Department of Computer Science
Technical Report

Complexity Analysis for Java with AProVE

Florian Frohn and Jürgen Giesl

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2017-04

RWTH Aachen · Department of Computer Science · June 2017

Complexity Analysis for Java with AProVE*

Florian Frohn and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

Abstract. While AProVE is one of the most powerful tools for termination analysis of Java since many years, we now extend our approach in order to analyze the complexity of Java programs as well. Based on a symbolic execution of the program, we develop a novel transformation of (possibly heap-manipulating) Java programs to integer transition systems (ITSs). This allows us to use existing complexity analyzers for ITSs to infer runtime bounds for Java programs. We demonstrate the power of our implementation on an established standard benchmark set.

1 Introduction

Our verifier AProVE [14] is one of the leading tools for termination analysis of languages like Java, C, Haskell, Prolog, and term rewrite systems, as witnessed by its success at the annual *Termination Competition* and the termination category of the *SV-COMP* competition.[1] However, in many cases one is not only interested in termination, but in estimating the runtime of a program. Thus, *automated complexity analysis* has become an increasingly important subject and there exist several tools which analyze the complexity of programs in different languages and formalisms, e.g., [1, 3, 4, 8, 10–12, 15, 17, 18, 21, 23].[2]

In this paper, we adapt our previous approach for termination analysis of Java [5, 6, 22] in order to infer complexity bounds. In particular, the contributions of the current paper and their implementation in AProVE are crucial in our joint project *CAGE* [9, 24] with Draper Inc. and the University of Innsbruck. In this project, AProVE is used interactively to analyze the complexity of large Java programs in order to detect vulnerabilities. To the best of our knowledge, COSTA [1] is currently the only other tool which analyzes the complexity of (possibly heap manipulating) Java programs fully automatically. However, COSTA's notion of "size" for data structures significantly differs from ours and hence our technique can prove bounds for many programs where COSTA is bound to fail. See Sect. 6 for a more detailed comparison with related work.

In Sect. 2, we explain the notion of *complexity* that we analyze for Java and recall *symbolic execution graphs* (SE graphs) [5, 6, 22], which represent all possible executions of the analyzed Java program. Up to now, AProVE automatically transformed these SE graphs into *term rewrite systems* with built-in integers to

* Support by DFG grant GI 274/6-1 and the Air Force Research Laboratory (AFRL).

[1] See `http://www.termination-portal.org/wiki/Termination_Competition` and `http://sv-comp.sosy-lab.org`.

[2] The work on worst-case execution time (WCET) for real-time systems [25] is largely orthogonal to the inference of symbolic loop bounds.

analyze termination. However, this transformation is not complexity preserving for programs with non-tree shaped objects and moreover, existing techniques for termination analysis of term rewriting with built-in integers have not yet been adapted to complexity analysis. Therefore, in Sect. 3 we present a novel[3] transformation of SE graphs to *integer transition systems* (ITSs), a simple representation of integer programs suitable for complexity analysis. These ITSs are then analyzed by standard complexity analysis tools for integer programs like CoFloCo [12] and KoAT [8]. In our implementation in AProVE, we coupled our approach with these two tools to obtain an automatic technique which infers upper complexity bounds for Java programs. So in our approach, we model a Java program in several different ways (as Java (Bytecode), SE graphs, and ITSs), where the reason for the new modeling of Java programs by ITSs is their suitability for complexity analysis. Sect. 4 explains how to avoid the analysis of called auxiliary methods by providing *summaries*. This allows us to use AProVE in an interactive way and it is crucial to scale our approach to large programs within the *CAGE* project. In Sect. 5, we show how our transformation to ITSs also handles Java programs which manipulate the heap. Finally, in Sect. 6 we evaluate the power of our implementation in AProVE by experiments with an established standard benchmark set and compare AProVE's performance with COSTA.

2 Complexity of Java and Symbolic Execution Graphs

Example 1 (Variant of `SortCount` *from the* Termination Problem Data Base (TPDB)[4]*). To illustrate our approach, consider the following program. The method* `sort` *sorts a list* `l` *of natural numbers. To this end, it enumerates* $0, \ldots,$ $\max(\mathtt{l})$ *and adds each number to the result list* `r` *if it is contained in* `l`*. Its runtime complexity is in* $\mathcal{O}(\text{length}(\mathtt{l}) \cdot \max(\mathtt{l}))$*. In this paper, we show how* AProVE *infers similar complexity bounds automatically.*

```
class List{
 private int val; private List next;
 static boolean mem(int n,
                    List l){...}

 static int max(List l) {
   int m = 0;
   while (l != null) {
     if (l.val > m) {
       m = l.val;
     }
     l = l.next;
   }
   return m;
 }
 static List sort(List l) {
   int n = 0;
   List r = null;
   while (max(l) >= n) {
     if (mem(n, l)) {
       List rNew = new List();
       rNew.next = r;
       rNew.val = n;
       r = rNew;
     }
     n++;
   }
   return r;
 } ... }
```

We restrict ourselves to Java programs without arrays, exceptions, static fields, floating point numbers, class initializers, recursion, reflection, and multi-

[3] We presented a preliminary extended abstract with our "size" definition at the *15th Int. Workshop on Termination*, an informal workshop without formal reviewing or published proceedings, cf. `http://cl-informatik.uibk.ac.at/events/wst-2016`.

[4] The TPDB is the collection of examples used for the annual *Termination Competition*, available from `http://termination-portal.org/wiki/TPDB`.

threading to ease the presentation. However, our implementation supports full Java except for floating point numbers, reflection, multi-threading, and recursion (which is currently only supported for termination analysis). Moreover, we abstract from the different types of integers in Java and consider unbounded integers instead, i.e., we do not handle problems related to overflows.

Symbolic execution is a well-known technique in program verification and transformation [19]. We recapitulate the notion of SE graphs used in AProVE and refer to [5,6,22] for details on their automated construction. First, the Java program is compiled to Java Bytecode (JBC) by any standard compiler.

*Example 2 (*Java Bytecode *for the Method* `max` *from Ex. 1).*

```
1 iconst_0     //load 0 to opstack
2 istore_1     //store 0 to var 1 (m)
3 aload_0      //load l to opstack
4 ifnull 16    //jump if l is null
5 aload_0      //load l to opstack
6 getfield val//load l.val to opstack
7 iload_1      //load m to opstack
8 if_icmple 12//jump if l.val <= m
9 aload_0      //load l to opstack
10 getfield val //load l.val to opstack
11 istore_1     //store l.val into m
12 aload_0      //load l to opstack
13 getfield next//load l.next to opstack
14 astore_0     //store l.next into l
15 goto 3
16 iload_1      //load m to opstack
17 ireturn      //return m
```

The SE graph is a finite graph that represents all executions of a JBC program. Its nodes are *abstract states* which differ from concrete program states by also allowing "symbolic" (unknown) values for integers and references (i.e., addresses in the heap). In the following, $\overline{\ell}, \overline{f}$ etc. denote sequences, $|\overline{\ell}|$ is the length of $\overline{\ell}$, and $\overline{\ell}|_j$ is the j^{th} element of $\overline{\ell}$, where $\overline{\ell}$'s first element has index 0.

Definition 3 (Abstract State). *Let* $\textsc{Ref}$ *be the set of all* symbolic references*, let* $\textsc{Int}$ *be the set of all* symbolic integers,[5] *and let* $\textsc{Sym} = \textsc{Ref} \uplus \textsc{Int}$ *be the set of all* symbolic variables*. We write* $o, \tilde{o}, \ldots$ *for elements of* $\textsc{Ref}$*,* $i, \tilde{i}, \ldots$ *for elements of* $\textsc{Int}$*, and* $x, \tilde{x}, \ldots$ *for arbitrary symbolic variables from* $\textsc{Sym}$*. An* abstract state *has the form* $s = (pp, \overline{\ell}, \overline{op}, h, p) \in \textsc{State}$*, where* $pp \in \mathbb{N}$ *is the program position, i.e., the index of the next instruction to evaluate. The sequences* $\overline{\ell}, \overline{op} \in \textsc{Sym}^*$ *represent the symbolic variables stored in the local (program) variables resp. the entries of the operand stack. Here,* $\overline{\ell}|_j$ *is the value of the* j^{th} *local variable, for all* $0 \leq j < |\overline{\ell}|$.[6] *Similarly,* $\overline{op}|_0$ *is the top entry of the operand stack. The partial function* $h : \textsc{Ref} \rightarrowtail \textsc{Object}$ *maps symbolic references to* abstract objects *(i.e.,* $h(o)$ *expresses information about the object at address* o *in the heap). An* abstract object *is either* `null` *or a pair* (cl, vl) *of a* class name cl *and a function* $vl : \mathit{Fields}(cl) \to \textsc{Sym}$ *that maps all fields of* cl *to symbolic variables. The last component of* s *is a set of* predicates p. Predicates *specify heap shapes and are of the form* $o!$ *("o may point to a non-tree shaped object"),* $o =^? \tilde{o}$ *("o and $\tilde{o}$ may alias"), or* $o \searrow\!\swarrow \tilde{o}$ *("o and $\tilde{o}$ may share"). We write* $o \xrightarrow{f}_h x$ *if* $h(o) = (cl, vl)$*,* $f \in \mathit{Fields}(cl)$*, and* $vl(f) = x$*, where we omit* f *if it is irrelevant. For a state* $s = (pp, \overline{\ell}, \overline{op}, h, p)$*, we define* $\textsc{Ref}(s) = \{o \in \textsc{Ref} \mid 0 \leq j < |\overline{\ell}|, \overline{\ell}|_j \to_h^* o\} \cup \{o \in \textsc{Ref} \mid 0 \leq j < |\overline{op}|, \overline{op}|_j \to_h^* o\}$*, where* $\to_h^*$ *is the reflexive–transitive closure of* $\to_h$*.* $\textsc{Int}(s)$ *and* $\textsc{Sym}(s)$ *are defined analogously.*

[5] As we do not regard floats, JBC represents all primitive Java types as integers.

[6] For the sake of simplicity, we assume that all states are well typed throughout this paper, i.e., local variables of type `int` always store symbolic integers, etc.

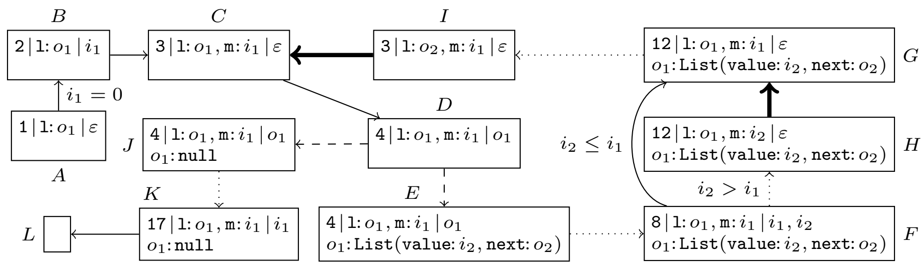


Fig. 1: SE Graph for `max`

Intuitively, an abstract state $(pp, \overline{\ell}, \overline{op}, h, p)$ can be seen as a collection of invariants. For example, if $\overline{\ell}|_0, \overline{\ell}|_1 \notin \mathsf{Dom}(h)$ (i.e., we have no concrete information about the objects at $\overline{\ell}|_0$ and $\overline{\ell}|_1$), then the *absence* of the predicate $\overline{\ell}|_0 \searrow\!\swarrow \overline{\ell}|_1$ in p means that the first two local variables do not share at program position pp, i.e., there is no path from $\overline{\ell}|_0$ to $\overline{\ell}|_1$ or vice versa, and $\overline{\ell}|_0$ and $\overline{\ell}|_1$ do not have a common successor. Let $\mathcal{T}(\mathcal{V})$ resp. $\mathcal{F}(\mathcal{V})$ be the set of all arithmetic terms resp. quantifier-free formulas over the set of variables $\mathcal{V}$ (where we only consider integer arithmetic). The edges $s \xrightarrow{\varphi} \tilde{s}$ of an SE graph are directed and labeled with formulas $\varphi \in \mathcal{F}(\textsc{Int})$ which restrict the control flow.

Fig. 1 shows an SE graph for `max`. The first line "$pp \mid \overline{\ell} \mid \overline{op}$" of a state in Fig. 1 describes its first three components, where "`l` : o_1, `m` : i_1" means that $\overline{\ell}$ is (o_1, i_1), `l` is the 0^{th} local variable, and `m` is the first local variable. In the next lines of a state, we show information about the heap, i.e., key-value pairs "$o : \ldots$" for each $o \in \mathsf{Dom}(h)$, and the predicates in p.

In State A, o_1 points to a (tree-shaped and hence acyclic) `List` or `null`, as `l` is of type `List` and $o_1 \notin \mathsf{Dom}(h)$. To express that o_1 may also point to non-tree shaped `List`s, one would need the predicate $o_1!$. Evaluating "`iconst_0`" at the program position 1 pushes the constant 0 to the operand stack, resulting in State B. This is indicated by a (solid) *evaluation edge* from A to B, labeled with the condition $i_1 = 0$ which is required to perform this evaluation step. Afterwards, i_1 is stored in the local variable `m` which yields State C. Evaluating "`aload_0`" in Line 3 pushes o_1 to the operand stack in State D. The next instruction "`ifnull 16`" jumps to Instruction 16 if the top operand stack entry is `null`. The dashed *refinement edges* connecting D with J (where o_1 is `null`) and E (where o_1 points to a `List`) correspond to a case analysis. Evaluating J results in K after two more evaluation steps (we abbreviate several evaluation edges by dotted edges). Finally, evaluating "`ireturn`" in Line 17 results in the end state L. In State E, the next four evaluation steps push `l.value` (i_2) and `m` (i_1) to the operand stack in order to compare them afterwards. In F, another case analysis is required. If `l.value` $\leq$ `m`, the next instruction is 12 (State G). Otherwise, the instructions 9 – 11 update `m` to `l.value` (i.e., `m` stores i_2 in State H) before reaching Instruction 12. Note that when evaluating F, the case analysis is not modeled by refining F, as conditions like $i_2 \leq i_1$ cannot be expressed in $\textsc{States}$. Instead, the edges connecting F with G and H are labeled with the corresponding conditions. The only difference between G and H is that we have `m` $=$ `l.value` in H, whereas m and `l.value` can be arbitrary in G. Hence, G is *more general* than H (denoted $H \sqsubseteq G$) and thus there is a (thick) *generalization edge* from H to G.

See App. A for a formal definition of when a state $\tilde{s} = (pp, \overline{\ell}, \overline{op}, \tilde{h}, \tilde{p})$ is "more general" than $s = (pp, \overline{\ell}, \overline{op}, h, p)$. Essentially, s and $\tilde{s}$ have to be at the same position pp and in both s and $\tilde{s}$, the same symbolic variables must be used for the local variables and the operand stack. We also require that all information on the heap of $\tilde{s}$ holds for s as well (i.e., we must have $\text{Ref}(\tilde{s}) \subseteq \text{Ref}(s)$, and $h(o) = \tilde{h}(o)$ for all $o \in \mathsf{Dom}(\tilde{h})$). In addition, concrete sharing (e.g., $o_1 \to_h o_3$ and $o_2 \to_h o_3$) and abstract sharing (as expressed by predicates like $(o_1 \searrow\!\!\swarrow o_2) \in p$) in s must be permitted in $\tilde{s}$ (e.g., by the predicate $(o_1 \searrow\!\!\swarrow o_2) \in \tilde{p}$) and, similarly, concrete and abstract non-tree shapes in s must be permitted in $\tilde{s}$, too. To weaken the requirement that s and $\tilde{s}$ must use the same symbolic variables, we also allow to rename the symbolic variables of the abstract state $\tilde{s}$. So we also have $s \sqsubseteq \tilde{s}$ if there is a renaming function $\mu : \text{Sym} \to \text{Sym}$ such that $s \sqsubseteq \mu(\tilde{s})$ (where we lift μ to abstract states in the obvious way). Then we say that μ *witnesses* $s \sqsubseteq \tilde{s}$. However, we only allow renaming functions μ where $\mu(o_1) = \mu(o_2)$ implies $(o_1 =^? o_2) \in \tilde{p}$ for all $o_1, o_2 \in \text{Ref}(\tilde{s})$ with $o_1 \neq o_2$. So symbolic references $o_1, o_2 \in \text{Ref}(\tilde{s})$ may only be unified by μ if aliasing is explicitly allowed by a corresponding predicate $o_1 =^? o_2$ in $\tilde{s}$. In contrast, $\mu(i_1) = \mu(i_2)$ is possible for any $i_1, i_2 \in \text{Int}(\tilde{s})$ (since different symbolic integers could represent the same number). If no renaming is required, then the witness of $s \sqsubseteq \tilde{s}$ is the identity.

Example 4 (Generalizing States). In Fig. 1, the witness for $H \sqsubseteq G$ is $\mu = \{i_1 \mapsto i_2\}$ (i.e., $\mu(i_1) = i_2$ and $\mu(x) = x$ for $x \in \{o_1, o_2, i_2\}$).

To see the effect of predicates, consider the states M and N on the side, where $N \sqsubseteq M$. Here, the predicate $o_1!$ is needed in M, as o_1 is cyclic and hence non-tree shaped in N since o_1's field `next` *points to o_1 itself. Thus, $N \sqsubseteq M$ is witnessed by $\mu = \{o_2 \mapsto o_1\}$. This witness function is valid, as we have $o_1 =^? o_2$ in M.*

M		N
$\ldots \mid \ldots \mid o_1$ o_1:`List(value:` i_1, `next:` o_2`)` $o_1!$, $o_1 =^? o_2$	$\longleftarrow$	$\ldots \mid \ldots \mid o_1$ o_1:`List(value:` i_1, `next:` o_1`)`

In Fig. 1, I results from G by setting `l` to `l.next` (Instructions 12 – 14) and going back to Step 3 ("`goto 3`" in Line 15). We draw a generalization edge from I to C, as I is a variable-renamed version of C, i.e., $I \sqsubseteq C$ with witness $\{o_1 \mapsto o_2\}$.

To see the connection between JBC and SE graphs, we now define which concrete JBC states are represented by an abstract state.

Definition 5 (Concrete State). *A* concrete state (c, τ) *is a pair of an abstract state $c = (pp, \overline{\ell}, \overline{op}, h, \varnothing) \in \text{State}$ with $\text{Ref}(c) \subseteq \mathsf{Dom}(h)$ and a valuation $\tau : \text{Int}(c) \to \mathbb{Z}$. We say that (c, τ) is* represented *by a state $s \in \text{State}$ if $c \sqsubseteq s$.*

So the heap of a concrete state has to be completely specified ($\text{Ref}(c) \subseteq \mathsf{Dom}(h)$) and hence predicates are not needed for c. The additional component τ determines the values of symbolic integers. For example, $(J, \{i_1 \mapsto n\})$ is a concrete state for all $n \in \mathbb{Z}$. Since $\sqsubseteq$ is transitive, $s \sqsubseteq \tilde{s}$ always guarantees that all concrete states represented by a state s are also represented by the state $\tilde{s}$.

As shown in [5,6,22], for any JBC program $\mathcal{P}$ one can automatically construct an SE graph $\mathcal{G}$ such that every JBC execution sequence can be *embedded* into

$\mathcal{G}$. This means that if $(c, \tau) \xrightarrow{jbc}_{\mathcal{P}} (\tilde{c}, \tilde{\tau})$ is a JBC execution step for two concrete states and $c \sqsubseteq s$ holds for some state s in $\mathcal{G}$, then $\mathcal{G}$ has a non-empty path from s to a state $\tilde{s}$ with $\tilde{c} \sqsubseteq \tilde{s}$. Hence, the paths in $\mathcal{G}$ are at least as long as the corresponding JBC sequences and therefore, SE graphs are a suitable representation for inferring upper bounds on the runtime complexity of JBC programs.

The complexity of a JBC program is a function from its inputs to its runtime. Our goal is to infer a representation of this function which is intuitive and as precise as possible. To this end, we over-approximate the complexity by a function on integers in closed form. As the inputs of a Java program can be arbitrary objects, we need a suitable mapping from objects to integers to achieve such a representation. Hence, we now define how we measure the size of objects.

Definition 6 (Size $\|\cdot\|$). *For a concrete state (c, τ) with heap h and $o \in$ $\textsc{Ref}(c)$, let $intSum(o) = 1 + \sum_{f \in Fields(cl), vl(f) = i \in \textsc{Int}} |\tau(i)|$ if $h(o) = (cl, vl)$ and $intSum(o) = 0$ if $h(o) =$ `null`. We define $\|o\|_{(c,\tau)} = \sum_{o \to_h^* \tilde{o}} intSum(\tilde{o})$.*

So the size $\|o\|_{(c,\tau)}$ of the object at the address o in (c, τ) is the number of all reachable objects $\tilde{o}$ plus the absolute values of all integers in their fields. If the same symbolic integer i is in several fields of $\tilde{o}$, then $|\tau(i)|$ is added several times.

In our opinion, this is the notion of "size" that is most suitable for measuring the runtime complexity of programs. The addend "1" in the definition of *intSum* covers features like the length of lists (i.e., an acyclic list is always greater than any of its proper sub-lists) or the number of nodes of trees. But in contrast to measures like "path length", we also take the elements of data structures into account (i.e., $\|\cdot\|$ is similar to "term size", see e.g., [13]). Since the second addend of *intSum* measures integer elements of data structures, we can analyze the complexity of algorithms like `sort` from Ex. 1 whose runtime (also) depends on the numbers that are stored in a list. Moreover in contrast to "path length", our notion of size is also suitable for cyclic objects. For example, consider a concrete state (N, τ) for State N of Ex. 4. Here, we have $\|o_1\|_{(N,\tau)} = intSum(o_1) = 1 + |\tau(i_1)|$, i.e., the size of such a cyclic list is finite.[7]

Now we can define the notion of complexity that we analyze. The *derivation height* $\mathsf{dh}_{\mathcal{P}}(c, \tau)$ of a concrete state is the length of the longest JBC execution sequence in the program $\mathcal{P}$ that starts in (c, τ). This corresponds to the usual definition of "derivation height" for other programming languages, cf. [16]. A *complexity bound* for an abstract state s is an arithmetic term $b_{\mathcal{P}}(s)$ over the variables $\mathcal{V}(s) = \{x_o \mid o \in \textsc{Ref}(s)\} \cup \textsc{Int}(s)$. Here, the variable x_o represents the *size* of the object at the symbolic reference o. Then for any valuation σ of $\mathcal{V}(s)$, $\sigma(b_{\mathcal{P}}(s))$ should be greater or equal to the length of the longest JBC execution sequence starting with a concrete state (c, τ) that is represented by s, where

[7] With this notion of *size*, the transformation from objects to *terms* that we used for termination analysis in [22] is unsound for complexity analysis, as it duplicates objects that can be reached by different fields: Consider a binary "tree" of n nodes where the left and right child of each inner node are the same. The size $\|\cdot\|$ of this object is linear in n, but the resulting transformed term would be exponential in n. This problem is avoided by our new transformation to *integers* instead of terms in Sect. 3.

the values of all $i \in \textsc{Int}(c)$ and the sizes of all $o \in \textsc{Ref}(c)$ correspond to the valuation σ. In the following, $\omega > n$ holds for all $n \in \mathbb{N}$ and for any $M \subseteq \mathbb{N} \cup \{\omega\}$, $\sup M$ is the least upper bound of M, where $\sup \varnothing = 0$.

Definition 7 (Derivation Height, Complexity Bound). *Let $\mathcal{P}$ be a* JBC *program. For every concrete state (c_0, σ_0) of $\mathcal{P}$, we define its* derivation height *as* $\mathsf{dh}_{\mathcal{P}}(c_0, \sigma_0) = \sup\{n \mid \exists (c_1, \sigma_1), \ldots, (c_n, \sigma_n) : (c_0, \sigma_0) \xrightarrow{jbc}_{\mathcal{P}} \ldots \xrightarrow{jbc}_{\mathcal{P}} (c_n, \sigma_n)\}$.

Let $s \in \textsc{State}$. A term $b_{\mathcal{P}}(s) \in \mathcal{T}(\mathcal{V}(s)) \cup \{\omega\}$ is a complexity bound for s in $\mathcal{P}$ *if for all valuations $\sigma : \mathcal{V}(s) \to \mathbb{Z}$ we have $\sigma(b_{\mathcal{P}}(s)) \geq \mathsf{dh}_{\mathcal{P}}(c, \tau)$ for any concrete state (c, τ) where some function μ witnesses $c \sqsubseteq s$, $\sigma(x_o) = \|\mu(o)\|_{(c,\tau)}$ for all $o \in \textsc{Ref}(s)$, and $\sigma(i) = \tau(\mu(i))$ for all $i \in \textsc{Int}(s)$.*

So if s is an abstract state that represents all possible concrete states at the start of a Java method in a program $\mathcal{P}$, then a complexity bound $b_{\mathcal{P}}(s)$ describes an upper bound for the runtime complexity of the Java method.

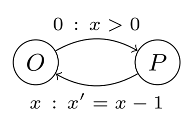
Example 8 (Runtime Complexity of `max`*). For any state $c \sqsubseteq A$ where o_1 is a list of length n, we get $\mathsf{dh}_{\mathcal{P}}(c, \tau) \leq 13 \cdot n + 6 \leq 13 \cdot \|o_1\|_{(c,\tau)} + 6$ for all valuations τ. Hence, $b_{\mathcal{P}}(A) = 13 \cdot x_{o_1} + 6$ is a complexity bound for A, which means that $13 \cdot \|\mathtt{l}\| + 6$ is an upper bound for the runtime complexity of* `max(l)` *from Ex. 2.*

3 From SE Graphs to ITSs

Now we introduce a new complexity-preserving transformation from SE graphs to *integer transition systems*. This allows us to use existing tools for complexity analysis of integer programs to infer bounds on the runtime of JBC programs.

Definition 9 (Integer Transition System). *Let $\mathcal{V}$ be a set of variables and let $\mathcal{V}' = \{x' \mid x \in \mathcal{V}\}$ be the corresponding* post-variables*. An ITS $\mathcal{I}$ is a directed graph $(\mathcal{L}, \mathcal{R})$ where $\mathcal{L}$ is the set of nodes (or* locations*) and $\mathcal{R}$ is the set of edges (or* transitions*). A* transition *$(s, \varphi, w, \tilde{s}) \in \mathcal{R}$ consists of a* source location *$s \in \mathcal{L}$, a* condition *$\varphi \in \mathcal{F}(\mathcal{V} \cup \mathcal{V}')$, a* weight *$w \in \mathcal{T}(\mathcal{V})$, and a* target location *$\tilde{s} \in \mathcal{L}$. Any valuation $\sigma : \mathcal{V} \to \mathbb{Z}$ induces a* post-valuation *$\sigma' : \mathcal{V}' \to \mathbb{Z}$ with $\sigma'(x') = \sigma(x)$ for all $x \in \mathcal{V}$. The* transition relation *$\to_{\mathcal{I}}$ of an ITS $\mathcal{I}$ operates on configurations (s, σ), where $s \in \mathcal{L}$ and σ is a valuation. For any $s, \tilde{s} \in \mathcal{L}$ and any valuations $\sigma, \tilde{\sigma} : \mathcal{V} \to \mathbb{Z}$, we have $(s, \sigma) \xrightarrow{m}_{\mathcal{I}} (\tilde{s}, \tilde{\sigma})$ if there exists a transition $(s, \varphi, w, \tilde{s}) \in \mathcal{R}$ such that $\sigma(w) = m$ and $\sigma \cup (\tilde{\sigma})'$ satisfies φ (i.e., φ is satisfied if all $x \in \mathcal{V}$ are instantiated by σ and all $x' \in \mathcal{V}'$ are instantiated according to $\tilde{\sigma}$).*

For any location s, a term $b_{\mathcal{I}}(s) \in \mathcal{T}(\mathcal{V}) \cup \{\omega\}$ is a complexity bound for s in $\mathcal{I}$ *if for all valuations $\sigma : \mathcal{V} \to \mathbb{Z}$ we have $\sigma(b_{\mathcal{I}}(s)) \geq \sum_{1 \leq j \leq n} m_j$ whenever $(s, \sigma) \xrightarrow{m_1}_{\mathcal{I}} \ldots \xrightarrow{m_n}_{\mathcal{I}} (\tilde{s}, \tilde{\sigma})$ holds for some $(\tilde{s}, \tilde{\sigma})$.*



Example 10 (ITS). Consider the ITS $\mathcal{I}$ on the right where each edge $(s, \varphi, w, \tilde{s})$ is labeled with "$w : \varphi$". It corresponds to a loop where a counter x is decremented (transition from P to O) as long as it is positive (transition from O to P). Due to the weight x of the transition from P to O, $b_{\mathcal{I}}(O) = \frac{(x+1) \cdot x}{2}$ is a complexity bound for O.

So given an initial state s, a complexity bound $b_{\mathcal{I}}(s)$ is an upper bound for the runtime complexity of $\mathcal{I}$. For instance, the complexity bound $b_{\mathcal{I}}(O)$ in Ex. 10 means that the runtime complexity of the ITS $\mathcal{I}$ is quadratic in x. We will now show how to automatically translate the SE graph of a Java program $\mathcal{P}$ into a corresponding ITS $\mathcal{I}$ such that any complexity bound $b_{\mathcal{I}}(s)$ for $\mathcal{I}$ is also a complexity bound $b_{\mathcal{P}}(s)$ for $\mathcal{P}$. We first consider programs that do not modify the heap and handle heap-manipulating programs in Sect. 5.

Let $\mathcal{G}$ be an SE graph with the states State and the edges Edge. To transform SE graphs to ITSs, we fix $\mathcal{V} = \bigcup_{s \in \text{State}} \mathcal{V}(s)$ and $\mathcal{L} = \text{State}$. Essentially, we define $(s,\ \nu_{s,\tilde{s}}(\varphi) \wedge \psi_s \wedge \rho,\ w,\ \tilde{s}) \in \mathcal{R}$ iff $s \xrightarrow{\varphi} \tilde{s} \in \text{Edge}$ where $w = 0$ if $s \xrightarrow{\varphi} \tilde{s}$ is a refinement or generalization edge and $w = 1$ if $s \xrightarrow{\varphi} \tilde{s}$ is an evaluation edge.

The substitution $\nu_{s,\tilde{s}}$ is defined as $\nu_{s,\tilde{s}}(x) = x'$ for $x \in \mathcal{V}(\tilde{s}) \setminus \mathcal{V}(s)$ and $\nu_{s,\tilde{s}}(x) = x$ for $x \in \mathcal{V}(s)$. Then $\nu_{s,\tilde{s}}(\varphi)$ is a condition on the values of the symbolic integers that must be satisfied in order to use the transition from s to $\tilde{s}$. For example, if the evaluated instruction is `iadd` (i.e., adding the two top elements of the operand stack), the operand stack of s starts with "i_1, i_2", and the operand stack of $\tilde{s}$ has the fresh symbolic integer "i_3" on top, then the edge $s \to \tilde{s}$ in the SE graph is labeled with $i_3 = i_1 + i_2$ and the corresponding transition in the ITS has the condition $i_3' = i_1 + i_2$. Thus, in the location $\tilde{s}$, the value of i_3 must be the sum of the values that i_1 and i_2 had in s.

While the semantics of arithmetic operations is captured by φ, the formula ψ_s expresses conditions on the variables x_o that represent the *sizes* $\|o\|$ of the objects in s. We define ψ_s to be the following formula, where h is the heap of s:

$$\bigwedge_{\substack{o \in \text{Ref}(s) \cap \mathsf{Dom}(h) \\ h(o) = \mathtt{null}}} x_o = 0 \quad \wedge \bigwedge_{\substack{o \in \text{Ref}(s) \cap \mathsf{Dom}(h) \\ h(o) \neq \mathtt{null}}} x_o \geq 1 \quad \wedge \bigwedge_{o \in \text{Ref}(s) \setminus \mathsf{Dom}(h)} x_o \geq 0$$

While this encoding might seem rather coarse, we achieve precision by defining a suitable formula ρ which encodes the relation between the values of the variables $x \in \mathcal{V}(s)$ and the post-variables x' (i.e., the values of the variables in $\tilde{s}$). The definition of ρ is straightforward for evaluation edges that do not modify the heap, because here the values of the symbolic variables do not change.

Definition 11 (Encoding Evaluation Edges). *Let $e = s \xrightarrow{\varphi} \tilde{s} \in \text{Edge}$ be an evaluation edge with $s = (pp, \overline{\ell}, \overline{op}, h, p)$ such that the instruction at program position pp is neither* `putfield` *nor* `new`*. Then the edge e is translated into the ITS transition $tr(e) = (s,\ \nu_{s,\tilde{s}}(\varphi) \wedge \psi_s \wedge \bigwedge_{x \in \mathcal{V}(s) \cap \mathcal{V}(\tilde{s})} x' = x,\ 1,\ \tilde{s})$.*

Example 12 (Encoding Evaluation Edges). For Fig. 1, we have $tr(C \to D) = (C,\ x_{o_1} \geq 0 \wedge x_{o_1}' = x_{o_1} \wedge i_1' = i_1,\ 1,\ D)$ and $tr(F \xrightarrow{i_2 \leq i_1} G) = (F,\ i_2 \leq i_1 \wedge x_{o_1} \geq 1 \wedge x_{o_2} \geq 0 \wedge \rho,\ 1,\ G)$ where ρ is $x_{o_1}' = x_{o_1} \wedge x_{o_2}' = x_{o_2} \wedge i_1' = i_1 \wedge i_2' = i_2$.

When transforming refinement edges to ITS transitions, we encode our knowledge about the object at the reference o that is "refined". We exploit that, by construction of $\|\cdot\|$, the size of any $\tilde{o}$ with $o \to^* \tilde{o}$ is bounded by the size of o.

Definition 13 (Encoding Refinement Edges). *Let $e = s \to \tilde{s} \in \text{Edge}$ be a refinement edge with $s = (pp, \overline{\ell}, \overline{op}, h, p)$, $\tilde{s} = (pp, \overline{\ell}, \overline{op}, \tilde{h}, \tilde{p})$, and let $o \in \text{Ref}(s)$*

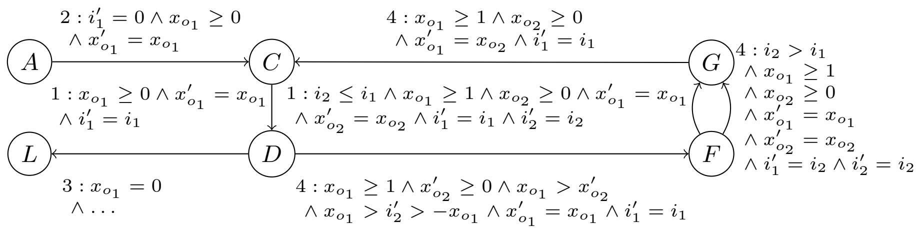


Fig. 2: ITS for the SE Graph of Fig. 1

be the symbolic reference of the object that was refined. Then $tr(e) = (s,\ \nu_{s,\tilde{s}}(\psi_{\tilde{s}}) \land \rho,\ 0,\ \tilde{s})$ *where* ρ *is* $\bigwedge_{x \in \mathcal{V}(s)} x' = x$ *if* $\tilde{h}(o) = \texttt{null}$*. Otherwise,* ρ *is*

$$\bigwedge_{x \in \mathcal{V}(s)} x' = x \quad \land \bigwedge_{\tilde{o} \in \mathrm{REF}(\tilde{s}),\ o \to_{\tilde{h}} \tilde{o}} x_o \underset{(-)}{\geq} x'_{\tilde{o}} \quad \land \bigwedge_{i \in \mathrm{INT}(\tilde{s}),\ o \to_{\tilde{h}} i} x_o > i' > -x_o.$$

Here, "$\underset{(-)}{\geq}$*" is "*$\geq$*" if* $o! \in p$ *and "*$>$*" if* $o! \notin p$*.*

Note that we can encode the knowledge from the more specialized state $\tilde{s}$ (i.e., we use $\psi_{\tilde{s}}$ instead of ψ_s), as the transition just has to be applicable in the case represented by $\tilde{s}$. The sizes of o's successor references $\tilde{o}$ are strictly smaller than $\|o\|$ if o is guaranteed to be acyclic (i.e., if $o! \notin p$), since in this case, $\tilde{o}$ reaches less objects than o. Otherwise, there might be a path from $\tilde{o}$ to o and hence we might have $\|\tilde{o}\| = \|o\|$. For symbolic integers $i \in \mathrm{INT}$ reachable from o, we know that $\|o\| > |i|$ holds due to the definition of $\|\cdot\|$. In Def. 13 we express this without using absolute values explicitly, since they are not supported by current complexity tools for ITSs.

Example 14 (Encoding Refinement Edges). For Fig. 1, we have $tr(D \to J) = (D,\ x_{o_1} = 0 \land x'_{o_1} = x_{o_1} \land i'_1 = i_1,\ 0,\ J)$*. Transforming* $D \to E$ *yields* $(D,\ x_{o_1} \geq 1 \land x'_{o_2} \geq 0 \land \rho,\ 0,\ E)$ *where* ρ *is* $x'_{o_1} = x_{o_1} \land i'_1 = i_1 \land x_{o_1} > x'_{o_2} \land x_{o_1} > i'_2 > -x_{o_1}$*.*

For generalization edges $s \to \tilde{s}$ where μ witnesses $s \sqsubseteq \tilde{s}$, the renaming μ describes how the names of the symbolic variables in s and $\tilde{s}$ are related.

Definition 15 (Encoding Generalization Edges). *Let* $e = s \to \tilde{s} \in \mathrm{EDGE}$ *be a generalization edge and let* μ *witness* $s \sqsubseteq \tilde{s}$*. We extend* μ *to* $\{x_o \mid o \in \mathrm{REF}(\tilde{s})\}$ *by defining* $\mu(x_o) = x_{\mu(o)}$*. Then* $tr(e) = (s,\ \psi_s \land \bigwedge_{x \in \mathcal{V}(\tilde{s})} x' = \mu(x),\ 0,\ \tilde{s})$*.*

Example 16 (Encoding Generalization Edges). The witness of $H \sqsubseteq G$ *is* $\{i_1 \mapsto i_2\}$*. Hence, we get* $tr(H \to G) = (H,\ x_{o_1} \geq 1 \land x_{o_2} \geq 0 \land x'_{o_1} = x_{o_1} \land x'_{o_2} = x_{o_2} \land i'_1 = i_2 \land i'_2 = i_2,\ 0,\ G)$*. Similarly, the witness of* $I \sqsubseteq C$ *is* $\{o_1 \mapsto o_2\}$*. Hence, we get* $tr(I \to C) = (I,\ x_{o_2} \geq 0 \land x'_{o_1} = x_{o_2} \land i'_1 = i_1,\ 0,\ C)$*.*

Example 17 (ITS for `max`*). Fig. 2 shows the ITS* $\mathcal{I}$ *obtained from the SE graph in Fig. 1 after simplifying it via* chaining*, i.e., subsequent transitions* $(s_1, \varphi_1, w_1, s_2)$*,* $(s_2, \varphi_2, w_2, s_3)$ *are combined to a single transition that corresponds to first applying* $(s_1, \varphi_1, w_1, s_2)$ *and then* $(s_2, \varphi_2, w_2, s_3)$*. In our implementation, we only chain such transitions if* s_2 *has exactly one incoming and one outgoing transition. Further chaining changes the original control flow of the program which was* *disadvantageous in our experiments. Again, each transition $(s, \varphi, w, \tilde{s})$ is labeled with "$w : \varphi$" in Fig. 2. State-of-the-art complexity analysis tools like* CoFloCo *[12] and* KoAT *[8] can easily infer a complexity bound $b_{\mathcal{I}}(A)$ which is linear in x_{o_1}. In Thm. 24 we will show that complexity bounds $b_{\mathcal{I}}(A)$ for the obtained ITS $\mathcal{I}$ are also upper bounds $b_{\mathcal{P}}(A)$ on the complexity of the original* Java *program $\mathcal{P}$.*

Slight modifications of our transformation *tr* from the SE graph to ITSs allow us to analyze different notions of complexity. For *space complexity*, we can simply change the weight of all evaluation edges to 0 except those that correspond to `new` instructions (i.e., in this way we infer an upper bound on the auxiliary heap space required by the method when ignoring any deallocation of memory by the garbage collector). Our technique can also easily analyze the *size* of a function's result. To this end, all transitions get weight 0 except evaluation edges that correspond to `ireturn` or `areturn` (returning an integer or a reference). Their weight is simply the top entry of the operand stack. Applying this transformation to Fig. 1 yields an ITS $\mathcal{I}'$ like Fig. 2, but the edge from D to L has weight i_1 and all other edges have weight 0. Then complexity tools can infer an upper bound like $b_{\mathcal{I}'}(A) = |x_{o_1}|$. This proves that the result of `max` is bounded by $\|\mathtt{l}\|$.

4 Summarizing Method Calls

In [5], we extended abstract states to represent the call stack. In this way, our implementation can analyze programs with method invocations like Ex. 1 fully automatically. As an alternative, we now introduce the possibility to use *summaries*, which is crucial for a modular incremental (possibly interactive) analysis of large programs. Summaries approximate the effect of method calls. Thus, AProVE can now use information about called methods without having to analyze them. Currently, such summaries have to be provided by the user as JSON files, but they can contain information obtained by previous runs of AProVE.

Example 18 (Summarizing `max`*). A possible summary for* `max` *looks as follows.*

```
"class": "List",
"methods": [{
  "name": "max",
  "descriptor": "(LList;)I",
  "static": true,
  "complexity": {
    "upperTime": "13 * arg0 + 6",
    "upperSpace": "0"
  },
  "upperSize": [{
    "pos": "ret",
    "bound": "arg0"
  }],
  "lowerSize": [{
    "pos": "ret",
    "bound": "0"
  }]
}]
```

So for a given class, each summarized method is identified by its name and descriptor.[8] *Upper bounds for the method's time and auxiliary heap space complexity can be provided as polynomials over* $\mathtt{arg0}, \ldots, \mathtt{arg}n$ *for static methods resp.* $\mathtt{this}, \mathtt{arg0}, \ldots, \mathtt{arg}n$ *for non-static methods, where* $\mathtt{arg}i$ *refers to the size of the method's i^{th} argument if it is an object resp. the value of the i^{th} argument*

[8] The descriptor specifies the argument types and return type of a method ("`LList;`" stands for the argument type `List` and "`I`" stands for the result type `int`), see `docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.3.3`

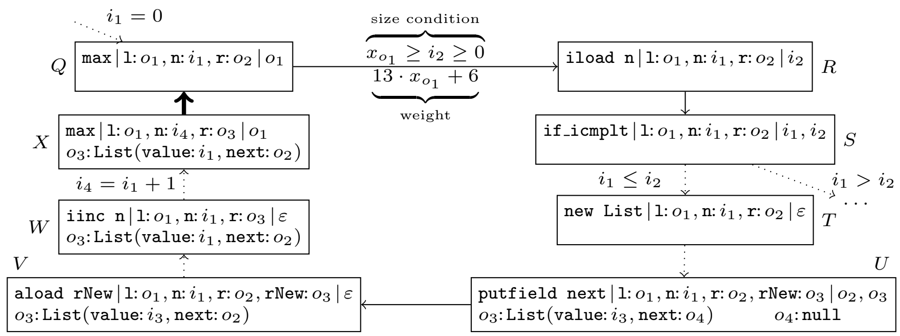


Fig. 3: SE Graph for `sort`

if it is an integer. Similarly, one can provide bounds (`upperSize` *and* `lowerSize`*) on the size of the method's result (*`ret`*).*

Our summaries are not yet expressive enough to describe heap shapes (we will improve them in future work). So for simplicity we assume that one only summarizes methods which do not manipulate the heap. Moreover, the summary for `max` is only correct if its argument is acyclic (otherwise, `max` fails to terminate). For soundness, one has to ensure that the pre-conditions of the summary are invariants of the respective class (e.g., that `List` only implements acyclic lists).

Fig. 3 shows the SE graph for `sort`. Here, we assume a summary for `mem` where `upperTime` is "`10 * arg1 + 4`", i.e., computing `mem(n,l)` takes at most $10 \cdot \|\texttt{l}\| + 4$ steps. For readability of Fig. 3, instead of program positions we described the respective JBC instructions and omitted the case `n` $>$ `max(l)` (indicated by the edge $S \xrightarrow{i_1 > i_2} \ldots$) and the case where `mem` returns `false`. The *summarization edge* $Q \to R$ is labeled with the *size condition* $x_{o_1} \geq i_2 \geq 0$ restricting the size of `max`'s result i_2 and the *weight* $13 \cdot x_{o_1} + 6$ which correspond to the summary from Ex. 18. Such summarization edges are only permitted for methods whose summary contains a finite upper runtime bound ($< \omega$). The (omitted) summarization edge for `mem` is labeled with the weight $10 \cdot x_{o_1} + 4$. The SE graph clearly reflects the quadratic complexity of `sort`: in each iteration, i_4 is set to $i_1 + 1$ (on the path from W to X) and afterwards i_4 is renamed back to i_1 (on the generalization edge $X \to Q$), i.e., i_1 is incremented. The program terminates as soon as the value of i_1 exceeds i_2, where i_2 is bounded by $\|o_1\|$. As $\|o_1\|$ never changes and i_1 is initialized to 0, the loop cannot be executed more than $\|o_1\|$ times. Since the complexity of each iteration is linear in $\|o_1\|$ due to the weights of `max` and `member`, the complexity of `sort` is quadratic. To show how we infer this quadratic bound for `sort` automatically, it remains to explain how we transform summarization edges and evaluation edges with `new` and `putfield` to ITS transitions. Encoding summarization edges is straightforward.

Definition 19 (Encoding Summarization Edges). *Let $e = s \to \tilde{s} \in \textsc{Edge}$ be a summarization edge with size condition φ and weight w. It is transformed into the ITS transition $tr(e) = (s,\ \nu_{s,\tilde{s}}(\varphi) \wedge \psi_s \wedge \bigwedge_{x \in \mathcal{V}(s) \cap \mathcal{V}(\tilde{s})} x' = x,\ w,\ \tilde{s})$.*

Example 20 (Encoding Summarization Edges). We have $tr(Q \to R) = (Q,\ x_{o_1} \geq i'_2 \geq 0 \wedge x_{o_1} \geq 0 \wedge x_{o_2} \geq 0 \wedge x'_{o_1} = x_{o_1} \wedge x'_{o_2} = x_{o_2} \wedge i'_1 = i_1,\ 13 \cdot x_{o_1} + 6,\ R)$.

5 Encoding Heap Modifications

Now we show how to encode the only instructions that modify the heap as ITS transitions. To encode the instruction `new`, we simply add the constraint $x'_o = 1$ for the newly created object o.

Example 21 (Encoding `new`*). For the (omitted) successor T' of T in Fig. 3 we have* $tr(T \to T') = (T, x_{o_1} \geq 0 \wedge x_{o_2} \geq 0 \wedge x'_{o_1} = x_{o_1} \wedge x'_{o_2} = x_{o_2} \wedge x'_{o_3} = 1 \wedge i'_1 = i_1, 1, T')$.

The only instruction that changes the size of objects is `putfield`. Note that `putfield` also changes the size of all predecessors $\tilde{o}$ of the object affected by the write access. However, our size measure $\|\cdot\|$ was defined in such a way that we can easily provide lower and upper bounds for the affected variables $x_{\tilde{o}}$.

Definition 22 (Encoding `putfield` for Object Fields). *Let $e = s \to \tilde{s} \in$ Edge be an evaluation edge with $s = (pp, \overline{\ell}, \overline{op}, h, p)$, let $\widetilde{o_{\mathtt{f}}}$ and o be the two top entries of $\overline{op}$, and let* `putfield f` *be the instruction at program position pp (i.e., $\widetilde{o_{\mathtt{f}}} \in$ Ref is written to the field* `f` *of $h(o)$). Moreover, let $o \xrightarrow{\mathtt{f}}_h o_{\mathtt{f}}$ (i.e., $o_{\mathtt{f}}$ is the former value of $h(o)$'s field* `f`*) and $\mathsf{PotPred} = \{\tilde{o} \in \textsc{Ref}(\tilde{s}) \mid \tilde{o} \rightsquigarrow o\}$ where $\tilde{o} \rightsquigarrow o$ iff $\tilde{o} \to_h^* o$ or $\tilde{o} \to_h^* \widehat{o}$ and $(\widehat{o} \searrow\!\swarrow o) \in p$ for some $\widehat{o} \in$ Ref.[9] Then $tr(e) = (s,\ \psi_s \wedge \bigwedge_{x \in \mathcal{V}(\tilde{s}) \setminus \{x_{\tilde{o}} \mid \tilde{o} \in \mathsf{PotPred}\}} x' = x \wedge \bigwedge_{\tilde{o} \in \mathsf{PotPred}} x_{\tilde{o}} + x_{\widetilde{o_{\mathtt{f}}}} \geq x'_{\tilde{o}} \geq x_{\tilde{o}} - x_{o_{\mathtt{f}}},\ 1,\ \tilde{s})$.*

So the size of all potential predecessors $\tilde{o}$ of o (captured by $\rightsquigarrow$) may change, but by definition of $\|\cdot\|$, the new size of $\tilde{o}$ is between $\|\tilde{o}\| - \|o_{\mathtt{f}}\|$ and $\|\tilde{o}\| + \|\widetilde{o_{\mathtt{f}}}\|$.

If an integer $\widetilde{i_{\mathtt{f}}}$ is written to a field by `putfield`, then we need to take the signs of $\widetilde{i_{\mathtt{f}}}$ and of the previous value $i_{\mathtt{f}}$ of the field into account, since integers contribute to $\|\cdot\|$ with their absolute value. To avoid case analyses, we infer integer invariants using standard techniques which often allow us to determine the signs of integers statically. Moreover, for simplicity we just encode the upper bound and use $x_{\tilde{o}} + \widetilde{i_{\mathtt{f}}} \geq x'_{\tilde{o}} \geq 0$ if $\widetilde{i_{\mathtt{f}}}$ is non-negative resp. $x_{\tilde{o}} - \widetilde{i_{\mathtt{f}}} \geq x'_{\tilde{o}} \geq 0$ if $\widetilde{i_{\mathtt{f}}}$ is negative, since 0 is a trivial lower bound for $\|\tilde{o}\|$. Hence, our encoding yields just one rule if the sign of $\widetilde{i_{\mathtt{f}}}$ can be determined statically and two rules, otherwise.

Example 23 (Encoding `putfield`*). We have $tr(U \to V) = (U,\ x_{o_1} \geq 0 \wedge x_{o_2} \geq 0 \wedge x_{o_3} \geq 1 \wedge x_{o_4} = 0 \wedge \rho,\ 1,\ V)$ where ρ is $x'_{o_1} = x_{o_1} \wedge x'_{o_2} = x_{o_2} \wedge i'_1 = i_1 \wedge i'_3 = i_3 \wedge x_{o_3} + x_{o_2} \geq x'_{o_3} \geq x_{o_3} - x_{o_4}$.*

The following theorem states that our transformation is sound for complexity analysis. For the proof, we refer to App. B.

Theorem 24 (Soundness Theorem). *Let $\mathcal{P}$ be a* JBC *program and $\mathcal{I}$ be the ITS which results from the SE graph for $\mathcal{P}$. Then for all $s \in$ State, any complexity bound $b_{\mathcal{I}}(s)$ for s in $\mathcal{I}$ is also a complexity bound for s in $\mathcal{P}$.*

For the initial state of the ITS resulting from the SE graph in Fig. 3, CoFloCo and KoAT infer complexity bounds in $\mathcal{O}(x_{o_1}^2)$. By Thm. 24, this proves that the runtime of `sort(l)` is quadratic in $\|\mathtt{l}\|$.

---

[9] While s may have the predicate $\widehat{o} \searrow\!\swarrow o$, it cannot contain $\widehat{o} =^? o$, as our symbolic execution rules require that if a field of o is written by `putfield`, then predicates of the form $\widehat{o} =^? o$ first have to be removed by refinement steps, cf. [5]. Similarly, $o \in \mathsf{Dom}(h)$ is enforced by refinements before symbolically evaluating `putfield`.

6 Experiments and Conclusion

Building upon AProVE's symbolic execution, we presented a new complexity-preserving transformation from heap-manipulating Java programs with user-defined data structures to integer transition systems. Furthermore, we explained how we achieve modularity using summaries. In contrast to AProVE's termination analysis which transforms Java to term rewrite systems with built-in integers, our new transformation allows us to apply powerful off-the-shelf solvers for integer programs like CoFloCo [12] and KoAT [8]. In our implementation, we run CoFloCo and KoAT in parallel to obtain complexity bounds that are as small as possible.

Clearly, our translation is also sound for termination analysis. In fact, AProVE was not able to prove termination of Ex. 1 so far. Coupling our translation with dedicated termination analysis tools for ITSs like T2 [7] is subject of future work.

Related approaches are presented in [1, 3, 4, 10, 15, 17, 18]. [3] analyzes the complexity of a Java-like language, but in contrast to our technique, it requires user-provided loop invariants. [17] analyzes ML, i.e., the considered input language differs significantly from ours. [15] regards C programs, but requires user-provided "quantitative functions over data structures" (which are similar to our optional summaries, cf. Sect. 4) and hence cannot analyze programs with data structures fully automatically. The approach in [10] also relies on user annotations to handle resource bounds that depend on the contents of the heap. The tool [4] analyzes Jinja [20], which is similar (but not equal) to a restricted subset of Java. Therefore, transforming Java to Jinja is non-trivial and no suitable tool to accomplish such a transformation is available.[10] Similarly, [18] analyzes the complexity of a language related to Java (RAJA), but a (possibly automated) transformation from full Java to RAJA is not straightforward.

Hence, we compare our implementation with COSTA [1], the only other tool for fully automated complexity analysis of Java we are aware of. Like our technique, COSTA transforms Java to an integer-based formalism (called *cost relations*). However, COSTA uses *path length* to measure the size of objects, i.e., lists are measured by length, trees by height, etc. Thus, COSTA fails for programs like Ex. 1 where one has to reason both about data structures and their elements, as `sort`'s runtime is not bounded by the length of the input list. So both COSTA and AProVE estimate how the number of executed instructions depends on the size of the program input. But as the tools use different size measures, the semantics of their results are incomparable. Thus, our experimental evaluation is just meant to give a rough impression of the capabilities of the tools.

To assess the power of our approach, we ran AProVE on all 300 non-recursive examples from the category "Java Bytecode" of the *Termination Problem Data Base* (TPDB), a well-established benchmark for automated termination analysis used at the annual *Termination Competition*, cf. Footnote 4. (So we did not include the 286 examples from the category "Java Bytecode Recursive".) We omitted 80 examples from two sub-collections of the TPDB which mainly consist of

[10] Java2Jinja (`http://pp.ipd.kit.edu/projects/quis-custodiet/Java2Jinja`) generates JinjaThreads-code, which is a superset of Jinja and cannot be handled by [4].

non-terminating examples as well as 8 further examples where AProVE proves non-termination and consequently fails to infer an upper bound. The remaining 212 examples contain 131 heap-manipulating and 81 numeric programs. AProVE finds runtime bounds for 78 heap-manipulating and 76 numeric examples, i.e., for 154 (73%) of all 212 examples, cf. Table 1. Here, n is the sum of the sizes of all object arguments and of the absolute values of all integer arguments. On average, AProVE needs 7.2 s to prove an upper bound and the median of the runtime is 4.6 s.

$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^{>3})$	?
31	102	15	1	5	58

Table 1: Results on the TPDB

Unfortunately, we cannot compare AProVE with COSTA on the TPDB directly. The reason is that the TPDB examples simulate numeric inputs by the lengths of the strings in the argument of the entry point `main(String[] args)` of the program. As COSTA abstracts arrays to their length, it loses all information about the elements of `args` and hence fails for almost all TPDB examples.

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \cdot \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^{>3})$	?
AProVE	28	0	102	0	13	2	4	62
COSTA	10	4	45	3	5	0	1	143

Table 2: Comparison with COSTA

So we adapted the 212 examples[11] of the TPDB such that they do not rely on `main`'s argument to simulate numeric inputs anymore. Instead, now a new entry point method with a suitable number of integer arguments is analyzed directly. However, this adaption is not always equivalent, as `main`'s argument array can be arbitrarily long (and hence can be used to simulate arbitrarily many numeric inputs), whereas the arity of the new entry point method is fixed. Thus, AProVE's results on these modified examples differ from the results on the TPDB in some cases. Table 2 compares both tools on these examples. AProVE succeeds in 149 cases, whereas COSTA proves an upper bound in 68 cases and infers a smaller bound than AProVE in 4 cases. Besides our novel size abstraction, further reasons why AProVE often yields better results are its precise symbolic execution and the use of more powerful back end tools (CoFloCo and KoAT) instead of COSTA's back end PUBS. On the other hand, COSTA can infer logarithmic bounds, which are not supported by AProVE. If instead of CoFloCo and KoAT we use COSTA's back end PUBS to analyze the ITSs generated by AProVE, then this modified version of AProVE still succeeds in 101 cases.

For more details on our experiments (including a selection of typical heap-manipulating programs where AProVE succeeds, but COSTA fails), the examples used to compare with COSTA, a web interface to access our implementation, and to download a virtual machine image of AProVE, we refer to [2].

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Comp. Sc.*, 413(1):142–159, 2012.
2. AProVE. `https://aprove-developers.github.io/jbc-complexity/`.

[11] We could not adapt `Julia_10_Iterative/RSA` as its sources are missing.

3. R. Atkey. Amortised resource analysis with separation logic. *Logical Methods in Computer Science*, 7(2), 2011.
4. M. Avanzini, G. Moser, and M. Schaper. TcT: Tyrolean complexity tool. In *Proc. TACAS '16*, LNCS 9636, pages 407–423, 2016.
5. M. Brockschmidt, C. Otto, C. von Essen, J. Giesl. Termination graphs for Java Bytecode. In *Verif., Induction, Termination Analysis*, LNCS 6463, pp. 17–37, 2010.
6. M. Brockschmidt, R. Musiol, C. Otto, J. Giesl. Automated termination proofs for Java programs with cyclic data. In *Proc. CAV '12*, LNCS 7358, pp. 105–122, 2012.
7. M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, and N. Piterman. T2: Temporal property verification. In *Proc. TACAS '16*, LNCS 9636, pages 387–393, 2016.
8. M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM TOPLAS*, 38(4):13:1–13:50, 2016.
9. Complexity Analysis-Based Guaranteed Execution. `http://www.draper.com/news/draper-s-cage-could-spot-code-vulnerable-denial-service-attacks`.
10. Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional certified resource bounds. In *Proc. PLDI '15*, pages 467–478, 2015.
11. S. Debray and N.-W. Lin. Cost analysis of logic programs. *ACM TOPLAS*, 15(5):826–875, 1993.
12. A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *Proc. APLAS '14*, LNCS 8858, pages 275–295, 2014.
13. S. Genaim, M. Codish, J. P. Gallagher, and V. Lagoon. Combining norms to prove termination. In *Proc. VMCAI '02*, LNCS 2937, pages 126–138, 2002.
14. J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017.
15. S. Gulwani, K. K. Mehra, T. M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. *Proc. POPL '09*, pp. 127–139. 2009.
16. D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In *Proc. RTA '89*, LNCS 355, pages 167–177, 1989.
17. J. Hoffmann, A. Das, and S.-C. Weng. Towards automatic resource bound analysis for OCaml. In *Proc. POPL '17*, pages 359–373, 2017.
18. M. Hofmann and D. Rodriguez. Automatic type inference for amortised heap-space analysis. In *Proc. ESOP '13*, LNCS 7792, pages 593–613, 2013.
19. R. Ji, R. Hähnle, and R. Bubel. Program transformation based on symbolic execution and deduction. In *Proc. SEFM '13*, LNCS 8137, pages 289–304, 2013.
20. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM TOPLAS*, 28(4):619–695, 2006.
21. L. Noschinski, F. Emmes, and J. Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *J. Automated Reasoning*, 51(1):27–56, 2013.
22. C. Otto, M. Brockschmidt, C. von Essen, J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. *Proc. RTA '10*, LIPIcs 6, pp. 259–276. 2010.
23. M. Sinn, F. Zuleger, and H. Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of Automated Reasoning*, 59(1):3–45, 2017.
24. Space/Time Analysis for Cybersecurity (STAC). `http://www.darpa.mil/program/space-time-analysis-for-cybersecurity`.
25. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM TECS*, 7(3), 2008.

A Generalizing States

In Def. 25, we give a formal definition for the notion of "more general states". This definition is a simplified variant of [5, Def. 3] (where $s \sqsubseteq \tilde{s}$ holds for even more pairs of states s and $\tilde{s}$). Here, for a sequence $\overline{f}$ of fields, let $o \xrightarrow{\overline{f}}{}^{*}_{h} x$ (resp. $o \xrightarrow{\overline{f}}{}^{+}_{h} x$ if $|\overline{f}| > 0$) denote $o \xrightarrow{\overline{f}|_1}{}_{h} \dots \xrightarrow{\overline{f}|_n}{}_{h} x$ for $n = |\overline{f}|$.

Definition 25 (Generalizing States, $\sqsubseteq$). *Let $s = (pp, \overline{\ell}, \overline{op}, h, p)$, $\tilde{s} = (pp, \overline{\ell}, \overline{op}, \tilde{h}, \tilde{p}) \in$ STATE. We say that $\tilde{s}$ is* more general *than s if $p \subseteq \tilde{p}$, $o \in \mathsf{Dom}(\tilde{h})$ implies $h(o) = \tilde{h}(o)$, and the following conditions* **(A)** - **(C)** *hold for all $o_1, o_2, o_3 \in$ REF(s). Here, let $\mathsf{ClosePred} : \text{REF}(s) \to 2^{\text{REF}(\tilde{s})}$ be defined as $\mathsf{ClosePred}(o) = \{o\}$ if $o \in \text{REF}(\tilde{s})$ and $\mathsf{ClosePred}(o) = \{\tilde{o} \in \text{REF}(\tilde{s}) \setminus \mathsf{Dom}(\tilde{h}) \mid \tilde{o} \to^{*}_{h} o\}$, otherwise.*

- **(A)** *If $o_1 \xrightarrow{\overline{f}}{}^{*}_{h} o_2$, $o_1 \xrightarrow{\overline{g}}{}^{*}_{h} o_2$, $o_1 \not\xrightarrow{\overline{g}}{}^{*}_{\tilde{h}} o_2$, $\overline{f} \neq \overline{g}$, and $\overline{f}$ and $\overline{g}$ do not have a common non-empty prefix, then $\widetilde{o_1}! \in \tilde{p}$ for all $\widetilde{o_1} \in \mathsf{ClosePred}(o_1)$.*
- **(B)** *If $o_1 \to^{+}_{h} o_3$, $o_2 \to^{+}_{h} o_3$, $o_1 \notin \mathsf{Dom}(\tilde{h})$, $o_2 \notin \mathsf{Dom}(\tilde{h})$, and $o_1 \neq o_2$, then*[12] *$(\widetilde{o_1} \curlyvee \widetilde{o_2}) \in \tilde{p}$ for all $\widetilde{o_1} \in \mathsf{ClosePred}(o_1)$, $\widetilde{o_2} \in \mathsf{ClosePred}(o_2)$ with $\widetilde{o_1} \neq \widetilde{o_2}$.*
- **(C)** *If $o_1 \to^{+}_{h} o_2$, $o_1 \notin \mathsf{Dom}(\tilde{h})$, and $o_1 \neq o_2$, then $(\widetilde{o_1} \curlyvee \widetilde{o_2}) \in \tilde{p}$ for all $\widetilde{o_1} \in \mathsf{ClosePred}(o_1)$, $\widetilde{o_2} \in \mathsf{ClosePred}(o_2)$ with $\widetilde{o_1} \neq \widetilde{o_2}$.*

In addition, we also have $s \sqsubseteq \tilde{s}$ if there is a function $\mu :$ SYM $\to$ SYM such that $s \sqsubseteq \mu(\tilde{s})$ (where we lift μ to abstract states in the obvious way), provided that $\mu(o_1) = \mu(o_2)$ implies $(o_1 =^{?} o_2) \in \tilde{p}$ for all $o_1, o_2 \in$ REF$(\tilde{s})$ with $o_1 \neq o_2$. We say that μ witnesses *$s \sqsubseteq \tilde{s}$ (and fix $\mu = \mathsf{id}$ if renaming symbolic variables is not required).*

For a symbolic reference $o \in$ REF(s), $\mathsf{ClosePred}(o)$ contains the "closest" predecessors of o which also exist in $\tilde{s}$. So if $o_1 \to_h o_2 \to_h o_3$ with $o_1, o_2 \in$ REF$(\tilde{s})$, $o_3 \notin$ REF$(\tilde{s})$, $o_1 \to_{\tilde{h}} o_2$, and $o_2 \notin \mathsf{Dom}(\tilde{h})$, then both o_1, o_2 are predecessors of o_3, but o_2 is closer (since $o_2 \notin \mathsf{Dom}(\tilde{h})$) and hence, we have $\mathsf{ClosePred}(o_3) = \{o_2\}$. According to **(A)**, if there is a concrete non-tree shape in s ("$o_1 \xrightarrow{\overline{f}}{}^{*}_{h} o_2$, $o_1 \xrightarrow{\overline{g}}{}^{*}_{h} o_2$, $\overline{f} \neq \overline{g}$") which does not exist in $\tilde{s}$ ("$o_1 \not\xrightarrow{\overline{g}}{}^{*}_{\tilde{h}} o_2$"), then the predicate $\widetilde{o_1}!$ is required in $\tilde{s}$ for all closest predecessors $\widetilde{o_1}$ of o_1. It suffices to require this for non-tree shapes resulting from "minimal" sequences of fields $\overline{f}$ and $\overline{g}$ (i.e., where $\overline{f}$ and $\overline{g}$ have no common prefix). Due to **(B)** and **(C)**, concrete sharing in s ("$o_1 \to^{+}_{h} o_3$, $o_2 \to^{+}_{h} o_3$" resp. "$o_1 \to^{+}_{h} o_2$") which does not exist in $\tilde{s}$ ("$o_1 \notin \mathsf{Dom}(\tilde{h})$") has to be allowed by the predicate "$\curlyvee$" in $\tilde{s}$.

B Soundness Proof

To prove Thm. 24, we introduce the following notion. Here, SYM $=$ REF $\cup$ INT are all symbolic variables of the SE graph and $\mathcal{V} = \{x_o \mid o \in \text{REF}\} \cup$ INT.

[12] Since both "$\curlyvee$" and "$=^{?}$" are symmetric, we do not distinguish between "$\widetilde{o_1} \curlyvee \widetilde{o_2}$" and "$\widetilde{o_2} \curlyvee \widetilde{o_1}$" resp. between "$o_1 =^{?} o_2$" and "$o_2 =^{?} o_1$".

Definition 26 (Corresponding Valuation). *Let s be an abstract state, (c,τ) be a concrete state, and let μ witness $c \sqsubseteq s$. We say that a valuation $\sigma : \mathcal{V} \to \mathbb{Z}$* corresponds to τ w.r.t. μ *if $\sigma(i) = \tau(\mu(i))$ for all $i \in \textsc{Int}(s)$ and $\sigma(x_o) = \|\mu(o)\|_{(c,\tau)}$ for all $o \in \textsc{Ref}(s)$. On all $x \in \textsc{Sym} \setminus \textsc{Sym}(s)$, $\sigma(x)$ can be arbitrary.*

For example, consider the abstract state B in Fig. 1 and a concrete state $c \sqsubseteq B$, where c maps the local variable `l` to the symbolic variable $\widehat{o}$, c's operand stack contains just $\widehat{i}$, and c's heap maps $\widehat{o}$ to `null`. Then the witness function for $c \sqsubseteq B$ is $\mu = \{o_1 \mapsto \widehat{o}, i_1 \mapsto \widehat{i}\}$, and for $\tau = \{\widehat{o} \mapsto 0, \widehat{i} \mapsto 0\}$, any valuation σ with $\sigma(i_1) = \tau(\mu(i_1)) = \tau(\widehat{i}) = 0$ and $\sigma(x_{o_1}) = \|\mu(o_1)\|_{(c,\tau)} = \|\widehat{o}\|_{(c,\tau)} = 0$ corresponds to τ w.r.t. μ. However, the value of $\sigma(i_2)$ or $\sigma(x_{o_2})$ can be arbitrary.

The crucial observation for the proof of Thm. 24 is the following lemma which shows that if an edge in the SE graph represents a step on concrete JBC states, then one can also perform a corresponding step with the ITS resulting from the SE graph.

Lemma 27 (Simulating Edges in the SE Graph by ITS Steps). *Let $\mathcal{P}$ be a JBC program and let $\mathcal{I}$ be the ITS which results from the SE graph for $\mathcal{P}$. Let (c,τ) be a concrete state, let there be a function μ which witnesses $c \sqsubseteq s$ for some state s in the SE graph, and let the valuation σ correspond to τ w.r.t. μ.*

(a) *If there is an evaluation edge $s \xrightarrow{\varphi} \tilde{s}$ in the SE graph where $(c,\tau) \xrightarrow{jbc}_{\mathcal{P}} (\tilde{c},\tilde{\tau})$, a function $\tilde{\mu}$ witnesses $\tilde{c} \sqsubseteq \tilde{s}$, and $\kappa(\varphi)$ is valid (where $\kappa(i) = \tau(\mu(i))$ for $i \in \textsc{Int}(s)$ and $\kappa(i) = \tilde{\tau}(\tilde{\mu}(i))$ for $i \in \textsc{Int}(\tilde{s}) \setminus \textsc{Int}(s)$), then $(s,\sigma) \xrightarrow{1}_{\mathcal{I}} (\tilde{s},\tilde{\sigma})$ for a valuation $\tilde{\sigma}$ which corresponds to $\tilde{\tau}$ w.r.t. $\tilde{\mu}$.*

(b) *If there is a refinement edge $s \to \tilde{s}$ in the SE graph where a function $\tilde{\mu}$ witnesses $c \sqsubseteq \tilde{s}$, then $(s,\sigma) \xrightarrow{0}_{\mathcal{I}} (\tilde{s},\tilde{\sigma})$ for a valuation $\tilde{\sigma}$ which corresponds to τ w.r.t. $\tilde{\mu}$.*

(c) *If there is a generalization edge $s \to \tilde{s}$ in the SE graph where $\tilde{\mu}$ witnesses $s \sqsubseteq \tilde{s}$, then $\mu \circ \tilde{\mu}$ witnesses $c \sqsubseteq \tilde{s}$ and $(s,\sigma) \xrightarrow{0}_{\mathcal{I}} (\tilde{s},\tilde{\sigma})$ for a valuation $\tilde{\sigma}$ which corresponds to τ w.r.t. $\mu \circ \tilde{\mu}$.*

(d) *If there is a summarization edge $s \to \tilde{s}$ in the SE graph with size condition φ and weight w where the summarized method does not manipulate the heap, $(c,\tau) \xrightarrow{jbc}{}^{k}_{\mathcal{P}} (\tilde{c},\tilde{\tau})$, a function $\tilde{\mu}$ witnesses $\tilde{c} \sqsubseteq \tilde{s}$, and $\kappa(\varphi)$ is valid (where $\kappa(i) = \tau(\mu(i))$ for $i \in \textsc{Int}(s)$, $\kappa(i) = \tilde{\tau}(\tilde{\mu}(i))$ for $i \in \textsc{Int}(\tilde{s}) \setminus \textsc{Int}(s)$, $\kappa(x_o) = \|\mu(o)\|_{(c,\tau)}$ for $o \in \textsc{Ref}(s)$, and $\kappa(x_o) = \|\tilde{\mu}(o)\|_{(\tilde{c},\tilde{\tau})}$ for $o \in \textsc{Ref}(\tilde{s}) \setminus \textsc{Ref}(s)$), then $(s,\sigma) \xrightarrow{m}_{\mathcal{I}} (\tilde{s},\tilde{\sigma})$ for a valuation $\tilde{\sigma}$ which corresponds to $\tilde{\tau}$ w.r.t. $\tilde{\mu}$ and a number $m \geq k$.*

Proof. **(a)**: There is an evaluation edge from s to $\tilde{s}$.

Let $s = (pp, \overline{\ell}, \overline{op}, h, p)$ and let $\tilde{\sigma}(i) = \tilde{\tau}(\tilde{\mu}(i))$ for all $i \in \textsc{Int}(\tilde{s})$ and $\tilde{\sigma}(x_o) = \|\tilde{\mu}(o)\|_{(\tilde{c},\tilde{\tau})}$ for all $o \in \textsc{Ref}(\tilde{s})$.

Case 1: There is no `putfield` or `new` instruction at position pp.

Then the ITS $\mathcal{I}$ has the transition $(s, \nu_{s,\tilde{s}}(\varphi) \wedge \psi_s \wedge \bigwedge_{x \in \mathcal{V}(s) \cap \mathcal{V}(\tilde{s})} x' = x, 1, \tilde{s})$.

We have to prove that the resulting transition can be used to transform (s,σ) to $(\tilde{s},\tilde{\sigma})$. To this end, we have to show that $\sigma\cup\tilde{\sigma}'$ satisfies $\nu_{s,\tilde{s}}(\varphi)\wedge\psi_s\wedge\bigwedge_{x\in\mathcal{V}(s)} x' = x$.

Recall that $\nu_{s,\tilde{s}}$ replaces all $i \in \textsc{Int}(\tilde{s}) \setminus \textsc{Int}(s)$ by i'. Hence, $(\sigma \cup \tilde{\sigma}') \circ \nu_{s,\tilde{s}}$ replaces all $i \in \textsc{Int}(\tilde{s}) \setminus \textsc{Int}(s)$ by $\tilde{\sigma}(i) = \tilde{\tau}(\tilde{\mu}(i)) = \kappa(i)$ and all $i \in \textsc{Int}(s)$ by $\sigma(i) = \tau(\mu(i)) = \kappa(i)$, since σ corresponds to τ w.r.t. μ. As $\kappa(\varphi)$ is valid, $(\sigma \cup \tilde{\sigma}')(\nu_{s,\tilde{s}}(\varphi))$ is valid as well.

Since ψ_s does not contain variables from $\mathcal{V}'$, we have $(\sigma \cup \tilde{\sigma}')(\psi_s) = \sigma(\psi_s)$. As σ corresponds to τ w.r.t. μ, we obtain $\sigma(x_o) = \|\mu(o)\|_{(c,\tau)}$ for all $o \in \textsc{Ref}(s)$. If $h(o) = \texttt{null}$, then the heap of c also maps $\mu(o)$ to $\texttt{null}$, since μ witnesses $c \sqsubseteq s$. Hence, we have $\sigma(x_o) = \|\mu(o)\|_{(c,\tau)} = 0$ and thus σ satisfies the formula $x_o = 0$. If $o \in \mathsf{Dom}(h)$ with $h(o) \neq \texttt{null}$, then the heap of c also maps $\mu(o)$ to an abstract object different from $\texttt{null}$, as μ witnesses $c \sqsubseteq s$. Therefore in (c,τ), we have $\mathit{intSum}(\mu(o)) \geq 1$ and therefore $\sigma(x_o) = \|\mu(o)\|_{(c,\tau)} \geq 1$ which means that σ satisfies the formula $x_o \geq 1$. Finally, for any $o \in \textsc{Ref}(s)$, we also have $\mu(o) \in \textsc{Ref}(c)$ since μ witnesses $c \sqsubseteq s$ and therefore, $\sigma(x_o) = \|\mu(o)\|_{(c,\tau)} \geq 0$, i.e., σ satisfies the formula $x_o \geq 0$. So to summarize, σ satisfies ψ_s.

Finally, we have to show that $\sigma \cup \tilde{\sigma}'$ satisfies $x' = x$ for all $x \in \mathcal{V}(s) \cap \mathcal{V}(\tilde{s})$, i.e., that we have $\tilde{\sigma}(x) = \sigma(x)$. The requirement follows from our definition of evaluation edges, because we use fresh symbolic variables in $\tilde{s}$ whenever a new integer value is computed. So for all $i \in \textsc{Int}(s) \cap \textsc{Int}(\tilde{s})$ we have $\sigma(i) = \tau(\mu(i)) = \tilde{\tau}(\tilde{\mu}(i)) = \tilde{\sigma}(i)$. Moreover, as the instruction does not modify the heap, we have $\textsc{Ref}(s) = \textsc{Ref}(\tilde{s})$ and for all $o \in \textsc{Ref}(s)$ we obtain $\sigma(x_o) = \|\mu(o)\|_{(c,\tau)} = \|\tilde{\mu}(o)\|_{(\tilde{c},\tilde{\tau})} = \tilde{\sigma}(x_o)$. This means that $\tilde{\sigma}(x) = \sigma(x)$ indeed holds for all $x \in \mathcal{V}(s) \cap \mathcal{V}(\tilde{s})$.

<u>*Case 2*:</u> There is a `new` instruction at position *pp*.

Then the ITS $\mathcal{I}$ has a similar transition as in Case 1, but if $o \in \textsc{Ref}(\tilde{s})$ is the symbolic reference for the newly created object, then in addition to the other constraints we have the constraint $x'_o = 1$. Clearly, $\sigma \cup \tilde{\sigma}'$ satisfies this constraint since $\tilde{\sigma}(x_0) = \|\mu(o)\|_{(\tilde{c},\tilde{\tau})} = 1$.

<u>*Case 3*:</u> There is a `putfield` instruction at position *pp*.

We only consider the case where `putfield` writes a value into an object field. The case where an integer field is written is similar. So the ITS $\mathcal{I}$ has the transition $(s,\, \psi_s \wedge \bigwedge_{x\in\mathcal{V}(\tilde{s})\setminus\{x_{\tilde{o}} \mid \tilde{o}\in\mathsf{PotPred}\}} x' = x \wedge \bigwedge_{\tilde{o}\in\mathsf{PotPred}} x_{\tilde{o}} + x_{\tilde{o}_{\mathtt{f}}} \geq x'_{\tilde{o}} \geq x_{\tilde{o}} - x_{o_{\mathtt{f}}},\, 1,\, \tilde{s})$.

As in Case 1, $\sigma\cup\tilde{\sigma}'$ satisfies ψ_s and $x' = x$ for all $x \in \mathcal{V}(\tilde{s})\setminus\{x_{\tilde{o}} \mid \tilde{o} \in \mathsf{PotPred}\}$, since the part of the heap that is reachable from $\mu(\tilde{o})$ in c resp. from $\tilde{\mu}(\tilde{o})$ in $\tilde{c}$ is not modified by the `putfield` instruction. For $\tilde{o} \in \mathsf{PotPred}$ we have

$$
\begin{aligned}
(\sigma \cup \tilde{\sigma}')(x'_{\tilde{o}}) &= \tilde{\sigma}(x_{\tilde{o}}) \\
&= \|\tilde{\mu}(\tilde{o})\|_{(\tilde{c},\tilde{\tau})} \\
&\geq \|\mu(\tilde{o})\|_{(c,\tau)} - \|\mu(o_{\mathtt{f}})\|_{(c,\tau)} \ (\dagger) \\
&= \sigma(x_{\tilde{o}}) - \sigma(x_{o_{\mathtt{f}}}) \\
&= (\sigma \cup \tilde{\sigma}')(x_{\tilde{o}} - x_{o_{\mathtt{f}}})
\end{aligned}
$$

where in the inequality (†), "$\geq$" can be replaced by "$=$" if both the following conditions hold:

- $\mu(\tilde{o})$ already reaches $\mu(\widetilde{o_{\mathtt{f}}})$ or $\mu(\widetilde{o_{\mathtt{f}}})$ is `null` in c
- $\tilde{\mu}(\tilde{o})$ does not reach $\tilde{\mu}(o_{\mathtt{f}})$ anymore or $\tilde{\mu}(o_{\mathtt{f}})$ is `null` in $\tilde{c}$

Similarly, we have

$$\begin{aligned}(\sigma \cup \tilde{\sigma}')(x_{\tilde{o}} + x_{\widetilde{o_{\mathtt{f}}}}) &= \sigma(x_{\tilde{o}}) + \sigma(x_{\widetilde{o_{\mathtt{f}}}}) \\ &= \|\mu(\tilde{o})\|_{(c,\tau)} + \|\mu(\widetilde{o_{\mathtt{f}}})\|_{(c,\tau)} \\ &\geq \|\tilde{\mu}(\tilde{o})\|_{(\tilde{c},\tilde{\tau})} \qquad (\dagger\dagger) \\ &= \tilde{\sigma}(x_{\tilde{o}}) \\ &= (\sigma \cup \tilde{\sigma}')(x'_{\tilde{o}}),\end{aligned}$$

where in the inequality (††), "$\geq$" can be replaced by "$=$" if both the following conditions hold:

- $\mu(\tilde{o})$ does not yet reach $\mu(\widetilde{o_{\mathtt{f}}})$ or $\mu(\widetilde{o_{\mathtt{f}}})$ is `null` in c
- $\tilde{\mu}(\tilde{o})$ still reaches $\tilde{\mu}(o_{\mathtt{f}})$ or $\tilde{\mu}(o_{\mathtt{f}})$ is `null` in $\tilde{c}$

(b): There is a refinement edge from s to $\tilde{s}$.

Let $s = (pp, \overline{\ell}, \overline{op}, h, p)$, $\tilde{s} = (pp, \overline{\ell}, \overline{op}, \tilde{h}, \tilde{p})$ and let $\tilde{\sigma}(i) = \tau(\tilde{\mu}(i))$ for all $i \in \textsc{Int}(\tilde{s})$ and $\tilde{\sigma}(x_o) = \|\tilde{\mu}(o)\|_{(\tilde{c},\tau)}$ for all $o \in \textsc{Ref}(\tilde{s})$. Note that for refinement edges, we have $\textsc{Sym}(s) \subseteq \textsc{Sym}(\tilde{s})$.

Now the ITS $\mathcal{I}$ has a transition of the form $(s, \nu_{s,\tilde{s}}(\psi_{\tilde{s}}) \wedge \rho, 0, \tilde{s})$, cf. Def. 13. We again have to show that $\sigma \cup \tilde{\sigma}'$ satisfies the condition of the transition.

First note that for all $x \in \mathcal{V}(s)$ we have $\tilde{\sigma}(x) = \sigma(x)$. The reason is that since both s and $\tilde{s}$ represent the same concrete state (c, τ) using the witness μ resp. $\tilde{\mu}$, we have $\mu(x) = \tilde{\mu}(x)$ for all $x \in \textsc{Sym}(s)$ and hence, $\sigma(i) = \tau(\mu(i)) = \tau(\tilde{\mu}(i)) = \tilde{\sigma}(i)$ for $i \in \textsc{Int}(s)$ and $\sigma(x_o) = \|\mu(o)\|_{(c,\tau)} = \|\tilde{\mu}(o)\|_{(c,\tau)} = \tilde{\sigma}(x_o)$ for $o \in \textsc{Ref}(s)$.

We first prove that $\sigma \cup \tilde{\sigma}'$ satisfies $\nu_{s,\tilde{s}}(\psi_{\tilde{s}})$. As in the proof of Case 1 in **(a)**, one can show that $\tilde{\sigma}$ satisfies $\psi_{\tilde{s}}$. Every constraint in $\psi_{\tilde{s}}$ contains a single variable of the form x_o (i.e., the constraint has one of the forms $x_o = 0$, $x_o \geq 1$, or $x_o \geq 0$). If $o \in \textsc{Ref}(\tilde{s}) \setminus \textsc{Ref}(s)$, then $\nu_{s,\tilde{s}}(x_o) = x'_o$ and hence $(\sigma \cup \tilde{\sigma}') \circ \nu_{s,\tilde{s}}$ satisfies all constraints about x_o as $(\sigma \cup \tilde{\sigma}')(\nu_{s,\tilde{s}}(x_o)) = \tilde{\sigma}(x_o)$. If $o \in \textsc{Ref}(s)$, then $\nu_{s,\tilde{s}}(x_o) = x_o$ and hence $(\sigma \cup \tilde{\sigma}')(\nu_{s,\tilde{s}}(x_o)) = \sigma(x_o)$. As $o \in \textsc{Ref}(s)$ implies $x_o \in \mathcal{V}(s)$, we have $\tilde{\sigma}(x_o) = \sigma(x_o)$ due to the observation above. Thus, as $\tilde{\sigma}$ satisfies all constraints in $\psi_{\tilde{s}}$, σ and hence $(\sigma \cup \tilde{\sigma}') \circ \nu_{s,\tilde{s}}$ also satisfies all constraints about x_o.

Now we show that $\sigma \cup \tilde{\sigma}'$ satisfies ρ. Clearly, for all $x \in \mathcal{V}(s)$, $\sigma \cup \tilde{\sigma}'$ satisfies $x' = x$ due to the observation above. In the case where $\tilde{h}(o) \neq$ `null` for the symbolic reference o of the object that was refined, $o \rightarrow_{\tilde{h}} \tilde{o}$ for $\tilde{o} \in \textsc{Ref}(\tilde{s})$ implies $\|\mu(o)\|_{(c,\tau)} = \|\tilde{\mu}(o)\|_{(c,\tau)} \geq \|\tilde{\mu}(\tilde{o})\|_{(c,\tau)}$ i.e., $\sigma \cup \tilde{\sigma}'$ satisfies $x_o \geq x'_{\tilde{o}}$. Moreover, this inequality is strict if $o! \notin p$, because then $\tilde{\mu}(\tilde{o})$ cannot reach $\mu(o) = \tilde{\mu}(o)$ in c. Similarly, $o \rightarrow_{\tilde{h}} i$ for $i \in \textsc{Int}(\tilde{s})$ implies $\|\mu(o)\|_{(c,\tau)} = \|\tilde{\mu}(o)\|_{(c,\tau)} > |\tau(\tilde{\mu}(i))|$ i.e., $\sigma \cup \tilde{\sigma}'$ satisfies $x_o > i' > -x_o$.

(c): There is a generalization edge from s to $\tilde{s}$.

Due to the prerequisites, $\tilde{\mu}$ witnesses $s \sqsubseteq \tilde{s}$. As μ witnesses $c \sqsubseteq s$, we immediately obtain that $\mu \circ \tilde{\mu}$ witnesses $c \sqsubseteq \tilde{s}$. Let $\tilde{\sigma}(i) = \tau(\mu(\tilde{\mu}(i)))$ for all $i \in \text{INT}(\tilde{s})$ and $\tilde{\sigma}(x_o) = \|\mu(\tilde{\mu}(o))\|_{(\tilde{c},\tau)}$ for all $o \in \text{REF}(\tilde{s})$.

The ITS $\mathcal{I}$ has the transition $(s, \psi_s \wedge \bigwedge_{x \in \mathcal{V}(\tilde{s})} x' = \tilde{\mu}(x), 0, \tilde{s})$. Again, we have to show that $\sigma \cup \tilde{\sigma}'$ satisfies the condition of the transition.

As in **(a)**, one can show that $(\sigma \cup \tilde{\sigma}')(\psi_s) = \sigma(\psi_s)$ is valid. It remains to show that for all $x \in \mathcal{V}(\tilde{s})$, we have $(\sigma \cup \tilde{\sigma}')(x') = \tilde{\sigma}(x) = \sigma(\tilde{\mu}(x)) = (\sigma \cup \tilde{\sigma}')(\tilde{\mu}(x))$. If x has the form x_o for some $o \in \text{REF}(\tilde{s})$, then we have $\tilde{\sigma}(x_o) = \|\mu(\tilde{\mu}(o))\|_{(\tilde{c},\tau)} = \sigma(x_{\tilde{\mu}(o)}) = \sigma(\tilde{\mu}(o))$. Similarly, if x is some symbolic integer $i \in \text{INT}(\tilde{s})$, then $\tilde{\sigma}(i) = \tau(\mu(\tilde{\mu}(i))) = \sigma(\tilde{\mu}(i))$.

(d): There is a summarization edge from s to $\tilde{s}$.

Let $\tilde{\sigma}(i) = \tilde{\tau}(\tilde{\mu}(i))$ for all $i \in \text{INT}(\tilde{s})$ and $\tilde{\sigma}(x_o) = \|\tilde{\mu}(o)\|_{(\tilde{c},\tilde{\tau})}$ for all $o \in \text{REF}(\tilde{s})$.

The ITS $\mathcal{I}$ has the transition $(s, \nu_{s,\tilde{s}}(\varphi) \wedge \psi_s \wedge \bigwedge_{x \in \mathcal{V}(s) \cap \mathcal{V}(\tilde{s})} x' = x, w, \tilde{s})$. We have to show that $\sigma \cup \tilde{\sigma}'$ satisfies the condition of the transition and that $\sigma(w) \geq k$.

As in the proof of **(a)**, $\nu_{s,\tilde{s}}$ replaces all $i \in \text{INT}(\tilde{s}) \setminus \text{INT}(s)$ by i'. Hence, $(\sigma \cup \tilde{\sigma}') \circ \nu_{s,\tilde{s}}$ replaces all $i \in \text{INT}(\tilde{s}) \setminus \text{INT}(s)$ by $\tilde{\sigma}(i) = \tilde{\tau}(\tilde{\mu}(i)) = \kappa(i)$ and all $i \in \text{INT}(s)$ by $\sigma(i) = \tau(\mu(i)) = \kappa(i)$, since σ corresponds to τ w.r.t. μ. Moreover, if $o \in \text{REF}(\tilde{s}) \setminus \text{REF}(s)$, then $\nu_{s,\tilde{s}}$ replaces x_o by x'_o and hence, $(\sigma \cup \tilde{\sigma}') \circ \nu_{s,\tilde{s}}$ replaces x_o by $\tilde{\sigma}(x_o) = \|\tilde{\mu}(o)\|_{(\tilde{c},\tilde{\tau})} = \kappa(x_o)$. Similarly, if $o \in \text{REF}(s)$, then $\nu_{s,\tilde{s}}(x_o) = x_o$ and hence, $(\sigma \cup \tilde{\sigma}') \circ \nu_{s,\tilde{s}}$ replaces x_o by $\sigma(x_o) = \|\mu(o)\|_{(c,\tau)} = \kappa(x_o)$. As $\kappa(\varphi)$ is valid, $(\sigma \cup \tilde{\sigma}')(\nu_{s,\tilde{s}}(\varphi))$ is valid as well.

The validity of $(\sigma \cup \tilde{\sigma}')(\psi_s) = \sigma(\psi_s)$ can also be shown as in the proof for **(a)**. It remains to show that $\sigma \cup \tilde{\sigma}'$ satisfies $x' = x$ for all $x \in \mathcal{V}(s) \cap \mathcal{V}(\tilde{s})$, i.e., that we have $\tilde{\sigma}(x) = \sigma(x)$. As in the proof for **(a)**, this requirement follows from our definition of evaluation edges, because we use fresh symbolic variables in $\tilde{s}$ whenever a new value is computed. So for all $i \in \text{INT}(s) \cap \text{INT}(\tilde{s})$ we have $\sigma(i) = \tau(\mu(i)) = \tilde{\tau}(\tilde{\mu}(i)) = \tilde{\sigma}(i)$. Moreover, since we restricted ourselves to the summarization of methods that do not manipulate the heap, for all $o \in \text{REF}(s) \cap \text{REF}(\tilde{s})$ we obtain $\sigma(x_o) = \|\mu(o)\|_{(c,\tau)} = \|\tilde{\mu}(o)\|_{(\tilde{c},\tilde{\tau})} = \tilde{\sigma}(x_o)$. This means that $\tilde{\sigma}(x) = \sigma(x)$ indeed holds for all $x \in \mathcal{V}(s) \cap \mathcal{V}(\tilde{s})$.

Finally, $\sigma(w) \geq k$ holds because of the soundness of the weight w. □

Based on the previous lemma, we can now show that any evaluation step for concrete JBC states can be simulated in the resulting ITS. Here, we write $(s, \sigma) \xrightarrow{m}^*_{\mathcal{I}} (\tilde{s}, \tilde{\sigma})$ if $(s, \sigma) \xrightarrow{m_1}_{\mathcal{I}} \ldots \xrightarrow{m_n}_{\mathcal{I}} (\tilde{s}, \tilde{\sigma})$ and $m = m_1 + \ldots + m_n$.

Lemma 28 (Simulating JBC Evaluation by ITS Steps). *Let $\mathcal{P}$ be a* JBC *program and let $\mathcal{I}$ be the ITS which results from the SE graph for $\mathcal{P}$. Let (c, τ) be a concrete state, let there be a function μ which witnesses $c \sqsubseteq s$ for some state s in the SE graph, and let the valuation σ correspond to τ w.r.t. μ.*

(a) *If $(c, \tau) \xrightarrow{jbc}_{\mathcal{P}} (\tilde{c}, \tilde{\tau})$ is a* JBC *execution step that does not correspond to a method that was summarized in the SE graph, then there exists an abstract* *state $\tilde{s}$ in the SE graph where some function $\tilde{\mu}$ witnesses $\tilde{c} \sqsubseteq \tilde{s}$ and we have $(s, \sigma) \xrightarrow{1}{}^{*}_{\mathcal{I}} (\tilde{s}, \tilde{\sigma})$ for a valuation $\tilde{\sigma}$ that corresponds to $\tilde{\tau}$ w.r.t. $\tilde{\mu}$.*

(b) *If $(c, \tau) \xrightarrow{jbc}{}^{k}_{\mathcal{P}} (\tilde{c}, \tilde{\tau})$ are k JBC execution steps that correspond to the full execution of a method that was summarized in the SE graph, then there exists an abstract state $\tilde{s}$ in the SE graph where some function $\tilde{\mu}$ witnesses $\tilde{c} \sqsubseteq \tilde{s}$ and we have $(s, \sigma) \xrightarrow{m}_{\mathcal{I}} (\tilde{s}, \tilde{\sigma})$ for a valuation $\tilde{\sigma}$ that corresponds to $\tilde{\tau}$ w.r.t. $\tilde{\mu}$ and a number $m \geq k$.*

Proof. For **(a)**, we showed in [5, Thm. 10 & 11] that every such JBC execution step can be *embedded* into the SE graph. So there is a path $s = s_0 \to \ldots \to s_{n-1} \xrightarrow{\varphi} s_n = \tilde{s}$ in the SE graph, whose last edge is an evaluation edge and whose previous $n - 1$ edges are refinement or generalization edges, such that for all $0 \leq j < n$ there is a witness μ_j for $c \sqsubseteq s_j$, there is a witness $\tilde{\mu}$ for $\tilde{c} \sqsubseteq s_n = \tilde{s}$, and $\kappa(\varphi)$ is valid, where $\kappa(i) = \tau(\mu_{n-1}(i))$ for $i \in \textsc{Int}(s_{n-1})$ and $\kappa(i) = \tilde{\tau}(\tilde{\mu}(i))$ for $i \in \textsc{Int}(s_n) \setminus \textsc{Int}(s_{n-1})$. Hence, by Lemma 27 we obtain $(s, \sigma) = (s_0, \sigma_0) \xrightarrow{0}_{\mathcal{I}} \ldots \xrightarrow{0}_{\mathcal{I}} (s_{n-1}, \sigma_{n-1}) \xrightarrow{1}_{\mathcal{I}} (s_n, \sigma_n) = (\tilde{s}, \tilde{\sigma})$ where for all $0 \leq j < n$, σ_j corresponds to τ w.r.t. μ_j, and $\sigma_n = \tilde{\sigma}$ corresponds to $\tilde{\tau}$ w.r.t. $\tilde{\mu}$.

For **(b)**, the concrete JBC execution $(c, \tau) \xrightarrow{jbc}{}^{k}_{\mathcal{P}} (\tilde{c}, \tilde{\tau})$ is represented by a summarization edge $s \to \tilde{s}$ in the SE graph with size condition φ and weight w. Thus, there is a witness $\tilde{\mu}$ for $\tilde{c} \sqsubseteq \tilde{s}$ and $\kappa(\varphi)$ is valid, where $\kappa(i) = \tau(\mu(i))$ for $i \in \textsc{Int}(s)$ and $\kappa(i) = \tilde{\tau}(\tilde{\mu}(i))$ for $i \in \textsc{Int}(\tilde{s}) \setminus \textsc{Int}(s)$. By Lemma 27 we obtain $(s, \sigma) \xrightarrow{m}_{\mathcal{I}} (\tilde{s}, \tilde{\sigma})$ for a valuation $\tilde{\sigma}$ that corresponds to $\tilde{\tau}$ w.r.t. $\tilde{\mu}$ and a number $m \geq k$. □

Now we can prove the main soundness theorem.

Theorem 24 (Soundness Theorem). *Let $\mathcal{P}$ be a JBC program and $\mathcal{I}$ be the ITS which results from the SE graph for $\mathcal{P}$. Then for all $s \in \textsc{State}$, any complexity bound $b_{\mathcal{I}}(s)$ for s in $\mathcal{I}$ is also a complexity bound for s in $\mathcal{P}$.*

Proof. Let $s \in \textsc{State}$ be an arbitrary abstract state and let $\sigma : \mathcal{V} \to \mathbb{Z}$ be an arbitrary valuation. Moreover, let (c, τ) be a concrete state where some function μ witnesses $c \sqsubseteq s$ and σ corresponds to τ w.r.t. μ. We show that if (c, τ) starts an JBC execution of length m with the program $\mathcal{P}$ (i.e., $(c, \tau) = (c_0, \tau_0) \xrightarrow{jbc}_{\mathcal{P}} (c_1, \tau_1) \xrightarrow{jbc}_{\mathcal{P}} \ldots \xrightarrow{jbc}_{\mathcal{P}} (c_m, \tau_m)$), then we have $\sigma(b_{\mathcal{I}}(s)) \geq m$.

W.l.o.g., the JBC execution of (c, τ) should be long enough such that (c_m, τ_m) is not within any method that has been summarized and is only represented by summarization edges in the SE graph. One can always continue the JBC execution of (c, τ) until a return state of this method is reached, as summarization edges are only permitted for methods with a finite upper runtime bound (i.e., for terminating methods).

Then by Lemma 28, we immediately obtain that in the corresponding ITS $\mathcal{I}$, (s, σ) starts an evaluation of at least weight m, which implies $\sigma(b_{\mathcal{I}}(s)) \geq m$. □

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`

2014-01 * Fachgruppe Informatik: Annual Report 2014
2014-02 Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software
2014-03 Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide
2014-04 Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata
2014-05 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic
2014-06 Esther Horbert, Germán Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video
2014-07 Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations
2014-08 Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs
2014-09 Thomas Ströder and Terrance Swift (Editors): Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014
2014-14 Florian Schmidt, Matteo Ceriotti, Niklas Hauser, and Klaus Wehrle: HotBox: Testing Temperature Effects in Sensor Networks
2014-15 Dominique Gückel: Synthesis of State Space Generators for Model Checking Microcontroller Code
2014-16 Hongfei Fu: Verifying Probabilistic Systems: New Algorithms and Complexity Results
2015-01 * Fachgruppe Informatik: Annual Report 2015
2015-02 Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications
2015-05 Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity

2015-06 Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"

2015-07 Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon

2015-08 Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization

2015-09 Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models

2015-11 Stefan Wüller, Marián Kühnel, and Ulrike Meyer: Information Hiding in the Public RSA Modulus

2015-12 Christoph Matheja, Christina Jansen, and Thomas Noll: Tree-like Grammars and Separation Logic

2015-13 Andreas Polzer: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen

2015-14 Niloofar Safiran and Uwe Naumann: Symbolic vs. Algorithmic Differentiation of GSL Integration Routines

2016-01 * Fachgruppe Informatik: Annual Report 2016

2016-02 Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems

2016-03 Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl: Lower Runtime Bounds for Integer Programs

2016-04 Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution

2016-05 Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization

2016-06 Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud

2016-07 Sebastian Biallas : Verification of Programmable Logic Controller Code using Model Checking and Static Analysis

2016-09 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann: Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic

2016-10 Stefan Wüller, Ulrike Meyer, and Susanne Wetzel: Towards Privacy-Preserving Multi-Party Bartering

2017-01 * Fachgruppe Informatik: Annual Report 2017

2017-02 Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.