes such as Instruction Selection.

Expanding on the idea of transformation-agnostic verification condition generation, we want to explore a machine learning approach on synchronization point inference. There is existing work on machine-learning assisted decompilation [88, 89, 90] and binary analysis [91, 92], but it involves supervised learning techniques, where the model in question is learned using a training set of input/output pairs. For example in [88], a corpus of C source code snippets and their corresponding binaries are used to train the proposed decompilation model. In the case of our interest, where we want to learn state mappings between the input and output programs (e.g., register and stack frame location mappings in the input and output of the register allocator), acquiring such a training corpus is not trivial. Therefore, we could look into reinforcement learning [93] instead, where the model is learned via trying to maximize a reward function, while searching the space of possible state mappings. Concrete execution traces could be used to arrive to a reward function, which assigns a score to a given state mapping based on how many mapped locations actually hold equal values in a set of concrete execution traces.

6.4 USING TRANSLATION VALIDATION TO DISCOVER COMPILER BUGS

In this dissertation, we propose a design for a sound but incomplete system for translation validation. This means that our system is not allowed to validate incorrect translations but it could flag a correct translation as invalid. This design is aimed towards compiler

users that want correctness guarantees for their compiled applications rather than towards compiler engineers looking for debugging tools for the compiler itself. The main reason for the latter is the fact that false reports that do not originate to an actual miscompilation are indistinguishable from the useful (for compiler debugging purposes) reports that do originate in non-equivalent input and output thus indicating a miscompilation.

An apparent future direction of the work is aiding the user of the proof system in their understanding of why a proof failed. This includes generating clear error messages when the system fails to validate a translation as well as the generation of a counterexample input for the input and output programs that triggers the difference in behavior. Specifically the latter provides a straightforward way for the user to evaluate the report. The counterexample would either make a miscompilation apparent or it could point towards a possible internal error of the proof system itself (e.g., an error in the language semantic definition). Generation of counterexamples requires exploration through symbolic execution similar to white- or black-box fuzzers such as Klee [94] and Sage [95]. In our case, the symbolic exploration should target synchronization points responsible for the proof failure, that is points for which KEQ could not prove that if used as starting points, another synchronization point could be reached. These are the points that are most likely to lead to a successful counterexample generation, although that is not guaranteed. Finally, rather than using an existing fuzzer that works for a specific language, a language-parametric fuzzer could be implemented as a $\mathbb{K}$ framework tool, similarly to KEQ.

CHAPTER 7: CONCLUSION

Compilation verification refers to techniques that provide a formal correctness guarantee for the compilation process. It is a necessary step for any system that provides formal guarantees so that those guarantees can be transferred to the compiled binary. More broadly, it assists software development by uncovering compilation bugs and it may contribute to debugging the compiler itself.

Compilation verification for existing compilers such LLVM is challenging due to the size and complexity of the compiler: a typical compilation involves many intermediate languages and transformations. Moreover, a compilation verification system cannot be developed by the same compiler engineers but rather by formal methods experts. Finally, the compilation verification system needs to be less complex than the compiler itself to make sense for it to be trusted.

In this work, we show that it is possible to develop a (mostly) language-independent, transformation-agnostic compilation verification system with support for different input/output languages for an optimizing, production-quality compiler.

Our design is based on Translation Validation, a process that verifies isolated instances of compilation by examining the input and output programs and trating the compiler mostly as a black box. Unlike traditional translation validation systems for compilers that are custom-tailored for a specific (set of) transformation(s) and language(s), our design includes a proof system that can be reused across the various transformations and languages of the compilation path. The transformation-specific logic is strictly limited in a set of custom verification condition generators, one per (set of) transformation(s) at hand.

First, we present a novel algorithmic approach for proving cross-language program equivalence. Our algorithm relies on *cut-bisimulation*, a general formalization of weak bisimulation realtions, that are better suited to equivalence proofs when the two programs may have unrelatable intermediate states. Our equivalence checking algorithm enables the reuse of a significant part of any Translation Validation system and allows developers to focus on the important aspects of the problem, which are the semantic definitions of the input and output languages and the generation of the proof obligations. Cut-bisimulation allows for generation of proof obligations that take the intuitive (from a compiler's perspective) form of synchronization points between the input and output programs. We have used this algorithm to develop KEQ, the first *language-independent* tool for program equivalence. KEQ accepts a set of synchronization points as well as formal semantics for the input and output languages (that are allowed to be different). It attempts to prove that the given synchronization points

are a cut-bisimulation relation, thus showing equivalence.

To showcase the modularity of our design, we use KEQ as the proof system of two different translation validation prototypes for the Instruction Selection and Register Allocation transformation of the LLVM compiler.

Instruction Selection is a major translation phase within LLVM that transforms LLVM IR into Machine IR, specifically x86-64 Machine IR for the context of this work. As such, this transformation has never been handled by translation validation systems in the literature in a straightforward way. Instead, the predominant method of dealing with different input/output languages in the literature is converting both programs into a third common representation. For Instruction Selection, we implement a verification condition generator that uses compiler-generated hints for the correspondence between temporaries in the input and output. The code added to the compiler for hint generation is minimal (roughly 500 LOC compared to the 140,000 LOC size Instruction Selection) and more importantly, it requires no formal method expertise and thus can be realistically be maintained by compiler engineers.

Register Allocation is an important back end pass that replaces the multiple temporary variables in the input program with a finite set of physical registers in the output. While the transformation's effect in the program code is simple, the algorithms employed by the pass are complicated, to the point that CompCert, a verified compiler, had to use an internal translation validation subsystem for register allocation because verifying the implementation was deemed not worth the effort. For Register Allocation, we implement a verification condition generator that employs a novel inference algorithm that infers the correspondence between temporaries in the input and physical registers and/or stack slots in the output.

Both prototypes use KEQ as the equivalence proof system that accepts a set of synchronization points generated by the respective verification condition generator. We test our prototypes by applying them to the translation of the SPEC 2006 GCC benchmark where we successfully prove the compilation of 91.52% and 96.67% functions with supported features respectively. Both prototypes use the same formal semantics definitions for LLVM IR and x86-64 Machine IR that are their own separate artifacts.

REFERENCES

[1] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[2] GCC, "GNU Compiler Collection," https://gcc.gnu.org, 2020, accessed: April 13, 2021.

[3] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931074 pp. 294–305.

[4] A. Corp., "iOS App Distribution Guide," https://developer.apple.com/library/etc/redirect/DTS/iOSAppDistGuide, 2019, accessed: February 2019.

[5] J. Horwitz, "Apple Watch apps instantly went 64-bit thanks to obscure Bitcode option," *VentureBeat*, 2018.

[6] D. Wheeler, "To bitcode, or not to bitcode?" *iovation*, 2015.

[7] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "Sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1629575.1629596 p. 207–220.

[8] T. A. L. Sewell, M. O. Myreen, and G. Klein, "Translation validation for a verified os kernel," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462183 pp. 471–482.

[9] LLVM, "LLVM Language Reference Manual," http://llvm.org/docs/LangRef.html, 2020, accessed: April 13, 2021.

[10] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, July 2009. [Online]. Available: http://doi.acm.org/10.1145/1538788.1538814

[11] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "Cakeml: A verified implementation of ml," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535841 pp. 179–191.

[12] A. Chlipala, "A certified type-preserving compiler from lambda calculus to assembly language," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250742 pp. 54–65.

[13] S. Rideau and X. Leroy, "Validating register allocation and spilling," in *Compiler Construction*, R. Gupta, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 224–243.

[14] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, ser. TACAS '98. London, UK, UK: Springer-Verlag, 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=646482.691453 pp. 151–166.

[15] G. C. Necula, "Translation validation for an optimizing compiler," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI '00. New York, NY, USA: ACM, 2000. [Online]. Available: http://doi.acm.org/10.1145/349299.349314 pp. 83–94.

[16] J.-B. Tristan, P. Govereau, and G. Morrisett, "Evaluating value-graph translation validation for llvm," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993533 pp. 295–305.

[17] F. S. Foundation, "GNU Compiler Collection Internals," %https://gcc.gnu.org/onlinedocs/gccint/Passes.html, 2019, accessed: August 2020.

[18] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, December 2016.

[19] LLVM, "LLVM Target-Independent Code Generation: Instruction Selection," http://llvm.org/docs/CodeGenerator.html\#instruction-selection-section, 2020, accessed: April 13, 2021.

[20] LLVM, "LLVM Target-Independent Code Generation: Register Allocator," http://llvm.org/docs/CodeGenerator.html#register-allocator, 2020, accessed: April 13, 2021.

[21] LLVM, "LLVM Target-independent Code Generator," http://llvm.org/docs/CodeGenerator.html#machine-code-representation, 2020, accessed: April 13, 2021.

[22] D. Sangiorgi, *Introduction to Bisimulation and Coinduction*. New York, NY, USA: Cambridge University Press, 2011.

[23] K. S. Namjoshi and L. D. Zuck, *Witnessing Program Transformations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 304–323. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38856-9_17

[24] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[25] M. Dahiya and S. Bansal, "Black-box equivalence checking across compiler optimizations," in *Programming Languages and Systems*, B.-Y. E. Chang, Ed. Cham: Springer International Publishing, 2017, pp. 127–147.

[26] SPEC, "SPEC CPU 2006 Benchmark," https://www.spec.org/cpu2006/, 2020, accessed: April 13, 2021.

[27] H. Samet, "Automatically proving the correctness of translations involving optimized code." Ph.D. dissertation, Stanford, CA, USA, 1975, aAI7525601.

[28] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg, "Voc: A methodology for the translation validation of optimizing compilers," *Journal of Universal Computer Science*, vol. 9, p. 2003, 2003.

[29] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1480881.1480915 pp. 264–276.

[30] S. Kundu, Z. Tatlock, and S. Lerner, "Proving optimizations correct using parameterized program equivalence," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542513 pp. 327–337.

[31] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, "Data-driven equivalence checking," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509509 pp. 391–406.

[32] C. Hawblitzel, S. K. Lahiri, K. Pawar, H. Hashmi, S. Gokbulut, L. Fernando, D. Detlefs, and S. Wadsworth, "Will you still compile me tomorrow? static cross-version compiler validation," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491442 pp. 191–201.

[33] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic Program Alignment for Equivalence Checking," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, June 2019.

[34] K. S. Namjoshi, *A simple characterization of stuttering bisimulation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 284–296. [Online]. Available: http://dx.doi.org/10.1007/BFb0058037

[35] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis, "The marriage of bisimulations and kripke logical relations," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103666 pp. 59–72.

[36] S. Dasgupta, S. Dinesh, D. Venkatesh, V. S. Adve, and C. W. Fletcher, "Scalable validation of binary lifters," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3385412.3385964 p. 655–671.

[37] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387. [Online]. Available: http://dx.doi.org/10.1007/11804192_17

[38] Ş. Ciobâcă, D. Lucanu, V. Rusu, and G. Roşu, *A Language-Independent Proof System for Mutual Program Equivalence*. Cham: Springer International Publishing, 2014, pp. 75–90. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11737-9_6

[39] Y. Huang, B. R. Childers, and M. L. Soffa, "Catching and identifying bugs in register allocation," in *Static Analysis*, K. Yi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 281–300.

[40] V. K. Nandivada, F. M. Q. Pereira, and J. Palsberg, "A framework for end-to-end verification and evaluation of register allocators," in *Static Analysis*, H. R. Nielson and G. Filé, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 153–169.

[41] G. Klein and T. Nipkow, "A machine-checked model for a java-like language, virtual machine, and compiler," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 4, p. 619–695, July 2006. [Online]. Available: https://doi.org/10.1145/1146809.1146811

[42] A. Lochbihler, "Verifying a compiler for java threads," in *Proceedings of the 19th European Conference on Programming Languages and Systems*, ser. ESOP'10. Berlin, Heidelberg: Springer-Verlag, 2010. [Online]. Available: https://doi.org/10.1007/978-3-642-11957-6_23 p. 427–447.

[43] Y. Bertot and P. Casteran, *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.

[44] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formal verification of ssa-based optimizations for llvm," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462164 pp. 175–186.

[45] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the llvm intermediate representation for verified program transformations," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2103656.2103709 p. 427–440.

[46] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, "Provably correct peephole optimizations with alive," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737965 pp. 22–32.

[47] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman, "Verified peephole optimizations for compcert," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2908080.2908109 p. 448–461.

[48] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes, "Taming undefined behavior in llvm," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3062341.3062343 p. 633–647.

[49] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen, "Proving correctness of compiler optimizations by temporal logic," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '02. New York, NY, USA: Association for Computing Machinery, 2002. [Online]. Available: https://doi.org/10.1145/503272.503299 p. 283–294.

[50] S. Kalvala, R. Warburton, and D. Lacey, "Program transformations using temporal logic side conditions," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 4, May 2009. [Online]. Available: https://doi.org/10.1145/1516507.1516509

[51] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[52] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, ser. SFCS '77. USA: IEEE Computer Society, 1977. [Online]. Available: https://doi.org/10.1109/SFCS.1977.32 p. 46–57.

[53] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, p. 244–263, Apr. 1986. [Online]. Available: https://doi.org/10.1145/5397.5399

[54] W. Mansky and E. Gunter, "A framework for formal verification of compiler optimizations," in *Proceedings of the First International Conference on Interactive Theorem Proving*, ser. ITP'10. Berlin, Heidelberg: Springer-Verlag, 2010. [Online]. Available: https://doi.org/10.1007/978-3-642-14052-5_26 p. 371–386.

[55] L. C. Paulson, "Isabelle: The next 700 theorem provers," *CoRR*, vol. cs.LO/9301106, 1993. [Online]. Available: https://arxiv.org/abs/cs/9301106

[56] W. Mansky, "Specifying and verifying program transformations with ptrans," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2014. [Online]. Available: http://hdl.handle.net/2142/49385

[57] G. Rosu, A. Stefanescu, S. Ciobaca, and B. M. Moore, "One-path reachability logic," in *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*. IEEE, June 2013, pp. 358–367.

[58] C. Ellison and G. Rosu, "An executable formal semantics of c with applications," in *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. ACM, January 2012, pp. 533–544.

[59] C. Hathhorn, C. Ellison, and G. Rosu, "Defining the undefinedness of c," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, June 2015, pp. 336–345.

[60] D. Bogdanas and G. Rosu, "K-Java: A Complete Semantics of Java," in *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, January 2015, pp. 445–456.

[61] D. Park, A. Stefanescu, and G. Rosu, "KJS: A complete formal semantics of JavaScript," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, June 2015, pp. 346–356.

[62] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, B. Moore, Y. Zhang, D. Park, and G. Roşu, "Kevm: A complete semantics of the ethereum virtual machine," in *Computer Security Foundations Symposium*, 2018.

[63] V. Buterin and Ethereum Foundation, "Ethereum White Paper," https://github.com/ethereum/wiki/wiki/White-Paper, 2013.

[64] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[65] R. Atkey, "Coqjvm: An executable specification of the java virtual machine using dependent types," in *Proceedings of the 2007 International Conference on Types for Proofs and Programs*, ser. TYPES'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 18–32.

[66] D. Park, "Semantics-based program verification," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2019. [Online]. Available: http://hdl.handle.net/2142/104795

[67] J. F. Groote, D. N. Jansen, J. J. A. Keiren, and A. J. Wijs, "An $O(m\log n)$ algorithm for computing stuttering equivalence and branching bisimulation," *ACM Trans. Comput. Logic*, vol. 18, no. 2, pp. 13:1–13:34, June 2017. [Online]. Available: http://doi.acm.org/10.1145/3060140

[68] J. R. Büchi, "On a Decision Method in Restricted Second-Order Arithmetic," in *International Congress on Logic, Methodology, and Philosophy of Science*. Stanford University Press, 1962, pp. 1–11.

[69] M. C. Browne, E. M. Clarke, and O. Grümberg, "Characterizing finite kripke structures in propositional temporal logic," *Theor. Comput. Sci.*, vol. 59, no. 1-2, pp. 115–131, July 1988. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(88)90098-9

[70] H. Rogers, Jr., *Theory of Recursive Functions and Effective Computability*. Cambridge, MA, USA: MIT Press, 1987.

[71] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS'08*, ser. LNCS, vol. 4963, 2008, pp. 337–340.

[72] Clang, "Clang: C Language Family Frontend for LLVM," http://clang.llvm.org, 2020, accessed: April 13, 2021.

[73] Swift, "Swift programming language," https://swift.org, 2020, accessed: April 13, 2021.

[74] Julia, "The Julia Language," http://julialang.org, 2020, accessed: April 13, 2021.

[75] LLVM, "TableGen," http://llvm.org/docs/TableGen/index.html, 2020, accessed: April 13, 2021.

[76] "Instruction Selection WAW miscompilation," https://bugs.llvm.org/show_bug.cgi?id=25154, 2015.

[77] "Instruction Selection load narrowing miscompilation," https://bugs.llvm.org/show_bug.cgi?id=4737, 2015.

[78] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, no. 1, pp. 47 – 57, 1981. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0096055181900485

[79] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN '82. New York, NY, USA: Association for Computing Machinery, 1982. [Online]. Available: https://doi.org/10.1145/800230.806984 p. 98–105.

[80] L. George and A. W. Appel, "Iterated register coalescing," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 3, p. 300–324, May 1996. [Online]. Available: https://doi.org/10.1145/229542.229546

[81] K. D. Cooper and L. Taylor Simpson, "Live range splitting in a graph coloring register allocator," in *Compiler Construction*, K. Koskimies, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 174–187.

[82] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, p. 428–455, May 1994. [Online]. Available: https://doi.org/10.1145/177492.177575

[83] R. K. Guy, *Unsolved problems in number theory. 3rd ed.*, 3rd ed. New York, NY: Springer-Verlag, 2004.

[84] J. C. Lagarias, "The Ultimate Challenge: The 3x+1 Problem," https://www.maa.org/press/maa-reviews/the-ultimate-challenge-the-3x1-problem, 2010, accessed: April 13, 2021.

[85] "When same reg is used for output and implicit-def, spill is inserted and overwrites output result," https://bugs.llvm.org/show_bug.cgi?id=41790, 2019.

[86] "Register Allocation implicit definition miscompilation," https://https://bugs.llvm.org/show_bug.cgi?id=21700, 2014.

[87] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, "A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, June 2019.

[88] E. Schulte, J. Ruchti, M. Noonan, D. Ciarletta, and A. Loginov, "Evolving exact decompilation," in *Binary Analysis Research*, ser. BAR '18, 2018.

[89] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 346–356, 2018.

[90] N. Hasabnis and R. Sekar, "Lifting assembly to intermediate representation: A novel approach leveraging compilers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872380 pp. 311–324.

[91] A. Jaffe, J. Lacomis, E. J. Schwartz, C. L. Goues, and B. Vasilescu, "Meaningful variable names for decompiled code: A machine translation approach," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196330 pp. 20–30.

[92] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt, "Learning to analyze binary computer code," in *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, ser. AAAI'08. AAAI Press, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1620163.1620196 pp. 798–804.

[93] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[94] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[95] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 122–131.