Computing Specification-Sensitive Abstractions for Program Verification

Tianhai Liu[1], Shmuel Tyszberowicz[3], Mihai Herda[1], Bernhard Beckert[1], Daniel Grahl[1], and Mana Taghdiri[2]

[1] Karlsruhe Institute of Technology, Germany
[2] Horus software GmbH, Germany
[3] The Academic College Tel Aviv Yaffo, Israel

Abstract. To enable scalability and address the needs of real-world software, deductive verification relies on modularization of the target program and decomposition of its requirement specification. In this paper, we present an approach that, given a Java program and a partial requirement specification written using the Java Modeling Language, constructs a semantic slice. In the slice, the parts of the program irrelevant w.r.t. the partial requirements are replaced by an abstraction. The core idea of our approach is to use bounded program verification techniques to guide the construction of these slices. Our approach not only lessens the burden of writing auxiliary specifications (such as loop invariants) but also reduces the number of proof steps needed for verification.

1 Introduction

Motivation. The power of deductive program verification has increased considerably over the last decades. To enable scalability and address the needs of real-world software, deductive verification relies on modularization of the target program. This requires annotating sub-procedures with formal auxiliary specifications (method contracts, loop invariants, etc.). To discover useful specifications that are fulfilled by the annotated sub-procedure and also meet the requirements of the calling procedures is, unfortunately, a difficult and error-prone effort (cf. [6, Chap. 5]). To ease the burden, verification engineers routinely break a complex requirement specification into conjunctions of *partial* specifications, i.e., they decompose not only the implementation but also the specification. Then, usually, only parts of the implementation are relevant for proving a partial property, and only partial and less complex auxiliary specifications are needed. To make use of that advantage, the verification engineer needs to identify the slice of the implementation relevant to the partial property. The main contribution of this paper is an automated method for computing such program slices defined by partial specifications.

Our approach. Given a Java program and a partial requirement specification, written using the Java Modeling Language (JML) [22], we construct a *semantic* *slice* (an abstract program). In the slice, the program parts that are irrelevant to the partial requirements are replaced by an abstraction (i.e., they are not completely removed), whereas the rest of the program (i.e., the relevant parts) remains unchanged. (In the rest of the paper we use the terms semantic slice and abstract program interchangeably.) As said above, verifying slices requires fewer auxiliary specifications (as the abstractions have less details), and their correctness—by their construction—implies the correctness of the original program w.r.t. the partial specification under consideration. As a result, our method liberates the verification engineers from finding the relevant slice manually.

Figure 1 illustrates the structure of our novel approach. The core idea is to use bounded program verification techniques to guide the construction of slices. Bounded program verification systems (such as JForge [14], Jalloy [27], and InspectJ [23]) do not require auxiliary specifications. They translate, based on user-provided bounds (that, e.g., limit the number of objects or the number of loop iterations), the analyzed program and its *negated* requirement specification into a satisfiability problem—an SMT [3] formula consisting of a set of constraints, and try to find a solution to that problem. If a solution to that satisfiability problem is found, then that is a counterexample to the correctness of the original program, and no further analysis is required. If no solution is found, the partial property holds—but only w.r.t. the bounds, thus a deductive verification—an unbounded program verification, is still needed.

Before continuing with the deductive verification we compute the slice of the program relevant to the partial requirements. The computation is based on the *unsatisfiable core* (unsat core)—a subset of constraints that is unsatisfiable, obtained during the unsatisfiability proof for the bounded problem. Then we minimize the unsat core to ensure that the proof requires all its elements. The Java program statements that are related to the constraints in the unsat core (by the construction of constraints from the Java code) are known to be relevant for the bounded proof of the requirement specification. We generate a semantic slice by over-approximating the behaviors of the other statements.

Finally, if the semantic slice can be verified using deductive program verification, which requires auxiliary specifications, the original program satisfies the specification as well (by the construction of the slice). Otherwise, we use counterexample-guided refinement to refine the abstraction and repeat the deductive verification.

The semantic slice is generated based on a particular bounded proof. Therefore it (i) may be too abstract, and thus deductive verification is not possible, and (ii) may exclude unnecessary, yet helpful, details, hence deductive verification may require more effort. But, as our evaluation shows, in practice the slice is sufficiently precise.

Our approach not only lessens the burden of writing auxiliary specifications but also eases the deductive verification: less proof steps are needed. Besides, by the *small-scope hypothesis* [20], if the program does not satisfy its specification, in many cases that will be detected during the bounded-verification phase of our approach, avoiding unnecessary attempts at deductive verification.

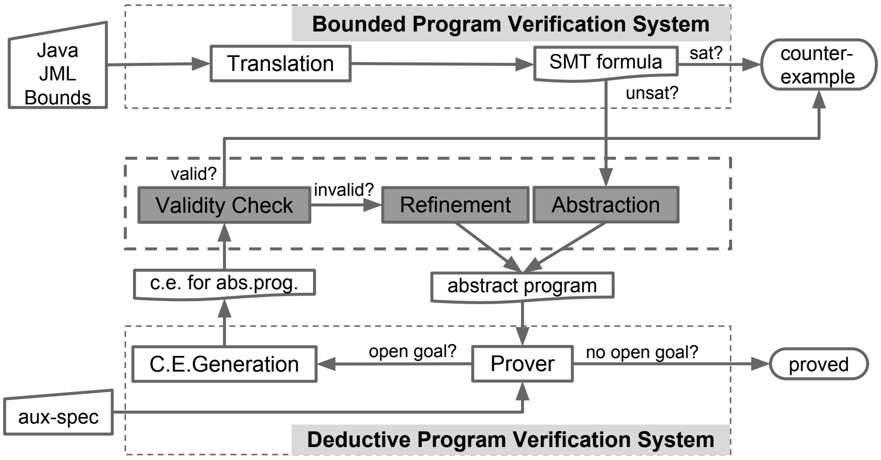


Fig. 1: Structure of our approach.

We have built a prototype tool, *AbstractJ*, that implements the abstraction as well as the refinement and validity check, and we have performed several experiments to evaluate the benefits of our approach.

2 Motivating Examples

We use two examples (Figs. 2 and 3) to demonstrate our approach. To specify the Java modules we employ JML, a behavioral interface specification language. We shortly explain the JML clauses used in the examples; for more details see [22]. The specification is written between `/*@` and `*/`. The `ensures` clause specifies properties that are guaranteed to hold at the end of the method call, and `\result` refers to the value returned by the method. (Both clauses refer to the case that the method terminated normally.) The `diverges` clause is used to specify when a method may either loop forever or not return normally to its caller. Writing `diverges true` means that non-termination is allowed for the method. The `assignable` clause provides the locations that can be assigned to during the execution of the method (frame conditions). The clause `assignable \strictly_nothing` denotes that the relevant methods neither modify heap locations nor allocate objects, whereas `assignable \nothing` allows object allocations; `assignable \everything` enables the method both to modify any heap location and to allocate objects.

The program in Fig. 2(a) computes the number of prime numbers between two given integers `x` and `y` (exclusive). The first line denotes that if the number of prime numbers is larger than `0`, then `x < y`. Carefully inspecting the code, a verification engineer will notice that the `ensures` clause becomes false only when `x >= y`. In that case, the outer loop (Fig. 2(a), statement 2) is never executed and the variable `size` remains equal to `0`. However, using traditional static slicing techniques, all statements (Fig. 2(a), statements 1-9) will be relevant w.r.t.

```
   /*@ ensures \result>0==>x<y;
     @ diverges true;
     @ assignable \everything;*/
   int numberOfPrime(int x, int y){
1    int size = 0;
2    for(int i=x; i<y; i++){
3      boolean isPrime = true;
4      for(int j=2; j<i; j++)
5        if (i%j==0){
6          isPrime=false;
7          break;
         }
8      if (isPrime)
9        size++;
     }
10   if(size > 0){
11     int[] a = new int[y-x];
     }
12   return size;
   }
```

(a) Original program

```
   /*@ ensures \result>0==>x<y;
     @ diverges true;
     @ assignable \everything;*/
   int numberOfPrime(int x, int y){
1    int size = 0;
2    for(int i=x; i<y; i++){
3      size = pure_int();







     }
4    pure_allocArrayInt();


5    return size;
   }
    //@ assignable \strictly_nothing;
    native int pure_int();
    //@ assignable \nothing;
    native int[] pure_intArray();
```

(b) Abstract program

Fig. 2: A data-structure-poor program to compute the number of prime numbers between two integers. The empty lines in the abstract program are left deliberately for an intuitive comparison.

the variables `x`, `y`, and `size` at the `return` statement. Thus loop invariants are required for the two loops. Our tool generates an abstract program as shown in Fig. 2(b) where the outer loop body (Fig. 2(a), statements 3-9) and the branch (Fig. 2(a), statements 10-11) are abstracted. Thus, it becomes easier to write loop invariants for outer loop and no loop invariant is needed for the inner loop. The abstract program is proved with KeY [1] (a deductive verification system) using 646 rules and 8 auxiliary specifications (counted as the number of JML constructs and logical connectors), while the original program is proved using 5802 rules and 26 auxiliary specifications.

The abstract statements over-approximate the behaviors of the irrelevant statements. The abstract statements invoke pure methods which have been generated automatically by our tool. The identifier of each pure method refers to the type of original statement. Each native method returns an unspecified value of the appropriate type.

Figure 3(a) shows the second example. It provides a map data type implemented using associative arrays. Keys and values are recorded in separate arrays,

```
class Key {} class Value {}
class Map {
 /*@ nullable */ Key[] keys;
 /*@ nullable */ Value[] values;
 /*@ ensures (\exists int i;0<=i&&
   @ i<values.length;values[i]==v);
   @ diverges true;
   @ assignable \everything; */
 void put(Key k, Value v){
  int pos = getIndexOf(k);
  if (pos>=0)
   values[pos] = v;
  else {
   addKey(k);
   addValue(v);
  }
 }
 int getIndexOf(Key k){
  int r = -1;
  for(int i=0;i<keys.length;i++)
   if (keys[i] == k)
    r = i;
  return r;
 }
 void addKey(Key k){
  Key[] oldKs = keys;
  keys = new Key[keys.length+1];
  keys[keys.length - 1] = k;
  for (int i=0;i<oldKs.length;i++)
   keys[i] = oldKs[i];
 }
 void addValue(Value v){
  Value[] oldVs = values;
  values=new Value[values.length+1];
  values[values.length - 1] = v;
  for (int i=0;i<oldVs.length;i++)
   values[i] = oldVs[i];
 }
}
```

(a) Original program

```
class Key {} class Value {}
class Map {
 /*@ nullable */ Key[] keys;
 /*@ nullable */ Value[] values;
 /*@ ensures (\exists int i;0<=i&&
   @ i<values.length;values[i]==v);
   @ diverges true;
   @ assignable \everything; */
 void put(Key k, Value v){
  int pos = pure_int(k);
  if(pure_boolean())
   values[pos] = v;
  else {
   impure_keys(k);
   addValue(v);
  }
 }
 void addValue(Value v){
  Value[] oldVs = values;
  values=new Value[values.length+1];
  values[values.length - 1] = v;
  for (int i=0;i<oldVs.length;i++)
   values[i] = pure_Value();
 }
 //@ assignable \strictly_nothing;
 native int pure_int();
 //@ assignable \strictly_nothing;
 native boolean pure_boolean();
 //@ assignable this.keys;
 native void impure_keys();
 //@ assignable \strictly_nothing;
 native /*@nullable*/ Value
    pure_Value();
}
```

(b) Abstract program

Fig. 3: A data-structure-rich program to put a key and a value to a map.

`keys` and `values`, respectively, and have the same index in the arrays. The method `put(k,v)` invokes the method `getIndexOf` to check whether `k` already exists in the map. If it exists, the old value is replaced by `v`; otherwise, the methods `addKey` and `addValue` reallocate the arrays `keys` and `values`, respectively, and add `k` and `v` to the new arrays. The `ensures` clause guarantees that the value `v` is in this map (the `\exists` quantifier). By default, referenced variables are not null, thus the `nullable` clause enables also the `null` value. The abstract program is shown in Fig. 3(b), where the methods `getIndexOf` and `addKey` are irrelevant to the property of interest, thus their call sites (Fig. 3(a), statements 4, 5) are abstracted, such that this fact is directly exposed to the verification engineers. The abstract method `put` contains half of the statements of the original program, relieving the user from the burden of writing some auxiliary specifications. The methods `getIndexOf` and `addKey` are abstracted, thus loop invariants are not needed for them. It is difficult to discover this fact without non-trivial efforts. Besides, the abstraction indicates that the loop (Fig. 3(a), statements 19-20) only modifies locations [0 ... ($values.length - 2$)] of the `values` array, whereas the concrete behaviors to modify the other slots can be left out completely in the loop invariants. The abstract program has been proved with KeY using 3879 rules and 4 auxiliary specifications, while the original program requires 14684 rules and 14 auxiliary specifications.

3 Techniques

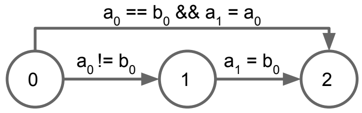
We now explain the principle techniques of our approach. We describe: program translation, program abstraction, validity check of counterexamples and refinement of abstract programs, and runtime exception handling. See Fig. 1.

We focus on analyzing object-oriented programs, and currently support a basic subset of Java that does not include floating point numbers, concurrency, and user-defined exceptions. We support a class hierarchy definition without interfaces and abstract classes. A detailed program syntax can be found in a previous work (cf. [23, Sect. 2]). We currently support a basic subset of JML that does not include model fields and exceptional behaviors.

3.1 Translation

```
void f(int a, int b){
  if (a != b) a = b; }
```

(a)



(b)

```
(assert (or E_0_1 E_0_2))
(assert (=> E_0_1 E_1_2))
```

(c)

```
(assert (=> E_1_2 (= a_1 a_0)))
```

(d)

```
(assert (=> E_0_1 (not (= a_0 b_0))))
(assert (=> E_0_2 (= a_0 b_0)))
(assert (=> E_1_2 (= a_1 b_0)))
```

(e)

Fig. 4: Encoding: (a) a sample Java code, (b) computation graph, (c) control constraints, (d) frame conditions, (e) data constraints.

We explain the translation techniques of bounded program verification (the "Translation" box in Fig. 1). Based on user-provided bounds we translate a Java program and its JML requirement specifications into an SMT formula. Some code transformations are performed on the analyzed program before the translation: Loops are unrolled the number of times defined in bounds; methods are inlined into their call sites; constructors are split into object allocation and initialization; and all variables and fields are renamed such that they are assigned at most once. The preprocessed program (called *bounded program* in the paper) is represented using a *computation graph* [27], a directed acyclic graph that has a single entry node and a single exit node. The nodes of the graph represent the control points in the bounded program, and the edges represent the state transitions. Figure 4 provides a simple example. The computation graph of the program in Fig. 4(a) is shown in Fig. 4(b), where variable names are indexes; the initial index is 0, and the index is incremented every time the variable is updated. Figure 4(c) gives the SMT constraints encoding the control flow. An SMT formula consists of logical conjunction-connected SMT constraints (enclosed in the `assert` command). Basic constraint are combined using the boolean operators `and`, `or`, `not`, and `=>` (implies).We introduce a boolean variable `E_i_j` to represent an edge from node `i` to node `j`; the data constraints in Fig. 4(e) provide the correct semantics for state transitions; the frame condition in Fig. 4(d) explicitly prevents variables to be unspecified. Variables (fields) in JML expressions are replaced by the appropriate variables (fields) in the pre-/post-state of the bounded program. More details can be found in a previous work [23].

3.2 Abstraction

When an SMT formula is unsatisfiable, an SMT solver capable of generating proofs is used to find a proof of invalidity, i.e., an *unsat core*. Minimization is performed on the core returned by an SMT solver to ensure the core is locally minimal: removing any single constraint from the core renders it satisfiable. (The algorithm is presented later in this section.) Let the set C denote the inconsistent constraints extracted from the SMT formula; i.e., C encodes the reason that no post-state violates the requirement specification. To discover which statements are responsible for a constraint in the unsat core, we maintain a *constraint map* $M := \{C \mapsto S\}$ to store the connection between the constraints C and statements S. When generating data constraints (e.g., Fig. 4(e)), the mapping from a data constraint $c \in C$ to the statement $s \in S$ (where the constraint is generated) is added to the constraint map M. Figure 5 presents the rules used for updating the constraint map M. Rules R_1 and R_2 shows data constraints are directly mapped to the simple assignment statements. We translate the assignments `e.f = e` to two constraints: $e.f' = e$ and $\forall T\, o, o \neq e \Rightarrow o.f' = o.f$ (T represents the type of e), where only the former is used to update the constraint map M (R_3). The translations of the create statement and the array update statement are handled in the same way (R_4 and R_5 respectively). The rule R_6 shows that the constraints translated from branch conditions are mapped to the branch (or loop) statement. The loop condition is negated after the last iteration, the rule R_7 maps the negation of the loop condition to the loop statement. The statements mapped by the constraints of C are the relevant statements w.r.t. the property under consideration for user-provided bounds.

$\mathbf{R}_1$: $\mathcal{T}_d[\![\texttt{T v = e;}, E]\!] \rightarrow M' = M \cup \{(E \Rightarrow (v_0 = \mathcal{T}_e[\![e]\!])) \mapsto S\}$

$\mathbf{R}_2$: $\mathcal{T}_d[\![\texttt{v = e;}, E]\!] \rightarrow M' = M \cup \{(E \Rightarrow (v' = \mathcal{T}_e[\![e]\!])) \mapsto S\}$

$\mathbf{R}_3$: $\mathcal{T}_d[\![\texttt{e.f = e;}, E]\!] \rightarrow M' = M \cup \{(E \Rightarrow (e.f' = \mathcal{T}_e[\![e]\!])) \mapsto S\}$

$\mathbf{R}_4$: $\mathcal{T}_d[\![\texttt{e = new T;}, E]\!] \rightarrow M' = M \cup \{(E \Rightarrow (e' = \mathcal{T}_e[\![newT]\!])) \mapsto S\}$

$\mathbf{R}_5$: $\mathcal{T}_d[\![\texttt{e[e] = e;}, E]\!] \rightarrow M' = M \cup \{(E \Rightarrow (e[\mathcal{T}_e[\![e]\!]] = \mathcal{T}_e[\![e]\!])) \mapsto S\}$

$\mathbf{R}_6$: $\mathcal{T}_d[\![\texttt{if (e)}, E_t, E_f]\!] \rightarrow M' = M \cup \{(E_t \Rightarrow \mathcal{T}_e[\![e]\!]) \mapsto S, (E_f \Rightarrow \mathcal{T}_e[\![!e]\!]) \mapsto S\}$

$\mathbf{R}_7$: $\mathcal{T}_d[\![\texttt{assume (e)}, E]\!] \rightarrow M' = M \cup \{(E \Rightarrow \mathcal{T}_e[\![e]\!]) \mapsto S\}$

Fig. 5: The rules for updating the constraint map M to M'. The new variables (or fields) are marked with apostrophes. $\mathcal{T}_d$ and $\mathcal{T}_e$ represent the translation of program statements and expressions respectively. S denotes a program statement and E represents the edge for the statement. E_t and E_f denote the outgoing edges of branch statements.

$\mathbf{R}_1$: $\mathcal{A}[\![\texttt{T v = e;}]\!]$:= `T v = pure_T();`

$\mathbf{R}_2$: $\mathcal{A}[\![\texttt{v = e;}]\!]$:= `v = pure_T();`

$\mathbf{R}_3$: $\mathcal{A}[\![\texttt{e.f = e;}]\!]$:= `e.f = pure_T();`

$\mathbf{R}_4$: $\mathcal{A}[\![\texttt{if (e)}]\!]$:= `if (pure_T())`

$\mathbf{R}_5$: $\mathcal{A}[\![\texttt{while (e)}]\!]$:= `while (pure_T())`

$\mathbf{R}_6$: $\mathcal{A}[\![\texttt{return e;}]\!]$:= `return pure_T();`

Fig. 6: The rules for transforming original statements to abstract statements. The transformation is denoted by $\mathcal{A}$. The concrete Java statements on the left are replaced by the abstract statements on the right. The pure_T methods return unspecified values.

These relevant statements are marked as *mustHave* statements and will not be abstracted. The other statements in the bounded program, that are named *mayHave* statements, are not necessary for bounded verification, but may be helpful in the deductive program verification. We generate an abstract program by over-approximating the behaviors of the statements in the original program when their transformed statements are mayHave statements—each transformed statement gets the location of its original statement. Thus all feasible executions of the original program are feasible in the abstract program, but not vice versa. Abstract programs are generated using the abstraction rules in Fig. 6. The original statement (on the left of Fig. 6), from which the mayHave statement has been transformed, is replaced with a statement (on the right of Fig. 6) that calls a JML-annotated *pure_T* method `/*@ assignable \strictly_nothing;*/ native /*@ nullable */ T pure_T();`. The JML `assignable` clause ensures that no memory location is changed by the pure method and that distinct unspecified values will be returned by the pure method, `T` represents an appropriate type required by the original statement, and the pure method returns an unspecified value of T which includes `null` as well. The Java keyword `native` is used to avoid implementations of the pure methods.

Using the rules in Fig. 6, the generated abstract programs provide to the verification engineers the information which statements are necessary for the properties under consideration. Thus, writing auxiliary specifications could be easier. However, it may increase the proof complexity compared to the concrete programs. Typically, a deductive program verification system (e.g., KeY [1]) symbolic executes a program and applies various calculus rules to make a proof. During symbolic execution, the symbolic states of the original program are very likely more concrete than those of the abstract program. Therefore, the symbolic execution paths which are invalid for the concrete programs are traversed when proving the abstract programs. Furthermore, symbolic execution of an abstract statement may require more rules than its original statement.

We optimize the abstract program. In order for the abstract programs to have appropriate concrete states, statements that are unnecessary for bounded program verification, yet helpful for the deductive program verification, are marked as mustHave statements. For example, assignment statements where the expression on the right-hand side is an object allocation, constant, etc., and that their defined variables are used in some mustHave statements. When possible, we abstract a set `S` of mayHave statements into one single statement, thus reducing the number of abstract statements. This is available for any two nodes `m` and `n` in the computation graph `g` (see, e.g., Fig. 4(b)), where `m` dominates `n`, `n` post-dominates `m`, and *all* the statements in `S` whose edges are in the paths from `m` to `n` are mayHave statements. The new abstract statement calls an impure method `/*@ assignable loc;*/ native T impure_T();`, where the JML specification denotes the memory locations modified by the statements. We compute the modifiable locations `loc` as a collection of the fields and variables that are updated in the mayHave statements.

Minimization of an unsat core. A locally minimal unsat core is useful for computing optimal abstractions. To the best of our knowledge, none of the SMT solvers guarantees its unsat core is locally minimal. We present an algorithm (Algorithm 1) that minimizes an unsat core by exhaustively checking whether a constraint is necessary for the unsatisfiability of the SMT formula. If the formula remains unsatisfiable when deactivating (negating) the constraints of a program statement, the constraints are not needed and their statement is a mayHave statement. The new unsat core returned by the SMT solver is the input for the next check. Otherwise, we reactivate the constraints and check the other constraints till all constraints are flipped.

Algorithm 1: Minimize an unsat core

```
Input: $\mathcal{C}$: unsatisfiable SMT constraints; $S \leftarrow \emptyset$: unnecessary constraints;
$muc \leftarrow \emptyset$: locally minimal unsat core.
for $c \in \mathcal{C}$ do
    if $c \notin S$ then
        if (check-sat $((\mathcal{C} - c) \setminus S)$) is UNSAT then
            $muc \leftarrow getUnsatCoreFromSolver()$;
            $S \leftarrow S \cup ((\mathcal{C} - c) \setminus muc)$
return $muc$
```

3.3 Validity Check and Refinement

To check the validity of the counterexamples, new bounds are always required since the abstract program fulfills the property w.r.t. the old bounds. For a counterexample `ce`, the new bound of class `C` is $CPre_{ce} + max(CPath_1, \ldots, CPath_n)$, where $CPre_{ce}$ is the number of `C` instances in `ce`, $CPath_n$ is the number of allocations of `C` instances on the `n`-th program path, and function `max` returns the maximum. To compute new loop unrolls, we transform a `while(cond){stmts;}` loop into `if(cond){stmts; if(!cond)var=var;}`, where `var` is a variable that is modifiable in `stmts`. This transformation prevents unrolling the loops that are irrelevant for the program correctness.

We compute a new SMT formula that is the conjunction of the translation of the counterexample and the translation of the original program for the new bounds. When the formula is satisfiable, then either the counterexample is valid, or the loop requires further iterations if the loop condition is still `true` after traversing the last iteration. In the latter case, we double the loop bounds and repeat the validity check. If the formula is unsatisfiable, we find the statements w.r.t. the counterexample using the techniques as shown in Sect. 3.2. In the bounded program, we highlight a mayHave statement as a mustHave statement when the statement is in the newly found statements. Finally, using the technique shown in Sect. 3.2, we generate a new abstract program for deductive verification.

3.4 Runtime Exceptions

For each property to be proved, verification systems also prove that no runtime exception is thrown. When more than one functional property has to be verified, the same proof steps for checking runtime exceptions are redone. Our approach separates the verification of functional properties from checking runtime exceptions: usually the statement `o = o.f`, e.g., is translated into $(o \neq null \Rightarrow o' = o.f) \vee (o = null \Rightarrow exc)$, where exc denotes runtime exception, whereas we translate it into $o \neq null \wedge o' = o.f$. To check that there are no runtime exceptions, we also inject guards into the code, such that if a guard passes an exception is thrown. We treat the possible exception types separately.

Figure 7 presents the code from Fig. 2(a) with one guard injected. We insert a guard (statements 11-13) which sets to true the flag *NASE* in the class *RTE* if a `NegativeArraySizeException` is about to be thrown (statement 14). Thus, when the program in Fig. 7 preserves the value of the exception flag (it is false when calling the method and when returning from it[4]), no `NegativeArraySizeException` is thrown in the original program, as the guard is checking the statement at line 14. All program parts not relevant to whether the exception is thrown are abstracted. In our approach, when there is no runtime exception and the functional properties have been fulfilled by the analyzed abstract programs, the original program is also verified.

---

[4] The `requires` clause specifies the method's precondition.

```
/*@ requires RTE.NASE = false;
  @ ensures RTE.NASE = false;
  @ diverges true;
  @ assignable \everything;*/
int numberOfPrime(int x, int y) {
   ...// statements 1-9 are omitted to save space.
   if (size>0){
      if (y-x < 0) {
         RTE.NASE = true;
         return;
      }
      this.a = new int[y-x];
   }
   return size;}
```

Fig. 7: The example from Fig. 2(a) with an injected guard.

4 Evaluation

The approach that we have presented (i) liberates verification engineers from finding the relevant program slices manually, and (ii) reduces the proof complexity especially for partial properties, for which most of the program slices are irrelevant.

We have implemented the techniques introduced in the paper in a prototype tool, *AbstractJ*. We use InspectJ [23] as the bounded verification tool and KeY [1] as the deductive verification tool. The KeY system performs *symbolic execution* [21] of sequential Java programs, using various calculus rules. Program verification with KeY is usually done in *auto-active* style: the user interacts with the system only through provided auxiliary specifications, while the proof result is obtained automatically. The number of rule applications is our primary measure of proof complexity. We have used 5 benchmark programs, all taken from the related program verification literature and from the KeY repository. Each program has 2 to 6 partial properties to be verified. We have considered also two other approaches to evaluate the effectiveness of our approach (*abstraction*) in program verification. One approach, *baseline*, proves the original programs using KeY as usual. The other approach, *highlight*, is similar to the *abstraction* approach, but it only highlights the relevant program statements and retains the irrelevant statements rather than abstracting them. We have completed 21 verification tasks using each approach, and in total we have completed 63 ($= 21 * 3$) verification tasks in our experiments. We have written the auxiliary specifications as compact as possible and measured the auxiliary specifications as the number of the operands of JML expressions, JML constructs, and logical connectors, e.g., `loop_invariant`, `assignable`, `forall`, `&&`, etc.[5] We used the SMT solver Z3 [25] to compute the unsat cores. For the experiments described in this

---

[5] Different engineers may write different auxiliary specifications for the same programs. We have asked an experienced KeY engineer to prove the original programs and a relatively inexperienced KeY user to prove the abstract programs. They carefully

Table 1: Evaluation Results

method	properties	origin	baseline		highlight		abstraction		
		stmts	specs	rules	specs	rules	stmts	specs	rules
List. merge(list)	nullPointer	27	22	3578	12	3046	4	0	196
	indexBounds	43	59	4641	46	4434	33	46	3717
	negSize	31	13	4316	14	2723	16	6	1188
	leElems	22	14	2962	14	2962	13	6	1715
	subset	22	82	6299	56	5715	15	52	4404
Map. put(key,value)	nullPointer	32	28	4485	14	3780	9	0	512
	indexBounds	48	61	6154	54	5557	48	54	5488
	negSize	32	17	4084	12	3753	16	0	654
	oldKey	26	30	4295	30	4295	11	22	1725
	sameValues	26	27	9823	34	8494	12	26	4647
	kvMatched	26	50	7327	50	7327	26	50	8814
LRS. doLRS()	nullPointer	39	11	3022	8	2818	13	0	753
	indexBounds	43	44	5006	14	4545	30	14	4502
	foundOrNot	26	32	4155	14	2908	17	10	1255
Set. intersect(set)	nullPointer	48	23	10937	18	10226	25	6	5505
	negSize	38	17	14555	14	9963	23	10	4586
	indexBounds	58	57	19715	33	12287	51	33	6714
	emptySet	33	94	64807	46	13557	16	38	3875
	subset	33	142	RO	60	136225	16	52	11211
Graph. remove(nodes)	sameNodes	54	78	RO	60	14985	13	39	3923
	sameEdges	54	119	RO	83	RO	18	67	12334

paper, we have used the default minimal bounds of InspectJ—at most 3 objects and at most 3 loop iterations. All experiments[6] have been performed on an Intel Core i5-2520M CPU with 2.50 GHz running on a 64-bit Linux.

To evaluate the effect of the *abstraction* approach on reducing the complexity of programs, we have compared the number of Java statements of original and abstract programs. The results are shown in Table 1. The column *method* shows the Java class and its method to be verified; the verified properties are listed in the column *properties*. The *nullPointer*, *indexBounds*, and *negSize* represent the runtime exceptions `NullPointerException`, `ArrayIndexOutOfBoundsException`, and `NegativeArraySizeException`, respectively. The *orgStmts* column displays the number of the original program statements.[7] The column *stmts* shows the number of the program statements that have been generated by the *abstraction* approach. On average, 49.5% (median 50%, maximum 85.2%) of statements in the original programs have been abstracted by the *abstraction* approach. There are 2 properties (`indexBounds`, and `kvMatched` for the method `put`) for which

---

inspected and ensured that the annotations are compact enough w.r.t. the requirement specifications.

[6] The complete experiments can be found at `http://asa.iti.kit.edu/458.php`.

[7] The injected guard statements are treated as original statements when handling runtime exceptions.

the approach *abstraction* seems has no effect. A careful inspection reveals that one single concrete statement is abstracted. From the results, the abstract programs contain less, yet enough details for partial properties. The more partial the verified property, the fewer details the abstract programs have. Conservatively speaking, even in the case where the abstract programs are identical to the original programs, the *abstraction* approach assists verification engineers at exploring the relevant statements—all program statements that have not been abstracted are necessary for the properties under consideration. The *highlight* approach shows the relevant program statements to verification engineers, while the *abstraction* approach provides additional benefits: (i) automatic generation of auxiliary specifications for the irrelevant program statements, and (ii) *possible* reduction of proof complexity for partial properties. Besides, the *abstraction* can increase users confidence in the correctness of their programs, before starting deductive verification.

For a fair comparison of the amount of manually written auxiliary specifications, the *highlight* approach reused the auxiliary specifications that have been written manually in the *abstraction* approach (shown in the column *specs* of the column *abstraction* in Table 1). The *abstraction* approach generates annotations for the unnecessary program slices, for which the verification engineers need to write annotations using the *highlight* approach. On average, 37.2% (median 26.7%) of annotations for the highlighted programs have been automatically generated by the *abstraction* approach.

All properties in Table 1 have been proved using the *abstraction* approach. When using the approaches *highlight* and *baseline*, several properties are unprovable. The column *rules* provides the number of rule applications. Any rule application beyond our threshold of 2000000[8] is denoted by *RO*. For 18 properties that have been proved by all approaches, the *abstraction* approach needed only 50.1% (median 55.2%) of the rules required by the *highlight* approach. It is not guaranteed that the *abstraction* approach requires less rule applications than the other two approaches for arbitrary properties. Besides of the reasons talked in Sect. 3.2, KeY creates branches for each abstract statement, to check its pre-/post-conditions.[9] When the rule cost introduced by the abstract statements is lower than the cost of symbolic execution of the irrelevant original statements, only then the *abstraction* approach requires fewer rules than other approaches, by assuming they use same auxiliary specifications. In other words, the more partial the verified property, the less proof complexity of the abstract programs. The property *kvMatched* is an example for less partial property.

Although we used small bounds for InspectJ in the experiments, there are no refinement cases in Table 1. On the other hand, when the refinement of an abstract program is needed, the abstract program will contain much less details, thus it is easy to find the relevant program statements. The verification engineers are free to provide even higher bounds for InspectJ. Given the same input formula,

---

[8] The time cost and memory consumption grow exponentially w.r.t. the rule applications. It required ∼30 min and more than 4 GB memory for 2000000 rules.

[9] The trivial pre-/post-conditions of each abstract statement requires ∼20-100 rules.

Z3 may find an unsat core that is different from the core found by other SMT solvers. AbstractJ may generate different abstract programs using other SMT solver, but the abstract programs will still contain less details than the concrete programs if the analyzed property is partial enough.

5 Related Work

Several methods have been proposed to split the program under analysis with respect to particular concerns. Traditional program slicing techniques (e.g., static/-dynamic slicing) generate a group of accessible statement (a slice) w.r.t. variables of interest at particular locations. Due to the complexity of the specification expressions and various data structures in the analyzed programs, it is very difficult to find specification-sensitive slices correctly.

Conditioned slicing techniques [4, 8, 9, 12, 13, 17] have been widely applied to simplify programs with respect to the specifications. Comuzzi et al. [12] introduced predicates as a slicing criterion; the slice contains the statements affecting the predicates. That idea has been extended by introducing preconditions [9], symbolic execution [4], and program verification [13] into conditioned slicing techniques. Typically, conditioned slicing produces a group of all accessible statements w.r.t. the specification by symbolic execution with the inputs generated by a solver. The pre/postcondition (generally formulas of first-order logic) are expressed in terms of the (input) variables at program locations of interest. However, intensive human interaction is required to guide the symbolic execution by choosing a suitable criterion. GamaSlicer [13] verifies the program w.r.t. specifications before generating semantic-based slices. Nevertheless, it may not terminate with a conclusive result, since it targets an undecidable logic. Our approach ensures that the soundness of the proof depends only on the deductive verification.

The following three approaches tried to improve the verification process using bounded analysis. Bormer et al. [6] claim that verifying programs using the bounded model checker LLBMC [24] facilitates proving with VCC [11]. Annotations written in VCCs specification language are translated into assertions that can be checked by LLBMC. El Ghazi et al. [16] try to verify Alloy problems using deductive verification, after the Alloy analyzer [20]—based on bounded analysis, fails in finding a counterexample in bounds dictated by the machine. Kroening et al. [15] combine k-induction and inductive invariant method to facilitate program verification using significantly weaker annotations. These approaches do not aim to reduce the overhead of writing specifications. However, the k-induction frequently allows using weaker loop invariants than are required by the inductive invariant approach. Our approach can reduce the burden of specifications not only for loops.

Using unsat core is not new in bounded program verification. The authors of [26] used the unsat core to refine the method summaries in program verification. In [14], a code coverage metric is constructed by the program statements that are mapped from the unsat core.

Counterexample-guided abstraction refinement (CEGAR) has been widely used in program verification. To the best of our knowledge, the abstractions have been constructed mostly at the predicate level [2,5,7,10,18] and rarely at the function level [26]. Our approach constructs the abstractions at the levels including the ones mentioned above and statement level.

6 Conclusion and Future Work

We presented a novel method to compute specification-sensitive abstractions for program verification. The abstractions are constructed with the help of bounded program verification. The counterexample-guided refinement framework has been used to refine the abstractions. We exploited the characteristics of the unsat core to discover irrelevant statements. The novelty of our approach is to abstract the program statements that are irrelevant for the properties of interest, to help verification engineers to write auxiliary specifications. We described how to: encode programs, map program statements to constraints, generate abstractions based on abstraction rules, and refine the abstractions with new bounds computation. We evaluated our experiments on 5 programs that were already used in related papers and in the KeY repository. Initial results show that our approach generates suitable abstract programs for verification, and all abstract programs have been proved for all 21 properties, while the original programs have been proved for 18 properties. Our tool took off 50% of the user's workload in writing auxiliary specifications. Only about half of the proof rules used to prove the original program are needed for proving the abstract program.

We plan to apply our approach to larger programs, and investigate incorporating loop invariant generators, e.g., Invgen [19], to improve the automation of the approach.

Acknowledgement. This work has been partially supported by GIF (grant No. 1131-9.6/2011) and by DFG under project "DeduSec" within SPP 1496 "RS^3" and by BMBF under project FIfAKS within the Software Campus program.

References

1. Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Herda, M., Klebanov, V., Mostowski, W., Scheben, C., Schmitt, P.H., Ulbrich, M.: The KeY platform for verification and analysis of Java programs. In: VSTTE. LNCS, vol. 8471, pp. 55–71. Springer (2014)
2. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In: iFM. pp. 1–20. LNCS, Springer (2004)
3. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: Version 2.5. Tech. rep., The University of Iowa (2015)
4. Barros, J.B., Carneiro da Cruz, D., Rangel Henriques, P., Sousa Pinto, J.: Assertion-based slicing and slice graphs. Formal Asp. Comput 24(2), 217–248 (2012)

5. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT 9(5-6), 505–525 (2007)
6. Bormer, T.: Advancing Deductive Program-Level Verification for Real-World Application: Lessons Learned from an Industrial Case Study. Ph.D. thesis, KIT (2014)
7. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Trans. Software Eng 30(6), 388–402 (2004)
8. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC. pp. 1284–1291. ACM (2012)
9. Chung, I.S.: Program slicing based on specification. In: SAC. pp. 605–609. ACM (2001)
10. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: TACAS. pp. 570–574. LNCS, Springer (2005)
11. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: TPHOLs. LNCS, vol. 5674, pp. 23–42. Springer (2009)
12. Comuzzi, J.J., Hart, J.M.: Program slicing using weakest preconditions. In: FME. LNCS, vol. 1051, pp. 557–575. Springer (1996)
13. Carneiro da Cruz, D., Rangel Henriques, P., Sousa Pinto, J.: GamaSlicer: an online laboratory for program verification and analysis. In: LDTA. ACM (2010)
14. Dennis, G.D.: A Relational Framework for Bounded Program Verification. Ph.D. thesis, MIT (2009)
15. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: SAS. pp. 351–368 (2011)
16. El Ghazi, A.A., Ulbrich, M., Gladisch, C., Tyszberowicz, S., Taghdiri, M.: JKelloy: A proof assistant for relational specifications of Java programs. In: NFM. pp. 173–187 (2014)
17. Fox, C., Danicic, S., Harman, M., Hierons, R.M.: ConSIT: a fully automated conditioned program slicer. Software: Practice and Experience 34(1), 15–46 (2004)
18. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. ACM SIGPLAN Notices 46(1), 331–344 (2011)
19. Gupta, A., Rybalchenko, A.: Invgen: An efficient invariant generator. In: CAV. LNCS, vol. 5643, pp. 634–640. Springer (2009)
20. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2012)
21. King, J.C.: Symbolic execution and program testing. CACM 19(7), 385–394 (1976)
22. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. ACM SIGSOFT SEN 31(3), 1–38 (2006)
23. Liu, T., Nagel, M., Taghdiri, M.: Bounded program verification using an SMT solver: A case study. In: ICST. pp. 101–110. IEEE (2012)
24. Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In: VSTTE. pp. 146–161. Springer (2012)
25. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
26. Taghdiri, M., Jackson, D.: Inferring specifications to detect errors in code. Automated Software Engineering 14(1), 87–121 (2007)
27. Vaziri, M.: Finding Bugs in Software with a Constraint Solver. Ph.D. thesis, MIT (2004)