Property Directed Reachability for Proving Absence of Concurrent Modification Errors

Asya Frumkin[1], Yotam M. Y. Feldman[1], Ondřej Lhoták[2], Oded Padon[1], Mooly Sagiv[1], and Sharon Shoham[1]

[1] Tel Aviv University, Tel Aviv, Israel
[2] University of Waterloo, Waterloo, Canada

Abstract. We define and implement an interprocedural analysis for automatically checking safety of recursive programs with an unbounded state space. The main idea is to infer modular universally quantified inductive invariants in the form of procedure summaries that are sufficient to prove the safety property. We assume that the effect of the atomic commands of the program can be modeled via effectively propositional logic. We then propose a variant of the IC3/PDR approach for computing universally quantified inductive procedure summaries that overapproximate the behavior of the program.

We show that Java programs that manipulate collections and iterators can be modeled in effectively propositional logic and that the invariants are often universal. This allows us to apply the new analysis to prove the absence of concurrent modification exceptions in Java programs. In order to check the feasibility of our method, we implemented our analysis on top of Z3, as well as a Java front-end which translates Java programs into effectively propositional formulas.

1 Introduction

Java programs enforce consistency of iterator usage by requiring the absence of concurrent modification exceptions (CME). Intuitively, the idea is to forbid accessing a collection via *stale* iterators. An iterator becomes stale when the collection it iterates is changed not via the iterator itself. The Java standard library imposes this restriction at runtime by throwing a runtime exception when stale iterators are accessed. Note that this can happen both in sequential and concurrent programs. A common example is when adding an element to a collection from inside a loop which iterates it.

In many cases a logical error in the program leads to a CME. Therefore, identifying potential CMEs at compile-time and proving the absence of concurrent modifications can be very useful. Indeed abstract interpretation has been used to prove the absence of CMEs in small to medium size programs [19, 23, 16]. These methods are sound, i.e., whenever the absence of CMEs is proved, the program indeed cannot raise a CME. However, these methods are incomplete, and may result in false alarms due to limitations of the abstraction. Common sources for imprecision are aliasing of objects in the heap, and complex interaction between various procedures of the program.

1.1 Main Results

Our key insight in this paper is that in many programs the inductive invariants required to prove the absence of CMEs can be expressed as universal first-order formulas and that the problem of checking inductiveness amounts to checking (un)satisfiability in effectively propositional logic. This is surprising since the natural modeling of this problem involves version numbers for iterators. It implies that SAT solvers can be used to check the absence of CMEs for programs annotated with inductive invariants.

In practice it is very hard for programmers to write inductive invariants. Furthermore, Java programs include deep nesting of method calls which makes it practically impossible. Therefore, we implemented a method for automatically inferring universal procedure summaries that applies to recursive programs.

We develop an iterative method for inferring the required procedure summaries for proving absence of CMEs. Technically, we combined the procedure of inferring universally quantified invariants in a property guided way [11] with the techniques of [7, 13] for computing procedure summaries.

This paper can be summarized as follows:

- We show that in many cases inductive invariants for proving absence of CMEs are expressible by universally quantifying over all the iterator objects in the program (whose number is unbounded). Specifically, for programs with (potentially recursive) procedures, procedure summaries (Hoare triples) can also be expressed in universal first order logic. This enables to mechanically check the inductiveness in a sound and complete way using EPR solvers [9].
- We develop an iterative method for inferring sufficiently strong universal procedure summaries in order to prove absence of CMEs. The method handles Java programs with recursive procedures and infers procedure summaries, i.e., Hoare triples. Technically this algorithm generalizes [11] and follows the idea of property directed reachability [7, 13] .
- We implemented the algorithm on top of Ivy [17] and Z3 [4].
- We implemented a front-end for Java using Soot [22] which converts Java programs into EPR and integrates with the above procedure.

2 The Concurrent Modification Problem

The Java Collections Framework (JCF) is an important part of the Java platform as it provides implementations for common collection data structures. An Iterator object is used in order to access the elements in a collection in a sequential manner. Multiple iterators can operate on the same collection. In general, it is an error to modify a collection while an iterator is operating on it and to continue to use the iterator after the modification. Iterators are usually implemented to be fail-fast, meaning that they throw an exception if iteration is resumed after the occurrence of a change in a collection, either directly or via another iterator. The exception, of type ConcurrentModificationException (CME), can be thrown dynamically even in a single-threaded program. All JCF non-concurrent collections provide fail-fast iterators as a safety measure for a very common bug.

In order to prove that a Java program cannot cause a CME to be thrown, a careful examination of the program should be performed. The proof should ensure that all possible execution paths do not cause such an error. Specifically, we call iterators whose collection is modified invalid or stale, and verify that no such iterator is used.

Example 1. Let us consider the code presented in Fig. 1, which contains a class manipulating a list of lists. The method *flatten* transforms a list of lists into a simple list by adding all the items held in the nested lists, to an output list, using the helper method *addList*. Most of the operations in the *main* method are ones that cannot cause a CME. The only location where a collection is modified during iteration is inside the loop in the *addList* method, which is called from *flatten*. The active iterators during this operation, *itr1* and *itr2*, are iterating over the input list *in* while items are being added to the output list *out*. Since in this program *flatten* is called from *main* method with two distinct lists as arguments, none of the iterators becomes invalidated, and the code terminates successfully. However, if the *flatten* method is examined separately from the whole program context, we can conclude that a CME throw is possible due to aliasing. If, for example, the *in* list is equal to the *out* list, both *itr1* and *itr2* would be invalidated inside *addList*'s loop, and the next *itr1.next()* call would throw an exception. Also, if the *out* list is included as one of the lists in the *in* list, *itr2* would be invalidated once *itr1* reaches *out* and the object pointed to by *itr2* is added to *out*. A subsequent *itr2.next()* call would throw an exception. This example demonstrates that both interprocedural analysis and aliasing information is required in order to prove that a CME does not occur in a program.

```
class ListOfLists {
  public static void
        flatten(List in, List out) {
    Iterator itr1 = in.iterator();
    while(itr1.hasNext()) {
      List l = (List) itr1.next();
      addList(out, l);
    }
  }

  public static void
       addList(List lst1, List lst2) {
    Iterator itr2 = lst2.iterator();
    while(itr2.hasNext()) {
      Object o = itr2.next();
      lst1.add(o);
    }
  }
}
```

```
class FlattenTest {
  public static void
        main(String args[]) {
    int length =
        Integer.parseInt(args[0]);
    // create list
    LinkedList ll = new LinkedList();
    LinkedList subl =
        new LinkedList();
    for(int j = 0; j < length; ++j) {
        subl.add(j*j);
    }
    ll.add(subl);
    ll.add(subl);
    List out = new LinkedList();
    ListOfLists.flatten(ll, out);
  }
}
```

Fig. 1. Usage of list of lists implementation that does not cause a CME to be thrown

Implementation of iterators in Java. The natural way to implement a check for CMEs in a Java program is by maintaining a version number for every collection. Each iterator

of a collection is initialized with the current version number of the collection. Every modification of the collection increments its version number, as well as the version number of the iterator performing it, if such exists. When an iterator is accessing or modifying the underlying collection, the iterator's version number is compared with the collection's version number. A difference between these values implies, that the collection was modified by another iterator, and the operation should not be completed.

This method, implemented as part of the runtime engine in JCF, involves maintaining an integer value for the version number and performing arithmetic operations, such as increment and comparison. However, in spite of the use of integer arithmetic in the implementation, it is possible to analyze CMEs without directly modeling the version numbers. As seen in [19], it suffices that the analysis tracks the Boolean notion of whether an iterator is stale or not. Thus EPR is an appropriate choice for encoding the CME problem.

3 EPR Verification Conditions for CME

In this section we present a description of Java programs that involve direct iterator and collection manipulation. We describe how to encode these programs and their CME verification conditions via EPR formulas.

EPR. The effectively-propositional (EPR) class of logical formulas, also known as Bernays-Schönfinkel-Ramsey class, is a decidable fragment of first-order logic. EPR formulas are of the form $\exists^*\forall^*\phi$ where ϕ is a quantifier-free formula, that contains relation symbols and equality, but no function symbols. The satisfiability of such formulas can be reduced to SAT by substituting the existential variables by Skolem constants and then replacing the universally quantified variables by all possible combinations of constants. The resulting formula is propositional, and is exponentially larger than the original formula. We use an extension of EPR, that allows *stratified* function symbols in the vocabulary, i.e. functions that obey the requirement: if there is a function mapping sort srt_1 to sort srt_2, there cannot be a function mapping srt_2 to srt_1. Extended-EPR formulas maintain the decidability of EPR [17]. In the sequel, we will use the term EPR to refer to the extended EPR fragment.

3.1 Program state

To facilitate CME verification, our model for Java programs partitions Java objects into three distinct types: *container*, *iterator* and *obj*. The container type includes object types that are part of the JCF, meaning classes that implement java.util.Collections or java.util.Map interfaces. The iterator type is used for iterator objects, whose class implements the java.util.Iterator interface. All other Java types, including primitive types, are modeled as obj.

Each iterator object *i* has a mapping *cnt(i)* to the container it operates on. A container can hold either objects or containers. The binary relations *member* and *member_cnt* represent an item's presence in a container. For each type X, there is a corresponding unary *used_X* relation that holds initialized items. The state of the

iterator is kept in the unary relations *stale* and *cme*, where the first keeps iterators that are invalid, and the second — iterators on which a CME would be thrown. In addition, for clarity, we use *points_to* and *points_to_cnt* as shorthands, defined by $points\_to(i, o) \stackrel{\text{def}}{=} \exists c_0.\ (cnt(i) = c_0) \wedge member(c_0, o)$ and $points\_to\_cnt(i, c) \stackrel{\text{def}}{=} \exists c_0.\ (cnt(i) = c_0) \wedge member\_cnt(c_0, c)$.

The constructs mentioned above are listed in Table 1 and used as building blocks for implementing a "library" for modeling programs that manipulate collections via iterators. This library supports actions such as add/remove to/from collection, iterator generation, iterator advancement etc. (see Table 2).

In addition to the above mentioned relations we allow program specific binary relations for class fields, where the first argument is an object and the second is the relevant field. We denote the set of all relations used to express the program state as $\mathcal{G}$.

Table 1. Abstraction relations and functions.

Name	Parameters	Description
$cnt(I)$	I: iterator	function which returns a container, maps iterators to their underlying containers
$member(C, O)$	C: container, O: obj	object is included in container
$member\_cnt(C_1, C_2)$	C_1: container, C_2: container	container is included in container
$used\_cnt(C)$	C: container	container is initialized
$used\_iter(I)$	I: iterator	iterator is initialized
$used\_obj(O)$	O: obj	object is initialized
$stale(I)$	I: iterator	iterator is invalid
$cme(I)$	I: iterator	concurrent modification occurred

Safety. The safety requirement of the CME problem is that a concurrent modification exception does not occur in any possible execution of the program. In our formulation, the requirement that a program state does not exhibit a CME violation is expressed by the formula $\forall I.\ \neg cme(I)$. Initially, the cme and $stale$ relations are empty.

3.2 Modeling Java operations in EPR

We provide the interpretation of each Java operation via a two-vocabulary EPR formula, expressing the connection between the pre- and post-states of the operation. The vocabulary $\mathcal{G}$ is used in order to express the pre-state, and $\mathcal{G}' = \{v' \mid v \in \mathcal{G}\}$ is the vocabulary used to express the post-state. Both $\mathcal{G}$ and $\mathcal{G}'$ include only global relations, and the encoding of each primitive operation via an EPR formula is given in Table 2.

Java's Iterator interface defines three methods: *hasNext*, *next* and *remove* (which is optional). An important note is that we do not model the current position of iterators,

but only the container they are pointing to, along with the state of the iterator that we represent by the *stale* and *cme* relations. Therefore, the *hasNext* method is not modeled directly. Our collection is unordered and hence, the *next* method is conservatively modeled as a nondeterministic retrieval of any item from the collection, that is, an item from *points_to*. The retrieved item can be either obj or container, depending on the type of the holding container. The operation also checks whether the iterator is in *stale* state and updates *cme* accordingly, simulating a throw of CME. The *remove* method is modeled by updates of the *member* and *member_cnt* relations as the iterator may be traversing either a collection of objects or a collection of collections. This operation also updates the *stale* relation to mark all iterators traversing the current container as stale, except for the current iterator. The *add* method is modeled by inserting the desired object to the *member*/*member_cnt* relations and updating *stale* to hold all iterators for the updated container. Fresh items of type X are created by updating the *used_X* relations of items that were not used yet. New iterators are created with *stale* set to false. [3] We note, that in addition to the abstraction involved in making collections unordered, we also do not model arithmetic operations and specific data properties, and conservatively overapproximate them via nondeterminism.

3.3 Modeling programs using EPR

From the formulas encoding the primitive operations, we derive a modular symbolic representation of programs by representing each procedure separately, based on the functional approach to interprocedural analysis [21]. For simplicity of the presentation, we consider only procedures without loops; if needed, loops can be transformed into recursive procedures.

Our definitions are inspired by [14]. A program $\mathcal{A}$ is a pair $\langle \Pi, \mathcal{G} \rangle$, where Π is a non-empty set of procedures with a designated procedure $\mathcal{M}$ (*main*) serving as the entry point. $\mathcal{G}$ is a set of predicate symbols representing the program's global state. For simplicity, we assume there are no global program variables. A procedure $\mathcal{P}$ is a tuple $\langle i_{\mathcal{P}}, o_{\mathcal{P}}, \Sigma_{\mathcal{P}}, \beta_{\mathcal{P}} \rangle$, where $i_{\mathcal{P}}$ and $o_{\mathcal{P}}$ are lists of constant symbols denoting the formal parameters and formal output variables, respectively, with the assumption that $i_{\mathcal{P}} \cap o_{\mathcal{P}} = \emptyset$.

$\Sigma_{\mathcal{P}}$ is a second-order predicate of arity $|i_{\mathcal{P}} \cup o_{\mathcal{P}} \cup \mathcal{G} \cup \mathcal{G}'|$ that represents the behavior of $\mathcal{P}$ when reasoning about the behavior of procedures that call $\mathcal{P}$. It is used as a placeholder for a description of the behavior of $\mathcal{P}$.

The method body $\beta_{\mathcal{P}}$ expresses the behavior of the procedure body of $\mathcal{P}$ w.r.t. the behavior of callee procedures by referring to the second-order predicates $\Sigma_{\mathcal{Q}}$ for every procedure $\mathcal{Q}$ called from $\mathcal{P}$.

$\beta_{\mathcal{P}}$ is an EPR formula defined over a vocabulary that consists of the following:

- $\mathcal{G}, \mathcal{G}'$ which are used to represent the global state before and after the execution of procedure $\mathcal{P}$.

[3] The translation also incorporates the fact that once a CME occurs, the normal control-flow of the program is interrupted by the exception.

Table 2. Java statements and their EPR interpretation. c denotes a container, i an iterator, and o an obj.

Java statement	Interpretation
$i.hasNext()$	–
$o = i.next()$	$points\_to(i, o) \wedge (\forall I.\ cme'(I) \iff cme(I) \vee (I = i \wedge stale(i)))$
$c = i.next()$	$points\_to\_cnt(i, c) \wedge (\forall I.\ cme'(I) \iff cme(I) \vee (I = i \wedge stale(i)))$
$c.add(o)$	$\forall C, O, I.\ (member'(C, O) \iff member(C, O) \vee (C = c \wedge O = o)) \wedge\ (stale'(I) \iff stale(I) \vee (cnt(I) = c \wedge used\_itr(I)))$
$c_1.add(c_2)$	$\forall C_1, C_2, I.\ (member\_cnt'(C_1, C_2) \iff member\_cnt(C_1, C_2) \vee (C_1 = c_1 \wedge C_2 = c_2)) \wedge (stale'(I) \iff stale(I) \vee (cnt(I) = c_1 \wedge used\_itr(I)))$
$i.remove()$	$\exists o, c.\ \forall C, C', O, I.$ $((used\_obj(o) \wedge (member'(C, O) \iff member(C, O) \wedge (C \neq cnt(i) \vee O \neq o)))$ $\vee (used\_cnt(c) \wedge (member\_cnt'(C, C') \iff member\_cnt(C, C') \wedge (C \neq cnt(i) \vee C' \neq c))))$ $\wedge (stale'(I) \iff stale(I) \vee (cnt(I) = cnt(i) \wedge I \neq i \wedge used\_itr(I)))$
$c = new\ Collection()$	$\neg used\_cnt(c) \wedge (\forall I.\ used\_itr(I) \Rightarrow (cnt(I) \neq c)) \wedge (\forall O.\ \neg member(c, O)) \wedge (\forall C.\ \neg member\_cnt(c, C) \wedge \neg member\_cnt(C, c)) \wedge (\forall C.\ used\_cnt'(C) \iff used\_cnt(C) \vee C = c)$
$i = c.iterator()$	$\neg used\_itr(i) \wedge (cnt(i) = c) \wedge (\forall I.\ stale'(I) \iff stale(I) \wedge I \neq i) \wedge (\forall I.\ used\_itr'(I) \iff used\_itr(I) \vee I = i)$
$o = new\ Object()$	$\neg used\_obj(o) \wedge\ (\forall C.\ \neg member(C, o)) \wedge (\forall O.\ used\_obj'(O) \iff used\_obj(O) \vee O = o)$

– Pairs of vocabularies $\mathcal{G}_{cs}$, $\mathcal{G}'_{cs}$ for every call site *cs* in $\mathcal{P}$, and the second-order predicate symbol $\Sigma_{\mathcal{Q}}$ for every procedure $\mathcal{Q}$ called by $\mathcal{P}$. In case of consecutive call sites cs_1, cs_2, the vocabularies $\mathcal{G}'_{cs_1}$ and $\mathcal{G}_{cs_2}$ will coincide.
– $i_{\mathcal{P}}$ which represents the formal parameters of $\mathcal{P}$, and $o_{\mathcal{P}}$ which represents the formal returns of $\mathcal{P}$.

The second-order predicates in $\beta_{\mathcal{P}}$ appear only positively and in the following form: let *cs* be a call-site in $\mathcal{P}$ that invokes $\mathcal{Q}$. In $\beta_{\mathcal{P}}$ this is expressed by $\Sigma_{\mathcal{Q}}(i_{cs}, o_{cs}, \mathcal{G}_{cs}, \mathcal{G}'_{cs})$, where i_{cs} are the actual parameters of the call and o_{cs} are its actual returns, which are either constants ($i_{\mathcal{P}}, o_{\mathcal{P}}$) or quantified variables (typically existentially quantified which represent local variables of $\mathcal{P}$).

Recall that the body of $\mathcal{P}$ consists of sequential code. Hence, such a formulation is derived in a straightforward way.

```
main() {
       R(a) = true;
       S(b) = false;
       f(a, b);
       f(b, a);
}
```

Fig. 2. A small program with procedure calls

Example 2. Let us consider the example in Figure 2. The program has two unary global relations *R*, *S* and an entry procedure *main*, that includes two local variables *a* and *b*. Procedure *f*, whose code is omitted, has two formal parameters and no formal returns. *main* procedure calls *f* twice, once with *a* serving as first argument and *b* as second, and then with *b* as first argument and *a* as second. The body of procedure *main* is given by the formula:

$$\beta_{\mathcal{M}} = \exists a, b.\ R'(a) \wedge \neg S'(b) \wedge \Sigma_f(\langle a, b\rangle, \emptyset, \langle R', S'\rangle, \langle R'', S''\rangle) \wedge \Sigma_f(\langle b, a\rangle, \emptyset, \langle R'', S''\rangle, \langle R''', S'''\rangle)$$

Note, that $\beta_{\mathcal{M}}$ includes 4 lists of global variables: $\langle R, S\rangle$, $\langle R', S'\rangle$, $\langle R'', S''\rangle$, $\langle R''', S'''\rangle$ (R, S are not constrained by this formula).

3.4 Verification conditions

The accurate semantics of $\beta_{\mathcal{P}}$ has a least-fixed-point characterization as described in [14]. We omit the semantics definition and provide only the verification conditions for safety properties.

In order to allow verification of interprocedural programs we will use *procedure summaries*.

Procedure summaries. The summary $\mathcal{S}_\mathcal{P}$ of a procedure $\mathcal{P}$ overapproximates the input/output relation of the procedure. In our setting, it is provided by a universal first-order formula over a vocabulary that consists of two copies of the global relations $\mathcal{G}, \mathcal{G}'$: one for the pre-state and one for the post-state, the formal arguments $i_\mathcal{P}$ and the formal returns $o_\mathcal{P}$.

Given the set of procedure summaries of a program $\mathcal{S}_\Pi$ we can now describe the behavior of procedure $\mathcal{P}$ w.r.t. the summaries $\mathcal{S}_\Pi$. Formally, let $\mathcal{S}_\Pi$ be a function mapping every procedure $\mathcal{Q}$ to a summary. When $\mathcal{S}_\Pi$ is clear from the context, we use $\mathcal{S}_\mathcal{Q}$ as a shorthand for $\mathcal{S}_\Pi(\mathcal{Q})$. Let $\mathcal{CS}$ be the set of all $(cs, \mathcal{Q})$ where cs is call-site in $\mathcal{P}$ that invokes $\mathcal{Q}$. Note that if $\mathcal{P}$ is recursive then $\mathcal{Q}$ might be $\mathcal{P}$ itself.

The behavior of $\mathcal{P}$ w.r.t. the summaries $\mathcal{S}_\Pi$ is obtained by replacing every predicate symbol $\Sigma_\mathcal{Q}$ in $\beta_\mathcal{P}$ with $\mathcal{S}_\mathcal{Q}$, applied over the vocabulary determined by the call-site. Formally, it is captured by the formula $\beta_\mathcal{P}(\mathcal{S}_\Pi)$, defined as follows:

$$\beta_\mathcal{P}(\mathcal{S}_\Pi) = \beta_\mathcal{P}\Big[\mathcal{S}_\mathcal{Q}\big[(i_{cs}, o_{cs}, \mathcal{G}_{cs}, \mathcal{G}'_{cs}) / (i_\mathcal{Q}, o_\mathcal{Q}, \mathcal{G}, \mathcal{G}')\big] \\ / \Sigma_\mathcal{Q}(i_{cs}, o_{cs}, \mathcal{G}_{cs}, \mathcal{G}'_{cs}) \;\Big|\; (cs, \mathcal{Q}) \in \mathcal{CS}\Big]$$

Every satisfying model of $\beta_\mathcal{P}(\mathcal{S}_\Pi)$ describes a pair of feasible input-output states of $\mathcal{P}$ when assuming that the semantics of every called procedure is its summary. It also describes the intermediate input-output states of every call-site. Formally, this is captured by the notion of a $\mathcal{P}$*-trace*, defined below.

$\mathcal{P}$-Traces and $\mathcal{P}$-Transitions. For a procedure $\mathcal{P}$, a $\mathcal{P}$-trace $\sigma_\mathcal{P}$ is a model over the signature of $\beta_\mathcal{P}$, excluding the second-order predicates. Note that $\sigma_\mathcal{P}$ includes interpretations to all the copies of the global relations in $\beta_\mathcal{P}$.

In contrast, a $\mathcal{P}$*-transition* is a structure over the vocabulary $i_\mathcal{P}, \mathcal{G}, o_\mathcal{P}, \mathcal{G}'$ that describes a pair of input-output states of $\mathcal{P}$. During the algorithm we extract transitions from traces in the following ways:

- Given a $\mathcal{P}$-trace $\sigma_\mathcal{P}$, we denote by $\sigma_\mathcal{P}(\mathcal{P})$ the $\mathcal{P}$-transition that is obtained from $\sigma_\mathcal{P}$ by dropping the interpretation of the call-site copies of $\mathcal{G}$.
- Given a $\mathcal{P}$-trace $\sigma_\mathcal{P}$, for every call-site $(cs, \mathcal{Q}) \in \mathcal{CS}$ of $\mathcal{P}$ we denote by $\sigma_\mathcal{P}(cs)$ the $\mathcal{Q}$-transition that is extracted from $\sigma_\mathcal{P}$ in the following way. $\sigma_\mathcal{P}(cs)$ is defined over the same domain as $\sigma_\mathcal{P}$ and it provides interpretation for $\mathcal{G}, \mathcal{G}'$ in the same way $\mathcal{G}_{cs}, \mathcal{G}'_{cs}$ are interpreted in $\sigma_\mathcal{P}$, and provides interpretation to $i_\mathcal{Q}, o_\mathcal{Q}$ as the interpretation of the actual parameters and returns of the call-site i_{cs}, o_{cs} in $\sigma_\mathcal{P}$. Intuitively this is a decomposition of the trace within $\mathcal{P}$ to a list of transitions, where each transition corresponds to a procedure call within the trace.

Verification conditions. A safety property is provided by a $\forall^*\exists^*$-formula *Safe* over $i_\mathcal{M}, \mathcal{G}, o_\mathcal{M}, \mathcal{G}'$ that specifies a requirement on the input-output relation of the *main* procedure (where $i_\mathcal{M}$ and $o_\mathcal{M}$ denote the input and output parameters of the *main* procedure, respectively). Note that for the CME problem, it suffices to require safety of *main* since code that is unreachable from *main* cannot cause an exception in run-time,

and *cme* becoming true in some procedure manifests itself in the *main* procedure as well in our translation. Further, once *cme* becomes true, it remains true.

The verification conditions for safety properties are provided by the following lemma.

Lemma 1 (Verification conditions). *Let $\mathcal{A}$ be a program with a set of procedures Π and a main procedure $\mathcal{M}$, and let* Safe *be the safety property of $\mathcal{M}$. Let $\mathcal{S}_\Pi$ be a function mapping each procedure in Π to a summary. If it holds that $\forall \mathcal{P} \in \Pi.\ \beta_\mathcal{P}(\mathcal{S}_\Pi) \Rightarrow \mathcal{S}_\mathcal{P}$ and $\beta_\mathcal{M}(\mathcal{S}_\Pi) \Rightarrow$* Safe *then the program is safe.*

The first condition in the lemma guarantees that the $\mathcal{S}_\Pi$ summaries are overapproximations of all the reachable behaviors of each procedure. The second condition establishes that *main* is safe w.r.t the summaries in $\mathcal{S}_\Pi$. Hence, the program is safe according to our definition of program safety.

Note that given that bodies $\beta_\mathcal{P}$ are EPR formulas and the safety property *Safe* is a $\forall^*\exists^*$ formula, then if the summaries $\mathcal{S}_\mathcal{P}$ are provided as universally quantified formulas, the verification conditions are EPR formulas. Namely, checking them amounts to checking unsatisfiability of EPR formulas and is hence decidable.

3.5 Illustrative Example

We illustrate the modeling and the use of procedure summaries for the flatten example from Fig. 1. We wish to prove that executing the *main* procedure does not lead to a CME. As explained, we do this in a modular way by constructing procedure summaries for the procedures transitively called from *main*, namely the *flatten* and *addList* procedures.

We start by discussing the *addList* procedure. This procedure does not lead to a CME as long as `lst1` and `lst2` are different lists. This can be expressed in the procedure summary by the following formula:

$$\forall I. \neg cme(I) \wedge lst1 \neq lst2 \rightarrow \neg cme'(I)$$

As for the *flatten* procedure, as discussed in Section 2, it does not lead to a CME as long as `in` and `out` point to different lists, and `out` is not a member of the `in` list. This can be expressed in the procedure summary by the following formula:

$$\forall I. \neg cme(I) \wedge in \neq out \wedge \neg member\_cnt(in, out) \rightarrow \neg cme'(I)$$

Using additional formulas, it is possible to construct summaries for this example that satisfy the conditions of Lemma 1, and thus prove that the program is safe. The algorithm presented in Section 4 constructs such summaries.

4 Inference of Universally Quantified Procedure Summaries

In the previous section we defined the verification conditions using EPR formulas, based on procedure summaries sufficient to imply the safety property. In this section we tackle the problem of inferring such procedure summaries for a given program and safety property. To this end we develop a property-directed reachability algorithm that infers universally quantified procedure summaries in an interprocedural fashion.

The algorithm is based on UPDR [11] and interprocedural PDR algorithms [7, 13]. Upon termination the algorithm returns either universal summaries of the procedures that are sufficient to prove the safety of the program, or a counterexample. Similarly to UPDR, a counterexample discovered by the algorithm is an *abstract* counterexample. This may correspond to a real, concrete counterexample, but it is also possible that the program is in fact safe. In the latter case the abstract counterexample is a proof that the safety of the program *cannot* be proved by universal procedure summaries, i.e. there is no approximation of each procedure's semantics by universal summaries that is accurate enough to establish the safety property. In our tool we attempt to distinguish between the cases using *bounded-model checking* in order to find a concrete counterexample that matches the abstract one. Note that termination of the algorithm is not guaranteed.

4.1 Definitions

We begin by defining the required notations to describe the algorithm. For the remainder of this section we fix a specific program $\langle \Pi, \mathcal{G} \rangle$ whose designated entry point is $\mathcal{M} \in \Pi$.

The algorithm maintains a sequence of frames $\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_n$. Intuitively a frame $\mathcal{F}_i$ provides procedure summaries that constitute an overapproximation of the possible input-output relations of the procedures when the call-stack depth is bounded by i. The sequence is gradually modified and extended throughout the algorithm's run.

Technically, a *frame* $\mathcal{F}_i$ maps every procedure in Π to a summary of the procedure. Summaries themselves consist of a conjunction of universally quantified clauses, also referred to as *lemmas*. The following properties of the frame sequence are maintained by the algorithm: for all $0 \leq i < n$,

1. $\mathcal{F}_i(\mathcal{M}) \Rightarrow \mathit{Safe}$
2. $\forall \mathcal{P} \in \Pi.\ \mathcal{F}_i(\mathcal{P}) \Rightarrow \mathcal{F}_{i+1}(\mathcal{P})$
3. $\forall \mathcal{P} \in \Pi.\ \beta_{\mathcal{P}}(\mathcal{F}_i) \Rightarrow \mathcal{F}_{i+1}(\mathcal{P})$

Intuitively, the first property means that $\mathcal{F}_i$ is sufficient to prove the safety property when the stack depth is bounded to i (for every frame except the last one, which the algorithm refines). The second property means that frames are monotonic (since increasing the allowed stack depth does not remove possible behaviors), and the third property means that the summary of a procedure in frame $\mathcal{F}_{i+1}$ encompasses at least the possible behaviors of the procedure when its callees behave according to the summaries in frame $\mathcal{F}_i$. These properties ensure that the summaries of every $\mathcal{F}_i$ indeed overapproximate the behavior when the stack depth is bounded to i. Note that if for some $i < n$ the implication of property 2 holds in the opposite direction as well, then $\mathcal{F}_i$ satisfies the requirements of Lemma 1 and hence the program is safe.

Generalization by Diagrams. The essence of UPDR and the key to obtaining universally quantified invariants is the way the algorithm generates more lemmas for strengthening the frames. This is based on the notion of a *diagram* [11] , which provides a structural abstraction of transitions by an existential formulae.

Let $s = (\sigma_{in}, \sigma_{out})$ be a finite $\mathcal{P}$-transition. A diagram $\mathcal{D}(s)$ is an existential formula defined over the same vocabulary that describes the set of models that contain $(\sigma_{in}, \sigma_{out})$

as a substructure. Let $U = \{e_1, , \ldots, , e_{|U|}\}$ be the universe of s. The diagram is defined [11] as

$$\mathcal{D}(\sigma_{in}, \sigma_{out}) = \exists x_{e_1}, \ldots, x_{e_{|U|}}.\ \varphi_{dist} \wedge \varphi_{const} \wedge \varphi_{rel}$$

where

- $x_{e_1}, \ldots, x_{e_{|U|}}$ are fresh variables,
- φ_{dist} is a conjunction of the inequalities $x_{e_i} \neq x_{e_j}$ for every $e_i \neq e_j \in U$,
- φ_{const} is a conjunction of the equalities $c = x_e$ for every constant symbol $c \in i_{\mathcal{P}} \cup o_{\mathcal{P}}$ and $e \in U$ whose interpretation in s is c,
- φ_{rel} is conjunction of atomic formulas $p(x_{e_{i_1}}, \ldots, x_{e_{i_a}})$ for every predicate symbol of arity a and elements $\bar{e} = e_{i_1}, \ldots, e_{i_a} \in U$ such that the interpretation of $p(\bar{e})$ in s is true, and $\neg p(x_{e_{i_1}}, \ldots, x_{e_{i_a}})$ if it is false.

The existentially quantified $x_{e_1}, \ldots, x_{e_{|U|}}$ represent elements that constitute a substructure isomorphic to s when x_{e_i} takes the role of e_i.

It should be noted that all the satisfiability checks performed by the algorithm are of EPR formulas, and since EPR enjoys the finite-model property [18], the structures used by the algorithm are indeed all finite.

4.2 Interprocedural UPDR

Algorithm 1 presents the algorithm as set of rules, following [7, 13]. In addition to the frame sequence $\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_n$, where n is the current frame index, the algorithm also maintains a queue of reachability queries $\mathcal{L}$. A reachability query c is a tuple $\langle i, \mathcal{P}, s \rangle$, where i is a frame index, $\mathcal{P}$ a procedure symbol, and s is a $\mathcal{P}$-transition. In addition, the algorithm holds a set $\mathcal{R}$ of queries that were found to be reachable. Initially, $\mathcal{L}$ and $\mathcal{R}$ are empty and $n = 0$. The first frame $\mathcal{F}_0$ is initialized to $\lambda \mathcal{Q}.\ \mathit{false}$, which expresses the idea that a stack depth of 0 does not allow any transition from pre-state to post-state to be made by $\mathcal{P}$.

The rules are applied, possibly in a non-deterministic order, until either the algorithm terminates with a proof of safety when the frame sequence converges, or the algorithm encounters a reachable bad transition of main and terminates with an abstract counterexample. In the latter case, bounded-model checking is performed as an attempt find a concrete counterexample matching the abstract one. As in UPDR, an abstract counterexample may rely on transitions to a substructure, here on a structure that describes a procedure's transition. Assuming that the abstract counterexample was found in frame i, the algorithm traverses the program call-graph until depth i and generates a formula that describes all the possible bounded executions that match the abstract counterexample call-tree. It then uses this formula to check whether a possible execution violates the safety property by performing a satisfiability check. If a concrete counterexample is found, it is reported. Otherwise, the abstract counterexample serves as a proof that no universal summary exists for this program although the program may still be safe.

Unfold opens a new frame with permissive procedure summaries, to be refined until the summaries of the frame are strong enough to exclude all the bad behaviors (although the summaries may not yet be inductive). While a bad transition is allowed by the summaries the algorithm attempts to strengthen the frame sequence to exclude

Algorithm 1: Interprocedural UPDR algorithm

Input: Program $\langle \Pi, \mathcal{G} \rangle$ with main procedure $\mathcal{M} \in \Pi$ and safety property *Safe*
Output: *Safe*, *Not Safe* (+ concrete counterexample) or *No Universal Summaries*
Data:

1. Current frame index $n \in \mathbb{N}$
2. Sequence of frames $\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_n$
3. Reachability query queue $\mathcal{L} = \langle c_1, \ldots, c_k \rangle$, where $c_j = \langle i, \mathcal{P}, s \rangle$ is a reachability query with $i \leq n$, $\mathcal{P} \in \Pi$ and s a transition.
4. Set of reachable transitions $\mathcal{R}$, which holds tuples of the form $\langle i, \mathcal{P}, s \rangle$ where $i \leq n$, $\mathcal{P} \in \Pi$ and s is a transition: a structure over $i_{\mathcal{P}}, \mathcal{G}, o_{\mathcal{P}}, \mathcal{G}'$.

Init: $n = 0$, $\mathcal{F}_0 = \lambda \mathcal{P}.\, \mathit{false}$, $\mathcal{L} = \emptyset$, $\mathcal{R} = \emptyset$.
while *true* **do**

- **Unreachable:** If there exists an $i < n$ s.t. $\forall \mathcal{P} \in \Pi.\ \mathcal{F}_{i+1}(\mathcal{P}) \Rightarrow \mathcal{F}_i(\mathcal{P})$ return *Safe*.
- **Reachable:** If there exists a query $c = \langle n, \mathcal{M}, s \rangle \in \mathcal{L}$ s.t. $c \in \mathcal{R}$, perform bounded model checking to find a concrete counterexample. If found, return *Not Safe*. Else, return *No Universal Summaries*.
- **Unfold:** If $\beta_{\mathcal{M}}(\mathcal{F}_n) \Rightarrow \mathit{Safe}$ then set $\mathcal{F}_{n+1} := \lambda \mathcal{P}.\ \mathit{true}$, increment n to $n + 1$, and set $\mathcal{L} := \emptyset$.
- **Candidate:** If exists a $\mathcal{M}$-trace $\sigma_{\mathcal{M}}$ s.t. $\sigma_{\mathcal{M}} \models \beta_{\mathcal{M}}(\mathcal{F}_n) \wedge \neg \mathit{Safe}$, add $\langle n, \mathcal{M}, \sigma_{\mathcal{M}}(\mathcal{M}) \rangle$ to $\mathcal{L}$.
- **Decide:** If there exists a query $c = \langle i, \mathcal{P}, s \rangle \in \mathcal{L}$ with $i > 0$ and a $\mathcal{P}$-trace $\sigma_{\mathcal{P}}$ s.t. $\sigma_{\mathcal{P}} \models \beta_{\mathcal{P}}(\mathcal{F}_{i-1}) \wedge \mathcal{D}(s)$, add $\langle i - 1, \mathcal{Q}, \sigma_{\mathcal{P}}(cs) \rangle$ to $\mathcal{L}$ for every $(cs, \mathcal{Q}) \in \mathcal{CS}$.
- **Reachable-Base:** If there exists a query $c = \langle 1, \mathcal{P}, s \rangle \in \mathcal{L}$ and a $\mathcal{P}$-trace $\sigma_{\mathcal{P}}$ such that $\sigma_{\mathcal{P}} \models \beta_{\mathcal{P}}(\mathcal{F}_0) \wedge \mathcal{D}(s)$, add c to $\mathcal{R}$.
- **Reachable-Ind:** If there exists a query $c = \langle i, \mathcal{P}, s \rangle \in \mathcal{L}$ with $i > 1$ and a $\mathcal{P}$-trace $\sigma_{\mathcal{P}}$ s.t. $\sigma_{\mathcal{P}} \models \beta_{\mathcal{P}}(\mathcal{F}_{i-1}) \wedge \mathcal{D}(s)$, and $\langle i - 1, \mathcal{Q}, \sigma_{\mathcal{P}}(cs) \rangle \in \mathcal{R}$ for every $(cs, \mathcal{Q}) \in \mathcal{CS}$, then add c to $\mathcal{R}$.
- **Strengthen:** If there exists a query $c = \langle i, \mathcal{P}, s \rangle \in \mathcal{L}$ with $i > 0$ s.t. $\beta_{\mathcal{P}}(\mathcal{F}_{i-1}) \Rightarrow \neg \mathcal{D}(s)$, then compute $\varphi = \text{UNSAT-CORE}(\beta_{\mathcal{P}}(\mathcal{F}_{i-1}),\ \mathcal{D}(s))$. Set $\mathcal{F}_j(\mathcal{P}) := \mathcal{F}_j(\mathcal{P}) \wedge \varphi$ for all $0 \leq j \leq i$.
- **Push:** If φ is a conjunct of $\mathcal{F}_{i-1}(\mathcal{P})$ and $\beta_{\mathcal{P}}(\mathcal{F}_{i-1}) \Rightarrow \varphi$ then set $\mathcal{F}_i(\mathcal{P}) := \mathcal{F}_i(\mathcal{P}) \wedge \varphi$.
- **Reachability-Cache:** If $\langle i, \mathcal{P}, s \rangle \in \mathcal{L}$ and $\langle j, \mathcal{P}, s_0 \rangle \in \mathcal{R}$ with $j \leq i$ and $s \models \mathcal{D}(s_0)$, then add $\langle i, \mathcal{P}, s \rangle$ to $\mathcal{R}$.

end

that bad transition (**Candidate**), in a way that maintains the frame sequence invariants. This is done by placing the bad transition in the reachability query queue for further processing.

The next rules process reachability queries in the queue. **Strengthen** applies when the summaries of the previous frame are restrictive enough to exclude the possibility of a transition in question in the successive frame. In this case we intuitively learned that the transition is impossible for the bounded behavior of the procedure, and we strengthen the summaries of the procedure in the frame sequence with a new lemma that reflects it. To obtain a universal strengthening of the frames we try to exclude not just the concrete transition itself, but all the transitions in the diagram. This makes the strengthening process more powerful since it excludes more transitions, which must be excluded if universal summaries are to be used. To achieve convergence we further generalize the strengthening lemma by interpolation (using unsat cores).

Decide applies when the procedure summaries of the previous frame are insufficient to exclude the transition in question. In this case we attempt to (recursively) refine procedure summaries of the called procedures. This is done by analyzing a specific trace of the procedure that matches the query, and trying to exclude at least one of the transitions made by called procedures along this trace. To this end, the transitions of all the called procedures are added to the queue of reachability queries. Note that in this sense, the reachability queries unfold into a tree where each node represents a procedure call and a transition associated with it. Note further that if at least one of these transitions will turn out to be unreachable (in the corresponding frame) by **Strengthen**, then its frame will be refined, causing the trace of the caller to be unreachable as well. **Reachable-Base** and **Reachable-Ind** handle the case that none of the called procedures' transitions can be excluded, since they are all reachable and hence the transition of the caller is marked reachable.

The rest of the rules provide optimizations over the basic algorithm. **Push** attempts to push learned summary lemmas to the next frames when possible, in an attempt to achieve two equivalent frames as quickly as possible. **Reachability-Cache** attempts to avoid the need to go back in the frame sequence in order to understand that a transition is in fact reachable by reusing known reachable transitions. Note that since we are working with relaxed reachability it is sufficient to find a reachable transition in the diagram of the transition in question.

Lemma 2 (Correctness). *If the algorithm terminates with the result* Safe *then the safety property holds for the program. If the algorithm terminates with the result* Not Safe *then there is an execution of the program that violates the property. If the algorithm terminates with the result* No Universal Summaries *then there is no function $\mathcal{S}_{\mathcal{P}}$ that maps every procedure to a universally quantified summary which satisfies the verification conditions of Lemma 1.*

5 Implementation and Experiments

We have implemented an interprocedural version of UPDR on top of the Ivy framework [17]. The framework is implemented in Python (2.7), and uses Z3 (4.4.2 32 bit)

for satisfiability checking. We also implemented a front-end in Java, based on the Soot framework [22], that translates Java programs to Ivy's relational modeling language (RML). The experiments reported here were run on a machine with a 3.4GHz Intel Core i7-4770 8-core, 32GB of RAM, running Ubuntu 14.04.

Table 3. Experimental results.

Program	#Lines	#Meth.	Time	#Sum. (#Tot)	Max F.	#Z3	Max sum.	D.	W.	N.
simple_loop	11	1	66	4(15)	7	706	7	0	1	1
call_outside_loop	17	2	89	8(20)	7	1507	12	0	1	1
call_in_loop	17	2	91	8(20)	13	1816	10	0	1	1
update_call_in_loop	18	2	142	8(20)	13	2726	17	1	1	1
call_depth2_in_loop	22	3	177	11(22)	15	3433	17	2	1	1
call_depth2	22	3	97	5(22)	14	1712	1	2	1	1
sm	53	6	514	8(27)	11	5263	14	1	2	1
div	27	2	291	7(22)	11	2767	15	rec	2	rec
worklist	26	5	577	8(26)	17	7475	41	3	2	1
map_test	40	3	306	10(24)	14	4753	16	1	1	1
c	18	3	33	8(18)	6	791	8	2	2	0
flatten	45	3	800	14(27)	18	10412	39	2	1	2
c_error	18	3	35	—	5	662	—	2	2	0
flatten_error	45	3	790	—	15	8746	—	2	1	2

#Lines denotes the number of code lines and **#Meth.** is the number of methods. **Time** is measured in seconds. **#Sum** is the number of non-trivial computed summaries, i.e. summaries that are not equivalent to true, while **#Tot** is the overall number of summaries. **Max F.** denotes the highest frame reached by the algorithm. **#Z3** represents the number of calls to Z3. **Max sum.** denotes the number of clauses in the largest obtained summary. **D.** denotes the maximal depth of the call-graph, where a collection update occurs (*main* method is treated as 0 depth). **W.** is the maximal number of calls to functions that update collections from a single method. **N.** is the maximal number of nested loops that contain updates. **rec** represents a recursive program.

The Soot-based front-end transforms Java programs to an intermediate language based on Ivy's RML. It splits every Java method to its basic blocks, and each basic block is transformed to a procedure that requires a summary, and can include calls to other procedures, including itself. For uniformity of implementation loops are effectively translated to tail recursions. A by-product of this is that the interprocedural analysis sees many procedures for every Java method, i.e., one for each basic block. In addition, due to our abstraction of the non-CME related operations, some of these procedures contain just a `skip` instruction, and their summary is trivial.

We evaluated our implementation on several Java programs that manipulate collections via iterators and are known to be safe. The test programs were inspired by code examples taken from [19] (*worklist*, *map_test*), an example (*flatten*) and a false alarm example (*sm*) taken from the tool in [16], and our own implementation (*div*, *c*). The

worklist example contains a class that manipulates a list class field. The *div* example includes a method that divides a list recursively and inserts some of its items to an output list. The *map_test* example performs basic checks on a sorted map structure. The *c* test includes two iterators operating on the same collection when one is an alias to the other. The *flatten* example includes a class that performs a flattening of a list of lists structure.

For all programs, the analysis was able to prove the absence of CMEs. The results are summarized in Table 3. In addition, we intentionally inserted bugs to the code of *c* and *worklist* and our tool succeeded in detecting them.

To examine the effect of different code properties on our algorithm, we applied our tool on a few slightly different versions of a simple safe program. The *simple_loop* example has a single main method that iterates over a collection and updates it. The *call_outside_loop* and *call_in_loop* programs are similar, but also contain a method that has no effect on the collection, and is called outside of the iteration loop (*call_outside_loop*) or from within the loop (*call_in_loop*). The *update_call_in_loop* example calls a method that updates a collection from the loop body. *call_depth2_in_loop*, *call_depth2* perform the same updates as before, but wraps the update inside another method. The properties we checked in this test are the depth of collection updates (add/remove) in the call-graph and the loop nesting depth of the updates. The results show that both properties have a major effect on the running times, frame number and number of Z3 calls.

The results indicate that the main factor affecting the run-time of the analysis is not the code length, but rather the location of collection updates in the call-graph. When the call-graph is deeper, the generated summaries are usually larger, more calls to Z3 are preformed, and run-time grows. Methods that do not manipulate collections do not appear to affect the size of the summaries. Thus, we expect that our method can be applied to large programs, with reasonably shallow update depth, width and nesting. We also expect that an additional engineering effort can significantly reduce the run-time of the analysis, as we regard our implementation as a proof-of-concept rather than an optimized implementation.

6 Related Work

Property Directed Reachabilty. The IC3/PDR algorithm [2, 5] has led to many successful applications both in software and in hardware verification. More recently, UPDR (Universal PDR) [11] introduced the idea of using diagrams as a way to lift PDR to infinite state systems. Our work builds on these works, and can be seen as an application of UPDR to infer universally quantified procedure summaries rather then loop invariants.

Interprocedural Analysis. Interprocedural analysis [21, 20] is an important theme for verification and program analysis. Following the introduction of the IC3/PDR algorithm, [7] and [13, 14, 12] developed a way to apply it to interprocedural analysis. The main idea, which we also apply in this work, is to infer procedure summaries in the same way PDR infers loop invariants. While this line of works has been applied in the context of various array theories and arithmetic, our work is the first to apply it using EPR, inferring universal summaries by using diagrams.

Modeling with Decidable Logics. While any logic that fully describes computer programs is undecidable, there have been many attempts to identify decidable logic fragments that are still useful for analyzing programs. Decidability has the potential to make program verification tools more predictable and useful. The array property fragment [3] and its variants are often used to analyze programs that use arrays and perform simple arithmetic. Logics such as Mona [6], Strand [15] and EPR [9, 8] have been used to model heap manipulating programs. In this context, our work is using the EPR logic, and identifies a new problem domain for which it can be useful: proving the absence of concurrent modification errors in Java programs. Extending the range of applicability of decidable logics is an ongoing research effort, and our work can be seen as another step in this process.

Concurrent Modification Errors and Analyzing Java Programs. Several static analyses have been developed to detect possible Concurrent Modification Exceptions in Java programs, as well as violations of other typestate properties involving multiple objects. [1] presented a flow-sensitive typestate analysis based on an intraprocedural must-alias analysis that can rule out CME violations if the collection is created, iterated over, and modified only within a single procedure. [16] evaluated an interprocedural context-sensitive analysis of aliasing and multi-object typestate that can rule out CME violations, but does not reason precisely about objects that escape to the heap and are no longer directly referenced by any local variable. [10] presented a specification language for describing properties such as CME violations and a pragmatic static verifier that is neither sound nor complete, but effective in practice.

Acknowledgments. We would like to thank Nikolaj Bjørner, Roman Manevich and Eran Yahav for their helpful discussions, and the Programming Languages team in TAU for their support and feedback on the paper. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no [321174].

References

1. E. Bodden, P. Lam, and L. J. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 36–47, 2008.
2. A. Bradley. Sat-based model checking without unrolling. In R. Jhala and D. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer Berlin Heidelberg, 2011.
3. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In E. A. Emerson and K. S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer, 2006.
4. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

5. N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation of property directed reachability. In P. Bjesse and A. Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 125–134. FMCAD Inc., 2011.
6. J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS*, pages 89–110, 1995.
7. K. Hoder and N. Bjørner. Generalized property directed reachability. In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2012.
8. S. Itzhaky, A. Banerjee, N. Immerman, O. Lahav, A. Nanevski, and M. Sagiv. Modular reasoning about heap paths via effectively propositional formulas. In *the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 385–396, 2014.
9. S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In *Computer Aided Verification - 25th International Conference, CAV*, pages 756–772, 2013.
10. C. Jaspan and J. Aldrich. Checking framework interactions with relationships. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, pages 27–51, 2009.
11. A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham. Property-directed inference of universal invariants or proving their absence. In *Computer Aided Verification - 27th International Conference, CAV*, pages 583–602, 2015.
12. A. Komuravelli, N. Bjørner, A. Gurfinkel, and K. L. McMillan. Compositional verification of procedural programs using horn clauses over integers and arrays. In R. Kaivola and T. Wahl, editors, *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pages 89–96. IEEE, 2015.
13. A. Komuravelli, A. Gurfinkel, and S. Chaki. Smt-based model checking for recursive programs. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 17–34. Springer, 2014.
14. A. Komuravelli, A. Gurfinkel, and S. Chaki. Smt-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.
15. P. Madhusudan and X. Qiu. Efficient decision procedures for heaps using STRAND. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, pages 43–59, 2011.
16. N. A. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 347–366, 2008.
17. O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 614–630, 2016.
18. R. Piskac, L. M. de Moura, and N. Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. *J. Autom. Reasoning*, 44(4):401–424, 2010.

19. G. Ramalingam, A. Varshavsky, J. Field, D. Goyal, and S. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 83–94, 2002.
20. T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In R. K. Cytron and P. Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61. ACM Press, 1995.
21. M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
22. R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13, 1999.
23. E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 25–34, 2004.