Abstract Semantic Diffing of Evolving Concurrent Programs

Ahmed Bouajjani, Constantin Enea, Shuvendu Lahiri

Abstract Semantic Diffing of Evolving Concurrent Programs*

Ahmed Bouajjani[1], Constantin Enea[1], and Shuvendu Lahiri[2]

[1] IRIF, Univ. Paris Diderot, {abou,cenea}@irif.fr
[2] Microsoft Research, shuvendu@microsoft.com

Abstract. We present an approach for comparing two closely related concurrent programs, whose goal is to give feedback about interesting differences without relying on user-provided assertions. This approach compares two programs in terms of cross-thread interferences and data-flow, under a parametrized abstraction which can detect any difference in the limit. We introduce a partial order relation between these abstractions such that a program change that leads to a "smaller" abstraction is more likely to be regression-free from the perspective of concurrency. On the other hand, incomparable or bigger abstractions, which are an indication of introducing new, possibly undesired, behaviors, lead to succinct explanations of the semantic differences.

1 Introduction

The lifetime of a software module includes multiple changes that range from refactoring, addition of new features to bug or performance fixes. Such changes may introduce regressions which in general are hard to detect and may reveal themselves much later in the software's life-cycle. Dealing with this issue is particularly difficult in the context of concurrent programs, where the bugs are characterized by subtle interleaving patterns that tend to manifest in the field while passing an extensive testing phase.

Checking whether a change in a program is regression-free reduces to a standard, single-program, verification problem assuming a specification of the possible regressions is provided, for instance, using assertions. However, such specifications are rarely present in practice.

A different perspective, which avoids the need for specifications, would be to compare the two versions of a program (before and after the change) under a certain abstraction, which is precise enough to distinguish common specifications. Typical examples involve (bi)simulations, sets of reachable configurations [3], and equality between input-output relations. Simulations define a partial order over

* This work is supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 678177).

[3] By configuration, we mean the tuple of thread-local states together with the state of the shared memory.

the set of all programs (bisimulations define an equivalence relation), which in practice, relates very few programs across refactoring, bug-fixes, or adding new features. For instance, a transformation that is widely used in bug-fixing consists in reordering program statements within the same thread. For realistic programs, there exists no simulation relation between a program obtained by applying such a transformation and the initial version, or vice-versa. Therefore, using simulations as an indicator of regression-freeness, i.e., the new version is considered regression-free when it is simulated by the old version, would lead to too many false negatives. The same holds when comparing two programs with respect to their reachable sets of configurations. Comparing input-output relations is also not suitable in our context, because of the concurrency. Such relations are hard to compute and also, hard to use for checking regression-freeness, because of the non-determinism introduced by the thread scheduler.

In this paper we propose a new approach for comparing two closely related concurrent programs (subsequent versions of programs), which allows to relate more programs than simulations, for instance. The goal of this approach is to give feedback about interesting differences as opposed to noise from any change, without relying on user-provided assertions. From the perspective of concurrency, interesting differences concern, for instance, enabling new interferences from other threads (e.g., reading new values written by other threads), or new violations of atomicity (for some decomposition of the program in atomic blocks, which is implicit in the mind of the programmer).

The starting point of our approach is a program semantics based on traces [22], which are compact representations of sets of interleavings. A trace is a graph where nodes represent read and write actions, and edges represent the *program order*, which relates every two actions executed by the same thread, and *data-flow dependencies*, i.e., which action writes the value read by a read action, and in which order values are written to the memory. A trace represents all the interleavings which are consistent with the program order and the data-flow dependencies. The traces of two programs can be compared assuming a matching relation between variables and statements in the two programs, such that matching statements read and respectively, write the same set of variables (modulo the variable matching). Roughly, if this matching relation is an isomorphism between two traces of different programs, then the sets of configurations reachable in the interleavings represented by these two traces are the same (modulo the constants used in the statements).

We define a partial order relation between programs based on abstract representations of sets of traces. We use abstract representations instead of sets of (concrete) traces because ordering programs with respect to the latter has the same disadvantages as the use of simulation relations or sets of reachable configurations (see Section 2 for an example). For instance, bug fixes based on statement reordering or modifying the placement of the synchronization primitives lead straightaway to incomparable sets of traces – the set of actions or the program order are different.

As a first abstraction step, we consider "projected" traces, where roughly, the program order and all the synchronization statements are omitted [4]. This allows us to expose differences that concern only the data-flow in the program and not, for instance, the order in which different variables are assigned, or the synchronization mechanisms used to constrain the interference between threads. Replacing lock/unlock primitives with wait/nofity or semaphores induces no difference with respect to sets of projected traces provided that the set of possible schedules remains the same.

Then, we define abstractions of sets of projected traces, called *abstract traces*. Every abstract trace contains a graph structure describing the *union* of the projected traces it represents. The nodes of this graph correspond to program statements and the edges correspond to data-flow dependencies present in *some* projected trace. We restrict ourselves to loop-free programs which implies that these graphs are of bounded size. Handling loops will require some predefined equivalence relation between statements, a node in the graph representing an equivalence class with respect to this relation. Adding information about which sets of dependencies are present together in the same projected trace allows to refine a given abstract trace. Abstract traces are parametrized by an integer k which bounds the size of the sets of dependencies that are tracked (whether they occur in the same trace). We define a partial order between abstract traces which essentially corresponds to the fact that every set of dependencies in one abstract trace occurs in the other one as well. An abstract trace not being "smaller" than another one implies that the set of concrete traces corresponding to the first one is not included in the set of concrete traces corresponding to the second one (and thus reveals a difference in thread interference). However, on the opposite side, the "smaller than" relation does not imply trace set inclusion unless k is big enough (roughly, the square of the program size). Instead, it can be thought of as an *indicator* for not introducing undesired behaviors, whose precision increases as bigger values of k are considered.

This abstraction framework enables a *succinct* representation of the difference between two programs. For a fixed k, the size of the abstract trace is polynomial in the size of the input program while the size of a complete set of traces is in general of exponential size. Small values of k allow to explain the difference between two programs in terms of small sets of dependencies that occur in the same execution, instead of a complete trace or interleaving.

We show that the problem of deciding the difference with respect to abstract traces of a fixed rank k between two versions of a loop-free program [5] (before and after a program transformation) can be reduced to a set of assertion checking queries. This reduction holds for programs manipulating arbitrary, possibly unbounded, data. The assertion checking queries can be discharged using the existing verification technology. In the context of loop-free *boolean* programs, we

[4] Our framework is not bound to a specific set of program order constraints and statements to be preserved in the projected traces – they can be chosen arbitrarily.

[5] This reduction can be applied to arbitrary programs assuming a bounded unrolling of loops.

show that this problem has a lower asymptotic complexity than the problem of deciding the difference with respect to concrete sets of traces. More precisely, we prove that the first problem can be reduced to a polynomial number of assertion checking queries and that it is Δ_2^P-complete, while the second problem is Σ_2^P-complete. (We recall that Δ_2^P, resp., Σ_2^P, is the class of decision problems solvable by a polynomial time, resp., NP time, Turing machine augmented by an oracle for an NP-complete problem.) This complexity gap shows that the latter problem cannot be reduced to a polynomial number of assertion checking queries unless P=NP.

As a proof of concept, we have applied our framework to a benchmark used for the ConcurrencySwapper synthesis tool [6]. This benchmark consists of pairs of programs, before and after a bug fix, that model real concurrency bug fixes reported in the Linux kernel development archive (`www.kernel.org`). The reachability queries have been discharged using the LazyCseq tool [13, 12] (with backend CBMC [9]). These experiments show that comparing abstract traces for small values of k, i.e., $k \in \{1, 2\}$, suffices to detect interesting semantic changes while ignoring the irrelevant ones. Moreover, the semantic changes are presented succinctly as a small set of data-flow dependencies between program statements, instead of a complex interleaving. This facilitates the task of spotting bugs by allowing the programmer to focus on small fragments of the program's behavior.

2 Motivating examples

We provide several examples to illustrate the abstract semantic diffing framework proposed in this paper and its potential use in verifying concurrency bug fixes.

The program on the left of Fig. 1 is a typical concurrency bug found in device drivers [6], where the second thread may read an uninitialized value of `x` (initially, all variables are 0). Since the second thread runs only when `flag` is set to 1, fixing such a bug consists in permuting the two instructions in the first thread such that `x` is initialized before `flag` is set to 1. The modified version is listed on the right of Fig. 1. Note that the two versions (before and after the fix) have incomparable sets of reachable configurations: the configuration ($\texttt{flag} = 1, \texttt{x} = 0$) is reachable in the first program but not in the second, and ($\texttt{flag} = 0, \texttt{x} = 1$) is reachable in the second but not in the first one. This also implies that there exists no simulation relation from the fixed version to the buggy one, or vice-versa.

Our approach compares abstract representations of data-flow dependencies [22] in the two programs. These dependencies come in two forms:

- *read-from* dependencies from actions writing to a variable to actions reading that variable (specifying the write that a read receives its value from), and
- *store-order* dependencies which specify the order in which writes to the same variable are executed in the memory.

The bottom part of Fig. 1 pictures an abstract trace for each of the two programs where only individual dependencies are tracked (whether they occur in some trace), i.e., of rank 1. We can notice that the set of dependencies in the

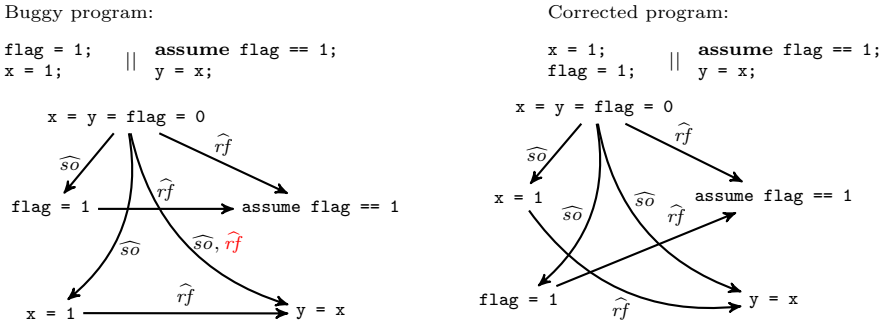

Fig. 1. The program on the left is considered buggy since there exists an execution where `y` takes an uninitialized value of `x`. The second program fixes this bug by permuting the statements in the first thread. The bottom part of the figure pictures their abstract traces of rank 1. Read-from, resp., store-order, dependencies are represented by edges labeled with $\widehat{rf}$, resp., $\widehat{so}$. The second program is a refinement of rank 1 of the first one, but the reverse is not true.

fixed version is a strict subset of the set of dependencies in the original (buggy) version. This fact suggests that the bug fix has removed some behaviors but introduced none. This is not a theoretical guarantee but its likelihood can be increased by considering abstract traces of bigger ranks. Moreover, the difference between the abstract trace of the buggy version and the one of the fixed version consists of one read-from dependency, from a fictitious write which assigns initial values to the variables, to the read of `x` in `y = x`. This dependency is a succinct description of all the interleavings containing the bug, which read an uninitialized value of `x`. The fact that this dependency doesn't occur anymore in the fixed version implies that the buggy behaviors have been removed.

In general, exposing the difference between the data-flow in two programs may require computing *sets* of data-flow dependencies occurring in the *same* execution of one program and not the other one, i.e., abstract traces of rank $k > 1$. Fig. 2 lists two programs doing two parallel increments of a shared variable `x`, without synchronization on the left and protected by locks on the right. In this case, there exists no data-flow dependency admitted only by the first program or only by the second, i.e., the abstract traces of rank 1 are identical. However, there exists a *pair* of data-flow dependencies which occur in the same execution of the buggy program (that has no synchronization) and not in the corrected one (that uses locks): the two reads of `x` (from the assignments to `temp1` and `temp2`) can both take their value from the initial state. Our framework allows to witness such differences for fixed values of the rank k.

3 Multi-Threaded Programs

We consider a simple multi-threaded programming model in which each thread executes a bounded sequence of steps corresponding to assignments, boolean

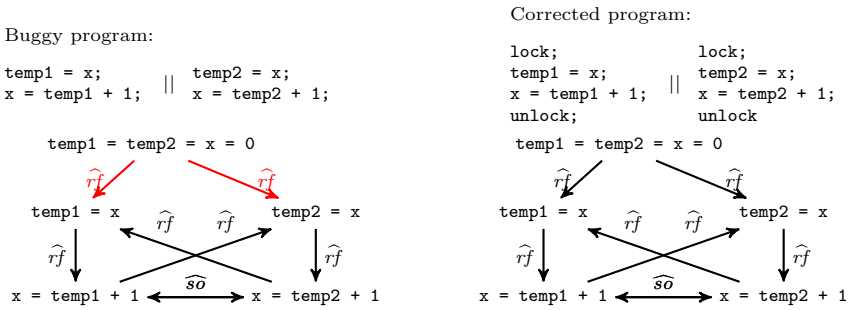

Fig. 2. Two programs doing two parallel increments of `x`. The bottom part of the figure pictures their abstract traces of rank 1. For readability, the $\widehat{so}$ dependencies starting from the assignment representing initial values are omitted. Considering abstract traces of rank 2, the pair of red $\widehat{rf}$ dependencies belongs to the abstract trace of the buggy program but not to that of the correct version. The second program is a refinement of rank 2 of the first one because it has less (pairs of) dependencies which occur in some execution. The reverse doesn't hold.

$$s ::= x \texttt{ := } e \mid \texttt{assume } e \mid \mathsf{lock} \mid \mathsf{unlock} \mid s \texttt{ [] } s \mid s\texttt{; } s \mid$$
$$P ::= s \mid P \,||\, P$$

Fig. 3. The syntax of our language. Each program P is the parallel composition of a fixed number of threads – `;` denotes the sequential composition and `[]` the non-deterministic choice between two control-flow paths. Also, $x \in \mathsf{Vars}$ and e is an expression over Vars.

tests, and synchronization primitives. The semantics of a program is defined as a set of traces [22], which are partially-ordered sets of read or write actions.

Let Vars be a set of variables. The grammar of Fig. 3 describes our language of multi-threaded programs. For generality, we leave the syntax of expressions e in assignments and `assume` statements unspecified. We allow expressions $e = *$ where $*$ is the (nullary) non-deterministic choice operator. Note that `if-then-else` conditionals can be modeled using `assume` statements and the non-deterministic choice. To simplify the exposition, we assume that the same variable doesn't appear in both the left and the right part of an assignment (e.g., we forbid assignments of the form x `:=` $x + 1$). This simplifies the trace semantics given hereafter, and it could be removed assuming that the program is first rewritten to static single assignment form. Also, we consider a minimal set of synchronization statements, $\mathsf{lock}/\mathsf{unlock}$ over a unique lock object. However, our approach easily extends to any class of synchronization primitives. The set of variables in a statement s, resp., a program P, denoted by $\mathsf{Vars}(s)$, resp., $\mathsf{Vars}(P)$, is defined as usual. The set of statements s over a set of variables $V \subseteq \mathsf{Vars}$ is denoted by $\mathsf{Stmts}(V)$. The set of statements of a program P is denoted by $\mathsf{Stmts}(P)$. When all the variables range over the booleans, the program is called a *boolean program*.

Program configurations are variable valuations, and program executions are defined as usual, as interleavings of statements (we assume a sequentially consistent semantics). In the following, we define representations of program executions called *traces*. For a variable x, $\mathbb{W}(x)$ is the set of assignments to x and $\mathbb{R}(x)$ is the set of `assume` e statements where e contains x together with the set of assignments reading the variable x (i.e., x occurs in the right part). We assume that $\mathsf{Stmts}(P)$ contains a fictitious statement init assigning initial values to all the program variables. We have that $\mathsf{init} \in \mathbb{W}(x)$ for every x. The synchronization primitives lock and unlock are interpreted as both a read and a write of a distinguished variable l. Thus, $\mathbb{W}(l) = \mathbb{R}(l) = \{\mathsf{lock}, \mathsf{unlock}\}$.

Essentially, a trace consists of three relations over the program statements, which represent the data and control dependencies from a program execution. The *store order so* represents the ordering of write accesses to each variable, and the *read-from relation rf* (from writes to reads) indicates the assignment that a read receives its value from. The *program order po* represents the ordering of events issued by the same thread. These relations represent a sequentially consistent execution when their union is consistent with the composition of *rf* and *so* (known also as the *conflict relation*).

Definition 1 (Trace). *A* trace of program P *is a tuple* $t = (S, po, so, rf)$ *where* $S \subseteq \mathsf{Stmts}(P)$, $\mathsf{init} \in S$, *and po, so, and rf are binary relations over* S *such that:*

1. *po relates statements included in the same thread,*
2. *so relates statements writing to the same variable, i.e.,* $so \subseteq \bigcup_x ((S \cap \mathbb{W}(x))^2$, *and for each variable* x, *it defines a total order between the writes to* x *where* init *is ordered before all the other writes,*
3. *rf relates writes and reads to the same variable, i.e.,* $rf \subseteq \bigcup_x (S \cap \mathbb{W}(x)) \times (S \cap \mathbb{R}(x))$, *and associates to every read of a variable* x *a write to* x, *i.e., the inverse of rf is a total function from* $S \cap \mathbb{R}(x)$ *to* $S \cap \mathbb{W}(x)$, *and*
4. *the union of po, so, rf, and* $rf \circ so$, *is acyclic.*

For a program P, let $\mathsf{Traces}(P)$ be its set of traces. Figure 4 lists two programs and their sets of traces.

4 Abstracting Traces

We are interested in comparing the set of behaviors of two programs according to *abstract* representations of traces. These representations are defined in two steps. We first define a projection operator that removes a given set of statements (defined by a set of variables), e.g., synchronization primitives, and the program order from all the traces of a given program [6]. Such a projection operator focuses on the differences in cross-thread data-flow interferences, and ignores details that are irrelevant for standard safety specifications (which are agnostic for instance to the state of the synchronization objects). Then, we define an abstract domain

[6] Our framework can be extended such that the projection operator removes only a user-specified fragment of the program order.

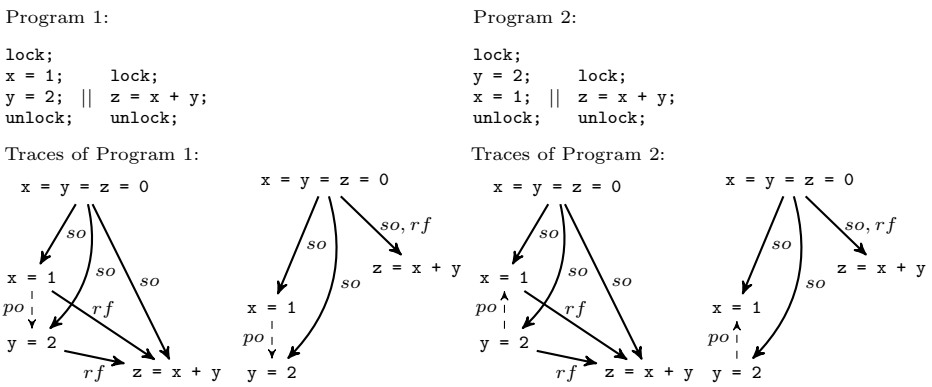

Fig. 4. Two programs over the same set of statements but with different program orders, and different sets of traces. For readability, we write `x=y=z=0` instead of init for the statement that assigns initial values to variables, and we omit `lock`/`unlock` statements.

for representing sets of traces obtained through projection, which is based on a graph structure describing the union of all the traces in a given set.

For a program P, a set V of variables is called *closed* when P doesn't include a statement s that uses both a variable in V and a variable outside of V, i.e., $\mathsf{Vars}(s) \subseteq V$ or $\mathsf{Vars}(s) \cap V = \emptyset$ for each $s \in \mathsf{Stmts}(P)$. For example, in the case of the programs in Fig. 4, the set of variables $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ is closed, and $\{\mathtt{x}, \mathtt{y}\}$ is not closed because of the statement `z := x + y`. For a closed set of variables V, a *V-trace* of P is a tuple $t = (\mathsf{Stmts}(V) \cap S, so, rf)$ obtained from a standard trace $t' = (S, po', so', rf')$ of P by preserving only the statements over the variables in V and removing the program order, i.e., $so = so' \cap (\mathsf{Stmts}(V) \cap S)^2$ and $rf = rf' \cap (\mathsf{Stmts}(V) \cap S)^2$. Since V is closed, the relations so and rf in t satisfy the properties (2) and (3) in Definition 1.

The set of all V-traces of a program P is denoted by $\mathsf{Traces}[V](P)$.

For example, the programs in Fig. 4 have the same set of V-traces where $V = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$. This holds because V-traces don't contain the $\mathsf{lock}/\mathsf{unlock}$ statements and the program order.

We define a parametrized abstraction for a set of V-traces that contains all the statements in those traces, the union of the store order, resp., read-from, relations, and for a parameter k, all the non-singleton sets of so or rf dependencies of size at most k that occur together in the same V-trace. As the parameter k increases, the abstraction is more precise. For two sets A and B, and $k \geq 2$, $\mathcal{P}_k(A, B)$ is the set of pairs (A', B') where $A' \subseteq A$, $B' \subseteq B$, and $2 \leq |A' \cup B'| \leq k$.

Definition 2 (Abstract trace). *For $k \geq 1$, an* abstract trace of rank k *is a tuple $\widehat{t} = (S, \widehat{so}, \widehat{rf}, \widehat{sets})$ where S is a set of statements with $\mathsf{init} \in S$ and*

- *$\widehat{so}$ and $\widehat{rf}$ are two relations over statements in S such that $\widehat{so} \subseteq \bigcup_x (S \cap \mathbb{W}(x))^2$, $\widehat{rf} \subseteq \bigcup_x (S \cap \mathbb{W}(x)) \times (S \cap \mathbb{R}(x))$, and for every variable x,*

  - $\widehat{so}$ *contains* (s_1, s_2) *or* (s_2, s_1)*, for every two assignments* $s_1, s_2 \in S \cap \mathbb{W}(x)$*, and*
  - *every read on* x *is related by* $\widehat{rf}$ *to at least one assignment to* x
- $\widehat{sets} = \emptyset$ *if* $k = 1$*, and* $\widehat{sets} \subseteq \mathcal{P}_k(\widehat{so}, \widehat{rf})$*, otherwise. When* $k \geq 2$*, we assume that* $A_1 \cup B_1 \not\subseteq A_2 \cup B_2$ *for all* $(A_1, B_1), (A_2, B_2) \in \widehat{sets}$*.*

The elements of $\widehat{sets}$ *are called* k*-clusters.*

The relations $\widehat{so}$ and $\widehat{rf}$ represent the union of the store order and read-from relations in a given set of V-traces, respectively. Therefore, $\widehat{so}$ is not necessarily a total order, and the inverse of $\widehat{rf}$ is not necessarily a total function, when considering statements that assign or read the same variable (i.e., they don't satisfy the properties (2) and (3) in Definition 1). Also, to avoid redundancy, we assume that the elements of $\widehat{sets}$ are incomparable. Fig. 1 and Fig. 2 contain examples of abstract traces.

The *concretization* of an abstract trace $\widehat{t}$ of rank k, denoted by $\gamma(\widehat{t})$, is the set of traces formed of some dependencies in $\widehat{t}$ and which contain at least one set of dependencies in $\widehat{sets}$, if $k \geq 2$. Formally, $\gamma(\widehat{t})$ for an abstract trace $\widehat{t} = (S, \widehat{so}, \widehat{rf}, \widehat{sets})$ of rank k is the set of V-traces $t = (S', so, rf)$ where $S' \subseteq S$, $so \subseteq \widehat{so}$, $rf \subseteq \widehat{rf}$, and if $k \geq 2$, then $u|_1 \subseteq so$ and $u|_2 \subseteq rf$ for some $u \in \widehat{sets}$. We use $u|_i$ to denote the i-th component of the tuple u. Note that a trace in the concretization of $\widehat{t}$ may not necessarily use all the statements in $\widehat{t}$.

We define an order relation $\leq$ between abstract traces, which requires that they contain the same set of statements and the "smaller" trace contains less dependencies or sets of dependencies.

Definition 3 (Order relation). *For* $k \geq 1$ *and two abstract traces* $\widehat{t_1} = (S, \widehat{so_1}, \widehat{rf_1}, \widehat{sets_1})$ *and* $\widehat{t_2} = (S, \widehat{so_2}, \widehat{rf_2}, \widehat{sets_2})$ *of rank* k*,*

$$\widehat{t_1} \leq \widehat{t_2} \text{ iff } \widehat{so_1} \subseteq \widehat{so_2},\ \widehat{rf_1} \subseteq \widehat{rf_2},\ \text{and } \widehat{sets_1} \subseteq \widehat{sets_2}.$$

Lemma 1. *The order relation* $\leq$ *defines a lattice over the set of abstract traces.*

5 Interference Refinement

We define a notion of refinement between two programs, called *interference refinement* (or refinement for short), which holds under the assumption that the two programs are structurally similar. Essentially, we assume that there exists a mapping between variables in the two programs, and a mapping between statements, such that every two related statements read and respectively, write the same set of variables (modulo the variable mapping). Then, interference refinement is defined as the inclusion of V-trace sets for some set of variables V (modulo the statement mapping). We then give an abstract notion of interference refinement that uses abstract traces instead of sets of V-traces.

Let P_1 and P_2 be two programs, and V_1 and V_2 closed sets of variables of P_1 and P_2, respectively. A pair (ν, σ) is called a *statement matching* when $\nu : V_1 \to V_2$

is a bijection and $\sigma : \mathsf{Stmts}(P_1) \cap \mathsf{Stmts}(V_1) \to \mathsf{Stmts}(P_2) \cap \mathsf{Stmts}(V_2)$ is a bijection such that $s \in \mathbb{W}(x)$ iff $\sigma(s) \in \mathbb{W}(\nu(x))$ and $s \in \mathbb{R}(x)$ iff $\sigma(s) \in \mathbb{R}(\nu(x))$ for each $s \in \mathsf{Stmts}(P_1) \cap \mathsf{Stmts}(V_1)$ and $x \in V_1$. To simplify the exposition, in the rest of the paper, we consider statement matchings where ν and σ are the identity. Extending our notions to the general case is straightforward.

Let P_1 and P_2 be two programs, and V a set of variables which is closed for both P_1 and P_2.

Definition 4 (V-Refinement). *A program P_1 is a* V-interference refinement *(or V-refinement for short) of another program P_2 iff $\mathsf{Traces}[V](P_1) \subseteq \mathsf{Traces}[V](P_2)$. Also, P_1 and P_2 are* V-interference equivalent *(or V-equivalent for short) iff P_1 is a V-interference refinement of P_2 and vice-versa.*

We define an approximation of V-refinement, called (V,k)-*refinement*, that compares abstract traces of rank k instead of concrete sets of V-traces. More precisely, (V,k)-refinement compares abstract traces that *represent* the V-traces of a program in the following sense: the sets of dependencies in the abstract trace are not spurious, i.e., they do occur together in a concrete V-trace, and the abstract trace contains all the sets of dependencies up to size k that occur in the same V-trace. Forbidding spurious (sets of) dependencies guarantees that V-refinement doesn't hold when the approximated version doesn't hold, while completeness allows to prove that the approximated version does imply V-refinement for big enough values of k.

Definition 5. *An abstract trace $\widehat{t} = (S, \widehat{so}, \widehat{rf}, \widehat{sets})$ of rank k* represents *a program P for a closed set of variables V when*

- *for every two statements $s_1, s_2 \in S$, $(s_1, s_2) \in \widehat{so}$, resp., $(s_1, s_2) \in \widehat{rf}$, iff there exists a V-trace $t = (S', so, rf) \in \mathsf{Traces}[V](P)$ such that $(s_1, s_2) \in so$, resp., $(s_1, s_2) \in rf$, and*
- *if $k \geq 2$, then for each $u \in \mathcal{P}_k(\widehat{so}, \widehat{rf})$, $u \in \widehat{sets}$ iff there exists a V-trace $t = (S', so, rf) \in \mathsf{Traces}[V](P)$ such that $u \in \mathcal{P}_k(so, rf)$.*

For any abstract trace $\widehat{t}$ representing a program P for a closed set of variables V, we have that $\mathsf{Traces}[V](P) \subseteq \gamma(\widehat{t})$.

Definition 6 ((V,k)-Refinement/Equivalence). *A program P_1 is a* (V,k)-refinement *of another program P_2 iff there exist $\widehat{t_1}$ and $\widehat{t_2}$ two abstract traces of rank k representing P_1 and P_2 for the set of variables V, respectively, such that $\widehat{t_1} \leq \widehat{t_2}$. Also, P_1 and P_2 are* (V,k)-equivalent *iff P_1 is a (V,k)-refinement of P_2 and vice-versa.*

When V is understood from the context, we may use refinement of rank k instead of (V,k)-refinement.

Example 1. Distinguishing two programs with respect to the notion of (V,k)-equivalence may require arbitrarily-large values of k (these values are however polynomially bounded by the size of the programs). Indeed, we show that there

exist two programs which are $(V, k-1)$-equivalent but not (V, k)-equivalent, for each $k \geq 2$.

Fig. 5 lists two programs that make k parallel increments to a variable `x`, for an arbitrary $k \geq 2$. The increments are non-atomic in the first program, and protected by a semaphore `s` initialized with $k-1$ permits in the second program (`acquire` acquires a permit from the semaphore, blocking until one is available, while `release` returns one permit to the semaphore) [7]. The first program admits all the executions of the second one and one more execution where all the k threads read the initial value of `x`. Therefore, the first program has a trace that contains the set of read-from dependencies from `init` to each assignment `temp1 = x`,...,`tempk = x` (the k read-from dependencies marked in red in Fig. 5). This is not true for the second program where the semaphore synchronization disallows such a trace.

Let us consider the closed set of variables $V = \{\texttt{x}, \texttt{temp1}, \ldots, \texttt{tempk}\}$. Every set of at most $k-1$ *so* or *rf* dependencies occur together in the same V-trace of one program iff this holds for the other program as well. Therefore, the two programs are $(V, k-1)$-equivalent. However, the two programs are *not* (V, k)-equivalent, more precisely, the first program is not a (V, k)-refinement of the second one. The abstract trace representing the first program contains a k-cluster which is the set of read-from dependencies from `init` to each assignment `temp1 = x`,...,`tempk = x`. □

A direct consequence of the definitions is that V-refinement and (V, k)-refinement coincide for big enough values of k. The number of read-from and respectively, store-order dependencies, in a V-trace is bounded by $|\mathsf{Stmts}(P) \cap \mathsf{Stmts}(V)|^2$. Therefore, there exist at most $2^{2 \cdot |\mathsf{Stmts}(P) \cap \mathsf{Stmts}(V)|^2}$ V-traces, which implies that V-refinement and (V, k)-refinement coincide when k reaches this bound. Otherwise, we have only that V-refinement implies (V, k)-refinement.

Theorem 1. *For every $k \geq 1$, P_1 is a (V, k)-refinement of P_2 when P_1 is a V-refinement of P_2. Moreover, there exists $k \leq 2^{2 \cdot |\mathsf{Stmts}(P) \cap \mathsf{Stmts}(V)|^2}$ such that P_1 is a V-refinement of P_2 iff P_1 is a (V, k)-refinement of P_2.*

6 Checking Interference Refinement

We show that checking whether a program is *not* a (V, k)-refinement of another one, for some closed set of variables V and some $k \geq 1$, is polynomial time reducible to assertion checking. This reduction holds for programs manipulating data coming from arbitrary, not necessarily bounded, domains. Instantiating this reduction to the case of boolean programs, we get that this problem is in Δ_2^P when k is fixed, and in Σ_2^P, otherwise. We show that these upper complexity bounds match the lower bounds. As a corollary, we get that deciding whether a program is *not* a V-refinement of another one is also Σ_2^P-complete.

[7] The simple syntax we considered in Section 3 doesn't include `acquire`/`release` actions, but they can be easily modeled using lock/unlock.

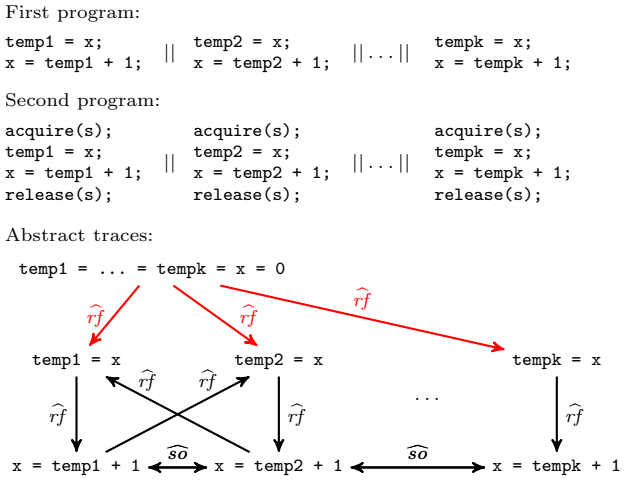

Fig. 5. Two programs doing k parallel increments of `x`. The two programs have the same abstract trace of rank 1 which is partially given in the bottom part of the figure; we omit some of the $\widehat{so}$ dependencies for readability. The abstract trace of rank k of the first program contains a k-cluster which is the set of read-from dependencies marked in red (they occur in the same trace) while this is not true for the second program.

The following intermediary result shows that checking whether a fixed set of data-flow dependencies occur together in some V-trace of a program P is reducible to assertion checking in an instrumentation of P. The instrumentation uses a set of boolean flags to witness the order between two assignments on the same variable, in the case of store order dependencies, or that an assignment on a variable x is the last such assignment before a statement reading the value of x, in the case of read-from dependencies. For instance, let us consider a fragment with three threads of the first program in Fig. 5.

```
temp1 = x;                 temp2 = x;                 [
[                          [                          temp3 = x;
x = temp1 + 1;        ||   x = temp2 + 1;        ||   if ( rf_saw_first &&
rf_saw_first = true;       if ( rf_saw_first &&           !rf_saw_write )
]                               !rf_saw_second )        rf_saw_second = true;
                             rf_saw_write = true;
                           ]                          ]
                                                      x = temp3 + 1;
```

The read-from dependency from the write to `x` in the first thread to the read of `x` in the third thread can be witnessed using three boolean flags `rf_saw_first`, `rf_saw_write`, and `rf_saw_second`, which are initially false and which are updated atomically with the program's statements. The flag `rf_saw_second` is true for all executions whose trace contains this read-from dependency (and only for

these executions) [8]. For readability, we use brackets instead of synchronization primitives to delimit atomic sections.

The flag `rf_saw_first` is set to true when the write in the first thread happens, `rf_saw_write` is set to true when any other write to `x`, i.e., the write to `x` in the second thread, happens after the one in the first thread, and `rf_saw_second` is set to true when the read of `x` in the last thread happens, provided that the write in the first thread was the last write to `x` before this read (which is equivalent to `rf_saw_write` being false). Dealing with store-order dependencies is simpler, it requires only two flags `so_saw_first` and `so_saw_second` to witness that a write happens before another one. Then, witnessing a *set* of data-flow dependencies can be done by adding such flags for each dependency, independently. Note that the placement of the instructions that set or check these flags is only based on syntax and their addition is easy to automate.

In formal terms, let

$$D_{rf} \subseteq \bigcup_x (\mathsf{Stmts}(V) \cap \mathbb{W}(x)) \times (\mathsf{Stmts}(V) \cap \mathbb{R}(x)) \text{ and}$$
$$D_{so} \subseteq \bigcup_x (\mathsf{Stmts}(V) \cap \mathbb{W}(x)) \times (\mathsf{Stmts}(V) \cap \mathbb{W}(x))$$

be two sets of read-from, resp., store-order dependencies, and let $D = D_{rf} \cup D_{so}$. For each $(s_1, s_2) \in D$, P is instrumented with two boolean variables $\texttt{saw\_first}[s_1, s_2]$ and $\texttt{saw\_second}[s_1, s_2]$ such that $\texttt{saw\_first}[s_1, s_2]$ is atomically set to true when s_1 is executed, and $\texttt{saw\_second}[s_1, s_2]$ is atomically set to true when s_2 is executed, provided that $\texttt{saw\_first}[s_1, s_2]$ is already true. Additionally, when $(s_1, s_2) \in D_{rf}$, a variable $\texttt{saw\_write}[s_1, s_2]$ is set to true whenever $\texttt{saw\_first}[s_1, s_2]$ is true, $\texttt{saw\_second}[s_1, s_2]$ is false, and a statement writing to the same variable as s_1 is executed. Also, $\texttt{saw\_second}[s_1, s_2]$ is set to true when additionally, $\texttt{saw\_write}[s_1, s_2]$ is false (this is to ensure that s_1 is the last write before s_2). The instrumented program is denoted by $P[D]$.

Lemma 2. *There exists a V-trace $t = (S, so, rf)$ of P such that $D_{rf} \subseteq rf$ and $D_{so} \subseteq so$ iff $P[D]$ reaches a program configuration where* $\texttt{saw\_second}[s_1, s_2]$ *is* `true` *for all $(s_1, s_2) \in D$.*

For a fixed k, checking (V, k)-refinement needs to consider only fixed size sets of dependencies. Therefore, the following holds.

Theorem 2. *Let P_1 and P_2 be two programs. Checking whether P_1 is* not *a (V, k)-refinement of P_2 is polynomial time reducible to assertion checking.*

Proof. The program P_1 is not a (V, k)-refinement of P_2 iff there exists a set of dependencies D (of size at most k) such that D occur together in some V-trace of P_1, but no V-trace of P_2. Since the number of possible sets D is polynomial

[8] Equivalently, the assignment `rf_saw_second = true` can be replaced by `assert false`. Then, this assertion fails whenever this read-from dependency occurs in some trace of the program.

in the size of P_1 and P_2, a polynomial reduction to assertion checking consists in enumerating all the possible instances of D and checking whether D occurs in the same V-trace of P_1 or P_2 using the result in Lemma 2. □

The algorithm proposed in the proof of Theorem 2 reduces the problem of checking non (V,k)-refinement, for a fixed k, to a polynomial set of assertion checking queries and leads the way to the reuse of the existing safety verification technology. This will be demonstrated in Section 7.

For boolean programs, assertion checking is NP-complete [9], so checking (V,k)-refinement for any k is in Σ_2^P. We show that it is also Σ_2^P-hard.

Theorem 3. *Let P_1 and P_2 be two boolean programs. Checking whether P_1 is* not *a (V,k)-refinement of P_2 is Σ_2^P-complete.*

Proof. A Σ_2^P algorithm for deciding non (V,k)-refinement starts by guessing a set of dependencies D (of size at most $2 \cdot |\mathsf{Stmts}(P) \cap \mathsf{Stmts}(V)|^2$), and then proceeds by checking that the dependencies in D occur in the same V-trace of P_1 (which by Lemma 2 can be decided in NP) and in none of the traces of P_2 (which again by Lemma 2 is in co-NP).

To prove Σ_2^P-hardness we show that deciding the satisfiability of an $\exists^*\forall^*$ boolean formula can be reduced to checking (V,k)-refinement for some k which depends on the number of existential variables in the boolean formula. Let $\exists\vec{x}\ \forall\vec{y}.\ \varphi$ be a boolean formula in prenex normal form (without free variables), where $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{y}$ are vectors of boolean variables. Also, let P_1 and P_2 be the following programs:

Program P_1:

```
x1 = 0 [] x1 = 1;          ||  assume done;
...                            t1 = x1;
xn = 0 [] xn = 1;          ||  ...
y⃗ = ⋆;                         tn = xn;
done = 1;                      b = a;
a = 1;                         assume ¬φ;
```

Program P_2:

```
x1 = 0 [] x1 = 1;          ||  assume done;
...                            t1 = x1;
xn = 0 [] xn = 1;          ||  ...
y⃗ = ⋆;                         tn = xn;
done = 1;                      assume ¬φ;
a = 1;                         b = a;
```

We assume that all variables are 0 in the initial state. Let $D_{\vec{x}}$ be a set of read-from dependencies that includes either $(x_i = 0, t_i = x_i)$ or $(x_i = 1, t_i = x_i)$ for each $1 \leq i \leq n$. Then, let $D = D_{\vec{x}} \cup \{(\mathtt{a} = 1, \mathtt{b} = \mathtt{a})\}$ (the latter is also in rf).

Since the assignment `b = a` in P_1 is executed in every complete interleaving, there exists a trace of P_1 that contains all the read-from dependencies from D. This set of dependencies occurs in a trace of P_2 only if there exists some valuation for $\vec{y}$ such that φ is `false`. This implies that P_1 is not a $(V, n+1)$-refinement of P_2 where V is the set of all variables of P_1 iff $\exists\vec{x}\ \forall\vec{y}.\ \varphi$ is satisfiable. □

Following the same lines of Theorem 3, we can show that the problem of checking non (V,k)-refinement becomes Δ_2^P-complete when k is fixed. Essentially, the set of dependencies that need to be tracked are now of fixed size and they can be enumerated explicitly (as stated in Theorem 2).

[9] Recall that we consider programs without looping constructs and procedure calls.

Theorem 4. *Let P_1 and P_2 be two boolean programs. For a fixed but arbitrary $k \geq 1$, checking whether P_1 is* not *a (V, k)-refinement of P_2 is Δ_2^P-complete.*

Proof. The problem can be decided using a similar algorithm as in Theorem 3. Instead of non-deterministically guessing the set of dependencies D, we enumerate all such sets of dependencies of size k which are at most $O(|\mathsf{Stmts}(P) \cap \mathsf{Stmts}(V)|^{2 \cdot k})$ many.

To prove Δ_2^P-hardness we show that deciding the satisfiability of an $\exists^* \wedge \forall^*$ boolean formula can be reduced to checking $(V, 1)$-refinement. Let $\exists \vec{x}.\ \varphi_1 \wedge \forall \vec{y}.\ \varphi_2$ be a boolean formula (without free variables), where $\vec{x}$ and $\vec{y}$ are vectors of boolean variables. Also, let P_1 and P_2 be the following programs:

Program P_1:

```
x⃗ = ⋆;                          assume (done && φ1);
a = 1;     ||                   b = a;
done = 1;
```

Program P_2:

```
y⃗ := ⋆;                         assume (done && ¬φ2);
a = 1;     ||                   b = a;
done = 1;
```

We assume that all variables are 0 in the initial state. Let $V = \{\mathtt{a}, \mathtt{b}\}$ and $D = \{(\mathtt{a} = 1, \mathtt{b} = \mathtt{a})\}$ be a singleton set of read-from dependencies.

The assignment `b = a` in P_1 is executed if and only if there exists some valuation for $\vec{x}$ such that φ_1 holds, i.e., the formula $\exists \vec{x}.\ \varphi_1$ is satisfiable. Therefore, the dependency $(\mathtt{a} = 1, \mathtt{b} = \mathtt{a})$ occurs in a trace of P_1 iff $\exists \vec{x}.\ \varphi_1$ is satisfiable. By the definition of V, this is the only dependency possible in P_1, which may imply non $(V, 1)$-refinement. Furthermore, this dependency doesn't occur in a trace of P_2 if and only if the formula φ_2 holds for all valuations of $\vec{y}$, i.e., the formula $\forall \vec{y}.\ \varphi_2$ is satisfiable. Consequently, P_1 is not a $(V, 1)$-refinement of P_2 iff $\exists \vec{x}.\ \varphi_1 \wedge \forall \vec{y}.\ \varphi_2$ is satisfiable. □

7 Experimental Evaluation

To demonstrate the practical value of our approach, we argue that our notion of (V, k)-refinement:

- can be checked using the existing verification technology,
- witnesses for semantic differences (bug introduction) with small values of k,
- enables succinct representations for the semantic difference,
- is a relevant indicator of regression-freeness.

To argue these points, we consider a set of bug fixes produced by the ConcurrencySwapper synthesis tool [6] which model concurrency bug fixes for Linux device drivers reported at `www.kernel.org` [10]. We check whether the fixed version is a (V, k)-refinement of the original one and vice-versa. We use this benchmark without modifications, except the use of the `pthread` library for managing threads (otherwise, the programs are written in C), and unfolding loops once.

We have added the annotation that reduces (V, k)-refinement checking to assertion checking (explained in Theorem 2) and used LazyCseq [13, 12] (with

[10] They are available at `https://github.com/thorstent/ConcurrencySwapper`

Name	#loc	#threads	k	# (sets of) possible dependencies	size of the difference	Time
r8169-1	24	2	1/2/3	6/21/41	1/5/11	6.35s/12.93s/20.27s
r8169-2	25	2	1/2/3	6/21/41	1/5/11	4.93s/10.22s/16.44s
r8169-3	33	3	1/2/3	3/6/7	1/3/3	2.74s/5.43s/8.03s
i2c-hid	27	2	1	27	2	45.65s
i2c-hid-noA	27	2	1/2	27/237	0/4	42.34s/24.3m
rtl8169	256	7	1	94	3	37.27m

Table 1. Experimental data for checking (V, k)-refinement. The size of the difference between the abstract trace of the original (buggy) and the fixed version, respectively, is the number of (sets of) dependencies occurring in one and not the other.

backend CBMC [9]) for checking the assertions. LazyCseq is a bounded model checker that explores round-robin schedules up to a given bound on the number of rounds. We have used a bound of 4 for the number of rounds, which was enough to compute abstract traces that represent the considered programs (according to Definition 5). We have checked manually that these abstract traces are complete, i.e., that they contain all the sets of dependencies which occur in the same V-trace (up to the given bound). The fact that they don't contain spurious sets of dependencies is implied by the completeness of the bounded model checker. All the bug fixes except `i2c-hid` and `i2c-hid-noA` that consist in adding locks, are based on statement reordering [11]. This allowed us to consider closed sets of variables that consist of all variables except variables of type lock, and statement matchings (ν, σ) where ν and σ are the identity.

The results are reported in Table 1. Each line corresponds to a pair of programs, the version before and after a bug fix or a set of bug fixes implemented during the evolution of a Linux driver, r8169, i2c-hid, or rtl8169. We list the number of lines of code (loc) and the number of threads of the original version (before the bug fix). Checking refinement of rank 1 requires enumerating all pairs of statements accessing the same variable, at least one being a write, called *possible dependencies*, and verifying whether they occur in some execution of the original or the fixed version. To indicate the difficultly of the benchmark we give the number of such possible dependencies, or sets of possible dependencies of size at most k, when $k > 1$. Note that the number of possible dependencies is usually much smaller than the square of the number of statements. All measurements were made on a MacBook Pro 2.5GHz Intel Core i7 machine.

We consider several values of k for each example and in all cases we get that the fixed version is a refinement of rank k of the buggy version. Also, except for `i2c-hid-noA` with $k = 1$, the abstract trace of the correct version is *strictly* smaller than the one of the buggy version. The `i2c-hid` example contains some assertions that fail only in the buggy version. These assertions participate in read-from dependencies which allow to distinguish the buggy from the corrected

[11] Studies of concurrency errors, e.g., [6, 19], have reported that reordering statements for fixing bugs is very frequent (around 30% of the fixes are based on reorderings).

version with abstract traces of rank 1. Removing these assertions requires abstract traces of rank 2 to distinguish the two versions. This fact is demonstrated in the `i2c-hid-noA` example which is exactly `i2c-hid` without those assertions.

These results indicate that comparing abstract traces of small ranks is enough to reveal interesting behaviors, in particular bugs (the abstract trace of the buggy version is always different from the one of the corrected version). Therefore, (V, k)-refinement for small values of k is a relevant indicator of regression-freeness. Note however that there is no theoretical connection between abstract trace difference and the presence of bugs. Moreover, (V, k)-refinement continues to hold when k is increased, as shown by the results in Table 1.

The difference between the abstract traces of the original and the fixed version, respectively, consists of few (sets of) dependencies. For the first three examples and $k = 1$, the difference consists of a single read-from dependency showing that a particular variable gets an uninitialized or undesired value (like in the example from Figure 1). In the case of the fourth example when assertions are present, the difference between abstract traces of rank 1 consists of 2 read-from dependencies which correspond to two failing assertions. When assertions are removed, i.e., in the example `i2c-hid-noA`, the difference between the abstract traces of rank 2 consists of few pairs of dependencies similar to the example in Figure 2. The buggy version of the example `rtl8169` contains 3 bugs that are repaired in the correct version. The difference between the abstract traces contains an explanation for each bug.

The running time increases with the number of threads and possible dependencies. However, since the presence of a set of dependencies (in some execution) reduces to an independently-checkable assertion, the verification process is easily parallelizable. Also, we didn't use assertion checking to exclude some dependencies that are obviously not feasible because of thread creation/join (i.e., reading from a write that belongs to a thread not yet created). As future work, we plan to investigate static analyses for filtering out such dependencies.

8 Related work

The work on refinement checking [1] provides a general framework for comparing traces of two programs. However, in most instances one of the programs serves as a specification with very limited concurrency.

Joshi et al. [15] checks if a given concurrent program fails an assertion more often on an input compared to another concurrent program — the second program is usually limited to sequential interleavings only. Our approach does not require the presence of assertions to compare the two concurrent programs as it exploits the structural similarity between the two programs. The work closest to ours is the work on regression verification for multi-threaded programs [8]. This paper proposes a proof rule to prove that the input-output relations for two multithreaded programs are the same. This approach cannot distinguish between two transformations that introduce and respectively, remove a bug. In both cases, the proof rule will fail to establish equivalence w.r.t. the input-output relation.

Generalizations of good or bad program executions using partial orders have been previously used in the context of assertion checking or program synthesis [6, 7, 10]. The notion of trace robustness proposed in the context of weak memory models [4, 5] compares a program running under a weak memory model with the same program running under Sequential Consistency (SC). The focus there is to check if a program admits behaviors which are not possible under SC while our goal is to compare two programs running under SC.

There has been interest in applying program analysis towards the problem of comparing two versions of a program, in the context of sequential programs. Jackson and Ladd [14] used the term *semantic diff* to compare two sequential programs in terms of the dependency between input and output variables. For most concurrency related transformations, such a metric is unlikely to yield any difference. There has been work on equivalence checking of sequential executions across program versions using uninterpreted function abstraction and program verifiers [11, 16]. Verification Modulo Versions [17, 18] compares two sequential programs w.r.t. a set of assertions. Differential symbolic execution [21] summarizes differences in summaries of two procedures, and Marinescu et al. [20] use symbolic execution for generating tests over program differences.

9 Conclusions

We have presented an approach for comparing two closely related concurrent programs whose goal is to give feedback about interesting differences, without relying on user-provided assertions. This approach is based on comparing abstract representations of the data-flow dependencies admitted by two subsequent versions of the same program. This comparison is reducible to assertion checking which enables the reuse of the existing verification technology.

As future work, we plan to investigate static analyses for discarding data-flow dependencies which are not interesting or not feasible. This can be also used to minimize the number of assertion checking queries when checking (V, k)-refinement. Moreover, we consider extending our theory to programs that contain loops where the main difficulty is that traces contain an unbounded number of copies of the same statement (when inside a loop). The idea would be define a new abstraction of traces that collapses together occurrences of the same statement from multiple iterations of a loop. On the practical side, we aim at a more thorough experimental evaluation of this approach in the context of other program transformations. On one side, we plan to consider more general program edits than reordering statements or modifying synchronization primitives which need to consider more general statement matchings than the identity. Also, we plan to investigate other classes of program transformations besides bug-fixing, such as refactoring, addition of new features or performance fixes. For instance, in the context of performance fixes, the new version of the program may allow more behaviors (interleavings). Our approach would produce a succinct representation of the new behaviors (in terms of small sets of dependencies), which may help in validating their correctness.

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[3] Armin Biere and Roderick Bloem, editors. *Computer Aided Verification - CAV 2014, Vienna, Austria, 2014. Proceedings*, volume 8559 of *LNCS*. Springer, 2014.

[4] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Robustness against relaxed memory models. In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Software Engineering 2014, Kiel, Deutschland*, volume 227 of *LNI*, pages 85–86. GI, 2014.

[5] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *CAV 2008*, pages 107–120, 2008.

[6] Pavol Cerný, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - CAV 2013, Saint Petersburg, 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 951–967. Springer, 2013.

[7] Pavol Cerný, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Regression-free synthesis for concurrency. In Biere and Bloem [3], pages 568–584.

[8] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Regression verification for multi-threaded programs (with extensions to locks and dynamic thread creation). *Formal Methods in System Design*, 47(3):287–301, 2015.

[9] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[10] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Inductive data flow graphs. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 129–142. ACM, 2013.

[11] Benny Godlin and Ofer Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Inf.*, 45(6):403–439, 2008.

[12] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In Myra B. Cohen, Lars Grunske, and Michael Whalen, editors, *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 807–812. IEEE Computer Society, 2015.

[13] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In Biere and Bloem [3], pages 585–602.

[14] Daniel Jackson and David A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance, ICSM 1994, Victoria, BC, Canada, September 1994*, pages 243–252. IEEE Computer Society, 1994.

[15] Saurabh Joshi, Shuvendu K. Lahiri, and Akash Lal. Underspecified harnesses and interleaved bugs. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 19–30. ACM, 2012.

[16] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, 2012.

[17] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 345–355. ACM, 2013.

[18] Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, and Sam Blackshear. Verification modulo versions: towards usable verification. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 32. ACM, 2014.

[19] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Susan J. Eggers and James R. Larus, editors, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 329–339. ACM, 2008.

[20] Paul Dan Marinescu and Cristian Cadar. KATCH: high-coverage testing of software patches. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 235–245. ACM, 2013.

[21] Suzette Person, Matthew B. Dwyer, Sebastian G. Elbaum, and Corina S. Pasareanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 226–237. ACM, 2008.

[22] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.