Automatic Analysis of Termination Arguments for Java Programs

by

David Kräutmann

Research Group Computer Science 2
RWTH Aachen University

First supervisor: Prof. Dr. Jürgen Giesl
Second supervisor: Prof. Dr. Thomas Noll

Advisor: David Keller

This thesis is submitted on June, 2020 in partial fulfilment of the requirements for the degree of Master of Science in Computer Science

Abstract

In order to facilitate further analysis, it's useful to know not only whether a program terminates, but also *why*. We developed a system to automatically determine termination-relevant invariants using Attestor for heap shape analysis, an conversion step to integer transition systems, and termination checking via T2. A novel proof certificate analysis then extracts the invariants. We then perform relevancy analysis, allowing us to determine which variables are relevant to program termination and which are not.

Contents

CHAPTER 1

INTRODUCTION

When verifying programs, functional correctness and termination are complementary approaches. Proofs of functional correctness, such as via Hoare logic, usually only prove that given some preconditions, some postconditions hold—crucially, they assume that the program is terminating. However, if it does not, arbitrary postconditions are provable. Similarly, termination checking only proves that the program terminates. By combining both termination checking and partial functional correctness proofs, one can prove total correctness.

Termination checking also produces deductions about the program—invariants that hold with each loop iteration. These deductions can be used in functional correctness checks as assertions, or to simply inform the user that the variables occurring in the invariants are relevant to termination. A more complicated analysis could then ignore all variables that are not relevant to termination.

Current termination checking approaches, such as in AProVE [1], abstract over both structural and size heap information in order to do termination analysis via integer transition systems (ITS). This kind of combined analysis leads to a large increase in complexity. A new approach is to separate size and structural analysis, perform a termination proof using those separate analyses, and then extract proof arguments from the termination proof to perform more reasoning about the structure.

We introduce a method to automatically analyse Java programs and extract proof invariants over integer and heap programs. We start with a structural analysis via Attestor [2], leading to a state space. The code is then converted into an ITS in a way that allows easy backpropagation from ITS variables to Jimple (and thus Java) variables. ITS termination checking is performed via T2 [3] and the termination certificate is analyzed to extract termination-relevant in-

variants, which can then be further refined into termination arguments. We use the invariants to extract termination-relevant variables. To the best of our knowledge, this method of extracting information from proof certificates is completely novel.

In chapter 2, we give an overview of integer transition systems, Attestor, and T2. Chapter 3 describes the transformation from Java code to ITS, backpropagation of ITS results to Java code, and extraction of invariants. A correctness proof is given in chapter 4, and results are discussed in chapter 5. Finally, a short overview of related work is given in chapter 6.

CHAPTER 2

BACKGROUND

2.1 Attestor

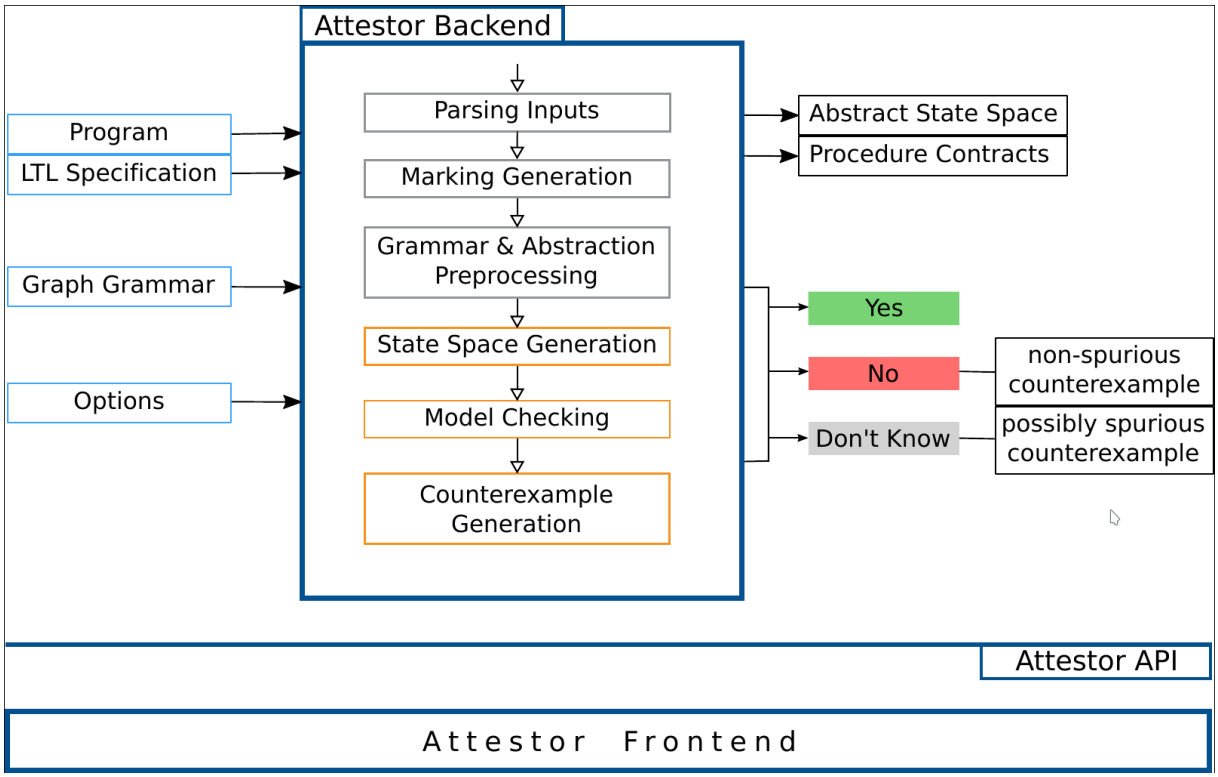


Figure 2.1: Attestor architecture—from [4]

Attestor [2, 4] is a verification tool for Java programs with dynamic heap data, capable of proving structural properties of programs. It constructs an abstract state space from the input program via symbolic execution, which can then be used for proving program properties via model checking. We use Attestor as

the first step in our analysis in order to acquire structural and aliasing information, and as a foundation for future work.

Attestor is structured in phases. A representation of those is given in fig. 2.1. Each phase potentially depends on the input of a previous phase.

- The *input* phase parses the Java program, graph grammar for generation of an abstract state space, linear temporal logic (LTL) specification, and potentially contracts that some methods will fullfil (i.e. axioms). Each input is parsed in their own phase. This phase also performs the Java-to-Jimple conversion and generates the abstract program semantics from Jimple.
- The *marking generation* phase is required for some LTL specifications. Here, special variables are generated that cannot be accessed by the actual program, which can be used to track fixed locations during state space generation.
- *Grammar and abstraction preprocessing* performs various tasks so that state space generation can be run.
- *State space generation* is the symbolic execution loop of Attestor. Abstract program semantics from the input phase are applied until concrete semantics, i.e. semantics for executing a program on concrete heap configurations, can be used. The resulting heap configuration is canonicalized and some cleanup is performed, and the state is inserted into the state space (unless it already exists or is subsumed by another state).
- During *model checking*, the LTL specifications are checked against the state space. Either the specification is satisfied, violated, or state space generation failed and the validity is unknown.
- If the LTL specification is violated, *counterexample generation* generates a counterexample.

The result of executing Attestor is the abstract state space, contracts for all analyzed methods for potential future use, and the results of the model checking operation. The abstract state space is the part that we require for our analysis. We will later build an ITS by recursing on the state space, treating the program counter associated with each state as a location and constructing transition actions according to the state transition. The information from the state space is required to ensure that soundness of termination is maintained, i.e. the ITS terminates if the program terminates.

2.1.1 Java abstract semantics

To understand the transformations and later prove their correctness, we need to go over the abstract Java semantics used by Attestor. We extend the abstract semantics to represent expressions in more detail, allowing us to accurately carry over integer calculations and boolean formulae performed by the program into the ITS.

The abstract semantics are based on Jimple [5], but only translate assignment, identity, goto, if, invoke, and return statements. Notably, throw or switch statements are unsupported.

We will define the target abstract semantics in this section—some nomenclature about types in **??** 2.1, the statement semantics in **??** 2.2 and the expression semantics in **??** 2.3. Definition 2.1 contains some preliminaries for reasoning about Jimple.

Definition 2.1 (Types)**.** Let $\mathcal{T}$ be the space of types. We consider $\tau : \mathcal{T}$ a primitive type iff it is one of `int`, `bool`, `char`, `byte`, `short`, `long`, `double`, `string`[a]. Otherwise it is an object type.

[a] Since strings are immutable, we do not consider them object types

Definition 2.2 (Statements)**.** Jimple statements are translated into one of the following abstract statements:

- `AssignInvoke` statements encode a Java statement of the form `local = someMethod(args);` where `local` is a local variable. `args` contains the object that the method belongs to.
- `AssignStmt` encodes a Java statement of the form `local = expr;`, `local.field = imm;`, or `staticfield = imm;` where `local` is a local variable, `local.field` is a field of an object, `staticfield` is a static field of an object and `imm` is a local variable or constant.
- `InvokeStmt` is a method call without assignment, i.e. `someMethod(args);` As with `AssignInvoke`, `args` contains the object that the method belongs to.
- `GotoStmt` encodes jumps to a program counter
- `IdentityStmt` models assignments of e.g. `this` or parameters to locals.
- `IfStmt` performs jumps to one program counter if a condition is true, and to another otherwise.
- `ReturnValueStmt` encodes `return value;`

- `ReturnVoidStmt` encodes `return;`

Each statement contains the current program counter and can compute the successor states given a current program state.

Definition 2.3 (Expression)**.** An expression is modelled by a `Value`, which can take one of the following forms:

- `Local` represents a local variable.
- `Field` is an object selector, e.g. `list.next` where `list` is a local variable.
- `NewExpr` is a newly created object, before any constructor is called.
- `LengthExpr` is the length of an array
- `ArithExpr` is a binary arithmetic operation on two `Values`
- `CompareExpr` is a boolean comparison between two `Values`
- `AndExpr`, `OrExpr`, `EqualExpr`, `UnequalExpr` and `NotExpr` correspond to their respective boolean operations
- `IntConstant` and `NullConstant` represent integer and `null` constants, respectively
- `UndefinedValue` are values not supported by the abstract semantics

All expressions have a type and potentially evaluate to a `ConcreteValue` given a program state, i.e. a node of the hypergraph.

Originally, Attestor did not have `CompareExpr` or `ArithExpr`. We added these for purposes of our analysis in order to allow for a tighter overapproximation and better invariants.

Definition 2.1 (Jimple semantics)**.** We denote a step in the standard Jimple semantics as $\langle C_s, s\rangle \to \langle C_t, t\rangle$ where s is a Jimple program state. A Jimple program state consists of a set of local variables V, a stack $\alpha : \forall v : V,\ \mathrm{Local}_{T(v)}$, and a heap $H : \forall t : \mathcal{T}, \mathrm{Ref}_t \to \mathrm{Obj}_t$.
$T : V \cup \mathrm{Fields} \to \mathcal{T}$ is a typing judgement.
Local_t is $\mathbb{Z}$ if t is an integer type (including `bool`), Ref_t if it is a reference, and $\varnothing$ otherwise—non-integer primitives are not relevant for our model.
Ref_t is an opaque typed reference or `null`, and Obj_t is a mapping $\forall f : \mathrm{Fields}(t),\ \mathrm{Local}_{T(f)}$.
The program counter at a statement C is given by $\mathrm{pc}(C)$ and uniquely identifies it.

2.2 Integer transition systems

An integer transition system (ITS) is an intermediary formal language to provide a language-independent way of defining program flow. Intuitively speaking, ITSs are a simple programming language with only gotos used as control flow, but no loops. The only data type in ITSs, as the name suggests, are the mathematical integers $\mathbb{Z}$. Alternately, ITSs can be considered automata—they have locations and transitions between locations. The state is represented by a location and a variable assignment. Each transition might also be guarded by a boolean expression over the integers, and has a formula that describes how variables are changed when taking the transition.

Since large parts of this work are based on the certified T2 project via ISAFoR, we utilize a similar formulation as given in [6] for logic transition systems (LTS). We will translate input programs into an ITS in order to prove termination during our analysis.

Definition 2.2 (Preliminaries)**.** An assignment α on a set of V of variables is a function that assigns each variable an integer, i.e. $\forall v : V,\ \alpha(v) \in \mathbb{Z}$. We denote $\mathrm{A}_V = V \to \mathbb{Z}$ as the type of assignments over V.
A formula $\Lambda(V)$ over integer variables V is defined inductively as follows: for $\phi,\ \psi : \Lambda(V)$ and $v,\ w : V$

- $v \ \mathsf{op}\ w : \Lambda(V)$ where $\mathsf{op} \in \{ <, >, \leq, \geq, =, \neq \}$
- $\phi \wedge \psi : \Lambda(V)$
- $\phi \vee \psi : \Lambda(V)$
- $\neg\psi : \Lambda(V)$

The disjoint union $\alpha \uplus \beta$ of two assignments $\alpha : V_1,\ \beta : V_2$ is defined only if $V_1 \cap V_2 = \varnothing$. In that case, $(\alpha \uplus \beta)(v) = \alpha(v)$ if $v \in V_1$, and $\beta(v)$ otherwise.
We say α satisfies ϕ—that is, $\alpha \vDash \phi$—iff, after every variable v in ϕ has been replaced by $\alpha(v)$, the formula evaluates to true under the standard integer logic model.

We can now proceed to define integer transition systems. Consider a fixed set L of locations and a set V of program variables.

Definition 2.3. A state $s : (l, \alpha)$ is a pair of a program location $l : L$ and an assignment $\alpha : \mathrm{A}_V$ on program variables V.

In order to define state transitions, we need the ability to capture variables after the transition. Thus we define v' for each variable $v \in V$. This extends

to other constructs as follows:

- $V' = \{v' \mid v \in V\}$
- $\alpha'(v') = \alpha(v)$
- ϕ' is ϕ with all occurrences of a variable v replaced by v'

Definition 2.4. A transition rule $\tau : l \xrightarrow{\phi} r$ is a triple of a source and target location l, r and a transition formula ϕ. An integer transition system P is a set of transition rules with a start location $l_0 : L$.

Definition 2.5. A transition step $\rightarrow_\tau$ for a transition rule $\tau : l \xrightarrow{\phi} r$ is a relation defined on two states so that $(l, \alpha) \rightarrow_\tau (r, \beta)$ if $\alpha \uplus \beta' \vDash \phi$. For an LTS P the transition step is just the union over all rules: $\rightarrow_P = \bigcup_{\tau \in P} \rightarrow_\tau$

Definition 2.6. An execution for a LTS P is a potentially infinite sequence of states s_i starting at l_0, i.e. $s_0 = (l_0, \alpha_0), s_1 = (l_1, \alpha_1), \cdots$ so that $s_i \rightarrow_P s_{i+1}$. An LTS is terminating iff it has no infinite execution.

2.3 The T2 temporal prover

In order to perform termination checking, we utilize the T2 temporal prover [7] in termination checking mode. T2 is a tool for termination checking using well-known techniques to show termination of ITSs. Our main goal is to automatically determine *why* programs terminate, so we use existing tools and analyze their proofs to then infer the termination arguments using the new techniques developed in this thesis.

Since we need termination certificates to extract termination arguments, we use the `cert` branch, which implements certified termination checking.

Because we want to infer the termination arguments from T2s proof of termination, we need to be able to construct an input to T2, parse the output format, and understand how T2 proves termination of ITSs.

2.3.1 Termination proving algorithm

The termination checking algorithm in T2 is currently based on Brockschmidt's cooperation approach [3]. As the actual inner workings of T2 are only moderately relevant, we will only provide a short overview here.

First, a cooperation graph C is constructed from the input program ITS P, split into two graphs: a graph for the safety prover $\textsc{Safety}(C)$, consisting of the unchanged reachable program states (i.e. a copy of the original program

graph P), and a graph for the termination prover $\text{TERMINATION}(C)$, consisting of the states for which termination has not yet been proven. These two graphs are connected at the cutpoints of the original program. Additionally, an error location is added that is reachable iff the termination argument is invalid.

Now, we only consider strongly connected components (SCC) for termination, since acyclic program graphs are trivially terminating. First, a short definition of a rank function:

Definition 2.7. Call f a T-orienting rank function if f is non-increasing for all T-transitions $l \xrightarrow{\phi} r$ such that ϕ holds, i.e. for all states α, β', if $\alpha \uplus \beta' \vDash \phi$, then $f(l, \alpha) \geq f(r, \beta')$ and there exists a decreasing transition $l_d \xrightarrow{\phi} r_d$ so that additionally $f(l_d, \alpha) > f(r_d, \beta')$. Rank functions are bounded by a minimal element.

If for an SCC S there exists a S-orienting rank function, then we remove all *decreasing* transitions from the cooperation graph C and SCC S. After this, an attempt is made to determine a counterexample, i.e. a path from the starting location to an error location. If such a counterexample is found, an attempt to determine a rank function is made. If a rank function exists that is S-orienting where S is the SCC in the termination graph relevant to the counterexample, then the decreasing locations are removed from C and the termination argument is strengthened. Otherwise, just a strengthening is performed. This is repeated until there are no more counterexamples. If at any point, there are no more rank function candidates, "unknown" is returned. Otherwise, the algorithm determined that the program is terminating.

2.3.2 Input format

T2 expects an input format reminiscent of control-flow graphs, with a `START` initial node and then edges between nodes annotated with `assume` and assignment statements. `assume` assumes that a formula is valid at this particular moment, whereas assignments either set a variable to some linear arithmetic expression or indeterminate (i.e. universally quantify it). A relevant excerpt of the grammar is provided in section 2.3.2.

The input format corresponds relatively closely to the ITS defined in section 2.2. An assignment emits an equality setting a post-variable to some expression. The conjunction of all assignments and assumptions at a particular location is the transition formula ϕ.

⟨*program*⟩ ::= `START :` ⟨*loc*⟩ `;` ⟨*blocks*⟩

⟨*blocks*⟩ ::= ⟨*block*⟩ `;` ⟨*blocks*⟩
| ϵ

⟨*block*⟩ ::= `FROM :` ⟨*loc*⟩ `;` ⟨*commands*⟩ `TO :` ⟨*loc*⟩

⟨*commands*⟩ ::= ⟨*command*⟩ `;` ⟨*commands*⟩
| ϵ

⟨*loc*⟩ ::= number | id

⟨*command*⟩ ::= id `:=` ⟨*term*⟩
| `assume (` ⟨*formula*⟩ `)`
| `assume (` ⟨*term*⟩ `)`

⟨*term*⟩ ::= number | id
| `-` ⟨*term*⟩
| `(` ⟨*term*⟩ `)`
| ⟨*term*⟩ `+` ⟨*term*⟩
| ⟨*term*⟩ `-` ⟨*term*⟩
| ⟨*term*⟩ `*` ⟨*term*⟩
| `nondet ( )`

⟨*formula*⟩ ::= ⟨*term*⟩ `<` ⟨*term*⟩
| ⟨*term*⟩ `>` ⟨*term*⟩
| ⟨*term*⟩ `<=` ⟨*term*⟩
| ⟨*term*⟩ `>=` ⟨*term*⟩
| ⟨*term*⟩ `==` ⟨*term*⟩
| ⟨*term*⟩ `!=` ⟨*term*⟩
| `!` ⟨*formula*⟩
| ⟨*formula*⟩ `&&` ⟨*formula*⟩
| ⟨*formula*⟩ `||` ⟨*formula*⟩
| `(` ⟨*formula*⟩ `)`

Figure 2.2: T2 grammar, excerpt

2.3.3 Certificates

T2 is capable of outputting proof certificates, which detail exactly how to verify a termination proof. T2 uses the ISAFoR framework, written in Isabelle, for this task. We will eventually use this certificate in order to extract invariants about the program.

A proof certificate is an XML file containing the input ITS and a proof tree detailing exactly how to reproduce a proof. We give a short overview of the relevant aspects in this section.

First, the certificate contains a description of the input transition system, with an initial location and a list of transition definitions. Each transition definition consists of a transition id, a source location, a target location, and a formula.

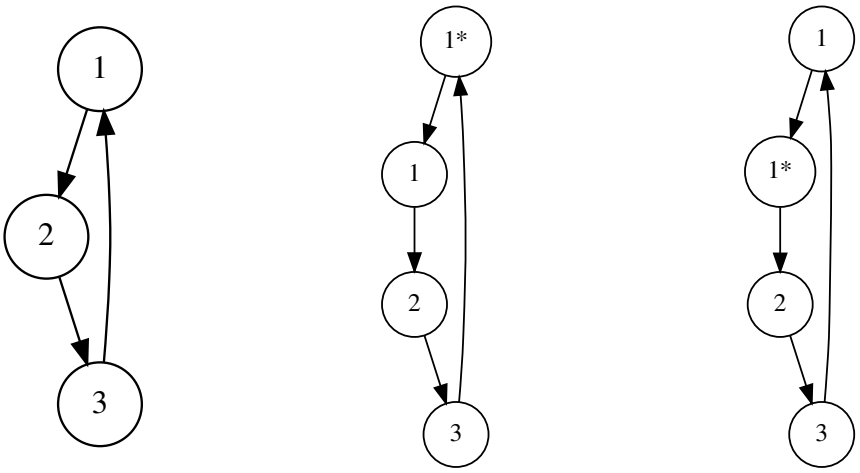
Then the cooperation proof itself consists of a list of cutpoints and a proof tree. The proof tree starts focused on the full termination graph, and each proof is one of 5 variants:

Case 1: SCC decomposition
The currently focused graph is restricted to a list of strongly connected components, and a proof tree is attached for each SCC.

Case 2: Cutpoint decomposition
The graph is decomposed into a list of partitions identified by a specific ϵ-transition for the relevant cutpoint. The proof tree continues on that partition.

Case 3: Transition removal
A list of transitions is removed from the currently focused graph. Every relevant location has a non-increasing ranking function attached, and the ranking function for the removed transitions is also strictly decreasing. Additionally, a lower bound is provided for the ranking functions. If necessary, there is also a subsequent proof tree.

Case 4: Location addition
The cooperation algorithm required an addition of a new location, given by a transition definition. Here, either the source or the target are fresh variables. This determines how to adjust the existing transitions: if the source is new, change all transitions targeting the target to target the source instead. Otherwise, change all transitions coming from the source to come from the new target instead. Finally, add the transition itself. The proof continues on the modified graph. An example for both cases is given in fig. 2.3



(a) Original graph **(b)** New source addition **(c)** New target addition

Figure 2.3: Location addition step–1^* is added in various positions

Case 5: Fresh variable addition
Some fresh variables are added as required by the algorithm. For the extraction of termination arguments, this is fairly uninteresting and we don't keep track of fresh variables. The proof continues on an unchanged graph.

Case 6: New invariants
The safety prover emitted some new invariant, and CeTA requires it to verify the proof. As we don't care about the safety prover results here, we just acquire the next proof object.

CHAPTER 3

TERMINATION ANALYSIS

In this section we describe the new methods we developed to prove termination and extract termination arguments from the proof certificate. The termination arguments consist of a set of termination-relevant variables. Our methods are based on well-known techniques for termination checking of Java programs, adapted so that sufficient information is preserved to allow relating the termination arguments from the ITS to Java variables, and ITS locations to program locations. This enables us to read the variables used in the termination proof from the certificate produced by T2.

In the first section we describe the transformation from Java to ITS in detail. The second section explains how we parse the certificate and extract the termination invariants and termination-relevant variables.

3.1 Java to ITS conversion

The input Java program is analyzed by Attestor as described in section 2.1. A Java-to-Jimple translation is performed via Soot, and then the Jimple statements and expressions are translated into an abstract semantics. We augment the semantics with a function `computeITSActions` to determine the ITS actions performed by a statement at some particular program state, and a function `asITSTerm` that returns the ITS equivalent of an expression. This translation is chosen so that variables do not get irreversibly mangled and ITS nodes correspond to Jimple lines.

To illustrate this process, we'll consider the simple program in listing 1, adapted from [8]. Its Jimple translation is visible in listing 2. It creates a 10-element linked list, then reverses it. Manual analysis shows that termination here relies on `i` for the first loop and `head` as the initial state, `current` as

the decreasing loop condition, `next` as an intermediary variable for the second loop. In the Jimple translation, this would correspond to `i` and `temp$3` in the first loop and `next`, `head`, `current`, `temp$0`, and `temp$1` in the second loop.

```
1  public class SLList {
2      private SLList next;
3
4      public static void main(String[] args){
5          SLList head = new SLList();
6          for (int i = 0; i < 10; i++) {
7              SLList next = head;
8              head = new SLList();
9              head.next = next;
10         }
11
12         SLList reversedPart = null;
13         SLList current = head;
14         while (current != null) {
15             SLList next = current.next;
16             current.next = reversedPart;
17             reversedPart = current;
18             current = next;
19         }
20     }
21 }
```

Listing 1: Java code for example program

We add an ITS generation phase after the state space generation, but before the model checking phase. Since T2 only supports linear integer logic, we need to convert our program first.

Boolean and integer values are used directly, as integer transition systems fully support those. All floating-point expressions are ignored by making them indeterminate values. Arithmetic and logic operations are converted 1:1 into the ITS framework where possible; otherwise the expression is a non-deterministic value. The translation for heap objects is similar to Westphal's [9], but simplifies some things to better suit Attestor's programming model.

Heap objects are abstracted by their size. We define an appropriate overapproximating measure in definition 3.1. The referenced object's size is tracked

```
    public static void main(java.lang.String[])
    {
        java.lang.String[] args;
        SLList head, temp$0, next, temp$1;
        SLList reversedPart, current, next;
        int i, temp$3;

        args := @parameter0: java.lang.String[];
        temp$0 = new SLList;
        specialinvoke temp$0.<SLList: void <init>()>();
        head = temp$0;
        i = 0;

     label1:
        if i < 10 goto label2;
        goto label3;

     label2:
        next = head;
        temp$1 = new SLList;
        specialinvoke temp$1.<SLList: void <init>()>();
        head = temp$1;
        temp$1.<SLList: SLList next> = next;
        temp$3 = i + 1;
        i = temp$3;
        goto label1;

     label3:
        reversedPart = null;
        current = head;

     label4:
        if current != null goto label5;
        goto label6;

     label5:
        next = current.<SLList: SLList next>;
        current.<SLList: SLList next> = reversedPart;
        reversedPart = current;
        current = next;
        goto label4;

     label6:
        return;
    }

}
```

Listing 2: Jimple conversion of listing 1

in an ITS variable. Bounds for object sizes are added at an assignment site later on.

Definition 3.1 (Object size)**.** Let $o : \text{Ref}$ be a reference to an object and $\text{Fields}(o)$ be its fields. We can then define $\|\cdot\| : \text{Ref} \to \mathbb{N}$:

$$\|o\| = \begin{cases} 1 + \sum_{o_f \in \text{Fields}(o)} \|o_f\| & \text{if } o \neq \texttt{null} \\ 0 & \text{otherwise} \end{cases}$$

Theorem 3.1 (Object size after assignments)**.** *Assuming no aliasing, sharing, or cycles, the following implications hold for a local variable x, y and a field f where x' represents x after the execution of the assignment and x before.*

$$\begin{aligned} x := y &\Rightarrow \|x'\| = \|y\| \\ x := y.f &\Rightarrow \|x'\| \geq 0 \wedge \|x'\| \leq \|y\| - 1 \\ x.f := y &\Rightarrow \|x'\| > \|y\| \wedge \|x'\| \leq \|x\| + \|y\| \end{aligned}$$

Proof. **Case 1:** $x := y \Rightarrow \|x'\| = \|y\|$
Trivial by reflexivity.

Case 2: $x := y.f \Rightarrow \|x'\| \geq 0 \wedge \|x'\| \leq \|y\| - 1$
$\|x'\| \geq 0$ holds by induction on the construction of objects. $\|x'\| \leq \|y\| - 1$ follows since the object referenced by y itself increases the size by at least 1, so the size of a field is at most the size of y reduced by 1.

Case 3: $x.f := y \Rightarrow \|x'\| > \|y\| \wedge \|x'\| \leq \|x\| + \|y\|$
By the same argument as in case 2, we know that $\|x.f\| \in [0, \|x\| - 1]$. When we perform an assignment to a field—given no sharing or cycles—we have $\|x'\| = \|x\| - \|x.f\| + \|y\|$ since the size of $\|x\|$ is first reduced by the size of $\|x.f\|$, then increased by $\|y\|$. This gives us a lower bound at $\|x.f\| = \|x\| - 1$: $\|x'\| \geq 1 + \|y\|$, or $\|x'\| > \|y\|$.

Similarly, we have an upper bound at $\|x.f\| = 0$, in which case $\|x'\| \leq \|x\| + \|y\|$. □

Finally, we can define `computeITSActions` for the translation of statements into ITS steps. Here, $\text{pc}(s)$ refers to the program counter at state s:

Case 1: `AssignInvoke/InvokeStmt`
As we restrict our scope to single-method programs, these statements are mostly skipped. Given a source state s and target state t, `AssignInvoke[local = someMethod(arg_1, ...)]` is modelled as

```
FROM pc(s);
local := nondet();
// [if local is object type]
assume(local >= 0);
// [for all arg_i if arg_i is object type]
arg_i := nondet();
assume(arg_i >= 0);
TO pc(t);
```

For `InvokeStmt`, it is the same, but without `local`.

Case 2: `AssignStmt`
Assume s is the source state and t is the target state. If we perform an assignment to a left-hand side (lhs) with a primitive type, we convert the right-hand side into an ITS term and assign it directly:

```
FROM pc(s);
lhs := rhs;
TO pc(t);
```

If the rhs is an array length expression, we add an assumption that `lhs` ≥ 0 since array length expressions are modelled as indeterminate:

```
FROM pc(s);
lhs := nondet();
assume(lhs >= 0);
TO pc(t);
```

If the lhs is an object variable, we perform a size approximation as described in definition 3.1:

- $x := y$
  We simply assign in this case.
  ```
  FROM pc(s);
  x := y;
  TO pc(t);
  ```
- $x := y.f$
  We don't know the exact value of the lhs x anymore, so it is indeterminate. But we can use the approximations based on the rhs y:

```
FROM pc(s);
x := nondet();
assume(x >= 0);
assume(x <= y - 1);
TO pc(t);
```

- $x.f := y$
  Similar to above, x is no longer determinate. However, the overapproximating bounds are different. We use bounds on the field access since we don't know how big the field is. Additionally, for local variables $x_1, \cdots, x_n$ aliasing x we set them to x afterwards. Aliasing detection is performed via the Attestor heap configuration at s. We consider variables an alias if they point to the same heap node. To determine the set of aliases, we look up the configuration node of x and enumerate all variables that point to the same node.

```
FROM pc(s);
x.f := nondet();
assume(x.f >= 0);
assume(x.f < x);
x := x - x.f + y;
x_1 := x;
...
x_n := x;
TO pc(t);
```

Case 3: `IfStmt`

An `IfStmt` in state s with condition b and jumps to t if true, f otherwise is translated into:

```
FROM pc(s);
assume(b);
TO pc(t);

FROM pc(s);
assume(!b);
TO pc(f);
```

where b is translated via `asITSTerm`

Case 4: `GotoStmt`
A `GotoStmt` from *s* to *t* is a `FROM pc(s); TO pc(t);` entry.

Case 5: `IdentityStmt`
As undefined variables are automatically assumed indeterminate, we ignore this statement and behave like `goto`

Case 6: `ReturnValueStmt/ReturnVoidStmt`
As we are restricted to single method programs, we ignore these statements like above.

Expressions are handled by `asITSTerm`:

Case 1: `Local`
Local variables are converted 1:1 into ITS variables. The names are mangled so that they are valid ITS identifiers: `$` is replaced by `__`.

Case 2: `Field`
Field values are translated into `nondet()`, since we don't keep track of structural information.

Case 3: `NewExpr`
`new` is translated to the constant `1`.

Case 4: `LengthExpr`
`array.length` is translated into `nondet()`, since we don't know the exact length of the array.

Case 5: `ArithExpr`
Binary arithmetic expressions are used one-to-one in a corresponding arithmetic ITS expression, recursing over the left and right operand.

Case 6: `CompareExpr`
Comparisons are converted analogously to `ArithExpr`, except with comparison operators.

Case 7: `AndExpr/OrExpr/ EqualExpr/UnequalExpr/NotExpr`
Boolean expressions are also translated one-to-one.

Case 8: `IntConstant/NullConstant`
Integer constants emit an integer constant, and null constants emit 0.

Case 9: `UndefinedValue`
Undefined values are `nondet()`.

We recursively apply `computeITSActions` to all program states reachable in the state space graph from the initial states, and add a separate empty step from a start node to the initial states and from the final states to an end node for easier bookkeeping.

This results in an ITS that can be represented by the graph in fig. 3.1. We start at the node labelled `start` and terminate at `end`. Each round node represents a program location, and each square node contains the actions executed during the transition. Notably, we can see the two strongly connected components between node 7 and node 16 and node 19 and node 25.

We can then execute T2 on the ITS. If it was found terminating, a certificate was generated and we move on to the next step of our analysis. Otherwise, we abort.

3.2 Certificate parsing

In the previous section we designed our translation in such a way that the LTS terminates if the original Java program terminates. This means that the reasoning behind the termination proof is also transferable to the Java program. As described in section 2.3, the important step in the termination proving algorithm of T2 is finding rank functions. A rank function on an ITS indicates that a transition can only be taken a finite number of times, since it is bounded and will eventually reach that bound. This logic can also be applied to the original program. A transition from program location a to program location b can only be executed a finite number of times since the rank function would eventually reach its bound.

In this section, we describe how we analyze the proof certificate. Since we preserved variable names in our transformation earlier, determining the Java variables used in the termination proof is comparatively straightforward.

First, we extract the LTS from the certificate, represented in fig. 3.2, to acquire the transitions and their IDs—this is necessary, as transitions are referred to by their IDs in the proofs. Once we have those, we start analyzing from the beginning of the cooperation proof in order to determine a list of termination-relevant invariants. The certificate contains a lot of information that is not directly relevant to our analysis—the actual wanted data is in the transition removal steps. However, context is important, so we need to work our way down the proof tree. The actions for each proof object are:

Case 1: SCC decomposition (SCC)
For a SCC decomposition, we focus the current graph on the SCC and recurse

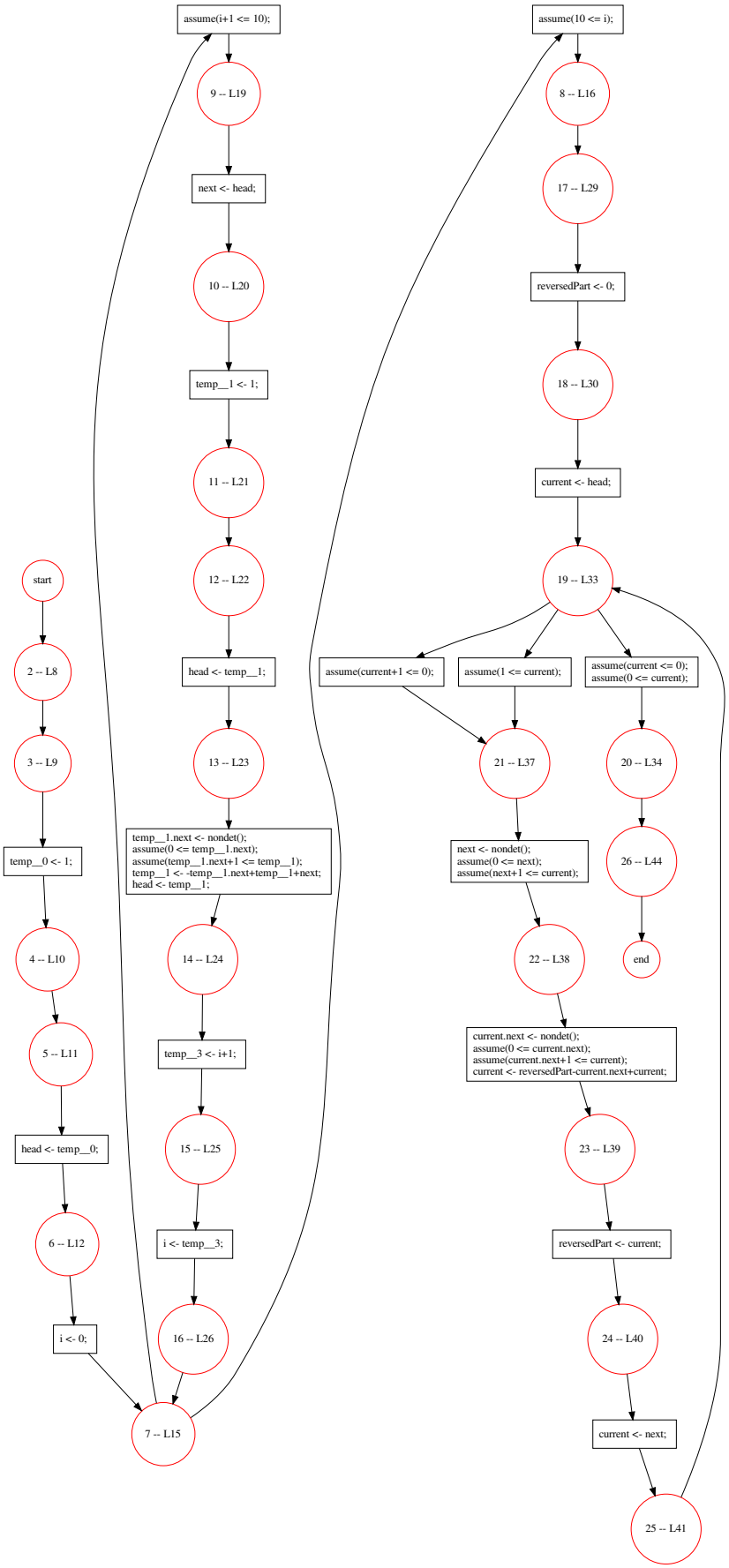


Figure 3.1: ITS generated from listing 1

on the attached proof, adding the results to the list.

Case 2: Cutpoint decomposition (Cut)
For a cutpoint split, we keep the graph focus unchanged and recurse.

Case 3: Location addition (LA)
We update the graph as described above and continue extraction with the next proof.

Case 4: New invariants (NI)/Fresh variable addition (FVA)
Nothing is changed and we step to the next proof.

Case 5: Transition removal (TR)
Here, we consider all transitions in the currently focused graph. Each location has an associated rank function, so we identify a rank function for the source and target of the transition. If the transition is removed, we note that the rank function is strictly decreasing, otherwise it's non-increasing. We construct an invariant of the form $\mathrm{RF}(s) > \mathrm{RF}(t)$ or $\mathrm{RF}(s) \geq \mathrm{RF}(t)$ depending whether it's decreasing or non-increasing, and add it to the list. Then we recurse onto the next proof.

Once all proofs are analyzed, we have a list of invariants; specifically, a list of invariants that is valid for a specific subgraph. The transition removal in (1) in fig. 3.2 contains mostly constants and gets rid of most non-cyclic transitions. (2) and (3) contain useful termination arguments. The actual certificate output for (2) and (3) actually consists of two subsequent transition removals (2.1)/(2.2) and (3.1)/(3.2). We omit (3.2) since it doesn't add anything useful to our leading example. Dashed lines are transitions that are removed from the termination graph of the ITS.

Our current primary method of analysis involves extracting the variables involved in each invariant. This leads to a termination relevant set for our leading example of `next`, `head`, `current`, `temp$0`, `temp$1`, `temp$3`, and `i`. Notably, termination-irrelevant variables like `reversedPart` are ignored. This is an exact match to our manual analysis from the beginning of the chapter. Overall just considering the variables is a fairly coarse approach—additional approaches are discussed in chapter 7.

$$(1)\ \dfrac{\mathrm{RF}_P \text{ valid with } Q, b \qquad \dfrac{\dfrac{\mathrm{RF}_{S_1} \text{ valid with } Q_1, b_1 \quad \overline{S_1 \setminus Q_1 \text{ trivial}}}{\mathrm{RF}_{S_1}, Q_1, b_1 \vdash S_1 \setminus Q_1 \text{ term.}}\,(\mathrm{TR}) \quad (2)\ \dfrac{\mathrm{RF}_{S_2} \text{ valid with } Q_2, b_2 \quad \overline{S_2 \setminus Q_2 \text{ trivial}}}{\mathrm{RF}_{S_2}, Q_2, b_2 \vdash S_2 \setminus Q_2 \text{ term.}}\,(\mathrm{TR}) \quad (3)}{S_1, S_2 \in \mathrm{SCC}(P \setminus Q \cup Q_{\mathrm{skip}}) \text{ term.}}\,(\mathrm{SCC})}{P \setminus Q \cup Q_{\mathrm{skip}} \text{ term.}}\,(\mathrm{LA})}{\mathrm{RF}_P, Q, b \vdash P \setminus Q \text{ term.}}\,(\mathrm{TR})}{\text{P term.}}$$

Figure 3.2: Certificate sketch for the termination of the ITS P in fig. 3.1

We consider a set of rank functions RF_P valid with LTS P, removed transitions Q and bound b if $\forall l \xrightarrow{\phi} r \in P,\ \phi \implies \mathrm{RF}_P(l) \geq \mathrm{RF}_P(r) \wedge \mathrm{RF}_P(l) \geq b$ and $\forall l \xrightarrow{\phi} r \in Q,\ \phi \implies \mathrm{RF}_P(l) > \mathrm{RF}_P(r)$ (Theorem 8 in [6])

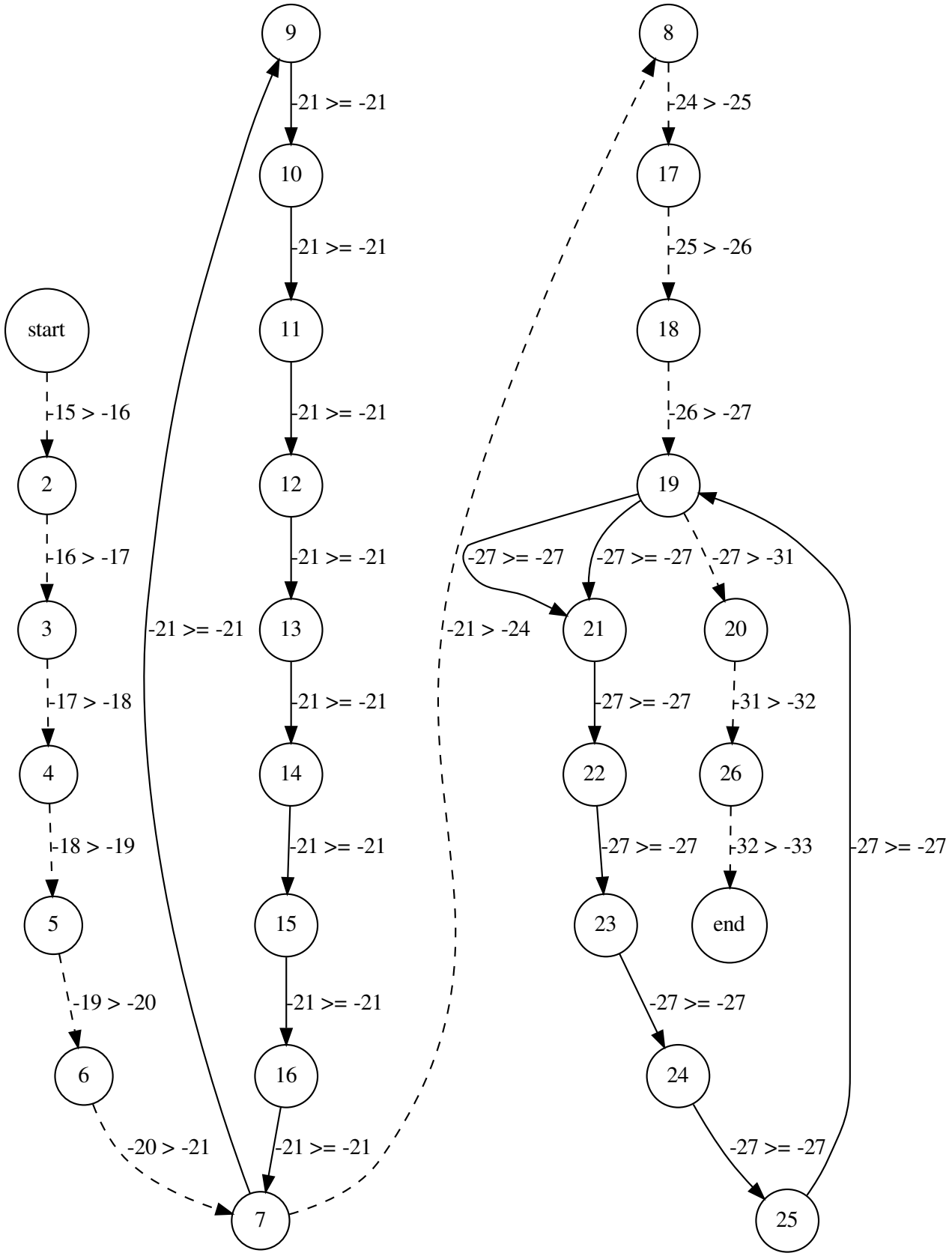
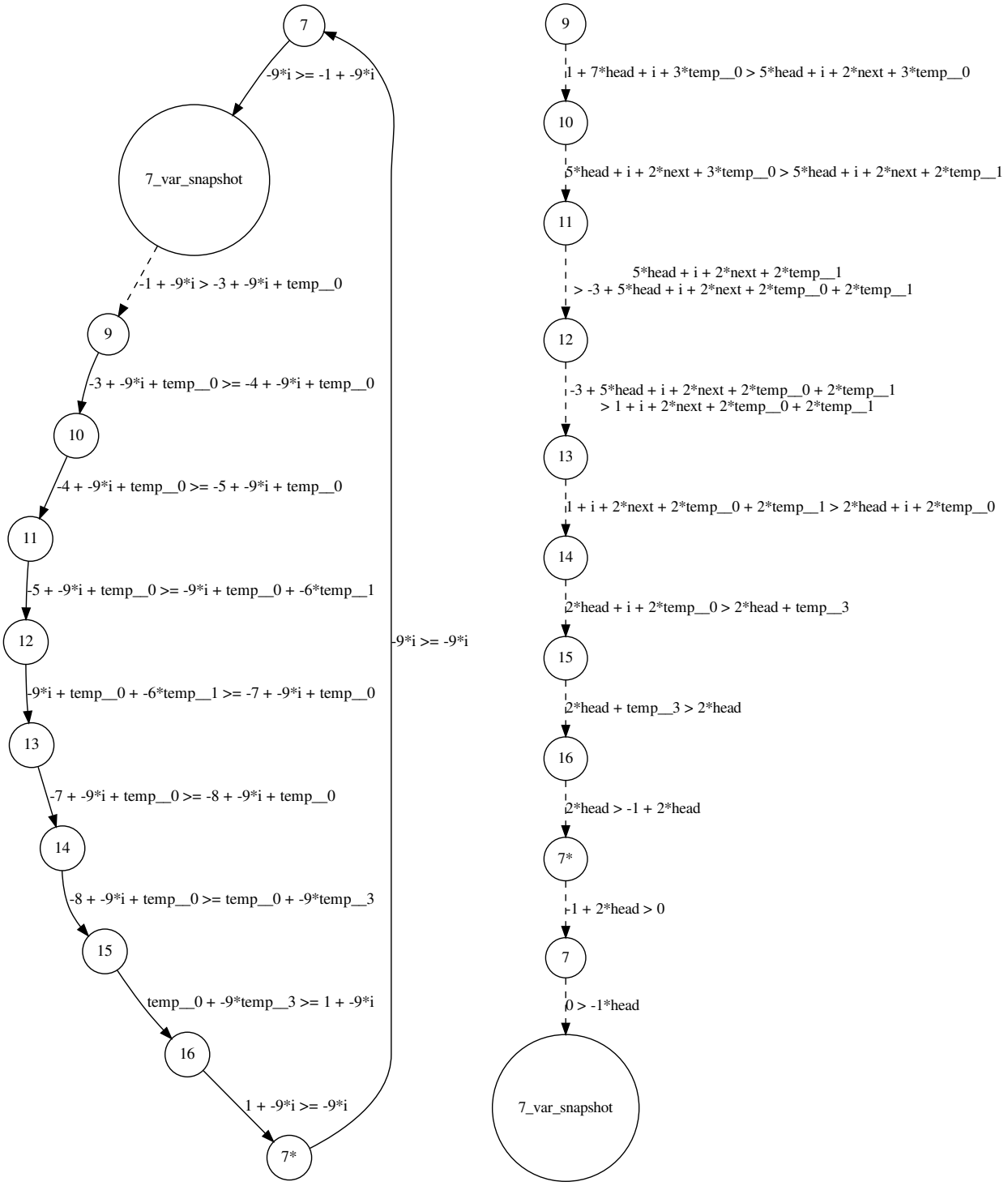


Figure 3.3: Transition removal for (1)



(a) (2.1) **(b)** (2.2)

Figure 3.4: Transition removal for (2)

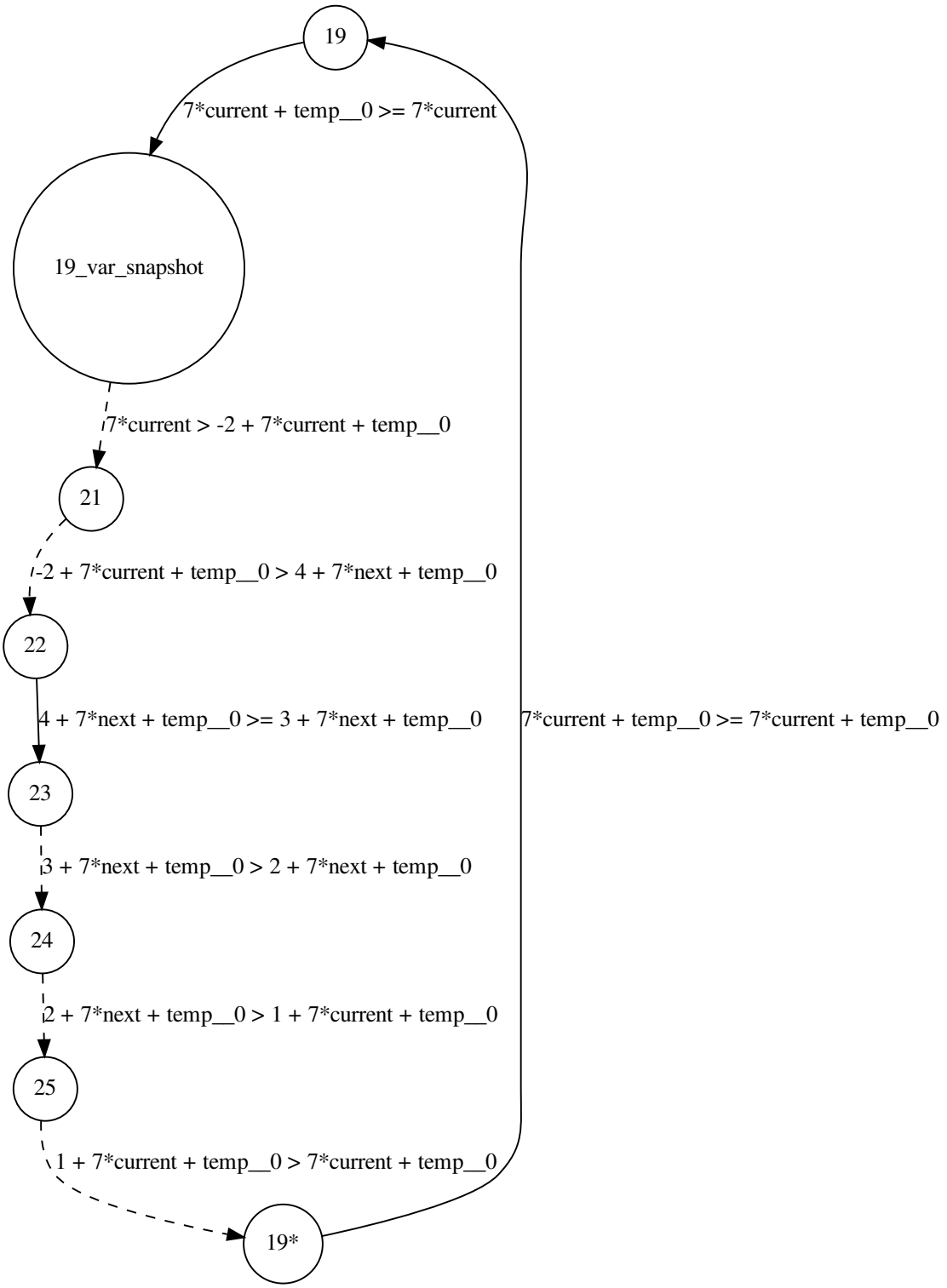


Figure 3.5: Transition removal for (3.1)

CHAPTER 4

CORRECTNESS

Since we don't have a formal semantics available for the full transformation stack, the amount of correctness we can prove is rather limited. The assumptions for this chapter are given in definition 4.1.

Definition 4.1 (Assumptions)**.** For the purposes of this chapter, we assume the following statements:

1. Java semantics are preserved during the transformation into Jimple
2. Jimple semantics are overapproximated when translated into Attestor abstract semantics
3. A recursive search from the initial Attestor states will reveal all potentially reachable states
4. All aliasing that occurs is detected by Attestor and stored in the heap configuration before an assignment to a potentially aliased local variable, and no sharing or cycles occur
5. No overflow or underflow is possible in any integer variable (i.e. they can be modelled by $\mathbb{Z}$)
6. All Jimple statements in the program are modelled by the abstract semantics
7. All invoked methods terminate, and the Jimple semantics treat method calls as a single step

Definition 4.2 (State-to-assignment conversion)**.** Given a Jimple program state s, the corresponding ITS assignment $[\![s]\!]$ is defined by:

$$[\![s]\!](v) = \begin{cases} \alpha_s(v) & \text{if } \alpha_s(v) \in \mathbb{Z} \\ \|\alpha_s(v)\| & \text{if } \alpha_s(v) \in \text{Ref} \end{cases}$$

The extension of $[\![s]\!]$ to arbitrary Jimple expressions is done according to standard Jimple semantics, e.g. $[\![s]\!](\texttt{x + y}) = [\![s]\!](x) + [\![s]\!](y)$

Theorem 4.1 (`asITSTerm` correctness)**.** *Given a Jimple expression x and its `asITSTerm` conversion x', for all program states s so that $[\![s]\!](x)$ is defined, $[\![s]\!] \vDash x'$.*

Proof. Under the above assumptions, this follows fairly automatically from the definition of `asITSTerm`. Since expressions that are not explicitly converted are `nondet()`, $[\![s]\!]$ still satisfies `asITSTerm`. □

Theorem 4.2 (Small-step translation)**.** *Given two concrete program states s, t and Jimple statements C_s, C_t so that $\langle C_s, s\rangle \to \langle C_t, t\rangle$, two abstract states s' and t' overapproximating s and t and the abstract semantics statement C'_s, C'_t corresponding to C_s and C_t respectively, then there exists an ITS step for the ITS rule $\tau : \text{pc}(C_s) \xrightarrow{\phi} \text{pc}(C_t)$ generated by `computeITSActions` on s', C' and t' so that $(\text{pc}(C_s), [\![s]\!]) \to_\tau (\text{pc}(C_t), [\![t]\!])$.*

Proof. By assumption 2, we can safely perform case analysis on C'_s instead of C_s.

Case 1: `AssignInvoke`: `local = someMethod(arg_1, ...);`
By assumption 7, all invoked methods terminate. The only locally reachable affected program state is the value of `local` and the fields of all references passed via an `arg_i`. Since we set those to `nondet()` in the ITS translation, their value is unrestricted and there exists an ITS step for any concrete program state s, t.

Case 2: `InvokeStmt`: `someMethod(arg_1, ...);`
Same as `AssignInvoke`, but without a local variable assignment.

Case 3: `AssignStmt`: `x = y;`
For an assignment whose operands have primitive type, the existence of an ITS step follows directly from theorem 4.1. Similarly for array lengths, since array lengths are always greater or equal 0 by the standard Jimple semantics.

For an assignment of object type operands, the expression for x is accurately approximated by theorem 3.1. Assuming that our aliasing information is cor-

rect, the $x.f = y$ case also correctly updates any possibly aliased variables. Thus there exists an ITS step.

Case 4: `IfStmt: if (b) goto label2;`
Jimple if statements are side-effect free. If b evaluates to `true` under the standard Jimple semantics, then it does under the `asITSTerm` conversion as well, by theorem 4.1. Thus there exists an ITS step from $\text{pc}(s)$ to $\text{pc}(label2)$ if b evaluates to `true`, and to $\text{pc}(t)$ otherwise.

Case 5: `IdentityStmt: local = @this; ...`
Any variables that are not explicitly set are the same as indeterminate variables, so there exists an ITS step for any concrete program state.

Case 6: `GotoStmt`
There exists an ITS step trivially for any program state as a result of the ITS rule not affecting any variables.

Case 7: `ReturnValueStmt, ReturnVoidStmt`
Since we do not analyze invoked methods, return statements are necessarily terminal statements. Thus a Jimple semantics step with a return value as the origin to another Jimple statement cannot occur, and this case is impossible.

□

Theorem 4.3 (Big-step translation)**.** *Let P be the ITS generated by our algorithm for a Jimple program. If $\langle C_0, s_0\rangle \to \langle C_1, s_1\rangle \to \cdots$ is a valid execution of the Jimple program, then there exists a valid execution of P so that $(\text{pc}(C_0), [\![s_0]\!]), (\text{pc}(C_1), [\![s_1]\!]), \cdots$.*

Proof. Assume there doesn't exist a valid execution. Then there is an execution step in the Jimple program whose translation is not a valid ITS step, which is contradicted by theorem 4.2 under the assumptions in definition 4.1. All relevant ITS rules are generated under assumption 3. □

Theorem 4.4 (Soundness of termination analysis)**.** *If the ITS translation P of a Jimple program terminates, so does the Jimple program.*

Proof. We consider an ITS terminating if there exists no infinite execution $(l_0, \alpha_0), (l_1, \alpha_1), \cdots$. Similarly, we consider a Jimple program terminating if there exists no infinite execution $\langle C_0, s_0\rangle \to \langle C_1, s_1\rangle \to \cdots$.

Assume there exists an infinite Jimple program execution. Then by theorem 4.3, there exists an infinite execution of the corresponding ITS P. This is contradicted by the hypothesis that P is terminating. □

Now that we have proven soundness of the analysis itself, we will prove that irrelevant variables do not affect termination.

Theorem 4.5 (Irrelevancy)**.** *Given a terminating ITS P and a set of irrelevant variables V extracted from the termination certificate of P, a transformation of P with all writes to variables from V removed also terminates.*

Proof. Removing all writes to a variable is equivalent to setting it to `nondet()` throughout. Since the only step in the certificate that removes transitions (and thus progresses the termination proof) is the transition removal step, we verify that the rank functions are still valid.

Consider a transition removal step with current graph P, rank function f, bound b and removed transitions $Q \subseteq P$. According to theorem 8 in [6], a transition removal step is valid if for a transition $l \xrightarrow{\phi} r \in Q$, $\phi \implies f(l) > f(r)$, and for a transition $l \xrightarrow{\phi} r \in P$, $\phi \implies f(l) \geq f(r) \wedge f(l) \geq b$. Since the removed variables by definition do not occur in $f(l)$ or $f(r)$, the implication still holds regardless of the value of the variables. Thus the transition removal step is still valid.[1] □

[1] More formally, all invariants occurring in the certificate might have to be adjusted, but that's an implementation detail and does not actually affect the relevant steps.

CHAPTER 5

Results and evaluation

Evaluating the results of our system is rather difficult empirically. We ran our modified version of Attestor on the supplied examples [8] and collected the total runtime, ITS runtime (including generation and T2 runtime), T2 output (TERMINATING, MAYBE or NONTERMINATING) and amount of relevant and irrelevant termination variables. The results are presented in table 5.1 for the T2 results and table 5.2 resp. fig. 5.1 for the relevancy analysis.

Total	TERMINATING	MAYBE	NONTERMINATING	ERROR
160	124	2	19	15

Table 5.1: Results—termination

Amount relevant	Amount irrelevant	Mean relevant	Mean irrelevant
149	366	1.20	2.95

Table 5.2: Results—relevance

A few notes on the results—since we do not analyze methods in this work, all possibly affected locations are overabstracted. These cases are potentially considered nonterminating as that only requires a counterexample to exist, not all branches to be nonterminating. Additionally, no attempt was made to filter out programs violating the assumptions of single-method, no sharing, and no cycles, as this is more of a quantitative comparison.

In fig. 5.1 there is an interesting trend to a large amount of irrelevant variables. Many Attestor examples supply an initial heap configuration, which potentially simplifies the heap enough that the resulting state space no longer contains a loop. Additionally, some `.attestor` files run on the same method, just with different Attestor parameters. The $(0,0)$ results are partially caused due to some of the settings files not generating a proper state space graph.

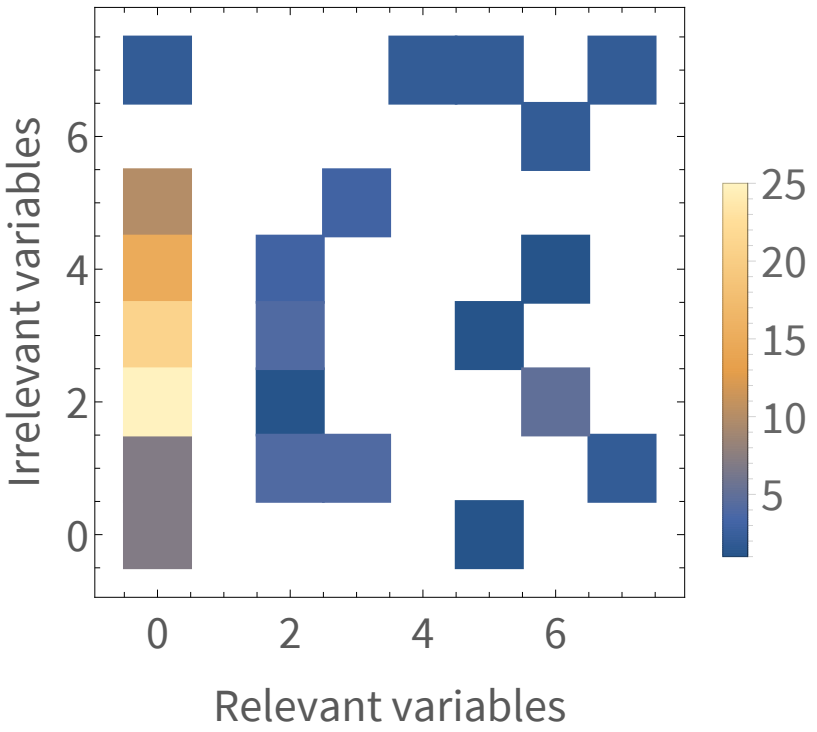


Figure 5.1: Density histogram of the relevancy analysis

Integer programs are fairly well supported. A program computing a factorial via a loop that also sums up the result with each iteration has both termination relevant variables correctly identified, with six irrelevant ones. Another test factorial program that writes its result into a linked list with each step has two termination relevant variables correctly identified and two superfluous ones, with four correctly identified irrelevant variables.

As for the error cases, those ran into a bug with T2. During the course of this thesis, we already fixed one bug, but we weren't able to find a fix for the second issue.

CHAPTER 6

Related work

All automated termination checking tools generate termination arguments in some form or another. However, many of them are not easily back-referencable to the original code. For example, AProVE [1] performs termination and complexity checking via TRS and can output an ITS for usage in T2. However, variables are sufficiently different that relating them to the original code is not easily possible.

Overall, our approach is similar to Westphal's [9] in that the ITS only includes size information for objects and assumes no sharing or cycles otherwise. However, this is done under the framework of Attestor, allowing for the resulting invariants to be used for e.g. model checking or relaxed structural analysis. Westphal's approach starts directly from Soot and does not include a heap shape analysis. This makes using structural data much more difficult.

Systems like ISaFoR/CeTA [10] use the certificate for its intended purpose of certifying a termination check. However, to the best of our knowledge, our approach of analyzing a termination certificate for further information is completely novel.

CHAPTER 7

Future work

The analysis is restricted to single method integer and heap programs with no sharing or cycles for any termination-relevant heap object.

An extension to multi-method nonrecursive programs is relatively easy by inlining the relevant methods, but involves merging state spaces in Attestor which is rather complicated. Attempts via method summaries similar to Frohn's work [11] failed due to T2 not outputting useful postconditions. If T2 were to be augmented with a way to determine postconditions, a scalable extension to multi-method programs is possible. Depending on the formulation of method summaries, even recursion might be modelable.

Sharing and cycles can be accounted for by using information from Attestor's structural analysis, like we already do for aliasing detection. However, it requires structuring the ITS transitions based on the state space identifiers and not program counters. This will hinder termination argument extraction for object variables—past formulations of this project did exactly that. In short, if using state space identifiers as edges, the loop over `current` in the example program will actually convert into an acyclic sequence and the termination relevant variables would no longer contain actually relevant variables. By using program counters, we achieve a compromise between result quality and structural analysis.

Currently, we perform only a very simple analysis on the extracted invariants—we extract the occurring variables. Future approaches could use the associated graph structure and transition pre/postconditions to further refine the results. We also evaluate the relevancy set for the whole program. It should be sound to evaluate it for a particular SCC (which usually corresponds to a loop), allowing for more fine-grained relevancy. Consider fig. 3.4—it has a relevant set of `i`, `temp$0` and `temp$3` in the cyclic part (2.1) and additionally `next` in the

non-cyclic part (2.2), but only `i` and `temp$3` is truly relevant there.

Sometimes T2 considers a termination-irrelevant variable in its rank functions. For example, the `SLList` variables in fig. 3.4 are not relevant to the termination of that SCC. There is potential work to be done to make T2 attempt to minimize the occurring variables. This could possibly also be done in a preprocessing step after parsing the certificate.

An original goal of this project was to also account for non-termination, especially since that would be useful for software developers as e.g. an IDE warning. However, T2 currently does not have a certified non-termination prover, and adding support for T2 to output the relevant data did not fit into the thesis timeframe. In addition, sound nontermination analysis requires a complete, not just sound, Java to ITS transformation.

CHAPTER 8

CONCLUSION

We developed a novel, integrated termination analysis that is capable of extracting termination-relevant invariants and variables from a Java program. By transforming a Java program into an integer transition system after structural analysis via Attestor, we can determine termination for integer programs and a subset of heap programs. Future Attestor extensions are now capable of utilizing termination checking and invariants resulting from it.

Evaluation showed a decent capability of detecting termination-relevant variables, consistent with manual evaluation on test programs. Almost all programs contained at least one irrelevant variable. Finally, we provided key points on further improving the analysis: for example, lifting the restriction on single-method programs or generating method summaries.

Bibliography

[1] Dirk Beyer. Automatic verification of C and Java programs: Sv-comp 2019. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 133–155. Springer, 2019.

[2] Hannah Arndt, Christina Jansen, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. Let this graph be your witness! In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 3–11, Cham, 2018. Springer International Publishing.

[3] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *International Conference on Computer Aided Verification*, pages 413–429. Springer, 2013.

[4] MOVES RWTH Team. Attestor. `https://github.com/moves-rwth/attestor`, 2019.

[5] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[6] Marc Brockschmidt, Sebastiaan J.C. Joosten, Rene Thiemann, and Akihisa Yamada. Certifying safety and termination proofs for integer transition systems. In *Proceedings of CADE'17*. Springer, June 2017.

[7] T2 Contributors. T2 temporal logic prover. `https://github.com/KaneTW/T2/tree/cert`, 2020.

[8] MOVES RWTH Team. Attestor examples. `https://github.com/moves-rwth/attestor-examples`, 2019.

[9] Oliver Westphal. Approximative transformation of Java programs for automated termination analysis. Master's thesis, RWTH Aachen, 2017.

[10] René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In *International Conference on Theorem Proving in Higher Order Logics*, pages 452–468. Springer, 2009.

[11] Florian Frohn and Jürgen Giesl. Modular heap shape analysis for Java programs. 2017.