Cut-Bisimulation and Program Equivalence

Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Rosu

University of Illinois at Urbana-Champaign

Abstract. 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.

1 Introduction

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.

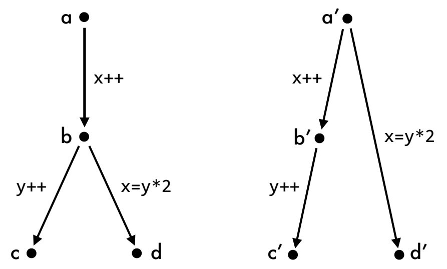
A simulation-based technique for proving equivalence of transition systems is a well-studied approach that admits a coinductive proof that deals with recursion, non-termination, and nondeterminism of the systems in a uniform and elegant way. In such a technique, the notion of equivalence is formulated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

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., [7]), but it does not admit program transformations that modify branching structures. For example, consider a simple program transformation

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

```
P:  while(true) { x++; if (*) { y++; } else { x=y*2; } }
P': while(true) { if (*) { x++; y++; } else { x=y*2; } }
```

(a) Two equivalent loops



(b) Transition systems of the loop bodies

Fig. 1: Program transformation example (as part of partial redundancy elimination). The `if(*)` statement denotes a non-deterministic branching operation.

example shown in Figure 1(b). This transformation is common in compiler optimization (e.g., partial redundancy elimination). The seemingly equivalent two programs, however, are not stuttering-bisimilar (thus neither strongly-bisimilar), mainly because of the different branching structure.[2] Specifically, the pair of the intermediate nodes b and b' cannot be related in a bisimulation relation. One may argue that there exists a stuttering bisimulation, $R = \{(a, a'), (b, a'), (c, b'), (c, c'), (d, d')\}$, but R is not useful for practical purposes, since it requires that the set of atomic predicates that hold at node a is equal to the one at node c.[3]

Trace equivalence is another notion of equivalence that is more relaxed than classic bisimulation variants. For example, the two programs in Figure 1(b) are trace-equivalent,[4] that is, they yield the same final state given the same initial state. The notion of trace equivalence, however, is not amenable to deal with recursive structures, especially when termination of given systems is not obvious. For example, consider the two loops in Figure 1(a). It is not straightforward to establish trace equivalence between the loops, especially due to the non-termination.

On the other hand, a natural way of reasoning about equivalence of the two loops in Figure 1(a) is to employ both proof techniques of bisimulation and

[2] Indeed, this example is reminiscent of the classic example of systems that are trace equivalent but not bisimilar.

[3] Specifically, consider four sets of atomic predicates, A, B, C, and D, where A holds at nodes a and a', B at b and b', C at c and c', and D at d and d'. Then R implies $A = B = C$ by the definition of stuttering bisimulation, while we are mostly interested in the case $A \neq C$ in practice.

[4] More precisely, throughout this paper, we consider completed trace equivalence where traces are identified by their state update, i.e., difference between their initial and final states.

trace equivalence. Specifically, 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 the trace equivalence proof technique, and the second step is to prove equivalence of the loops using the bisimulation-based technique, while assuming that their loop bodies are equivalent.

In this paper, we present a bisimulation variant, named *cut-bisimulation*, that captures this combination of trace equivalence 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 2). A nice property of the cut is that there exists no infinite path between the cut states, thus proving trace equivalence between them is tractable. Intuitively, a cut-bisimulation proof amounts to proving trace equivalence between the cut states, and proving a strong bisimulation 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 the plain trace 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 2). We will also show that cut-bisimulation can be tuned to stuttering bisimulation by properly setting the cut set (see Section 3). 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 [10]. 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 4). 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 1 into LLVM bitcode, and proved equivalence of the two LLVM programs using the tool (see Section 5).

2 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 can establish a cut bisimulation even for non-equivalent programs.[5] 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. Thus, 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.

Notations Given a binary relation $R \subseteq \mathcal{S}_1 \times \mathcal{S}_2$, we write $a \ R \ b$ to denote $(a,b) \in R$; and $R_1 = \{a \mid \exists b \,.\, a \ R \ b\}$ and $R_2 = \{b \mid \exists a \,.\, a \ R \ b\}$ to denote the projections $\Pi_i(R)$ for $i \in \{1,2\}$.

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 $T = (S, \xi, \to)$ 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* $\to \ \subseteq S \times S$. Let *next*$(s)$ denote the set $\{s' \mid s \to s'\}$. T is *finitely branching* iff *next*(s) is finite for each $s \in S$. Let $\to^*$ be the reflexive and transitive closure of $\to$, and $\to^+$ be the transitive closure of $\to$.

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

[5] 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.

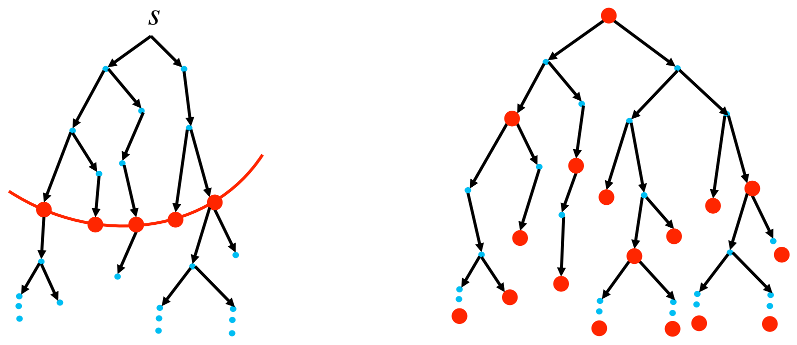


Fig. 2: Left: a cut C for state s (each complete s-trace intersects C). Right: a cut C for a transition system (C contains the initial state and is a cut for itself, i.e., for each state in C)

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 $\mathit{next}(\mathsf{final}(\tau)) = \emptyset$.

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 2.*

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 [2]; 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 $T = (S, \xi, \to, C)$ 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 $T_i = (S_i, \xi_i, \to_i, C_i)$ 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 [11]. A cut bisimulation on $(S_i, \xi_i, \to_i, C_i)$ is a bisimulation on $(S_i, \xi_i, \to_i)$, when $C_i = S_i$. Also, cut-bisimulation becomes bisimulation if we cut-abstract the transition systems:

Definition 4 (Cut-Abstract Transition System). *Let T be a cut transition system $(S, \xi, \to, C)$. The* cut-abstract transition system *of T, written $\overline{T}$, is the (standard) transition system $(C, \xi, \rightsquigarrow)$.*

Proposition 1. *Let $T_i = (S_i, \xi_i, \to_i, C_i)$ 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 $T_i = (S_i, \xi_i, \to_i, C_i)$ 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 $s_1 \to_1^+ s'_1$, there exists some $s'_2 \in C_2$ with $s_2 \to_2^+ s'_2$ 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 $T_i = (S_i, \xi_i, \to_i, C_i)$ 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 $\xi_1 \mathrel{\mathsf{P}} \xi_2$. 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 $\xi_1 \mathrel{\mathsf{P}} \xi_2$.*

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 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 $\xi_1 \rightarrow_1^+ s_1$. Then we have two cases:

- When $s_1 \in C_1$. There exists s_2 such that $\xi_2 \rightarrow_2^+ s_2$ and $s_1 \sim s_2$ by Corollary 1.
- When $s_1 \notin C_1$. There exists s_1' such that $s_1 \rightarrow_1^+ s_1'$ 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 $\xi_2 \rightarrow_2^+ s_2$ and $s_1' \sim s_2$ by Corollary 1.

2.1 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 $T = (S, \xi, \rightarrow, C)$ be a cut transition system. Let P be a predicate over a domain D, and $f : S \rightarrow D$ be a state normalization function. Let P_f be a predicate over S, defined by $P_f(s) \overset{\text{def}}{=} P(f(s))$ 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 $T_i = (S_i, \xi_i, \rightarrow_i, C_i)$ 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 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 $f_1(s_1') = f_2(s_2')$.

3 Comparison to Stuttering Bisimulation

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

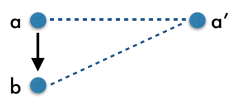
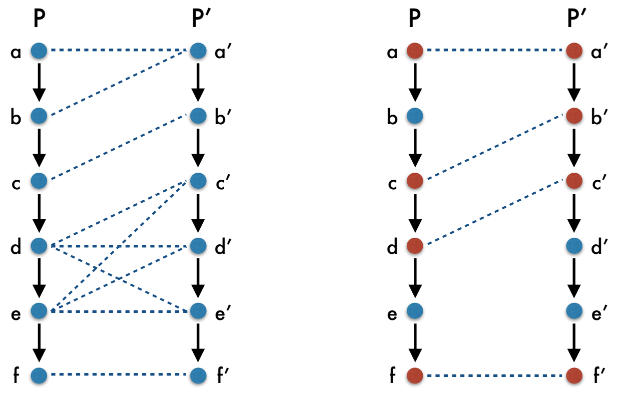


Fig. 3: Systems that are stuttering-bisimilar, but not cut-bisimilar.

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 1, there exist systems that are cut-bisimilar but not stuttering-bisimilar, as shown in Figure 1. Also, there exist (trivial) systems that are stuttering-bisimilar but not cut-bisimilar, as shown in Figure 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 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 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 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



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

Fig. 4: Conversion of stuttering bisimulation to cut-bisimulation

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

A *partition* ϕ of a finite trace τ is a finite sequence of indexes $p_0, p_1, \cdots, p_n$, where $0 = p_0 < p_1 < \cdots < p_n < \mathsf{size}(\tau)$, 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 $0 \le k \le n$. 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 [1].

Definition 6 (Stuttering Bisimulation [1]). *Let $T_i = (S_i, \xi_i, \to_i, L_i)$ 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:*

$$\mathsf{size}(\phi_1) = \mathsf{size}(\phi_2), \text{ and for any } 0 \le k < \mathsf{size}(\phi_1),\ [\![\phi_1[k]]\!] \times [\![\phi_2[k]]\!] \subseteq R \quad (1)$$

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 7. *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 $k \geq 0$, $p_k < p_{k+1}$:*

$$M(\tau[i]) = M(\tau[j]) \quad \text{for} \;\; p_k \leq i, j < p_{k+1}, \tag{2}$$

$$\tau[p_k] \neq \tau[i] \quad \text{for} \;\; p_k < i < p_{k+1}, \;\text{and} \tag{3}$$

$$M(\tau[p_{k+1} - 1]) \neq M(\tau[p_{k+1}]) \quad \text{or} \quad \tau[p_k] = \tau[p_{k+1}]. \tag{4}$$

*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 8 (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* (1)*, and the converse also holds.*

Theorem 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 4 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 2 and Lemma 5, 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 8, 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 5, 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.

The lemmas used in the above proof are presented in Appendix A.

4 Language-Parametric Program Equivalence Checker

We implemented a language-independent equivalence checking tool on top of the $\mathbb{K}$ framework [10]. $\mathbb{K}$ 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 [9], 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 5 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.

Our KEQ implementation follows the model of the theoretical Algorithm 1. Function main essentially checks whether P is a cut-bisimulation: for each pair

```
   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($p_1, p_2$) $=$ false then
4              return false;
5          end
6      end
7      return true;
8  Function check($p_1, p_2$):
9      $N_1 \leftarrow$ next$_1(p_1)$;  $N_2 \leftarrow$ next$_2(p_2)$;
10     foreach $(n_1, n_2) \in N_1 \times N_2$ do
11         if $\boxed{(n_1, n_2) \in \mathsf{P}}$ then          // $\boxed{[\![(n_1, n_2)]\!] \subseteq [\![\overline{\mathsf{P}}]\!]}$
12             $n_1$.color $\leftarrow$ black;  $n_2$.color $\leftarrow$ black;
13         end
14     end
15     if $\forall n \in N_1 \cup N_2.\ n$.color $=$ black then
16         return true;
17     end
18     return false
19 // Returns cut-successors of $n$
20 Function next$_i(n)$:
21     $N \leftarrow \{n\}$;  $Ret \leftarrow \emptyset$;
22     while $N$ is not empty do
23         choose $n$ from $N$;  $N \leftarrow N \setminus \{n\}$;
24         $N' \leftarrow \{n' \mid n \boxed{\to_i} n'\}$;          // $\boxed{\overline{\to}_i}$
25         foreach $n' \in N'$ do
26             if $\boxed{n' \in C_i}$ then          // $\boxed{[\![n']\!] \subseteq [\![\overline{C_i}]\!]}$
27                 $n'$.color $\leftarrow$ red;
28                 $Ret \leftarrow Ret \cup \{n'\}$;
29             else
30                 $N \leftarrow N \cup \{n'\}$;
31             end
32         end
33     end
34     return $Ret$;
```

Algorithm 1: Equivalence checking algorithm. For checking cut-simulation, replace $N_1 \cup N_2$ with N_1 at line 15. 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.

$(p_1, p_2) \in \mathsf{P}$, 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 9), and checks whether each pair of the successors is related in P (line 11). The pairs found to be related in P are marked in black (line 12), while the others remain in red. If all of the successors are in black, it returns true (line 16). 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 16, 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 1 may not terminate in practice, since P could be infinite. Line 2 assumes that P is recursively enumerable, so iterable. Furthermore, lines 24 and 10 terminate only if T_i is finitely branching. We will explain how to refine Algorithm 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 4 (Correctness of Algorithm 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 1 does not terminate with false, then $T_1 \sim_{\mathcal{A}} T_2$.*

Indeed, suppose that Algorithm 1 does not terminate with *false*. Then none of the `check(`p_1`,`p_2`)` calls (line 3) terminates with *false* (line 18). Since the loop at Line 10 always terminates (T_1 and T_2 are finitely branching), it means that all nodes are colored black at line 15. 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), that is, $T_1 \sim_{\mathcal{A}} T_2$ (Definition 5).

Note that Algorithm 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 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 24. Also, $\mathbb{K}$ is fully integrated with the Z3 solver [4], allowing us to implement the inclusions at lines 11 and 26 by requesting Z3 to solve the implications of the

```
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;
}
```

Fig. 5: Program transformation example. Two programs are equivalent provided that `n` is an even natural number.

corresponding formulae. It is clear that the symbolic variant of Algorithm 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).

We implemented the symbolic variant of Algorithm 1 in a tool called KEQ for checking language-independent program equivalence.[6]

Example To illustrate how KEQ works, consider the example in Figure 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

$$\begin{aligned} s_{p_{\mathsf{init}}} &\equiv \mathtt{i} \mapsto i \,*\, \mathtt{n} \mapsto n \quad \text{where } \ n \bmod 2 = 0 \\ s'_{p_{\mathsf{init}}} &\equiv \mathtt{i} \mapsto i' \,*\, \mathtt{n} \mapsto n' \quad \text{where } \ n' \bmod 2 = 0 \\ \psi_{p_{\mathsf{init}}} &\equiv n = n' \end{aligned}$$

$s_{p_{\mathsf{init}}}$ 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

$$\begin{aligned} s_{p_{\mathsf{loop}}} &\equiv \mathtt{i} \mapsto i \,*\, \mathtt{n} \mapsto n \quad \text{where } \ i \bmod 2 = 0 \ \wedge\ n \bmod 2 = 0 \\ s'_{p_{\mathsf{loop}}} &\equiv \mathtt{i} \mapsto i' \,*\, \mathtt{n} \mapsto n' \quad \text{where } \ i' \bmod 2 = 0 \ \wedge\ n' \bmod 2 = 0 \\ \psi_{p_{\mathsf{loop}}} &\equiv i = i' \ \wedge\ n = n' \end{aligned}$$

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

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

$$\begin{aligned} s_{p_{\mathsf{final}}} &\equiv \mathtt{i} \mapsto i \ * \ \mathtt{n} \mapsto n \\ s'_{p_{\mathsf{final}}} &\equiv \mathtt{i} \mapsto i' \ * \ \mathtt{n} \mapsto n' \\ \psi_{p_{\mathsf{final}}} &\equiv i = i' \ \wedge \ n = n' \end{aligned}$$

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 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 1) and executes the function `check` with it. In `check`, it first symbolically executes each program (lines 9 and 24) until they reach another synchronization point (line 26). 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:

$$\begin{aligned} s_1 &= \mathtt{i} \mapsto 0 \ * \ \mathtt{n} \mapsto n \quad \text{where } \ n \bmod 2 = 0 \\ s'_1 &= \mathtt{i} \mapsto 0 \ * \ \mathtt{n} \mapsto n' \quad \text{where } \ n' \bmod 2 = 0 \end{aligned}$$

KEQ checks if $(s_1, s'_1, \psi_{p_{\mathsf{init}}})$ is matched by p_{loop} (line 11), which is true. Since s_1 is the only pair that reaches p_{loop}, the `check` function returns true (line 16).

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:

$$\begin{aligned} s_2 &= \mathtt{i} \mapsto i+2 \ * \ \mathtt{n} \mapsto n \quad \text{where } \ i \bmod 2 = 0 \ \wedge \ n \bmod 2 = 0 \\ s'_2 &= \mathtt{i} \mapsto i'+2 \ * \ \mathtt{n} \mapsto n' \quad \text{where } \ i' \bmod 2 = 0 \ \wedge \ n' \bmod 2 = 0 \end{aligned}$$

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 26). KEQ checks if $(s_2, s'_2, \psi_{p_{\mathsf{loop}}})$ is matched by p_{loop} (line 11), 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 7).

On the other hand, Figure 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:

```
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;
}
```

Fig. 6: Two equivalent programs with the out-of-order loop iteration.

- 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.

5 Preliminary Evaluation of Keq

As a preliminary evaluation, we revisit the example of Figure 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 7 shows two LLVM programs that correspond to the code shown in Figure 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 7(a) and line 8 in Figure 7(b).

Figure 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 starting from $p1$ will reach the entry of the loop through the back edge with $\mathtt{\%x.1} = \mathtt{\%x.1}' = 2$ and $\mathtt{\%y.1} = \mathtt{\%y.1}' = \textit{if-then-else}(\mathtt{@check()}, 2, 1)$, where primed variable names refer to the variables in Figure 7(b). These values satsify the constraints for synchronization point $p2$, so it will be reached from $p1$. Finally, traces starting from $p2$, where $\mathtt{\%x.1} = \mathtt{\%x.1}' = X$ and $\mathtt{\%y.1} = \mathtt{\%y.1}' = Y$, will reach the entry of the loop through

```
 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

Fig. 7: A simple partial redundancy elimination transformation in LLVM. The two LLVM programs mirror the loops shown in Figure 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`.

the back edge with $\texttt{\%x.1} = \texttt{\%x.1}' = \textit{if-then-else}(\texttt{@check()}, X + 1, 2Y)$ and $\texttt{\%y.1} = \texttt{\%y.1}' = \textit{if-then-else}(\texttt{@check()}, Y + 1, Y)$. 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 8 is straightforward enough to be automatically inferred. Indeed, Necula *et al.* [8] 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.

6 Related Work

The program equivalence literature is rich, and here we discuss only the formal proof systems of program equivalence that are closely related to ours. For the classic bisimulation and its variants, we refer the reader to [11].

Sync Point	*Previous Block*	*Constraints*
$p0$	-	-
$p1$	`%entry`	-
$p2$	`%if.end`	$\texttt{\%x.1} = \texttt{\%x.1}' \wedge \texttt{\%y.1} = \texttt{\%y.1}'$

Fig. 8: Synchronization points for the PRE transformation in Figure 7. The primed variable names refer to the variables in Figure 7(b).

Namjoshi *et al.* [7] uses a variant of stuttering-bisimulation with ranking functions, first introduced in [6]. 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.

Hur *et al.* [5] 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.

Ciobaca *et al.* [3] proposes the notion of mutual equivalence and presents its proof system, by which our equivalence checking algorithm was inspired. 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 different languages.

7 Conclusion

In this paper, we presented cut-bisimulation, a new notion of bisimulation that combines traditional bisimulation and trace equivalence to provide a unified formal notion of program equivalence, especially for the purpose of translation validation. We have implemented a cut-bisimulation-based, language-parametric equivalence checking tool, Keq, and instantiated it with an LLVM semantics, toward its use in translation validation of the LLVM compiler.

References

1. Browne, M.C., Clarke, E.M., Grümberg, O.: Characterizing finite kripke structures in propositional temporal logic. Theor. Comput. Sci. **59**(1-2), 115–131 (Jul 1988). https://doi.org/10.1016/0304-3975(88)90098-9, `http://dx.doi.org/10.1016/0304-3975(88)90098-9`
2. Büchi, J.R.: On a Decision Method in Restricted Second-Order Arithmetic. In: International Congress on Logic, Methodology, and Philosophy of Science. pp. 1–11. Stanford University Press (1962)
3. Ciobâcă, Ş., Lucanu, D., Rusu, V., Roşu, G.: A Language-Independent Proof System for Mutual Program Equivalence, pp. 75–90. Springer International Publishing, Cham (2014)
4. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS'08. LNCS, vol. 4963, pp. 337–340 (2008)
5. Hur, C.K., Dreyer, D., Neis, G., Vafeiadis, V.: The marriage of bisimulations and kripke logical relations. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 59–72. POPL '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2103656.2103666, `http://doi.acm.org/10.1145/2103656.2103666`
6. Namjoshi, K.S.: A simple characterization of stuttering bisimulation, pp. 284–296. Springer Berlin Heidelberg, Berlin, Heidelberg (1997). https://doi.org/10.1007/BFb0058037, `http://dx.doi.org/10.1007/BFb0058037`
7. Namjoshi, K.S., Zuck, L.D.: Witnessing Program Transformations, pp. 304–323. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), `http://dx.doi.org/10.1007/978-3-642-38856-9_17`
8. Necula, G.C.: Translation validation for an optimizing compiler. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. pp. 83–94. PLDI '00, ACM, New York, NY, USA (2000). https://doi.org/10.1145/349299.349314, `http://doi.acm.org/10.1145/349299.349314`
9. Rogers, Jr., H.: Theory of Recursive Functions and Effective Computability. MIT Press, Cambridge, MA, USA (1987)
10. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
11. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press, New York, NY, USA (2011)

A Lemmas for Theorem 3

Lemma 2. *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 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 7.

Let us prove the claim 2. If τ is infinite, we immediately have $\mathsf{size}(\phi_M(\tau)) > 1$. Suppose $\tau = s_0 s_1 \dots s_n$ is finite, where $n \geq 1$. Assume that $\mathsf{size}(\phi_M(\tau)) = 1$. Then, by the equation (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 $M(s_0) = M(s_n)$.

Lemma 3. *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 (4) for $k = 0$ in Definition 7, which is a contradiction because there exists an M-induced partition of τ.

Lemma 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. 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 2. 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 7. 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, 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 1.

Lemma 5. *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. 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 $1 \leq i < \mathsf{size}(\phi_M(\tau))$,*

$$\tau[p_i] \textit{ is an entry vertex, and} \tag{5}$$

$$\tau[k] \textit{ is not an entry vertex for any } p_{i-1} < k < p_i. \tag{6}$$

Proof. Let us first prove the equation (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 (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 (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 (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 (2).
- When $\tau[p_{i-1}]$ and $\tau[k]$ are in the same M-induced component. Then, by the condition (i) of Definition 7, $\tau[p_{i-1}] = \tau[k]$, which is a contradiction because of the equation (3).