PROPERTY DIRECTED REACHABILITY FOR PROVING ABSENCE OF CONCURRENT MODIFICATION ERRORS

by

Asya Frumkin

under the supervision of
Prof. Mooly Sagiv and Dr. Sharon Shoham

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science

2017
Abstract

Property Directed Reachability for Proving Absence of Concurrent Modification Errors

Asya Frumkin
Master of Science
School of Computer Science
Tel Aviv University

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.
Acknowledgements

I would like to express my gratitude to my supervisors Prof. Mooly Sagiv and Dr. Sharon Shoham for their guidance along the way. Mooly’s vast knowledge and experience contributed greatly to this research while Sharon inspired me greatly with her drive for formalism and accuracy. Furthermore, I like to thank Yotam M. Y. Feldman, for his major role in the algorithm implementation and formulation and Oded Padon for his insightful ideas and useful discussions. A very special thanks goes to the rest of the Programming Languages team in Tel Aviv University: Kalev, Hila, Or and Elazar, for the motivating talks and helpful remarks.

I would like to thank my family and friends, who have supported me throughout the entire process. Last but not least, I would like to thank my dearest one, for his love, optimism and encouragement and for believing in me even in the darkest moments.
Contents

1 Introduction 1
   1.0.1 Main Results ......................................................... 1

2 The Concurrent Modification Problem 3

3 EPR Verification Conditions for CME 7
   3.1 Program state ............................................................ 8
   3.2 Modeling Java operations in EPR ...................................... 9
   3.3 Modeling programs using EPR ........................................ 9
   3.4 Verification conditions ............................................... 12
      3.4.1 Procedure summaries ............................................. 12
      3.4.2 Verification conditions ......................................... 13
   3.5 Illustrative Example .................................................. 13

4 Inference of Universally Quantified Procedure Summaries 15
   4.0.1 Definitions ........................................................... 15
   4.0.2 Interprocedural UPDR ............................................... 17

5 Implementation and Experiments 21

6 Related Work 23

Bibliography 24
List of Tables

3.2 Java statements and their EPR interpretation .......................... 10
5.1 Experimental results ....................................................... 22
List of Figures

2.1 List of lists implementation .................................................. 5

3.1 A small program with procedure calls ................................. 11
Chapter 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 [RVF+02, YR04, NL08]. 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.

1.0.1 Main Results

Our key insight in this thesis 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 [KBI+15] with the techniques of [HB12, KGC14] for computing procedure summaries.
This thesis 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 [IBI+13].

- 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 [KBI+15] and follows the idea of property directed reachability [HB12, KGC14].

- We implemented the algorithm on top of Ivy [PMP+16] and Z3 [dMB08].

- We implemented a front-end for Java using Soot [VCG+99] which converts Java programs into EPR and integrates with the above procedure.
Chapter 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.

Problem Statement

The Concurrent Modification Problem is do determine, given a Java program, whether the execution of the program can cause a CME exception. The program is safe if no CME exception can be thrown.

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 2.0.1. Let us consider the code presented in Fig. 2.1, 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 \texttt{itr1.next()} call would throw an exception. Also, if the out list is included as one of the lists in the \texttt{in} list, \texttt{itr2} would be invalidated once \texttt{itr1} reaches out and the object pointed to by \texttt{itr2} is added to out. A subsequent \texttt{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.

**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 [RVF+02], 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.
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);
        }
    }
}

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);
    }
}

Figure 2.1: Usage of list of lists implementation that does not cause a CME to be thrown
CHAPTER 2. THE CONCURRENT MODIFICATION PROBLEM
Chapter 3

EPR Verification Conditions for CME

In this chapter 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 method fragment of first-order logic. EPR formulas are of the form $\exists^* \forall^* \phi$ where $\phi$ 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. This means that the sorts can be ordered by a total order $<$ such that if there is a function mapping sorts $srt_1, \ldots, srt_n$ to sort $srt'$ then $srt' < srt_i$ for every $1 \leq i \leq n$ (i.e., the directed graph of sort dependencies induced by the functions does not contain cycles). Extended-EPR formulas maintain the decidability of EPR \cite{PMP+16}. In the sequel, we will use the term EPR to refer to the extended EPR fragment.

Representation of programs in EPR. When using EPR to represent a program, a state of the program is given by a finite first-order structure over a many-sorted vocabulary that consists of sorted constants, relation symbols and stratified function symbols. Objects in the heap of the program are modeled by individuals in the first order structure. The constant symbols represent the program variables, and the function and relation symbols represent relations between variables and other elements in the domain (e.g., the heap). Sets of states can then be represented by formulas, where a closed formula represents the set of states (structures) that satisfy it. Similarly, by using copies of the vocabulary, one can also describe sets of transitions and sets of finite executions of the program.

In the following, we explain how we realize this framework for defining an abstraction of Java programs that targets finding CME exceptions.
Table 3.1: Abstraction relations and functions in the vocabulary.

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(cnt(I))</td>
<td>(I): iterator</td>
<td>function which returns a container, maps iterators to their underlying</td>
</tr>
<tr>
<td></td>
<td></td>
<td>containers</td>
</tr>
<tr>
<td>(member(C,O))</td>
<td>(C): container, (O): obj</td>
<td>object is included in container</td>
</tr>
<tr>
<td>(member_cnt(C_1,C_2))</td>
<td>(C_1): container, (C_2): container</td>
<td>container is included in container</td>
</tr>
<tr>
<td>(used_cnt(C))</td>
<td>(C): container</td>
<td>container is initialized</td>
</tr>
<tr>
<td>(used_iter(I))</td>
<td>(I): iterator</td>
<td>iterator is initialized</td>
</tr>
<tr>
<td>(used_obj(O))</td>
<td>(O): obj</td>
<td>object is initialized</td>
</tr>
<tr>
<td>(stale(I))</td>
<td>(I): iterator</td>
<td>iterator is invalid</td>
</tr>
<tr>
<td>(cme(I))</td>
<td>(I): iterator</td>
<td>concurrent modification occurred</td>
</tr>
</tbody>
</table>

3.1 Program state

In this section we describe the vocabulary used to define a (abstract) program state. To facilitate CME verification, our model for Java programs partitions Java objects into three distinct types: container, iterator and obj. Each of these types corresponds to a sort in the vocabulary. 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 \(i\) has a mapping \(cnt(i)\) to the container it operates on. This mapping is captured via the function symbol \(cnt\). 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 for 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 shorthand, defined by \(points\_to(i,o) \overset{\text{def}}{=} \exists c_0. (cnt(i) = c_0) \land member(c_0,o)\) and \(points\_to\_cnt(i,c) \overset{\text{def}}{=} \exists c_0. (cnt(i) = c_0) \land member\_cnt(c_0,c)\).

The constructs mentioned above are listed in Table 3.1 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 3.2).

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 \(G\).
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 $G$ is used in order to express the pre-state, and $G' = \{v' \mid v \in G\}$ is the vocabulary used to express the post-state. Both $G$ and $G'$ include only global relations, and the encoding of each primitive operation via an EPR formula is given in Table 3.2.

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.\footnote{The translation also incorporates the fact that once a CME occurs, the normal control-flow of the program is interrupted by the exception.} 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 [SP81]. 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 [KGC16]. A program $A$ is a pair $\langle \Pi, G \rangle$, where $\Pi$ is a non-empty set of procedures with a designated procedure $M$ (main) serving as the entry point. $G$ is a set of predicate
Table 3.2: Java statements and their EPR interpretation. $c$ denotes a container, $i$ an iterator, and $o$ an obj.

<table>
<thead>
<tr>
<th>Java statement</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$.hasNext()</td>
<td>$-$</td>
</tr>
<tr>
<td>$o = i$.next()</td>
<td>$\text{points}_\text{to}(i, o) \land (\forall I. \ cme'(I) \iff cme(I) \lor$</td>
</tr>
<tr>
<td></td>
<td>$(I = i \land \text{stale}(i)))$</td>
</tr>
<tr>
<td>$c = i$.next()</td>
<td>$\text{points}<em>\text{to}</em>\text{cnt}(i, c) \land (\forall I. \ cme'(I) \iff$</td>
</tr>
<tr>
<td></td>
<td>$cme(I) \lor (I = i \land \text{stale}(i)))$</td>
</tr>
<tr>
<td>$c$.add($o$)</td>
<td>$\forall C, O, I. \ (\text{member}^C(C, O) \iff \text{member}(C, O) \lor$</td>
</tr>
<tr>
<td></td>
<td>$(C = c \land O = o)) \land (\text{stale}'(I) \iff \text{stale}(I) \lor$</td>
</tr>
<tr>
<td></td>
<td>$(\text{cnt}(I) = c \land \text{used}_\text{itr}(I)))$</td>
</tr>
<tr>
<td>$c_1$.add($c_2$)</td>
<td>$\forall C_1, C_2, I. \ (\text{member}^C(C_1, C_2) \iff$</td>
</tr>
<tr>
<td></td>
<td>$\text{member}_\text{cnt}(C_1, C_2) \lor (C_1 = c_1 \land C_2 = c_2)) \land$</td>
</tr>
<tr>
<td></td>
<td>$(\text{stale}'(I) \iff \text{stale}(I) \lor (\text{cnt}(I) = c_1 \land$</td>
</tr>
<tr>
<td></td>
<td>$\text{used}_\text{itr}(I))))$</td>
</tr>
<tr>
<td>$i$.remove()</td>
<td>$\exists o, c. \forall C, C', O, I. \ (\text{used}_\text{obj}(o) \land$</td>
</tr>
<tr>
<td></td>
<td>$(\text{member}^C(C, O) \iff \text{member}(C, O) \lor$</td>
</tr>
<tr>
<td></td>
<td>$(C \neq \text{cnt}(i) \lor O \neq o))$</td>
</tr>
<tr>
<td></td>
<td>$\lor (\text{used}<em>\text{cnt}(c) \land (\text{member}</em>\text{cnt}(C, C') \iff$</td>
</tr>
<tr>
<td></td>
<td>$\text{member}_\text{cnt}(C, C') \land (C \neq \text{cnt}(i) \lor$</td>
</tr>
<tr>
<td></td>
<td>$C' \neq c))) \land (\text{stale}'(I) \iff \text{stale}(I) \lor (\text{cnt}(I) =$</td>
</tr>
<tr>
<td></td>
<td>$\text{cnt}(i) \land I \neq i \land \text{used}_\text{itr}(I))))$</td>
</tr>
<tr>
<td>$c = \text{new Collection}()$</td>
<td>$\neg \text{used}<em>\text{cnt}(c) \land (\forall I. \ (\text{used}</em>\text{itr}(I) \Rightarrow (\text{cnt}(I) \neq c)) \land$</td>
</tr>
<tr>
<td></td>
<td>$(\forall O. \neg \text{member}(c, O)) \land (\forall C. \neg \text{member}_\text{cnt}(c, C) \land$</td>
</tr>
<tr>
<td></td>
<td>$\neg \text{member}<em>\text{cnt}(C, c)) \land (\forall C. \text{used}</em>\text{cnt}(C) \iff$</td>
</tr>
<tr>
<td></td>
<td>$\text{used}_\text{cnt}(C) \lor C = c)$</td>
</tr>
<tr>
<td>$i = c$.iterator()</td>
<td>$\neg \text{used}_\text{itr}(i) \land (\text{cnt}(i) = c) \land (\forall I. \text{stale}'(I) \iff$</td>
</tr>
<tr>
<td></td>
<td>$\text{stale}(I) \land I \neq i \land (\forall I. \text{used}_\text{itr}'(I) \iff$</td>
</tr>
<tr>
<td></td>
<td>$\text{used}_\text{itr}(I) \lor I = i)$</td>
</tr>
<tr>
<td>$o = \text{new Object}()$</td>
<td>$\neg \text{used}_\text{obj}(o) \land (\forall C. \neg \text{member}(C, o)) \land$</td>
</tr>
<tr>
<td></td>
<td>$(\forall O. \text{used}<em>\text{obj}(O) \iff \text{used}</em>\text{obj}(O) \lor O = o)$</td>
</tr>
</tbody>
</table>
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 \( |i_\mathcal{P} \cup o_\mathcal{P} \cup G \cup G'| \) whose role is to represent 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 \( \beta_\mathcal{P} \) represents the behavior (executions) of the procedure body of \( \mathcal{P} \) w.r.t. the behavior (executions) 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:

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

- Pairs of vocabularies \( G_{cs}, G'_{cs} \) 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 \( G'_{cs_1} \) and \( G_{cs_2} \) 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 \( \Sigma_\mathcal{Q}(i_{cs}, o_{cs}, G_{cs}, G'_{cs}) \), 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.

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

Figure 3.1: A small program with procedure calls

**Example 3.3.1.** Let us consider the example in Figure 3.1. 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. \( \Sigma_f \) denotes the second-order predicate used to represent \( f \) in the body of other procedures. \( main \) 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:

\[
\beta_M = \exists a, b. \ R'(a) \land \neg S'(b) \land \Sigma_f((a, b), \emptyset, \langle R', S' \rangle, \langle R'', S'' \rangle) \land \\
\Sigma_f((b, a), \emptyset, \langle R'', S'' \rangle, \langle R''', S''' \rangle)
\]

Note, that \( \beta_M \) includes 4 lists of global variables: \( \langle R, S \rangle \), \( \langle R', S' \rangle \), \( \langle R'', S'' \rangle \), \( \langle R''', S''' \rangle \) where each represents the program state at a different point along the execution of \( \text{main} \) (\( R, S \) are not constrained by this formula as they represent the state before the execution starts).

### 3.4 Verification conditions

The accurate semantics of \( \beta_P \) has a least-fixed-point characterization as described in [KGC16]. 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.

#### 3.4.1 Procedure summaries.

The summary \( S_P \) of a procedure \( P \) 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 \( G, G' \): one for the pre-state and one for the post-state, the formal arguments \( i_P \) and the formal returns \( o_P \).

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

The behavior of \( P \) w.r.t. the summaries \( S_{\Pi} \) is obtained by replacing every predicate symbol \( \Sigma_Q \) in \( \beta_P \) with \( S_Q \), applied over the vocabulary determined by the call-site. Formally, it is captured by the formula \( \beta_P(S_{\Pi}) \), defined as follows:

\[
\beta_P(S_{\Pi}) = \beta_P\left[ S_Q\left[ \left( i_{cs}, o_{cs}, G_{cs}, G'_{cs} \right) \cup \left( i_Q, o_Q, G, G' \right) \right] \bigg/ \Sigma_Q(i_{cs}, o_{cs}, G_{cs}, G'_{cs}) \mid (cs, Q) \in CS \right]
\]

Every satisfying model of \( \beta_P(S_{\Pi}) \) describes a pair of feasible input-output states of \( P \) 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 \( P \)-trace, defined below.

**\( P \)-Traces and \( P \)-Transitions.** For a procedure \( P \), a \( P \)-trace \( \sigma_P \) is a model over the signature of \( \beta_P \), excluding the second-order predicates. Note that \( \sigma_P \) includes interpretations to all the copies of the global relations in \( \beta_P \).
In contrast, a \( P \)-transition is a structure over the vocabulary \( i_P, G, o_P, G' \) that describes a pair of input-output states of \( P \). During the algorithm we extract transitions from traces in the following ways:

- Given a \( P \)-trace \( \sigma_P \), we denote by \( \sigma_P(P) \) the \( P \)-transition that is obtained from \( \sigma_P \) by dropping the interpretation of the call-site copies of \( G \).

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

### 3.4.2 Verification conditions.

A safety property is provided by a \( \forall^* \exists^* \)-formula \( Safe \) over \( i_M, G, o_M, G' \) that specifies a requirement on the input-output relation of the main procedure (where \( i_M \) and \( o_M \) 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 3.4.1** (Verification conditions). Let \( A \) be a program with a set of procedures \( \Pi \) and a main procedure \( M \), and let \( Safe \) be the safety property of \( M \). Let \( S_{\Pi} \) be a function mapping each procedure in \( \Pi \) to a summary. If it holds that \( \forall P \in \Pi. \beta_P(S_{\Pi}) \Rightarrow S_P \) and \( \beta_M(S_{\Pi}) \Rightarrow Safe \) then the program is safe.

The first condition in the lemma guarantees that the \( S_{\Pi} \) 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 \( S_{\Pi} \). Hence, the program is safe according to our definition of program safety.

Note that given that bodies \( \beta_P \) are EPR formulas and the safety property \( Safe \) is a \( \forall^* \exists^* \) formula, then if the summaries \( S_P \) 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. 2.1. 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 \textit{addList} procedure. This procedure does not lead to a CME as long as \texttt{lst1} and \texttt{lst2} are different lists. This can be expressed in the procedure summary by the following formula:

\[ \forall I. \neg \text{cme}(I) \land \texttt{lst1} \neq \texttt{lst2} \rightarrow \neg \text{cme}'(I) \]

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

\[ \forall I. \neg \text{cme}(I) \land \texttt{in} \neq \texttt{out} \land \neg \text{member} \_ \text{cnt}(\texttt{in}, \texttt{out}) \rightarrow \neg \text{cme}'(I) \]

Using additional formulas, it is possible to construct summaries for this example that satisfy the conditions of Lemma 3.4.1 and thus prove that the program is safe. The algorithm presented in Chapter 4 constructs such summaries.
Chapter 4

Inference of Universally Quantified Procedure Summaries

In the previous chapter we defined the verification conditions using EPR formulas, based on procedure summaries sufficient to imply the safety property. In this chapter 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 \[KBI^{+15}\] and interprocedural PDR algorithms \[HB12, KGC14\]. 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.

4.0.1 Definitions

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

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

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 3.4.1 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* \([KBI+15]\), which provides a structural abstraction of transitions by an existential formulae.

Let \(s = (\sigma_{in}, \sigma_{out})\) be a finite \(P\)-transition. A diagram \(D(s)\) 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 \([KBI+15]\) as

\[
D(\sigma_{in}, \sigma_{out}) = \exists x_{e_1}, \ldots, x_{e_{|U|}}. \varphi_{dist} \land \varphi_{const} \land \varphi_{rel}
\]

where

- \(x_{e_1}, \ldots, x_{e_{|U|}}\) are fresh variables,
- \(\varphi_{dist}\) is a conjunction of the inequalities \(x_{e_i} \neq x_{e_j}\) for every \(e_i \neq e_j \in U\),
- \(\varphi_{const}\) is a conjunction of the equalities \(c = x_e\) for every constant symbol \(c \in p \cup o\) and \(e \in U\) whose interpretation in \(s\) is \(c\),
- \(\varphi_{rel}\) 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 \([PdB10]\), the structures used by the algorithm are indeed all finite.
4.0.2 Interprocedural UPDR

Algorithm presents the algorithm as set of rules, following \cite{HB12,KGC14}. In addition to the frame sequence $F_0, F_1, \ldots, F_n$, where $n$ is the current frame index, the algorithm also maintains a queue of reachability queries $L$. A reachability query $c$ is a tuple $\langle i, P, s \rangle$, where $i$ is a frame index, $P$ a procedure symbol, and $s$ is a $P$-transition. In addition, the algorithm holds a set $R$ of queries that were found to be reachable. Initially, $L$ and $R$ are empty and $n = 0$. The first frame $F_0$ is initialized to $\lambda Q. \text{false}$, 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 $P$.

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 to 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 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
Algorithm 1: Interprocedural UPDR algorithm

**Input:** Program $⟨Π, G⟩$ with main procedure $M ∈ Π$ and safety property Safe

**Output:** Safe, Not Safe (+ concrete counterexample) or No Universal Summaries

**Data:**

1. Current frame index $n ∈ N$
2. Sequence of frames $F_0, F_1, ..., F_n$
3. Reachability query queue $L = ⟨c_1, ..., c_k⟩$, where $c_j = ⟨i, P, s⟩$ is a reachability query with $i ≤ n$, $P ∈ Π$ and $s$ a transition.
4. Set of reachable transitions $R$, which holds tuples of the form $⟨i, P, s⟩$ where $i ≤ n$, $P ∈ Π$ and $s$ a transition: a structure over $i_P, G, o_P, G'$.

**Init:** $n = 0, F_0 = λP. false, L = ∅, R = ∅$.

while true do

Unreachable: If there exists an $i < n$ s.t. $∀P ∈ Π. F_{i+1}(P) ⇒ F_i(P)$ return Safe.

Reachable: If there exists a query $c = ⟨n, M, s⟩ ∈ L$ s.t. $c ∈ R$, perform bounded model checking to find a concrete counterexample. If found, return Not Safe. Else, return No Universal Summaries.

Unfold: If $β_M(F_n) ⇒ Safe$ then set $F_{n+1} := λP. true$, increment $n$ to $n + 1$, and set $L := ∅$.

Candidate: If exists a $M$-trace $σ_M$ s.t. $σ_M \models β_M(F_n) ∧ ¬Safe$, add $⟨n, M, σ_M(M)⟩$ to $L$.

Decide: If there exists a query $c = ⟨i, P, s⟩ ∈ L$ with $i > 0$ and a $P$-trace $σ_P$ s.t. $σ_P \models β_P(F_{i-1}) ∧ D(s)$, add $⟨i - 1, Q, σ_P(cs)⟩$ to $L$ for every $(cs, Q) ∈ CS$.

Reachable-Base: If there exists a query $c = ⟨1, P, s⟩ ∈ L$ and a $P$-trace $σ_P$ such that $σ_P \models β_P(F_0) ∧ D(s)$, add $c$ to $R$.

Reachable-Ind: If there exists a query $c = ⟨i, P, s⟩ ∈ L$ with $i > 1$ and a $P$-trace $σ_P$ s.t. $σ_P \models β_P(F_{i-1}) ∧ D(s)$, and $⟨i - 1, Q, σ_P(cs)⟩ ∈ R$ for every $(cs, Q) ∈ CS$, then add $c$ to $R$.

Strengthen: If there exists a query $c = ⟨i, P, s⟩ ∈ L$ with $i > 0$ s.t. $β_P(F_{i-1}) ⇒ ¬D(s)$, compute $ϕ = UNSAT-CORE(β_P(F_{i-1}), D(s))$. Set $F_j(P) := F_j(P) ∧ ϕ$ for all $0 ≤ j ≤ i$.

Push: If $ϕ$ is a conjunct of $F_{i-1}(P)$ and $β_P(F_{i-1}) ⇒ ϕ$ then set $F_i(P) := F_i(P) ∧ ϕ$.

Reachability-Cache: If $⟨i, P, s⟩ ∈ L$ and $⟨j, P, s_0⟩ ∈ R$ with $j ≤ i$ and $s \models D(s_0)$, then add $⟨i, P, s⟩$ to $R$.

end
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 4.0.1** (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 \( S_P \) that maps every procedure to a universally quantified summary which satisfies the verification conditions of Lemma 3.4.1.
Chapter 5

Implementation and Experiments

We have implemented an interprocedural version of UPDR on top of the Ivy framework \cite{PMP16}. 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 \cite{VCG99}, 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.

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 \texttt{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 \cite{RVF02} (worklist, map\_test), an example (flatten) and a false alarm example (sm) taken from the tool in \cite{NL08}, 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 5.1. 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.

Table 5.1: Experimental results

<table>
<thead>
<tr>
<th>Program</th>
<th>#Lines</th>
<th>#Meth.</th>
<th>Time</th>
<th>#Sum. (#Tot)</th>
<th>Max F.</th>
<th>#Z3</th>
<th>Max sum.</th>
<th>D.</th>
<th>W.</th>
<th>N.</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple_loop</td>
<td>11</td>
<td>1</td>
<td>66</td>
<td>4(15)</td>
<td>7</td>
<td>706</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>call_outside_loop</td>
<td>17</td>
<td>2</td>
<td>89</td>
<td>8(20)</td>
<td>7</td>
<td>1507</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>call_in_loop</td>
<td>17</td>
<td>2</td>
<td>91</td>
<td>8(20)</td>
<td>13</td>
<td>1816</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>update_call_in_loop</td>
<td>18</td>
<td>2</td>
<td>142</td>
<td>8(20)</td>
<td>13</td>
<td>2726</td>
<td>17</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>call_depth2_in_loop</td>
<td>22</td>
<td>3</td>
<td>177</td>
<td>11(22)</td>
<td>15</td>
<td>3433</td>
<td>17</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>call_depth2</td>
<td>22</td>
<td>3</td>
<td>97</td>
<td>5(22)</td>
<td>14</td>
<td>1712</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sm</td>
<td>53</td>
<td>6</td>
<td>514</td>
<td>8(27)</td>
<td>11</td>
<td>5263</td>
<td>14</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>div</td>
<td>27</td>
<td>2</td>
<td>291</td>
<td>7(22)</td>
<td>11</td>
<td>2767</td>
<td>15</td>
<td>rec2</td>
<td>rec</td>
<td></td>
</tr>
<tr>
<td>worklist</td>
<td>26</td>
<td>5</td>
<td>577</td>
<td>8(26)</td>
<td>17</td>
<td>7475</td>
<td>41</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>map_test</td>
<td>40</td>
<td>3</td>
<td>306</td>
<td>10(24)</td>
<td>14</td>
<td>4753</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>18</td>
<td>3</td>
<td>33</td>
<td>8(18)</td>
<td>6</td>
<td>791</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>flatten</td>
<td>45</td>
<td>3</td>
<td>800</td>
<td>14(27)</td>
<td>18</td>
<td>10412</td>
<td>39</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>c_error</td>
<td>18</td>
<td>3</td>
<td>35</td>
<td>—</td>
<td>5</td>
<td>662</td>
<td>—</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>flatten_error</td>
<td>45</td>
<td>3</td>
<td>790</td>
<td>—</td>
<td>15</td>
<td>8746</td>
<td>—</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

#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 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.
Chapter 6

Related Work

Property Directed Reachability. The IC3/PDR algorithm [Bra11, EMB11] has led to many successful applications both in software and in hardware verification. More recently, UPDR (Universal PDR) [KBI+15] 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 than loop invariants.

Interprocedural Analysis. Interprocedural analysis [SP81, RHS95] is an important theme for verification and program analysis. Following the introduction of the IC3/PDR algorithm, [HB12] and [KGC14, KGC16, KBGM15] 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 [BMS06] and its variants are often used to analyze programs that use arrays and perform simple arithmetic. Logics such as Mona [HJJ+95], Strand [MQ11] and EPR [IBI+13, IBI+14] 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. [BLH08] 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. [NL08] 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. [JA09] 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.
Bibliography


trakir

 trabalho זו מציגה הגדרה ומימוש של אנליזה אינטרפרוכדורית ליבירה אוטומטית של בטיחות של תוכניות רקורסיביות עם מרחב מצבים בלתי מוגבל. הרעיון המרכזי הוא להציג שמותר לתוכנית את הפעולות המפורעות במכרנים על ידי סיכומי פקודות התוכנית. אנו מניחים כי ההשפעה של הפקודות האטומיות של התוכנית ניתנת למטר באמצעות נוסחאות של EPR. על כן, אנחנו מציעים גרסה אחרת של גישה IC3/PDR לחישוב סיכומי פרוכדורות אוניברסליים כדי歩いて את התוכנית. אנו מראים כי תוכניות ג’אווה אותן מתבצע אוספים ואובייקטים יכליים נשמרים אוניברסליים. זה מאפשר לנו להפעיל את האנליזה neuen כדי להתאים את תוכניות ג’אווה לנוסחאות EPR. על מנת לבחון את התכונות הללו, מציגים את האנליזה של חוני השימוע בıklı Z3 וקצחו בשפת ג’אווה המתרגמת תוכניות G’אווה.

EPR.
הוכחת היעדר שגיאות
שינוי מקבלי באמצעות
ישיגות תכונתי

חיבור זה הוגש כחלק מהדרישות לקבלת התואר
"מוסמך האוניברסיטה (M.Sc.)"

על ידי
אסיה פרומקין

עבודת המחקר בוצעה בהנחייתם של
פרופ' מולי שגיב וד"ר שרון שוהם