RWTH Aachen

Department of Computer Science
Technical Report

Automated Detection of Non-Termination and `NullPointerExceptions` for Java Bytecode

Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2011-19

RWTH Aachen · Department of Computer Science · September 2011

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Automated Detection of Non-Termination and `NullPointerExceptions` for **Java Bytecode***

Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

Abstract. Recently, we developed an approach for automated termination proofs of Java Bytecode (JBC), which is based on constructing and analyzing *termination graphs*. These graphs represent all possible program executions in a finite way. In this paper, we show that this approach can also be used to detect *non-termination* or `NullPointerException`s. Our approach automatically generates *witnesses*, i.e., calling the program with these witness arguments indeed leads to non-termination resp. to a `NullPointerException`. Thus, we never obtain "false positives". We implemented our results in the termination prover AProVE and provide experimental evidence for the power of our approach.

1 Introduction

To use program verification in the software development process, one is not only interested in proving the validity of desired program properties, but also in providing a witness (i.e., a counterexample) if the property is violated.

Our approach is based on our earlier work to prove termination of JBC [4, 6, 17]. Here, a JBC program is first automatically transformed to a *termination graph* by symbolic evaluation. Afterwards, a term rewrite system is generated from the termination graph and existing techniques from term rewriting are used to prove termination of the rewrite system. As shown in the annual *International Termination Competition*,[1] our corresponding tool AProVE is currently among the most powerful ones for automated termination proofs of Java programs.

Termination graphs finitely represent all runs through a program for a certain set of input values. Similar graphs were used for many kinds of program analysis (e.g., to improve the efficiency of software verification [7], or to ensure termination of program optimization [22]). In this paper, we show that termination graphs can also be used to detect non-termination and `NullPointerException`s.

In Sect. 2, we recapitulate termination graphs. In contrast to [4, 6, 17], we also handle programs with *arrays* and we present an algorithm to *merge* abstract states in a termination graph which is needed in order to keep termination graphs finite. In Sect. 3 we show how to automatically generate *witness states* (i.e., suitable inputs to the program) which result in errors like `NullPointerException`s. Sect. 4 presents our approach to detect non-termination. Here, we use an SMT

* Supported by the DFG grant GI 274/5-3, the G.I.F. grant 966-116.6, and the DFG Research Training Group 1298 (*AlgoSyn*).

[1] See `http://www.termination-portal.org/wiki/Termination_Competition`

solver to find different forms of non-terminating loops and the technique of Sect. 3 is applied to generate appropriate witness states.

Concerning the detection of `NullPointerException`s, most existing techniques try to prove *absence* of such exceptions (e.g., [15, 23]), whereas our approach tries to prove *existence* of `NullPointerException`s and provides counterexamples which indeed lead to such exceptions. So in contrast to bug finding techniques like [2, 9], our approach does not yield "false positives".

Methods to detect *non-termination* automatically have for example been studied for term rewriting (e.g., [11, 19]) and logic programming (e.g., [18]). We are only aware of two existing tools for automated non-termination analysis of Java: The tool Julia transforms JBC programs into constraint logic programs, which are then analyzed for non-termination [20]. The tool Invel [24] investigates non-termination of Java programs based on a combination of theorem proving and invariant generation using the KeY [3] system. In contrast to Julia and to our approach, Invel only has limited support for programs operating on the heap. Moreover, neither Julia nor Invel return witnesses for non-termination. In Sect. 5 we compare the implementation of our approach in the tool AProVE with Julia and Invel and show that our approach indeed leads to the most powerful automated non-termination analyzer for Java so far.

Moreover, [14] presents a method for non-termination proofs of C programs. In contrast to our approach, [14] can deal with non-terminating recursion and integer overflows. On the other hand, [14] cannot detect *non-periodic* non-termination (where there is no fixed sequence of program positions that is repeated infinitely many times), whereas this is no problem for our approach, cf. Sect. 4.2.

There also exist tools for testing C programs in a randomized way, which can detect candidates for potential non-termination bugs (e.g., [13, 21]). However, they do not provide a proof for non-termination and may return "false positives".

2 Termination Graphs

```
public class Loop {
 public static void main(String[] a){
  int i = 0;
  int j = a.length;
  while (i < j) {
   i += a[i].length(); }}}
```

Fig. 1. Java Program

```
   main(String[] a):
00: iconst_0     #load 0 to stack
01: istore_1     #store to i
02: aload_0      #load a to stack
03: arraylength  #get array length
04: istore_2     #store to j
05: iload_1      #load i to stack
06: iload_2      #load j to stack
07: if_icmpge 22 #jump to end if i >= j
10: iload_1      #load i to stack
11: aload_0      #load a to stack
12: iload_1      #load i to stack
13: aaload       #load a[i]
14: invokevirtual length #call length()
17: iadd         #add length and i
18: istore_1     #store to i
19: goto 05
22: return
   length():
00: aload_0        #load this to stack
01: getfield count #load count field
04: ireturn        #return it
```

Fig. 2. JBC Program

We illustrate our approach by the `main` method of the Java program in Fig. 1. The `main` method is the entry point when starting a program. Its only argument is an array of `String` objects corresponding to the arguments specified on the command line. To avoid dealing with all syntactic constructs of Java, we analyze JBC instead. JBC [16] is an assembly-like object-oriented language designed as intermediate format for the execution of Java. The corresponding JBC for our example, obtained automatically by the standard `javac` compiler, is shown in Fig. 2 and will be explained in Sect. 2.2.

The method `main` increments `i` by the length of the `i`-th input string until `i` exceeds the number `j` of input arguments. It combines two typical problems:

(a) The accesses to `a.length` and `a[i].length()` are not guarded by appropriate checks to ensure memory safety. Thus, if `a` or `a[i]` are `null`, the method ends with a `NullPointerException`. While this cannot happen when the method is used as an entry point for the program, another method could for instance contain `String[] b = {null}; Loop.main(b)`.

(b) The method may not terminate, as the input arguments could contain the empty string. If `a[i] = ""`, then the counter `i` is not increased, leading to *looping* non-termination, as the same program state is visited again and again. For instance, the call `java Loop ""` does not terminate.

We show how to automatically detect such problems and to synthesize appropriate witnesses in Sect. 3 and 4. Our approach is based on *termination graphs* that over-approximate all program executions. After introducing our notion of states in Sect. 2.1, we describe the construction of termination graphs in Sect. 2.2. Sect. 2.3 shows how to create "merged" states representing two given states.

2.1 Abstract States

Our approach is related to *abstract interpretation* [8], since the states in termination graphs are *abstract*, i.e., they represent a (possibly infinite) set of concrete system configurations of the program. We define the set of all states as $\textsc{States} = (\textsc{PPos} \times \textsc{LocVar} \times \textsc{OpStack})^* \times (\{\bot\} \cup \textsc{Refs}) \times \textsc{Heap} \times \textsc{Annotations}$.

Consider the program from Fig. 1. The initial state A in Fig. 3 represents all system configurations entering the `main` method with arbitrary tree-shaped (and thus, acyclic) non-`null` arguments. A state consists of four parts: the call stack, exception information, the heap, and annotations for possible sharing effects.

00 | `a`: a_1 | ε
a_1: `String[]` i_1 i_1: $[\geq 0]$

Fig. 3. State A

The call stack consists of stack frames, where *several* frames may occur due to method calls. For readability, we exclude recursive programs, but our results easily extend to the approach of [6] for recursion. We also disregard multithreading, reflection, static fields, static initialization of classes, and floats.

Each stack frame has three components. We write the frames of the call stack below each other and separate their components by "|". The first component of a frame is the program position, indicated by the number of the next instruction (00 in Fig. 3). The second component represents the local variables by a list of references to the heap, i.e., $\textsc{LocVar} = \textsc{Refs}^*$. To avoid a special treatment of primitive values, we also represent them by references. In examples, we write the names of variables instead of their indices. Thus, "`a`: a_1" means that the value of the 0-th local variable `a` is the reference a_1 (i.e., a_1 is the address of an array object). Of course, different local variables can point to the same address. The third component is the operand stack that JBC instructions work on, where $\textsc{OpStack} = \textsc{Refs}^*$. The empty stack is "$\varepsilon$" and "$i_6, i_4$" is a stack with i_6 on top.

Information about thrown exceptions is represented in the second part of our states. If no exception is currently thrown, this part is $\perp$ (which we do not display in example states). Otherwise it is a reference to the exception object.

Below the call stack, information about the heap is given by a partial function from $\textsc{Heap} = \textsc{Refs} \rightarrow (\textsc{Integers} \cup \textsc{Unknown} \cup \textsc{Instances} \cup \textsc{Arrays} \cup \{\texttt{null}\})$ and by a set of annotations which specify possible sharing effects.

Our representation of integers abstracts from the different bounded types of integers in Java and considers arbitrary integer numbers instead (i.e., we do not handle overflows). To represent unknown integer values, we use possibly unbounded intervals, i.e., $\textsc{Integers} = \{\{x \in \mathbb{Z} \mid a \leq x \leq b\} \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \leq b\}$. We abbreviate $(-\infty, \infty)$ by $\mathbb{Z}$ and intervals like $[0, \infty)$ by $[\geq 0]$. So "i_1: $[\geq 0]$" means that any non-negative integer can be at the address i_1.

$\textsc{Classnames}$ contains the names of all classes and interfaces in the program. $\textsc{Types} = \textsc{Classnames} \cup \{t[\,] \mid t \in \textsc{Types}\}$ contains $\textsc{Classnames}$ and all resulting array types. So a type $t[\,]$ can be generated from any type t to describe arrays with entries of type t.[2] We call t' a *subtype* of t iff $t' = t$; or t' `extends`[3] or `implement`s a subtype of t; or $t' = \hat{t}'[\,]$, $t = \hat{t}[\,]$, and $\hat{t}'$ is a subtype of $\hat{t}$.

The values in $\textsc{Unknown} = \textsc{Types} \times \{?\}$ represent tree-shaped (and thus acyclic) objects and arrays where we have no information except the type. For example, for a class `List` with the field `next` of type `List`, "o_1 : `List`(?)" means that the object at address o_1 is `null` or of a subtype of `List`.

$\textsc{Instances}$ represent objects of some class. They are described by the values of their fields, i.e., $\textsc{Instances} = \textsc{Classnames} \times (\textsc{FieldIDs} \rightarrow \textsc{Refs})$. For cases where field names are overloaded, the $\textsc{FieldIDs}$ also contain the respective class name to avoid ambiguities, but we usually do not display it in our examples. So "o_1 : `List`(`next` $= o_2$)" means that at the address o_1, there is a `List` object and the value of its field `next` is o_2. For all $(cl, f) \in \textsc{Instances}$, the function f is defined for all fields of the class cl and all of its superclasses.

In contrast to our earlier papers [4, 6, 17], in this paper we also show how to handle *arrays*. An array can be represented by an element from $\textsc{Types} \times \textsc{Refs}$ denoting the array's type and length (specified by a reference to an integer value). For instance, "a_1: `String`[ ] i_1" means that at the address a_1, there is a `String` array of length i_1. Alternatively, the array representation can also contain an additional list of references for the array entries. So "a_2: `String`[ ] i_1 $\{o_1, o_2\}$" denotes that at the address a_2, we have a `String` array of length i_1, and its entries are o_1 and o_2. Thus, $\textsc{Arrays} = (\textsc{Types} \times \textsc{Refs}) \cup (\textsc{Types} \times \textsc{Refs} \times \textsc{Refs}^*)$.

In our representation, no sharing can occur unless explicitly stated. So an abstract state containing the references o_1, o_2 and not mentioning that they could be sharing, only represents concrete states where o_1 and the references reachable from o_1 are disjoint[4] from o_2 and the references reachable from o_2. Moreover, then the objects at o_1 and o_2 must be tree-shaped (and thus acyclic).

[2] We do not consider arrays of primitives in this paper, but our approach can easily be extended to handle them, as we did in our implementation.

[3] For example, any type (implicitly) `extend`s the type `java.lang.Object`.

[4] Disjointness is not required for references pointing to $\textsc{Integers}$ or to `null`.

Certain sharing effects are represented directly (e.g., "o_1:`List`(`next`=o_1)" is a cyclic singleton list). Other sharing effects are represented by three kinds of *annotations*, which are only built for references o where $h(o) \notin \textsc{Integers} \cup \{\texttt{null}\}$ for the heap h. The first kind of annotation is called *equality annotation* and has the form "$o_1 =^? o_2$". Its meaning is that the addresses o_1 and o_2 could be equal. We only use such annotations if the value of at least one of o_1 and o_2 is Unknown. *Joinability annotations* are the second kind of annotation. They express that two objects "may join" ($o_1 \searrow\!\!\swarrow o_2$). We say that a non-integer and non-`null` reference o' is a direct successor of o in a state s (denoted $o \to_s o'$) iff the object at address o has a field whose value is o' or if the array at address o has o' as one of its entries. The meaning of "$o_1 \searrow\!\!\swarrow o_2$" is that there could be an o with $o_1 \to_s^* o \leftarrow_s^+ o_2$ or $o_1 \to_s^+ o \leftarrow_s^* o_2$, i.e., o is a common successor of the two references. However, $o_1 \searrow\!\!\swarrow o_2$ does not imply $o_1 =^? o_2$. Finally, as the third type of annotations, we use *cyclicity annotations* "$o!$" to denote that the object at address o is not necessarily tree-shaped (so in particular, it could be cyclic).

2.2 Constructing Termination Graphs

Starting from the initial state A, the termination graph in Fig. 4 is constructed by symbolic evaluation. In the first step, we have to evaluate `iconst_0`, i.e., we load the integer 0 on top of the operand stack. The second instruction `istore_1` stores the value 0 on top of the operand stack in the first local variable `i`.[5]

After that, the value of the 0-th local variable `a` (the array in the input argument) is loaded on the operand stack and the instruction `arraylength` retrieves its (unknown) length i_1. That value is then stored in the second local variable `j` using the instruction `istore_2`. This results in the state B in Fig. 4. We connect A and B by a dotted arrow, indicating several evaluation steps (i.e., we omitted the states between A and B for space reasons in Fig. 4).

From B on, we load the values of `i` and `j` on the operand stack and reach C.[6] The instruction `if_icmpge` branches depending on the relation of the two elements on top of the stack. However, based on the knowledge in C, we cannot determine whether `i >= j` holds. Thus, we perform a case analysis (called *integer refinement* [4, Def. 1]), obtaining two new states D and E. We label the *refinement edges* from C to D and E (represented by dashed arrows) by the reference i_1 that was refined. In D, we assume that `i >= j` holds. Hence, i_1 (corresponding to `j`) is ≤ 0 and from $i_1 : [\geq 0]$ in state C we conclude that i_1 is 0. We thus reach instruction 22 (`return`), where the program ends (denoted by $\square$).

In E, we consider the other case and replace i_1 by i_2, which only represents positive integers. We mark what relation holds in this case by labeling the evaluation edge from E to its successor with $0 < i_2$. In general, we always use a fresh reference name like i_2 when generating new values by a case analysis, to

[5] If we have a reference whose value is from a singleton interval like $[0, 0]$ or `null`, we replace all its occurrences in states by 0 resp. by `null`. So in state B, we simply write "`i`:0". Such abbreviations will also be used in the labels of edges.

[6] The box around C and the following states is dashed to indicate that these states will be removed from the termination graph later on.

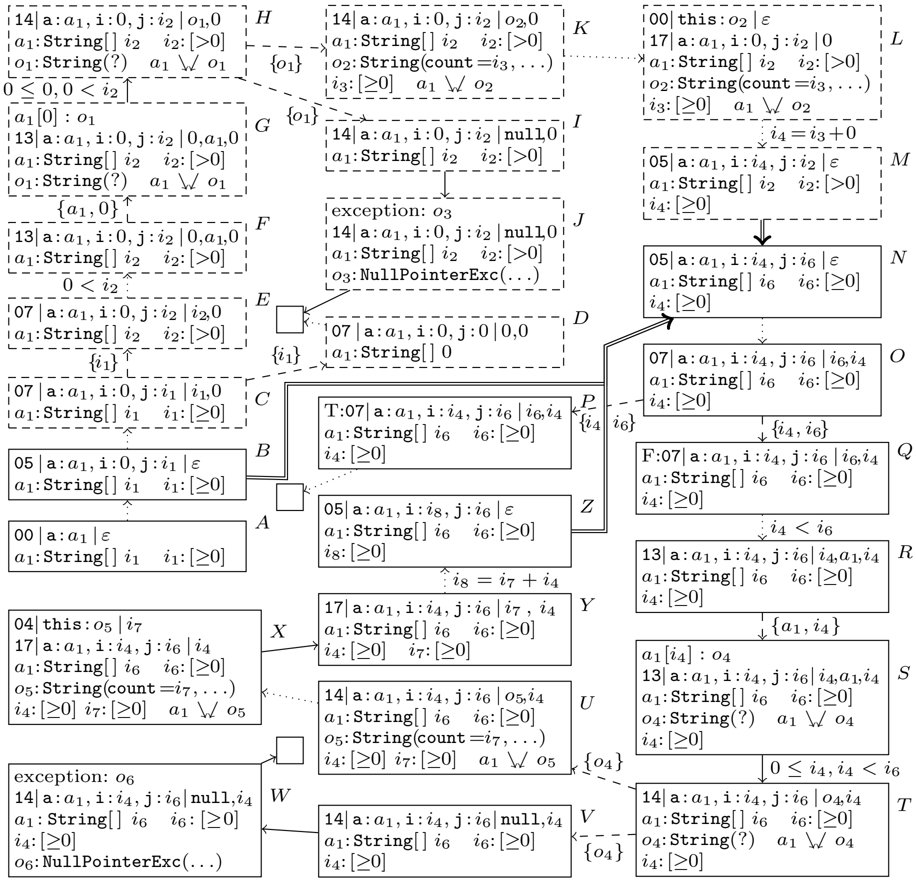

Fig. 4. Termination Graph

ensure single static assignments, which will be useful in the analysis later on. We continue with instruction `10` and load the values of `i`, `a`, and `i` on the operand stack, obtaining state F. To evaluate `aaload` (i.e., to load the 0-th element from the array a_1 on the operand stack), we add more information about a_1 at the index 0 and label the refinement edge from F to G accordingly. In G, we created some object o_1 for the 0-th entry of the array a_1 and marked that o_1 is reachable from a_1 by adding the *joinability annotation* $a_1 \mathrel{\vee\!\!\!\!\!\!\;\searrow\!\!\swarrow} o_1$.[7]

Now evaluation of `aaload` moves o_1 to the operand stack in state H. Whenever an array access succeeds, we label the corresponding edge by the condition that the used index is ≥ 0 and smaller than the length of the array.

In H, we need to `invoke` the method `length()` on the object o_1. However, we do not know whether o_1 is `null` (which would lead to a `NullPointerException`).

[7] If we had already retrieved another value o' from the array a_1, it would also have been annotated with $a_1 \mathrel{\vee\!\!\!\!\!\!\;\searrow\!\!\swarrow} o'$ and we would consequently add $o_1 \mathrel{\vee\!\!\!\!\!\!\;\searrow\!\!\swarrow} o'$ and $o_1 =^? o'$ when retrieving o_1, indicating that the two values may share or even be equal.

Hence, we perform an *instance refinement* [4, Def. 5] and label the edges from H to the new states I and K by the reference o_1 that is refined. In I, o_1 has the value `null`. In K, we replace the reference o_1 by o_2, pointing to a concrete `String` object with unknown field values. In Fig. 4, we only display the field `count`, containing the integer reference i_3. In this instance refinement, one uses the special semantics of the pre-defined `String` class to conclude that i_3 can only point to a non-negative integer, as `count` corresponds to the *length* of the string. In I, further evaluation results in a `NullPointerException`. A corresponding exception object o_3 is generated and the exception is represented in J. As no exception handler is defined, evaluation ends and the program terminates.

In K, calling `length()` succeeds. In L, a new stack frame is put on top of the call stack, where the implicit argument `this` is set to o_2. In the called method `length()`, we load o_2 on the operand stack and get the value i_3 of its field `count`. We then return from `length()`, add the returned value i_3 to 0, and store the result in the variable `i`. Afterwards, we jump back to instruction `05`. This is shown in state M and the computation $i_4 = i_3 + 0$ is noted on the evaluation edge.

But now M is at the same program position as B. Continuing our symbolic evaluation would lead to an infinite tree, as we would always have to consider the case where the loop condition `i < j` is still true. Instead, our goal is to obtain a finite termination graph. The solution is to automatically generate a new state N which represents all concrete states that are represented by B or M (i.e., N results from *merging* B and M). Then we can insert *instance edges* from B and M to N (displayed by double arrows) and continue the graph construction with N.

2.3 Instantiating and Merging States

To find differences between states and to merge states, we introduce *state positions*. Such a position describes a "path" through a state, starting with some local variable, operand stack entry, or the exception object and then continuing through fields of objects or entries of arrays. For the latter, we use the set $\text{ArrayIdxs} = \{[j] \mid j \geq 0\}$ to describe the set of all possible array indices.

Definition 1 (State Positions SPos). *Let* $s = (\langle fr_0, \ldots, fr_n \rangle, e, h, a)$ *be a state where each stack frame* fr_i *has the form* (pp_i, lv_i, os_i)*. Then* $\text{SPos}(s)$ *is the smallest set containing all the following sequences* π*:*

- $\pi = \text{LV}_{i,j}$ *where* $0 \leq i \leq n$, $lv_i = \langle o_{i,0}, \ldots, o_{i,m_i} \rangle$, $0 \leq j \leq m_i$. *Then* $s|_\pi$ *is* $o_{i,j}$.
- $\pi = \text{OS}_{i,j}$ *where* $0 \leq i \leq n$, $os_i = \langle o'_{i,0}, \ldots, o'_{i,k_i} \rangle$, $0 \leq j \leq k_i$. *Then* $s|_\pi$ *is* $o'_{i,j}$.
- $\pi = \text{EXC}$ *if* $e \neq \bot$. *Then* $s|_\pi$ *is* e.
- $\pi = \pi' \, v$ *for some* $v \in \text{FieldIDs}$ *and some* $\pi' \in \text{SPos}(s)$ *where* $h(s|_{\pi'}) = (cl, f) \in \text{Instances}$ *and where* $f(v)$ *is defined. Then* $s|_\pi$ *is* $f(v)$.
- $\pi = \pi' \, \text{LEN}$ *for some* $\pi' \in \text{SPos}(s)$ *where* $h(s|_{\pi'}) = (t, i) \in \text{Arrays}$ *or* $h(s|_{\pi'}) = (t, i, d) \in \text{Arrays}$. *Then* $s|_\pi$ *is* i.
- $\pi = \pi' \, [j]$ *for some* $[j] \in \text{ArrayIdxs}$ *and some* $\pi' \in \text{SPos}(s)$*, where* $h(s|_{\pi'}) = (t, i, \langle r_0, \ldots, r_q \rangle) \in \text{Arrays}$ *and* $0 \leq j \leq q$. *Then* $s|_\pi$ *is* r_j.

For any position π*, let* $\overline{\pi}_s$ *denote the maximal prefix of* π *such that* $\overline{\pi}_s \in \text{SPos}(s)$*. We write* $\overline{\pi}$ *if* s *is clear from the context.*

For example, in state K, the position $\pi = \textsc{os}_{0,0}$ `count` refers to the reference i_3, i.e., we have $K|_\pi = i_3$ and for the position $\tau = \textsc{lv}_{0,0}$ LEN, we have $K|_\tau = i_2$. As the field `count` was introduced between H and K by an instance refinement, we have $\pi \notin \text{SPos}(H)$ and $\overline{\pi}_H = \textsc{os}_{0,0}$, where $H|_{\overline{\pi}} = o_1$. We can now see that B and M only differ in the positions $\textsc{lv}_{0,0}$ LEN, $\textsc{lv}_{0,1}$, and $\textsc{lv}_{0,2}$.

A state s' is an *instance* of another state s (denoted $s' \sqsubseteq s$) if both are at the same program position and if whenever there is a reference $s'|_\pi$, then either the values represented by $s'|_\pi$ in the heap of s' are a subset of the values represented by $s|_\pi$ in the heap of s or else, $\pi \notin \text{SPos}(s)$. Moreover, shared parts of the heap in s' must also be shared in s. As we only consider verified JBC programs, the fact that s and s' are at the same program position implies that they have the same number of local variables and their operand stacks have the same size. For a formal definition of "instance", we refer to Def. 9 in the appendix, where we extended the "instance" definition from [4, Def. 3] to arrays.

For example, B is not an instance of M since $h_B(B|_{\textsc{lv}_{0,2}}) = [0, \infty) \not\subseteq [1, \infty) = h_M(M|_{\textsc{lv}_{0,2}})$ for the heaps h_B and h_M of B and M. Similarly, $M \not\sqsubseteq B$ because $h_M(M|_{\textsc{lv}_{0,1}}) = [0, \infty) \not\subseteq \{0\} = h_B(B|_{\textsc{lv}_{0,1}})$. However, we can automatically synthesize a "merged" (or "widened") state N with $B \sqsubseteq N$ and $M \sqsubseteq N$ by choosing the values for common positions π in B and M to be the union of the values in B and M, i.e., $h_N(N|_\pi) = h_B(B|_\pi) \cup h_M(M|_\pi)$. Thus, we have $h_N(N|_{\textsc{lv}_{0,2}}) = [0, \infty) \cup [1, \infty) = [0, \infty)$ and $h_N(N|_{\textsc{lv}_{0,1}}) = \{0\} \cup [0, \infty) = [0, \infty)$.

Algorithm `mergeStates(`s, s'`)`:
$\hat{s}$ `= new State(`s`)`
`for` $\pi \in \text{SPos}(s) \cap \text{SPos}(s')$:
  ref `= mergeRef(`$s|_\pi$, $s'|_\pi$`)`
  $\hat{h}(\mathit{ref})$ `= mergeVal(`$h(s|_\pi)$, $h'(s'|_\pi)$`)`
  $\hat{s}|_\pi = \mathit{ref}$
`for` $\pi \neq \pi' \in \text{SPos}(s)$:
  `if` $(s|_\pi = s|_{\pi'} \vee s|_\pi =^? s|_{\pi'})$
    $\wedge\ h(s|_\pi) \notin \textsc{Integers} \cup \{\texttt{null}\}$:
    `if` $\pi, \pi' \in \text{SPos}(\hat{s})$:
      `if` $\hat{s}|_\pi \neq \hat{s}|_{\pi'}$: Set $\hat{s}|_\pi =^? \hat{s}|_{\pi'}$
    `else`:
      Set $\hat{s}|_{\overline{\pi}} \mathrel{\searrow\!\swarrow} \hat{s}|_{\overline{\pi'}}$
  `if` $s|_\pi \mathrel{\searrow\!\swarrow} s|_{\pi'}$: Set $\hat{s}|_{\overline{\pi}} \mathrel{\searrow\!\swarrow} \hat{s}|_{\overline{\pi'}}$
`for` $\pi \in \text{SPos}(s)$:
  `if` $s|_\pi!$: Set $\hat{s}|_{\overline{\pi}}!$
  `if` $\exists \rho \neq \rho' : \pi\rho, \pi\rho' \in \text{SPos}(s) \wedge s|_{\pi\rho} = s|_{\pi\rho'}$
    $\wedge\ \rho, \rho'$ have no common prefix $\neq \varepsilon$
    $\wedge\ h(s|_{\pi\rho}) \notin \textsc{Integers} \cup \{\texttt{null}\}$:
    `if` $\pi\rho, \pi\rho' \in \text{SPos}(\hat{s}) \wedge \hat{s}|_{\pi\rho} \neq \hat{s}|_{\pi\rho'}$:
      Set $\hat{s}|_\pi!$
    `if` $\{\pi\rho, \pi\rho'\} \not\subseteq \text{SPos}(\hat{s})$: Set $\hat{s}|_{\overline{\pi}}!$
... same for $\text{SPos}(s')$ ...
`return` $\hat{s}$

Fig. 5. Merging Algorithm

This merging algorithm is illustrated in Fig. 5. Here, $h, h', \hat{h}$ refer to the heaps of the states $s, s', \hat{s}$, respectively. With `new State(`s`)`, we create a fresh state at the same program position as s. The auxiliary function `mergeRef` is an injective mapping from a pair of references to a fresh reference name. The function `mergeVal` maps two heap values to the most precise value from our abstract domains that represents both input values. For example, `mergeVal`$([0, 1], [10, 15])$ is $[0, 15]$, covering both input values, but also adding $[2, 9]$ to the set of represented values. For values of the same type, e.g., `String(count`$= i_1, \ldots$`)` and `String(count`$= i_2, \ldots$`)`, `mergeVal` returns a new object of same type with field values obtained by `mergeRef`, e.g., `String(count`$= i_3, \ldots$`)` where i_3 `= mergeRef(`i_1, i_2`)`. When merging values of differing types or `null`, a value from UNKNOWN with the most precise common supertype is returned.

To handle sharing effects, in a second step, we check if there are "sharing" references at some positions π and π' in s or s' that do not share anymore in the merged state $\hat{s}$. Then we add the corresponding annotations to the maximal prefixes $\hat{s}|_{\overline{\pi}}$ and $\hat{s}|_{\overline{\pi'}}$. Furthermore, we check if there are non-tree shaped objects at some position π in s or s', i.e., if one can reach the same successor using different paths starting in position π. Then we add the annotation $\hat{s}|_{\overline{\pi}}!$.

Theorem 2. *Let $s, s' \in$ STATES and $\hat{s} =$ `mergeStates(s,s')`. Then $s \sqsubseteq \hat{s} \sqsupseteq s'$.*[8]

In our example, we used the algorithm `mergeStates` to create the state N and draw instance edges from B and M to N. Since the computation in N also represents the states C to M (marked by dashed borders), we now remove them.

We continue symbolic evaluation in N, reaching state O, which is like C. In C, we refined our information to decide whether the condition `i >= j` of `if_icmpge` holds. However, now this case analysis cannot be expressed by simply refining the intervals from INTEGERS that correspond to the references i_6 and i_4 (i.e., a relation like $i_4 \geq i_6$ is not expressible in our states). Instead, we again generate successors for both possible values of the condition `i >= j`, but do not change the actual information about our values. In the resulting states P and Q, we mark the truth value of the condition `i >= j` by "T" and "F". The refinement edges from O to P and Q are marked by the references i_4 and i_6 that are refined. P leads to a program end, while we continue the symbolic evaluation in Q. As before, we label the refinement edge from Q to R by $i_4 < i_6$.

R and S are like F and G. The refinement edge from R to S is labeled by a_1 and i_4 which were refined in order to evaluate `aaload` (note that since we only reach R if $i_4 < i_6$, the array access succeeds). As in H, we then perform an instance refinement to decide whether calling `length()` on the object o_4 succeeds, leading to U and V. From V, we again reach a program end after a `NullPointerException` was thrown in W. From U, we reach X by evaluating the call to `length()`. Between X to Y, we return from `length()`. After that, we add the two non-negative integers i_7 and i_4, creating a non-negative integer i_8. The edge from Y to Z is labeled by the computation $i_8 = i_7 + i_4$.

Z is again an instance of N. We can also use the algorithm `mergeStates` to determine whether one state is an instance of another: When merging s, s' to obtain a new state $\hat{s}$, one adapts `mergeStates(s,s')` such that the algorithm terminates with failure whenever we widen a value of s or add an annotation to $\hat{s}$ that did not exist in s (e.g., when we add $\hat{s}|_\pi =^? \hat{s}|_{\pi'}$ and there is no $s|_\pi =^? s|_{\pi'}$). Then the algorithm terminates successfully iff $s' \sqsubseteq s$ holds. After drawing the instance edge from Z to N (yielding a *cycle* in our graph), all leaves of the graph are program ends and thus the graph construction is finished.

We now define termination graphs formally. We extend our earlier definition from [4] slightly by labeling edges with information about the performed refinements and about the relations of integers. Let $\mathcal{RelOp} = \{i \circ i' \mid i, i' \in \text{REFS}, \circ \in \{<, \leq, =, \neq, \geq, >\}\}$ denote the set of relations between two integer references such as $i_4 < i_6$ and $\mathcal{ArithOp} = \{i = i' \bowtie i'' \mid i, i', i'' \in \text{REFS}, \bowtie \in \{+, -, *, /, \%\}\}$

[8] For all proofs, we refer to the appendix.

denote the set of arithmetic computations such as $i_8 = i_7 + i_4$.

Termination graphs are constructed by repeatedly expanding those leaves that do not correspond to program ends. Whenever possible, we use *symbolic evaluation* $\stackrel{SyEv}{\longrightarrow}$. Here, $\stackrel{SyEv}{\longrightarrow}$ extends the usual evaluation relation for JBC such that it can also be applied to *abstract* states representing several concrete states. For a formal definition of $\stackrel{SyEv}{\longrightarrow}$, we refer to [4, Def. 6]. In the termination graph, the corresponding *evaluation edges* can be labeled by a set $C \subseteq \mathcal{A}rith\mathcal{O}p \cup \mathcal{R}el\mathcal{O}p$ which corresponds to the arithmetic operations and (implicitly) checked relations in the evaluation. For example, when accessing the index `i` of an array `a` succeeds, we have implicitly ensured $0 \leq$ `i` and `i` $<$ `a.length` and this is noted in C.

If symbolic evaluation is not possible, we refine the information for some references R by case analysis and label the resulting *refinement edges* with R.

To obtain a *finite* graph, we create a more general state by *merging* whenever a program position is visited a second time in our symbolic evaluation and add appropriate *instance edges* to the graph. However, we require all cycles of the termination graph to contain at least one evaluation edge. By using an appropriate strategy for merging resp. widening states, we can automatically generate a finite termination graph for any program.

Definition 3 (Termination Graph). *A graph* (V, E) *with* $V \subseteq \textsc{States}$*,* $E \subseteq V \times \big((\{\textsc{Eval}\} \times 2^{\mathcal{A}rith\mathcal{O}p \cup \mathcal{R}el\mathcal{O}p}) \cup (\{\textsc{Refine}\} \times 2^{\textsc{Refs}}) \cup \{\textsc{Ins}\}\big) \times V$ *is a* termination graph *if every cycle contains at least one edge labeled with some* $\textsc{Eval}_C$ *and one of the following holds for each* $s \in V$*:*

- *s has just one outgoing edge $(s, \textsc{Eval}_C, s')$, $s \stackrel{SyEv}{\longrightarrow} s'$, and C is the set of integer relations that are checked (resp. generated) in this step*
- *the outgoing edges of s are $(s, \textsc{Refine}_R, s_1), \ldots, (s, \textsc{Refine}_R, s_n)$ and $\{s_1, \ldots, s_n\}$ is a refinement of s on the references $R \subseteq \textsc{Refs}$*
- *s has just one outgoing edge $(s, \textsc{Ins}, s')$ and $s \sqsubseteq s'$*
- *s has no outgoing edge and $s = (\varepsilon, e, h, a)$ (i.e., s is a program end)*

We refer to [6, 17] for methods to use termination graphs for termination proofs and to [4] for soundness proofs which show that if c is a concrete state with $c \sqsubseteq s$ for some state s in the termination graph, then the JBC evaluation of c is represented in the termination graph. In Sect. 3 and 4 we show how to use termination graphs to detect `NullPointerException`s and non-termination.

3 Generating Witnesses for `NullPointerExceptions`

In our example, an uncaught `NullPointerException` is thrown in the "error state" W, leading to a program end. Such violations of memory safety can be immediately detected from the termination graph. In particular, if the graph does not contain any such exceptions, then memory safety is proved.[9]

To report a possible violation of memory safety to the user, we now show

[9] In languages like C, *memory safety* (or *pointer safety*) means absence of (i) accesses to `null`, (ii) dangling pointers, and (iii) memory leaks [25]. In Java, (ii) and (iii) are ensured by the JVM and only `NullPointerException`s can destroy memory safety.

how to automatically generate a *witness* (i.e., an assignment to the arguments of the program) that leads to the exception. Our termination graph allows us to generate such witnesses automatically. This technique for witness generation will also be used to construct witnesses for non-termination in Sect. 4.

So our goal is to find a *witness state* A' for the initial state A of the method `main` *w.r.t. the "error state"* W. This state A' describes a subset of arguments, all of which lead to an instance of W, i.e., to a `NullPointerException`.

Definition 4 (Witness State). *Let* $s, s', w \in \textsc{States}$. *The state* s' *is a* witness state for s w.r.t. w *iff* $s' \sqsubseteq s$ *and* $s' \xrightarrow{SyEv}{}^{*} w'$ *for some state* $w' \sqsubseteq w$.

To obtain a witness state A' for A automatically, we start with the error state W and traverse the edges of the termination graph backwards until we reach A. In general, let $s_0, s_1, \ldots, s_n = w$ be a path in the termination graph from the initial state s_0 to the error state s_n. Assuming that we already have a witness state s'_i for s_i w.r.t. w, we show how to generate a witness state s'_{i-1} for s_{i-1} w.r.t. w. To this end, we revert the changes done to the state s_{i-1} when creating the state s_i during the construction of the termination graph (i.e., we apply the rules for termination graph construction "backwards"). Of course, this generation of witness states can fail (in particular, this happens for error states that are not reachable from any concrete instantiation of the initial state s_0). So in this way, our technique for witness generation is also used as a check whether certain errors can really result from initial method calls.

In our example, the error state is W. Trivially, W itself is a witness state for W w.r.t. W. The only edge leading to W is from V. Thus, we now generate a witness state V' for V w.r.t. W. The edge from V to W represents the evaluation of the instruction `invokevirtual` that triggered the exception. Reversing this instruction is straightforward, as we only have to remove the exception object from W again. Thus, V is a witness state for V w.r.t. W.

The only edge leading to V is a refinement edge from T. As a refinement corresponds to a case analysis, the information in the target state is more precise. Hence, we can reuse the witness state for V, since V is an instance of T. So V is also a witness state for T w.r.t. W.

13| `a`: a_2, `i`: 0, `j`: 1| 0, a_2, 0
a_2: `String[]` 1 {`null`}

Fig. 6. State R'

To reverse the edge between T and S, we have to undo the instruction `aaload`. This is easy since S contains the information that the entry at index i_4 in the array a_1 is o_4. Thus the witness state S' for S w.r.t. W is like S, but here o_4's value is not an unknown object, but `null`. Reversing the refinement between S and R is more complex. Note that not every state represented by R leads to a `NullPointerException`. In S we had noted the relation between the newly created reference o_4 and the original array a_1. In other words, in S we know that $a_1[i_4]$ is o_4, where o_4 has the value `null` in the witness state S' for S. But in R, o_4 is missing. To solve this problem, in the witness state R' for R, we instantiate the abstract array a_1 by a concrete one that contains the entry `null` at the index i_4. We use a simple heuristic[10] to choose a suitable

[10] Such heuristics cannot affect soundness, but just the power of our approach (choosing unsuitable values may prevent us from finding a witness for the initial state).

length i_6 for this concrete array, which tries to find "minimal" values. Here, our heuristic chooses a_1 to be an array of length one (i.e., i_6 is chosen to be 1), which only contains the entry `null` (at the index 0, i.e., i_4 is chosen to be 0). The resulting witness state R' for R w.r.t. W is displayed in Fig. 6.

Reversing the evaluation steps between R and Q yields a witness state Q' for Q w.r.t. W. From O to Q, we have a refinement edge and thus, Q' is also a witness for O.

`0` | `a` : a_2 | ε
a_2: `String[]` 1 {`null`}

Fig. 7. State A'

The steps from N to O can also be reversed easily. In N, we use a heuristic to decide whether to follow the incoming edge from Z or from B. Our heuristic chooses B as it is more concrete than Z. From there, we continue our reversed evaluation until we reach a witness state A' for the initial state A of the method w.r.t. W, cf. Fig. 7. So any instance of A' evaluates to an instance of W, i.e., it leads to a `NullPointerException`. If the `main` method is called directly (as the entry point of the program), then the JVM ensures that the input array does not contain `null` references. But if the `main` method is called from another method, then this violation of memory safety can indeed occur, cf. problem (a) in Sect. 2.

The following theorem summarizes our procedure to generate witness states. If there is an edge from a state s_1 to a state s_2 in the termination graph and we already have a witness state s_2' for s_2 w.r.t. w, then Thm. 5 shows how to obtain a witness state s_1' for s_1 w.r.t. w. Hence, by repeated application of this construction, we finally obtain a witness state for the initial state of the method w.r.t. w. If there is an *evaluation edge* from s_1 to s_2, then we first apply the reversed rules for symbolic evaluation on s_2'. Afterwards, we instantiate the freshly appearing references (for example, those overwritten by the forward symbolic evaluation) such that s_1' is indeed an instance of s_1. If there is a *refinement edge* from s_1 to s_2, then the witness state s_1' is like s_2', but when reading from abstract arrays (such as between R and S), we instantiate the array to a concrete one in s_1'. If there is an *instance edge* from s_1 to s_2, then we *intersect* the states s_1 and s_2' to obtain a representation of those states that are instances of both s_1 and s_2'.

Theorem 5 (Generating Witnesses). *Let (s_1, l, s_2) be an edge in the termination graph and let s_2' be a witness state for s_2 w.r.t. w. Let $s_1' \in$ STATES with:*

- *if $l = \text{EVAL}_C$, then s_1' is obtained from s_2' by applying the symbolic evaluation used between s_1 and s_2 backwards. In s_1', we instantiate freshly appearing variables such that $s_1' \sqsubseteq s_1$ and $s_1' \xrightarrow{SyEv} s_2'$ holds.*
- *if $l = \text{REFINE}_R$, then $s_1' \sqsubseteq s_2'$.*
- *if $l = \text{INS}$, then $s_1' = s_1 \cap s_2'$ (for the definition of $\cap$, see [6, Def. 2]).*

Then s_1' is a witness state for s_1 w.r.t. w.

4 Proving Non-Termination

Now we show how to prove non-termination automatically. Sect. 4.1 introduces a method to detect *looping* non-termination, i.e., infinite evaluations where the *interesting references* (that determine the termination behavior) are unchanged. Sect. 4.2 presents a method which can also detect *non-looping* non-termination.

4.1 Looping Non-Termination

For each state, we define its *interesting* references that determine the control flow and hence, the termination behavior. Which references are interesting can be deduced from the termination graph, because whenever the (changing) value of a variable may influence the control flow, we perform a refinement. Hence, the references in the labels of refinement edges are "interesting" in the corresponding states. For example, the references i_4 and i_6 are interesting in the state O.

We propagate the information on interesting references backwards. For evaluation edges, those references that are interesting in the target state are also interesting in the source state. Thus, i_4 and i_6 are also interesting in N.

When drawing refinement or instance edges, references may be renamed. But if a reference at position π is interesting in the target state of such an edge, the reference at π is also interesting in the source state. So $i_8 = Z|_{\text{LV}_{0,1}}$ and $i_6 = Z|_{\text{LV}_{0,2}}$ are interesting in Z, as $i_4 = N|_{\text{LV}_{0,1}}$ and $i_6 = N|_{\text{LV}_{0,2}}$ are interesting in N.

Furthermore, if an interesting reference i of the target state was the result of some computation (i.e., the evaluation edge is labeled with $i = i' \bowtie i''$), we mark i' and i'' as interesting in the source state. The edge from Y to Z has the label $i_8 = i_7 + i_4$. As i_8 is interesting in Z, i_7 and i_4 are interesting in Y.

Definition 6 (Interesting References). *Let $G = (V, E)$ be a termination graph, and let $s, s' \in V$ be some states. Then $I(s) \subseteq \{s|_\pi \mid \pi \in \text{SPos}(s)\}$ is the set of* interesting references *of s, defined as the minimal set of references with*

- *if $(s, \text{Refine}_R, s') \in E$, then $R \subseteq I(s)$.*
- *if $(s, l, s') \in E$ with $l \in \{\text{Refine}_R, \text{Ins}\}$, then we have $\{s|_\pi \mid \pi \in \text{SPos}(s) \cap \text{SPos}(s'), s'|_\pi \in I(s')\} \subseteq I(s)$.*
- *if $(s, \text{Eval}_C, s') \in E$, then $I(s') \cap \{s|_\pi \mid \pi \in \text{SPos}(s)\} \subseteq I(s)$.*
- *if $(s, \text{Eval}_C, s') \in E$, $i = i' \bowtie i'' \in C$ and $i \in I(s')$, then $\{i', i''\} \subseteq I(s)$.*

Note that if there is an evaluation where the same program position is visited repeatedly, but the values of the interesting references do not change, then this evaluation will continue infinitely. We refer to this as *looping* non-termination.

To detect such non-terminating loops, we look at cycles $s = s_0, s_1, \ldots, s_{n-1}$, $s_n = s$ in the termination graph. Our goal is to find a state $v \sqsubseteq s$ such that when executing the loop, the values of the interesting references in v do not change. More precisely, when executing the loop in v, one should reach a state v' with $v' \sqsubseteq_\Pi v$. Here, Π are the positions of interesting references in s and $\sqsubseteq_\Pi$ is the "instance" relation restricted to positions with prefixes from Π, whereas the values at other positions are ignored. The following theorem proves that if one finds such a state v, then indeed the loop will be executed infinitely many times when starting the evaluation in a concrete instance of v.

Theorem 7 (Looping Non-Termination). *Let s occur in a cycle of the termination graph. Let $\Pi = \{\pi \in \text{SPos}(s) \mid s|_\pi \in I(s)\}$ be the positions of interesting references in s. If there is a $v \sqsubseteq s$ where $v \overset{SyEv}{\longrightarrow}{}^{+} v'$ for some $v' \sqsubseteq_\Pi v$, then any concrete state that is an instance of v starts an infinite* JBC *evaluation.*

We now automate Thm. 7 by a technique consisting of four steps (the first three steps find suitable states v automatically and the fourth step checks whether v can be reached from the initial state of the method). Let $s = s_0, s_1, \ldots, s_{n-1}, s_n = s$ be a cycle in the termination graph such that there is an instance edge from s_{n-1} to s_n. In Fig. 4, $N, \ldots Z, N$ is such a cycle (i.e., here s is N).

1. Find suitable values for interesting integer references. In the first step, we find out how to instantiate the interesting references of *integer* type in v. To this end, we convert the cycle $s = s_0, \ldots, s_n = s$ edge by edge to a formula φ over the integers. Then every model of φ indicates values for the interesting integer references that are not modified when executing the loop.

Essentially, φ is a conjunction of all constraints that the edges are labeled with. More precisely, to compute φ, we process each edge (s_i, l, s_{i+1}). If l is $\textsc{Refine}_R$, then we connect the variable names in s_i and s_{i+1} by adding the equations $s_i|_\pi = s_{i+1}|_\pi$ to φ for all those positions π where $s_i|_\pi$ is in R and points to an integer. Thus, for the edge from O to Q, we add the trivial equations $i_4 = i_4 \wedge i_6 = i_6$, as the references were not renamed in this refinement step.

If $l = \textsc{Eval}_C$, we add the constraints and computations from C to the formula φ.[11] Thus, for the edge from Q to R we add the constraint $i_4 < i_6$, for the edge from S to T we add $0 \leq i_4 \wedge i_4 < i_6$, and the edge from Y to Z yields $i_8 = i_7 + i_4$. If l is $\textsc{Ins}$, we again connect the reference names in s_i and s_{i+1} by adding the equations $s_i|_\pi = s_{i+1}|_\pi$ for all $\pi \in \mathrm{S}\textsc{Pos}(s_{i+1})$ that point to integers. Thus, for the edge from Z to N, we get $i_6 = i_6 \wedge i_8 = i_4$. So for the cycle $N, \ldots, Z, N$, φ is $i_4 < i_6 \wedge 0 \leq i_4 \wedge i_8 = i_7 + i_4 \wedge i_8 = i_4$ (where tautologies have been removed).

To find values for the integer references that are not modified in the loop, we now try to synthesize a model of φ. In our example, a standard SMT solver easily proves satisfiability and returns a model like $i_4 = 0$, $i_6 = 1$, $i_7 = 0$, $i_8 = 0$.

2. Guess suitable values for interesting non-integer references. We want to find a state $v \sqsubseteq s$ such that executing the loop does not change the values of interesting references in v. We have determined the values of the interesting *integer* references in v (i.e., i_4 is 0 and i_6 is 1 in our example). It remains to determine suitable values for the other interesting references (i.e., for a_1 in our example)

```
05| a : a3, i : 0, j : 1| ε
a3: String[] 1
```

Fig. 8. State Z'

To this end, we use the following heuristic. We instantiate the integer references in s_{n-1} according to the model found for φ, yielding a state $s'_{n-1} \sqsubseteq s_{n-1}$. So in our example (where $s_n = s$ is N and s_{n-1} is Z), we instantiate i_6 and i_8 in Z by 1 resp. 0, resulting in the state Z' in Fig. 8 (i.e., here s'_{n-1} is Z').

```
05| a : a3, i : 0, j : 1| ε
a3: String[] 1 {o6}
o6: String(count = 0, ...)
```

Fig. 9. State N'

Afterwards, we traverse the path from s_{n-1} backwards to s_0 and use the technique of witness generation from Sect. 3 to generate a witness v for s_0 w.r.t. s'_{n-1} (i.e., $v \sqsubseteq s_0$ such that $v \xrightarrow{SyEv}{}^{+} v'$ for some $v' \sqsubseteq s'_{n-1}$). In our example,[12] the

[11] Remember that we use a single static assignment technique. Thus, we do not have to perform renamings to avoid name clashes.

[12] During the witness generation, one again uses the model of φ for intermediate integer references. So when reversing the `iadd` evaluation between Y and Z, we choose 0 as value for the newly appearing reference i_7.

witness generation results in the state N' in Fig. 9. Note that the witness generation technique automatically "guessed" a suitable instantiation for the array (i.e., it was instantiated by a 1-element array containing just the empty string). Indeed, $N' \sqsubseteq N$ and $N' \stackrel{SyEv}{\longrightarrow}^{+} v'$ for an instance v' of Z' (i.e., in our example $s_0 = s$ is N and v is N'). Here, v' is like Z', but instead of "a_3:`String[]` 1", we have "a_3:`String[]` 1 $\{o_6\}$" and "o_6:`String(count = 0, ...)`". Thus, $v' = N'$.

3. Check whether the guessed values for non-integer references do not change in the loop. While our construction ensures that the interesting *integer* references remain unchanged when executing the loop, this is not ensured for the interesting non-integer references. Hence, in the third step, we now have to check whether $v' \sqsubseteq_{\Pi} v$ holds, where Π are the positions of interesting references in s.

To this end, we adapt our algorithm `mergeStates(`v`,`v'`)` such that it terminates with failure whenever we widen a value of v or add an annotation that did not exist in v at a position with a prefix from Π. Then the algorithm terminates successfully iff $v' \sqsubseteq_{\Pi} v$. In our example where $v = v' = N$, we clearly have $v' \sqsubseteq_{\Pi} v$. Hence by Thm. 7, any instance of v (i.e., of N') starts an infinite execution.

4. Check whether the non-terminating loop can be reached from the initial state. In the fourth step, we finally check whether N' can be reached from the initial state of the method. Hence, we again use the witness generation technique from Sect. 3 to create a witness state for A w.r.t. N'. This witness state has the stack frame "00 | a : a_3 | ε" where a_3 is a 1-element array containing just the empty string. In other words, we automatically synthesized the counterexample to termination indicated in problem (b) of Sect. 2.

4.2 Non-Looping Non-Termination

```
static void nonLoop(
  int x, int y) {
   if (y >= 0) {
     while(x >= y) {
       int z = x - y;
       if (z > 0) {
         x--;
       } else {
         x = 2*x + 1;
         y++; }}}}
```

Fig. 10. `nonLoop(x,y)`

A loop can also be non-terminating if the values of interesting references are modified in its body. We now present a method to find such *non-looping* forms of non-termination. In contrast to the technique of Sect. 4.1, this method is restricted to loops that have no sub-loops and whose termination behavior only depends on integer arithmetic (i.e., the interesting references in all states of the loop may only refer to integers). Then we can construct a formula that represents the loop condition and the computation on each path through the loop. If we can prove that no variable assignment that satisfies the loop condition violates it in the next loop iteration, then we can conclude non-termination under the condition that the loop condition is satisfiable.

The method `nonLoop` in Fig. 11 does not terminate if $\mathtt{x} \geq \mathtt{y} \geq 0$. For example, if $\mathtt{x} = 2$, $\mathtt{y} = 1$ at the beginning of the loop, then after one iteration we have $\mathtt{x} = 1$, $\mathtt{y} = 1$. In the next iterations, we obtain $\mathtt{x} = 3$, $\mathtt{y} = 2$; $\mathtt{x} = 2$, $\mathtt{y} = 2$; and $\mathtt{x} = 5$, $\mathtt{y} = 3$, etc. So this non-termination is non-looping and even *non-periodic* (since there is no fixed sequence of program positions that is repeated infinitely many times). Thus, non-termination cannot be proved by techniques like [14].

Consider the termination graph, which is shown in a simplified version in

Fig. 11. A node in a cycle with a predecessor outside of the cycle is called a *loop head node*. In Fig. 11, A is such a node. We consider all paths $p_1, \ldots, p_n$ from the loop head node back to itself (without traversing the loop head node in between), i.e., $p_1 = A, \ldots, B, A$ and $p_2 = A, \ldots, C, A$. Here, p_1 corresponds to the case where $\mathtt{x} \geq \mathtt{y}$ and $\mathtt{z} = \mathtt{x} - \mathtt{y} > 0$, whereas p_2 handles the case where $\mathtt{x} \geq \mathtt{y}$ and $\mathtt{z} = \mathtt{x} - \mathtt{y} \leq 0$. For each path p_j, we generate a *loop condition formula* φ_j (expressing the condition for entering this path) and a *loop body formula* ψ_j (expressing how the values of the interesting references are changed in this path).

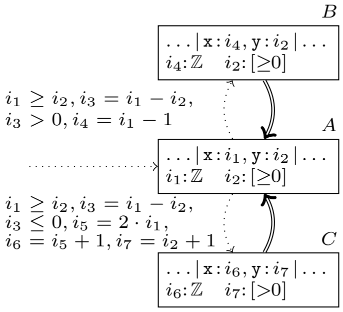

Fig. 11. Graph for `nonLoop`

The formulas φ_j and ψ_j are generated as in Step 1 of Sect. 4.1, where we add relations from $\mathcal{RelOp}$ to φ_j and constraints from $\mathcal{ArithOp}$ to ψ_j. In our example, φ_1 is $i_1 \geq i_2 \wedge i_3 > 0$ and φ_2 is $i_1 \geq i_2 \wedge i_3 \leq 0$. Moreover, ψ_1 is $i_3 = i_1 - i_2 \wedge i_4 = i_1 - 1$ and ψ_2 is $i_3 = i_1 - i_2 \wedge i_5 = 2 \cdot i_1 \wedge i_6 = i_5 + 1 \wedge i_7 = i_2 + 1$. To connect these formulas, we use a labeling function ℓ^k where for any formula ξ, $\ell^k(\xi)$ results from ξ by labeling all variables with ^{k}. We use the labels $^1, \ldots, ^n$ for the paths through the loop and the label ^{r} for the **r**esulting variables (in the second run, leaving the loop). We construct the formula

$$\rho(p_1, \ldots, p_n) = \underbrace{\mu}_{\text{invariants}} \wedge \underbrace{(\bigvee\nolimits_{j=1}^{n} (\ell^j(\varphi_j) \wedge \ell^j(\psi_j) \wedge \iota_j))}_{\text{first run through the loop}} \wedge \underbrace{(\bigwedge\nolimits_{j=1}^{n} (\neg \ell^r(\varphi_j) \wedge \ell^r(\psi_j)))}_{\text{second run, leaving the loop}}$$

Here, μ is a set of invariants that are known in the loop head node. So as we know "$i_2{:}[\geq 0]$" in state A, μ is $i_2 \geq 0$ for our example. The formula ι_j connects the variables labeled with ^{j} to the unlabeled variables in μ and to the variables labeled with ^{r} in the formulas for the second iteration. So for every integer reference i in the loop head node, ι_j contains $i = i^j$. Moreover, if i is an integer reference at position π in the loop head node s and i' is at position π in the predecessor s' of s (where there is an instance edge from s' to s), then ι_j contains $i'^j = i^r$. For our example, ι_1 is $i_1 = i_1^1 \wedge i_2 = i_2^1 \wedge i_4^1 = i_1^r \wedge i_2^1 = i_2^r$.

Intuitively, satisfiability of the first two parts of $\rho(p_1, \ldots, p_n)$ corresponds to one successful run through the loop. The third part encodes that none of the loop conditions holds in the next run. Here, we do not only consider the negated conditions $\neg \ell^r(\varphi_j)$, but we also need $\ell^r(\psi_j)$, as φ_j can contain variables computed in the loop body. For example in `nonLoop`, $\ell^r(\varphi_1)$ contains $i_3^r > 0$. But to determine how i_3^r results from the "input arguments" i_1^r, i_2^r, one needs $\ell^r(\psi_1)$ which contains $i_3^r = i_1^r - i_2^r$. If an SMT solver proves unsatisfiability of $\rho(p_1, \ldots, p_n)$, we know that whenever a variable assignment satisfies a loop condition, then after one execution of the loop body, a loop condition is satisfied again (i.e., the loop runs forever). Note that we generalized the notion of "loop conditions", as we discover the conditions by symbolic evaluation of the loop. Consequently, we can also handle loop control constructs like `break` or `continue`.

So unsatisfiability of $\rho(p_1, \ldots, p_n)$ implies that the loop is non-terminating, provided that the loop condition can be satisfied at all. To check this, we use an SMT solver to find a model for $\sigma(p_1, \ldots, p_n) = \mu \wedge (\bigvee_{j=1}^{n} (\ell^j(\varphi_j) \wedge \ell^j(\psi_j) \wedge \iota_j))$.

Theorem 8 (Non-Looping Non-Termination). *Let s be a loop head node in a termination graph where $I(s)$ only point to integer values and let $p_1, \ldots, p_n$ be all paths from s back to s. Let $\rho(p_1, \ldots, p_n)$ be unsatisfiable and let $\sigma(p_1, \ldots, p_n)$ be satisfiable by some model M (i.e., M is an assignment of integer references to concrete integers). Let $c \sqsubseteq s$ be a concrete state where every integer reference in c has been assigned the value given in M. Then c starts an infinite JBC evaluation.*

From the model M of $\sigma(p_1, \ldots, p_n)$, we obtain an instance v of the loop head node where we replace unknown integers by the values in M. Then the technique from Sect. 3 can generate a witness for the initial state of the method w.r.t. v. For our example, $i_1 = i_1^1 = i_3^1 = 1, i_2 = i_2^1 = i_2^r = i_4^1 = i_1^r = 0$ satisfies $\sigma(p_1, \ldots, p_n)$. From this, we obtain a witness for the initial state with $\mathtt{x} = 1$ and $\mathtt{y} = 0$, i.e., we automatically generate a non-terminating counterexample.

5 Evaluation and Conclusion

Based on termination graphs for Java Bytecode, we presented a technique to generate witnesses w.r.t. arbitrary error states. We then showed how to use this technique to prove the reachability of `NullPointerException`s or of non-terminating loops, which we detect by a novel SMT-based technique.

We implemented our new approach in the termination tool AProVE [12], using the SMT solver Z3 [10] and evaluated it on a collection of 325 examples. They consist of all 268 JBC programs from the *Termination Problem Data Base* that is used in the annual *International Termination Competition*,[13] all 55 examples from [24] used to evaluate the Invel tool, and the two examples from this paper. For our evaluation, we compared the old version of AProVE (without support for non-termination), the new version AProVE-No containing the results of the present paper, and Julia [20]. We were not able to obtain a running version of Invel, and thus we only compared to the results of Invel reported in [24].

We used a time-out of 60 seconds for each example. "**Y**es" and "**N**o" indicate how often termination (resp. non-termination) could be proved, "**F**ail" states how often the tool failed in less than 1 minute, "**T**" indicates how many examples led to a **T**ime-out, and "**R**" gives the average **R**untime in seconds for each example. The experiments clearly show the power of our contributions, since AProVE-No is the most powerful tool for automated non-termination proofs of Java resp. JBC programs. Moreover, the comparison between AProVE-No and AProVE indicates that the runtime for termination proofs did not increase due to the added non-termination techniques. To experiment with our implementation via a web interface and for details on the experiments, we refer to [1].

	Invel Ex.					Other Ex.				
	Y	**N**	**F**	**T**	**R**	**Y**	**N**	**F**	**T**	**R**
AProVE-No	1	51	0	3	5	204	30	12	24	11
AProVE	1	0	5	49	54	204	0	27	39	15
Julia	1	0	54	0	2	166	22	82	0	4
Invel	0	42	13	0	?					

References

1. `http://aprove.informatik.rwth-aachen.de/eval/JBC-Nonterm/`.

[13] We removed a controversial example whose termination depends on integer overflows.

2. N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.
3. B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. 2007.
4. M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl. Termination graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, LNCS 6463, pages 17–37, 2010. Extended version (with proofs) available at [1].
5. M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated detection of non-termination and `NullPointerException`s for Java Bytecode. Report AIB 2011-19, RWTH Aachen, 2011. Available at [1] and at `aib.informatik.rwth-aachen.de`.
6. M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. In *Proc. RTA '11*, LIPIcs 10, pages 155–170, 2011. Extended version (with proofs) available at [1].
7. R. Bubel, R. Hähnle, and R. Ji. Interleaving symbolic execution and partial evaluation. In *Proc. FMCO '09*, LNCS 6286, pages 247–277, 2010.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL '77*, pages 238–252. ACM Press, 1977.
9. C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17:8:1–8:37, 2008.
10. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS '08*, LNCS 4963, pages 337–340, 2008.
11. J. Giesl, R. Thiemann, P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. FroCoS '05*, LNAI 3717, pages 216–231, 2005.
12. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR '06*, LNAI 4130, pages 281–286, 2006.
13. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI '05*, pages 213–223. ACM Press, 2005.
14. A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu. Proving non-termination. In *Proc. POPL '08*, pages 147–158. ACM Press, 2008.
15. L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-`null` annotations. In *Proc. FMOODS '08*, LNCS 5051, pages 132–149, 2008.
16. T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Prentice Hall, 1999.
17. C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proc. RTA '10*, LIPIcs 6, pages 259–276, 2010. Extended version (with proofs) available at [1].
18. É. Payet and F. Mesnard. Nontermination inference of logic programs. *ACM Trans. Prog. Lang. Syst.*, 28:256–289, 2006.
19. É. Payet. Loop detection in term rewriting using the eliminating unfoldings. *Theoretical Computer Science*, 403:307–327, 2008.
20. É. Payet and F. Spoto. Experiments with non-termination analysis for Java Bytecode. In *Proc. BYTECODE '09*, ENTCS 5, pages 83–96, 2009.
21. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. FSE '10*, pages 263–272. ACM Press, 2005.
22. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In *Proc. ILPS '95*, pages 465–479. MIT Press, 1995.
23. F. Spoto. Precise `null`-pointer analysis. *Softw. Syst. Model.*, 10:219–252, 2011.
24. H. Velroyen and P. Rümmer. Non-termination checking for imperative programs. In *Proc. TAP '08*, LNCS 5051, pages 154–170, 2008.
25. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn. Scalable shape analysis for systems code. *Proc. CAV '08*, LNCS 5123, p. 385–398, 2008.

Appendix A. Proofs

To give the proof for Thm. 2, we first recapitulate the formal definition of "instance" from [4, Def. 3], where we extended it in order to handle arrays.

Definition 9 (Instance). *Let $s' = (\langle fr'_0, \ldots, fr'_n \rangle, e', h', a')$ and $s = (\langle fr_0, \ldots, fr_n \rangle, e, h, a)$, where $fr'_i = (pp'_i, lv'_i, os'_i)$ and $fr_i = (pp_i, lv_i, os_i)$. We call s' an* instance *of s (denoted $s' \sqsubseteq s$) iff $pp_i = pp'_i$ for all i and for all $\pi, \pi' \in \mathrm{SPos}(s')$:*

(a) if $h'(s'|_\pi) \in \textsc{Integers}$ and $\pi \in \mathrm{SPos}(s)$, then $h'(s'|_\pi) \subseteq h(s|_\pi) \in \textsc{Integers}$.
(b) if $h'(s'|_\pi) = \texttt{null}$ and $\pi \in \mathrm{SPos}(s)$, then $h(s|_\pi) \in \{\texttt{null}\} \cup \textsc{Unknown}$.
(c) if $h'(s'|_\pi) = (t', ?) \in \textsc{Unknown}$ and $\pi \in \mathrm{SPos}(s)$, then $h(s|_\pi) = (t, ?) \in \textsc{Unknown}$ and t' is a subtype of t.
(d) if $h'(s'|_\pi) = (t', i') \in \textsc{Arrays}$ and $\pi \in \mathrm{SPos}(s)$, then
$h(s|_\pi) = (t, i) \in \textsc{Arrays}$
or $h(s|_\pi) = (t, ?) \in \textsc{Unknown}$ and t' is a subtype of t
(e) if $h'(s'|_\pi) = (t', i', d') \in \textsc{Arrays}$ and $\pi \in \mathrm{SPos}(s)$, then
$h(s|_\pi) = (t, i, d) \in \textsc{Arrays}$
or $h(s|_\pi) = (t, i) \in \textsc{Arrays}$
or $h(s|_\pi) = (t, ?) \in \textsc{Unknown}$ and t' is a subtype of t
(f) if $h'(s'|_\pi) = (cl', f') \in \textsc{Instances}$ and $\pi \in \mathrm{SPos}(s)$, then $h(s|_\pi) = (cl, ?)$ or $h(s|_\pi) = (cl', f) \in \textsc{Instances}$, where cl' must be a subtype of cl.
(g) if $s'|_\pi \neq s'|_{\pi'}$ and $\pi, \pi' \in \mathrm{SPos}(s)$, then $s|_\pi \neq s|_{\pi'}$.
(h) if $s'|_\pi = s'|_{\pi'}$ and $\pi, \pi' \in \mathrm{SPos}(s)$ where $h'(s'|_\pi) \notin \textsc{Integers} \cup \{\texttt{null}\}$, then $s|_\pi = s|_{\pi'}$ or $s|_\pi =^? s|_{\pi'}$.
(i) if $s'|_\pi =^? s'|_{\pi'}$ and $\pi, \pi' \in \mathrm{SPos}(s)$, then $s|_\pi =^? s|_{\pi'}$.
(j) if $\big(s'|_\pi = s'|_{\pi'}$ or *$s'|_\pi =^? s'|_{\pi'}$* where *$h'(s'|_\pi) \notin \textsc{Integers} \cup \{\texttt{null}\}\big)$ and $\{\pi, \pi'\} \not\subseteq \mathrm{SPos}(s)$ with $\pi \neq \pi'$, then $s|_{\overline{\pi}} \searrow\!\swarrow s|_{\overline{\pi'}}$.*
(k) if $s'|_\pi \searrow\!\swarrow s'|_{\pi'}$, then $s|_{\overline{\pi}} \searrow\!\swarrow s|_{\overline{\pi'}}$.
(l) if $s'|_\pi!$, then $s|_{\overline{\pi}}!$.
(m) if there exist $\rho, \rho' \in (\textsc{FieldIDs} \cup \textsc{ArrayIdxs})^*$ without common non-empty prefix where $\rho \neq \rho'$, $s'|_{\pi\rho} = s'|_{\pi\rho'}$, $h'(s'|_{\pi\rho}) \notin \textsc{Integers} \cup \{\texttt{null}\}$, and ($\{\pi\rho, \pi\rho'\} \not\subseteq \mathrm{SPos}(s)$ or $s|_{\pi\rho} =^? s|_{\pi\rho'}$), then $s|_{\overline{\pi}}!$.

To prove the soundness of the algorithm `mergeStates`, we repeat its definition with added line numbers in Fig. 12.

Theorem 2. *Let $s, s' \in \textsc{States}$ and $\hat{s} = \texttt{mergeStates}(s,s')$. Then $s \sqsubseteq \hat{s} \sqsupseteq s'$.*

Proof. We only show $s \sqsubseteq \hat{s}$, as $s' \sqsubseteq \hat{s}$ is completely analogous. Let h be the heap of s and $\hat{h}$ be the heap of $\hat{s}$. We assume that the algorithm never adds the annotation $\hat{s}|_\pi =^? \hat{s}|_{\pi'}$ for $\hat{h}(\hat{s}|_\pi), \hat{h}(\hat{s}|_{\pi'}) \notin \textsc{Unknown}$. If that happens in line 10 of the algorithm, we replace one of the two values $\hat{h}(\hat{s}|_\pi)$ or $\hat{h}(\hat{s}|_{\pi'})$ by a value from $\textsc{Unknown}$ to be compatible with our requirement that in any annotation $o_1 =^? o_2$, at least one of the references points to a value from $\textsc{Unknown}$. Then we restart the annotation computation in line 6.

We prove $s \sqsubseteq \hat{s}$ by checking each of the conditions from Def. 9. Let $\pi, \pi' \in \mathrm{SPos}(s)$:

Algorithm `mergeStates`$(s,\ s')$:
01 $\hat{s}$ = `new State`(s)
02 **for** $\pi \in \text{SPos}(s) \cap \text{SPos}(s')$:
03 $\quad ref$ = `mergeRef`$(s|_\pi,\ s'|_\pi)$
04 $\quad \hat{h}(ref)$ = `mergeVal`$(h(s|_\pi),\ h'(s'|_\pi))$
05 $\quad \hat{s}|_\pi = ref$
06 **for** $\pi \neq \pi' \in \text{SPos}(s)$:
07 $\quad$ **if** $(s|_\pi = s|_{\pi'} \vee s|_\pi =^? s|_{\pi'})$
08 $\qquad \wedge\ h(s|_\pi) \notin \text{Integers} \cup \{$`null`$\}$:
09 $\qquad$ **if** $\pi, \pi' \in \text{SPos}(\hat{s})$:
10 $\qquad\quad$ **if** $\hat{s}|_\pi \neq \hat{s}|_{\pi'}$: Set $\hat{s}|_\pi =^? \hat{s}|_{\pi'}$
11 $\qquad$ **else**:
12 $\qquad\quad$ Set $\hat{s}|_{\overline{\pi}} \searrow\!\swarrow \hat{s}|_{\overline{\pi'}}$
13 $\quad$ **if** $s|_\pi \searrow\!\swarrow s|_{\pi'}$: Set $\hat{s}|_{\overline{\pi}} \searrow\!\swarrow \hat{s}|_{\overline{\pi'}}$
14 **for** $\pi \in \text{SPos}(s)$:
15 $\quad$ **if** $s|_\pi!$: Set $\hat{s}|_{\overline{\pi}}!$
16 $\quad$ **if** $\exists \rho \neq \rho' : \pi\rho, \pi\rho' \in \text{SPos}(s) \wedge s|_{\pi\rho} = s|_{\pi\rho'}$
17 $\qquad \wedge\ \rho, \rho'$ have no common prefix $\neq \varepsilon$
18 $\qquad \wedge\ h(s|_{\pi\rho}) \notin \text{Integers} \cup \{$`null`$\}$:
19 $\qquad$ **if** $\pi\rho, \pi\rho' \in \text{SPos}(\hat{s}) \wedge \hat{s}|_{\pi\rho} \neq \hat{s}|_{\pi\rho'}$:
20 $\qquad\quad$ Set $\hat{s}|_\pi!$
21 $\qquad$ **if** $\{\pi\rho, \pi\rho'\} \not\subseteq \text{SPos}(\hat{s})$: Set $\hat{s}|_{\overline{\pi}}!$
22 ... same for $\text{SPos}(s')$ ...
23 **return** $\hat{s}$

Fig. 12. Merging Algorithm

(a)-(f): These conditions hold due to the (correct) merging of values by `mergeVal` in line 4.

(g): If $s|_\pi \neq s|_{\pi'}$ and $\pi, \pi' \in \text{SPos}(\hat{s})$, then by construction $\pi, \pi' \in \text{SPos}(s')$ and thus $\hat{s}|_\pi =$ `mergeRef`$(s|_\pi,\ s'|_\pi)$ and $\hat{s}|_{\pi'} =$ `mergeRef`$(s|_{\pi'},\ s'|_{\pi'})$. Since `mergeRef` is injective, we have $\hat{s}|_\pi \neq \hat{s}|_{\pi'}$.

(h): If $s|_\pi = s|_{\pi'}$, $h(s|_\pi) \notin \text{Integers} \cup \{$`null`$\}$, and $\pi, \pi' \in \text{SPos}(\hat{s})$, then $\hat{s}|_\pi = \hat{s}|_{\pi'}$ or by line 10, $\hat{s}|_\pi =^? \hat{s}|_{\pi'}$.

(i): Analogous to (h).

(j): If $\pi \neq \pi'$, $s|_\pi = s|_{\pi'}$ or $s|_\pi =^? s|_{\pi'}$, $h(s|_\pi) \notin \text{Integers} \cup \{$`null`$\}$, and $\{\pi, \pi'\} \not\subseteq \text{SPos}(\hat{s})$, then $\hat{s}|_{\overline{\pi}} \searrow\!\swarrow \hat{s}|_{\overline{\pi'}}$ by line 12.

(k), (l): Similar to (j) (by line 13 resp. line 15).

(m): If ρ, ρ' have no common non-empty prefix, $\rho \neq \rho'$, $s|_{\pi\rho} = s|_{\pi\rho'}$, and $h(s|_\pi) \notin \text{Integers} \cup \{$`null`$\}$, then
$\quad \pi\rho, \pi\rho' \in \text{SPos}(\hat{s})$ and $\hat{s}|_{\pi\rho} = \hat{s}|_{\pi\rho'}$,
or $\pi\rho, \pi\rho' \in \text{SPos}(\hat{s})$ and $\hat{s}|_{\pi\rho} =^? \hat{s}|_{\pi\rho'}$ and by line 20, $\hat{s}|_\pi!$,
or $\{\pi\rho, \pi\rho'\} \not\subseteq \text{SPos}(\hat{s})$ and by line 21, we have $\hat{s}|_{\overline{\pi}}!$. □

Theorem 5 (Generating Witnesses). *Let (s_1, l, s_2) be an edge in the termination graph and let s_2' be a witness state for s_2 w.r.t. w. Let $s_1' \in$ States with:*

- *if* $l = \text{EVAL}_C$*, then* s_1' *is obtained from* s_2' *by applying the symbolic evaluation used between* s_1 *and* s_2 *backwards. In* s_1'*, we instantiate freshly appearing variables such that* $s_1' \sqsubseteq s_1$ *and* $s_1' \stackrel{SyEv}{\longrightarrow} s_2'$ *holds.*
- *if* $l = \text{REFINE}_R$*, then* $s_1' \sqsubseteq s_2'$*.*
- *if* $l = \text{INS}$*, then* $s_1' = s_1 \sqcap s_2'$ *(for the definition of* $\sqcap$*, see [6, Def. 2]).*

Then s_1' *is a witness state for* s_1 *w.r.t.* w*.*

Proof. For evaluation edges, we have $s_1' \sqsubseteq s_1$ and $s_1' \stackrel{SyEv}{\longrightarrow} s_2' \stackrel{SyEv}{\longrightarrow}{}^{*} w'$ for an instance $w' \sqsubseteq w$, since s_2' is a witness state w.r.t. w.

For refinement edges, we have $s_2 \sqsubseteq s_1$. As s_2' is a witness for s_2 (and thus $s_2' \sqsubseteq s_2$), we obtain $s_1' \sqsubseteq s_2' \sqsubseteq s_2 \sqsubseteq s_1$. Moreover, since $s_2' \stackrel{SyEv}{\longrightarrow}{}^{*} w'$ for an instance $w' \sqsubseteq w$, $s_1' \sqsubseteq s_2'$ implies that $s_1' \stackrel{SyEv}{\longrightarrow}{}^{*} w''$ for an instance $w'' \sqsubseteq w' \sqsubseteq w$.

For instance edges, $s_1' = s_1 \sqcap s_2'$ implies $s_1' \sqsubseteq s_1$ and $s_1' \sqsubseteq s_2'$. Hence, we obtain $s_1' \stackrel{SyEv}{\longrightarrow}{}^{*} w''$ for an instance $w'' \sqsubseteq w$ as for refinement edges. □

Theorem 7 (Looping Non-Termination). *Let* s *occur in a cycle of the termination graph. Let* $\Pi = \{\pi \in \text{SPOS}(s) \mid s|_\pi \in I(s)\}$ *be the positions of interesting references in* s*. If there is a* $v \sqsubseteq s$ *where* $v \stackrel{SyEv}{\longrightarrow}{}^{+} v'$ *for some* $v' \sqsubseteq_\Pi v$*, then any concrete state that is an instance of* v *starts an infinite* JBC *evaluation.*

Proof. Let $s = s_0, \ldots, s_n = s$ be the cyclic path from s to s in the termination graph. We assume that there is a concrete state $c \sqsubseteq v$ that starts no infinite computation. As $v \stackrel{SyEv}{\longrightarrow}{}^{+} v'$, there is a corresponding concrete JBC evaluation $c \stackrel{SyEv}{\longrightarrow}{}^{+} c'$ with $c' \sqsubseteq v'$. As $v' \sqsubseteq_\Pi v \sqsubseteq s$, this corresponds to one run through the loop.

By assumption, c and hence, c' start no infinite computation. Thus when continuing the evaluation of c', one stays within the loop for some more time, but then one leaves the loop. Let e be the last state in the evaluation of c' that is still within the loop and let e' be the first state in this evaluation which is outside the loop. So we have $c' \stackrel{SyEv}{\longrightarrow}{}^{*} e \stackrel{SyEv}{\longrightarrow} e'$ and there is a state s_i from the loop with $e \sqsubseteq s_i$ but $e' \not\sqsubseteq s_{i+1}$ (i.e, $e \stackrel{SyEv}{\longrightarrow} e'$ is the evaluation step where the loop is left).

In the termination graph, the state s_i has several successors (at least one in the loop and one outside of it). By soundness of the graph construction, e' is an instance of one of the successors of s_i.

By the definition of termination graphs, the only states with more than one successor are states with outgoing refinement edges. Let R be the set of references that these edges are marked with. Then the references from R are interesting in s_i and by the definition of interesting references, the corresponding references are also interesting in s. This means that Π contains the positions of those references whose values determine whether one leaves the loop in an instance of s_i.

Let $v_i \sqsubseteq s_i$ be the state in the evaluation $v \stackrel{SyEv}{\longrightarrow}^+ v'$ that corresponds to the state s_i (i.e., $v \stackrel{SyEv}{\longrightarrow}^* v_i \stackrel{SyEv}{\longrightarrow}^* v'$). Since $c' \sqsubseteq v' \sqsubseteq_\Pi v$ and since the evaluation steps that evaluated c' to e were also used to evaluate v to v_i, we also obtain $e \sqsubseteq_\Pi v_i$. Note that when evaluating v_i further, we stay in the loop. But as Π contains the positions of those references whose values determine whether one leaves the loop, one obtains a corresponding evaluation in e, i.e., one cannot leave the loop when evaluating e. In other words, the successor state e' cannot be outside the loop, which is a contradiction. $\square$

Theorem 8 (Non-Looping Non-Termination). *Let s be a loop head node in a termination graph where $I(s)$ only point to integer values and let $p_1, \ldots, p_n$ be all paths from s back to s. Let $\rho(p_1, \ldots, p_n)$ be unsatisfiable and let $\sigma(p_1, \ldots, p_n)$ be satisfiable by some model M (i.e., M is an assignment of integer references to concrete integers). Let $c \sqsubseteq s$ be a concrete state where every integer reference in c has been assigned the value given in M. Then c starts an infinite JBC evaluation.*

Proof. Assume that c does not start an infinite evaluation, i.e., its evaluation is finite. We now consider the sequence $c = c_0 \stackrel{SyEv}{\longrightarrow}^+ c_1 \stackrel{SyEv}{\longrightarrow}^+ \ldots \stackrel{SyEv}{\longrightarrow}^+ c_{m-1} \stackrel{SyEv}{\longrightarrow}^+ c_m$, which is part of this evaluation. Here, all c_i are instances of the loop head node s and all states in the computations between c_i and c_{i+1} are no instances of s. Moreover, no state reached after c_m in the evaluation is an instance of s. We know that m is at least 1, as M is a satisfying assignment for the disjunction of loop conditions $\sigma(p_1, \ldots, p_n)$ and thus, the loop is traversed at least once.

From the values of the integer variables in c_{m-1}, we can construct another assignment M' satisfying $\sigma(p_1, \ldots, p_n)$, as by construction, the loop is traversed once more (leading to c_m). But as the loop is left after reaching c_m, none of the loop conditions $\ell^r(\varphi_j)$ holds for these values. Moreover, M' can also be extended to satisfy all $\ell^r(\psi_j)$, which are conjunctions of equations of the form $i = i' \bowtie i''$, where i is always a fresh variable. Then M' is also satisfying assignment for $\rho(p_1, \ldots, p_n)$, which contradicts its unsatisfiability. $\square$

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

http://aib.informatik.rwth-aachen.de/

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: biblio@informatik.rwth-aachen.de

2008-01 * Fachgruppe Informatik: Jahresbericht 2007

2008-02 Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing

2008-03 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, Harald Zankl: Maximal Termination

2008-04 Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler

2008-05 Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations

2008-06 Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs

2008-07 Alexander Nyßen, Horst Lichter: The MeDUSA Reference Manual, Second Edition

2008-08 George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs

2008-09 George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs

2008-10 George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time

2008-11 George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows

2008-12 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages

2008-13 Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs

2008-14 Bastian Schlich: Model Checking of Software for Microcontrollers

2008-15 Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves

2008-16 Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study

2008-17 Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving

2008-18 Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems
2008-19 Dirk Wilking: Empirical Studies for the Application of Agile Methods to Embedded Systems
2009-02 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications
2009-03 Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices
2009-04 Daniel Klünder: Entwurf eingebetteter Software mit abstrakten Zustandsmaschinen und Business Object Notation
2009-05 George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs
2009-06 George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete
2009-07 Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I
2009-08 Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs
2009-09 Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem
2009-10 Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm
2009-11 Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs
2009-12 Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes
2009-13 Martin Zimmermann: Time-optimal Winning Strategies for Poset Games
2009-14 Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV'09)
2009-15 Joost-Pieter Katoen, Daniel Klink, Martin Neuhäußer: Compositional Abstraction for Stochastic Systems
2009-16 George B. Mertzios, Derek G. Corneil: Vertex Splitting and the Recognition of Trapezoid Graphs
2009-17 Carsten Kern: Learning Communicating and Nondeterministic Automata
2009-18 Paul Hänsch, Michaela Slaats, Wolfgang Thomas: Parametrized Regular Infinite Games and Higher-Order Pushdown Strategies
2010-02 Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time
2010-03 Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering
2010-04 René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme

2010-05 Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme

2010-06 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata

2010-07 George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms

2010-08 Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting

2010-09 George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs

2010-10 Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut

2010-11 Martin Zimmermann: Parametric LTL Games

2010-12 Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut

2010-13 Ashraf Armoush: Design Patterns for Safety-Critical Embedded Systems

2010-14 Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination

2010-15 Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode

2010-16 Christian Berger: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles

2010-17 Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten

2010-18 Ibrahim Armaç: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit

2010-19 Felix Reidl: Experimental Evaluation of an Independent Set Algorithm

2010-20 Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games

2011-02 Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting

2011-03 Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems

2011-04 Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars

2011-07 Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV

2011-08 Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog

2011-11 Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains

2011-12 Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems
2011-13 Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP
2011-14 Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games
2011-16 Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM
2011-18 Kamal Barakat: Introducing Timers to pi-Calculus

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de to obtain copies.