Semantics-Based Program Verification
Thesis Proposal

Daejun Park

University of Illinois

1 Introduction

Most of the existing formal program analysis tools (e.g., program verifiers) are not scalable in terms of flexibility and reusability. They were developed for a fixed target language, and the language semantics is usually hardcoded within the implementation of their formal analysis algorithm. Due to their monolithic design nature, retargeting them to another language requires significant effort. The retargeting requires either to replace the hardcoded language semantics with the new target language semantics, or to implement a translation from the new language to the existing language. The replacement of the hardcoded semantics almost amounts to rewriting the entire codebase, which is not viable in most cases. The language translation approach could be better than the semantics replacement, requiring affordable effort in case that the new language is similar to the existing one. However, if the new language is quite different from the existing language (for example, a register-based assembly language versus a stack-based one), translation would be at best ineffective if not infeasible. Moreover, the new language semantics (either defined in hardcoded form or indirectly defined via translation) may not be reusable for other formal tools. The lack of reusability leads to fragmentation in the formal tool community.

Language-independent formal methods [45,44,43,41,11] have been proposed to mitigate the reusability issue. The idea is to develop universal formal methods and tools that are *parameterized by language semantics*, and derive formal analysis tools for each target language by instantiating the universal ones with plugging-in the target language semantics. Moreover, theories [11,41,33] have been established that allow to plug-in *operational* language semantics. The underlying theories enable formal reasoning about programs directly over operational semantics without the burden of having to specify additional axiomatic semantics and prove its soundness with respect to the operational semantics. These operational-semantics-based formal methods benefit from the fact that operational semantics is easier to specify than axiomatic semantics. Indeed, specifying operational semantics amounts to writing an interpreter for the target language, which is doable even without extensive logical-theoretical background.

While the underlying theory to support the proposed methodology has already been established, its practicality has not been comprehensively evaluated. Although a proof of concept implementation and its preliminary evaluation were provided [45], they are rather limited.

The goal of my thesis is to demonstrate and improve the scalability and practicality of the language-independent formal methods parameterized by operational semantics. Required work to support this goal falls into the following categories:

- Specifying real-world language semantics and measuring the specification effort.
- Improving existing and developing new language-independent formal methods.
- Instantiating the above-mentioned language-independent formal methods by plugging-in various real-world programming language semantics.
- Applying the derived formal analysis tools to real-world systems and applications, demonstrating their practical feasibility both in isolation and by comparison to state-of-the-art tools specifically crafted for the target languages.

2 Current Results

This section presents current results of the thesis work.

2.1 Complete Formal Semantics of JavaScript

To evaluate the effectiveness of specifying operational language semantics, as well as to use it for instantiating the universal formal methods, I completely specified an operational semantics of JavaScript [38], the most popular client-side programming language. The JavaScript semantics, called KJS, is the most complete and thoroughly tested formal JavaScript semantics, specifically of ECMAScript 5.1, the latest language standard at the time of writing it. It has been tested against the ECMAScript conformance test suite, and passed all 2,782 test programs for the core language. KJS is far more complete than any other semantics, and even more standards-compliant than production JavaScript engines such as Safari WebKit and Firefox SpiderMonkey. Despite the complex nature of the language semantics, the development of KJS took only four months by a single person with no prior knowledge of JavaScript or of the semantic framework. This result supports the argument that specifying operational semantics is affordable, and the operational-semantics-based formal methods benefit from that.

2.2 Deductive Program Verification

Heap Manipulating Programs As a comprehensive evaluation of the universal formal methods and performance of their derived formal analysis tools, I instantiated a language-independent deductive program verifier [48] by plugging-in three real-world language semantics, C, Java, and JavaScript, and used them to verify full functional correctness of challenging heap-manipulating programs such as AVL tree and red-black tree, written in each different language. Performance of the derived verifiers is comparable to other state-of-the-art language-specific

verifiers. For example, VCDryad [39], a separation logic verifier for C build on the top of VCC [10], takes 260 seconds to verify only the `balance` function in AVL, while it takes the derived C verifier 210 seconds to verify AVL `insert` (including `balance`). This result is a supporting evidence for reusability of the universal formal methods.

Smart Contracts For more thorough evaluation, I applied the universal program verifier to Ethereum smart contracts. The Ethereum smart contract is one of the most important real-world applications that requires rigorous formal methods to ensure correctness and security properties of the contract.

Our methodology for formal verification of smart contracts is as follows. First, we formalize the high-level business logic of the smart contracts, based on a typically informal specification provided by the contract developers, to provide us with a precise and comprehensive specification of the functional correctness properties of the smart contracts. This high-level specification needs to be confirmed by the developer, possibly after several rounds of discussions and changes, to ensure that it correctly captures the intended behavior of their contracts. Then we refine the specification all the way down to the Ethereum Virtual Machine (EVM) level, often in multiple steps, to capture the EVM-specific details. The role of the final EVM-level specification is to ensure that nothing unexpected happens at the bytecode level, that is, that only what was specified in the high-level specification will happen when the bytecode is executed. To precisely reason about the EVM bytecode without missing any EVM quirks, we adopted KEVM [22], a complete formal semantics of the EVM, and instantiated the language-independent deductive program verifier [48] to generate a correct-by-construction deductive program verifier for the EVM. We use the verifier to verify the compiled EVM bytecode of the smart contract against its EVM-level specification. Note that the compiler of a high-level contract language (such as Solidity or Vyper) is not part of our trust base, since we directly verify the compiled EVM bytecode. Therefore, our verification results do not depend on the correctness of the compilers.

ERC20 Token Contracts I verified full functional correctness of multiple ERC20 token contracts, for the first time to the best of my knowledge. First, I took an existing high-level specification of ERC20, called ERC20-K [42], that formalizes the ERC20 standard document [51]. Then, I refined it to obtain a EVM-level specification, called ERC20-EVM. I used the derived EVM program verifier to verify high-profile ERC20 token contracts against the ERC20-EVM specification, and found divergent behaviors across these contracts, illuminating potential security vulnerabilities for any API clients assuming consistent behavior across ERC20 implementations.

As part of this verification effort, I improved the scalability of the EVM verifier. Although it is correct-by-construction and sound, the initial out-of-box EVM verifier was relatively slow and failed to prove many correct programs. I improved its scalability by introducing EVM-specific abstractions and lemmas

to expedite proof search. These abstractions and lemmas allow the verifier to avoid unnecessary symbolic reasoning about the EVM-specific low-level details.

Commercial Smart Contracts I also verified the Bihu KEY token operation contract, a commercially deployed smart contract on Ethereum mainnet. Bihu is a blockchain-based ID system, and KEY is the utility token for the Bihu ID system and community.

I adopted the same verification methodology, where I formalized the high-level business logic of the target smart contract, refined it down to the EVM level, and verified the contract bytecode against the EVM-level specification. A notable difference from that of ERC20 token contract verification, however, is that here the refinement was conducted in multiple steps. Indeed, I defined the following four (refined) specifications, where each subsequent specification refines the previous one:

1. High-level definitional specification
2. High-level constructive specification
3. Solidity-level functional specification
4. EVM-level functional specification

Here, (1) is the high-level specification written with the purpose of communication with the client to ensure that it correctly captures the intended behavior of their contract. While (1) is rather a mathematical definition, (2) refines (1) to make the computation steps explicit, being more constructive. Since (2) employs simply the real arithmetic for the computation steps, (3) refines it to capture the unsigned integer arithmetic of Solidity (and thus EVM) including rounding errors of integer division. Finally, (4) refines (3) further down to the EVM level to capture EVM-specific details.

Specification refinement is critical for this verification effort because of the inherent gap between the (high-level) code written by developers and the (low-level) code that actually runs on the blockchain/EVM. Moreover, we split the refinement process into multiple small steps, which makes it easier to prove soundness of each refinement step.

2.3 Checking Program Equivalence

Intuitively, two (possibly non-terminating) programs are equivalent when given the same input they reach the same relevant states in the same order.

In the formal methods literature, the notion of program (or semantic) equivalence is usually formalized as a bisimulation relation between pairs of states of the two programs that are subject to prove equivalence (which in turn are represented as state transition systems). Reducing program equivalence to proving a relation to be a bisimulation, allows for a coinductive proof that deals with recursion, loops, and non-termination in a uniform and elegant way.

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 strictly harder than recursive or co-recursive enumerability.

It is therefore impossible to find an algorithm that can decide whether any two given programs are equivalent or not. The best we can do is to find techniques and algorithms that work well enough in practice. A possible approach is to find a (witness) relation P and show that it is a 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 bisimulation.

Cross-Language Program Equivalence We have developed an algorithmic semantics-based approach for proving equivalence of programs written in different languages. A novel notion of bisimulation, which we call *cut-bisimulation*, allows the two programs to semantically synchronize at relevant "cut" points, but to evolve independently otherwise. Using the K semantic framework, we have implemented the first language-independent tool for proving program equivalence, parametric in the formal semantics of the two languages.

Our language-independent equivalence checking algorithm is parameterized with the input and output language semantics. The algorithm employs our novel notion of cut-bisimulation and the program point pairs needed for the proof are provided as an input to the algorithm, and are called synchronization points. Intuitively, synchronization points are symbolic descriptions that capture the set of pairs of relevant (concrete) states of the input and output programs. For example, they include the pairs of (symbolic) input and output states of the two programs, and the beginnings of the same loop or cyclic structure of them.

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 a witness. Intuitively, KEQ symbolically executes the two programs, each with its respective (symbolic) input state, until it reaches another pair of states presented in the synchronization points. Then, KEQ re-starts the symbolic execution from the newly reached states until reaching another pair of states. KEQ repeats this process until it reaches the pair of output states, which concludes the equivalence proof.

3 Proposed Work

Below is a summary of part of the thesis work (Section 2) accomplished so far:

- I specified a complete semantics of a high-profile language JavaScript, and showed that the specification effort is affordable.
- I improved the existing universal deductive program verifier, and newly developed the cross-language program equivalence checker with a novel notion of cut-bisimulation.
- I instantiated the universal program verifier with four real-world programming languages, C, Java, JavaScript, and EVM, where both JavaScript and EVM verifiers are the first deductive program verifier for the languages to the best of my knowledge.

– I applied the derived program verifiers to challenging heap manipulation programs and high-profile commercial smart contracts, demonstrating their scalability and practicality.

The remaining part of the thesis work is about comprehensive evaluation of the program equivalence checker especially in the domain of Ethereum smart contracts.

– To verify the production Vyper compiler [17] via translation validation. The translation validation is a natural application of the program equivalence checker. I plan to apply the KEQ tool between a Vyper program and its compiled EVM bytecode.
– To check equivalence between Solidity-generated [16] and Vyper-generated EVM bytecode. This equivalence is useful for ensuring that Vyper and Solidity are able to produce completely compatible code, lending assurance to the independently developed compilers of both languages. *This task is optional.*
– To apply cut-bisimulation to specification refinement. When a high-level specification is refined into a low-level one, a soundness relation between them can be formulated as a (cut-)bisimulation. The idea is to consider a specification as a transition system and establish a (cut-)bisimulation relation between them. For example, a class specification can be seen as a transition system, where a node is a state of the class, and an edge is a pair of pre-/post-states of a method/function. This way, one can systematically reason about the soundness of specification refinements.
– To develop property preserving cut-bisimulation. A property preservation theory of cut-bisimulation has not been fully established. However, the property preserving cut-bisimulation is useful since it allows to indirectly prove that a transition system holds properties by showing that it is cut-bisimilar to another transition system that is known to hold the properties. Combined with the specification refinement, it enables to prove meta-properties[1] of a refined specification by only showing the cut-similarity to the original specification that holds the meta-properties. This approach has values especially when the meta-properties are easier to be proved in the original specification than the refined specification.

4 Proposed Approach

This section provides background and technical details of the proposed approach.

4.1 Notations and Preliminary Notions

We use the following notations in the rest of the paper. 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 $i \in \{1, 2\}$. Let $\mathcal{S}$ be a set of states (thought of as all possible configurations/states of a language,

[1] Indeed, many of security properties are formulated as meta-properties.

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 $\rightarrow$. A (possibly infinite) *trace* $\tau = s_0 s_1 \cdots s_n \cdots$ is a sequence of states with $s_i \rightarrow s_{i+1}$ 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 $next(\mathsf{final}(\tau)) = \emptyset$.

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 [46]) that we attempted, but failed, to use for our task.

Triple $(S, L, \rightarrow)$ is a *labeled transition system* (LTS) when L is a set of labels and $\rightarrow \subseteq S \times L \times S$ is a *labeled transition relation*; we write $p \rightarrow^{\mu} q$ when $(p, \mu, q) \in \rightarrow$. 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. Relation R is a *strong simulation* if, whenever $p \ R \ q$, for each $p \rightarrow^{\mu} p'$, there exists q' such that $q \rightarrow^{\mu} q'$ 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 $\rightarrow^{\epsilon}$. Let $\Rightarrow^{\mu}$ be the composition of $\Rightarrow$, $\rightarrow^{\mu}$, and $\Rightarrow$. Let $\Rightarrow^{\widehat{\mu}}$ be $\Rightarrow^{\mu}$ if $\mu \neq \epsilon$, and $\Rightarrow$ otherwise. Relation R is a *weak simulation* if, whenever $p \ R \ q$, for each $p \rightarrow^{\mu} p'$, 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 4.2, 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 [46]. 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.

4.2 Formalizing Program Equivalence

Intuitively, two (possibly non-terminating) programs are equivalent when given the same input they reach the same relevant states in the same order. When different programming languages are involved, however, the existing notions of bisimulation (see Section 4.1) do not appear to naturally capture this intuitive concept of program equivalence.[2] That is because the existing notions are defined assuming that each of the two transition systems can decide what counts as an observable (or relevant) or as an internal transition, independently from the other, and that the capability to perform such transitions alone is sufficient to define equivalence.

In our case study with LLVM and x86-64, we found it very difficult to say, in isolation, whether a transition is observable or internal. Consider, for example, a 32-bit operation in the generated x86-64. If it corresponds to a 32-bit operation in the original LLVM code then it needs to be observable, but if it is part of implementing a more complex operation in the original LLVM code, say a 64-bit operation, then it needs to be considered an internal transition. Similarly, sometimes sequences of LLVM instructions can be translated into one x86-64 instruction (or inversely), in which case it is not clear which of the original LLVM (or x86-64) instructions are observable and which not.

On the other hand, in the context of program translation it is relatively easy to establish *synchronization points* between the two programs, i.e., pairs of points (more specifically, pairs of states) in the respective programs where the two programs are expected to synchronize their behaviors. For example, the two programs may synchronize at the corresponding entry points of functions or procedures, or entry points of loops, but not necessarily when performing well-determined intermediate computations. We found it convenient and easy to determine, for each language in isolation, which of its instructions need to be considered as potentials for synchronization points. We call the corresponding

[2] For presentation simplicity, we say "program equivalence" and "bisimulation", but our results and algorithms also support "program refinement" and "simulation".

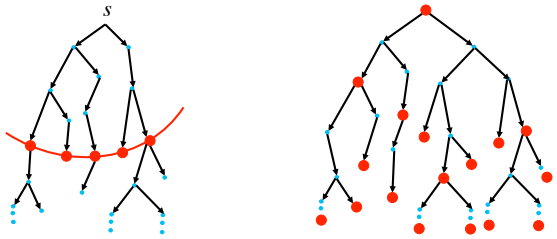

Figure 1. Left: a cut C for state s (each complete s-trace intersects C). Right: a cut C for a transition system (C contains the initial state and is a cut for itself, i.e., for each state in C)

semantic configurations/states *cut points*, and their set simply a *cut*. The intuition for the cut of a transition system corresponding to 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. Then we can define bisimulations only between cut states; we call these *cut bisimulations*.

In order for cut bisimulations to correctly capture program equivalence, 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. What it precisely means for two values 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 parameterize our theoretical results, algorithms and implementation in a binary relation on states $\mathcal{A}$, which we call an *acceptability* (or *compatibility* or *indistinguishability*) relation.

We next introduce our novel notion of cut bisimulation, which makes it easier to deal with intermediate states that are not relevant in identifying equivalence of programs in different languages.

Definition 1 (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 1.*

In a cut transition system, any finite complete trace starting with the initial state terminates in a cut state, and any infinite trace starting with the initial state goes through cut states infinitely often:

Lemma 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 $\tau[k] \in C$.

This result is reminiscent of the notion of Büchi acceptance [5]; 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 2 (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 (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 [46]. A cut-bisimulation on $(S_i, \xi_i, \rightarrow_i, C_i)$ is a bisimulation on $(S_i, \xi_i, \rightarrow_i)$, when $C_i = S_i$. The cuts, however, allow us to consider only the relevant states when comparing two program executions, and *completely hide* the irrelevant intermediate states in each of the two transition systems. As discussed earlier in this section, in

our application domain of cross-language translation-validation, this hiding of irrelevant states is critical. It is not sufficient to consider them internal states connected via ϵ-transitions in the sense of the various weaker notions of bisimulation [46], simply because the execution of one of the programs may step through intermediate states which are not observable or related to intermediate states in the execution of the other program. Nevertheless, cut-bisimulation becomes bisimulation if we cut-abstract the transition systems:

Definition 4 (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 $(C, \xi, \rightsquigarrow)$.*

Proposition 1. *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 $\overline{T_2}$.*

Corollary 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.*

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 5. *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 transitions systems reach compatible states at cut points, and, furthermore, that they cannot indefinitely avoid the cut points:

Theorem 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 5 and the fact that $\sim$ is the union of all cut-bisimulations. Let s_1 be a state with $\xi_1 \to_1^+ s_1$. 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 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 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 1.

4.3 Property Preserving Simulation

Let $T = (S, \to)$ and $T' = (S', \to')$ be two transition systems. Suppose T simulates T', (in other word, T' refines T), denoted by $T > T'$, that is, for all $s_1' \to' s_2'$, there exists $s_1 \to s_2$ such that $s_1 > s_1'$ and $s_2 > s_2'$.[3] Let P_f be a predicate over S (via f), defined by $P_f(s) \stackrel{\texttt{def}}{=} P(f(s))$ for some $s \in S$.

Lemma 2. *Suppose $T > T'$. Suppose P_f is inductive, that is, $P_f(s_1) \wedge s_1 \to s_2 \implies P_f(s_2)$. Let $P_{f'}$ be a predicate over S' (via f'). Suppose $f(s) = f'(s')$ if $s > s'$. Then, $P_{f'}$ is also inductive, that is, $P_{f'}(s_1') \wedge s_1' \to s_2' \implies P_{f'}(s_2')$.*

Proof. Suppose $P_{f'}(s_1')$ and $s_1' \to s_2'$. Since $T > T'$, there exists $s_1 \to s_2$ such that $s_1 > s_1'$ and $s_2 > s_2'$. Then, $P_f(s_1)$ since $f(s_1) = f'(s_1')$. Since P_f is inductive, $P_f(s_2)$. Thus, $P_{f'}(s_2')$ since $f(s_2) = f'(s_2')$.

Example 1. Consider two maps $\texttt{balances} : \mathit{ID} \xrightarrow{\texttt{fin}} \mathbb{N}$ and $\texttt{storage} : \mathbb{N} \xrightarrow{\texttt{fin}} \mathbb{N}$. Suppose `storage` is a refinement of `balances`, denoted by $\texttt{balances} > \texttt{storage}$, defined by:

$$\begin{aligned}&\texttt{balances} > \texttt{storage} \iff \\ &\qquad \forall k \in \mathit{dom}(\texttt{balances}).\ \texttt{balances}(k) = \texttt{storage}(\mathsf{hash}(k)) \\ &\qquad \wedge\ \mathit{dom}(\texttt{balances}) = \{k' \mid k' \in \mathit{dom}(\texttt{storage}) \wedge \exists k.\ \mathsf{hash}(k) = k'\}\end{aligned}$$

where we assume that hash is a crypto-theoretical collision-free hash function for the simplicity of presentation.

Let `total` (and $\texttt{total}'$, resp.) be a function that calculates the total number of balances of `balances` (and `storage`, resp.), defined as follows:

$$\begin{aligned}\texttt{total}(B) &= \Sigma\{v \mid k \mapsto v \in B\} \\ \texttt{total}'(B') &= \Sigma\{v' \mid k' \mapsto v' \in B' \wedge \exists k.\ \mathsf{hash}(k) = k'\}\end{aligned}$$

Now we have:

$$\texttt{total}(\texttt{balances}) = \texttt{total}'(\texttt{storage})$$

[3] This definition is stronger than the classic definition, but they are equivalent if $>$ is right-total. $>$ is abused for both transition systems and states.

Let $P_{\texttt{total}}$ and $P_{\texttt{total}'}$ be predicates over `balances` and `storage`, respectively, defined by:

$$P_{\texttt{total}} \stackrel{\text{def}}{=} \texttt{total}(\texttt{balances}) = \texttt{totalSupply}$$
$$P_{\texttt{total}'} \stackrel{\text{def}}{=} \texttt{total}'(\texttt{storage}) = \texttt{totalSupply}$$

Then, if $P_{\texttt{total}}$ is inductive, then $P_{\texttt{total}'}$ is also inductive. This means that we only need to show that $P_{\texttt{total}}$ is an inductive invariant since $P_{\texttt{total}'}$ follows immediately.

5 Related Work

I present related work for both program verification and program equivalence.

5.1 Program Verification

The program verification literature is rich. We only discuss work close to ours, omitting theoretical work that has not been applied to large languages or work on interactive verification.

A popular approach to building program verifiers for real-world languages is to translate to an IVL and do verification at the IVL level. This results in some re-usability, as the VC generation and reasoning about state properties are implemented only once, at the IVL level. However, the development of translators is both time consuming and susceptible to bugs. Boogie [2] is a popular IVL integrated with Z3. There are several verifiers built on top of Boogie, including VCC [10], HAVOC [27], Spec# [3], Dafny [28], and Chalice [29]. VCDrayd [39] is a separation logic based verifier built on top of VCC. Why3 [15] is another IVL, also integrated with SMT solvers (and other provers). Tools built on top of Why3 include Frama-C [15] and Krakatoa [14]. There are many other practical VC generation based tools (with or without an IVL), including Verifast [25] and jStar [13]. In contrast, we use existing operational semantics directly for verification, without any translation to IVLs or language-specific VC generation.

Recent work proposes translating to a set of Horn clauses instead of an IVL [19]. A semantics based-approach to translation to Horn clauses for a fragment of C is presented in [12], but it is unclear if the approach is generic enough to scale to the entire C or to other real-world languages. An approach for using the interpreter source code as a model of the language in for symbolic execution is proposed in [6], but it is used to generate tests, not verify programs.

We fully share the goal of the mechanical verification community to reduce the correctness of program verification to a trusted formal semantics of the target language [18,37,31,24,1], although our methods are different. Instead of a framework to ease the task of giving multiple semantics of the same language and proving systematic relationships between them, we advocate developing *only one* semantics, operational, and offering an underlying theory and framework with the necessary machinery to achieve the benefits of multiple semantics without the costs. Bedrock [8] is a Coq framework which uses computational higher-order separation logic and supports semi-automated proofs. It can serve as an

IVL, and be the target of translations from other languages which can be certified in Coq based on their operational semantics. Our approach works with the operational semantics directly, and thus does not need any such proofs.

Dynamic logic [20] adds modal operators to FOL to embed program fragments within specifications, but still requires language-specific proof rules (e.g., invariant rules). KeY [4] offers automatic verification for Java based on dynamic logic. Matching logic also combines programs and specifications for static properties, but dynamic properties are expressed in reachability logic which has a language-independent proof system that works with any operational semantics.

Semantics-Based Verification A first version of a language-independent proof system for reachability is given in [45], and [44] shows a mechanical translation of Hoare logic proof derivations for IMP to it. The Circularity proof rule was introduced in [43]. Support for operational semantics using conditional rules is introduced in [41], and support for non-determinism in [11]. These previous results are mostly theoretical, with MatchC a prototype hand-crafted for KernelC mixing language-independent reasoning with the operational semantics of KernelC.

5.2 Program Equivalence

Verified Compilers A different approach to the problem of compiler verification is the full formal verification of the compiler, as in CompCert [30], CakeML [26], and the lambda calculus to typed assembly compiler in [7]. Also formal verification of specific compiler transformation passes, e.g., SSA-based transformations [52] and peephole optimizations [32], has been proposed. Full formal verification is attractive because it gives a development-time guarantee of correctness, whereas the TV approaches discussed later may detect errors only when actually compiling programs and are 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 [30], a verified compiler for C, has been written in the Coq Proof Assistant's specification language, so that the compiler implementation and the corresponding specification theorems are part of the same logic, thus simplifying the formal verification proofs. Such design decisions and development processes cannot easily be applied retroactively in existing compilers and therefore we believe progress in translation validation methods for compilers is important even in the presence of verified compilers.

Translation Validation Systems Translation Validation as a method of verifying the correctness of compilation processes has been proposed by Pnueli *et al.* in [40] and has been used in various settings to prove correctness of specific compiler optimization passes [53,36,35,50,49], discover compiler bugs [21], as well as prove correctness of the whole compilation process [40,47]. VOC-64 [53] for the SGI Pro-64 compiler, Necula *et al.* [36] for the GNU C compiler, Peggy [49] for the Soot Java bytecode optimizer, LLVM-MD [50] and Namjoshi *et al.* [35] for LLVM, are all tools that perform translation validation for optimization passes in the corresponding compilers. Sewell *et al.* [47] presents a TV approach for

the compilation of the seL4 kernel from C to binary. Hawblitzel *et al.* [21] use a TV approach 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.

The proof of program equivalence in the majority of these TV tools [40,53,36,21,47] is based on the generation of a set 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 weaker notion of simulation, since the classic notions are too strong (read inflexible) for practical use in the context of translation validation for compilation processes. We claim that our definition of 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 [53] 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.

Namjoshi *et al.* [35] uses a variant of stuttering-bisimulation with ranking functions, first introduced in [34]. Informally, the ranking function returns an integer rank for each pair in the relation which should represent how many times it is allowed for one of the transition systems to stutter while the other advances before the former has to advance. 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 it eliminates the need for the proof generator to produce ranking functions along with the set of synchronization points.

Finally, LLVM-MD [50] and Peggy [49] move away of simulation proofs, and instead use graph isomorphism techniques to prove equivalence.

All the previous work on Translation Validation we know of assumes that the input and output languages either have essentially equivalent semantics or are translated to a common language or representation: [40] and [53] requires translation to a common representation called Transition Systems, [36] assumes GNU RTL, [35] assumes LLVM IR, [49] and [50] assume a value graphs, [21] requires that the two assembly programs are decompiled into a common language, Boogie [2]. Even, the translation validation approach for the seL4 kernel proposed in [47] requires translation of the input C code and decompilation of the output binary to a common graph language used for equivalence checking. On the other hand, our equivalence checking algorithm is parametric to the input and output language semantics serving as a generalization of the original approach of

Pnueli *et al.* that factors out the requirement for a common semantic framework. The implementation of the algorithm in KEQ is the first known to the authors tool for program equivalence checking that is truly language-independent.

Relation Transition Systems Hur *et al.* [23] presents 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 two different language programs 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.

Hints vs Heuristics 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 [35]. Other works discuss various heuristics that can be used instead. In particular Necula *et al.* [36] describes an inference algorithm to generate a 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 to describe the effect of each basic block, which are generated using symbolic execution. 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.

Mutual Equivalence Proof System Our equivalence checking algorithm was inspired from the language-independent proof system for mutual equivalence introduced in [9]. Instead of a proof system, here 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.

6 Conclusion

The goal of my thesis is to demonstrate and improve the scalability and practicality of the language-independent formal methods parameterized by operational semantics. To support this, I specified various real-world language semantics, showing the affordability of the specification effort. I improved the existing universal deductive program verifier, instantiated it with various language semantics, and applied the derived verifiers to high-profile real-world applications, demonstrating their scalability and practicality. Also, I newly developed the cross-language program equivalence checker with a novel notion of cut-bisimulation. I will thoroughly evaluate the program equivalence checker especially in the domain of Ethereum smart contracts. I believe that this approach would drastically reduce the fragmentation in the verification tool community by eliminating the need for dedicated verifiers for different languages.

References

1. Andrew W. Appel. Verified software toolchain. In *ESOP*, volume 6602 of *LNCS*, pages 1–17, 2011.
2. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *LNCS*, pages 364–387, 2006.
3. Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.
4. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-oriented Software: The KeY Approach.* Springer-Verlag, 2007.
5. Julius R. Büchi. On a Decision Method in Restricted Second-Order Arithmetic. In *International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.
6. Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In *ASPLOS*, pages 239–254. ACM, 2014.
7. Adam 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*, PLDI '07, pages 54–65, New York, NY, USA, 2007. ACM.
8. Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245, 2011.
9. Ştefan Ciobâcă, Dorel Lucanu, Vlad Rusu, and Grigore Roşu. *A Language-Independent Proof System for Mutual Program Equivalence*, pages 75–90. Springer International Publishing, Cham, 2014.
10. Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42, 2009.
11. Andrei Ştefănescu, Stefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. In *RTA*, volume 8560 of *LNCS*, pages 425–440, July 2014.

12. Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Semantics-based generation of verification conditions by program specialization. In *PPDP*, pages 91–102. ACM, 2015.
13. Dino Distefano and Matthew J. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, pages 213–226. ACM, 2008.
14. Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In *CAV*, volume 4590 of *LNCS*, pages 173–177, 2007.
15. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *ESOP*, volume 7792 of *LNCS*, pages 125–128, 2013.
16. Ethereum Foundation. The solidity contract-oriented programming language. `https://github.com/ethereum/solidity`.
17. Ethereum Foundation. Vyper: New experimental smart contract programming language. `https://github.com/ethereum/vyper`.
18. Chris George, Anne E. Haxthausen, Steven Hughes, Robert Milne, SÃÿren Prehn, and Jan Storbank Pedersen. *The RAISE Development Method*. Prentice Hall, 1995.
19. Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012.
20. David Harel, Dexter Kozen, and Jerzy Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604, 1984.
21. Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott 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*, ESEC/FSE 2013, pages 191–201, New York, NY, USA, 2013. ACM.
22. Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Dwight Guth, Philip Daian, and Grigore Roşu. Kevm: A complete semantics of the ethereum virtual machine.
23. Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. The marriage of bisimulations and kripke logical relations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 59–72, New York, NY, USA, 2012. ACM.
24. Bart Jacobs. Weakest pre-condition reasoning for Java programs with JML annotations. *J. Logic and Algebraic Programming*, 58(1-2):61–88, 2004.
25. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of the Third International Conference on NASA Formal Methods (NFM'11)*, volume 6617 of *LNCS*, pages 41–55, 2011.
26. Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: A verified implementation of ml. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM.
27. Shuvendu K. Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In *POPL*, pages 115–126, 2006.
28. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, pages 348–370, 2010.
29. K. Rustan M. Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In *ESOP*, volume 6012 of *LNCS*, pages 407–426, 2010.

30. Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.
31. Hanbing Liu and J. Strother Moore. Java program verification via a JVM deep embedding in ACL2. In *TPHOLs*, volume 3223 of *LNCS*, pages 184–200, 2004.
32. Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 22–32, New York, NY, USA, 2015. ACM.
33. Brandon Moore, Lucas Peña, and Grigore Roşu. Program verification by coinduction. In *27th European Symposium on Programming (ESOP)*, 2018.
34. Kedar S. Namjoshi. *A simple characterization of stuttering bisimulation*, pages 284–296. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.
35. Kedar S. Namjoshi and Lenore D. Zuck. *Witnessing Program Transformations*, pages 304–323. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
36. George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.
37. Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.
38. Daejun Park, Andrei Stefanescu, and Grigore Rosu. KJS: A complete formal semantics of JavaScript. In *PLDI*, pages 346–356. ACM, 2015.
39. Edgar Pek, Xiaokang Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in C using separation logic. In *PLDI*, pages 440–451. ACM, 2014.
40. Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166, London, UK, UK, 1998. Springer-Verlag.
41. Grigore Roşu, Andrei Ştefănescu, Stefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *LICS*, pages 358–367. IEEE, 2013.
42. Grigore Rosu. Erc20-k: Formal executable specification of erc20. `https://github.com/runtimeverification/erc20-semantics`, 2017.
43. Grigore Roșu and Andrei Ștefănescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012.
44. Grigore Roșu and Andrei Ștefănescu. From Hoare logic to matching logic reachability. In *FM*, volume 7436 of *LNCS*, pages 387–402, 2012.
45. Grigore Roșu and Andrei Ștefănescu. Towards a unified theory of operational and axiomatic semantics. In *ICALP*, volume 7392 of *LNCS*, pages 351–363, 2012.
46. Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 2011.
47. Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified os kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 471–482, New York, NY, USA, 2013. ACM.
48. Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 74–91, New York, NY, USA, 2016. ACM.
49. Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '09, pages 264–276, New York, NY, USA, 2009. ACM.

50. Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for llvm. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 295–305, New York, NY, USA, 2011. ACM.
51. Fabian Vogelsteller and Vitalik Buterin. Erc-20 token standard. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`, 2015.
52. Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal verification of ssa-based optimizations for llvm. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 175–186, New York, NY, USA, 2013. ACM.
53. Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. Voc: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9:2003, 2003.