Predicate Abstraction of Java Programs with Collections

Pavel Parízek Ondřej Lhoták

David R. Cheriton School of Computer Science, University of Waterloo

{pparizek,olhotak}@uwaterloo.ca

Abstract

Our goal is to develop precise and scalable verification techniques for Java programs that use collections and properties that depend on their content.

We apply the popular approach of predicate abstraction to Java programs and collections. The main challenge in this context is precise and compact modeling of collections that enables practical verification.

We define a predicate language for modeling the observable state of Java collections at the interface level. Changes of the state by API methods are captured by weakest preconditions. We adapt existing techniques for construction of abstract programs. Most notably, we designed optimizations based on specific features of the predicate language.

We evaluated our approach on Java programs that use collections in advanced ways. Our results show that interesting properties, such as consistency between multiple collections, can be verified using our approach. The properties are specified using logic formulas that involve predicates introduced by our language.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification

Keywords Java, collections, predicate abstraction

1. Introduction

Our goal is to develop practical verification techniques for Java programs that heavily use the collections provided by the standard library. We call them *client programs* because they are clients of the collection library. We target properties that depend on the content of collections in some way — for example, (i) assertions over program variables that contain values retrieved from collections, (ii) consistency between multiple collections, and (iii) validity of method calls on collection objects depending on the current state and parameter values. For illustration, consider the Java program in Figure 1, which computes a schedule from available information about threads. Two properties of interest are the following: (1) the object variables schTh and actTh are not null before the field accesses on them, and (2) there is a key-value pair in the map from a thread ID to a ThreadInfo object for each active thread. The program fails if either of these two properties is violated during its execution. A verification technique for such Java programs and properties would be practically useful if it is precise (i.e., it does not report spurious errors) and scales to large programs with multiple classes and methods. To achieve the necessary precision, the verification technique must be path-sensitive and inter-procedural. Some form of abstraction has to be used for scalability.

A popular approach to automated software verification is to use predicate abstraction [21] and then run a model checker to analyze the abstract program, which is much simpler than the original program. The key idea behind predicate abstraction is to represent the program state using predicates defined in first-order logic with specific theories (e.g., linear arithmetic) over specific domains (e.g., integers). Each predicate gives some information about the values of program variables. The effect of individual statements on program state can be captured using weakest preconditions.

Techniques based on predicate abstraction have been proposed mainly for low-level C programs like device drivers [1, 2, 23]. They are used especially for checking assertions (reachability properties) and temporal properties like validity of event sequences (e.g., whether each operation of acquiring a lock is eventually followed by an operation that releases the same lock).

We apply predicate abstraction to Java programs and collections. The main challenges are (1) precise and compact modeling of the collection state with a small number of predicates, (2) precise modeling of operations upon collections, (3) support for preserving information between any two program code locations, and (4) efficient construction of abstract programs. Addressing these challenges is important

OOPSLA'12, October 19–26, 2012, Tuscon, Arizona, USA.

```
Map<Integer,ThreadInfo> id2thread = new HashMap();
Set<Integer> active = new HashSet();

// initialize map with data for several threads
id2thread.put(1, new ThreadInfo(1,5));
id2thread.put(2, new ThreadInfo(2,18));
id2thread.put(3, new ThreadInfo(3,10));

// some threads are put into the active state
active.add(2);
active.add(3);

List<Integer> schedule = new LinkedList();

Iterator<Integer> actIt = active.iterator();
while (actIt.hasNext()) {
  int actID = actIt.next();
  ThreadInfo actTh = id2thread.get(actID);

  for (int i = 0; i < schedule.size(); i++) {
    int schID = schedule.get(i);
    ThreadInfo schTh = id2thread.get(schID);

    if (actTh.priority > schTh.priority) {
      schedule.add(i, actID);
      break;
    }
  }
}
```

Figure 1. Example: thread scheduling

for scalability to large programs and to enable verification of interesting properties.

Many program verification techniques and frameworks that target collections and data structures have been proposed in recent years — symbolic execution for data structures [25], shape analysis based on predicate abstraction [7, 16, 37] or other abstractions [6], logics and decision procedures for heap data structures [7, 29], techniques based on separation logic [18], and various techniques for verification of linked list implementations [9, 10, 28] and general programs that use collections against functional specifications [40, 41]. Most of these approaches model collections at the representation level. They explicitly consider the internal data structures that implement higher-level collections, heap nodes that represent objects, and pointers between the heap nodes — for example, the nodes and edges of the tree that is used internally to implement the TreeMap class from the Java library. However, it is not necessary to consider the internal representation of collections and their implementation details to verify programs that use collections through a well-defined public interface (API), since the functional behavior of a client program does not depend on the internal data structures and the shape of heap regions used by them.

There also exist approaches to modeling program behavior that can be used for collections, such as the popular contract specification languages JML [30] and Spec# [5], and various logics for reasoning about data structures [27, 33, 45]. However, specifications defined in the contract languages (preconditions, postconditions, and invariants) have limited expressive power, because they cannot refer to local variables and they can be associated only with particular code locations (e.g., method boundaries). The corresponding verification techniques [4, 19] are modular at the granularity of procedures, which means that they are efficient, but they cannot be used for checking properties that span multiple procedures (not in a caller-callee relationship) and for consistency properties between collections owned by unrelated objects. The logics for data structures are either very complex and not suitable for predicate abstraction, or they are not supported by state-of-the-art decision procedures yet.

Our approach is to verify client Java programs in a path-sensitive and inter-procedural manner by using predicate abstraction and modeling Java collections at the interface level. This approach requires fewer predicates about the program state than if collections were modeled at the representation level, and it therefore enables more efficient and scalable verification. We consider only the state and behavior observable by client programs as specified by the textual API documentation. We assume that the implementation of the collection classes is correct and conforms to the documentation. The latter should be true for every implementation of the Java collections API.

In this paper, we describe our approach to modeling Java collections at the interface level using predicates and its usage for practical verification of Java programs that use collections. We define the following:

- a translation from the Java collections into abstract maps,
- a predicate language for modeling the externally observable state of collections, and
- weakest preconditions that capture changes of the observable state by API methods.

We adapt existing techniques for constructing abstract programs that were introduced for programs in the C language [2, 3]. We describe aspects of the construction that are specific to our approach and target domain. In particular, we designed optimizations that exploit specific features of the predicate language.

Finally, we evaluate our approach on Java programs that use collections in advanced ways. Our experiments show that interesting properties of the Java programs can be successfully verified using our approach. In particular, our predicate language allows to capture all information about the state of collections that is necessary to verify the properties specified for the benchmark programs.

2. Overview

We start with a general overview of our approach on the example Java program in Figure 1.

In order to verify the properties mentioned in the previous section, it is necessary to have information about the content of the collections pointed to by the variables id2thread and active. The variables schTh and actTh refer to data retrieved from collections. The stored data form the observable state of both collections in this case. Changes of their content by operations like add and put must be captured too.

We do not have to consider the internal implementation of collection classes (e.g., HashMap), because the properties are specific to the given client program. However, note that the collections cannot be abstracted away completely, because it would not be possible to verify the given properties without any information about the content of the maps.

The necessary information about the collections is captured by predicates such as $mget(\mathrm{map}, \mathrm{id2thread}, 1) \neq \bot$, $mget(\mathrm{map}, \mathrm{active}, 2) = \mathrm{true}$, and $\mathrm{actTh} = \mathrm{null}$. The predicates describe the content of the map id2thread and the set active, and the value of the variable actTh.

The actual verification of the given Java program with our approach consists of three steps:

1. predicates about collections and other program variables that are necessary for the verification of the properties are acquired (provided by the user or inferred automatically);
2. an abstract program is generated for the given client Java program, a set of properties, and a set of predicates about collections and other program variables;
3. the abstract program is verified using the Java Pathfinder model checker.

The abstract program looks like the one in Figure 2 — it is a simplified fragment. It contains boolean variables, assignment statements, and Java control-flow structures. For each assignment statement, the new value of a target boolean variable is determined using the weakest preconditions defined later in this paper.

Java Pathfinder (JPF) [44] performs exhaustive traversal of the abstract program. If an assertion violation is detected by JPF, then relevant information is extracted from the error trace and reported to the user.

```
boolean bv1 = false; // mget(map,id2thread,1) != bot
boolean bv2 = false; // mget(map,active,2) = true
boolean bv3 = true; // actTh = null
// more boolean variables for other predicates

// statement: id2thread.put(1, new ThreadInfo(1,5))
atomic { bv1 = true; ... }

// statement: active.add(2)
atomic { bv2 = true; ... }

while (...) {
  // statement: actTh = id2thread.get(actID)
  if (bv1 && ...) bv3 = false;

  // property check
  if (bv3) assert false : "actTh == null";

  ...
}
```

Figure 2. Example: abstract program

3. Preliminaries

Predicate abstraction. An abstract program is constructed for a given set of predicates about variables in the input program. The predicates are supplied by the user or inferred automatically. Either a single set of predicates can be used for the whole input program [2] and associated with every statement, or each statement s in the program can be associated with a (possibly different) set of predicates that covers only the aspects of program state relevant for the statement [23].

For each statement s in the original program and each predicate p whose truth value may be changed by the execution of s, the abstraction (abstract program) must capture the change of the truth value of p after the execution of s. The effect of individual statements on the truth values of the predicates, i.e. on the program state, is determined by weakest preconditions in the following way. For the predicate $p \in P$ associated with the statement s, the new truth value of p after the execution of s is determined using the weakest precondition $WP(s, \nu(p))$ of s with respect to $\nu(p)$, where $\nu(p)$ is a literal over p (i.e., p or $\neg p$). Let the set $\nu(P)$ represent the actual truth values of all predicates from the set P just before execution of the statement s, i.e. for each $p_i \in P$ the set $\nu(P)$ contains p_i or $\neg p_i$. The new value of p after the execution of s is (a) true if the formula $\bigwedge \nu(P) \Rightarrow WP(s, p)$ is valid, (b) false if the formula $\bigwedge \nu(P) \Rightarrow WP(s, \neg p)$ is valid, and (c) a non-deterministic boolean value ($\mathrm{unknown}$) otherwise. A decision procedure (e.g., an SMT solver) is used to check the validity of the formulas.

For example, given the statement $v := e$ and the predicate $v < 2$, the weakest precondition $e < 2$ determines the new truth value of the predicate after the assignment.

Array theory. The array theory [34] for first-order logic defines a pair of function symbols that represent array accesses. The symbol $select(a, i)$ returns the element $a[i]$. The symbol $store(a, i, v)$ returns a new array a' which is equal to a except that $a'[i] = v$. The meaning of the symbols is defined by the *read-over-write axiom*: $select(store(a, i, e), i) = e$. The elements of a with indices not equal to i are not modified by the $store$ function.

4. Modeling Java Collections with Predicates

Our predicate language enables faithful modeling of Java collections with respect to state and behavior observable from client programs. Specifically, it can be used to model the common usage patterns of Java collections in real programs and the dependence of program behavior on the collection state. We support both associative collections (maps and sets) and position-based collections (lists), and iterators over these collections. We also support nested collections (e.g., a map from integers to linked lists). First we introduce features of the predicate language on several example programs, and then we provide formal definitions.

4.1 Examples

Associative collections. Consider again the program fragment in Figure 1. The program would crash at the field access actTh.priority if the object variable actTh has the null value. To decide whether the variable actTh can be null, we need information about all key-value pairs in the map id2thread, all elements of the set active, and the current value of the variable actID.

If the map id2thread contains some mapping for every possible key stored in actID and the associated value is not null, then the value of actTh cannot be null. We capture the presence of a key-value pair (k, v) in the map id2thread by the predicate $mget(\text{map}, \text{id2thread}, k) = v$ and absence of a mapping for a key k by the predicate $mget(\text{map}, \text{id2thread}, k) = \bot$. The symbols $mget$, map, and $\bot$ are introduced by our predicate language.

The possible values of the variable actID correspond to elements of the set active. We model sets as maps from possible elements to boolean values, where the boolean value for a particular element indicates its presence in the set. For example, the presence of a value i in the set active is captured by the predicate $mget(\text{map}, \text{active}, i) = \text{true}$.

The property $\text{actTh} \neq \text{null}$ holds if there is some mapping in id2thread for every element of the set active, i.e. if the formula $\forall i : (mget(\text{map}, \text{active}, i) = \text{true} \Rightarrow mget(\text{map}, \text{id2thread}, i) \neq \bot)$ holds. To determine the truth value of the formula, we need the following predicates: $mget(\text{map}, \text{id2thread}, 1) \neq \bot$, $mget(\text{map}, \text{id2thread}, 2) \neq \bot$, $mget(\text{map}, \text{id2thread}, 3) \neq \bot$, $mget(\text{map}, \text{active}, 2) = \text{true}$, and $mget(\text{map}, \text{active}, 3) = \text{true}$.

Ordered lists. The program in Figure 1 also uses the list schedule. The program accesses list elements at specific positions (via schedule.get(i)) and adds new elements at a specific position (via schedule.add(i, actID)). Such method calls would fail if the position argument (index) is out of the valid range.

We model lists as maps from arbitrary integer values to stored elements. However, the integers do not represent element positions (see Section 4.2). The presence of the element e in the list schedule is captured by the predicate $mget(\text{map}, \text{schedule}, k) = e$ for an arbitrary integer k.

```
class Image {
  public List<Rectangle> rectangles;
  public int[][] pixels;

  public static void main(String[] args) {
    Image img = new Image();
    img.load();
    img.render();
  }

  public void load() {
    rectangles = new ArrayList();
    pixels = new int[50][50];
    rectangles.add(new Rectangle(5, 10, 20, 20, 3));
    rectangles.add(new Rectangle(20, 5, 10, 35, 1));
  }

  public void render() {
    Iterator<Rectangle> recIt = rectangles.iterator();
    while (recIt.hasNext()) {
      Rectangle rec = recIt.next();
      for (int i = 0; i < rec.width; i++)
        for (int j = 0; j < rec.height; j++)
          pixels[rec.left+i][rec.top+j] = rec.color;
    }
  }
}
```

Figure 3. Example: rendering image with several rectangles

Reasoning about collection size. To determine whether the index argument of a call to schedule.get(i) is in the valid range, we need information about the current size of the list schedule and the current value of the variable i. The valid range of the index argument i is specified by the formula $\text{i} \geq 0 \wedge \text{i} < msize(\text{msz}, \text{schedule})$, where the expression $msize(\text{msz}, \text{schedule})$ represents the size of the list schedule. We need the following predicates to capture all of the necessary information: $msize(\text{msz}, \text{schedule}) = 0$, $msize(\text{msz}, \text{schedule}) = 1$, $msize(\text{msz}, \text{schedule}) = 2$, $\text{i} = 0$, $\text{i} = 1$, and $\text{i} = 2$,

Modeling object fields and arrays. The program in Figure 3 renders an image containing several rectangles. For each rectangle, it sets the relevant pixels to a given color. An important correctness property is the consistency between the list of rectangles and the array of pixels — if a pixel has some color then there must be some rectangle at its position, and vice versa. To verify this property, we need information about (1) the values of the fields of Rectangle objects stored in the list rectangles and (2) the values of specific elements of the array pixels. Moreover, the information about objects stored in the list rectangles must be preserved between the calls of methods load and render.

We model the values of fields by expressions of the form $fread(f, o)$, where f is the field name and o is the object variable, and the values of array elements by expressions of

```
List<Cyclist> cyclists = new ArrayList();

cyclists.add(new Cyclist(2, 3725, 5));
cyclists.add(new Cyclist(56, 3569, 10));
cyclists.add(new Cyclist(40, 3766, 50));

TreeMap<Integer,Cyclist> results = new TreeMap();

Iterator<Cyclist> cycIt = cyclists.iterator();
while (cycIt.hasNext()) {
  Cyclist cl = cycIt.next();
  results.put(cl.time - cl.bonus, cl);
}

Collection<Cyclist> resCyclists = results.values();
Iterator<Cyclist> resIt = resCyclists.iterator();

Cyclist bestCL = resIt.next();
int bestTime = bestCL.time - bestCL.bonus;
print(bestCL.id + "␣" + bestTime);

while (resIt.hasNext()) {
  Cyclist cl = resIt.next();
  int diff = cl.time - cl.bonus - bestTime;
  print(cl.id + "␣" + diff)
}
```

Figure 4. Example: processing results of a cycling race

the form $aread(\mathrm{arr}, a, i)$, where a is an array variable and i is the index. Arrays with multiple dimensions are modeled by nested $aread$ expressions.

The consistency property between the list of rectangles and the array of pixels is expressed with the formula

$$\forall x, y : (aread(\mathrm{arr}, aread(\mathrm{arr}, \mathrm{pixels}, y), x) \neq 0 \Rightarrow$$
$$(\exists q : mget(\mathrm{map}, \mathrm{rectangles}, q) = r \ \wedge \ fread(\mathrm{top}, r) \leq y \wedge$$
$$fread(\mathrm{top}, r) + fread(\mathrm{height}, r) \geq y \wedge fread(\mathrm{left}, r) \leq x \wedge$$
$$fread(\mathrm{left}, r) + fread(\mathrm{width}, r) \geq x))$$

where the variable r points to a Rectangle object and q is a logic variable.

Global predicates over object fields and constants allow preserving information about collections between method calls (inter-procedurally). Here, we use predicates of the form $mget(\mathrm{map}, \mathrm{rectangles}, c) = r$, where the symbol c denotes an integer constant — their truth values would be set during the execution of the method load and used to determine the property status in the method render.

Iteration and sorted collections. The program in Figure 4 takes as input the raw (unsorted) results of a cycling race and prints the final results in a standard format — first the winner's name and race time, and then differences for all other riders. The final results are printed correctly if the first element of the map results with respect to the iteration order has the smallest key.

The iteration order over a specific collection coll is modeled with predicates of the form $morder(\mathrm{mit}, \mathrm{coll}, q_1, q_2)$, where q_1 and q_2 are keys in the map that represents coll. The key k is the first one in the iteration order for the map results if the predicate $morder(\mathrm{mit}, \mathrm{results}, \bot, k)$ holds — we use the symbol $\bot$ also to capture iteration boundaries. If the predicate $morder(\mathrm{mit}, \mathrm{results}, \mathrm{resIt}, \bot)$ holds for the iterator variable resIt, then it has reached the end and any additional call of next() would fail. Therefore, the property $\neg morder(\mathrm{mit}, \mathrm{results}, \mathrm{resIt}, \bot)$ must hold before every call of next on the iterator to guarantee that an exception is not thrown for an attempt to iterate over the end of the collection.

The map results is sorted correctly if for any two keys k_1 and k_2 such that the predicate $morder(\mathrm{mit}, \mathrm{results}, k_1, k_2)$ holds, it is also true that $k_1 < k_2$. Sorted collections are implemented using the proper iteration order.

Note that the iterator resIt is associated with the collection resCyclists that represents a list view of all values in the map results. We model this association with the predicate $mvalues(\mathrm{mvs}, \mathrm{results}, \mathrm{resCyclists})$.

Nested collections. The program in Figure 5 models a simple data flow analysis. It involves two collections — a map from integers to integer arrays that represents a control-flow graph (IDs of nodes are mapped to successor nodes), and a map from integers (node IDs) to sets that represent facts associated with nodes.

Reasoning about the content of sets of facts that are stored in the map cfgnode2facts is possible through formulas like $mget(\mathrm{map}, mget(\mathrm{map}, \mathrm{cfgnode2facts}, id), e) = \mathrm{true}$, where e is a particular data flow fact and id is an ID of a node in the control-flow graph. Similarly, formulas like $aread(\mathrm{arr}, mget(\mathrm{map}, \mathrm{cfg}, 2), 0) = 3$ capture the content of arrays stored in the map cfgnode2facts and thus enable reasoning about the control-flow graph.

The association between the map cfg and the set cfgNodes, which provides a view of the map keys, is captured by the predicate $mkeys(\mathrm{mks}, \mathrm{cfg}, \mathrm{cfgNodes})$.

Summary. The examples above show that the state of a Java collection observable by a client program consists of its content, size, iteration order, and views. These aspects of collection state must be captured to allow precise reasoning about the behavior of client programs.

In the rest of this section, we provide a more formal description of our approach to modeling Java collections.

4.2 Abstract Maps

We define our modeling approach on *abstract maps* that support specific operations and *iterators* over keys.

Each abstract map is a set of key-value pairs with a specific iteration order over the keys. The following operations are supported: get, put, putAhead, remove, clear, size, containsKey, containsValue, createIterator, keysView, valuesView, and findKey. The operation $\mathrm{putAhead}(k, v, l)$ inserts a new key-value pair (k, v) into the map such that

```
// control flow graph
// a map from node IDs to successor nodes' IDs
Map<Integer, int[]> cfg = ... // initialization

// facts associated with cfg nodes
Map<Integer, Set<Integer>> cfgnode2facts = ...

// initialize with empty set of facts for each node
Set<Integer> cfgNodes = cfg.keySet();
Iterator<Integer> cfgIt = cfgNodes.iterator();
while (cfgIt.hasNext()) {
  int nodeID = cfgIt.next();
  cfgnode2facts.put(nodeID, new HashSet<Integer>());
}

List<Integer> queue = new LinkedList<Integer>();

queue.add(1); // start with the entry node

while (queue.size() > 0) {
  int cfgnodeID = queue.remove(0);
  Set<Integer> oldFacts = cfgnode2facts.get(cfgnodeID);

  Set<Integer> newFacts = new HashSet<Integer>();
  newFacts.addAll(oldFacts);
  ... // facts are updated in some way
  cfgnode2facts.put(cfgnodeID, newFacts);

  if ( ! oldFacts.equals(newFacts) ) {
    // update the queue based on CFG
    int[] succ = cfg.get(cfgnodeID);
    for (i = 0; i < succ.length; i++) queue.add(succ[i]);
  }
}
```

Figure 5. Example: a simple data flow analysis

k precedes l in the iteration order. The operation $\mathrm{findKey}$ returns a key for the given value. Views over maps are also supported. The operation $\mathrm{keysView}$ returns a set of keys that is associated with the map and the operation $\mathrm{valuesView}$ returns a list of values.

Iterators are modeled using the iteration order over map keys. The current position of a newly created iterator is at the beginning of the iteration order (ahead of the first key). The following operations on abstract iterators are supported: $\mathrm{hasMore}$, $\mathrm{getCurrent}$, and $\mathrm{moveNext}$. The operation $\mathrm{getCurrent}$ returns a key such that the current iterator's position is behind the key.

In the rest of this section, we describe how Java collections are implemented using the abstract maps.

Maps. Map classes and interfaces provided by the Java collection API directly correspond to the abstract map. Methods defined by the interface java.util.Map are translated to operations supported by the abstract map in a natural way.

Iterators. The methods of the interface java.util.Iterator are translated into sequences of operations upon abstract iterators and maps. For example, the method remove, which deletes the key-value pair at the current position, is implemented via the operation $\mathrm{getCurrent}$ on the iterator and subsequent remove on the associated map.

Sets. We model a *set* of elements of type T as an abstract map with keys of type T and boolean values, where the boolean value for a specific key indicates the presence of the corresponding element in the set. The methods defined by the interface java.util.Set directly correspond to operations supported by the abstract map.

Sorted associative collections. In the case of Java classes that provide the functionality of sorted maps and sets (e.g., TreeMap and TreeSet), all methods except addition of new elements are translated directly into operations on the abstract maps. The addition of a key-value pair (k, v) into a map is modeled by a sequence of operations that consists of two parts. The first step is to find the key l that would be the immediate successor of k in the iteration order, and the second step is to insert the new key-value pair using the $\mathrm{putAhead}$ operation.

Lists. An *ordered list* of elements of type T is modeled by an abstract map where the keys are positive integers and the values have the type T. The keys do not represent the positions of stored elements. Instead, the actual position (index) of any element can be deduced from the iteration order over keys. This approach eliminates the need to shift many keys in the abstract map upon any modification of the list — it is only necessary to update the set of key-value pairs stored in the map and make a local change to the iteration order.

The methods defined by the interface java.util.List are translated into sequences of operations over the abstract map. For methods that take an index argument (e.g., get(i)), the sequence of operations has two parts — the purpose of the first part is to find a key whose position is equal to the value of the index argument, and the second part implements the actual task using the operations supported by the abstract map. For addition methods, the sequence also contains operations whose purpose is to find the smallest unused key that could be associated with the new element. Internal temporary variables are used in the translation (e.g., to store the key corresponding to a given index).

Other position-based collections, such as a queue or a stack, are special cases of an ordered list, so they can be modeled using an abstract map in the same way.

4.3 Client Programs

We define our modeling approach formally on programs with the grammar in Figure 6. A program consists of one or more classes, where each class has zero or more fields and methods. The programs can use abstract maps and iterators, integer and boolean variables, objects, field accesses, and arrays. The grammar distinguishes between variables,

$v \in \mathit{Variables}$ $\quad f \in \mathit{FieldNames}$ $\quad m \in \mathit{MethodNames}$

$\mathit{Program} ::= \mathit{Class}^+$ $\quad \mathit{Class} ::= T\ \{\mathit{Field}^*\ \mathit{Method}^*\ \}$ $\quad \mathit{Field} ::= T\ f$

$\mathit{Method} ::= T\ m(v, \ldots, v)\{\ \mathit{MethodBody}\ \}$ $\quad \mathit{MethodBody} ::= \mathit{Stmt}^*$

$e \in \mathit{Expr} ::= ic \mid \text{true} \mid \text{false} \mid \text{null} \mid v \mid e.f \mid e[e] \mid \text{unop}\ e \mid e\ \text{binop}\ e$

$\mathit{Stmt} ::= v := e \mid v := \texttt{new C} \mid e.f := e \mid v := \texttt{new T[]} \mid e[e] := e \mid \texttt{assert}\ e \mid \mathit{Stmt}\ ;\ \mathit{Stmt} \mid v := e.m(e, \ldots, e) \mid$
$\quad \texttt{while}\ e\ \texttt{do}\ \mathit{Stmt} \mid \texttt{if}\ e\ \texttt{then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt} \mid v := \texttt{new map} \mid v := e.\texttt{createIterator}() \mid \mathit{MapOper} \mid \mathit{IterOper}$
$\mathit{MapOper} ::= v := e.\texttt{get}(e) \mid v := e.\texttt{findKey}(e) \mid v := e.\texttt{size}() \mid e.\texttt{put}(e, e) \mid e.\texttt{putAhead}(e, e, e) \mid e.\texttt{remove}(e) \mid$
$\quad e.\texttt{clear}() \mid v := e.\texttt{containsKey}(e) \mid v := e.\texttt{containsValue}(e) \mid v := e.\texttt{keysView}() \mid v := e.\texttt{valuesView}()$
$\mathit{IterOper} ::= v := e.\texttt{hasMore}() \mid v := e.\texttt{getCurrent}() \mid e.\texttt{moveNext}()$

Figure 6. Grammar of a client program

class names (types), field names, method names, and expressions. The symbol ic represents an integer constant. Objects can have fields of all supported types. Expressions of any supported type can be used as keys and values in maps. In particular, nested collections are supported because a map can be used as a key or value in another map.

The grammar requires that the result of each method call be saved into a variable. It does not allow using method calls on maps and iterators directly in composite expressions such as if (m.contains(k)) x = 5 or x = m.get(k) + 5. An example of a valid code fragment is b = m.contains(k); if (b) x = 5.

4.4 Predicate Language

Information about the program state, including the observable state of maps and iterators, is expressed by formulas in first-order logic with the theory of linear arithmetic and the array theory. The formulas are built from atomic predicates over program variables. The domain of formulas is the set of integer numbers. All other data types are translated into integers. References are modeled using integer values that represent memory addresses. Equality between reference variables models aliasing.

We introduce several function symbols and predicate symbols for modeling the observable state of maps — mget, $\mathit{mupdate}$, msize, $\mathit{mresize}$, morder, mkeys, and $\mathit{mvalues}$. All of these symbols are aliases for the functions select and store defined by the array theory. The purpose of the aliases is to explicitly indicate which aspect of the state of a map is captured with the given predicate. We use special arrays with the names map, msz, mit, mks, and mvs.

The function symbol mget is used to denote the presence of a specific key-value pair in a given map. The expression $\mathit{mget}(\mathrm{map}, m, k)$ returns the value associated with the key k in the map m. The expression $\mathit{mupdate}(\mathrm{map}, m, k, v)$ returns a new array that is equal to map except that the key k is now associated with the value v in the map m. For example, the truth value of the predicate $\mathit{mget}(\mathrm{map}, m, 1) = 5$ says whether the map m contains the key-value pair $(1, 5)$. We use predicates of the form $\mathit{mget}(\mathrm{map}, m, k) = \bot$ with the special symbol $\bot$ to express the fact that the map m contains no value for the key k. Arbitrary nesting of function symbols mget and $\mathit{mupdate}$ is possible. The following axiom holds for the symbols: $\mathit{mget}(\mathit{mupdate}(\mathrm{map}, m, k, v), m, k) = v$. The full content of some map (i.e., a set of the stored key-value pairs) can be expressed using a conjunction of predicates with the function mget. Information about a value stored in a nested map can be expressed by a predicate of the form $\mathit{mget}(\mathrm{map}, \mathit{mget}(\mathrm{map}, m_1, k_1), k_2) = v$.

The function symbols msize and $\mathit{mresize}$ are used to model the size of a map. The expression $\mathit{msize}(\mathrm{msz}, m)$ returns the current size of the map m. The expression $\mathit{mresize}(\mathrm{msz}, m, sz)$ returns a new array that is equal to msz except that the element corresponding to the current size of the map m is set to sz. The following axiom holds for the function symbols: $\mathit{msize}(\mathit{mresize}(\mathrm{msz}, m, sz), m) = sz$.

We use the predicate symbol morder to model the iteration order. The expression $\mathit{morder}(\mathrm{mit}, m, k_1, k_2)$ returns true if the key k_1 precedes the key k_2 in the iteration order for the map m. An iterator variable can be used as an argument for the symbol morder to express the current position of a specific iterator. The truth value of the predicate $\mathit{morder}(\mathrm{mit}, m, it, k)$ says whether the iterator's current position is just ahead of the key k. Bounds are represented by the $\bot$ symbol. For example, the value of the expression $\mathit{morder}(\mathrm{map}, m, \bot, 2)$ says whether the key 2 is the first element in the iteration order for m. The key k is the last element in the iteration order of the map m if the predicate $\mathit{morder}(\mathrm{mit}, m, k, \bot)$ holds. The full iteration order for a map can be expressed via a conjunction of predicates with the symbol morder.

The predicate symbols mkeys and $\mathit{mvalues}$ are used to capture associations between maps and views. The expression $\mathit{mkeys}(\mathrm{mks}, m, ms)$ returns true if the map ms represents a set view of all keys in the map m. The expression

Our predicate language	Array theory
$mget(\text{map}, m, k)$	$select(\text{map}, m, k)$
$mupdate(\text{map}, m, k, v)$	$store(\text{map}, m, k, v)$
$msize(\text{msz}, m)$	$select(\text{msz}, m)$
$mresize(\text{msz}, m, sz)$	$store(\text{msz}, m, sz)$
$morder(\text{mit}, m, d_1, d_2)$	$select(\text{mit}, m, d_1, d_2)$
$mkeys(\text{mks}, m, ms)$	$select(\text{mks}, m, ms)$
$mvalues(\text{mvs}, m, ml)$	$select(\text{mvs}, m, ml)$
$fread(f, o)$	$select(f, o)$
$fwrite(f, o, e)$	$store(f, o, e)$
$aread(\text{arr}, a, i)$	$select(\text{arr}, a, i)$
$awrite(\text{arr}, a, i, e)$	$store(\text{arr}, a, i, e)$

Table 1. Translation from symbols introduced by our predicate language to the array theory functions

$mvalues(\text{mvs}, m, ml)$ returns true if the map ml represents a list view of all values in the map m.

The symbol $\perp$ is modeled by a logic constant of an integer type whose value is different from the values of all other expressions that can appear in a given formula.

We model objects, fields, and arrays using the approach described in [13], which is also based on the array theory. For each field name, there is a one-dimensional array indexed by objects that have the given field. The function expression $fread(f, o)$ returns the value of the field f of the object o and the expression $fwrite(f, o, e)$ encodes update of the field f in o with the new value e. Array accesses are captured by the function symbols $aread$ and $awrite$. The expression $aread(\text{arr}, a, i)$ returns the element of the array a with the index i. Nested arrays (with multiple dimensions) are also possible. The value null of a reference type is modeled by a logic constant of an integer type whose value is different from the values of all other expressions that can appear in a given formula — in particular, it is different from the constant that represents $\perp$.

The value of a field f of an object stored in the map m can be expressed by the predicate $fread(f, mget(\text{map}, m, k)) = v$. The symbol $fread$ can be also used as an argument for the function symbol $mget$. For example, the result of the code fragment this.data.put(5,v) is captured by the predicate $mget(\text{map}, fread(data, this), 5) = v$.

Table 1 summarizes the mapping from the function symbols introduced by our predicate language to the functions defined by the array theory.

Locality. We distinguish between predicates and formulas that (1) refer only to constants and static fields, (2) refer also to object fields, and (3) refer to all kinds of variables including method local variables. Each formula is associated with a certain scope — the whole program, a specific class, or a specific method. Using different sets of predicates for different scopes is important especially in cases where local variables of the same name are used in two methods or two classes define fields with the same name.

For each predicate and formula that is associated with a specific method, we consider the actual scope inside the method body in which it is relevant. A special case are predicates over temporary variables used in the translation from method calls on Java collections to operations on the abstract maps and iterators. The scope of such a predicate is the particular method call.

A very important feature of our predicate language is the support for preserving information about objects and their fields between method calls. However, this requires usage of suitable predicates with respect to the scope of program variables. Let p be a predicate that describes the content of a map pointed to by a specific variable m. If the variable m is defined as a field of some object and the information captured by p must be preserved between method calls (for the whole lifetime of m), then other expressions referenced by p must have at least the same scope as m. For example, consider the predicates $mget(\text{map}, fread(data, this), 2) = 3$ and $mget(\text{map}, fread(data, this), id) = 5$, where id is a local variable of some method mth. The information expressed by the first predicate is preserved for the whole lifetime of the map pointed to by this.data. On the other hand, the information captured by the second predicate will not be preserved after the exit from mth because of the local variable id.

4.5 Modeling Statements by Weakest Preconditions

The execution of any statement, including operations on abstract maps and iterators, may change the truth values of some predicates that cover relevant aspects of the program state. We distinguish two categories of supported operations on maps and iterators associated with the maps: (1) *updating operations* that change the observable state of the target map m and possibly some other maps, and (2) *query operations* that change the current value of a program variable that is used to store the result of the operation. Maps other than m can be changed by an updating operation because of propagation between views and the underlying map — if the operation changes the state of the map m, then it also changes the state of maps that represent views over m and maps over which m is a view.

We specify changes of the truth values of predicates using weakest preconditions. They reflect the following:

- the semantics of the Java language, including assignments to local variables, field accesses, and object construction;
- the behavior of the Java collection API methods that is described in the textual documentation;
- aliasing between reference variables, including map variables, and aliasing between map elements.

Statement s	Predicate p	$WP(s,p)$
r = m.get(k)	$r = e$	$\exists q_m : q_m = m \wedge e = mget(\mathrm{map}, q_m, k)$
m.put(k,v)	$mget(\mathrm{map}, m', k') = v'$	$mget(mupdate(\mathrm{map}, m, k, v), m', k') = v'$
	$msize(\mathrm{msz}, m')$ relop u	$ite($ $(m = m' \wedge mget(\mathrm{map}, m', k) = \bot) \vee m \neq m'$, $msize(mresize(\mathrm{msz}, m, msize(\mathrm{msz}, m) + 1), m')$ relop u, $msize(\mathrm{msz}, m')$ relop u $)$
it.moveNext()	$morder(\mathrm{mit}, m', it, \bot)$	$\exists q_k : (morder(\mathrm{mit}, m', it, q_k) \wedge morder(\mathrm{mit}, m', q_k, \bot))$

Table 2. Weakest preconditions: selected examples

As an illustration, Table 2 shows the weakest preconditions for selected combinations of the statements of the language from Figure 6 and predicates about map content. The symbols q_k and q_m denote logic variables. The construct $ite(a, b, c)$ is defined as $ite(a, b, c) \equiv (a \wedge b) \vee (\neg a \wedge c)$. The logic variable q_m is used in the weakest precondition for the statement r = m.get(k) and the predicate $r = e$ to capture aliasing between maps. In the case of the statement m.put(k,v), the weakest preconditions capture aliasing implicitly based on whether the symbols m and m' have equal values and point to the same element of the underlying array (map or msz).

The complete definition of the weakest preconditions for all statements and predicates is in Appendix A.

5. Construction of Abstract Programs

An abstract program is constructed from the following input: the original Java program, a set P_I of predicates over program variables, and a set of properties.

We assume that one set of predicates is defined for the whole input program, and this set applies to every statement. The properties are defined as logic formulas built from atomic predicates defined in our language. Each property is associated with specific code locations in the input program. For example, there can be a property $o \neq \mathrm{null}$ in front of a field access on a specific object variable o.

Abstract programs are Java programs in which the only primitive data type is boolean. The programs use methods provided by the Java Pathfinder API — most notably the method getBoolean of the Verify class, which returns the non-deterministic boolean value unknown, and the call Verify.assertTrue(false) that represents a violated assertion.

The process of abstraction preserves all classes defined in the original program (i.e., custom non-library data types). An abstracted method is generated for each method in the original program.

We adapt existing techniques for the construction of abstract programs [2, 3] that were introduced for the C language. We summarize the basic approach first and then discuss modifications and aspects specific to our approach.

Basic approach. Each predicate p in the set P_I is represented with a boolean variable (an object field or a method local variable) in the abstract program, which encodes the truth value of p. The control flow structures in the original Java program are preserved in the abstract program. Support for exceptions is very limited — throwing an exception is modeled as an unconditional error (assert false). Assignment statements and method calls on collection objects are replaced with their abstraction in the way described below.

Using the terminology of [3], the abstraction algorithm that we use is based on computing the abstract "post" operator for every statement in every method of the input program.

Each statement s in the original program is processed separately. First, a set $P_s \subseteq P_I$ of predicates about the program state whose truth values may be changed due to the execution of s is identified. For each predicate $p \in P_s$, the set $P_{s,p} \subseteq P_I$ of predicates that may determine the new truth value of p is also identified. The sets P_s and $P_{s,p}$ are identified based on the semantics of s. For example, when processing the statement x := e, it is necessary to update all predicates over the variable x (and possible aliases of x) based on the predicates over the expression e. We present the algorithm for selecting relevant predicates in Section 5.1.

The abstraction of the statement s contains code that updates boolean variables that represent predicates in P_s. It has the form of a sequence of if-else statements, where one such statement is generated for each predicate $p \in P_s$. The if-else statement for p represents a truth table over the set $P_{s,p}$. It contains branches that correspond to cubes over $P_{s,p}$. A specific *cube* over $P_{s,p}$ is the set $\nu(P_{s,p})$ that for each $p_i \in P_{s,p}$ contains either p_i or $\neg p_i$. Every branch contains an assignment statement that sets a new value of the boolean variable b_p that represents p.

The new value of the variable b_p (i.e., the new truth value of p) in a particular branch of the if-else statement is computed based on the weakest precondition $WP(s, p)$ and the cube $\nu(P_{s,p})$ in the way described in Section 3. One call of the decision procedure (an SMT solver) is made for every cube over $P_{s,p}$ that must be considered. We describe in Section 5.1 which cubes may be safely pruned. The input for a call of the decision procedure is the formula $\bigwedge \nu(P_{s,p}) \wedge F_{aux} \Rightarrow WP(s, p)$, where the symbol F_{aux} represents auxiliary formulas (see below).

For each property, the code if (!b) Verify.assertTrue(false) is generated at the corresponding location in the abstract program, where b is a boolean variable that contains the truth

value of the property formula. It is derived from the values of the boolean variables that represent atomic predicates in the property formula.

Auxiliary formulas. We use *auxiliary formulas* to express general properties and semantics of the Java language, including its type system and heap model, and the semantics of abstract maps. They capture properties and semantics that are not specific to any statement s or predicate p, and because of that they are not a part of the weakest preconditions.

We define auxiliary formulas to capture the following semantic constraints:

- program variables of different types cannot be equal — this applies especially to pairs of integer variables and map variables, which can both appear as map values (as v in the predicate $mget(\mathrm{map}, m, k) = v$), and to pairs of integer variables and iterator variables, which can be used as the last two arguments in $morder(\mathrm{mit}, m, d_1, d_2)$;
- the symbols $\perp$ and null cannot be equal to any program variable or any integer constant explicitly used in the program, and they must differ from each other;
- the predicate $mget(\mathrm{map}, m, k) = \perp$ is true if no predicate specifying a value for the key k in the map m is true in the current program state;
- if some predicate that specifies a value for the key k in the map m is true in the current program state, then the predicate $mget(\mathrm{map}, m, k) = \perp$ cannot be true;
- the iteration order over a map is an anti-symmetric relation, i.e., if the predicate $morder(\mathrm{mit}, m, k_1, k_2)$ is true then $morder(\mathrm{mit}, m, k_2, k_1)$ cannot be true.

The constraints enforce a semantically correct heap model, where distinct objects have different heap addresses.

In addition, we use auxiliary formulas to restrict the values of logic variables used in the weakest preconditions. For each logic variable q in $WP(s, p)$ and a specific cube $\nu(P_{s,p})$, expressions that can instantiate q are identified by syntactic matching between predicates in the cube and predicates in $WP(s, p)$. The formulas permit q to get the value e only if q and e have matching positions in some predicates and the relevant elements of the cube evaluate to true.

Aliasing. For every statement upon a target variable v of a reference type in the input program (e.g., a field access or a method call), the abstract program contains the code that updates the truth values of relevant predicates over v and predicates over other variables that may be aliased with v. The variables possibly aliased with v are taken from equality predicates of the structure $v = w$, where the symbol w represents some other program variable of a reference type. Our weakest preconditions guarantee that aliasing between variables is reflected correctly.

Views over maps. The abstract program must also correctly reflect updates of views for any given map m. Each update of the map m must be propagated to all views over m, and each update of some view over the map m must be propagated to m and all other views.

We implemented the propagation of changes between views and the underlying map by generating code that is driven by truth values of predicates with the symbols $mkeys$ and $mvalues$. The following code is generated for the operation m.put(k,v), where $\mathrm{ms1}, \ldots, \mathrm{msN}$ and $\mathrm{ml1}, \ldots, \mathrm{mlN}$ are possible views over m such that the corresponding predicates $mkeys(\mathrm{mks}, m, ms)$ and $mvalues(\mathrm{mvs}, m, ml)$ exist:

```
m.put(k,v)
if (mkeys(mks,m,ms1)) ms1.add(k)
 ...
if (mkeys(mks,m,msN)) msN.add(k)
if (mvalues(mvs,m,ml1)) ml1.add(v)
 ...
if (mvalues(mvs,m,mlN)) mlN.add(v).
```

Boolean variables representing the predicates over $mkeys$ and $mvalues$ are used in the actual code.

The statements m.remove(k) and m.clear() are abstracted in a similar way.

Method calls. The process of abstraction preserves calls of methods defined in the program (i.e., non-library methods). Every abstracted method has boolean parameters that correspond to predicates over parameters of the original method, and it may return multiple boolean values that correspond to predicates over the expression returned from the original method. Abstractions for method calls and returns are generated using a similar approach to the one described in [2].

For each method call in the abstract program, the actual parameter values in the abstracted caller are derived from the truth values of predicates over the actual parameters in the original caller. The returned values correspond to the truth values of the predicates over the variable used to store the result of the method call (*result variable*) in the original caller, and they are derived from the predicates over the returned expression in the original callee. Multiple truth values are returned in a bit vector that is implemented using a single integer. Output parameters (collections and reference variables that may be updated in the callee) are handled in the same way as returned expressions and result variables.

We consider inheritance, method overriding, and interfaces with their implementing classes. For every call site, the abstract program contains a non-deterministic choice between all methods that can be invoked.

Library methods are conservatively approximated in such a way that all boolean variables representing predicates over fields and the result variable are set to the value $\mathrm{unknown}$.

Predicate locality. For every statement s in the method m in the original program, only those predicates over local variables of m whose scope includes the statement s (i.e., predicates live at s) are considered by the abstraction algorithm. The predicate p is live at s only if all local variables referred to by p are live at s. If the variable v is live at s, then every program variable that may be aliased with v is also live at s.

The scope for every predicate in the set P_I is determined using static analysis of the original program code. A few restrictions apply to predicates over collections. A predicate over an iterator variable it and a map variable m is live only if the iterator it is associated with m at the given code location. The scope of a map variable (collection) must enclose the scope of iterator variables related to the map. A local variable v used as an argument for the operations put and putAhead on a local map variable m is live between the point of the first usage of v and the end of m's lifetime.

Precision. The generated abstract program is an over-approximation of the original Java program (in general). It captures all execution paths of the original Java program (in particular all errors), and possibly some additional execution paths that are not feasible for the original program. The number of additional infeasible execution paths depends on the input predicates.

5.1 Optimizations

The overall performance of the construction of an abstract program depends on the number of calls to the decision procedure, which are expensive. In this section, we describe optimizations that exploit (i) the structure and semantics of formulas that capture program state using the predicates defined in our language, and (ii) the semantics of operations upon abstract maps and iterators. The main goal of these optimizations is to reduce the number of calls to the decision procedure by (1) creating small sets P_s and $P_{s,p}$, and (2) pruning some cubes over $P_{s,p}$. A secondary goal, but also important, is the reduced complexity of the generated abstract program and the elimination of some spurious execution paths (that are not feasible in the original program). We also adapt the optimizations proposed in [2] to our predicate language.

Selecting relevant predicates. For every operation $oper$ on an abstract map or an abstract iterator, there is a set P_{oper} of predicates whose truth value must be updated to simulate the effects of $oper$'s execution.

If the operation $oper$ has a return value, the set P_{oper} contains predicates over the variable r used to store the return value and all variables that may be aliased with r. This applies, for example, to the operation v = m.get(k).

If the operation $oper$ updates the map content or iterator position, then the set P_{oper} contains predicates over the target variable (a method call receiver) and all other variables aliased with it.

For the particular operation $oper$ and each predicate $p \in P_{oper}$, the set $P_{oper,p}$ of predicates that determine the new truth value of p is a subset of P_I that depends on the weakest precondition $WP(oper, p)$. More specifically, the set $P_{oper,p}$ contains the following predicates:

- every predicate in P_I that matches some atomic predicate in the weakest precondition,
- every predicate in P_I that has a common operand with some predicate in the weakest precondition, and
- every equality predicate in P_I that refers to a variable used as an argument for the function symbol $mget$ or for the predicate $morder$ in the weakest precondition.

We use simple syntactic matching (unification) between predicates. The matching procedure considers also aliasing between variables. A logic variable in the weakest precondition can match any program variable or a constant value. A program variable can match any constant of the same type. Temporary variables used in the translation from Java collections to abstract maps match integer expressions.

Conflicting literals. The first optimization is to prune cubes over $P_{s,p}$ that include conflicting literals. It was inspired by the "enforce" construct described in [2].

Two or more literals in the given cube $\nu(P_{s,p})$ are *conflicting* if they cannot be simultaneously true. For example, the predicates $v = 3$ and $v = 5$ are conflicting because the variable v cannot be equal to two different constants at the same time. Similarly, the predicates $morder(\mathrm{mit}, m, 2, \bot)$ and $morder(\mathrm{mit}, m, 3, \bot)$ are conflicting because only one key stored in the map m can be at the end of the iteration order for m.

We define tuples of conflicting literals that express the following inherent semantic properties of abstract maps and iterators: (i) only a single value or $\bot$ can be associated with a given key k, (ii) there exist unique first and last elements in the iteration order, (iii) each element must have a unique predecessor and successor in the iteration order, (iv) and only keys stored in the map can appear in the iteration order. In addition, we define a tuple of literals as conflicting when one element of the tuple is true only for an empty map and some other element is true only for a non-empty map. This applies, for example, to the literals $mget(\mathrm{map}, m, k) \neq \bot$ and $morder(\mathrm{mit}, m, \bot, \bot)$.

Conflicting literals are reflected as follows in the abstract program. When generating the abstraction for the operation $oper$ and the updated predicate p, cubes over the set $P_{oper,p}$ that contain any conflicting literals are pruned. Code that forces JPF to backtrack is generated for the pruned cubes. We eliminate spurious execution paths (and errors) in this way too, because it cannot happen during the execution of the original program that two conflicting literals are simultaneously true in any state of the program.

This optimization significantly reduces the number of calls to the decision procedure and the size of the abstract program, because typically many cubes over $P_{s,p}$ contain pairs of conflicting literals.

Irrelevant cubes with respect to aliasing relationship. We exploit aliasing between map variables to prune additional cubes. Suppose that a statement s affects a map variable m that may be aliased with another map variable m_2, and p_i is some predicate over m_2 in $P_{s,p}$. We prune every cube over

$P_{s,p}$ that contains both of the literals $\neg(m = m_2)$ and p_i. It is sound to prune such a cube because the predicate p_i over m_2 is not relevant when m and m_2 are not aliased.

Ambiguous information. We also prune cubes that give ambiguous information about the values of the temporary variables used in the translation from Java collections to abstract maps. A cube over the set $P_{s,p}$ gives ambiguous information about the value of a temporary variable tv if for every predicate $tv = c$ in $P_{s,p}$, where c is an integer constant, the cube contains the literal $\neg(tv = c)$. For example, if the cube contains only the literals $\neg(tv = 1)$ and $\neg(tv = 2)$ for tv, then it does not specify a precise value for tv.

This optimization is sound because temporary variables must have a precise value in any reachable state of the abstract program due to the way we designed the translation from Java collections to abstract maps. Code that forces JPF to backtrack is generated for every ambiguous cube.

6. Evaluation

In this section, we describe the prototype implementation of our approach and present results of experiments with several benchmark programs.

6.1 Implementation

We implemented the predicate language and the algorithm for the construction of abstract programs, including all of the optimizations, in our J2BP tool [43]. The tool generates an abstract program in Java for an input client program, a set of properties defined as logic formulas, and a set of predicates. It uses the WALA library [46] to analyze the input Java program, the ASM library [42] for bytecode generation, and the Yices SMT solver [47] for checking satisfiability of logic formulas. Java Pathfinder (JPF) [44] is used for verification of the generated abstract programs.

Formulas representing input properties can include quantifiers over logic variables. Our tool eliminates quantifiers in property formulas based on the possible values of the logic variables that are identified by syntactic matching over the set of input predicates. Each sub-formula with an existential quantifier over the logic variable q at the root level is replaced with a disjunction where each clause is an instantiation of the original sub-formula with a particular value of q. A conjunction of such clauses is used as a replacement for a universal quantifier.

Our implementation supports classes from the Java collection library and their methods that can be modeled using abstract maps and operations upon them as described in Section 4.2. The calls of methods like List.containsAll, which are not supported yet, must be replaced with loops that implement them before running J2BP.

Artificial temporary return variables are used for processing composite expressions that involve method calls in cases where the result of a method call is not stored in an explicitly defined variable. This applies, for example, to expressions such as m.values().iterator() and m.get(k) + 2. The names of these artificial return variables start with the prefix "tmpr " and they can be referenced from the predicates.

6.2 Benchmarks

We evaluated our approach on two groups of Java programs that use collections in advanced ways: (1) small programs created by Dillig et al. [17] as a testbed for verification tools, and (2) the example programs from Section 4.1.

The programs created by Dillig et al. contain various assertions that concern values stored in the collections. We translated the programs from C++ into Java and replaced all variables and constants of the string type with integers. The properties for our example programs are defined externally, i.e., not as assertions embedded in the source code, but in separate files. All benchmarks, including our translation of programs created by Dillig et al., are available at `http://plg.uwaterloo.ca/~pparizek/oopsla12`.

Table 3 provides a short description of the properties that we tried to verify for each benchmark program and the size of the programs in terms of Java code lines. As an example, Figure 7 shows the code of the program named "Relationship between keys and values" in the table. The program adds several key-value pairs into the map m — three mappings with fixed keys (1-3) and one mapping with the key equal to the parameter s. Assertions check whether the map m contains the right key-value pairs depending on the value of the parameter s. If the value of s is 1, then the map should contain the key-value pair (1,56), and otherwise it should contain the key-value pair (1,3).

6.3 Experiments and Results

The goal of our experiments was to find out (1) whether our predicate language has sufficient expressive power to allow the verification of the properties defined for the benchmark programs, and (2) how many predicates are needed to capture all of the necessary information about the state of collections and other program variables.

For the purpose of the experiments, we defined all of the predicates by hand. We are currently working on an automated procedure for the inference of predicates about collections.

We bounded the maximal size of collections that is considered by the verification procedure, and defined predicates that reflect this bound. Restricting the size of data structures is a common approach used by many testing and verification techniques [11, 26, 39]. Note, however, that our approach is general and can be used to model collections of any finite size (provided that the corresponding predicates are available). For the purpose of our experiments, we set the bound such that all execution paths of the benchmark programs are verified.

All of the properties defined for the benchmark programs were successfully verified using predicates defined in our language. Quantitative results of the experiments are shown

Program	Property	Java LOC
Dillig et al.		
List copy	equality between lists	35
Map copy	equality between maps	39
Reverse map	correctly reversed input map	40
Set of map keys	a set contains exactly all map keys	29
Map of lists	correct size and content of nested lists	36
List of sets	valid content of nested sets after updates	35
Multimap	correct size and content of nested sets	29
Map values	map contains only non-null values	40
List elements	elements are not aliased	43
List of key-value pairs	equality between a list and a map	54
Relationship between keys and values	valid content of a map after updates	31
Examples from Section 4.1		
Thread scheduling	object variables are not null	54
Rendering image	consistency between a list and an array	61
Processing results of a cycling race	correctly sorted map	57
Simple data-flow analysis	set of facts exists for every CFG node	63

Table 3. Benchmark programs and verified properties

Program	Predicates	J2BP time	SMT calls	Bytecode	JPF time	States
Dillig et al.						
List copy	43	146 s	2086	5862	1 s	91
Map copy	35	79 s	1114	2721	1 s	315
Reverse map	54	332 s	3854	8239	1 s	595
Set of map keys	33	21 s	312	1516	1 s	63
Map of lists	77	14769 s	111331	88414	2 s	283
List of sets	54	958 s	10836	27347	1 s	25
Multimap	26	180 s	2566	4606	1 s	175
Map values	54	477 s	6224	18955	1 s	9
List elements	52	643 s	8456	12488	1 s	29
List of key-value pairs	78	302 s	3324	6328	1 s	945
Relationship between keys and values	6	9 s	198	695	1 s	29
Examples from Section 4.1						
Thread scheduling	30	52 s	782	2844	1 s	67
Rendering image	65	2612 s	20272	16827	17 s	120871
Processing results of a cycling race	104	6654 s	67390	85178	9 s	51739
Simple data-flow analysis	74	978 s	11344	11761	1 s	177

Table 4. Experimental results

in Table 4. For each benchmark program, we provide the number of predicates that were used, the running time of the J2BP tool (construction of the abstract program), the number of calls to the SMT solver during construction of the abstract program, size of the generated abstract program in terms of the number of bytecode instructions, the running time of JPF (exhaustive verification of the abstract program), and the number of states explored by JPF. The number of predicates is equal to the number of boolean variables in the generated abstract program. The number of states explored by JPF indicates the number of non-deterministic choices in the abstract program. The actual predicates and property formulas used for each benchmark program are available at `http://plg.uwaterloo.ca/~pparizek/oopsla12`.

Returning to the program in Figure 7, the assertions in the program code were expressed by the formulas $i = 3$ and $i = 56$, and the following six predicates were necessary to verify them: $s = 1$, $mget(\mathrm{map}, m, 1) = 3$, $mget(\mathrm{map}, m, 1) = 56$, $mget(\mathrm{map}, m, 1) = \bot$, $i = 3$, and $i = 56$.

```
public static void bar(int s) {
  Map<Integer,Integer> m = new HashMap();

  m.put(1,3);
  m.put(2,9);
  m.put(3,34);
  m.put(s,56);

  m.remove(3);

  int i = m.get(1);
  if (s != 1) {
    assert (i == 3) : "m.get(1) != 3";
  }
  else {
    assert (i == 56) : "m.get(1) != 56";
  }
}

public static void main(String[] args) {
  bar(8);
  bar(2);
}
```

Figure 7. Benchmark program: relationship between keys and values

The measured running times indicate that more work remains to be done to improve the performance and scalability of the construction of abstract programs. This is one of our goals for future work.

7. Related Work

We describe selected existing approaches to modeling program behavior and reasoning about it that explicitly consider data structures and collections in some way. For each approach, we discuss the main differences from our work.

7.1 Contract Specification Languages

There exist specification and modeling languages for object-oriented programs that allow to define valid functional behavior in terms of contracts for individual methods and object invariants. We collectively call them *contract specification languages*. Popular examples are JML [30], Spec# [5], and Dafny [31].

A contract for a method consists of a precondition and a postcondition. Each class can be equipped with an invariant that must hold for every object of the class before and after the call of any method on the object. Contracts are defined as logic formulas that refer to program variables, model variables and functions, and abstract data types (e.g., sets and sequences). Method postconditions can also involve special constructs for accessing (i) values that were current at the time of a method call and (ii) the return value. The contract languages support basic set operations on the respective data types, accessing sequence elements by their index, and getting the current size of a given collection.

The motivation behind contract specification languages such as JML and Spec# is to permit modular verification (i.e., one method at a time). Let the method m_1 be equipped with a contract and the method m_2 be the caller of m_1. The implementation of m_1 must first be verified against its contract. Later, in the verification of m_2, the contract for m_1 can be used as an abstraction of its behavior.

Verification tools for the contract specification languages have been developed — for example, ESC/Java for JML [15, 19], Boogie for Spec# [4], and the verifier for Dafny [31]. We focus on the ESC/Java tool as it targets Java programs. It detects simple errors like null dereferences and array index bound violations, and it also checks whether methods satisfy their contracts defined by the user and whether they preserve all invariants. The verification algorithm consists of two big steps: a verification condition is generated from the method's code, and then an automatic theorem prover is used to check the verification condition.

Modular verification of contracts is very scalable but it reports false errors. The result depends very much on the precision and the completeness of the user-defined contracts (annotations) — the user must write many annotations to capture the necessary information and eliminate spurious warnings.

Comparison. The main limitations of method contracts are that they can be associated only with method boundaries, and that they cannot refer to local variables. It is not possible to use contracts for expressing the value of a specific variable at an arbitrary program location. Moreover, an invariant defined for a class C can describe the valid states of a single object o of the class and the objects pointed to by the fields of o (i.e., objects owned by o), but it cannot specify the relation between the fields of o and the values of completely unrelated variables (e.g., fields of a different object).

In our approach, predicates and properties can capture information about the values of local variables, and they can express the current value of any variable at any program code location. Properties built using our predicate language can specify a relation between collections from different objects — this is an important prerequisite for expressing consistency properties between multiple collections. For example, our approach allows defining a consistency property between a collection local to some method m of an object o and a collection pointed to by some field of o, which should hold at a particular location in m.

Regarding verification, our approach is not modular unlike the algorithm used by ESC/Java. We aim at path-sensitive inter-procedural analysis of the whole program.

7.2 Logics for Data Structures

There exist many logic-based approaches to modeling data structures and collections. Sets and maps (hash tables) are

typically modeled using the array theory [12]. However, our approach to modeling maps differs from the encoding of hash tables described in [12, Section 7.3] in the following aspects: (1) we do not support set operations over keys, (2) we support predicates over map size and iteration order, and (3) we support views of stored keys and values. Some approaches also support lists and other data structures. For example, the theorem prover Z3 [36] supports lists and trees defined recursively in a way that resembles modeling collections at the representation level. A first-order theory of sequences based on concatenation was proposed in [20]. It supports lists defined in the recursive style. A decidable version of this theory does not support indexed access to elements and therefore does not allow updating of individual elements, but it supports membership queries and various properties over the whole sequences (e.g., sortedness and partitioning).

Kroening et al. [45] propose theories for reasoning about finite sets, maps, and lists in the context of the SMT-Lib standard [38]. Their work has similar goals to our approach, i.e. modeling externally observable state of various collections. However, we focus on designing a predicate language suitable for program verification, while the authors of [45] propose general theories. The main difference between their approach and ours is that we use the array theory instead of defining new custom theories, and therefore logic formulas built using our predicate language can be checked with existing SMT solvers that support the array theory. Regarding specific features, the approach of Kroening et al. supports basic set operations like union and intersection, but it has no support for iterators. It allows defining sets and lists by enumeration of elements, but that can be simulated via conjunctions of atomic predicates in our approach.

Powerful logics and complex theories for representing data structures and checking properties about them were recently proposed [27, 33]. The STRAND logic [33] allows reasoning about heap data structures and the values stored in them. However, such complex logics are not needed for (1) modeling collections at the interface level and (2) verification of programs that use collections against the properties that we are interested in.

7.3 Program Verification

In this section, we describe program verification techniques that model collections at the interface level in some way, and techniques that could be used for this purpose.

Blanc et al. [8] proposed a technique for checking whether a C++ program uses STL containers correctly with respect to preconditions and postconditions of individual methods. Their approach focuses on position-based containers and iterators. It does not support maps, sets, and nested containers.

Kapur et al. [24] proposed a method for verifying programs that use data structures through interface functions assuming a correct implementation of these data structures. The method is based on abstraction refinement (CEGAR), and it uses interpolants with symbolic execution to find new predicates. It has the same goals as our overall approach, but in [24], the authors focused on computing interpolants for the various theories rather than on faithful modeling of collection state and behavior. Library methods are abstracted with formulas that express transition relations — they specify, for example, how the return value of a method depends on the parameter values. Strong updates are possible through functions that return a modified data structure, but the abstraction does not make it possible to capture side effects. To capture the behavior of Java collections precisely, it is necessary to model side effects such as increasing the collection size upon addition of a new element.

An optimization of symbolic execution that models collections at the interface level was proposed in [26]. The necessary information about the collection state is captured in terms of sets of actual elements and the operation $\mathrm{contains}$. It supports only maps and sets, and not ordered lists.

The static technique proposed by Dillig et al. [17] computes information about the possible content of collections during a program run at the level of individual elements, and uses this information for checking memory safety of C++ programs. It uses a unified abstract model of position-based containers and associative containers. It supports iterators and nested collections. The content of a particular collection can be expressed by a graph, where nodes represent collections (heap structures) or atomic values and edges represent the containment relation. Constraints on indexes that are attached to edges indicate possible associations between keys (indexes) and values — for each possible (tracked) value the constraints define positions at which it may be stored. The main limitations of this technique compared to our approach are the following: (1) each analysis fact can give information only about the content of a particular collection and all collections nested in it, and (2) it cannot maintain precise information about collection size. In our approach, a single formula that uses predicates defined in our language can describe the content of multiple separate collections (that are not in the nesting relation).

Program verification techniques and tools that model collections at the representation level were also published. For example, the Jahob verification system proposed by Zee et al. [40, 41] analyzes the whole Java program at the pointer-level with the goal of proving correctness of individual methods with respect to some specification of their functional behavior. It supports a very powerful language for specifying properties and integrates various reasoning tools (automated SMT solvers and interactive theorem provers). The verification procedure of Jahob is modular such that preconditions and postconditions of already checked methods are used to verify their callers. It involves automated generation of verification conditions (proof targets), but users may be required to provide hints.

Other techniques aim at verifying programs that manipulate linked lists and dynamically allocated memory. They model lists at the pointer level and impose some restrictions on the programs that they can verify. For example, the approach proposed in [9] works only for programs that do not access data stored in the lists at all or that only compare data in some nodes (for sorting purposes). Lahiri and Qadeer proposed a logic for specifying properties over linked lists and a decision procedure for checking programs in a subset of C against such specifications [29]. This approach supports only loop-free and call-free programs that use explicit deallocation. All loops must be unrolled and method calls must be inlined (or replaced with their preconditions and postconditions). It does not support library calls. The logic allows to specify invariants over the content and structure of linked lists (e.g., whether they are sorted) and properties related to memory safety.

7.4 Predicate Abstraction for Collections and the Heap

Some work has also been done in shape analysis techniques based on predicate abstraction. These techniques work upon the heap graph, i.e. upon the pointer-level representation of data structures. The key idea of [37] is to encode the heap graph by special predicates that express pointer relations between objects (i.e., edges in the graph). The approach proposed by Dams and Namjoshi [16] uses predicates that can express the reachability of an address a_2 from the address a_1, and performs shape analysis via model checking of the predicate abstraction of a given program.

8. Conclusion

We have proposed an approach for modeling the state and behavior of Java collections observable from client programs, which is based on predicates and abstract maps with iterators. It captures all of the important features of Java collections, including the following: iteration order, size, views, aliasing between elements, and nested collections.

We use the model of Java collections based on maps and the predicate language to verify interesting properties of client programs by means of predicate abstraction. The predicate language has sufficient expressive power to capture the information necessary for successful verification of properties that depend on the collection state, as we illustrate in the results of our experiments and in the examples in Section 4.1. Note that all of the features of the predicate language are required for this purpose.

Future work. Our main priorities for future work are the following: (1) automated inference of predicates about collections that are needed to verify a given property, and (2) further improving the performance and scalability of constructing abstract programs. We aim at inference of predicates that would yield a sufficiently precise abstraction, for which the model checker reports no spurious errors and finds real errors (if any exist in the original input program).

We also plan to improve the practical usefulness of our tool chain by supporting projection of error traces back to the source code of input programs. Another possible direction is the usage of our tool in existing CEGAR-based verification frameworks [14] — this would require support for automated checking whether an error trace is spurious and identification of predicates that can be used to eliminate the given spurious error trace from the abstract program.

In the long term, we plan to use the proposed predicate language in other program verification and bug finding techniques. For example, the weakest preconditions that we defined could be used in backward symbolic execution that aims at efficient bug finding [13]. Another option is to use the predicates as elements of an abstract domain in static analysis (abstract interpretation). We could extend the technique described in [32], which aims to verify the correctness of dereferences using weakest preconditions and backward data flow analysis, so that it handles collections more precisely. We might also consider interpolation-based model checking procedures [22, 24, 35] that construct the abstraction on-the-fly during the traversal of the program state space.

Finally, we would like to extend the predicate language with support for set operations over the content of maps, assuming that a map is actually a set of key-value pairs.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada and the Ontario Ministry of Economic Development and Innovation.

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S.K. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers. In Proceedings of EuroSys 2006, ACM.

[2] T. Ball, R. Majumdar, T. Millstein, and S.K. Rajamani. Automatic Predicate Abstraction of C Programs. In Proceedings of PLDI 2001, ACM.

[3] T. Ball, A. Podelski, and S.K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. In TACAS 2001, LNCS, vol. 2031.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and R. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In FMCO 2005, LNCS, vol. 4111.

[5] M. Barnett, R. Leino, and W. Schulte. The Spec# Programming System: An Overview. In CASSIS 2004, LNCS, vol. 3362.

[6] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape Analysis for Composite Data Structures. In CAV 2007, LNCS, vol. 4590.

[7] J. Bingham and Z. Rakamaric. A Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs. In VMCAI 2006, LNCS, vol. 3855.

[8] N. Blanc, A. Groce, and D. Kroening. Verifying C++ with STL Containers via Predicate Abstraction. In ASE 2007, ACM.

[9] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists Are Counter Automata. In CAV 2006, LNCS, vol. 4144.

[10] A. Bouajjani, C. Dragoi, C. Enea, A. Rezine and M. Sighireanu. Invariant Synthesis for Programs Manipulating Lists with Unbounded Data. In CAV 2010, LNCS, vol. 6174.

[11] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates, In ISSTA 2002, ACM.

[12] A.R. Bradley, Z. Manna, and H.B. Sipma. What's Decidable About Arrays?. In VMCAI 2006, LNCS, vol. 3855.

[13] S. Chandra, S.J. Fink, and M. Sridharan. Snugglebug: A Powerful Approach to Weakest Preconditions. In PLDI 2009, ACM.

[14] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement, In CAV 2000, LNCS, vol. 1855.

[15] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML, In CASSIS 2004.

[16] D. Dams and K. Namjoshi. Shape Analysis through Predicate Abstraction and Model Checking. In Proceedings of VMCAI 2003, LNCS, vol. 2575.

[17] I. Dillig, T. Dillig, and A. Aiken. Precise Reasoning for Programs Using Containers. In Proceedings of POPL 2011, ACM.

[18] D. Distefano and M. Parkinson. jStar: Towards Practical Verification for Java. In Proceedings of OOPSLA 2008, ACM.

[19] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended Static Checking for Java, In Proceedings of PLDI 2002, ACM.

[20] C.A. Furia. What's Decidable about Sequences?. In ATVA 2010, LNCS, vol. 6252.

[21] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In Proceedings of CAV 1997, LNCS, vol. 1254.

[22] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In POPL 2010, ACM.

[23] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In Proceedings of POPL 2002, ACM.

[24] D. Kapur, R. Majumdar, and C.G. Zarba. Interpolation for Data Structures. In Proceedings of FSE 2006, ACM.

[25] S. Khurshid, C.S. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In TACAS 2003, LNCS, vol. 2619.

[26] S. Khurshid and Y.L. Suen. Generalizing Symbolic Execution to Library Classes. In Proceedings of PASTE 2005, ACM.

[27] V. Kuncak, R. Piskac, P. Suter, and T. Wies. Building a Calculus of Data Structures. In VMCAI 2010, LNCS, vol. 5944.

[28] S.K. Lahiri and S. Qadeer. Verifying Properties of Well-Founded Linked Lists. In Proceedings of POPL 2006, ACM.

[29] S.K. Lahiri and S. Qadeer. Back to the Future: Revisiting Precise Program Verification Using SMT Solvers. In Proceedings of POPL 2008, ACM.

[30] G. Leavens, A. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. ACM SIGSOFT Software Engineering Notes, 31(3), 2006.

[31] R. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In LPAR 2010, LNAI, vol. 6355.

[32] R. Madhavan and R. Komondoor. Null Dereference Verification via Over-approximated Weakest Pre-conditions Analysis. In OOPSLA 2011, ACM.

[33] P. Madhusudan, G. Parlato, and X. Qiu. Decidable Logics Combining Heap Structures and Data. In POPL 2011, ACM.

[34] J. McCarthy. A Basis for a Mathematical Theory of Computation, Technical report, MIT, Cambridge, MA, USA, 1962.

[35] K. McMillan. Lazy Abstraction with Interpolants. In CAV 2006, LNCS, vol. 4144.

[36] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In Proceedings of TACAS 2008, LNCS, vol. 4963.

[37] A. Podelski and T. Wies. Boolean Heaps. In SAS 2005, LNCS, vol. 3672.

[38] S. Ranise and C. Tinelli. The SMT-LIB standard, version 1.2, August 2006.

[39] W. Visser, C.S. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder, In ISSTA 2004, ACM.

[40] K. Zee, V. Kuncak, and M. Rinard. Full Functional Verification of Linked Data Structures. In PLDI 2008, ACM.

[41] K. Zee, V. Kuncak, and M. Rinard. An Integrated Proof Language for Imperative Programs. In PLDI 2009, ACM.

[42] ASM: Java bytecode manipulation and analysis framework. `http://asm.ow2.org/`.

[43] J2BP tool for predicate abstraction of Java programs. `http://plg.uwaterloo.ca/~pparizek/j2bp/`.

[44] Java Pathfinder system for verification of Java programs. `http://babelfish.arc.nasa.gov/trac/jpf/`.

[45] SMT-LIB Format for Finite Lists, Sets, and Maps. `http://www.cprover.org/SMT-LIB-LSM/`.

[46] T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net/`.

[47] Yices: An SMT solver. `http://yices.csl.sri.com/`.

A. Weakest Preconditions

For each statement s of the language in Figure 6, we define the weakest preconditions for predicates whose truth value may be changed by the execution of s. The weakest preconditions are shown in Tables 5-9. We consider only statements whose execution may change the truth value of some predicate — assignment statements, object construction, and supported operations on abstract maps and iterators.

The weakest precondition $WP(s, p)$ is in many cases defined in terms of the predicate p and certain syntactic substitutions. The symbol F_x denotes any valid atomic predicate over the expression x that is not covered by other lines of the table for a given statement. The symbols q, q_k and q_m denote logic variables whose names are different from the names of all program variables. The symbol d represents any expression that can have a specific position in the iteration order of some map (i.e., a key or an iterator variable).

Statement s	Predicate p	$WP(s, p)$
w = w'	F_w	$p[w'/w]$
w = e	w relop u	e relop u
	F_w	false
o.f = e	$\mathit{fread}(f, o')$ relop u	$\mathit{fread}(\mathit{fwrite}(f, o, e), o')$ relop u
	$F_{\mathit{fread}(f,o)}$	false
o = new C	$o = w$	false
	$o = \text{null}$	false
	$\mathit{fread}(f, o)$ relop u	false
	F_o	false
a[i] = e	$\mathit{aread}(\text{arr}, a', i')$ relop u	$\mathit{aread}(\mathit{awrite}(\text{arr}, a, i, e), a', i')$ relop u
	$F_{\mathit{aread}(\text{arr},a,i)}$	false
m = new map	$\mathit{mget}(\text{map}, m, k) = \bot$	true
	$\mathit{mget}(\text{map}, m, k)$ relop v	false
	$\mathit{mget}(\text{map}, m, k_1) = \mathit{mget}(\text{map}, m, k_2)$	true
	$\mathit{msize}(\text{msz}, m) = 0$	true
	$\mathit{msize}(\text{msz}, m)$ relop sz	false
	$\mathit{morder}(\text{mit}, m, d_1, d_2)$	false
	$\mathit{morder}(\text{mit}, m, \bot, \bot)$	true
	$\mathit{mkeys}(\text{mks}, m, ms)$	false
	$\mathit{mkeys}(\text{mks}, m', m)$	false
	$\mathit{mvalues}(\text{mvs}, m, ml)$	false
	$\mathit{mvalues}(\text{mvs}, m', m)$	false
	$m = m'$	false

Table 5. Weakest preconditions for assignment statements

Statement s	Predicate p	$WP(s, p)$
r = m.get(k)	$r = \mathit{mget}(\text{map}, m', k')$	$(m = m' \wedge k = k') \vee (\mathit{mget}(\text{map}, m, k) = \mathit{mget}(\text{map}, m', k'))$
	$r = \text{null}$	$\exists q_m : q_m = m \wedge \mathit{mget}(\text{map}, q_m, k) = q \wedge (q = \bot \vee q = \text{null})$
	$r = e$	$\exists q_m : q_m = m \wedge e = \mathit{mget}(\text{map}, q_m, k)$
	$\mathit{fread}(f, r) = e$	$\exists q_m : q_m = m \wedge \mathit{fread}(f, \mathit{mget}(\text{map}, q_m, k)) = e$
	F_r	$\exists q_m\ \exists q \neq \bot : q_m = m \wedge \mathit{mget}(\text{map}, q_m, k) = q \wedge F_r[q/r]$
b = m.containsKey(k)	$b = \text{true}$	$\exists q_m\ \exists q \neq \bot : q_m = m \wedge \mathit{mget}(\text{map}, q_m, k) = q$
	$b = \text{false}$	$\exists q_m : q_m = m \wedge \mathit{mget}(\text{map}, q_m, k) = \bot$
b = m.containsValue(v)	$b = \text{true}$	$\exists q_m\ \exists q : q_m = m \wedge \mathit{mget}(\text{map}, q_m, q) = v$
	$b = \text{false}$	$\neg(\exists q_m\ \exists q : q_m = m \wedge \mathit{mget}(\text{map}, q_m, q) = v)$
r = m.findKey(v)	$\mathit{mget}(\text{map}, m', r) = v'$	$m = m' \wedge v = v'$
	F_r	$\exists q_m\ \exists q : q_m = m \wedge \mathit{mget}(\text{map}, q_m, q) = v \wedge F_r[q/r]$
sz = m.size()	sz relop u	$\exists q_m : q_m = m \wedge \mathit{msize}(\text{msz}, q_m)$ relop u

Table 6. Weakest preconditions for statements that query map state

Statement s	Predicate p	$WP(s,p)$
m.put(k,v)	$mget(\text{map}, m', k') = v'$	$mget(mupdate(\text{map}, m, k, v), m', k') = v'$
	$mget(\text{map}, m', k') = mget(\text{map}, m'', k'')$	$(m = m' \wedge k = k' \wedge mget(\text{map}, m'', k'') = v) \vee (m = m'' \wedge k = k'' \wedge mget(\text{map}, m', k') = v) \vee ((m \neq m' \vee k \neq k') \wedge (m \neq m'' \vee k \neq k'') \wedge p)$
	$msize(\text{msz}, m')$ relop u	$ite(\ (m = m' \wedge mget(\text{map}, m', k) = \perp) \vee m \neq m',\ p[mresize(\text{msz}, m, msize(\text{msz}, m) + 1)/\text{msz}],\ p\ )$
	$morder(\text{mit}, m', \perp, d)$	$ite(\ m = m', ite(\ d = k, morder(\text{mit}, m', \perp, \perp),\ p\ ),\ p\ )$
	$morder(\text{mit}, m', d, \perp)$	$ite(\ m = m', d = k,\ p\ )$
	$morder(\text{mit}, m', \perp, \perp)$	$ite(\ m = m', \text{false},\ p\ )$
	$morder(\text{mit}, m', d_1, d_2)$	$ite(\ m = m', d_1 \neq k \wedge ((d_2 = k \wedge mget(\text{map}, m', d_1) \neq \perp) \vee p),\ p\ )$
m.putAhead(k,v,l)	$morder(\text{mit}, m', \perp, d)$	$ite(\ m = m', ite(\ d = k,\ p[\perp/d] \vee p[l/d],\ d \neq l \wedge p\ ),\ p\ )$
	$morder(\text{mit}, m', d, \perp)$	$ite(\ m = m' \wedge d = k, mget(\text{map}, m', l) = \perp,\ p\ )$
	$morder(\text{mit}, m', \perp, \perp)$	$ite(\ m = m', \text{false},\ p\ )$
	$morder(\text{mit}, m', d_1, d_2)$	$ite(\ m = m', ite(\ d_1 = k, d_2 = l, (d_2 = k \wedge d_1 \neq l \wedge mget(\text{map}, m', d_1) \neq \perp) \vee p\ ),\ p\ )$
m.remove(k)	$mget(\text{map}, m', k') = v'$	$mget(mupdate(\text{map}, m, k, \perp), m', k') = v'$
	$mget(\text{map}, m', k') = mget(\text{map}, m'', k'')$	$(m \neq m' \wedge m \neq m'' \wedge p) \vee (m = m' \wedge m' = m'' \wedge k' = k'') \vee (m = m' \wedge m \neq m'' \wedge ite(k \neq k', p, mget(\text{map}, m'', k'') = \perp)) \vee (m = m'' \wedge m \neq m' \wedge ite(k \neq k'', p, mget(\text{map}, m', k') = \perp))$
	$msize(\text{msz}, m')$ relop u	$ite(\ (m = m' \wedge mget(\text{map}, m', k) \neq \perp) \vee m \neq m',\ p[mresize(\text{msz}, m, msize(\text{msz}, m) - 1)/\text{msz}],\ p\ )$
	$morder(\text{mit}, m', \perp, d)$	$ite(\ m = m', d \neq k \wedge p[k/d] \wedge p[k/\perp],\ p\ )$
	$morder(\text{mit}, m', d, \perp)$	$ite(\ m = m', d \neq k \wedge p[k/\perp] \wedge p[k/d],\ p\ )$
	$morder(\text{mit}, m', \perp, \perp)$	$ite(\ m = m', ite(\ mget(\text{map}, m', k) = \perp, p, morder(\text{mit}, m', \perp, k) \wedge morder(\text{mit}, m', k, \perp)\ ),\ p\ )$
	$morder(\text{mit}, m', d_1, d_2)$	$ite(\ m = m', d_1 \neq k \wedge d_2 \neq k \wedge (p \vee (p[k/d_2] \wedge p[k/d_1])),\ p\ )$
m.clear()	$mget(\text{map}, m', k') = v'$	$ite(\ m = m', v' = \perp,\ p\ )$
	$mget(\text{map}, m', k') = mget(\text{map}, m'', k'')$	$(m \neq m' \wedge m \neq m'' \wedge p) \vee (m = m' \wedge m' = m'') \vee (m = m' \wedge m \neq m'' \wedge mget(\text{map}, m'', k'') = \perp) \vee (m = m'' \wedge m \neq m' \wedge mget(\text{map}, m', k') = \perp)$
	$msize(\text{msz}, m') = 0$	$ite(\ m = m', \text{true},\ p\ )$
	$msize(\text{msz}, m')$ relop sz	$ite(\ m = m', \text{false},\ p\ )$
	$morder(\text{mit}, m', \perp, d)$	$ite(\ m = m', \text{false},\ p\ )$
	$morder(\text{mit}, m', d, \perp)$	$ite(\ m = m', \text{false},\ p\ )$
	$morder(\text{mit}, m', \perp, \perp)$	$ite(\ m = m', \text{true},\ p\ )$
	$morder(\text{mit}, m', d_1, d_2)$	$ite(\ m = m', \text{false},\ p\ )$

Table 7. Weakest preconditions for statements that update map state

The symbol $relop$ denotes a relational operator that is defined for the type of variables used as operands (e.g., only $=$ for booleans). For the operation putAhead, we show in the table only those pairs of a predicate and a weakest precondition where the precondition is different than in the case of the put operation.

Several notes about the behavior (semantics) encoded by the weakest preconditions follow. An execution of the assignment statement w = e sets to false any predicate in which w is an argument for $mget$ and $morder$, because now the variable w may have a different value than when it was used as an argument for the put operation. The statement m = new map sets to false all predicates over the variable m except the predicates saying that the map is empty. A map created by the operation valuesView (which represents the list of all values in m) contains keys in the range from 1 to the size of m. The expression $mupdate(\text{map}, m, k, \perp)$ indicates that the key-value pair involving k is being removed from the map m. An execution of the statement o = new C sets the truth value of all aliasing predicates over the reference variable o to false and assigns the non-deterministic boolean value unknown to all predicates over fields of the object pointed to by o. This expresses the absence of any knowledge about the values of fields of the object immediately after its allocation. The predicates $mget(\text{map}, m', k') = mget(\text{map}, m'', k'')$ and the respective weakest preconditions are used to express aliasing between map elements.

Statement s	Predicate p	$WP(s, p)$
ms = m.keysView()	$mget(\text{map}, ms, k) = \text{true}$	$\exists q \neq \bot : mget(\text{map}, m, k) = q$
	$msize(\text{msz}, ms)$ relop sz	$p[m/ms]$
	$morder(\text{mit}, ms, \bot, d)$	$p[m/ms]$
	$morder(\text{mit}, ms, d, \bot)$	$p[m/ms]$
	$morder(\text{mit}, ms, \bot, \bot)$	$p[m/ms]$
	$morder(\text{mit}, ms, d_1, d_2)$	$p[m/ms]$
	$mkeys(\text{mks}, m', ms)$	$m = m'$
	$mkeys(\text{mks}, ms, ms')$	false
	$mvalues(\text{mvs}, m', ms)$	false
	$mvalues(\text{mvs}, ms, ml)$	false
	$ms = ms'$	$mkeys(\text{mks}, m, ms')$
ml = m.valuesView()	$mget(\text{map}, ml, k) = v$	$\exists q : mget(\text{map}, m, q) = v$
	$msize(\text{msz}, ml)$ relop sz	$p[m/ml]$
	$morder(\text{mit}, ml, \bot, d)$	$d = 1$
	$morder(\text{mit}, ml, d, \bot)$	$d = msize(\text{msz}, m)$
	$morder(\text{mit}, ml, \bot, \bot)$	$msize(\text{msz}, m) = 0$
	$morder(\text{mit}, ml, d_1, d_2)$	$d_1 < d_2 \wedge 1 \leq d_1 \wedge d_2 \leq msize(\text{msz}, m)$
	$mkeys(\text{mks}, m', ml)$	false
	$mkeys(\text{mks}, ml, ms')$	false
	$mvalues(\text{mvs}, m', ml)$	$m = m'$
	$mvalues(\text{mvs}, ml, ml')$	false
	$ml = ml'$	$mvalues(\text{mvs}, m, ml')$

Table 8. Weakest preconditions for statements that create views over maps

Statement s	Predicate p	$WP(s, p)$
it = m.createIterator()	$it = it'$	false
	$morder(\text{mit}, m', \bot, it)$	$m = m'$
	$morder(\text{mit}, m', it, \bot)$	$m = m' \wedge p[\bot/it]$
	$morder(\text{mit}, m', it, d)$	$m = m' \wedge p[\bot/it]$
	$morder(\text{mit}, m', d, it)$	false
b = it.hasMore()	$b = \text{true}$	$\exists q_m : \neg morder(\text{mit}, q_m, it, \bot)$
	$b = \text{false}$	$\exists q_m : morder(\text{mit}, q_m, it, \bot)$
r = it.getCurrent()	$morder(\text{mit}, m', r, it)$	true
	F_r	$\exists q_m \; \exists q_k : (morder(\text{mit}, q_m, q_k, it) \wedge F_r[q_k/r])$
it.moveNext()	$morder(\text{mit}, m', \bot, it)$	false
	$morder(\text{mit}, m', it, \bot)$	$\exists q_k : (p[q_k/\bot] \wedge p[q_k/it])$
	$morder(\text{mit}, m', it, k')$	$\exists q_k : (p[q_k/k'] \wedge p[q_k/it] \wedge \neg p[\bot/it])$
	$morder(\text{mit}, m', k', it)$	$morder(\text{mit}, m', it, k')$

Table 9. Weakest preconditions for statements over iterators