Lower-Bound Synthesis Using Loop Specialization and Max-SMT

Elvira Albert^{1,2}, Samir Genaim^{1,2}, Enrique Martin-Martin¹,
Alicia Merayo¹, and Albert Rubio^{1,2}

¹ Fac. Informática, Complutense University of Madrid, Madrid, Spain
amerayo@ucm.es

² Instituto de Tecnología del Conocimiento, Madrid, Spain

Abstract. This paper presents a new framework to synthesize lower-bounds on the worst-case cost for non-deterministic integer loops. As in previous approaches, the analysis searches for a *metering function* that under-approximates the number of loop iterations. The key novelty of our framework is the *specialization* of loops, which is achieved by restricting their enabled transitions to a subset of the inputs combined with the narrowing of their transition scopes. Specialization allows us to find metering functions for complex loops that could not be handled before or be more precise than previous approaches. Technically, it is performed (1) by using quasi-invariants while searching for the metering function, (2) by strengthening the loop guards, and (3) by narrowing the space of non-deterministic choices. We also propose a Max-SMT encoding that takes advantage of the use of soft constraints to force the solver look for more accurate solutions. We show our accuracy gains on benchmarks extracted from the 2020 Termination and Complexity Competition by comparing our results to those obtained by the LoAT system.

1 Introduction

One of the most important problems in program analysis is to automatically –and accurately– bound the cost of program’s executions. The first automated analysis was developed in the 70s [24] for a strict functional language and, since then, a plethora of techniques has been introduced to handle the peculiarities of the different programming languages (see, e.g., for Integer programs [5], for Java-like languages [2, 19], for concurrent and distributed languages [16], for probabilistic programs [15, 18], etc.) and to increase their accuracy (see, e.g., [10, 14, 21, 22]). The vast majority of these techniques have focused on inferring *upper bounds* on the worst-case cost, since having the assurance that none execution of the program will exceed the inferred amount of resources (e.g., time, memory, etc.) has crucial applications in safety-critical contexts. On the other hand, *lower bounds*

This work was funded partially by the Spanish MCIU, AEI and FEDER (EU) project RTI2018-094403-B-C31, by the CM project S2018/TCS-4314 co-funded by EIE Funds of the European Union, and by the UCM CT42/18-CT43/18 grant.

© The Author(s) 2021

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 863–886, 2021.

https://doi.org/10.1007/978-3-030-81688-9_40

on the best-case cost characterize the minimal cost of any program execution and are useful in task parallelization (see, e.g., [3,9,10]). There are a third type of important bounds which are the focus of this work: *lower bounds on the worst-case cost*, they bound the worst-case cost from below. Their main application is that, together with the upper bounds on worst-case, allow us to infer tighter worst-case cost bounds (when they coincide ensuring that the inferred cost is exact) what can be crucial in safety-critical contexts. Besides, lower bounds on the worst-case cost will give us families of inputs that lead to an expensive cost, what could be used to detect performance bugs. In what follows, we use the acronyms LB^w and LB^b to refer to *worst-case* and *best-case* lower-bounds, resp.

State-of-the-Art in LB^w . An important difference between LB^w and LB^b is that, while the best-case must consider *all* program runs, LB^w holds for (usually infinite) families of the most expensive program executions. This is why the techniques applicable to LB^b inference (e.g., [3,9,10]) are not useful for LB^w in general, since they would provide too inaccurate (low) results. The state-of-the-art in LB^w inference is [12,13] (implemented in the LoAT system) which introduces a variation of ranking functions, called *metering functions*, to underestimate the number of iterations of *simple* loops, i.e., loops without branching nor nested loops. The core of this method is a simplification technique that allows treating general loops (with branchings and nested loops) by using the so-called *acceleration*: that replaces a transition representing one loop iteration by another rule that collects the effect of applying several consecutive loop iterations using the original rule. Asymptotic lower bounds are then deduced from the resulting simplified programs using a special-purpose calculus and an SMT encoding.

Motivation. Our work is motivated by the limitation of state-of-the-art methods when, by treating each simple loop separately, a LB^w bound cannot be found or it is too imprecise. For example, consider the interleaved loop in Fig. 1, that is a simplification of the benchmark `SimpleMultiple.koat` from the Termination and Complexity competition. Its *transition system* appears to the right (the transition system is like a control-flow graph (CFG) in which the transitions τ are labeled with the applicability conditions and with the updates for the variables, primed variables denote the updated values). In every iteration x or y can decrease by one, and these behaviors can interleave. The worst case is obtained for instance when x is decreased to 0 (x iterations) and then y is decreased to 0 (y iterations), resulting in $x + y$ iterations, or when y is first decreased to 1 and then x to -1 , etc. The approach in [12,13] accelerates independently both τ_1 and τ_4 , resulting in accelerated versions $\tau_1^a = x \geq -1 \wedge y > 0 \wedge x' = -1 \wedge y' = y$ with cost $x + 1$ and $\tau_4^a = x \geq 0 \wedge y \geq 0 \wedge x' = x \wedge y' = 0$ with cost y . Applying one accelerated version results in that the other accelerated version cannot be applied because of the final values of the variables. Thus, the overall knowledge extracted from the loop is that it can iterate $x + 1$ or y times, whereas the precise LB^w is $x + y$ iterations. Our challenge for inferring more precise LB^w is to devise a method that can handle all loop transitions simultaneously, as disconnecting them leads to a semantics loss that cannot be recovered by acceleration.

Fig. 1. Interleaved loop (left) and its representation as a transition system (right)

Non-Termination and LB^w . Our work is inspired by [17], which introduces the powerful concept of *quasi-invariant* to find witnesses for non-termination. A quasi-invariant is an invariant which does not necessarily hold on initialization, and can be found as in template-based verification [23]. Intuitively, when there is a loop in the program that can be mapped to a quasi-invariant that forbids executing any of the outgoing transitions of the loop, then the program is non-terminating. This paper leverages such powerful use of quasi-invariants and Max-SMT in non-termination analysis to the more difficult problem of LB^w inference. Non-termination and LB^w are indeed related properties: in both cases we need to find witnesses, resp., for non-terminating the loop and for executing at least a certain number of iterations. For LB^w , we additionally need to provide such under-estimation for the number of iterations and search for LB^w behaviors that occur for a class of inputs rather than for a single input instantiation (since the LB^w for a single input is a concrete (i.e., constant) cost, rather than a parametric LB^w function as we are searching for). Instead, for non-termination, it is enough to find a non-terminating input instantiation.

Our Approach. A fundamental idea of our approach is to *specialize* loops in order to guide the search of the metering functions of complex loops, avoiding the inaccuracy introduced by disconnecting them into simple loops. To this purpose, we propose specializing loops by combining the addition of constraints to their transitions with the restriction of the valid states by means of quasi-invariants. For instance, for the loop in Fig. 1, our approach automatically narrows τ_1 by adding $x > 0$ (so that x is decreased until $x = 0$) and τ_4 by adding $x \leq 0$ (so that τ_4 can only be applied when $x = 0$). This specialized loop has lost many of the possible interleavings of the original loop but keeps the worst case execution of $x + y$ iterations. These specialized guards do not guarantee that the loop executes $x + y$ iterations in every possible state, as the loop will finish immediately for $x < 0$ or $y \leq 0$, thus our approach also infers the quasi-invariant $x \geq 0 \wedge x \leq y$. Combining the specialized guards and the quasi-invariant, we can assure that when reaching the loop in a valid state according to the quasi-invariant, $x + y$ is a lower bound on the number of iterations of the loop, i.e., its cost. Using quasi-invariants that include all (invariant) inequalities syntactically appearing

in loop transitions might work for the case of loops with single path. However, for the general case, the specialized guards usually lead to essential quasi-invariants that do not appear in the original loop. The specialization achieved by adding constraints could be also applied in the context of non-termination to increase the accuracy of [17], as only quasi-invariants were used. Therefore, we argue that our work avoids the precision loss caused by the simplification in [12, 13] and, besides, introduces a loop specialization technique that can also be applied to gain precision in non-termination analysis [17].

Contributions. Briefly, our main theoretical and practical contributions are:

1. In Sect. 3 we introduce several semantic specializations of loops that enable the inference of *local* metering functions for complex loops by: (1) restricting the input space by means of automatically generated quasi-invariants, (2) narrowing transition guards and (3) narrowing non-deterministic choices.
2. We propose a template-based method in Sect. 4 to automate our technique which is effectively implemented by means of a Max-SMT encoding. Whereas the use of templates is not new [6], our encoding has several novel aspects that are devised to produce better lower-bounds, e.g., the addition of (soft) constraints that force the solver look for larger lower-bound functions.
3. We implement our approach in the LOBER system and evaluate it on benchmarks from the Integer Transition Systems category of the 2020 Termination and Complexity Competition (see Sect. 5). Our experimental results when compared to the existing system LoAT [12] are promising: they show further accuracy of LOBER in challenging examples that contain complex loops.

2 Background

This section introduces some notation on the program representation and recalls the notion of LB^w we aim at inferring.

2.1 Program Representation

Our technique is applicable to sequential non-deterministic programs with integer variables and commands whose updates can be expressed in linear (integer) arithmetic. We assume that the non-determinism originates from non-deterministic assignments of the form “ $x := \text{nondet}()$ ”, where x is a program variable and $\text{nondet}()$ can be represented by a fresh non-deterministic variable u . This assumption allows us to also cover non-deterministic branching, e.g., “if $(*)\{..\}$ else $\{..\}$ ” as it can be expressed by introducing a non-deterministic variable u and rewriting the code as “ $u := \text{nondet}()$; if $(u \geq 0)\{..\}$ else $\{..\}$ ”.

Our programs are represented using *transition systems*, in particular using the formalization of [17] that simplifies the presentation of some formal aspects of our work. A transition system (abbrev. TS) is a tuple $\mathcal{S} = \langle \bar{x}, \bar{u}, \mathcal{L}, \mathcal{T}, \Theta \rangle$, where $\bar{x}$ is a tuple of n integer program variables, $\bar{u}$ is a tuple of integer (non-deterministic) variables, $\mathcal{L}$ is a set of locations, $\mathcal{T}$ is a set of transitions, and Θ is

a formula that defines the valid input and is specified by a conjunction of linear constraints of the form $\bar{a} \cdot \bar{x} + b \diamond 0$ where $\diamond \in \{>, <, =, \geq, \leq\}$. A transition is of the form $(\ell, \ell', \mathcal{R}) \in \mathcal{T}$ such that $\ell, \ell' \in \mathcal{L}$, and $\mathcal{R}$ is a formula over $\bar{x}, \bar{u}$ and $\bar{x}'$ that is specified by a conjunction of linear constraints of the form $\bar{a} \cdot \bar{x} + \bar{b} \cdot \bar{u} + \bar{c} \cdot \bar{x}' + d \diamond 0$ where $\diamond \in \{>, <, =, \geq, \leq\}$, and primed variables $\bar{x}'$ represent the values of the unprimed corresponding variables after the transition. We sometimes write $\mathcal{R}$ as $\mathcal{R}(\bar{x}, \bar{u}, \bar{x}')$, use $\mathcal{R}(\bar{x})$ to refer to the constraints that involve only variables $\bar{x}$ (i.e., the *guard*), and use $\mathcal{R}(\bar{x}, \bar{u})$ to refer to the constraints that involve only variables $\bar{u}$ and (possibly) $\bar{x}$. W.l.o.g., we may assume that constraints involving primed variables are of the form $x'_i = \bar{a} \cdot \bar{x} + \bar{b} \cdot \bar{u} + c$. This is because non-determinism can be moved to $\mathcal{R}(\bar{x}, \bar{u})$ – if a primed variable x'_i appears in any expression that is not of this form, we replace x'_i by a fresh non-deterministic variable u_i in such expressions and add the equality $x'_i = u_i$. We require that for any $\bar{x}$ satisfying $\mathcal{R}(\bar{x})$, there are $\bar{u}$ satisfying $\mathcal{R}(\bar{x}, \bar{u})$, formally

$$\forall \bar{x}. \exists \bar{u}. \mathcal{R}(\bar{x}) \rightarrow \mathcal{R}(\bar{x}, \bar{u}) \quad (1)$$

This guarantees that for any state $\bar{x}$ satisfying the condition, there are values for the non-deterministic variables $\bar{u}$ such that we can make progress. A transition that does not satisfy this condition is called *invalid*. Note that (1) does not refer to $\bar{x}'$ since they are set in a deterministic way, once the values of $\bar{x}$ and $\bar{u}$ are fixed. W.l.o.g., we assume that all coefficients and free constants, in all linear constraints, are integer; and that there is a single *initial location* $\ell_0 \in \mathcal{L}$ with no incoming transitions, and a single *final location* ℓ_e with no outgoing transitions.

Example 1. The TS graphically presented in Fig. 1 is expressed as follows, considering that all inputs are valid ($\Theta = \text{true}$):

$$\begin{aligned} \mathcal{S} \equiv \langle & \{x, y\}, \emptyset, \{\ell_0, \ell_1, \ell_e\}, \\ & \{(\ell_0, \ell_1, x' = x \wedge y' = y), \\ & (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x - 1 \wedge y' = y), \\ & (\ell_1, \ell_e, x < 0 \wedge x' = x \wedge y' = y), \\ & (\ell_1, \ell_e, y \leq 0 \wedge x' = x \wedge y' = y), \\ & (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x \wedge y' = y - 1)\}, \text{true} \rangle \end{aligned}$$

A configuration C is a pair (ℓ, σ) where $\ell \in \mathcal{L}$ and $\sigma : \bar{x} \mapsto \mathbb{Z}$ is a mapping representing a state. We abuse notation and use σ to refer to $\bigwedge_{i=1}^n x_i = \sigma(x_i)$, and also write σ' for the assignment obtained from σ by renaming the variables to primed variables. There is a transition from (ℓ, σ_1) to (ℓ', σ_2) iff there is $(\ell, \ell', \mathcal{R}) \in \mathcal{T}$ such that $\exists \bar{u}. \sigma_1 \wedge \sigma_2' \models \mathcal{R}$. A (valid) trace t is a (possibly infinite) sequence of configurations $(\ell_0, \sigma_0), (\ell_1, \sigma_1), \dots$ such that $\sigma_0 \models \Theta$, and for each i there is a transition from (ℓ_i, σ_i) to $(\ell_{i+1}, \sigma_{i+1})$. Traces that are infinite or end in a configuration with location ℓ_e are called complete. A configuration (ℓ, σ) , where $\ell \neq \ell_e$, is *blocking* iff

$$\sigma \not\models \bigvee_{(\ell, \ell', \mathcal{R}) \in \mathcal{T}} \mathcal{R}(\bar{x}) \quad (2)$$

A TS is non-blocking if no trace includes a blocking configuration. We assume that the TS under consideration is non-blocking, and thus any trace is a prefix of a complete one. Throughout the paper, we represent a TS as a CFG, and analyze its strongly connected components (SCC) one by one. An SCC is said to be *trivial* if it has no edge.

2.2 Lower-Bounds

For simplicity, we assume that an execution step (a transition) costs 1. Under this assumption, the cost of a trace t is simply its length $\text{len}(t)$ where the length of an infinite trace is ∞ . In what follows, the set of all configurations is denoted by $\mathcal{C}$, the set of all valid complete traces (using a transition system $\mathcal{S}$) when starting from configuration $C \in \mathcal{C}$ is denoted by $\text{Traces}_{\mathcal{S}}(C)$, and $\mathbb{R}_{\geq 0} = \{k \in \mathbb{R} \mid k \geq 0\} \cup \{\infty\}$. For a non-empty set $M \subseteq \mathbb{R}_{\geq 0}$, $\sup M$ is the least upper bound of M and $\inf M$ is the greatest lower bound of M . The worst-case cost of an initial configuration C is the cost of the most expensive complete trace starting from C and the best-case cost is the less expensive complete trace.

Definition 1 (worst- and best-case cost). *Let $\mathcal{S}$ be a TS. Its worst-case cost function $wc_{\mathcal{S}} : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$ is $wc_{\mathcal{S}}(C) = \sup \{\text{len}(t) \mid t \in \text{Traces}_{\mathcal{S}}(C)\}$ and its best-case cost function $bc_{\mathcal{S}} : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$ is $bc_{\mathcal{S}}(C) = \inf \{\text{len}(t) \mid t \in \text{Traces}_{\mathcal{S}}(C)\}$.*

Clearly, $wc_{\mathcal{S}}$ and $bc_{\mathcal{S}}$ are not computable. Our goal in this paper is to automatically find a lower-bound function $\rho : \mathbb{Z}^n \rightarrow \mathbb{R}_{\geq 0}$ such that for any initial configuration $C = (\ell_0, \sigma)$ we have $wc_{\mathcal{S}}(C) \geq \rho(\sigma(\bar{x}))$, i.e., it is an LB^w . An LB^b would be a function $\rho : \mathbb{Z}^n \rightarrow \mathbb{R}_{\geq 0}$ that ensures that $bc_{\mathcal{S}}(C) \geq \rho(\bar{x})$ for any initial configuration $C = (\ell_0, \sigma)$. In what follows, for a function $\rho(\bar{x})$, we let $\|\rho(\bar{x})\| = \lceil \max(0, \rho(\bar{x})) \rceil$ to map all negative valuations of ρ to zero.

Example 2. Consider the TS $\mathcal{S} = \langle \{x\}, \{u\}, \{\ell_0, \ell_1, \ell_e\}, \mathcal{T}, \text{true} \rangle$ with transitions:

$$\begin{aligned} \mathcal{T} \equiv \{ & \tau_1 = (\ell_0, \ell_1, x \geq 0), \\ & \tau_2 = (\ell_1, \ell_1, x > 0 \wedge x' = x - u \wedge u \geq 1 \wedge u \leq 2), \\ & \tau_3 = (\ell_1, \ell_e, x \leq 0 \wedge x' = x) \} \end{aligned}$$

$\mathcal{S}$ contains a loop at ℓ_1 where variable x is non-deterministically decreased by 1 or 2. From any initial configuration $C_0 = (\ell_0, \sigma_0)$, the longest possible complete trace decreases x by 1 in every iteration with τ_2 , therefore $wc_{\mathcal{S}}(C_0) = \|\sigma_0(x)\| + 2$ because of the $\|\sigma_0(x)\|$ iterations in ℓ_1 plus the cost of τ_1 and τ_3 . The most precise lower bound for $wc_{\mathcal{S}}$ is $\rho(x) = \|x\| + 2$, although $\rho(x) = \|x\|$ or $\rho(x) = \|x - 2\|$ are also valid lower bounds. The shortest complete trace from C_0 decreases x by 2 in every iteration, so $bc_{\mathcal{S}}(C_0) = \|\frac{\sigma_0(x)}{2}\| + 2$. There are several valid lower bounds for $bc_{\mathcal{S}}(C_0)$ like $\rho(x) = \|\frac{x}{2}\| + 2$, $\rho(x) = \|\frac{x}{2}\|$, or $\rho(x) = 2$.

3 Local Lower-Bound Functions

Focus on Local Bounds. Existing techniques and tools for cost analysis (e.g., [1, 12]) work by inferring *local* (iteration) bounds for those parts of the TS that

correspond to loops, and then combining these bounds by propagating them “backwards” to the entry point in order to obtain a *global* bound. For example, suppose that our program consists of the following two loops:

```
assert (x>0 && z>0);
while (z > 0) { x=x+z; z--; }
while (x > 0) x--;
```

where the second loop makes x iterations (when considering the value of x just before executing the loop), and the first loop makes z iterations and increments x by z in each iteration. We are interested in inferring a global function that describes the total number of iterations of both loops, in terms of the input values x_0 and z_0 . While both loops have linear complexity locally, i.e., iteration bounds z and x , the second one has quadratic complexity w.r.t the initial values. This can be inferred automatically from the local bounds z and x by inferring how the value of x changes in the first loop, and then rewriting x in terms of the initial values to $e = x_0 + \frac{z_0 \cdot (z_0 - 1)}{2}$ (e.g., by solving corresponding recurrence relations). Now the global cost would be e plus the cost of the first loop z_0 . Rewriting the loop bound x as above is done by propagating it backwards to the entry point, and there are several techniques in the literature for this purpose that can be directly adopted in our setting to produce global bounds. These techniques can infer global bounds for nested-loops as well, given the iteration bounds of each loop. Thus, we focus on inferring local lower-bounds on the number of iterations that non-nested loops (more precisely, parts of the TS that correspond to loops) can make, and assume that they can be rewritten to global bounds by adopting the existing techniques of [1, 12] (our implementation indeed could be used as a black-box which provides local lower-bounds to these tools). Namely, we aim at inferring, for each non-nested loop, a function $\|\rho(\bar{x})\| = \lceil \max(0, \rho(x)) \rceil$ that is a (local) LB^w on its number of iterations, i.e., whenever the loop is reached with values $\bar{v}$ for the variables $\bar{x}$, it is possible to make at least $\|\rho(\bar{v})\|$ iterations.

Loops and TSs. For ease of presentation, we first consider a special case of TSs in which all locations, except the initial and exit ones define loops, and Sect. 3.6 explains how the techniques can be used for the general case. In particular, we consider that each non-trivial SCC consists of a single location ℓ and at least one transition, and we call it *loop* ℓ . Transitions from ℓ to ℓ are called *loop transitions* and their guards are called *loop guards*, and transitions from ℓ to $\ell' \neq \ell$ are called *exit transitions*. The number of iterations of a loop ℓ in a trace t is defined as the number of transitions from ℓ to ℓ , which we refer to as the cost of loop ℓ as well (since we are assuming that the cost of transitions is always 1, see Sect. 2.2). The notions of best-case and worst-case cost in Definition 1 naturally extend to the cost of a loop ℓ , i.e., we can ask what is the best-case and worst-case number of iterations of a given loop.

Overview of the Section. The overall idea of our approach is to *specialize* each loop ℓ , by restricting the initial values and/or adding constraints to its transitions, such that it becomes possible to obtain a metering function for the

specialized loop. A function that is a LB^b of the specialized loop is by definition a LB^w of loop ℓ , as it does not necessarily hold for all execution traces but rather for the class of restricted ones. Technically, inferring a LB^b of a (specialized) loop is done by inferring a metering function ρ [13], such that whenever the (specialized) loop is reached with a state σ , it is guaranteed to make at least $\|\rho(\sigma(\bar{x}))\|$ iterations. Besides, specialization is done in such away that the TS obtained by putting all specialized loops together is non-blocking, i.e., there is an execution that is either non-terminating or reaches the exit location, and thus the cost of this execution is, roughly, the sum of the costs of all (specialized) loops that are traversed. The rest of this section is organized as follows. In Sect. 3.1 we generalize the basic definition of metering function for simple loops from [12] to general types of loops and explore its limitations. Then, in the following 3 sections, we explain how to overcome these limitations by means of the following specializations: using quasi-invariants to narrow the set of input values (Sect. 3.2); narrowing loop guards to make loop transitions mutually exclusive and force some execution order between them (Sect. 3.3); and narrowing the space of non-deterministic choices to force longer executions (Sect. 3.4). Sect. 3.5 states the conditions, to be satisfied when specializing loops, in order to guarantee that the TS obtained by putting all specialized loops together is non-blocking.

3.1 Metering Functions

Metering functions were introduced by [13], as a tool for inferring a lower-bound on the number of iterations that a given loop can make. The definition is analogue to that of (linear) ranking function which is often used to infer upper-bounds on the number of iterations. The definition as given in [13] considers a loop with a single transition, and assumes that the exit condition is the negation of its guard. We start by generalizing it to our notion of loop.

Definition 2 (Metering function). *We say that a function ρ_ℓ is a metering function for a loop $\ell \in \mathcal{L}$, if the following conditions are satisfied*

$$\forall \bar{x}, \bar{u}, \bar{x}'. \mathcal{R} \rightarrow \rho_\ell(\bar{x}) - \rho_\ell(\bar{x}') \leq 1 \quad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \quad (3)$$

$$\forall \bar{x}, \bar{u}, \bar{x}'. \mathcal{R} \rightarrow \rho_\ell(\bar{x}) \leq 0 \quad \textbf{for each } (\ell, \ell', \mathcal{R}) \in \mathcal{T} \quad (4)$$

Intuitively, Condition (3) requires ρ_ℓ to decrease at most by 1 in each iteration, and Condition (4) requires ρ_ℓ to be non-positive when leaving the loop.

Assuming (ℓ, σ) is a reachable configuration in $\mathcal{S}$, it is easy to see that loop ℓ will make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations when starting from (ℓ, σ) . We require (ℓ, σ) to be reachable in $\mathcal{S}$ since we are interested only in non-blocking executions. Typically, we are interested in linear metering functions, i.e., of the form $\rho_\ell(\bar{x}) = \bar{a} \cdot \bar{x} + a_0$, since they are easier to infer and cover most loops in practice. Non-linear lower-bound functions will be obtained when rewriting these local linear lower-bounds in terms of the initial input at location ℓ_0 (see beginning of Sect. 3) and by composing nested loops (see Sect. 3.6).

Example 3 (Metering function). Consider the following loop on location ℓ_1 that decreases x (τ_1) until it takes non-positive values and exits to ℓ_2 (τ_2):

$$\tau_1 = (\ell_1, \ell_1, x \geq 0 \wedge x' = x - 1) \quad \tau_2 = (\ell_1, \ell_2, x < 0 \wedge x' = x)$$

The function $\rho_{\ell_1}(x) = x + 1$ is a valid metering function because it decreases by exactly 1 in τ_1 and becomes non-positive when τ_2 is applicable ($x < 0 \rightarrow x + 1 \leq 0$, Condition (3) of Definition 2). The function $\rho'_{\ell_1}(x) = \frac{x}{2}$ is also metering because its value decreases by less than 1 when applying τ_1 ($\frac{x}{2} - \frac{x-1}{2} = \frac{1}{2} \leq 1$) and becomes non-positive in τ_2 . Even a function as $\rho''_{\ell_1}(x) = 0$ is trivially metering, as it satisfies (3) and (4). Although all of them are valid metering functions, $\rho_{\ell_1}(x)$ is preferable as it is more accurate (i.e., larger) and thus captures more precisely the number of iterations of the loop. Note that functions like $\rho^*_{\ell_1}(x) = 2x$ or $\rho^{**}_{\ell_1}(x) = x + 5$ are not metering because they do not verify (3) (because $2x - 2(x - 1) = 2 \not\leq 1$ for $\rho^*_{\ell_1}$) or (4) (because $x < 0 \not\rightarrow x + 5 \leq 0$ for $\rho^{**}_{\ell_1}$).

3.2 Narrowing the Set of Input Values Using Quasi-Invariants

Metering functions typically exist for loops with simple loop guards. However, when guards involve more than one inequality they usually do not exist in a simple (linear) form. This is because such loops often include several exit transitions with unrelated conditions, where each one corresponds to the negation of an inequality of the guard. It is unlikely then that a non-trivial (linear) function satisfies (4) for all exit transitions. This is illustrated in the next example.

Example 4. Consider the following loop that iterates on ℓ_1 if $x \geq 0 \wedge y > 0$, and exits when $x < 0$ or $y \leq 0$:

$$\begin{aligned} \tau_1 &= (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x - 1 \wedge y' = y) \\ \tau_2 &= (\ell_1, \ell_2, x < 0 \wedge x' = x \wedge y' = y) \\ \tau_3 &= (\ell_1, \ell_2, y \leq 0 \wedge x' = x \wedge y' = y) \end{aligned}$$

Intuitively, this loop executes $x + 1$ transitions, but $\rho_{\ell_1}(x, y) = x + 1$ is not a valid metering function because it does not satisfy (4) for τ_3 : $y \leq 0 \not\rightarrow x + 1 \leq 0$. Moreover, no other function depending on x (e.g., $\frac{x}{2}$, $x - 2$, etc.) will be a valid metering function, as it will be impossible to prove (4) for τ_3 only from the information $y \leq 0$ on its guard. The only valid metering function for this loop will be the trivial one $\rho_{\ell_1}(x, y) = c$ with $c \leq 0$, which does not provide any information about the number of iterations of the loop.

Our proposal to overcome the imprecision discussed above is to consider only a subset of the input values s.t. conditions (3,4) hold in the context of the corresponding reachable states. For example, the reachable states might exclude some of the exit transitions, i.e., it is guaranteed that they are never used, and then (4) is not required to hold for them. A metering function in this context is a LB^b of the loop when starting from that specific input, and thus it is a LB^w (i.e., not necessarily best-case) of the loop when the input values are not restricted.

Technically, our analysis materializes the above idea by relying on quasi-invariants [17]. A quasi-invariant for a loop ℓ is a formula $\mathcal{Q}_\ell$ over $\bar{x}$ such that

$$\forall \bar{x}, \bar{u}, \bar{x}'. \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R} \rightarrow \mathcal{Q}_\ell(\bar{x}') \quad \text{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \quad (5)$$

$$\exists \bar{x}. \mathcal{Q}_\ell(\bar{x}) \quad (6)$$

Intuitively, $\mathcal{Q}_\ell$ is similar to an inductive invariant but without requiring it to hold on the initial states, i.e., once $\mathcal{Q}_\ell$ holds it will hold during all subsequent visits to ℓ . This also means that for executions that start in states within $\mathcal{Q}_\ell$, it is guaranteed that $\mathcal{Q}_\ell$ is an over-approximation of the reachable states. Condition (6) is used to avoid quasi-invariants that are *false*. Given a quasi-invariant $\mathcal{Q}_\ell$ for ℓ , we say that ρ_ℓ is a metering function for ℓ if the following holds

$$\forall \bar{x}, \bar{u}, \bar{x}'. \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R} \rightarrow \rho_\ell(\bar{x}) - \rho_\ell(\bar{x}') \leq 1 \quad \text{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \quad (7)$$

$$\forall \bar{x}, \bar{u}, \bar{x}'. \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R} \rightarrow \rho_\ell(\bar{x}) \leq 0 \quad \text{for each } (\ell, \ell', \mathcal{R}) \in \mathcal{T} \quad (8)$$

Intuitively, these conditions state that (3,4) hold in the context of the states induced by $\mathcal{Q}_\ell$. Assuming that (ℓ, σ) is reachable in $\mathcal{S}$ and that $\sigma \models \mathcal{Q}_\ell$, loop ℓ will make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in (ℓ, σ) .

Example 5. Recall that the loop in Example 4 only admitted trivial metering functions because of the exit transition τ_3 . It is easy to see that $\mathcal{Q}_{\ell_1} \equiv x < y$ verifies (5,6), because y is not modified in τ_1 and x decreases, and thus it is a quasi-invariant. In the context of $\mathcal{Q}_{\ell_1}$, function $\rho_{\ell_1}(x, y) = x + 1$ is metering because when taking τ_3 the value of x is guaranteed to be negative, i.e., τ_3 satisfies (8) because $x < y \wedge y \leq 0 \rightarrow x + 1 \leq 0$. Notice that $\rho_{\ell_1}(x, y) = x + 1$ will still be a valid metering function considering other quasi-invariants of the form $\mathcal{Q}'_{\ell_1} \equiv y > c$ with $c \geq 0$, as they would completely disable transition τ_3 .

3.3 Narrowing Guards

The loops that we have considered so far consist of a single loop transition, what makes easier to find a metering function. This is because there is only one way to modify the program variables (with some degree of non-determinism induced by the non-deterministic variables). However, when we allow several loop transitions, we can have loops for which a non-trivial metering function does not exist even when narrowing the set of input values.

Example 6. Consider the extension of the loop in Example 4 with a new transition τ_4 that decrements y (it corresponds to the example in Sect. 1):

$$\tau_1 = (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x - 1 \wedge y' = y)$$

$$\tau_4 = (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x \wedge y' = y - 1)$$

$$\tau_2 = (\ell_1, \ell_2, x < 0 \wedge x' = x \wedge y' = y)$$

$$\tau_3 = (\ell_1, \ell_2, y \leq 0 \wedge x' = x \wedge y' = y)$$

The most precise LB^w of this loop is $\|\rho_{\ell_1}(x, y)\|$ where $\rho_{\ell_1}(x, y) = x + y$. As mentioned, this corresponds, e.g., to an execution that uses τ_1 until $x = 0$, i.e., x times, and then τ_4 until $y = 0$, i.e., y times. It is easy to see that if we start from

a state that satisfies $x \geq 0 \wedge x \leq y$, then it will be satisfied during the particular execution that we just described. Moreover, assuming that $\mathcal{Q}_{\ell_1} \equiv x \geq 0 \wedge x \leq y$ is a quasi-invariant, it is easy to show that together with ρ_{ℓ_1} we can verify (7,8), and thus ρ_{ℓ_1} will be a metering function. However, unfortunately, $\mathcal{Q}_{\ell_1}$ is not a quasi-invariant since the above loop can make executions other than the one described above (e.g., decreasing y to 1 first and then x to 0).

Our idea to overcome this imprecision is to narrow the set of states for which loop transitions are enabled, i.e., strengthening loop guards by additional inequalities. This, in principle, reduces the number of possible executions, and thus it is more likely to find a metering function (or a better quasi-invariant), because now they have to be valid for fewer executions. For example, this might force an execution order between the different paths, or even disable some transitions by narrowing their guard to *false*. Again, a metering function for the specialized loop is not a valid LB^b of the original loop, but rather its a valid LB^w that is what we are interested in. Next, we state the requirements that such narrowing should satisfy. The choice of a narrowing that leads to longer executions is discussed in Sect. 4.

A *guard narrowing* for a loop transition $\tau \in \mathcal{T}$ is a formula $\mathcal{G}_\tau(\bar{x})$, over variables $\bar{x}$. A specialization of a loop is obtained simply by adding these formulas to the corresponding transitions. Conditions (5)-(8) can be specialized to hold only for executions that use the specialized loop as follows. Suppose that for a loop $\ell \in \mathcal{L}$ we are given a narrowing $\mathcal{G}_\tau$ for each loop transition τ , then $\mathcal{Q}_\ell$ and ρ_ℓ are quasi-invariant and metering function resp. for the corresponding specialized loop if the following conditions hold

$$\forall \bar{x}, \bar{u}, \bar{x}'. \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \wedge \mathcal{R} \rightarrow \mathcal{Q}_\ell(\bar{x}') \quad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \quad (9)$$

$$\exists \bar{x}. \mathcal{Q}_\ell(\bar{x}) \quad (10)$$

$$\forall \bar{x}, \bar{u}, \bar{x}'. \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \wedge \mathcal{R} \rightarrow \rho_\ell(\bar{x}) - \rho_\ell(\bar{x}') \leq 1 \quad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \quad (11)$$

$$\forall \bar{x}. \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R}(\bar{x}) \rightarrow \rho_\ell(\bar{x}) \leq 0 \quad \textbf{for each } (\ell, \ell', \mathcal{R}) \in \mathcal{T} \quad (12)$$

Conditions (9,10) guarantee that $\mathcal{Q}_\ell$ is a non-empty quasi-invariant for the specialized loop, and conditions (11,12) guarantee that ρ_ℓ is a metering function for the specialized loop in the context of $\mathcal{Q}_\ell$. However, in this case, function ρ_ℓ induces a lower-bound on the number of iterations only if the specialized loop is non-blocking for states in $\mathcal{Q}_\ell$. This is illustrated in the following example.

Example 7. Consider the loop from Example 3 where we have specialized the guard of τ_1 by adding $x \geq 5$:

$$\tau_1 = (\ell_1, \ell_1, x \geq 0 \wedge x \geq 5 \wedge x' = x - 1) \quad \tau_2 = (\ell_1, \ell_2, x < 0 \wedge x' = x)$$

With this specialized guard and considering $\mathcal{Q}_{\ell_1} \equiv \text{true}$, the metering function $\rho_{\ell_1}(x) = x + 1$ still satisfies (11,12), and $\mathcal{Q}_{\ell_1}$ trivially satisfies (9,10). However, ρ_{ℓ_1} is not a valid measure of the number of transitions executed because the loop gets blocked whenever x takes values $0 \leq x \leq 5$, and thus it will never execute $x + 1$ transitions.

To guarantee that the specialized loop is non-blocking for states in $\mathcal{Q}_\ell$, it is enough to require the following condition to hold

$$\forall \bar{x}. \mathcal{Q}_\ell(\bar{x}) \rightarrow \bigvee_{\tau=(\ell, \ell, \mathcal{R}) \in \mathcal{T}} (\mathcal{R}(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x})) \bigvee_{\tau=(\ell, \ell', \mathcal{R}) \in \mathcal{T}} \mathcal{R}(\bar{x}) \quad (13)$$

Intuitively, it states that from any state in $\mathcal{Q}_\ell$ we can make progress, either by making a loop iteration or exiting the loop. Assuming that (ℓ, σ) is reachable in $\mathcal{S}$ and that $\sigma \models \mathcal{Q}_\ell$, the specialized loop ℓ will make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in (ℓ, σ) . This also means that the original loop *can* make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in (ℓ, σ) .

Example 8. In Example 6, we have seen that if $\mathcal{Q}_{\ell_1} \equiv x \leq y \wedge x \geq 0$ was a quasi-invariant, then function $\rho_{\ell_1}(x, y) = x + y$ becomes metering. We can make $\mathcal{Q}_{\ell_1}$ a quasi-invariant by specializing the guards of the loop in transitions τ_1 and τ_4 to force the following execution with $x + y$ iterations: first use τ_1 until $x = 0$ (x iterations) and then use τ_4 until $y = 0$ (y iterations). This behavior can be forced by taking $\mathcal{G}_{\tau_1} \equiv x > 0$ and $\mathcal{G}_{\tau_4} \equiv x \leq 0$. With $\mathcal{G}_{\tau_1}$ we assure that x stops decreasing when $x = 0$, and with $\mathcal{G}_{\tau_4}$ we assure that τ_4 is used only when $x = 0$. Now, $\mathcal{Q}_{\ell_1} \equiv x \leq y \wedge x \geq 0$ and $\rho_{\ell_1}(x, y) = x + y$ are valid quasi-invariant and metering, resp. Function ρ_{ℓ_1} decreases by exactly 1 in τ_1 and τ_4 , is trivially non-positive in τ_2 because that transition is indeed disabled ($x \geq 0$ from $\mathcal{Q}_{\ell_1}$ and $x < 0$ from the guard) and is non-positive in τ_3 ($x \leq y \wedge y \leq 0 \rightarrow x + y \leq 0$). Regarding $\mathcal{Q}_{\ell_1}$, it verifies (9,10), and more importantly, the loop in ℓ_1 is non-blocking w.r.t $\mathcal{Q}_{\ell_1}$, $\mathcal{G}_{\tau_1}$, and $\mathcal{G}_{\tau_4}$, i.e., Condition (13) holds.

3.4 Narrowing Non-deterministic Choices

Loop transitions that involve non-deterministic variables, might give rise to executions of different lengths when starting from the same input values. Since we are interested in LB^w , we are clearly searching for longer executions. However, since our approach is based on inferring LB^b , we have to take all executions into account which might result in less precise, or even trivial, LB^w .

Example 9. Consider a modification of the loop in Example 6 in which the variable x in τ_1 is decreased by a non-deterministic positive quantity u :

$$\tau_1 = (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x - u \wedge u \geq 1 \wedge y' = y)$$

The effect of this non-deterministic variable u is that τ_1 can be applied x times if we always take $u = 1$, $\lceil \frac{x}{2} \rceil$ times if we always take $u = 2$ or even only once if we take $u > x$. As a consequence, $\rho_{\ell_1}(x, y) = x + y$ is no longer a valid metering function because x can decrease by more than 1 in τ_1 . Moreover, $\mathcal{Q}_{\ell_1} \equiv x \leq y \wedge x \geq 0$ is not a quasi-invariant anymore since $x' = x - u \wedge u \geq 1$ does not entail $x' \geq 0$. In fact, no metering function involving x will be valid in τ_1 because x can decrease by any positive amount.

To handle this complex situation, we propose narrowing the space of non-deterministic choices, and thus metering functions should be valid *wrt.* fewer

executions and more likely be found and be more precise. Next we state the requirements that such narrowing should satisfy. The choice of a narrowing that leads to longer executions is discussed in Sect. 4.

A *non-deterministic variables narrowing* for a loop transition $\tau \in \mathcal{T}$ is a formula $\mathcal{U}_\tau(\bar{x}, \bar{u})$, over variables $\bar{x}$ and $\bar{u}$, that is added to τ to restrict the choices for variables $\bar{u}$. A specialized loop is now obtained by adding both $\mathcal{G}_\tau$ and $\mathcal{U}_\tau$ to the corresponding transitions. Suppose that for loop $\ell \in \mathcal{L}$, in addition to $\mathcal{G}_\tau$, we are also given $\mathcal{U}_\tau$ for each of its loop transitions τ . For $\mathcal{Q}_\ell$ and ρ_ℓ to be quasi-invariant and metering function for the specialized loop ℓ , we require conditions (9)-(13) to hold but after adding $\mathcal{U}_\tau$ to the left-hand side of the implications in (9) and (11). Besides, unlike narrowing of guards, narrowing of non-deterministic choices might make a transition invalid, i.e., not satisfying Condition (1), and thus $\|\rho_\ell(\bar{x})\|$ cannot be used as a lower-bound on the number of iterations. To guarantee that specialized transitions are valid we require, in addition, the following condition to hold

$$\forall \bar{x} \exists \bar{u}. \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R}(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \rightarrow \mathcal{R}(\bar{x}, \bar{u}) \wedge \mathcal{U}_\tau(\bar{x}, \bar{u}) \quad \text{for each } (\ell, \tau, \mathcal{R}) \in \mathcal{T} \quad (14)$$

This condition is basically (1) taking into account the inequalities introduced by the corresponding narrowings. Assuming that (ℓ, σ) is reachable in $\mathcal{S}$ and that $\sigma \models \mathcal{Q}_\ell$, the specialized loop ℓ will make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in (ℓ, σ) , which also means, as before, that the original loop *can* make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in (ℓ, σ) .

Example 10. To solve the problems shown in Example 9 we need to narrow the non-deterministic variable u to take bounded values that reflect the worst-case execution of the loop. Concretely, we need to take $\mathcal{U}_{\tau_1} \equiv u \leq 1$, which combined with $u \geq 1$ entails $u = 1$ so x decreases by exactly 1 in τ_1 . Considering the narrowing $\mathcal{U}_{\tau_1}$, the resulting loop is equivalent to the one presented in Example 8 so we could obtain the precise metering function $\rho_{\ell_1}(x, y) = x + y$ with the quasi-invariant $\mathcal{Q}_{\ell_1} \equiv x \leq y \wedge x \geq 0$. Note that (14) holds for τ_1 because $u = 1$ makes the consequent true for every value of x and y : $\forall \bar{x} \exists \bar{u}. (x \leq y \wedge x \geq 0) \wedge (x \geq 0 \wedge y > 0) \wedge x > 0 \rightarrow u \geq 1 \wedge u \leq 1$

3.5 Ensuring the Feasibility of the Specialized Loops

In order to enable the propagation of the local lower-bounds back to the input location (as we have discussed at the beginning of Sect. 3), we have to ensure that there is actually an execution that starts in ℓ_0 and passes through the specialized loop. In other words, we have to guarantee that when putting all specialized loops together, they still form a non-blocking TS for some set of input values. We achieve this by requiring that the quasi-invariants of the preceding loops ensure that the considered quasi-invariant for this loop also holds on initialization (i.e., it is an invariant for the considered context). Technically, we require, in addition to (9)-(14), the following conditions to hold for each loop ℓ :

$$\forall \bar{x}, \bar{u}, \bar{x}'. \mathcal{Q}_{\ell'}(\bar{x}) \wedge \mathcal{R} \rightarrow \mathcal{Q}_\ell(\bar{x}') \quad \text{for each } (\ell', \ell, \mathcal{R}) \in \mathcal{T} \quad (15)$$

$$\forall \bar{x}. \mathcal{Q}_{\ell_0} \rightarrow \Theta \quad (16)$$

Condition (15) means that transitions entering loop ℓ , strengthened with the quasi-invariant of the preceding location ℓ' , must lead to states within the quasi-invariant $\mathcal{Q}_\ell$. Condition (16) guarantees that $\mathcal{Q}_{\ell_0}$ defines valid input values, i.e., within the initial condition Θ .

Theorem 1 (soundness). *Given $\mathcal{Q}_\ell$ for each non-exit location $\ell \in \mathcal{L}$, narrowings $\mathcal{G}_\tau$ and $\mathcal{U}_\tau$ for each loop transition $\tau \in \mathcal{T}$, and function ρ_ℓ for each loop location ℓ , such that (9)-(16) are satisfied, it holds:*

1. *The TS $\mathcal{S}'$ obtained from $\mathcal{S}$ by adding $\mathcal{G}_\tau$ and $\mathcal{U}_\tau$ to the corresponding transitions, and changing the initial condition to $\mathcal{Q}_{\ell_0}$, is non-blocking.*
2. *For any complete trace t of $\mathcal{S}'$, if $C = (\ell, \sigma)$ is a configuration in t , then t includes at least $\|\rho_\ell(\sigma(\bar{x}))\|$ visits to ℓ after C (i.e., $\|\rho_\ell(\bar{x})\|$ is a lower-bound function on the number of iterations of the loop defined by location ℓ).*

The proof of this soundness result is straightforward: it follows as a sequence of facts using the definitions of the conditions (9)-(16) given in this section.

We note that when there is an unbounded overlap between the guards of the loop transitions and the guards of exit transitions, it is likely that a non-trivial metering function does not exist because it must be non-positive on the overlapping states. To overcome this limitation, instead of using the exit transitions in (12), we can use ones that correspond to the negation of the guards of loop transitions, and thus it is ensured that they do not overlap. However, we should require (13) to hold for the original exit transitions as well in order to ensure that the non-blocking property holds. Another way to overcome this limitation is to simply strengthen the exit transitions by the negation of the guards.

As a final comment, we note that it is not needed to assume that the TS $\mathcal{S}$ that we start with is non-blocking (even though we have done so in Sect. 2.1 for clarity). This is because our formalization above finds a subset of $\mathcal{S}$ ($\mathcal{S}'$ in Theorem 1) that is non-blocking, which is enough to ensure the feasibility of the local lower-bounds. This is useful not only for enlarging the set of TSs that we accept as input, but also allows us to start the analysis from any subset of $\mathcal{S}$ that includes a path from ℓ_0 to the exit location. For example, it can be used to remove trivial execution paths from $\mathcal{S}$, or concentrate on ones that include more sequences of loops (since we are interested in LB^w).

3.6 Handling General TSs

So far we have considered a special case of TSs in which all locations, except the entry and exit ones, are multi-path loops. Next we explain how to handle the general case. It is easy to see that we can allow locations that correspond to trivial SCCs. These correspond to paths that connect loops and might include branching as well. For such locations, there is no need to infer metering functions or apply any specialization, we only need to assign them quasi-invariants that satisfy (15) to guarantee that the overall specialized TS is non-blocking.

The more elaborated case is when the TS includes non-trivial SCCs that do not form a multi-path loop. In such case, if a SCC has a single cut-point, we

can unfold its edges and transform it into a multi-path following the techniques of [1]. It is important to note that when merging two transitions, the cost of the new one is the sum of their costs. In this case the number of iterations is still a lower-bound on the cost of the loop, however, we might get a better one by multiplying it by the minimal cost of its transitions.

If a SCC cannot be transformed into a multi-path loop by unfolding its transitions, then it might correspond to a nested loop, and, in such case, we can recover the nesting structure and consider them as separated TSs that are “called” from the outer one using loop extraction techniques [25]. Each inner-loop is then analyzed separately, and replaced (in the original TS, where is “called”) by a single edge with its lower-bound as cost for that edge, and then the outer is analyzed taking that cost into account. Besides, to guarantee that the specialized program corresponds to a valid execution, we require the quasi-invariant of the inner loop to hold in the context of the quasi-invariant of the outer loop. This approach is rather standard in cost analysis of structured programs [1, 3, 12].

Another issue is how to compose the (local) lower-bounds of the specialized loops into a global-lower bound. For this, we can rely on the techniques [1, 3] that rewrite the local lower-bounds in terms of the input values by relying on invariant generation and recurrence relations solving.

4 Inference Using Max-SMT

This section presents how metering functions and narrowings can be inferred automatically using Max-SMT, namely how to automatically infer all $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, $\mathcal{Q}_\ell$, and ρ_ℓ such that (9)-(16) are satisfied. We do it in a modular way, i.e., we seek $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, $\mathcal{Q}_\ell$, and ρ_ℓ for one loop at a time following a (reversed) topological order of the SCCs, as we describe next. Recall that (16) is required only for loops connected directly to ℓ_0 , and w.l.o.g. we assume there is only one such loop.

4.1 A Template-Based Verification Approach

We first show how the template-based approach of [6, 17] can be used to find $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, and $\mathcal{Q}_\ell$ by representing them as template constraint systems, i.e., each is a conjunction of linear constraints where coefficients and constants are unknowns. Also, ρ_ℓ is represented as a linear template function $\bar{a} \cdot \bar{x} + a_0$ where $(a_0, \bar{a})$ are unknowns. Then, the problem is to find concrete values for the unknowns such that all formulas generated by (9)-(16) are satisfied:

- Each $\forall$ -formula generated by (9)-(16), except those of (14) that we handle below, can be viewed as an $\exists\forall$ problem where the $\exists$ is over the unknowns of the templates and the $\forall$ is over (some of) the program variables. It is well-known that solving such an $\exists\forall$ problem, i.e., finding values for the unknowns, can be done by translating it into a corresponding $\exists$ problem over the existentially quantified variables (i.e., the unknowns) using Farkas’ lemma [20], which can then be solved using an off-the-shelf SMT solver.

- To handle (14) we follow [17], and eliminate $\exists \bar{u}$ using the skolemization $u_i = \bar{a} \cdot \bar{x} + a_0$ where $(a_0, \bar{a})$ are fresh unknowns (different for each u_i). This allows handling it using Farkas' lemma as well. However, in addition, when solving the corresponding $\exists$ problem we require all $(a_0, \bar{a})$ to be integer. This is because the domain of program variables is the integers, and picking integer values for all $(a_0, \bar{a})$ guarantees that the values of any x'_i that depends on $\bar{u}$ will be integer as well¹.

The size of templates for $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, and $\mathcal{Q}_\ell$, i.e., the number of inequalities, is crucial for precision and performance. The larger the size is, the more likely that we get a solution if one exists, but also the worse the performance is (as the corresponding SMT problem will include more constraints and variables). In practice, one typically starts with templates of size 1, and iteratively increases it by 1 when failing to find values for the unknowns, until a solution is found or the bound on the size is reached.

Alternatively, we can use the approach of [17] to construct $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, and $\mathcal{Q}_\ell$ incrementally. This starts with templates of size 1, but instead of requiring all (9)-(16) to hold, the conditions generated by (12) are marked as soft constraints (i.e., we accept solutions in which they do not hold) and use Max-SMT to get a solution that satisfies as many of such soft conditions as possible. If all are satisfied, we are done, if not, we use the current solution to instantiate the templates, and then add another template inequality to each of them and repeat the process again. This means that at any given moment, each template will include at most one inequality with unknowns. Finally, to guarantee progress from one iteration to another, soft conditions that hold at some iteration are required to hold at the next one, i.e., they become hard.

The use of (12) as soft constraint is based on the observation [12] that when seeking a metering function, the problematic part is often to guarantee that it is negative on exit transitions, which is normally achieved by adding quasi-invariants that are incrementally inferred. By requiring (12) to be soft we handle more exit transitions as the quasi-invariant gets stronger until all are covered.

4.2 Better Quality Solutions

The precision can also be affected by the quality of the solution picked by the SMT solver for the corresponding $\exists$ problem. Since there might be many metering functions that satisfy (9)-(16), we are interested in narrowing the search space of the SMT solver in order to find more accurate ones, i.e., lead to longer executions. Next we present some techniques for this purpose.

Enabling More Loop Transitions. We are interested in guard narrowings that keep as many loop transitions as possible, since such narrowings are more likely

¹ Because we assumed that constraints involving primed variables are of the form $x'_i = \bar{a} \cdot \bar{x} + \bar{b} \cdot \bar{u} + c$.

to generate longer executions. This can be done by requiring the following to hold

$$\exists \bar{x}. \bigvee_{\tau=(\ell, \ell, \mathcal{R}) \in \mathcal{T}} (\mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R}(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x})) \quad (17)$$

We also use Max-SMT to require a solution that satisfies as many disjuncts as possible and thus eliminating less loop transitions (if $\mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R}(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x})$ is *false* for a transition τ , then it is actually disabled). Note that this condition can be used instead of (10) that requires the quasi-invariant to be non-empty.

Larger Metering Functions. We are interested in metering functions that lead to longer executions. One way to achieve this is to require metering functions to be ranking as well, i.e., in addition to (11) we require the following to hold

$$\forall \bar{x}, \bar{u}, \bar{x}'. \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \wedge \mathcal{U}_\tau(\bar{x}, \bar{u}) \wedge \mathcal{R} \rightarrow \rho_\ell(\bar{x}) - \rho_\ell(\bar{x}') \geq 1 \quad \text{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \quad (18)$$

$$\forall \bar{x}, \bar{u}. \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \wedge \mathcal{R}(\bar{x}) \rightarrow \rho_\ell(\bar{x}) \geq 0 \quad \text{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \quad (19)$$

These new conditions are added as soft constraints, and we use Max-SMT to ask for a solution that satisfies as many conditions as possible.

Unbounded Metric Functions. We are interested in metering functions that do not have an upper bound, since otherwise they will lead to constant lower-bound functions. For example, for a loop with a transition $x \geq 0 \wedge x' = x - 1$, we want to avoid quasi-invariants like $x \leq 5$ which would make the metering function x bounded by 5. For this, we rely on the following lemma.

Lemma 1. *A function $\rho(\bar{x}) = \bar{a} \cdot \bar{x} + a_0$ is unbounded over a polyhedron $\mathcal{P}$, iff $\bar{a} \cdot \bar{y}$ is positive on at least one ray $\bar{y}$ of the recession cone of $\mathcal{P}$.*

It is known that for a polyhedron $\mathcal{P}$ given in constraints representation, its recession cone $\text{cone}(\mathcal{P})$ is the set specified by the constraints of $\mathcal{P}$ after removing all free constants. Now we can use the above lemma to require that the metering function $\rho_\ell(\bar{x}) = \bar{a} \cdot \bar{x} + \bar{a}_0$ is unbounded in the quasi-invariant $\mathcal{Q}_\ell$ by requiring the following condition to hold

$$\exists \bar{x}. \text{cone}(\mathcal{Q}_\ell) \wedge \bar{a} \cdot \bar{x} > 0 \quad (20)$$

where $\text{cone}(\mathcal{Q}_\ell)$ is obtained from the template of $\mathcal{Q}_\ell$ by removing all (unknowns corresponding to) free constants, i.e., it is the *recession cone* of $\mathcal{Q}_\ell$.

Note that all encodings discussed in this section generate non-linear SMT problems, because they either correspond to $\exists\forall$ problems that include templates on the left-hand side of implications, or to $\exists$ problems over templates that include both program variables and unknowns.

Finally, it is important to note that the optimizations described provide theoretical guarantees to get better lower bounds: the one that adds (18,19) leads to a bound that corresponds exactly to the worst-case execution (of the specialized program), and the one that uses (20) is essential to avoid constant bounds.

5 Implementation and Experimental Evaluation

We have implemented a *Lower-Bound synthesizer*, named LOBER, that can be used from an online web interface at <http://costa.fdi.ucm.es/lober>. LOBER is built as a pipeline with the following processes: (1) it first reads a KoAT file [5] and generates a corresponding set of multi-path loops, by extracting parts of the TS that correspond to loops [25], applying unfolding, and inferring loop summaries to be used in the calling context of nested loops, as explained in Sect. 3.6; (2) it then encodes in SMT the conditions (9)–(13) defined through the paper, for each loop separately, by using template generation, a process that involves several non-trivial implementations using Farkas’ lemma (this part is implemented in Java and uses Z3 [8] for simple (linear) satisfiability checks when producing the Max-SMT encoding); (3) the problem is solved using the SMT solver Barcelogic [4], as it allows us to use non-linear arithmetic and Max-SMT capabilities in order to assert soft conditions and implement the solutions described in Sect. 4; (4) in order to guarantee the correctness of our system results, we have added to the pipeline an additional checker that proves that the obtained metering function and quasi-invariants verify conditions (9)–(13) by using Z3. To empirically evaluate the results of our approach, we have used benchmarks from the *Termination Problem Data Base (TPDB)*, namely those from the category *Complexity_ITS* that contains Integer Transition Systems. We have removed non-terminating TSs and terminating TSs whose cost is unbounded (i.e., the cost depends on some non-deterministic variables and can be arbitrarily high) or non-linear, because they are outside the scope of our approach. In total, we have considered a set of 473 multi-path loops from which we have excluded 13 that were non-linear. Analyzing these 473 programs took 199 min, an average of 25 sec by program, approximately. For 255 of them, it took less than 1 s.

Table 1 illustrates our results and compares them to those obtained by the LoAT [12, 13] system, which also outputs a pair $(\rho, \mathcal{Q})$ of a lower-bound function ρ and initial conditions $\mathcal{Q}$ on the input for which ρ is a valid lower-bound. In order to automatically compare the results obtained by the two systems, we have implemented a comparator that first expresses costs as functions $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ over a single variable n and then checks which function is greater. To obtain this unary cost function from the results $(\rho, \mathcal{Q})$, we use convex polyhedra manipulation libraries to maximize the obtained cost ρ wrt. $\mathcal{Q} \wedge -n \leq x_i \leq n$, where x_i are the TS variables, and express that maximized expressions in terms of n . Therefore, $f(n)$ represents the maximum cost when the variables are bounded by $|x_i| \leq n$ and satisfy the corresponding initial condition $\mathcal{Q}$, a notion very similar to the runtime complexity used in [12, 13]. Once we have both unary linear costs $f_1(n) = k_1n + d_1$ and $f_2(n) = k_2n + d_2$, we compare them in $n \geq 0$ by inspecting k_1 and k_2 .

Each row of the table contains the number of loops for which both tools obtain the same result ($=$), the number of loops where LOBER is better than LoAT ($>$) and the number of loops where LoAT is better than LOBER ($<$). The subcategories are obtained directly from the name of the innermost folder, except for the cases in which this folder contains too few examples that we merge them

Table 1. Results of the experiments.

Benchmark set	Total	=	>	<	Benchmark set	Total	=	>	<
BROCKSCHMIDT_16					FGPSF09/Misc	20	16	3	1
c-examples/ABC	33	33	0	0	KoAT-2013	10	10	0	0
c-examples/SPEED	29	25	4	0	KoAT-2014	14	14	0	0
c-examples/WTC	45	39	4	2	SAS10	46	40	1	5
c-examples/Misc	9	9	0	0	FLORES-MONTOYA_16	176	158	16	2
costa	6	5	1	0	HARK_20				
FGPSF09/Beerendonk	28	24	4	0	Ben_Amram_Genaim	10	7	2	1
FGPSF09/patrs	18	16	2	0	Nils_2019	16	16	0	0

all in a Misc folder in the parent directory. The total number of loops that are considered in each subcategory appears in column **Total**. BROCKSCHMIDT_16 and HARK_20 have their first row empty as all their results are contained in their subcategories. Globally, both tools behave the same in 412 programs (column “=”), obtaining equivalent linear lower bounds in 376 of them and a constant lower bound in the remaining ones. Our tool LOBER achieves a better accuracy in 37 programs (column “>”), while LoAT is more precise in 11 programs (column “<”). Let us discuss the two sets of programs in which both tools differ. As regards the 37 examples for which we get better results, we have that LoAT crashes in 4 cases and it can only find a constant lower bound in 1 example while our tool is able to find a path of linear length by introducing the necessary quasi-invariants. For the remaining 32 loops, both tools get a linear bound, but LOBER finds one that leads to an unboundedly longer execution: 18 of these loops correspond to cases that have implicit relations between the different execution paths (like our running examples) and require semantic reasoning; for the remaining 14, we get a better set of quasi-invariants. The following techniques have been needed to get such results in these 37 better cases (note that (i) is not mutually exclusive with the others):

- (i) 1 needs narrowing non-deterministic choices,
- (ii) 5 do not need quasi-invariants nor guard narrowing,
- (iii) 14 need quasi-invariants only,
- (iv) 18 need both quasi-invariants and guard narrowing (in 3 of them this is only used to disable transitions).

Therefore, this shows experimentally the relevance of all components within our framework and its practical applicability thanks to the good performance of the Max-SMT solver on non-linear arithmetic problems. In general, for all the set of programs, we can solve 308 examples without quasi-invariants and 444 without guard-narrowing. The intersection of these two sets is: 298 examples (63% of the programs), that leaves 175 programs that need the use of some of the proposed techniques to be solved.

As regards the 11 examples for which we get worse results than LoAT, we have two situations: (1) In 6 cases, the SMT-solver is not able to find a solution.

We noticed that too many quasi-invariants were required, what made the SMT problem too hard. To improve our results, we could start, as a preprocessing step, from a quasi-invariant that includes all invariant inequalities that syntactically appear in the loop transitions, something similar to what is done by LoAT when inferring what they call conditional metering function [12]. This is left for future experimentation. (2) In the other 5 cases, our tool finds a linear bound but with a worse set of quasi-invariants, which makes the LoAT bound provide unboundedly longer executions. We are investigating whether this can be improved by adding new soft constraints that guide the solver to find these better solutions. Finally, let us mention that, for the 13 problems that LoAT gives a non-linear bound and have been excluded from our benchmarks as justified above, we get a linear bound for the 12 that have a polynomial bound (of degree 2 or more), and a constant bound for the additional one that has a logarithmic lower bound. This is the best we can obtain as our approach focuses on the inference of precise local linear bounds, as they constitute the most common type of loops.

All in all, we argue that our experimental results are promising: we triple LoAT in the number of benchmarks for which we get more accurate results and, besides, many of those examples correspond to complex loops that lead to worse results when disconnecting transitions. Besides, we see room for further improvement, as most examples for which LoAT outperforms us could be handled as accurately as them with better quasi-invariants (that is somehow a black-box component in our framework). Syntactic strategies that use invariant inequalities that appear in the transitions, like those used in LoAT, would help, as well as further improvements in SMT non-linear arithmetic.

Application Domains. The accuracy gains obtained by LOBER have applications in several domains in which knowing the precise cost can be fundamental. This is the case for predicting the gas usage [26] of executing *smart contracts*, where gas cost amounts to monetary fees. The caller of a transaction needs to include a gas limit to run it. Giving a too low gas limit can end in an “out of gas” exception and giving a too high gas limit can end in a “not enough eth (money)” error. Therefore having a tighter prediction is needed to be safe on both sides. Also, when the UB is equal to the LB, we have an exact estimation, e.g., we would know precisely the runtime or memory consumption of the most costly executions. This can be crucial in safety-critical applications and has been used as well to detect potential vulnerabilities such as denial-of-service attacks. In <https://apps.dtic.mil/sti/pdfs/AD1097796.pdf>, vulnerabilities are detected in situations in which both bounds do not coincide. For instance, in password verification programs, if the UB and LB differ due to a difference on the delays associated to how many characters are right in the guessed password, this is identified as a potential attack.

6 Related Work and Conclusions

We have proposed a novel approach to synthesize precise lower-bounds from integer non-deterministic programs. The main novelties are on the use of loop

specialization to facilitate the task of finding a (precise) metering function and on the Max-SMT encoding to find larger (better) solutions. Our work is related to two lines of research: (1) non-termination analysis and (2) LB inference. In both kinds of analysis, one aims at finding classes of inputs for which the program features a non-terminating behavior (1) or a cost-expensive behavior (2). Therefore, techniques developed for non-termination might provide a good basis for developing a LB analysis. In this sense, our work exploits ideas from the Max-SMT approach to non-termination in [17]. The main idea borrowed from [17] has been the use of quasi-invariants to specialize loops towards the desired behavior: in our case towards the search of a metering function, while in theirs towards the search of a non-termination proof. However, there are fundamental differences since we have proposed other new forms of loop specialization (see a more detailed comparison in Sect. 1) and have been able to adapt the use of Max-SMT to accurately solve our problem (i.e., find larger bounds). As mentioned in Sect. 1, our loop specialization technique can be used to gain precision in non-termination analysis [17]. For instance, in this loop: “while ($x \geq 0$ and $y \geq 0$) {if (*) { $x++$; $y--$;} else { $x--$; $y++$;} }” no sub SCC (considering only one of the transitions) is non-terminating and no quasi-invariant can be found to ensure we will stay in the loop (when considering both transitions), hence cannot be handled by [17]. Instead if we narrow the transitions by adding $y \geq x$ in the if-condition (and hence $x > y$ in the else), we can prove that $x \geq 0 \wedge y \geq 0 \wedge x + y = 1$ is quasi-invariant, which allow us to prove non-termination in the way of [17] (as we will stay in the loop forever).

As regards LB inference, the current state-of-the-art is the work by Frohn et al. [12, 13] that introduces the notion of metering function and acceleration. Our work indeed tries to recover the semantic loss in [12, 13] due to defining metering functions for simple loops and combining them in a later stage using acceleration. Technically, we only share with this work the basic definition of metering function in Sect. 3.1. Indeed, the definition in conditions (3) and (4) already generalizes the one in [12, 13] since it is not restricted to simple loops. This definition is improved in the following sections with several loop specializations. While [12, 13] relies on pure SMT to solve the problem, we propose to gain precision using Max-SMT. We believe that similar ideas could be adapted by [12, 13]. Due to the different technical approaches underlying both frameworks, their accuracy and efficiency must be compared experimentally wrt. the LoAT system that implements the ideas in [12, 13]. We argue that the results in Sect. 5 justify the important gains of using our new framework and prove experimentally that, the fact that we do not lose semantic relations in the search of metering functions is key to infer LB for challenging cases in which [12, 13] fails. Originally, the LoAT [12, 13] system only accelerated simple loops by using metering functions, so the overall precision of the lower bound relied on obtaining valid and precise metering functions. However, the framework in [12, 13] is independent of the accelerating technique applied. In order to increase the number of simple loops that can be accelerated, Frohn [11] proposes a calculus to combine different conditional acceleration techniques (monotonic increase/decrease, eventual

increase/decrease, and metering functions). These conditional acceleration techniques assume that all the iterations of the loop verify some condition φ , and the calculus applies the techniques in order and extract those conditions φ from fragments of the loop guard. Although more precise and powerful, the combined acceleration calculus considers only simple loops, so it does not solve the precision loss when the loop cost involves several interleaved transitions. Moreover, the techniques in [11] are integrated into LoAT, so the experimental evaluation in Sect. 5 compares our approach to the framework in [12, 13] extended with several techniques to accelerate loops (not only metering functions).

Finally, our approach presents similarities to the CTL* verification for ITS in [7] as both extend transition guards of the original ITS. The difference is that in [7] the added constraints only contain newly created *prophecy variables* and the transitions to modify are detected directly using graph algorithms; whereas our SMT-based approach adds constraints only over existing variables to satisfy the properties that characterize a good metering function. Additionally, both approaches differ both in the goal (CTL* verification vs. inference of lower-bounds) and the technologies applied (CTL model checkers vs. Max-SMT solvers).

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 221–237. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69166-2_15
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of java bytecode. In: De Nicola, Rocco (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_12
3. Albert, E., Genaim, S., Masud, A.N.: On the inference of resource usage upper and lower bounds. *ACM Trans. Comput. Logic* **14**(3), 1–35 (2013)
4. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: The barcelogic SMT solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 294–298. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_27
5. Brockschmidt, M., et al.: Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.* **38**(4), 1–50 (2016)
6. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_39
7. Cook, B., Khlaaf, H., Piterman, N.: On automation of CTL* verification for infinite-state systems. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 13–29. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_2
8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

9. Debray, S.K., López-García, P., Hermenegildo, M., Lin, N.-W.: Lower Bound Cost Estimation for Logic Programs. The MIT Press, In Logic Programming (1997)
10. Flores-Montoya, A.: Upper and lower amortized cost bounds of programs expressed as cost relations. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 254–273. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_16
11. Frohn, F.: A calculus for modular loop acceleration. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 58–76. Springer International Publishing (2020)
12. Frohn, F., Naaf, M., Brockschmidt, M., Giesl, J.: Inferring lower runtime bounds for integer programs. ACM Trans. Program. Lang. Syst. **42**(3), 1–50 (2020)
13. Frohn, F., Naaf, M., Hensel, J., Brockschmidt, M., Giesl, J.: Lower runtime bounds for integer programs. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 550–567. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_37
14. Gulwani, S., Mehra, K.K., Chilimbi, T.: SPEED. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2009. ACM Press (2008)
15. Hark, M., Kaminski, B.L., Giesl, J., Katoen, J.-P.: Aiming low is harder: induction for lower bounds in probabilistic program verification. In: Proceedings of the ACM on Programming Languages, 4(POPL), pp. 1–28, January 2020
16. Hoffmann, J., Shao, Z.: Automatic static cost analysis for parallel programs. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 132–157. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_6
17. Larraz, D., Nimkar, K., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving non-termination using Max-SMT. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 779–796. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_52
18. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: resource analysis for probabilistic programs. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, June 2018
19. Otto, C., Brockschmidt, M., Von Essen, C., Giesl, J.: Automated termination analysis of java bytecode by term rewriting. In: Lynch, C. (eds.) RTA, vol. 6 of LIPIcs, pp. 259–276. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)
20. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Chichester (1986)
21. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 745–761. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_50
22. Sinn, M., Zuleger, F., Veith, H.: Complexity and resource bound analysis of imperative programs using difference constraints. J. Autom. Reasoning **59**(1), 3–45 (2017). <https://doi.org/10.1007/s10817-016-9402-4>
23. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. ACM SIGPLAN Not. **44**(6), 223–234 (2009)
24. Wegbreit, B.: Mechanical program analysis. Commun. ACM **18**(9), 528–539 (1975)
25. Wei, Tao, Mao, Jian, Zou, Wei, Chen, Yu.: A new algorithm for identifying loops in decompilation. In: Nielson, Hanne Riis, Filé, Gilberto (eds.) SAS 2007. LNCS, vol. 4634, pp. 170–183. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_11
26. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger (2014)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.