TESIS DOCTORAL

Análisis de recursos de programas enteros y abstractos

Resource analysis of integer and abstract programs

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Alicia Merayo Corcoba

Directora

Elvira María Albert Albiol

Madrid

Universidad Complutense de Madrid
Facultad de Informática

Tesis doctoral

Análisis de recursos de programas enteros y abstractos
Resource Analysis of Integer and Abstract Programs

Memoria para optar al grado de doctora

presentada por

Alicia Merayo Corcoba

Directora

Elvira María Albert Albiol

Análisis de recursos de programas enteros y abstractos

Resource Analysis of Integer and Abstract Programs

TESIS DOCTORAL

Memoria presentada para obtener el grado de doctora en Doctorado en Ingeniería Informática (RD 99/2011)
por
Alicia Merayo Corcoba

Dirigida por la profesora
Elvira María Albert Albiol

Facultad de Informática
Universidad Complutense de Madrid

Madrid, 2022

Análisis de recursos de programas enteros y abstractos

TESIS DOCTORAL

Memoria presentada para obtener el grado de doctora en Doctorado en Ingeniería Informática (RD 99/2011)
por
Alicia Merayo Corcoba

Dirigida por la profesora
Elvira María Albert Albiol

Facultad de Informática
Universidad Complutense de Madrid

Madrid, 2022

Resource Analysis of Integer and Abstract Programs

PhD THESIS

Alicia Merayo Corcoba

Advisor:
Elvira María Albert Albiol

Facultad de Informática
Universidad Complutense de Madrid

Madrid, 2022

Financial Support. This work was funded partially by the Spanish MINECO project TIN2015-69175-C4-2-R, by the Spanish MICINN and FEDER project RTI2018-094403-B-C31, by the CM projects S2013/ICE-3006 and S2018/TCS-4314, by the UCM CT42/18-CT43/18 grant and by the Spanish MECD FPU2017 grant.

Agradecimientos

Un día, hace un par de años, leí esta historia:

Un día, el hijo de un acaudalado empresario le preguntó a su padre qué era ser pobre. El empresario, queriendo que su hijo lo supiera de primera mano, lo dejó tres días en casa de unos granjeros. Al cabo de esos tres días, volvió a recoger a su hijo.

- ¿Qué te pareció la experiencia? - le preguntó el padre a su hijo cuando regresaban a casa.

- Buena. - contestó su hijo mirando hacia la lejanía.

- Y... ¿qué aprendiste? - insistió su padre.

Entonces el hijo, mirando al padre, respondió:

- Que nosotros tenemos un perro y ellos tienen cuatro. Nosotros tenemos un jacuzzi y ellos un río enorme de agua cristalina con peces. Nosotros tenemos que poner focos para alumbrar el jardín mientras ellos se alumbran con las estrellas y la luna. Nosotros tenemos un patio que llega hasta la cerca y el suyo, hasta el horizonte. Nosotros tenemos que comprar nuestra comida mientras que ellos siembran y cosechan la suya de forma natural. Nosotros tenemos que oír la música con auriculares mientras ellos viven continuamente con el trino de los pájaros. Nosotros cocinamos en hornos microondas y ellos cocinan todo con el sabor de la leña. Nosotros necesitamos muros y alarmas para protegernos y ellos viven con las puertas abiertas protegidos por la amistad de los vecinos. Nosotros vivimos conectados a la tecnología y las redes sociales... pero ellos conectan con la naturaleza y con su familia.

El padre, atónito, miró a su hijo. Este, tras mirarlo a los ojos, terminó:

- Gracias, papá. Gracias por haberme enseñado lo pobres que somos nosotros y lo ricos que son ellos.

Todo lo que ese niño descubrió en esa granja yo lo tuve desde pequeña, eso y más. Me crié con animales, ríos con cascadas, lluvias de estrellas fugaces desde el jardín, atardeceres de mil colores, comida con sabor a casa, despertares escuchando a los pájaros por las mañanas, una puerta siempre abierta... y amor, sobretodo mucho amor.

Un día, hace diez años, me fui a Madrid. Volvía a ser rica en vacaciones, cuando hablaba por teléfono con casa, cuando me llevaba parte de esa riqueza a Madrid en la maleta... Hasta que llegó el COVID y yo volví a ser muy rica muchos meses seguidos: un confinamiento es menos confinamiento en el campo.

Gracias, papá y mamá, por darme siempre esa riqueza, ese amor, ese lugar al que

siempre volver llamado casa. Gracias a mi hermana y a Diego, por compartir esa riqueza y hacerla más grande con Martín. Y gracias a Sidrín, por devolverme toda esa riqueza, dándome los motivos que necesitaba para volver definitivamente a casa, el apoyo estos dos años y la fuerza para afrontar las cosas nuevas.

Gracias a Elvira, por todo lo que me has guiado y cuidado estos años. Madrid ha sido más bonito gracias a ti. Gracias por entender que, aunque Madrid significaba una gran oportunidad, volver a casa para mí significaba aún más.

Gracias al resto del grupo COSTA, en especial a Samir, por guiarme desde mis primeros pasos en la investigación. Gracias a todos los coautores de las publicaciones de esta tesis. Gracias al personal de la Facultad de Informática, muchos habéis sido maestros pero todos habéis sido amigos. Gracias a Pablo, Miguel y Jesús por acogerme desde el principio, a mis compañeros del antiguo Aula 16 y a los compañeros de cafés.

Gracias a los tíos y a Ana, por ser casa a 400 km de mi casa original.

Gracias a todos con los que he creado recuerdos en Madrid. Soy quien soy ahora gracias a lo que aprendí con cada uno de vosotros.

Y gracias a ti, Madrid, por todos estos años maravillosos.

Acknowledgments

One day, a couple of years ago, I read this story:

One day, the son of a rich businessman asked his father what it was like to be poor. The businessman, wanting his son to know firsthand, left him three days on a farm with a family. At the end of those three days, he returned to pick up his son home.

- What do you think about the experience? - asked the father when returning home.

- It was great - replied the son, looking into the distance.

- And what did you learn? - his father asked again.

Then the son, looking at his father, replied:

- I learned that we have a dog and they have four. We have a jacuzzi and they have a huge river of clear water with many fishes. We have to put spotlights to illuminate the garden while they light up with the stars and the moon. We have a garden that goes up to the fence and theirs, up to the horizon. We have to buy our food while theirs is natural. We have to hear the music with headphones while they continuously live with the song of the birds. We cook in a microwave oven and they cook everything with the flavour of firewood. We need walls and alarms to protect us while they live with open doors protected by the friendship of neighbours. We live connected to technology and social networks... but they connect with nature and with his family.

The father, astonished, looked at his son. His son, after looking into his eyes, finished:

- Thanks, Dad. Thank you for having taught me how poor we are and how rich they are.

I had everything that that guy discovered on that farm since I was little. I grew up with animals, rivers with waterfalls, shooting stars, sunsets of a thousand colours, homemade food, waking ups listening to the birds in the morning, a door that was always open... and love, lots of love.

One day, ten years ago, I went to Madrid. I was rich again on vacation, when I spoke on the phone with home, when I took a piece of it to Madrid in the suitcase... Until COVID came and I was very rich again for many months in a row: one confinement is less confinement in the field.

Thank you, Dad and Mom, for always giving me that wealth, that love, that place to always come back called home. Thanks to my sister and Diego, for sharing that wealth and making it bigger with Martin. And thanks to Sidrín, for giving me back all that wealth, giving me the reasons I needed to definitely return home, the support I needed

these two years and the strength to face new things.

Thanks to Elvira, for everything you have guided and cared for me over these years. Madrid has been more beautiful thanks to you. Thank you for understanding that, although Madrid meant a great opportunity, coming home meant even more to me.

Thanks to the rest of the COSTA group, especially Samir, for guiding me during my first steps in research. Thanks to all the co-authors of the papers included in this thesis. Thanks to the staff of the Faculty of Computing, many of you have been teachers but you have all been friends. Thanks to Pablo, Miguel and Jesús for welcoming me from the beginning, to my colleagues at Classroom 16 and also cafe colleagues.

Thanks to the uncles and Ana, for being a house 400 km away from my original house.

Thanks to everyone with whom I have created memories in Madrid. I am who I am now thanks to what I learned with each one of you.

And thanks to you, Madrid, for all these wonderful years.

Contents

Resumen **vii**

Abstract **xi**

I Contents of the thesis **1**

1 Introduction **3**

1.1 Static Cost Analysis 3

1.1.1 Automated Cost Analysis 3

1.1.2 Application Domains 5

1.2 Verification and Certification 6

1.3 Contributions and Structure 6

2 Quantitative Abstract Execution **11**

2.1 Quantitative Abstract Execution by Example 12

2.2 Abstract Cost Analysis 16

2.2.1 Inference of Abstract Cost Relations 16

2.2.2 From Abstract Cost Relation Systems to Abstract Cost Invariants 19

2.2.3 From Cost Invariants to Postconditions 21

2.3 Experimental Evaluation 21

3 Inference of Lower Bounds on the Worst-Case Cost **23**

3.1 Background 25

3.1.1 Program Representation 25

3.1.2 Lower Bounds 26

3.2 Local Lower-Bound Functions 27

3.2.1 Metering Functions 28

3.2.2 Narrowing the Set of Input Values Using Quasi-Invariants 29

3.2.3 Narrowing Guards 30

3.2.4 Narrowing Non-deterministic Choices 33

3.3 Inference Using Max-SMT 35

3.4 Implementation and Experimental Evaluation 35

4 Certified Abstract Cost Analysis with Upper and Lower Bounds **39**
4.1 Cost Annotations by Example 40
4.2 Abstract Cost Analysis 43
4.2.1 Soundness of Abstract Cost Relations 43
4.2.2 From Abstract Cost Relation Systems to Abstract Cost Invariants 45
4.2.3 From Cost Invariants to Postconditions 48
4.3 Experimental Evaluation 48

5 Conclusions and Future Work **51**

Bibliography **59**

II Papers of the thesis **61**

6 List of Publications **63**
Certified Abstract Cost Analysis 65
Lower-Bound Synthesis using Loop Specialization and Max-SMT 87
Certified Cost Bounds for Abstract Programs 111

Resumen

Desde el comienzo de la computación automática a mediados del siglo pasado, el avance de la informática ha ido ligado a una cada vez mayor importancia en todos los ámbitos de la sociedad actual. La inclusión de procesos informáticos en la vida cotidiana y, en particular, su inclusión en situaciones críticas, no puede ir ligada solo a la generación del hardware y el software, sino también al análisis y verificación de todos sus componentes. Mientras que el *análisis de hardware* es crucial para la generación de la infraestructura informática y el mantenimiento de la misma, detectando o prediciendo componentes que puedan funcionar de manera errónea, el *análisis de software* se enfoca hacia el análisis del comportamiento de los programas informáticos para abordar propiedades como la seguridad, la corrección o la optimalidad. Dependiendo del tipo de análisis aplicado al software, podremos detectar fragmentos de código potencialmente vulnerables, especificaciones incorrectas, aplicar optimizaciones en base al coste máximo y mínimo de los programas, calcular el consumo de recursos de un programa... Muchas veces este análisis es necesario para cumplir con estándares de calidad de software y con estándares de seguridad, habiéndose desarrollado en los últimos tiempos numerosas metodologías y herramientas para el análisis. Además, la verificación y certificación del software durante el proceso de análisis ayuda a generar programas que cumplan de manera eficiente los requisitos exigidos, comprobando la credibilidad y precisión de los datos aportados y proporcionando certificados que pueden usarse como pruebas formales de este comportamiento verificado.

El capítulo dos de la tesis está dedicado a una generalización del análisis automático de recursos en el ámbito de los *programas abstractos*, programas que contienen símbolos o marcadores para representar instrucciones o expresiones sin especificar. Estos símbolos aparecen habitualmente en reglas de transformación de programas usadas en refactorización, compilación, optimización o paralelización de programas. *Quantitative Abstract Execution* generaliza el análisis automático de recursos a un marco en el cual se manejan programas abstractos para, de esta manera, poder realizar un análisis del efecto de las transformaciones aplicadas a los programas. Este tipo de análisis se basa en componentes complejos que a menudo son externos al análisis de coste, por lo que, además de realizar este análisis de recursos, se añade al proceso de análisis la verificación y certificación de los resultados. De esta manera, la precisión de las cotas es comprobada antes de generar los resultados finales del análisis, generando además pruebas formales que pueden ser usadas en una etapa de certificación de los resultados.

El análisis de recursos utilizado en *Quantitative Abstract Execution* está basado en la inferencia de cotas superiores sobre el coste peor de los programas, ámbito al que se han dedicado la mayoría de los análisis de coste. Esto es debido a su amplia aplicación en el análisis de propiedades de seguridad de los programas que impliquen requisitos de uso,

análisis de aplicaciones de tiempo real donde los programas tienen un tiempo máximo de ejecución, validación de aserciones de eficiencia... Sin embargo, hay otras dos cotas importantes en el contexto del análisis de recursos: las cotas inferiores en el caso mejor y las cotas inferiores en el caso peor. Sobre las cotas inferiores en el caso peor versa el tercer capítulo de esta tesis, en el cual se desarrolla un nuevo método para sintetizar cotas inferiores en el caso peor de bucles enteros no deterministas. Esta técnica está basada en la búsqueda de una función que acota por debajo el número de iteraciones que realiza un bucle en el caso peor. La novedad más importante de esta técnica respecto a las anteriores es la inclusión de especialización de los bucles mediante la adición de restricciones a las posibles ejecuciones. Lo que se persigue con esta especialización es encontrar cotas inferiores más precisas basándose en el hecho de que una cota inferior en el caso mejor del bucle especializado es una cota inferior en el caso peor del bucle original, ya que las restricciones añadidas a lo sumo reducen el espacio de posibles estados de ejecución del programa. La efectividad de esta técnica ha sido comprobada en un amplio conjunto de ejemplos, donde los resultados obtenidos han sido comparados con el sistema LoAT [26, 27], referencia en la obtención de cotas inferiores en el caso peor. En la comparación de los resultados se ha comprobado que esta técnica obtiene resultados iguales o incluso mejores que los de LoAT.

La unificación de *Quantitative Abstract Execution* con la síntesis de cotas inferiores culmina los contenidos aportados por esta tesis en el capítulo cuatro. Una de las aplicaciones más importantes de las cotas inferiores en el caso peor es, junto con las cotas superiores, la inferencia de cotas más exactas sobre el coste. Esto hace la combinación de ambas cotas un resultado deseable, ya que en muchos casos no es posible calcular el coste exacto de un programa pero sí una cota de este coste. La combinación de ambas cotas nos da una información más ajustada sobre el coste exacto que puede tener el programa. *Quantitative Abstract Execution with Upper and Lower Bounds* combina el análisis de coste de cotas inferiores y superiores con la verificación y certificación de los resultados obtenidos. El proceso de verificación da, de nuevo, la información necesaria para obtener cotas e invariantes suficientemente precisos para la certificación de los resultados.

Dentro del marco de esta tesis, los resultados más importantes perseguidos han sido:

- Estudiar la inclusión del análisis de recursos en el contexto de programas abstractos. Para ello, ha sido necesaria la extensión de los invariantes de los bucles para capturar también el coste, llegando así al concepto de *invariantes de coste*, que recalan en costes abstractos de programas y dan lugar a pos-condiciones sobre el coste.

- Desarrollo de un análisis de cotas inferiores del coste en el caso peor mediante funciones que acotan por debajo el número de iteraciones de los bucles y la especialización de los mismos.

- Unificación del análisis de coste de programas abstractos y de la inferencia de cotas inferiores para la consecución de un análisis de programas abstractos más preciso.

Estos objetivos se han visto reflejados en las siguientes publicaciones:

- El trabajo publicado en FASE'21 [11] presenta la técnica de *Quantitative Abstract Execution*, donde se parte del analizador de coste COSTA [7] y el sistema de verificación KeY [43] para obtener y verificar el coste de programas abstractos, proporcionando los mecanismos para la certificación de los resultados. Este trabajo fue nominado al "Best Paper" de EAPLS 2021 junto con otras tres publicaciones del congreso de ETAPS 2021.

- El trabajo publicado en CAV'21 [9] contiene todos los resultados referentes a la inferencia de cotas inferiores en el caso peor de bucles, mediante la especialización de bucles y el uso de un resolutor Max-SMT.

- El trabajo enviado para su publicación en TOSEM (actualmente en proceso de revisión) contiene los resultados más relevantes de la unificación de las dos publicaciones anteriores [11, 9] para la inclusión en el análisis de coste de programas abstractos no solo de cotas superiores sobre el coste sino también de cotas inferiores.

Abstract

Since the beginning of automated computing in the middle of the last century, the development of computer science has been linked to an increasing importance in all areas of the current society. The inclusion of computer science processes in everyday life and, in particular, its inclusion in critical situations, cannot go linked only to the generation of hardware and software, but also to the analysis and verification of all its components. While *hardware analysis* is crucial for the generation and maintenance of the computation infrastructure, as it is able to detect or predict components that can have a wrong behavior, *software analysis* focuses on analyzing the behavior of computer programs to address properties such as security, correctness or optimality. Depending on the type of analysis applied to the software, we can detect potential vulnerabilities in the code, find incorrect specifications, apply optimizations based on the maximum and minimum cost of the programs, calculate the resource consumption of a program... Frequently, this analysis is necessary to fulfill software quality and security standards, which is reflected in the big amount of methodologies and tools developed for analysis, especially in the past years. In addition, the verification and certification of the software during the analysis process helps to generate programs that efficiently fulfill the requirements, checking the credibility and accuracy of the provided data and resulting on formal proofs that can be used as certificates of this verified behavior.

The chapter two of the thesis is devoted to a generalization of automatic resource analysis in the field of *abstract programs*, i.e. programs that contain placeholders to represent unspecified statements or expressions. These placeholders usually appear in program transformation rules used in refactoring, compilation, optimization or parallelization. *Quantitative Abstract Execution* generalizes automatic resource analysis to a framework in which abstract programs are analyzed in order to observe the effect of the transformations applied to the programs. This type of analysis, however, requires very precise cost estimations that are not always generated by a cost analysis, so in addition to performing the resource analysis, the framework completes the process including verification and certification. This way, the precision of the inferred bounds is checked before generating the final results of the analysis, which include formal proofs that can be used in a certification stage.

The resource analysis used in Quantitative Abstract Execution is based on the inference of upper bounds on the worst-case cost of the programs; that are the bounds on which most of the cost analysis have focused. This is due to its great application in the analysis of security properties of programs that involve usage requirements, real-time application analysis where programs have a maximum execution time, validation of efficiency assertions, etc. However, there are two other important bounds in the context of resource analysis:

lower bounds on the best-case cost and the lower bounds on the worst-case cost. The chapter three of this thesis focuses on lower bounds on the worst-case cost, presenting a new method that is able to synthesize these lower bounds in the case of integer non-deterministic loops. This technique is based on finding a function that under-approximates the number of iterations that a given loop performs at most. The most important novelty of this technique, compared to the previous lower bounds analysis, is the inclusion of specialization of the loops by adding constraints to the possible executions. What is pursued with this specialization is to find more accurate lower bounds relying on that a lower bound in the best-case cost of the specialized loop is a lower bound in the worst-case cost of the original one. This comes from the fact that the specialization can reduce the possible executions but never increase them. The effectiveness of this technique has been proven in a wide set of examples, where the results obtained have been compared with the LoAT system [27, 26], a system of reference for obtaining lower bounds in the worst-case cost. While comparing the results, it has been found that the presented technique obtains results that are equal to or even better than LoAT ones.

The combination of Quantitative Abstract Execution and Lower Bound Synthesis ends the contributions of this thesis in Chapter four. One of the most important applications of lower bounds on the worst-case cost is, together with upper bounds, to give a more accurate cost estimation, as when both bounds match, then we have an exact cost. This fact makes the combination of both bounds a desirable result, since in many cases in cost analysis it is not is possible to calculate the exact cost but we can find a bound on it. *Quantitative Abstract Execution with Upper and Lower Bounds* combines the cost analysis of lower and upper bounds with the verification and certification of the obtained results. The verification process gives, again, the necessary information to obtain bounds and invariants precise enough for the certification of the results.

Within the framework of this thesis, the most important results have been:

- Study of the generalization of resource analysis to the context of abstract programs. To achieve this, it has been necessary to extend the concept of loop invariant to also capture the cost of the programs, which has given rise to the concept of cost invariants. These cost invariants are based on abstract costs of programs, and they give rise to cost postconditions that are used, later, when comparing the effect of program transformations.

- Development of a lower bound on the worst-case cost analysis by means of specialization of loops and functions that under-approximate their number of iterations.

- Combination of cost analysis of abstract programs and inference of lower bounds to achieve a more precise abstract programs resource analysis.

The consecution of these objectives have given rise to the following publications:

- The work published in FASE'21 [11] presents the *Quantitative Abstract Execution* technique, where the cost analyzer COSTA [7] and the KeY verification system [2] are combined to obtain and verify the cost of abstract programs, providing the mechanisms for the certification of the results. This work was proposed for the "Best Paper" of EAPLS 2021 together with other three papers of ETAPS 2021.

- The work published in CAV'21 [9] contains all the results concerning to the inference of lower bounds on the worst-case cost of loops, by means of loop specialization and the use of a Max-SMT solver.
- The work submitted for publication in TOSEM (currently under review) contains the most relevant results of the combination of the two previous publications [9, 11], where cost analysis of abstract programs is not only based on upper bounds but also in lower bounds. This combination gives rise to *Quantitative Abstract Execution with Upper and Lower Bounds.*

Part I

Contents of the thesis

Chapter 1

Introduction

1.1 Static Cost Analysis

Resource usage analysis, also known as cost analysis, aims at determining the number of resources required to safely execute a given program. A resource can be any quantitative aspect of the program, such as memory consumption, runtime, energy, execution steps... We use the term *cost model* to denote any of these aspects. A cost model formally describes how to measure the resource consumption, i.e., the cost that is associated with each execution step.

Cost analysis techniques can be classified into dynamic and static. While dynamic techniques consist in monitoring, testing and evaluating programs during runtime, static techniques analyze the source code of the program without executing it. This difference in the way the analysis is made leads also to a main difference in the results of both analysis: while dynamic analysis results are only valid for a particular input (or set of inputs), static analysis yields a result that is valid for infinite families of inputs, as it examines all possible execution paths. Analyzing all possible paths makes that static analysis can detect situations that might not be taken into account in a dynamic situation, when they are not visible in the set of traces to be analyzed. On the other hand, performing a static cost analysis requires relying on formal methods that mathematically prove the result for all possible inputs, which can become undoable and/or inaccurate in some cases. This fact leads dynamic analysis to be especially useful to detect situations which are too complex to be discovered by static techniques. Both types of analysis can be used complementarily to reach a stronger result. In what follows in this thesis, we will explore concepts related to the field of *static* cost analysis.

1.1.1 Automated Cost Analysis

The first automated analysis was developed in the 70s [45] 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 [17], for Java-like languages [5, 38], for concurrent and distributed languages [32], for probabilistic programs [37, 31], etc.) and to increase their accuracy (see, e.g., [29, 39, 24, 40]).

The outcome of a static analyzer is, typically, a function that maps input values to the cost of executing the program on such inputs. However, rather than having an exact cost, we rely on bounds over the cost, as in most cases it is not possible to find an exact solution but it is possible to find a bound.

In the seminal work of Wegbreit [45], cost analysis is based on establishing recurrence relations that capture the cost of the program and then transforming them into closed-form bounds without recurrences. In [4], it is introduced an automated cost analysis based on *cost relations*, an extended form of standard recurrence relations, which can be applied to more programs than recurrence relations. This analysis is composed of two phases. First, given a program and a cost model, they first produce recursive equations that capture the cost of the program in terms of the size of its input data; these recursive equations are known as *cost relations*. Then, in a second phase, cost relation systems are solved into closed-form formulas known as *upper bounds on the worst-case cost*, shortly named upper bounds, that are upper bounds over the maximal cost of any execution. Cost relations differ from traditional recurrence relations in three points: (1) they allow non-determinism, permitting both non-determinism of the programming language and non-determinism introduced by size abstractions (for example, when having input data that is not numerical, where we could abstract an array to its size); (2) they may involve not only equalities but inequalities; and (3) they can depend on multiple arguments rather than in one. These features are what makes [4] applicable to more programs. To get an upper bound from a cost relation system, the analysis is based on ranking functions, loop invariants, and partial evaluation. This calculus is closely related to termination analysis: given a loop, a *ranking function f* is a positive function that decreases in any two consecutive calls of the cost relation system. The existence of a ranking function is directly related to the existence of an upper bound and, by definition, guarantees the absence of an infinite execution trace. However, finding an upper bound over the cost is stronger than proving termination, e.g., to prove termination we only need to know that a ranking function exists and, to get an upper bound, a concrete ranking function has to be found.

Even if the vast majority of cost analysis techniques have focused on inferring *upper bounds* on the worst-case cost, there exist two other important bounds in cost analysis: *lower bounds on the best-case cost*, i.e., under-approximations of the minimal cost of any execution, and *lower bounds on the worst-case cost*, i.e., under-approximations of the maximal cost of any execution.

[10] presents a method to calculate lower bounds on the best-case cost. This method is a dual process to the calculus of upper-bounds on the worst-case cost presented in [4]. Instead of finding a ranking function that decreases at least by one in each call of the cost relation system, they generate a recurrence relation system and look for a function lb that increases by one in each execution step. Finding a lower bound on the length of any execution trace reduces then to finding a minimum value for lb so that the base-case equation of the recurrence relation system is applicable.

Lower bounds on the worst-case cost have recently been studied as well. In [27, 26], the concept of *metering function* was introduced as a tool for inferring a lower bound of the number of iterations that a given loop can make. Its definition is analogue to that of a ranking function: while a ranking function is expected to decrease at least by one in each iteration, a metering function is required to decrease at most by one in each step. In addition, while ranking functions are expected to be positive, metering functions are required to be non-positive when leaving the loop. Finding a metering function leads us to a lower bound on the worst-case cost, which is closely related to non-termination, just as ranking functions were linked to termination. A program is non-terminating if there are infinite execution traces, or at least one, i.e. there is a non-empty subset of execution states G such that for any state in G can only go to another state in G. This subset is a *witness* of non-termination. An infinite lower bound on the worst-case cost is directly linked to a witness G and an infinite execution trace.

1.1.2 Application Domains

Cost analysis has many applications in computer science, next we classify these applications for each type of bound.

The most common bounds generated by automatic resource analysis are *upper bounds on the worst-case cost*, since having the assurance that no execution of the program will exceed the inferred amount of resources (e.g., time, memory, etc.) has crucial applications in safety-critical contexts (see, for example, [6, 10]). Upper bounds are used in real-time applications analysis, where programs are required to execute within a certain maximum amount of time. They are used in resource bound certification, where security properties can involve resource usage requirements. In the context of program synthesis and optimization, such as partial evaluation, where many programs may be produced in the process, upper bounds can be used to detect the most efficient ones. When dealing with performance debugging and validation, upper bounds can also be used to check assertions about efficiency included in the code of the programs.

In addition to their use in performance debugging, verification and optimization, *lower bounds on the best-case cost* are useful in task parallelization (see, e.g., [23, 10, 24]). Basing on these lower bounds, a task is not parallelized unless its best-case cost is large enough to make it worth it. This is due to the fact that parallelizing a task involves an overhead of performing this parallelization and, then, a lower bound on the best-case cost smaller than the overhead could make the decision of parallelizing the task inefficient.

The third type of bounds that we have explained, *lower bounds on the worst-case cost*, have a main application related to upper bounds: both types of bounds together allow us to infer tighter worst-case cost bounds, as when they coincide it is ensured that we have an exact estimation. For example, we would know precisely the runtime or memory consumption of the most costly executions, which can be crucial in safety-critical applications. Also, when the upper bound is equal to the lower bound, they have been used 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 upper bound and the lower bound differ due to a difference in the delays associated with how many characters are right in the guessed password, this is identified as a potential attack. Lower bounds on the worst-case cost have also a clear application domain in *smart contracts*, where cost amounts to monetary fees. This is the case for predicting the gas usage [47] 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. Besides, lower bounds on the worst-case cost will give us families of inputs that lead to an expensive cost, which could be used to detect performance bugs such as non-terminating programs, as an infinite lower bound on the worst-case cost is directly related to an infinite execution trace.

1.2 Verification and Certification

At the start of the automatic inference of upper bounds, manually written soundness proofs were used for the techniques used in the analysis [4, 7], typically based on the theory of *Abstract Interpretation* [20]. However, it is hard to apply pencil-and-paper proofs to real-size analyzers. Basically, the proofs guarantee that (theoretically) the results are correct, that is, no execution of the program will ever exceed the inferred upper bound. However, the implementation could be buggy, and hence using the inferred bounds for safety critical purposes could be risky. Proof assistants allow us to automatically specify and prove correct resource usage analysis. They provide formal verification of all correctness proofs. In [8], the strengths of both the COSTA analyzer [7] and the KeY verifier [2] were used to formally *verify* the soundness of the inferred bounds, that gave formal guarantees that the results were correct.

Program proving, also known as *deductive software verification*, expresses the correctness of a program by means of mathematical statements. These mathematical statements, the verification conditions, consist of formal proofs of the property to be verified. This requirement gives us another strength in the analysis: these proofs can be used as certificates that can be checked independently, opening this way the possibility of having a *certified static analysis*, that is, an analysis whose validity has been formally proved correct by means of either automated or interactive theorem provers.

Among all possible applications of resource analysis, we find *resource certification*, whereby programs are coupled with information about their resource usage. Adding this information to a program allows deciding whether the program should or should not be run, depending on the number of resources used in the program execution. Programs that are not certified can consume more resources than was expected and, then, could be potentially unsafe. Moreover, certification is crucial for the correctness of quantitative relational properties when analyzing the effect of program transformations, an aspect of big importance in this thesis. Without certification, the inferred bounds might not be precise enough to establish, for example, that a program transformation does not increase the cost. This is only established at the certification stage, where relational properties are formally verified. This need of certification will be explained later in Chapter 2 of the thesis.

1.3 Contributions and Structure

This thesis has a “publication format”. Publications, attached at the end of the work, contain complete information about the proposed techniques. Two of these papers, *Certified Abstract Cost Analysis* [11] and *Lower Bound Synthesis using Loop Specialization and Max-SMT* [9] have been published in the proceedings of highly prestigious international conferences. The work *Certified Cost Bounds for Abstract Programs* is under review in *ACM Transactions on Software Engineering and Methodology (TOSEM).* The summary of the main contributions of this thesis is:

- *Quantitative Abstract Execution.* We present, to the best of our knowledge, the first method to analyze the cost impact of program transformations.

  In [11], we use COSTA [7] and the analysis presented in [4] to compute *upper bounds on the worst-case cost.* The programs that are used in this paper are *abstract programs*, i.e., programs containing placeholders for unspecified statements. To verify the soundness of this work, in paper [11] we generate *abstract cost relation systems*, a generalization of the cost relation systems that were introduced in [4].

  Cost annotations inferred by abstract cost analysis, that is, cost invariants and abstract cost bounds, are automatically *certified* by a deductive verification system, extending the approach reported in [8] to abstract cost and abstract programs. This is possible because the specification (the cost bound) and the loop (cost) invariants are inferred by the cost analyzer—the verification system does not need to generate them.

  Arguing correctness of an abstract cost analysis is complex, because it relies on a number of complex components (e.g. static analysis, inference of invariants...). For this reason alone, it is useful to certify the abstract cost inferred for a given abstract program: during development of the abstract cost analysis reported, several errors in abstract cost computation were detected—analysis of the failed verification attempt gave immediate feedback on the cause.

- *Inference of lower bounds on the worst-case cost.* In [9], we present a new technique to infer lower bounds on the maximal cost of any execution. While the vast majority of techniques have focused on inferring *upper bounds* on the worst-case cost, *lower-bounds on the worst-case cost* have been less studied. We develop a framework to obtain precise lower-bounds on the worst-case cost for different types of loops. This work was motivated by the limitation of the state-of-the-art methods for obtaining these bounds [27, 26], as the techniques introduced in these works suffer from an important loss of information in *multipath* loops (loops with more than one way of making progress while iterating). The technique is based, as in [26, 27], on finding *metering functions*, functions that underestimate the number of iterations of a given loop. To be able to deal also with these multipath loops, we base our approach on transition systems and introduce several semantic specializations of loops that enable the inference of metering functions for complex loops by (1) restricting the input space of the programs, (2) narrowing the constraints of the transitions of the loop (the guards) and (3) narrowing non-deterministic choices.

  We propose a template-based method to automate our technique which is effectively implemented by means of a Max-SMT encoding. Whereas the use of templates is not new [19], our encoding has several novel aspects that are devised to produce better lower-bounds, for example, the addition of conditions that force the solver to look for larger lower-bound functions.

- *Quantitative Abstract Execution for Upper and Lower Bounds* unifies both concepts of [11] and [9] to leverage the certified abstract cost analysis of [11] to a framework that, in addition to dealing with upper bounds on the worst-case cost, is able to verify and certify also lower bounds. As it is explained in [9], our technique to infer lower bounds on the *worst-case* cost for a loop is based on specializing the loop and then finding a lower bound on the *best-case* cost for the specialized loop. This characteristic of the tool gives us the possibility of having a lower bound on the best-case cost if we disable the specializations. This lower bound, then, can be used to assert between which bounds will be the cost of the analyzed program, as all possible resource consumptions of the program will be between a lower bound on the best-case cost and an upper bound on the worst-case cost, that is, between a lower bound on the minimal cost of any execution and an upper bound on the maximal cost of any execution.

To handle the verification of the properties, cost annotations inferred by abstract cost analysis are extended to cover not only the upper bound case, but also the lower bound: for each cost annotation we will have an *upper* annotation and a *lower* annotation. All these specifications are automatically inferred by the cost analyzer.

The remaining of this thesis is structured as follows:

- In Chapter 2, *Quantitative Abstract Execution*, we introduce the results of the paper *Certified Abstract Cost Analysis* [11], in which *Quantitative Abstract Execution* was presented. In this chapter, the main concepts are introduced in Section 2.1 by means of a motivating example, while in Section 2.2 we explore how we produce the cost annotations that are used to verify the analysis. In Section 2.3, we summarize the results of the experiments.

- Chapter 3, Inference of Lower Bounds on the Worst-Case Cost, is based on paper *Lower Bound Synthesis using Loop Specialization and Max-SMT* [9]. In this chapter, a running example of a multipath loop is presented to show the deficiencies in the state-of-the-art methods to compute lower bounds on the worst-case cost. In Section 3.1, we introduce the program representation and introduce the concept of lower bounds on the worst-case cost. In Section 3.2, we present the technique used to infer *local* lower bounds (bounds for single loops) by means of (1) metering functions, (2) specialization of the input states, (3) narrowing of guards and (4) narrowing of non-deterministic choices. In Section 3.3 we briefly introduce how we obtain these metering functions by means of a Max-SMT solver. In Section 3.4, we explain the implementation and the main results of the experiments.

- In Chapter 4, *Certified Abstract Cost Analysis with Upper and Lower Bounds*, we combine both results of [11] (Chapter 2) and [9] (Chapter 3) to present *Certified Cost Bounds for Abstract Programs*, an article that is under revision in the journal *ACM Transactions on Software Engineering and Methodology (TOSEM)*. In Section 4.1 we introduce our approach informally by means of an example, and set up the terminology that is going to be used. Section 4.2 is focused on the automatic inference of cost invariants and, as the final step in the analysis, cost postconditions. Section 4.3 contains the main novelties of the experimental evaluation compared with Chapter 2.

- Chapter 5, *Conclusions and Future Work*, concludes and discusses future work.

- Chapter 6, *List of Publications*, contains the papers that support this thesis. Here, the technical details of the concepts explained in the thesis are contained.

At the end of the thesis, the publications on which the thesis is based are attached.

- *Elvira Albert, Reiner Hähnle, Alicia Merayo, and Dominic Steinhöfel. Certified Abstract Cost Analysis. In Esther Guerra and Mariëlle Stoelinga, editors, Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Proceedings, volume 12649 of Lecture Notes in Computer Science, pages 24–45. Springer, 2021.* Core B, class 2.

  This work presents the *Quantitative Abstract Execution* technique. It was proposed for the "Best Paper" of EAPLS 2021 together with other three papers of ETAPS 2021.

- *Elvira Albert, Samir Genaim, Enrique Martin-Martin, Alicia Merayo, and Albert Rubio. Lower-Bound Synthesis Using Loop Specialization and Max-SMT. Computer Aided Verification- 33rd International Conference, CAV 2021. Proceedings, Part II, volume 12760 of Lecture Notes in Computer Science, pages 863–886. Springer, 2021.* Core A*, class 1.

  This work, published in CAV'21, contains all the results concerning the inference of lower bounds on the worst-case cost of loops.

- *Elvira Albert, Reiner Hähnle, Alicia Merayo, and Dominic Steinhöfel. Certified Cost Bounds for Abstract Programs. Under revision in ACM Transactions on Software Engineering and Methodology (TOSEM).*

  This work contains the most relevant results of the combination of the two previous publications. Here, cost analysis of abstract programs is not only based on upper bounds but also in lower bounds.

Chapter 2

Quantitative Abstract Execution

This chapter, whose complete information can be found in paper *Certified Abstract Cost Analysis* [11] (Chapter 6, page 65), generalizes the Automated Cost Analysis of [4] to an extended framework that can handle programs containing placeholders for unspecified statements. Our work defines and implements an abstract cost analysis to infer abstract cost bounds, such that the implementation consists of an automatic abstract cost analysis tool and an automatic certifier for verifying the correctness of inferred abstract bounds. Both steps are performed within an approach called *Quantitative Abstract Execution.*

As mentioned in the introduction, Automated Cost Analysis [4] is a static analysis that consists of two phases. First, given a program and a cost model, the analysis produces *cost relation systems.* These systems are made up of *cost relations*, recursive equations that capture the cost of the program in terms of the size of its input data. In the second phase, the recurrences that appear in the cost relation systems are resolved, giving rise to *closed-form* formulas known as *upper bounds.* The solving process includes also bounding the maximal number of applications of the recursive equations; this can be bounded by means of ranking functions. Ranking functions, by definition, guarantee that the length of any execution trace cannot exceed its value and, then, they are fundamental to compute an upper bound over the cost.

Cost analysis occupies an interesting middle ground between termination checking and full functional verification in the static program analysis portfolio. The main problem in functional verification is that one has to come up with a functional specification of the intended behavior, as well as with auxiliary specifications including loop invariants and contracts [30]. In contrast, termination is a generic property and it is sufficient to come up with a suitable term order or ranking function [13]. For many programs, termination analysis is vastly easier to automate than verification. The issue is if verification of cost analysis can be automatized.

Computation cost is not a generic property, but it is usually schematic: One fixes a class of cost functions (for example, polynomial) that can be handled. A cost analysis then must come up with parameters (degree, coefficients) that constitute a valid bound (lower, upper, exact) for all inputs of a given program with respect to a cost model (number of instructions, allocated memory, etc.). If this is performed bottom up with respect to a program's call graph, it is possible to *infer* a cost bound for the top-level function of a program. Such a cost expression is often *symbolic*, because it depends on the program's input parameters.

A central technique for inferring symbolic cost of a piece of code with high precision is *symbolic execution* [15, 34]. The main difficulty is to render symbolic execution of loops with symbolic bounds finite. This is achieved with *loop invariants* that generalize the behavior of a loop body: an invariant is valid at the loop head after arbitrarily many iterations. To infer sufficiently strong invariants automatically is generally an unsolved problem in functional verification, but much easier in the context of cost analysis, because invariants do not need to characterize functional behavior: it suffices that they permit to infer schematic cost expressions.

Our work in Quantitative Abstract Execution introduces an important technical innovation: the notion of *cost invariant*. In automated cost analysis, one infers cost bounds often from loop invariants, ranking functions, and size relations computed during symbolic execution [7, 25, 48, 18]. For *abstract* programs, we need a more general concept, namely a loop invariant expressing a *valid abstract cost bound* at the beginning of any iteration. We call this a *cost invariant*. This new concept, apart from being essential in our technique, also increases the modularity of cost analysis, as each loop can be verified and certified separately.

Another contribution of our work is that we have extended cost analysis to handle programs with placeholders for unspecified statements that represent abstract pieces of code. It remains to verify these results functionally, which is done by means of *Abstract Execution* [44], a recent generalization of symbolic execution that allows specifying and verifying relational program properties. Then, the information inferred by abstract cost analysis, i.e. cost invariants and abstract cost bounds, is automatically certified by the deductive verification system KeY [2], completing that way the cost analysis of abstract programs.

Abstract programs occur in program transformation rules used in compilation, optimization, parallelization, refactoring, etc.: Transformations are specified as rules over program schemata which are nothing but abstract programs. If we can perform cost analysis of abstract programs, we can analyze the cost effect of program transformations. Our approach is *the first method to analyze the cost impact of program transformations.*

2.1 Quantitative Abstract Execution by Example

We introduce our approach and terminology informally by means of a motivating example: *Code Motion* [1] is a compiler optimization technique moving a statement not affected by a loop from the beginning of the loop body to before the loop. This code transformation should preserve behavior provided the loop is executed at least once, but can be expected to improve computation effort, i.e. *quantitative* properties of the program, such as execution time and memory consumption: The moved code block is executed just once in the transformed context, leading to less instructions (less energy consumed) and, in case it allocates memory, less memory usage. In the following, we subsume any quantitative aspect of a program under the term *cost* expressed in an unspecified *cost model* that assigns cost values to programming language instructions with the understanding that it can be instantiated to specific cost measures, such as number of instructions, number of allocated bytes, energy consumed, etc.. To use a generic cost model allows us to work with arbitrary quantitative properties.

To formalize code motion as a transformation rule, we describe in- and output of the

Program Before

```
int i = 0;
//@ loop_invariant i ≥ 0 && i ≤ t;
//@ cost_invariant
      i · (acP (t, w) + acQ (t, z) + 2) ;
//@ decreases t − i;
while (i < t) {
   //@ assignable x;
   //@ accessible t,w;
   //@ cost_footprint t,w;
   \abstract_statement P;
   //@ assignable y;
   //@ accessible i,t,y,z;
   //@ cost_footprint t,z;
   \abstract_statement Q;
   i ++;
}
//@ assert \cost ==
      2 + t · (acP (t, w) + acQ (t, z) + 2) ;
```

Program After

```
int i = 0;
//@ assignable x;
//@ accessible t,w;
//@ cost_footprint t,w;
\abstract_statement P;
//@ loop_invariant i ≥ 0 && i ≤ t;
//@ cost_invariant
      i · (acQ (t, z) + 2) ;
//@ decreases t − i;
while (i < t) {
   //@ assignable y;
   //@ accessible i,t,y,z;
   //@ cost_footprint t,z;
   \abstract_statement Q;
   i ++;
}
//@ assert \cost ==
      2 + acP (t, w) + t · (acQ (t, z) + 2) ;
```

Preconditions and Postconditions

Inputs: t, w, x, y, z   Precondition: $t > 0$ Postcondition: **cost_1** $\geq$ **cost_2**

Figure 2.1: Motivating example on relational quantitative properties.

transformation *schematically*. Figure 2.1 depicts such a schema in a language based on JAVA. An *Abstract Statement* with identifier *Id*, declared as

abstract_statement *Id*;

represents an arbitrary concrete statement. It is obviously unsafe to extract arbitrary, possibly non-invariant, code blocks from loops. For this reason, the abstract statement P in question has a *specification* restricting the allowed behavior of its instances. For compatibility with JAVA we base our specification language on the *Java Modeling Language* (JML) [36]. Specifications are attached to code via structured comments that are marked as JML by an "@" symbol. JML keyword "**assignable**" defines the memory locations that may occur in the frame of an AS; similarly, "**accessible**" restricts the footprint.

Figure 2.1 contains further keywords explained below.

The input to Quantitative Abstract Execution is the abstract program to analyze, including annotations (highlighted in light gray in Figure 2.1) that express restrictions on the permitted instances of abstract statements. In addition to the frame and footprint, the *cost footprint* of an abstract statement, denoted with the keyword "**cost_footprint**", is a subset of its footprint listing the locations the cost expressions in abstract statement instances may depend on. In Figure 2.1, the cost footprint of abstract statement Q excludes accessible variables i and y. Annotations highlighted in dark gray are *automatically inferred* by abstract cost analysis and are input for the certifier. As usual, loop invariants (keyword "**loop_invariant**") are needed to describe the behavior of loops with symbolic bounds. The loop invariant in Figure 2.1 allows inferring the final value t of loop counter i after loop termination. To prove termination, the loop *variant* (keyword "**decreases**") is inferred.

So far, this is standard automated cost analysis [7]. The ability to *infer automatically* the remaining annotations represents our main contribution:

Abstract Execution [43, 44] extends symbolic execution by permitting abstract statements to occur in programs. We extend the Abstract Execution framework to Quantitative Abstract Execution by adding cost specifications that extend the specification of an abstract statement with an annotated cost expression. An *abstract cost expression* is a function whose value may depend on any memory location in the cost footprint of the abstract statement it specifies. Each abstract statement P has an associated *abstract cost* function parametric in the locations of its cost footprint, represented by an abstract cost symbol $\mathsf{ac_P}$. The symbol $\mathsf{ac_P}(\mathsf{t},\mathsf{w})$ in the "**assert**" statement in Figure 2.1 can be instantiated with any concrete function parametric in t, w being a valid cost bound for the instance of P. For example, for the instantiation "$\mathsf{P} \equiv$ x=t+1;" the constant function $\mathsf{ac_P}(\mathsf{t},\mathsf{w}) = 1$ is the correct *exact* cost, while $\mathsf{ac_P}(\mathsf{t},\mathsf{w}) = \mathsf{t}$ with $\mathsf{t} \geq 1$ is a correct *upper bound* cost. For simplicity, we only consider normally completing Java code as instances in this work: an instance may not throw an exception, break from a loop, etc. All occurrences of an abstract statement with the same identifier symbol have the same legal instances.

In standard automated cost analysis, provided an initial state and an initial assignment of the variables, a finite execution of the program is linked to a finite execution trace, that is, a complete trace corresponds to a terminating execution. The cost of a program can be computed based on execution traces, by summing up the costs of the executed instructions with respect to the chosen cost model. The generalization to Quantitative Abstract Execution, i.e., the cost of abstract programs, is defined similarly. While a standard cost analysis generalizes over all initial stores, abstract cost analysis generalizes also over all concrete instances of the abstract program.

To realize Quantitative Abstract Execution on top of the existing functional verification layer provided by the Abstract Execution, we translate non-functional (cost) properties to functional ones.

The translation consists of three elements: (1) A global "ghost" variable "cost" (representing keyword "**\cost**") for tracking accumulated cost; (2) explicit encoding of a chosen cost model by suitable ghost setter methods that update this variable; (3) functional loop invariants and method postconditions expressing cost invariants and cost postconditions.

Regarding item (c), we support three kinds of cost specifications. These are, descending in the order of their strength: *exact*, *upper bound*, and *asymptotic* cost. At the analysis stage, it is usually impossible to determine the best match. For this reason, there is merely one **cost_invariant** keyword, not three. However, when translating cost to functional properties, a decision has to be made. A natural strategy is to start with the strongest kind of specification, then proceed towards the weaker ones when a proof fails.

An exact cost invariant has the shape "cost == *expr*", an upper bound on the invariant cost is specified by "cost <= *expr*"; asymptotic cost is expressed by the idiom "asymptotic(cost) <= asymptotic(*expr*)". The function "asymptotic" abstracts from constant symbols in the argument. For example, the (exact) cost postcondition of the abstract

```
//@ ghost int cost = 0;
int  i  = 0;
//@ set cost = cost + 1;
//@ assignable x, cost;
//@ accessible t, w;
//@ ensures cost == \before(cost) +
      ac_P (t, w);
\abstract_statement P;

//@ ghost int iCost = 0;
//@ loop_invariant i ≥ 0 && i ≤ t
//@ && iCost == i · (ac_Q (t, z) + 2) ;
//@ decreases t − i;
while (i  < t) {
    //@ set iCost = iCost + 1;
    //@ \textbf{assignable} y, cost;
    //@ \textbf{accessible} i, t, y, z;
    //@ ensures cost == \before(cost)
    + ac_Q (t, z);
    \abstract_statement Q;
    i ++;
    //@ set iCost = iCost + 1;
}
//@ set cost = cost + 1;
//@ set cost = cost + iCost;
```

Listing 2.2: Translation of cost model and cost invariants to Abstract Execution.

program on the right in Figure 2.1 is:

$$2 + \mathsf{ac_P}\,(\mathsf{t},\mathsf{w}) + \mathsf{t} \cdot (\mathsf{ac_Q}\,(\mathsf{t},\mathsf{z}) + 2) \tag{2.1}$$

Asymptotic cost would be expressed as $\mathsf{asymptotic(cost)}\ \ <= \mathsf{asymptotic}(2 + \mathsf{ac_P}\,(\mathsf{t},\mathsf{w}) + \mathsf{t} \cdot (\mathsf{ac_Q}\,(\mathsf{t},\mathsf{z}) + 2))$ where the right-hand side of the equation is equivalent to
$\mathsf{asymptotic}(\mathsf{ac_P}\,(\mathsf{t},\mathsf{w}) + \mathsf{t} \cdot (\mathsf{ac_Q}\,(\mathsf{t},\mathsf{z})))$.

Figure 2.2 shows the result of translating the cost invariant in Figure 2.1 to a functional loop invariant (highlighted lines), using cost model $\mathcal{M}_{\mathsf{instr}}$ in ghost setters and postconditions of abstract statement "**ensures**" clauses). Abstract statements P, Q must include the ghost variable "cost" in their frame, because they update its value. The keyword **before** in the postcondition of an abstract statement refers to the value a variable had just before executing the abstract statement. In loops, we use "inner" cost variables "iCost" tracking the cost inside the loop. When the loop terminates, we add the final value of "iCost" to "cost". After every evaluation of the guard of the loop, the cost is incremented accordingly. Using the translation in Figure 2.2 of the inferred annotations in Figure 2.1, the Abstract Execution system proves cost postcondition 2.1 automatically.

Apart from the translation of inferred quantitative annotations to functional AE specifications, we implemented the axiomatization of the asymptotic function and extended the AE system's *proof script* language. This made it possible to define a highly automated proof strategy for non-linear arithmetic problems generated by some cost analysis benchmarks.

As pointed out at the beginning of this chapter, we require *cost invariants* to capture the cost of each loop iteration. They are declared by the keyword "**cost_invariant**". To generate them, it is necessary to infer the *cost growth* of abstract programs that bounds the number of loop iterations executed so far.

In Section 2.2 we describe the automated inference of cost invariants including the generation of cost growth for all loops. Our technique is compositional and also works in the presence of nested loops.

2.2 Abstract Cost Analysis

We need to infer annotations for abstract cost invariants and cost postconditions. To achieve this, we leverage a cost analysis framework for concrete programs to the abstract setting.

As mentioned previously, the automated cost analysis in [4] performs the analysis in two phases: it first generates a cost relation system and, then, it eliminates recurrences to get a closed-form formula. In this section, we follow a similar procedure: in Section 2.2.1 we first define the notion of an abstract cost relation system (ACRS) used in cost analysis for the abstract setting. Then, the automatic generation of inductive cost invariants for abstract programs from ACRSs is defined in Section 2.2.2. Finally, Section 2.2.3 presents how to generate the cost postconditions used to prove relational properties and required to handle nested loops.

2.2.1 Inference of Abstract Cost Relations

There are two main cost analysis approaches: those using recurrence equations in the style of Wegbreit [46], and those based on type systems [22, 33]. Our formalization is based on the first kind, but the main ideas for extending the framework to abstract programs would be also applicable to the second. The key issue when extending a recurrences-based framework to the abstract setting is the notion of *abstract cost relation* for loops which generalizes the concept of cost recurrence equations for a loop to an abstract setting. We start with notation for loops and technical details on assumed size relations.

```
while (G) {
  //@ accessible $r_{1,1}, \dots, r_{1,h_{r1}}$
  //@ assignable $w_{1,1}, \dots, w_{1,h_{w1}}$
  //@ cost_footprint $c_{1,1}, \dots, c_{1,h_{c1}}$
  \abstract_statement $A_1$;
  non_abstract_statement  $N_1$;
  ...
}
```

Loops. In our formalization we consider while-loops containing n abstract statements and m non-abstract statements. Non-abstract statements include any concrete instruction of the target language (arithmetic instructions, conditionals, method calls, ...). We assume loops L have the general outline displayed on the right. Each abstract statement has a frame specification, abstract and non-abstract statements may appear in any order, either might be empty.

Size relations. We assume that for each loop sets of *size constraints* have been computed. These sets capture the size relation among the variables in the loop upon exit (called *base case*, denoted φ_B), and when moving from one iteration to the next (denoted φ_I). Abstract statements are ignored by the size analysis. While this would be unsound in general, it will be correct under the requirements we impose in Definition 2 and with the handling of abstract statements in Definition 1. Size relations are available from any cost analyzer by means of a static analysis [21] that records the effect of concrete program statements on variables and propagates it through each loop iteration. In our examples, since we work on integer data, size analysis corresponds to a value analysis [16] tracking the value of the integer variables.[1]

[1]For complex data structures, one would need heap analyses [41] to infer size relations.

Example 1. *The size relations for the loop on the left in 2.1 are* $\varphi_B = \{\mathsf{i} \geq \mathsf{t}\}$ *and* $\varphi_I = \{\mathsf{i} < \mathsf{t},\ \mathsf{i}' = \mathsf{i} + 1\}$*.* φ_B *is inferred from the loop guard and* φ_I *from the guard and the increment of i (primed variables refer to the value of the variable after the loop execution).*

Based on pre-computed size relations, we define the cost of executing a loop by means of an *abstract cost relation system* (ACRS). This is a set of cost equations characterizing the abstract cost of executing a loop for any input with respect to a given cost model $\mathcal{M}$. Cost equations consist of a cost expression governed by size constraints containing applicability conditions for the equation (like $\mathsf{i} < \mathsf{t}$ in φ_I above) and size relations between loop variables (like $\mathsf{i}' = \mathsf{i} + 1$ in φ_I).

Definition 1 (Abstract Cost Relation System)**.** *Let L be a loop as above with n abstract and m non-abstract statements. Let* $\overline{x}$ *be the set of variables accessed in L. Let* φ_I*,* φ_B *be sound size relations for L, and* $\mathcal{M}$ *a cost model. The ACRS for L is defined as the following set of cost equations:*

$$\begin{array}{ll} C(\overline{x}) = \mathtt{C_B} & ,\ \varphi_B \\ C(\overline{x}) = \sum_{j=1}^{n} \mathsf{ac_j}\left(c_{j,1}, \ldots, c_{j,h_{cj}}\right) + \sum_{i=1}^{m} \mathtt{C_{N_i}} + C(\overline{x}'), & \varphi_I \end{array}$$

where:

1. $\mathtt{C_B} \geq 0$ *is the cost of exiting the loop (executing the base case) w.r.t.* $\mathcal{M}$*.*

2. *Each* $\mathsf{ac_j}(\cdot) \geq 0$ *represents the abstract cost for the abstract statement* A_j *in L w.r.t. to* $\mathcal{M}$*. Each* $\mathsf{ac_j}$ *is parameterized with the variables in the cost footprint of the corresponding* A_j*, as it may depend on any of them.*

3. *Each* $\mathtt{C_{N_i}} \geq 0$ *is the cost of the non-abstract statement* N_i *w.r.t. to* $\mathcal{M}$*.*

4. *C is a recursive call.*

5. $\overline{x}'$ *are variables* $\overline{x}$ *when renamed after executing the loop.*

6. *The assignable variables* $w_{j,*}$ *in the* $\mathsf{ac_j}$ *get an unknown value in* $\overline{x}'$ *(denoted with "_" in the examples below).*

Ignoring the abstract statements, one can apply a complete algorithm for cost relation systems [13] to an ACRS to obtain automatically a *linear* ranking function f for loop L: f is a linear, non-negative function over $\overline{x}$ that decreases strictly at every loop iteration. Function f yields directly the "//@ **decreases** f;" annotation required for Quantitative Abstract Execution.

As in Section 2.1, the definition of ACRS assumes a generic cost model $\mathcal{M}$ and uses `C` to refer in a generic way to cost according to $\mathcal{M}$. For example, to infer the number of executed steps, `C` is 1 per instruction, while for memory usage `C` refers to the amount of memory allocated by an instruction.

Example 2. *The ACRSs of the programs in 2.1 are (left program above line, right program* *below):*

$$
\begin{array}{rll}
C_{\mathit{before}}(\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) &= c_{\mathit{before}} + C_{w_0}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}), & \{\mathsf{i}=0\}\\
C_{w_0}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) &= c_{B_{w_0}}, & \{\mathsf{i}\geq \mathsf{t}\}\\
C_{w_0}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) &= c_{w_0} + \mathsf{ac}_{\mathsf{P}}(\mathsf{t},\mathsf{w}) + \mathsf{ac}_{\mathsf{Q}}(\mathsf{t},\mathsf{z}) + C_{w_0}(\mathsf{i}',\mathsf{t},\_,\mathsf{w},\_,\mathsf{z}), & \{\mathsf{i}'=\mathsf{i}+1,\mathsf{i}<\mathsf{t}\}\\
\hline
C_{\mathit{after}}(\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) &= c_{\mathit{after}} + \mathsf{ac}_{\mathsf{P}}(\mathsf{t},\mathsf{w}) + C_{w_1}(\mathsf{i},\mathsf{t},\_,\mathsf{w},\mathsf{y},\mathsf{z}), & \{\mathsf{i}=0\}\\
C_{w_1}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) &= c_{B_{w_1}}, & \{\mathsf{i}\geq \mathsf{t}\}\\
C_{w_1}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) &= c_{w_1} + \mathsf{ac}_{\mathsf{Q}}(\mathsf{t},\mathsf{z}) + C_{w_1}(\mathsf{i}',\mathsf{t},\mathsf{x},\mathsf{w},\_,\mathsf{z}), & \{\mathsf{i}'=\mathsf{i}+1,\mathsf{i}<\mathsf{t}\}
\end{array}
$$

Notation c refers to the generic cost that can be instantiated to a chosen cost model $\mathcal{M}$. Cost equation C_{before} for the first program is composed of the instructions appearing before the loop is c_{before} plus the cost of executing the while loop C_{w_0}. The size constraint fixes the initial value of i. *Following Definition 1, there are two equations corresponding to the base case of the loop and executing one iteration, respectively. Observe that assignable variables in abstract statements have unknown values in the ACRS (according to 6 in Definition 1). Program after has a similar structure. A ranking function for both loops is* $\mathsf{t}-\mathsf{i}$ *which is used to generate the annotation "//@* **decreases** *t−i;" inserted just before each loop in 2.1.*

To guarantee soundness of abstract cost analysis, it is mandatory that

1. no abstract statement in the loop modifies any of the variables that influence loop cost, i.e., they do not *interfere with cost*, and
2. the cost of the abstract statement in the loop is independent of the variables modified in the loop. We call the latter abstract statements *cost neutral.*

The first requirement is guaranteed by item 6 in Definition 1, because the value of assignable variables is "forgotten" in the equations. It is implemented, as usual in static analysis, by using a name generator for *fresh* variables. If cost depends on assignable variables in an abstract statement, then the ACRS will not be solvable (i.e., the analysis returns "unbound cost"). The ACRS in the example contains "_" in equations that do not prevent solvability of the system nor its evaluation, because they do not interfere with cost. However, if we had "forgotten" a cost-relevant variable (such as `t`), we would be unable to solve or evaluate the equations: without knowing `t` the equation guard is not evaluable. Requirement 2 is ensured by the following definition that guarantees that variables in the cost footprint are not modified by other statements in the loop.

Definition 2 (Cost neutral abstract statement)**.** *Given a loop L, where*

- *$W(L)$ is the set of variables written by the non-abstract statements of L.*
- `Abstr`(L) *is the set of all abstract statements in loop L.*
- *Frame*$(\texttt{Abstr}(L))$ *is the set of variables assigned by any abstract statement $A \in$* `Abstr`(L)*.*
- *CostFootprint*(A) *is the set of variables which the cost of an A depends on.*

L is a loop with cost neutral *abstract statements if, for all $A \in \texttt{Abstr}(L)$, we have that $(W(L) \cup \mathit{Frame}(\texttt{Abstr}(L))) \cap \mathit{CostFootprint}(A) = \emptyset$.*

Example 3. *It is easy to check that both loops in 2.1 have cost neutral ASs. On the left:* $W(L) = \{\mathsf{i}\}$, $\mathit{Frame}(\{P, Q\}) = \{\mathsf{x}, \mathsf{y}\}$, $\mathit{CostFootprint}(P) = \{\mathsf{t}, \mathsf{w}\}$, *and* $\mathit{CostFootprint}(Q) = \{\mathsf{t}, \mathsf{z}\}$, *so* $(W(L) \cup \mathit{Frame}(\{P, Q\})) \cap \mathit{CostFootprint}(P) = \emptyset$, *and* $(W(L) \cup \mathit{Frame}(\{P, Q\})) \cap \mathit{CostFootprint}(Q) = \emptyset$. *The program on the right is checked analogously.*

2.2.2 From Abstract Cost Relation Systems to Abstract Cost Invariants

To enable automated Quantitative Abstract Execution, we need to obtain from them *closed-form* cost invariants and postconditions, i.e., non-recursive expressions. We introduce the novel concept of *abstract cost invariant* that enables automated, inductive proofs over cost in a deductive verification system. The crucial difference to (non-inductive) cost postconditions as inferred by existing cost analyzers is that abstract cost invariants can be proven inductively for each loop iteration. Hence, they integrate naturally into deductive verification systems that use loop invariants [30].

In contrast to abstract cost invariants, postconditions provide a bound for the cost *after* execution of the *whole* loop they refer to. Typically, a postcondition bound for a loop has the form $\mathit{max\_iter} * \mathit{max\_cost} + \mathit{max\_base}$, where $\mathit{max\_iter}$ is the maximal number of iterations of the loop, $\mathit{max\_cost}$ is the maximal cost of any loop iteration, and $\mathit{max\_base}$ is the maximal cost of executing the loop with no iterations. Instead, an abstract cost invariant has the form $\mathit{growth} * \mathit{max\_cost} + \mathit{max\_base}$, where *growth* counts how many times the loop has been executed and hence provides a bound after *each* loop iteration. The challenge is to design an automated technique that infers *growth*. We propose to obtain it from the ranking function:

Definition 3 (Growth)**.** *Given a loop with ranking function* $F = c + \sum_i a_i \cdot v_i$, *where* c *and* v_i *are the constant and variable parts of the function, respectively, and* a_i *are constant coefficients. If we denote with* v_i^0 *the initial value of variable* v_i *before entering the loop, then* $\mathit{growth} = \sum_i a_i \cdot \left(v_i^0 - v_i\right)$.

Example 4. *We look at four simple loops with ranking function decreases and the* growth *inferred automatically by applying 3:*

int *i = 0;* **while** *(i < t)* *i++;*	**int** *i = t;* **while** *(i > 0)* *i−−;*	**int** *i = 0;* **while** *(i < t)* *i += 2;*	**int** *i = t;* **while** *(i > 0)* *i −= 2;*
decreases $\mathsf{t}-\mathsf{i}$	*decreases* i	*decreases* $\frac{\mathsf{t}-\mathsf{i}+1}{2}$	*decreases* $\frac{\mathsf{i}+1}{2}$
growth i	*growth* $\mathsf{t}-\mathsf{i}$	*growth* $\frac{\mathsf{i}}{2}$	*growth* $\frac{\mathsf{t}-\mathsf{i}}{2}$

We can now define the concept of abstract cost invariant that relies on abstract cost relations defined in 2.2.1 and growth as defined above.

Definition 4 (Abstract Cost Invariant)**.** *Given an ACRS as in Definition 1 and its growth as in Definition 3, an* abstract cost invariant *is defined as follows:* $\mathtt{cinv}(\overline{x}) = \mathtt{C_B}^{\mathtt{max}} + \mathit{growth} \cdot \left(\sum_{j=1}^{n} \mathsf{ac_j}\left(c_{j,1}, \ldots, c_{j,h_{cj}}\right) + \sum_{i=1}^{m} \mathtt{C_{N_i}}^{\mathtt{max}}\right)$ *where* $\mathtt{C_B}^{\mathtt{max}}$ *stands for the maximal value that the expression* $\mathtt{C_B}$ *can take under the constraints* φ_B, *and* $\mathtt{C_{N_i}}^{\mathtt{max}}$ *the maximal value of* $\mathtt{C_{N_i}}$ *under* φ_I. *We generate the annotation*
"//@ **cost_invariant** $\mathtt{cinv}(\overline{x})$*;".*

To obtain the maximal cost of a cost expression under a set of constraints, we use existing maximization procedures [12].

From Definition 4 we obtain abstract cost invariants as closed-form abstract cost expressions of the form `abexpr` = `cexpr` | `ac` | $\texttt{abexpr}_1$+$\texttt{abexpr}_2$ | $\texttt{abexpr}_1$∗$\texttt{abexpr}_2$ where `ac` represents an abstract cost function as defined in Section 2.1 and `cexpr` is a concrete cost expression. The definition above yields linear bounds, however, the extension to infer postconditions in the subsequent section leads to polynomial expressions (of arbitrary degree).[2]

Example 5 (Abstract Cost Invariant). *Consider the first loop in Example 4 (where growth = i) with the following frame and footprint:*

```
//@ assignable j;
//@ accessible i,t,j,k;
//@ cost_footprint k;
```

Using $\mathcal{M}_{\mathsf{instr}}$, the evaluation of the loop guard and the increase of i *both have unit cost, so the ACRS is:*

$$\begin{array}{ll} C(\mathsf{i},\mathsf{t},\mathsf{j},\mathsf{k}) = 1 & \{\mathsf{i} \geq \mathsf{t}\} \\ C(\mathsf{i},\mathsf{t},\mathsf{j},\mathsf{k}) = \mathsf{ac}_\mathsf{P}\,(\mathsf{k}) + 2 + C(\mathsf{i}',\mathsf{t},\_,\mathsf{k}) & \{\mathsf{i}' = \mathsf{i}+1,\ \mathsf{i} < \mathsf{t}\} \end{array}$$

The value of the assignable variable j *in the recursive call is "forgotten" (item 6 in Definition 1), but this information loss does not affect solvability of the ACRS. We obtain the following abstract cost invariant: "*`//@ cost_invariant 1 + i * (2 + acP(k));`*".*

Example 6 (Upper Bound Abstract Cost Invariant). *Sometimes an Abstract Cost Invariant is over-approximating cost, resulting in an* upper bound Abstract Cost Invariant. *To illustrate this, we add an instruction that creates an array of non-constant size "*i*" to the program in the previous example and measure memory consumption instead of instruction count.*

```
while (i  < t) {
    a = new int[i];
    //@ assignable j;
    //@ accessible i,t,j,a,k;
    //@ cost_footprint k;
    \abstract_statement P;
    i ++;
}
```

The resulting ACRS thus accumulates cost "i" at each iteration, plus the memory consumed by the abstract statement:

$$\begin{array}{ll} C(\mathsf{i},\mathsf{t},\mathsf{j},\mathsf{k}) = 0, & \{\mathsf{i} \geq \mathsf{t}\} \\ C(\mathsf{i},\mathsf{t},\mathsf{j},\mathsf{k}) = \mathsf{ac}_\mathsf{P}\,(\mathsf{k}) + \mathsf{i} + C(\mathsf{i}',\mathsf{t},\_,\mathsf{k}), & \{\mathsf{i}' = \mathsf{i}+1, \mathsf{i} < \mathsf{t}\} \end{array}$$

Now, maximizing the expression $\texttt{C}_{\texttt{N}_1} = \mathsf{i}$ under $\{\mathsf{i}' = \mathsf{i}+1, \mathsf{i} < \mathsf{t}\}$ results in ${\texttt{C}_{\texttt{N}_1}}^{\texttt{max}} = \mathsf{t}-1$ and upper bound abstract cost invariant

"`//@ cost_invariant i * (t - 1 + acP(k));`".

[2]As our approach is based on a recurrences-based framework [46] that works for exponential and logarithmic expressions, the results in this section generalize to these expressions. However, the Abstract Execution deductive verification system is not able to deal with them automatically at the moment, so we skip these expressions in our account.

2.2.3 From Cost Invariants to Postconditions

To handle programs with nested loops and to prove relational properties it is necessary to infer *cost postconditions* for abstract programs. For nested loops, the cost postcondition states the abstract cost after complete execution of the inner loop and it is used to compute the invariant of the outer loop. For relational properties, the cost postconditions of two abstract programs are compared. Cost postconditions for concrete programs are obtained by upper bound solvers (e.g., COSTA [7], CoFloCo [25], AProVE [28]) that compute *max_iter*, an upper bound on the number of iterations that a loop performs. To do so, one relies on ranking functions. We do this as well, but generalize the computation of postconditions to abstract programs. The cost postcondition is obtained by substituting `growth` by `max_iter` in the formula of $\texttt{cinv}(\overline{x})$ in 4 as follows.

Definition 5 (Cost Postcondition)**.** *Let L be a loop, max_iter be an upper bound on the number of iterations of L. Given the ACRS for L in 1, we infer the cost postcondition for L as*

$$post(\overline{x}) = \mathtt{C_B}^{\mathtt{max}} + max\_iter(\overline{x}) \cdot \left(\sum_{j=1}^{n} \mathsf{ac_j}\left(c_{j,1}, \ldots, c_{j,h_{cj}}\right) + \sum_{i=1}^{m} \mathtt{C_{N_i}}^{\mathtt{max}} \right)$$

and generate the annotation "//@ **assert** *cost $==post(\overline{x})$;".*

To infer the postcondition for a complete abstract program, we take the sum of all *cost postconditions* of its top-level loops plus the cost of the non-iterative fragments. 2.1 shows the cost postconditions for our running example obtained by replacing in the abstract cost invariant the growth i of the invariant with the bound t on the loop iterations and requiring $\mathsf{t} \geq 0$. The generation of inductive abstract cost invariants for nested loops uses the cost postcondition of inner loops to compute the invariants of the outer ones.

2.3 Experimental Evaluation

The implemented prototype is a command-line implementation backed by an existing cost analysis library for (non-abstract) Java bytecode as well as the deductive verification system KeY [2] including the Abstract Execution framework [37, 38]. Our implementation consists of three components: (1) An extension of a cost analyzer (written in Python) to handle abstract Java programs, (2) a conversion tool (written in Java) translating the output of the analyzer to a set of input files for KeY, (3) a bash script orchestrating the whole toolchain, specifically, the interplay between item (1), item (2) and the two libraries. In case of a failed certification attempt, the script offers the choice to open the generated proof in KeY for further debugging.

To assess effectiveness and efficiency of our approach, we used the Quantitative Abstract Execution implementation to analyze seven typical code optimization rules. The set of experiments is composed by: (1) A loop unrolling transformation duplicating the body of a loop, where each copy of the body is put inside a conditional guarded by the loop guard. In this example it was needed to switch to asymptotic analysis: the cost analyzer over-approximates the number of iterations of the unrolled loop, since there are different possible control flows in the body. This was automatically detected by the certifier which failed to find a proof when exact cost invariants are conjectured and succeeds with asymptotic ones. (2) The CodeMotion example from Section 2.1 where the gain in efficiency is proven. The result reflects the cost decrease in the sense that less instructions need to be executed by the transformed program (3) A loop tiling optimization at compiler level in which a single loop with $n \cdot m$ iterations is transformed into two nested loops, an outer one looping until n and an inner one until m. Since our cost analyzer only handles linear size expressions, the first program is written using an auxiliary parameter t that is then instantiated to value $n \cdot m$ (4) A transformation splitting a loop with two independent parts into two separate loops. We prove that this transformation does not affect the cost up to a constant factor. (5) A combination of two loops with the same body structure into one loop. (6) A three loops example, with a nested and a simple one, that are combined into on nested loop. (7) An array optimization, where an array declaration is moved in front of a loop, initializing it with an auxiliary parameter that is the sum of all the initial sizes.

To compute these results, two different cost models were used: number of instructions and memory consumption. Even if there are more possible cost models, such as the number of calls, we opted by performing the analysis using always the cost model of number of instructions as it is the most used in practice. However, in the array example, as the transformation is an optimization in terms of heap, the cost model of memory consumption was used to better understand the effect of the transformation. In the complete data table, that is available in *Experimental Evaluation* Section of paper *Certified Abstract Cost Analysis* (Chapter 6, page 65), there is information about the type of bound yield by the analysis (exact, upper or asymptotic), the growths and cost postconditions that were inferred for each example, and different metrics about the analysis execution times and the size of the generated proof.

Even though the time needed for certification is significantly higher than for cost analysis (which is to be expected), each analysis can be performed within one minute. The time to check a proof certificate amounts to approximately one fourth to one third of the time needed to generate it. We stress that all analyses are fully automatic.

Chapter 3

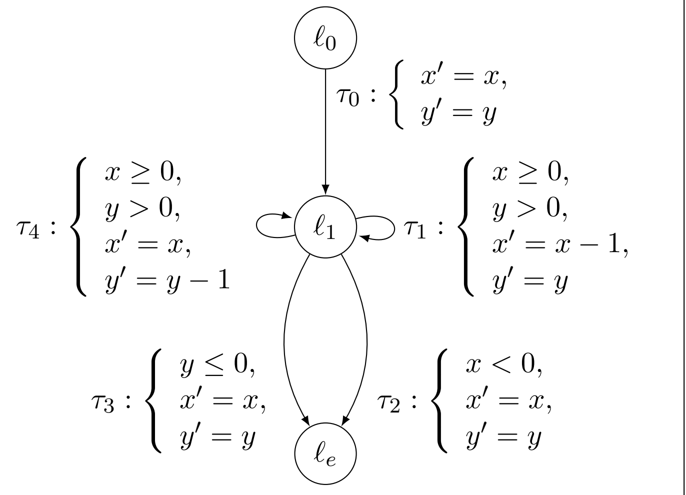
Inference of Lower Bounds on the Worst-Case Cost

The vast majority of cost analysis approaches have focused on inferring *upper bounds on the worst-case cost*, which bound the worst-case from above. Chapter 2 is indeed developed for upper bounds on the worst-case cost. In this chapter of the thesis, whose complete information can be found in paper *Lower Bound Synthesis using Loop Specialization and Max-SMT* [9] (Chapter 6, page 87), we focus on inferring another type of important bounds: *lower bounds on the worst-case cost*, which bound the worst-case cost from below.

In lower bounds analysis, apart from the worst-case cost bounds, we find another type of bounds: *lower bounds on the best-case cost*, that characterize the minimal cost of any program execution. An important difference between lower bounds on the worst-case cost and lower bounds on the best-case cost is that, while the best-case must consider *all* program runs, the worst-case holds for (usually infinite) families of the most expensive program executions. This is why the techniques applicable to the best-case inference (e.g., [23, 10, 24]) are not useful for the worst-case in general, since they would provide too inaccurate (low) results. The state-of-the-art in lower bounds on the worst-case inference is [27, 26] (implemented in the LoAT system) which introduces a variation of ranking functions, called *metering functions*, to under-estimate 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.

This work is motivated by the limitation of state-of-the-art methods when, by treating each simple loop separately, a lower bound on the worst-case cannot be found or it is too imprecise. For example, consider the interleaved loop in Figure 3.1, which 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 in which the transitions τ are labelled 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

```
while (x >= 0 && y > 0) {
  if (*) {
    x = x - 1;
  } else {
    y = y - 1;
  }
}
```



ℓ_0
$\tau_0 : \begin{cases} x' = x, \\ y' = y \end{cases}$
$\tau_4 : \begin{cases} x \geq 0, \\ y > 0, \\ x' = x, \\ y' = y - 1 \end{cases}$
ℓ_1
$\tau_1 : \begin{cases} x \geq 0, \\ y > 0, \\ x' = x - 1, \\ y' = y \end{cases}$
$\tau_3 : \begin{cases} y \leq 0, \\ x' = x, \\ y' = y \end{cases}$
$\tau_2 : \begin{cases} x < 0, \\ x' = x, \\ y' = y \end{cases}$
ℓ_e

Figure 3.1: Interleaved loop (left) and its representation as a transition system (right)

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 [27, 26] accelerates independently both τ_1 and τ_4, resulting in accelerated versions

$$\tau_1^a \ : \begin{cases} x \geq -1, \\ y > 0, \\ x' = -1, \\ y' = y \end{cases} \qquad\qquad \tau_4^a \ : \begin{cases} x \geq 0, \\ y \geq 0, \\ x' = x, \\ y' = 0 \end{cases}$$

with cost $x + 1$ 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 worst-case lower bound is $x + y$ iterations.

Our challenge for inferring more precise lower bounds on the worst-case 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.

This work is inspired by [35], 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 [42]. 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 work leverages such powerful use of quasi-invariants and Max-SMT in non-termination analysis to the more difficult problem of lower bounds on the worst-case inference. Non-termination and worst-case lower bounds are indeed related properties: in both cases we need to find witnesses, respectively, for non-terminating the loop and for executing at least a certain number of iterations. For lower bounds on the worst-case, we additionally need to provide such under-estimation for the number of iterations and search for lower bounds on the worst-case behaviors that occur for a class of inputs rather than for a single input instantiation, since the worst-case lower bound for a single input is a concrete (constant) cost, rather than a parametric lower bound on the worst-case cost function as we are searching for. Instead, for non-termination, it is enough to find a non-terminating input instantiation.

A fundamental idea of our approach is to *specialize* loops in order to guide the search of the metering functions of complex loops, in an analogous way as it guides the search of a non-termination witness in [35], avoiding the inaccuracy introduced by disconnecting them into simple loops. The idea is that a lower bound on the best-case of the specialized loop is a lower bound on the worst-case of the general loop. 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 Figure 3.1, our approach automatically infers templates to narrow 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, that is, its cost.

3.1 Background

This section introduces some notation on the program representation and recalls the notion of lower bound on the worst-case cost that we aim at inferring.

3.1.1 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:=nondet();", where x is a program variable and nondet() can be represented by a fresh non-deterministic variable u. This assumption allows us to also cover non-deterministic branching, for example, "if (*){..} else {..}" as it can be expressed by introducing a non-deterministic variable u and rewriting the code as "u:=nondet(); if (u≥0){..} else {..}".

Our programs are represented using *transition systems*, in particular using the formalization of [35] that simplifies the presentation of some formal aspects of our work. Throughout this chapter, we represent a transition system as a control flow graph, and analyze its strongly connected components one by one.

In what follows, we formalize the rules of our technique using the notation: (1) $\bar{x}$ is a tuple of n integer program variables, (2) $\bar{u}$ is a tuple of integer non-deterministic variables, (3) $\mathcal{L}$ is the set of locations (nodes) in the transition system (control flow graph), (4) a transition is of the form $(\ell, \ell', \mathcal{R})$ such that $\ell, \ell' \in \mathcal{L}$ are locations in the transition system and $\mathcal{R}$ is a formula over $\bar{x}$, $\bar{u}$ and $\bar{x}'$, where $\bar{x}'$ represents the values of the unprimed corresponding variables after the transition. We use $\mathcal{R}(\bar{x})$ to refer to the constraints that involve only variables $\bar{x}$ (the *guard* of the transition), and use $\mathcal{R}(\bar{x}, \bar{u})$ to refer to the constraints that involve only non-deterministic variables $\bar{u}$ and (possibly) $\bar{x}$. In addition, 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 7. *The set of transitions of the control flow graph in Figure 3.1 is:*

$$\begin{array}{l}\{(\ell_0, \ell_1, x' = x \wedge y' = y),\\ \ (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x - 1 \wedge y' = y),\\ \ (\ell_1, \ell_e, x < 0 \wedge x' = x \wedge y' = y),\\ \ (\ell_1, \ell_e, y \leq 0 \wedge x' = x \wedge y' = y),\\ \ (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x \wedge y' = y - 1)\}\end{array}$$

A requirement to handle non-determinism is that for any state $\bar{x}$ satisfying the guard, there are values for the non-deterministic variables $\bar{u}$ such that we can make progress. A transition that holds (respectively does not hold) this condition is called a *valid transition* (respectively *invalid* transition).

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 $\sigma(\overline{x})$, 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 $(\ell, \ell', \mathcal{R})$ such that there exists an assignment $\overline{u}$ for the non-deterministic variables so that the constraints of $\mathcal{R}$ are fulfilled by the values of $\overline{u}$, σ_1 and σ_2' (σ_2 with the prime variables defined in $\mathcal{R}$).

A valid trace t is a possibly infinite sequence of configurations $(\ell_0, \sigma_0), (\ell_1, \sigma_1), \ldots$ such that 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 we reach a state where no outgoing transition can be executed, that is, where no guard of the outgoing transitions is fulfilled by the values defined in σ.

A transition system is non-blocking if no trace includes a blocking configuration. We assume that the transition system under consideration is non-blocking, and thus any trace is a prefix of a complete one.

3.1.2 Lower Bounds

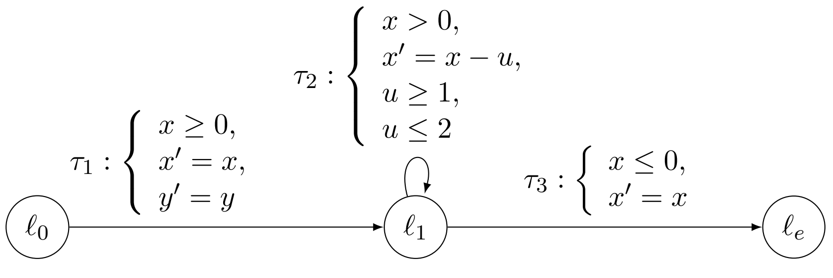
For simplicity, we assume that an execution step (a transition) costs 1. Under this assumption, the cost of a trace t is simply its length, where the length of an infinite trace is ∞. In what follows, the set of all configurations is denoted by $\mathcal{C}$, and the set of all valid complete traces (using a transition system $\mathcal{S}$) when starting from configuration $C \in \mathcal{C}$ is denoted by $\mathit{Traces}_{\mathcal{S}}(C)$. 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 cost of the less expensive complete trace.

Definition 6 (worst- and best-case cost)**.** *Let $\mathcal{S}$ be a transition system. For an initial configuration C, its worst-case cost function $wc_{\mathcal{S}}(C)$ is the least upper bound of the length of all complete traces beginning with configuration C, and its best-case cost function $bc_{\mathcal{S}}(C)$ is the greatest lower bound of the length of all these complete traces beginning at C.*

Clearly, $wc_{\mathcal{S}}$ and $bc_{\mathcal{S}}$ are not computable. Our goal is to automatically find a lower-bound function $\rho : \mathbb{Z}^n \to \mathbb{R}_{\geq 0}$ such that for any initial configuration $C = (\ell_0, \sigma)$ we have

$wc_{\mathcal{S}}(C) \geq \rho(\sigma(\bar{x}))$, that is, it is a lower bound on the worst-case cost. A lower bound on the best-case cost would be a function $\rho : \mathbb{Z}^n \to \mathbb{R}_{\geq 0}$ that ensures that $bc_{\mathcal{S}}(C) \geq \rho(\sigma(\bar{x}))$ for any initial configuration $C = (\ell_0, \sigma)$. In what follows, for a function $\rho(\bar{x})$, we let $\|\rho(\bar{x})\| = \lceil \max(0, \rho(x)) \rceil$ to map all negative valuations of ρ to zero.

Example 8. *Consider the transition system*


$\tau_2 : \begin{cases} x > 0, \\ x' = x - u, \\ u \geq 1, \\ u \leq 2 \end{cases}$
$\tau_1 : \begin{cases} x \geq 0, \\ x' = x, \\ y' = y \end{cases}$
$\tau_3 : \begin{cases} x \leq 0, \\ x' = x \end{cases}$
ℓ_0
ℓ_1
ℓ_e

This transition system 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 $\rho(x) = 2$, although the first one is the most precise one.

3.2 Local Lower-Bound Functions

Existing techniques and tools for cost analysis (for example, [26, 3]) work by inferring *local* (iteration) bounds for those parts of the transition system that correspond to loops, and then combining these bounds by propagating them "backwards" to the entry point in order to obtain a *global* bound. They can infer global bounds for nested-loops as well, given the iteration bounds of each loop. In Chapter 2 we have seen already this composition in its motivating example (Figure 2.1). Thus, we focus on inferring local lower bounds on the number of iterations that non-nested loops (more precisely, parts of the transition system that correspond to loops) can make, and assume that they can be rewritten to global bounds by adopting the existing techniques of [26, 3] (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 $\|\rho(\bar{x})\|$ that is a (local) lower bound on the worst-case cost on its number of iterations, that is, 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.

For ease of presentation, we consider a special case of transition systems in which all locations, except the initial and exit ones, define loops. In paper [9] (Chapter 6, page 87), there is complete information about how the techniques can be used for the general case. In particular, we consider that each non-trivial strongly connected component 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, as mentioned in Section 3.1.2. The notions of best-case and worst-case cost in Definition 6 naturally extend to the cost of a loop ℓ, that is, we can ask what is the best-case and worst-case number of iterations of a given loop.

The overall idea of this 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 lower bound on the best-case cost of the specialized loop is by definition a lower bound on the worst-case cost of loop ℓ, as it does not necessarily hold for all execution traces but rather for the class of restricted ones. Technically, inferring a lower bound on the best-case of a specialized loop is done by inferring a metering function ρ [27], 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 a way that the transition system obtained by putting all specialized loops together is non-blocking, that is, 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.

Through this section, we generalize the basic definition of metering function for simple loops from [26] to general types of loops and explore its limitations (Section 3.2.1), and we explain how to overcome these limitations by means of the following specializations: using quasi-invariants to narrow the set of input values (Section 3.2.2); narrowing loop guards to make loop transitions mutually exclusive and force some execution order between them (Section 3.2.3); and narrowing the space of non-deterministic choices to force longer executions (Section 3.2.4).

3.2.1 Metering Functions

Metering functions were introduced by [27], 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 [27] 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.

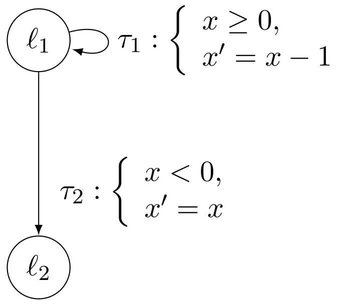
Definition 7 (Metering function)**.** *We say that a function ρ_ℓ is a metering function for a loop $\ell \in \mathcal{L}$ if the two following conditions hold:*

1. ***Decreases at most by one.*** *The function decreases at most by one in each iteration of the loop.*

2. ***Non-positive on exit.*** *The metering is lower than or equal to zero when leaving the loop.*

Assuming (ℓ, σ) is a reachable configuration in the transition system, 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 the transition system since we are interested only in non-blocking executions. Typically, we are interested in linear metering functions, since they are easier to infer and cover most loops in practice.

Example 9 (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):*

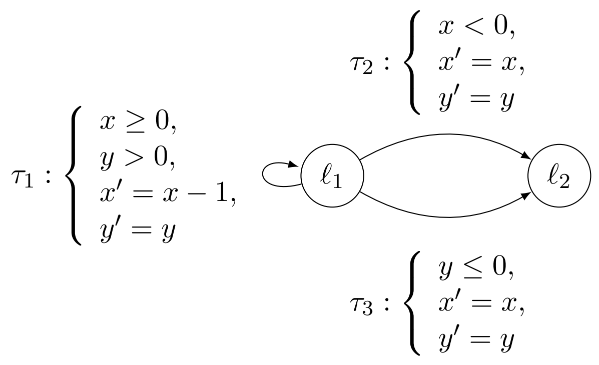

ℓ_1
$\tau_1 : \begin{cases} x \geq 0, \\ x' = x - 1 \end{cases}$
$\tau_2 : \begin{cases} x < 0, \\ x' = x \end{cases}$
ℓ_2

The function $\rho_{\ell_1}(x) = x+1$ is a valid metering function because it decreases by exactly one in τ_1 and becomes non-positive when τ_2 is applicable ($x < 0 \rightarrow x + 1 \leq 0$, so it holds both conditions of Definition 7). 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 (1) *and* (2)*. Although all of them are valid metering functions, $\rho_{\ell_1}(x)$ is preferable as it is more accurate (larger) and thus captures more precisely the number of iterations of the loop.*

3.2.2 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 the condition of non-positiveness of the metering when leaving the loop for all exit transitions. This is illustrated in the next example.

Example 10. *Consider the following loop that iterates on ℓ_1 if $x \geq 0 \wedge y > 0$, and exits when $x < 0$ or $y \leq 0$:*


$\tau_2 : \begin{cases} x < 0, \\ x' = x, \\ y' = y \end{cases}$
$\tau_1 : \begin{cases} x \geq 0, \\ y > 0, \\ x' = x - 1, \\ y' = y \end{cases}$
ℓ_1
ℓ_2
$\tau_3 : \begin{cases} y \leq 0, \\ x' = x, \\ y' = y \end{cases}$

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 the condition of being non-positive for τ_3, as

$y \leq 0 \not\to x+1 \leq 0$. *Moreover, no other function depending on x (for example, $\frac{x}{2}$, $x-2$, etc.) will be a valid metering function, as it will be impossible to prove non-positiveness condition 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 such that conditions of the metering function (Definition 7) hold in the context of the corresponding reachable states. For example, the reachable states might exclude some of the exit transitions.If these transitions are guaranteed to never be used, then the condition of the non-positiveness of the metering is not required to hold for them. A metering function in this context is a lower bound on the best-case cost of the loop when starting from that specific input, and thus it is a lower bound on the worst-case cost (but 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 [35].

Definition 8 (Quasi-invariant)**.** *A quasi-invariant for a loop ℓ is a formula $\mathcal{Q}_\ell$ over $\bar{x}$ that restricts the valid states of ℓ. It fulfills that:*

1. *Once $\mathcal{Q}_\ell$ holds, it will hold during all subsequent visits to ℓ.*
2. *There is an assignment of the variables that makes the quasi-invariant true.*

Intuitively, $\mathcal{Q}_\ell$ is similar to an inductive invariant but without requiring it to hold on the initial states. The second condition ensures that the subset of states induced by the quasi-invariant is not empty.

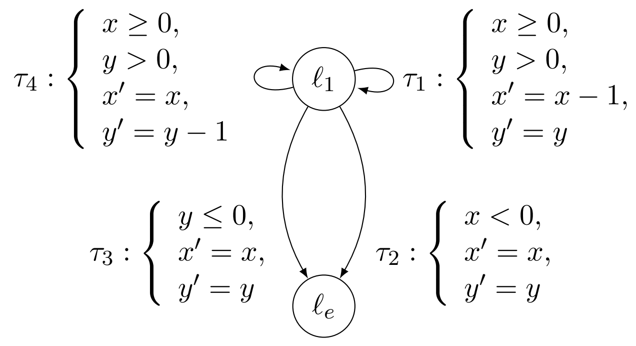
Given a quasi-invariant $\mathcal{Q}_\ell$ for ℓ, we say that ρ_ℓ is a metering function for ℓ if both conditions of Definition 7 hold in the context of the states induced by $\mathcal{Q}_\ell$.

Example 11. *Recall that the loop in Example 10 only admitted trivial metering functions because of the exit transition τ_3. It is easy to see that $\mathcal{Q}_{\ell_1} \equiv x < y$ satisfies the conditions to be a quasi-invariant in the states induced by $\mathcal{Q}_\ell$, because y is not modified in τ_1 and x decreases. 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, that is, τ_3 satisfies the non-positiveness when leaving the loop because $x < y \wedge y \leq 0 \to x+1 \leq 0$.*

3.2.3 Narrowing Guards

The loops that we have considered so far consist of a single loop transition, which 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 12. *Consider the loop in Example 10 with a new transition τ_4 that decrements y (it corresponds to the loop of the example in Figure 3.1):*


$\tau_4 : \begin{cases} x \geq 0, \\ y > 0, \\ x' = x, \\ y' = y - 1 \end{cases}$
ℓ_1
$\tau_1 : \begin{cases} x \geq 0, \\ y > 0, \\ x' = x - 1, \\ y' = y \end{cases}$
$\tau_3 : \begin{cases} y \leq 0, \\ x' = x, \\ y' = y \end{cases}$
ℓ_e
$\tau_2 : \begin{cases} x < 0, \\ x' = x, \\ y' = y \end{cases}$

The most precise lower bound on the worst-case of this loop is $\|\rho_{\ell_1}(x, y)\|$ where $\rho_{\ell_1}(x, y) = x + y$. As mentioned, this corresponds, for example, to an execution that uses τ_1 until $x = 0$, that is, x times, and then τ_4 until $y = 0$, that is, 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 both conditions of Definition 7 in the states induced by $\mathcal{Q}_{\ell}$, 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 lower bound on the best-case of the original loop, but rather it is a valid lower bound on the worst-case, that is what we are interested in.

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.

Metering function and quasi-invariant conditions can be specialized to hold only for executions that use the specialized loop. Suppose that for a loop $\ell \in \mathcal{L}$ we are given a narrowing $\mathcal{G}_\tau$ for each loop transition τ, then ρ_ℓ and $\mathcal{Q}_\ell$ are metering function and quasi-invariant respectively for the corresponding specialized loop if the next conditions hold.

Metering function

Decreases at most by one. The function decreases at most by one in each iteration of the specialized loop.

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \wedge \mathcal{R} \rightarrow \rho_\ell(\bar{x}) - \rho_\ell(\bar{x}') \leq 1 \textbf{ for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \tag{3.1}$$

Non-positiveness on exit. The metering is lower than or equal to zero when leaving the loop from the states induced by the quasi-invariant.

$$\forall \bar{x}.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R}(\bar{x}) \rightarrow \rho_\ell(\bar{x}) \leq 0 \textbf{ for each } (\ell, \ell', \mathcal{R}) \in \mathcal{T} \tag{3.2}$$

Quasi-invariant

Quasi-invariance. Once the quasi-invariant holds for the specialized loop, it will always hold.

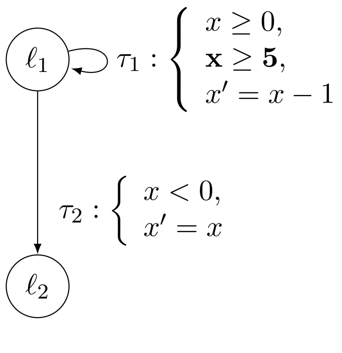
$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \wedge \mathcal{R} \rightarrow \mathcal{Q}_\ell(\bar{x}') \textbf{ for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \tag{3.3}$$

Non-empty quasi-invariant. There is an assignment of the variables that satisfies the constraints of the quasi-invariant.

$$\exists \bar{x}.\ \mathcal{Q}_\ell(\bar{x}) \tag{3.4}$$

Conditions (3.3,3.4) guarantee that $\mathcal{Q}_\ell$ is a non-empty quasi-invariant for the specialized loop, and conditions (3.1,3.2) 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.

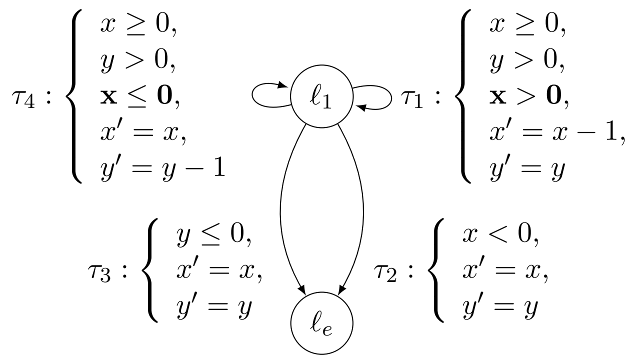
Example 13. *Consider the loop from Example 9 where we have specialized the guard of τ_1 by adding $x \geq 5$:*


ℓ_1
$\tau_1 : \begin{cases} x \geq 0, \\ \mathbf{x \geq 5}, \\ x' = x - 1 \end{cases}$
$\tau_2 : \begin{cases} x < 0, \\ x' = x \end{cases}$
ℓ_2

With this specialized guard and considering $\mathcal{Q}_{\ell_1} \equiv true$, the metering function $\rho_{\ell_1}(x) = x + 1$ and $\mathcal{Q}_{\ell_1}$ satisfy their conditions. 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 guard narrowing to hold that, from any state defined by the quasi-invariant, we can make progress, either by making a loop iteration with the guard narrowing or exiting the loop.

Example 14. *In Example 12, 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 $\mathcal{G}_{\tau_4} \equiv x \leq 0$.*

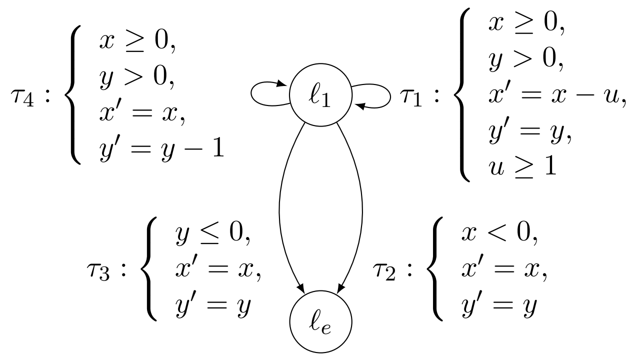

$\tau_4 : \begin{cases} x \geq 0, \\ y > 0, \\ \mathbf{x \leq 0}, \\ x' = x, \\ y' = y - 1 \end{cases}$
ℓ_1
$\tau_1 : \begin{cases} x \geq 0, \\ y > 0, \\ \mathbf{x > 0}, \\ x' = x - 1, \\ y' = y \end{cases}$
$\tau_3 : \begin{cases} y \leq 0, \\ x' = x, \\ y' = y \end{cases}$
ℓ_e
$\tau_2 : \begin{cases} x < 0, \\ x' = x, \\ y' = y \end{cases}$

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, respectively. 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 the quasi-invariant $\mathcal{Q}_\ell$, it verifies its conditions (3.3,3.4), and more importantly, the loop in ℓ_1 is non-blocking w.r.t $\mathcal{Q}_{\ell_1}$, $\mathcal{G}_{\tau_1}$, and $\mathcal{G}_{\tau_4}$, that is, the condition of being non-blocking holds.

3.2.4 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 lower bounds on the worst-case, we are clearly searching for longer executions. However, since our approach is based on inferring lower bounds on the best-case, we have to take all executions into account which might result in less precise, or even trivial, lower bounds on the worst-case.

Example 15. *Consider a modification of the loop in Example 12 in which the variable x in τ_1 is decreased by a non-deterministic positive quantity u:*


$\tau_4 : \begin{cases} x \geq 0, \\ y > 0, \\ x' = x, \\ y' = y - 1 \end{cases}$
ℓ_1
$\tau_1 : \begin{cases} x \geq 0, \\ y > 0, \\ x' = x - u, \\ y' = y, \\ u \geq 1 \end{cases}$
$\tau_3 : \begin{cases} y \leq 0, \\ x' = x, \\ y' = y \end{cases}$
$\tau_2 : \begin{cases} x < 0, \\ x' = x, \\ y' = y \end{cases}$
ℓ_e

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 also narrow the space of non-deterministic choices. This way, metering functions should be valid with respect to fewer executions and more likely be found and be more precise.

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 (3.1)-(3.4) to hold but after adding $\mathcal{U}_\tau$ to the left-hand side of the implications in (3.3) and (3.1). We also have to require the system to be non-blocking after guards specialization, as explained at the end of Section 3.2.3. Besides, unlike narrowing of guards, narrowing of non-deterministic choices might make a transition invalid, that is, not satisfying the definition of *valid transition* from Section 3.1.1. The solution comes from modifying this definition to take into account the conditions introduced by the corresponding narrowings.

Example 16. *To solve the problems shown in Example 15 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.*

$$\begin{tikzcd} \ell_1 \arrow[loop left, "\tau_4"] \arrow[loop right, "\tau_1"] \arrow[d, bend right, "\tau_3"'] \arrow[d, bend left, "\tau_2"] \\ \ell_e \end{tikzcd}$$

$$\tau_4 : \begin{cases} x \geq 0, \\ y > 0, \\ x' = x, \\ y' = y - 1 \end{cases} \qquad \tau_1 : \begin{cases} x \geq 0, \\ y > 0, \\ x' = x - u, \\ y' = y, \\ u \geq 1, \\ \mathbf{u \leq 1} \end{cases}$$

$$\tau_3 : \begin{cases} y \leq 0, \\ x' = x, \\ y' = y \end{cases} \qquad \tau_2 : \begin{cases} x < 0, \\ x' = x, \\ y' = y \end{cases}$$

Considering the narrowing $\mathcal{U}_{\tau_1}$, the resulting loop is equivalent to the one presented in Example 14 so we could obtain the precise metering function $\rho_{\ell_1}(x, y) = x + y$ with the quasi-invariant $\mathcal{Q}_{\ell_1} \equiv x \leq y \wedge x \geq 0$.

3.3 Inference Using Max-SMT

Metering functions and narrowings can be inferred automatically using Max-SMT. The template-based approach of [19, 35] can be used to find $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, and $\mathcal{Q}_\ell$ by representing them as template constraint systems, that is, each is a conjunction of linear constraints where coefficients and constants are unknowns.

Since there might be many metering functions that satisfy the conditions explained in this work, we are interested in narrowing the search space of the SMT solver in order to find more accurate ones, that is, leading to longer executions. We have proposed the next 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 to generate longer executions.
- **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, when 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.

3.4 Implementation and Experimental Evaluation

We have implemented a *LO*wer-*B*ound synthesiz*ER*, named LOBER, that can be used from an online web interface at `http://costa.fdi.ucm.es/lober`. 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 transition systems and terminating transition systems whose cost is unbounded (that is, 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 sec.

We compare the obtained results to those obtained by the LoAT [27, 26] 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.

Globally, both tools behave the same in 412 programs, 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, while LoAT is more precise in 11 programs. 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 four cases and it can only find a constant lower bound in one 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 [14]. 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), which 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 six 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. (2) In the other five cases, our tool finds a linear bound but with a worse set of quasi-invariants, which makes the LoAT bound provide unboundedly longer executions.

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).

Chapter 4

Certified Abstract Cost Analysis with Upper and Lower Bounds

In Chapter 2, we presented a generalization of Automated Cost Analysis for abstract programs, where the cost was bounded from above by means of *upper bounds on the worst-case cost.* In Chapter 3, the presented technique is focused on inferring *lower bounds on the worst-case cost* by means of loop specialization. In this chapter, whose complete information can be found in *Certified Cost Bounds for Abstract Programs* (Chapter 6, page 111), we extend the framework presented in Chapter 2 with the tool developed in Chapter 3, so we can exploit the power of both Quantitative Abstract Execution and the synthesis of lower bounds together within the verification framework for inferring the cost of abstract programs.

As explained in Chapter 3, LOBER [9] bases the search of lower bounds on the worst-case cost on the specialization of loops. The basic idea is that a lower bound on the best-case cost of the specialized loop is a lower bound on the worst-case cost of the original one. While this system was initially designed to compute lower bounds on the worst-case cost, it allows disabling loop specialization, so as to provide lower bounds on the best-case cost instead. We use the system in that way to compute lower bounds on the best-case cost. This capability of LOBER of controlling the specializations of loops in order to obtain accurate lower bounds on the best-case cost is essential to extend Quantitative Abstract Execution to lower bounds.

The aim of using lower bounds on the best-case cost instead of lower bounds on the worst-case cost is to capture the result of all possible executions of the analyzed program. Let $cost_w$ be the worst cost of the program, $cost_b$ the best cost, ub_w the upper bound on the worst-case cost, and lb_b the lower bound on the best-case cost, it holds that

$$lb_b \leq cost_b \leq cost \leq cost_w \leq ub_w$$

Given a lower bound on the best-case cost and an upper bound on the worst-case cost, all possible costs will be bounded independently from the execution.

As mentioned in Chapter 2, arguing correctness of an abstract cost analysis is complex, because it must be valid for an infinite set of concrete programs and it contains several complex components, so it is useful to verify and certify the inferred abstract cost, as the analysis of the failed verification attempt gives immediate feedback on the cause.

Moreover, in Chapter 2, we relied on the certification stage to determine whether an inferred cost was exact or an upper bound. Including also lower bounds in the cost analysis gives us now the possibility of knowing before verification if the inferred cost will be exact, as if both lower bound on the best-case cost and upper bound on the worst-case cost match, then we will have an exact estimation of the cost. This fact stands out in the optimization of Quantitative Abstract Execution, as we can avoid testing exact cases if both bounds do not match.

In automated cost analysis, one infers cost bounds often from loop invariants, ranking functions, metering functions, and size relations computed during Symbolic Execution [7, 25, 48, 18]. However, as explained in Chapter 2, for abstract programs, we need a more general concept: a cost invariant. While previously we used only one cost invariant for each loop verification, now we have to differentiate between lower bounds and upper bounds, giving rise to *lower cost invariants* and *upper cost invariants.* Cost invariants of Chapter 2 correspond to the latter, as previously we worked only with upper bounds. Similarly, we have to extend the concept of cost post-conditions to handle also lower bounds.

4.1 Cost Annotations by Example

We introduce the new terminology of cost annotations informally by means of the motivating example used in Chapter 2 (Figure 2.1). We subsume any quantitative aspect of a program under the term *cost* expressed in an unspecified *cost model* with the understanding that it can be instantiated to specific cost measures.

Input to QAE is the abstract program to analyze, including annotations (highlighted in light gray in Figure 4.1) that express restrictions on the permitted instances of abstract statements. Annotations highlighted in dark gray are automatically inferred by abstract cost analysis and are input for the certifier. Each abstract statement P has an associated *abstract cost* function parametric in the locations of its footprint, represented by an abstract cost symbol ac_P. As our framework works for both upper and lower bounds, we distinguish the upper and lower bounds of the abstract cost of an abstract statement P using the cost symbols $\mathsf{ac}^l_\mathsf{P}(t, w)$ and $\mathsf{ac}^u_\mathsf{P}(t, w)$, respectively. These symbols can be instantiated with any concrete function parametric in t, w being a valid cost bound for the instance of P. For example, for the instance "P ≡ x=t+1;" the constant function 1 is the correct *exact* cost, while t with $t \geq 1$ is a correct *upper bound* cost. On the other hand, for the instance

```
k = 0;
while (k<t){
k++;
}
```

the exact cost is t, while correct (but not tight) upper and lower bound costs are t^2 and 1, respectively. The most accurate upper and lower bounds, in this case, are t. As both bounds match, the inferred cost is exactly t.

As pointed out at the beginning of the chapter, we need *cost invariants* to capture the cost of each loop iteration. They are declared by the keywords "**cost_invariant_upper**" and

Program Before

```
int i = 0;
//@ loop_invariant i ≥ 0 && i ≤ t;
//@ cost_invariant_upper
  $i \cdot (ac^u_P(t,w) + ac^u_Q(t,z) + 2)$;
//@ cost_invariant_lower
  $i \cdot (ac^l_P(t,w) + ac^l_Q(t,z) + 2)$;
//@ decreases t − i;
while (i < t) {
  //@ assignable x;
  //@ accessible t,w;
  //@ cost_footprint t,w;
  \abstract_statement P;
  //@ assignable y;
  //@ accessible i,t,y,z;
  //@ cost_footprint t,z;
  \abstract_statement Q;
  i ++;
}
//@ assert $2 + t \cdot (ac^l_P(t,w) +$
    $ac^l_Q(t,z) + 2)$ <=\cost;
//@ assert \cost<= $2 + t \cdot$
    $(ac^u_P(t,w) + ac^u_Q(t,z) + 2)$;
```

Program After

```
int i = 0;
//@ assignable x;
//@ accessible t,w;
//@ cost_footprint t,w;
\abstract_statement P;
//@ loop_invariant i ≥ 0 && i ≤ t;
//@ cost_invariant_upper
  $i \cdot (ac^u_Q(t,z) + 2)$;
//@ cost_invariant_lower
  $i \cdot (ac^l_Q(t,z) + 2)$;
//@ decreases t − i;
while (i < t) {
  //@ assignable y;
  //@ accessible i,t,y,z;
  //@ cost_footprint t,z;
  \abstract_statement Q;
  i ++;
}
//@ assert $2 + ac^l_P(t,w) +$
    $t \cdot (ac^l_Q(t,z) + 2)$ <= \cost
//@ assert \cost <= 2+
    $ac^u_P(t,w) + t \cdot (ac^u_Q(t,z) + 2)$;
```

Preconditions and Postconditions

Inputs: t, w, x, y, z   Precondition: $t > 0$ Postcondition: **\cost_1** $\geq$ **\cost_2**

Figure 4.1: Motivating example.

"**cost_invariant_lower**". To generate them, it is necessary to infer the *cost growth* (upper and lower) of abstract programs, that bounds the number of loop iterations executed so far.

In Figure 2.1 the number of loop iterations is determined exactly by the increase of variable i until it reaches t. However, we can handle more general programs, where the number of performed iterations might not be precisely inferred. For example, consider the following program, where "$*$" randomly returns true or false, causing a different increase of i in each case:

```
while (i  < t){
        if  (*)
                i ++;
        else
                i +=2;
}
```

Now the most precise information obtainable on the number of loop iterations is that they are between $\frac{t}{2}$ and t. The lower bound inference is reflected in the growth and cost invariants. In Section 4.2 we describe the automated inference of cost invariants including the generation of cost growth for all loops.

The framework can express and prove quantitative relational properties. The assertions in the last lines in Figure 4.1 use the expression **cost** referring to the total accumulated cost of the program, i.e., the quantitative *postconditions*. We support quantitative relational postconditions such as **cost_1** $\geq$ **cost_2** , where **cost_1**, **cost_2** refer to the total cost of the original (on the left) and transformed (on the right) program, respectively. To prove relational properties, it is necessary to infer matching upper and lower bounds for the number of loop iterations. Only then the comparison of the invariants allows concluding that the programs from which they derive satisfy the stipulated relational property. Otherwise, over- and under-approximation introduced by the cost analysis can make the relation hold for the postconditions, but the relational property does not necessarily hold for the programs. Now, we support three kinds of cost specifications. These are: *exact*, *upper bound*, and *lower and upper bound* cost.

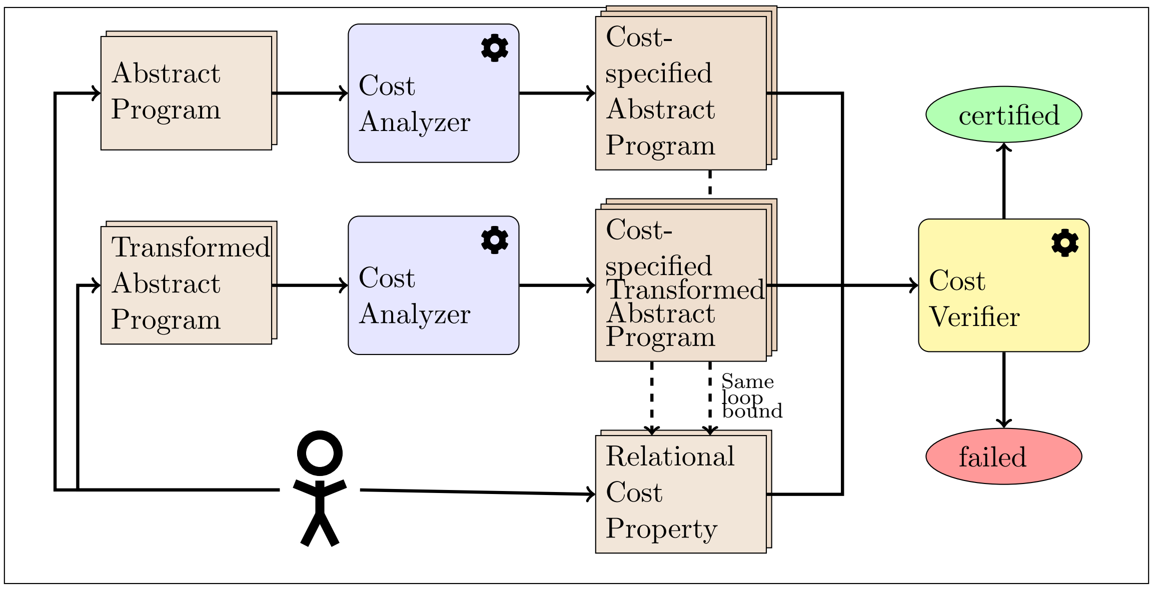


Figure 4.2: Schema of tool chain for cost certification of abstract programs

In Figure 4.2, we present a schema of the toolchain for proving relational properties. Given two abstract programs (the original and the transformed one), we obtain the corresponding cost-specified programs by means of the cost analyzer. If both upper and lower bounds of the loop match, then the tool verifies that the relational property holds. If the property does not hold, the tool performs a relaxation on the property and re-executes the verification step.

4.2 Abstract Cost Analysis

While the initial approach of Chapter 2 was focused on the generation and verification of upper bounds on the cost of abstract programs, now we extended it to generate and certify also lower bounds on the abstract cost. We have also relaxed the conditions to guarantee soundness of the abstract cost analysis framework. In particular, the concept of cost neutral abstract statement is less restrictive than in Chapter 2.

This section is structured as follows: Subsection 4.2.1 strengthens the notion of cost neutral abstract statements when including lower bounds as well, Subsection 4.2.2 generalizes the automatic inference of inductive cost invariants for abstract programs from abstract cost relation systems. Subsection 4.2.3 describes how to generate cost postconditions used to prove relational properties and required to handle nested loops.

4.2.1 Soundness of Abstract Cost Relations

For soundness of abstract cost analysis it is essential that

1. no abstract statement in the loop modifies any of the variables that influence loop cost, i.e., they do not *interfere with cost*, and
2. the cost of the abstract statement in the loop is independent of the variables modified in the loop. We call the latter *cost neutral abstract statements.*

To fulfill requirement 2 we introduce the new notion of a *cost neutral AS* to ensure that variables in the cost footprint are not modified by other statements in the loop that decrease the cost.

Due to the number of involved components of cost specifications, the following definition is a little long-winded, but conceptually straightforward.

Definition 9 (Cost Neutral Abstract Statement)**.** *Given a loop L, where:*

- $W(L)$ *is the set of variables written to by the non-abstract statements of L;*
- $\texttt{Abstr}(L)$ *is the set of all abstract statements in loop L;*
- $\mathit{Frame}(\texttt{Abstr}(L))$ *is the set of variables assigned by any abstract statement $A \in \texttt{Abstr}(L)$;*
- $\mathit{CostFootprint}(A)$ *is the set of variables on which the cost of an A depends;*
- $\mathit{CostFootprint}(\texttt{Abstr}(L))$ *is the set of variables which the cost of any abstract statement A of L depends on;*
- $\mathit{Conditions}(\texttt{Abstr}(L))$ *is the set of conditions defined in the cost footprint of any abstract statement $A \in \texttt{Abstr}(L)$;*
- $\mathit{Vars}(\texttt{Conditions}(\texttt{Abstr}(L)))$ *is the set of variables involved in* $\texttt{Conditions}(\texttt{Abstr}(L))$*;*

- *$R(L)$ is the ranking function of loop L;*
- *$m(L)$ is the metering function of loop L;*
- *ac^u_P and ac^l_P are the upper and lower bounds, respectively, on the abstract cost of abstract statement P.*

L is a loop with cost neutral *abstract statements if, for any of its abstract statements A either*

1. *any variable not occurring in conditions of the cost footprint and that is assigned in the loop does not occur in the cost footprint; or*
2. *if a variable occurs in a cost footprint condition, then the changes in the cost referring to that variable are safe. That is, changes in the abstract statement costs preserve upper and lower bounds of the corresponding costs and guarantee that the upper and lower bounds on the iterations remain valid.*

□

Example 17. *Both loops in Figure 4.1 have cost neutral abstract statements. No conditions are declared in the cost footprint, so Definition 9 applies. In the left program*

$$\begin{aligned}&\mathit{Frame}(\{P,Q\}) \cup W(L) = \{\mathsf{x},\mathsf{y},\mathsf{i}\}\,,\\ &\mathit{CostFootprint}(P) = \{\mathsf{t},\mathsf{w}\}\,,\\ &\mathit{CostFootprint}(Q) = \{\mathsf{t},\mathsf{z}\}\,,\end{aligned}$$

so the condition is fulfilled. Similar for the program on the right. □

Given a program $\mathcal{P}$ with variables $\overline{x}$ and Abstract Cost Relation System following Definition 1 with initial equation $C_{ini}(\overline{x})$. We denote by $\mathit{eval}(C_{ini}(\overline{x}),\sigma_0)$ the evaluation of the abstract cost relation system for a given initial assignment σ_0 of the variables. This is a standard evaluation of recurrence equations performed by instantiating the right-hand side of the equations with the values of the variables in σ_0 and checking the satisfiability of the size constraints (if the expression being checked or accumulated contains "_", the evaluation returns "unbound"). As usual, the process is repeated until an equation without calls is reached. Moreover, we denote by $\mathit{eval}(C_{ini}(\overline{x}),\sigma_0)^u$ the *upper-bound of the evaluation of the abstract cost relation system*, obtained by substituting each abstract statement ac_P in $\mathit{eval}(C_{ini}(\overline{x}),\sigma_0)$ by its upper bound ac^u_P. Similarly, we define the *lower bound of the evaluation $\mathit{eval}(C_{ini}(\overline{x}),\sigma_0)^l$* by substituting each abstract statement ac_P by its lower bound ac^l_P. It is trivial to see that $\mathit{eval}(C_{ini}(\overline{x}),\sigma_0)^l \leq \mathit{eval}(C_{ini}(\overline{x}),\sigma_0) \leq \mathit{eval}(C_{ini}(\overline{x}),\sigma_0)^u$.

Example 18. *Consider the abstract cost relation system of the left program in Figure 4.1 with variables $\overline{\mathsf{x}} = (\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z})$, initial state $\sigma_0 = (2,0,0,0,0)$, and cost model $\mathcal{M}_{inst}$ (i.e., c_{before}, $c_{B_{w_0}}$, and c_{w_0} assume values 1, 1, and 2, respectively). The evaluation of the ACRS is $\mathit{eval}(C_{ini}(\overline{\mathsf{x}}),\sigma_0) = 6 + 2\cdot\mathsf{ac}_\mathsf{P}(2,0) + 2\cdot\mathsf{ac}_\mathsf{Q}(2,0)$. Its upper bound is $\mathit{eval}(C_{ini}(\overline{\mathsf{x}}),\sigma_0)^u = 6 + 2\cdot\mathsf{ac}^u_\mathsf{P}(2,0) + 2\cdot\mathsf{ac}^u_\mathsf{Q}(2,0)$, its lower bound is $\mathit{eval}(C_{ini}(\overline{\mathsf{x}}),\sigma_0)^l = 6 + 2\cdot\mathsf{ac}^l_\mathsf{P}(2,0) + 2\cdot\mathsf{ac}^l_\mathsf{Q}(2,0)$.* □

The following theorem states the soundness of the abstract cost relation system.

Theorem 1 (Soundness of the Abstract Cost Relation System)**.** *Let $\mathcal{M}$ be a cost model and $\mathcal{P}$ an abstract program with cost neutral abstract statements in loops (Definition 9), $c_{\mathcal{P}}$ the abstract cost of $\mathcal{P}$, and C_{ini} the initial equation for the ACRS obtained by Definition 1. Then, for any initial state of the variables $\sigma_0 \in \mathbb{Z}^{n_m}$, it holds that $eval(C_{ini}(\overline{x}), \sigma_0)^l \leq c_{\mathcal{P}}(\sigma_0) \leq eval(C_{ini}(\overline{x}), \sigma_0)^u$.*

4.2.2 From Abstract Cost Relation Systems to Abstract Cost Invariants

Example 18 shows that abstract cost relation systems are evaluable for concrete instances. However, to enable automated Quantitative Abstract Execution, we need to obtain from them *closed-form* cost invariants and postconditions, i.e., non-recursive expressions. In Chapter 2, we introduced the concept of *abstract cost invariant*, that we have to extend now to *upper cost invariants* and *lower cost invariants.*

In contrast to abstract cost invariants, postconditions provide a bound for the cost *after* execution of the *whole* loop they refer to. Typically, an upper bound postcondition for a loop has the form $max\_iter * max\_cost + max\_base$, where max_iter is the maximal number of iterations of the loop, max_cost is the maximal cost of any loop iteration, and max_base is the maximal cost of executing the loop with no iterations. A lower bound post condition is defined similarly by using the lower bound on the number of iterations, the lower bounds on the abstract costs and the minimal concrete costs (i.e., $min\_iter * min\_cost + min\_base$).

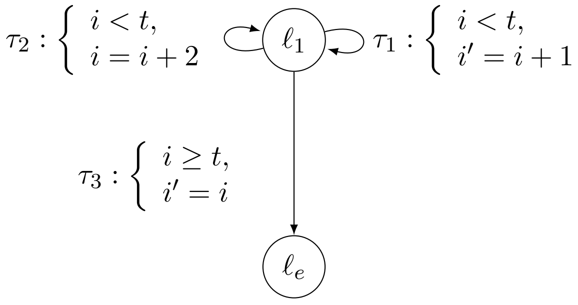
An upper bound abstract cost invariant has the form $growth_u * max\_cost + max\_base$, where $growth_u$ counts how many times the loop has been executed at most and hence provides an upper bound after *each* loop iteration. Similarly, a lower bound abstract cost invariant has the form $growth_l * min\_cost + min\_base$, where $growth_l$ counts how many times the loop has been executed at least and provides a lower bound after *each* loop iteration. The challenge is to design automated techniques that infer upper and lower bounds $growth_u$, $growth_l$.

Definition 10 (Upper and lower growth)**.** *Given a loop with ranking function $R = c + \sum_i a_i \cdot v_i$, metering function $m = d + \sum_i b_i \cdot v_i$, where c, d are the constant parts of the functions, v_i are the variable parts, and a_i, b_i are constant coefficients. If we denote with v_i^0 the initial value of variable v_i before entering the loop, then the upper growth and the lower growth are*

$$growth_u = \sum_i a_i \cdot \left(v_i^0 - v_i\right) \qquad\qquad growth_l = \sum_i b_i \cdot \left(v_i^0 - v_i\right)$$

For any loop execution, $growth_l \leq growth \leq growth_u$, where growth is the exact number of performed iterations.

Example 19. *Let the following loop, that we introduced in Section 4.1.*


$\tau_2 : \begin{cases} i < t, \\ i = i + 2 \end{cases}$
ℓ_1
$\tau_1 : \begin{cases} i < t, \\ i' = i + 1 \end{cases}$
$\tau_3 : \begin{cases} i \geq t, \\ i' = i \end{cases}$
ℓ_e

Let i_0 be the initial value of i. We have that a ranking function is $R = t - i$ and a metering function is $m = \frac{t-i}{2}$. Then, both growths are

$$growth_u = i - i_0 \qquad\qquad growth_l = \frac{i - i_0}{2}$$

□

We can now define the concept of abstract cost invariant that relies on abstract cost relations and growths as defined above.

Definition 11 (Abstract Cost Invariant)**.** *Given an abstract cost relation system as in Definition 1 with growth as in Definition 10, its upper and lower* abstract cost invariants *are defined as follows:*

- $\mathtt{cinv}_u(\overline{x}) = \mathtt{C_B}^{\mathtt{max}} + growth_u \cdot \left(\sum_{j=1}^{n} \mathtt{ac}_{\mathtt{j}}^{u}\left(c_{j,1}, \ldots, c_{j,h_{cj}}\right) + \sum_{i=1}^{m} \mathtt{C_{N_i}}^{\mathtt{max}}\right)$
- $\mathtt{cinv}_l(\overline{x}) = \mathtt{C_B}^{\mathtt{min}} + growth_l \cdot \left(\sum_{j=1}^{n} \mathtt{ac}_{\mathtt{j}}^{l}\left(c_{j,1}, \ldots, c_{j,h_{cj}}\right) + \sum_{i=1}^{m} \mathtt{C_{N_i}}^{\mathtt{min}}\right)$

where $\mathtt{C_B}^{\mathtt{max}}$ and $\mathtt{C_B}^{\mathtt{min}}$ stand for the maximal and minimal value, respectively, the expression $\mathtt{C_B}$ can take under the constraints φ_B; $\mathtt{C_{N_i}}^{\mathtt{max}}$ and $\mathtt{C_{N_i}}^{\mathtt{min}}$ are the maximal and minimal value, respectively, of $\mathtt{C_{N_i}}$ under φ_I.

The maximal and minimal can be provided by cost analyzers and they give rise to automatically generated annotations

"//@ **cost_invariant_upper** $\mathtt{cinv}_u(\overline{x})$;" *and* "//@ **cost_invariant_lower** $\mathtt{cinv}_l(\overline{x})$;".

Example 20. *Consider the Example 19 (where $growth_u = i - i_0$ and $growth_l = \frac{i-i_0}{2}$) with the following abstract statement specification:*

```
//@ assignable w;
//@ accessible i, t, w;
//@ cost_footprint t;
\abstract_statement P;
```

Under $\mathcal{M}_{\mathsf{instr}}$, the evaluation of the loop guard and the increasing of i *both have unit cost, so the abstract cost relation system is:*

$$\begin{array}{ll} C(\mathsf{i},\mathsf{t},\mathsf{w}) = 1 & \{\mathsf{i} \geq \mathsf{t}\} \\ C(\mathsf{i},\mathsf{t},\mathsf{w}) = \mathsf{ac_P}\,(\mathsf{t}) + 2 + C(\mathsf{i}',\mathsf{t},\_) & \{\mathsf{i}' = \mathsf{i}+1,\ \mathsf{i} < \mathsf{t}\} \\ C(\mathsf{i},\mathsf{t},\mathsf{w}) = \mathsf{ac_P}\,(\mathsf{t}) + 2 + C(\mathsf{i}',\mathsf{t},\_) & \{\mathsf{i}' = \mathsf{i}+2,\ \mathsf{i} < \mathsf{t}\} \end{array}$$

The value of the assignable variable w *in the recursive call is "forgotten" (item 6 in Definition 1), but this information loss does not affect the solvability of the abstract cost relation system. We obtain the following abstract cost invariants:*

"//@ **cost_invariant_upper** *1 +* $(i{-}i_0)\cdot$ *(2 +* $\mathsf{ac_P}$*(k));*"
"//@ **cost_invariant_lower** *1 +* $\frac{i-i_0}{2}\cdot$ *(2 +* $\mathsf{ac_P}$*(k));*"

□

Example 21.
*Sometimes the cost of a single iteration cannot be computed exactly by cost analyzers, resulting in maximized and minimized cost expressions for concrete instructions. To illustrate this phenomenon, we go over Example 6 again, which creates an array of non-constant size "*i*" in each iteration of a loop and measure memory consumption instead of instruction count.*

```
while (i < t) {
  a = new int[i];
  //@ assignable j;
  //@ accessible i,t,j,a,k;
  //@ cost_footprint k;
  \abstract_statement P;
  i ++;
}
```

*The resulting ACRS accumulates cost "*i*" at each iteration, plus the memory consumed by the abstract statement:*

$$\begin{array}{ll} C(\mathsf{i},\mathsf{t},\mathsf{j},\mathsf{k}) = 0, & \{\mathsf{i} \geq \mathsf{t}\} \\ C(\mathsf{i},\mathsf{t},\mathsf{j},\mathsf{k}) = \mathsf{ac_P}\,(\mathsf{k}) + \mathsf{i} + C(\mathsf{i}',\mathsf{t},\_,\mathsf{k}), & \{\mathsf{i}' = \mathsf{i}+1, \mathsf{i} < \mathsf{t}\} \end{array}$$

Now, maximizing the expression $\mathtt{C}_{\mathtt{N}_1} = \mathsf{i}$ *under* $\{\mathsf{i}' = \mathsf{i}+1, \mathsf{i} < \mathsf{t}\}$ *results in* ${\mathtt{C}_{\mathtt{N}_1}}^{\mathtt{max}} = 4 \cdot (\mathsf{t}-1)$ *and minimizing it results in* ${\mathtt{C}_{\mathtt{N}_1}}^{\mathtt{min}} = 4$*, as we are working with 32 bit integers.*[1] *Accordingly, we obtain the following cost invariants:*

//@ **cost_invariant_upper** *i ∗ (4∗(t − 1) +* $\mathsf{ac_P}$*(k));*
//@ **cost_invariant_lower** *i ∗ (4 +* $\mathsf{ac_P}$*(k));*.

While the maximized expression is computed automatically, the minimized expression was manually calculated, because LOBER [9] computes lower bounds on the number of iterations of a given loop, but it does not minimize the cost of single iterations (by default it assumes each iteration has unit cost). For this reason, in the current implementation, the tool only infers upper bounds for the heap *cost model.*

□

Let c_L denote the abstract cost of executing a loop L. We denote by c_I the portion of the cost in c_L up to the execution of iteration I.

Proposition 1. *Let* $\mathcal{L}$ *be a cost neutral loop, and* c_I *the abstract cost of* $\mathcal{L}$ *in iteration* I*,* $\mathtt{cinv}_u(\overline{x})$ *and* $\mathtt{cinv}_l(\overline{x})$ *its abstract cost invariants, and* $\sigma_I \in \mathbb{Z}^{n_m}$ *be the store after performing iteration* I *of* $\mathcal{L}$*. Then the following holds:*

[1]Better maximization and minimization results can with techniques such as in [10].

1. $\mathtt{cinv}_u(\overline{x})$ *and* $\mathtt{cinv}_l(\overline{x})$ *are true on entering the loop;*

2. $\mathtt{cinv}_l(\sigma_I) \le c_I(\sigma_I) \le \mathtt{cinv}_u(\sigma_I)$*.*

4.2.3 From Cost Invariants to Postconditions

To handle programs with nested loops and to prove relational properties it is necessary to infer *cost postconditions* for abstract programs. Cost postconditions for concrete programs are obtained by upper and lower bound solvers (e.g., COSTA [7], CoFloCo [25], AProVE [28], LoAT [26], Lober [9]) that compute *max_iter*, an upper bound on the number of iterations and *min_iter*, a lower bound on the number of iterations that a loop performs, by relying on ranking and metering functions. The cost postcondition is obtained by substituting $\mathtt{growth}_u$ by `max_iter` and $\mathtt{growth}_l$ by `min_iter` in the formula of $\mathtt{cinv}_u(\overline{x})$ and $\mathtt{cinv}_l(\overline{x})$ in Definition 11 as follows.

Definition 12 (Cost Postcondition)**.** *Let L be a loop, max_iter be an upper bound, and min_iter a lower bound on the number of iterations of L. From the abstract cost relation system for $\mathcal{L}$ (Definition 1) we infer the cost postconditions for $\mathcal{L}$:*

- $post_l(\overline{x}) = \mathtt{C_B}^{\mathtt{min}} + min\_iter(\overline{x}) \cdot \left(\sum_{j=1}^{n} \mathtt{ac}_{\mathtt{j}}^{l}\left(c_{j,1},\ldots,c_{j,h_{cj}}\right) + \sum_{i=1}^{m} \mathtt{C_{N_i}}^{\mathtt{min}}\right)$
- $post_u(\overline{x}) = \mathtt{C_B}^{\mathtt{max}} + max\_iter(\overline{x}) \cdot \left(\sum_{j=1}^{n} \mathtt{ac}_{\mathtt{j}}^{u}\left(c_{j,1},\ldots,c_{j,h_{cj}}\right) + \sum_{i=1}^{m} \mathtt{C_{N_i}}^{\mathtt{max}}\right)$

and generate the annotations

```
//@ assert post_l (x̄) ≤ cost
//@ assert cost ≤ post_u(x̄)
```

To infer the postcondition for a complete abstract program, we take the sum of all *cost postconditions* of its top-level loops plus the cost of the non-iterative fragments. Figure 4.1 shows the cost postconditions for our running example obtained by replacing the growth i of the invariant with the bound $\mathtt{t}$ on the loop iterations and requiring $\mathtt{t} \ge 0$.

The following theorem states soundness of cost postconditions:

Theorem 2. *Let $\mathcal{L}$ be a loop over variables $\overline{x}$ satisfying Definition 9, and $c_{\mathcal{L}}$ the abstract cost of $\mathcal{L}$. Let $\sigma_{\mathcal{L}} \in \mathbb{Z}^{m_n}$ be the state when $\mathcal{L}$ terminates. Then $post_l(\sigma_{\mathcal{L}}) \le c_{\mathcal{L}}(\sigma_{\mathcal{L}}) \le post_u(\sigma_{\mathcal{L}})$.*

4.3 Experimental Evaluation

The implemented prototype of [11] has been extended to include lower bounds in the analysis. The seven transformation examples of Section 2.3 have been processed in the updated tool. From these examples, five of them have now information about lower bounds included in their final analysis. The two examples that do not have lower bounds are the loop unrolling transformation, due to over-approximations, and the array optimization example, as LOBER is able to automatically infer lower bounds on the number of iterations, but not on the cost of a single iteration, as explained in Example 21.

In addition, we have studied two lower bound examples in which lower bounds on the best-case cost are not equal to lower bounds on the worst-case cost. One of them corresponds to a loop with the behaviour of Example 19 and the other example corresponds to the running example of Chapter 3. All the results obtained in these analyses satisfy the results that were expected by using the techniques explained in this chapter.

Chapter 5

Conclusions and Future Work

In Chapter 2, we have presented the first approach to analyze the cost of schematic programs with placeholders. We can infer and verify cost bounds for a potentially infinite class of programs once and for all. In particular, for the first time, it is possible to analyze and prove changes in efficiency caused by program transformations — valid for all input programs. *Quantitative Abstract Execution* supports exact, upper-bound and asymptotic cost and a configurable cost model. The toolchain is based on a cost analyzer and a program verifier which analyzes and formally certifies abstract cost bounds in a fully automated manner. Certification is essential, because when handling solely upper bounds, only the verifier can determine whether the bounds inferred by the cost analyzer are exact.

This work required the new concept of an *abstract cost invariant*. This is interesting in itself, because

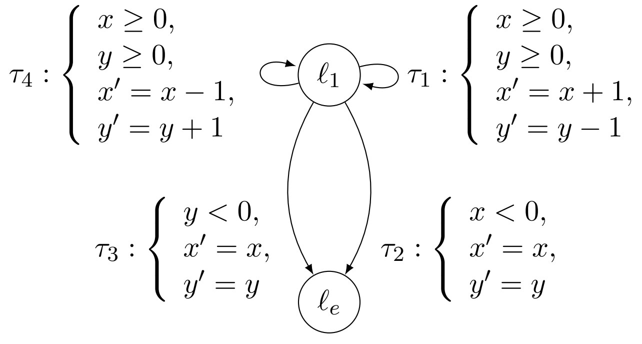
1. it renders the analysis of nested loops modular and

2. provides an interface to backends (such as verifiers) that characterizes the cost of code in iterations.

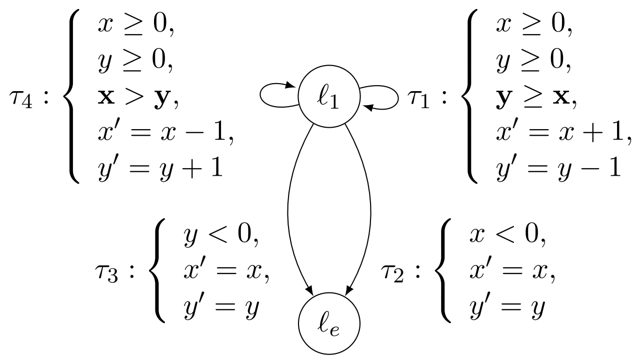
The future work in *Quantitative Abstract Execution* mentioned issues that have been already approached in the journal article on which Chapter 4 is based, such as the improvement of the analysis to get exact bounds, that has been achieved by including also the calculus of lower bounds. A new definition of cost neutral loops was also added to this chapter, in order to make the definition of the cost of an abstract program more general.

In Chapter 3, 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: non-termination analysis and lower bound inference.

In both kinds of analysis, one aims at finding classes of inputs for which the program features a non-terminating behavior or a cost-expensive behavior. Therefore, techniques developed for non-termination might provide a good basis for developing a lower bound analysis. In this sense, our work exploits ideas from the Max-SMT approach to non-termination in [35]. The main idea borrowed from [35] 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 and have been able to adapt the use of Max-SMT to accurately solve our problem (i.e., find larger bounds). Our loop specialization technique can be used to gain precision in non-termination analysis [35]. For instance, in this loop:


$\tau_4 : \begin{cases} x \geq 0, \\ y \geq 0, \\ x' = x - 1, \\ y' = y + 1 \end{cases}$
ℓ_1
$\tau_1 : \begin{cases} x \geq 0, \\ y \geq 0, \\ x' = x + 1, \\ y' = y - 1 \end{cases}$
$\tau_3 : \begin{cases} y < 0, \\ x' = x, \\ y' = y \end{cases}$
$\tau_2 : \begin{cases} x < 0, \\ x' = x, \\ y' = y \end{cases}$
ℓ_e

no sub strongly connected component (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 [35]. Instead, if we narrow the transitions as


$\tau_4 : \begin{cases} x \geq 0, \\ y \geq 0, \\ \mathbf{x} > \mathbf{y}, \\ x' = x - 1, \\ y' = y + 1 \end{cases}$
ℓ_1
$\tau_1 : \begin{cases} x \geq 0, \\ y \geq 0, \\ \mathbf{y} \geq \mathbf{x}, \\ x' = x + 1, \\ y' = y - 1 \end{cases}$
$\tau_3 : \begin{cases} y < 0, \\ x' = x, \\ y' = y \end{cases}$
$\tau_2 : \begin{cases} x < 0, \\ x' = x, \\ y' = y \end{cases}$
ℓ_e

we can prove that $x \geq 0 \land y \geq 0 \land x + y = 1$ is quasi-invariant, which allows us to prove non-termination in the way of [35] (as we will stay in the loop forever).

As regards lower bound inference, the current state-of-the-art is the work by Frohn et al. [27, 26] that introduces the notion of metering function and acceleration. Our work indeed tries to recover the semantic loss in [27, 26] 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. Indeed, the conditions of the definition of metering function from Chapter 3 already generalize the one in [27, 26] since it is not restricted to simple loops. This definition is improved with several loop specializations. While [27, 26] relies on pure SMT to solve the problem, we propose to gain precision using Max-SMT. Due to the different technical approaches underlying both frameworks, their accuracy and efficiency must be compared experimentally with the LoAT system that implements the ideas in [27, 26]. The results in the experimental section of the paper in which Chapter 3 is based 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 lower bounds for challenging cases in which LoAT fails.

In Chapter 4, *Quantitative Abstract Execution* and lower bound synthesis are unified. The abstract cost analysis framework, first based only on the COSTA analyzer [7] is extended in this chapter to use also LOBER [9] in order to include lower bounds in the analysis tool. This improvement of the analysis allows including the *exact* cost decision in the cost analysis stage and not leave it only to the verification. Previously, it was needed to verify the results to decide whether the inferred cost was exact or it was needed to use upper bounds. This was made in an incremental way: first trying the exact results and, then, the upper bounds. However, now a match between lower and upper bounds gives the assurance, in an early step, of the exactitude of the inferred results. The inclusion of lower bounds in this analysis turns our approach, that was to our knowledge the first approach to infer upper bounds on the cost of schematic programs with placeholders, also the first approach to infer lower bounds on the cost of abstract programs. In addition, the inclusion of verification in the process of the analysis gives an extra warranty on the correctness of the lower bound results and produces the needed formal proofs for a certification stage.

The future work of these techniques includes:

- In *Quantitative Abstract Execution*, the future work involves extending the analyzed target language. Cost analysis and deductive verification are already possible for a large JAVA fragment [7, 43]. More interesting—and more challenging—is the analysis of program transformations that parallelize code. The extension to larger classes of cost functions, such as logarithmic or exponential, could be realized by integrating more advanced cost analyzers into the toolchain.

- In *Lower Bound Synthesis*, when comparing the obtained results with LoAT we noticed that too many quasi-invariants were required, which made the SMT problem too hard. To improve these results, we could start, as a preprocessing step, from a quasi-invariant that includes all invariant inequalities that syntactically appear in the loop transitions. This is something similar to what is done by LoAT when inferring what they call conditional metering function.

  Moreover, the Max-SMT encoding is based on the use of soft constraints to obtain more accurate results. Adding soft conditions to the system is a field to be explored, as the inclusion of new requirements cannot only gain in precision of the results but also in the efficiency of the tool.

  In addition, the framework of lower bounds is now only applied to simple loops. A work to be done is to extend the implementation to handle also more complex programs, including nested loops.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *LNCS*. Springer, 2016.

[3] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, volume 5079 of *LNCS*, pages 221–237. Springer, 2008.

[4] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-form upper bounds in static cost analysis. *J. Autom. Reasoning*, 46(2):161–203, 2011.

[5] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of java bytecode. In *Programming Languages and Systems*, pages 157–172. Springer Berlin Heidelberg, 2007.

[6] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Resource Usage Analysis and its Application to Resource Certification. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 258–288. Springer, 2009.

[7] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.

[8] Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, Germán Puebla, and Guillermo Román-Díez. A formal verification framework for static analysis - as well as its instantiation to the resource analyzer COSTA and formal verification tool KeY. *Software and Systems Modeling*, 15(4):987–1012, 2016.

[9] Elvira Albert, Samir Genaim, Enrique Martin-Martin, Alicia Merayo, and Albert Rubio. Lower-bound synthesis using loop specialization and max-smt. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings,*

Part II, volume 12760 of *Lecture Notes in Computer Science*, pages 863–886. Springer, 2021.

[10] Elvira Albert, Samir Genaim, and Abu Naser Masud. On the inference of resource usage upper and lower bounds. *ACM Transactions on Computational Logic*, 14(3):1–35, aug 2013.

[11] Elvira Albert, Reiner Hähnle, Alicia Merayo, and Dominic Steinhöfel. Certified abstract cost analysis. In Esther Guerra and Mariëlle Stoelinga, editors, *Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12649 of *Lecture Notes in Computer Science*, pages 24–45. Springer, 2021.

[12] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[13] Roberto Bagnara, Fred Mesnard, Andrea Pescetti, and Enea Zaffanella. A new look at the automatic synthesis of linear ranking functions. *Inf. Comput.*, 215:47–67, 2012.

[14] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The barcelogic SMT solver. In *Computer Aided Verification*, pages 294–298. Springer Berlin Heidelberg, 2008.

[15] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, June 1975.

[16] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th Intl. Conf., TACAS, Grenoble, France*, volume 8413 of *LNCS*, pages 140–155. Springer, 2014.

[17] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer programs. *ACM Transactions on Programming Languages and Systems*, 38(4):1–50, oct 2016.

[18] Marc Brockschmidt, Richard Musiol, Carsten Otto, and Jürgen Giesl. Automated termination proofs for Java programs with cyclic data. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *LNCS*, pages 105–122. Springer, 2012.

[19] Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification*, pages 420–432. Springer Berlin Heidelberg, 2003.

[20] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[21] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978.

[22] Karl Crary and Stephanie Weirich. Resource bound certification. In Mark N. Wegman and Thomas W. Reps, editors, *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, pages 184–198. ACM, 2000.

[23] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower bound cost estimation for logic programs. In *Logic Programming*. The MIT Press, 1997.

[24] Antonio Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In *FM 2016: Formal Methods*, pages 254–273. Springer International Publishing, 2016.

[25] Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In Jacques Garrigue, editor, *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, volume 8858 of *LNCS*, pages 275–295. Springer, 2014.

[26] Florian Frohn, Matthias Naaf, Marc Brockschmidt, and Jürgen Giesl. Inferring lower runtime bounds for integer programs. *ACM Transactions on Programming Languages and Systems*, 42(3):1–50, dec 2020.

[27] Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl. Lower runtime bounds for integer programs. In *Automated Reasoning*, pages 550–567. Springer International Publishing, 2016.

[28] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th Intl. Joint Conf., IJCAR, Vienna, Austria*, volume 8562 of *LNCS*, pages 184–191. Springer, 2014.

[29] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '09*. ACM Press, 2008.

[30] Reiner Hähnle and Marieke Huisman. Deductive verification: from pen-and-paper proofs to industrial tools. In Bernhard Steffen and Gerhard Woeginger, editors,

Computing and Software Science: State of the Art and Perspectives, volume 10000 of *LNCS*, pages 345–373. Springer, 2019.

[31] Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. Aiming low is harder: induction for lower bounds in probabilistic program verification. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, jan 2020.

[32] Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs. In *Programming Languages and Systems*, pages 132–157. Springer Berlin Heidelberg, 2015.

[33] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 410–423, New York, NY, USA, 1996. Association for Computing Machinery.

[34] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[35] Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving non-termination using max-SMT. In *Computer Aided Verification*, pages 779–796. Springer International Publishing, 2014.

[36] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. JML Reference Manual, May 2013. Draft revision 2344.

[37] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: resource analysis for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, jun 2018.

[38] Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of java bytecode by term rewriting. In Christopher Lynch, editor, *RTA*, volume 6 of *LIPIcs*, pages 259–276. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[39] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Computer Aided Verification*, pages 745–761. Springer International Publishing, 2014.

[40] Moritz Sinn, Florian Zuleger, and Helmut Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of Automated Reasoning*, 59(1):3–45, jan 2017.

[41] Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for Java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.*, 32(3):8:1–8:70, 2010.

[42] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. *ACM SIGPLAN Notices*, 44(6):223–234, may 2009.

[43] Dominic Steinhöfel and Reiner Hähnle. Abstract execution. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *LNCS*, pages 319–336. Springer, 2019.

[44] Dominic Steinhöfel. *Abstract Execution: Automatically Proving Infinitely Many Programs*. PhD thesis, Technical University of Darmstadt, Department of Computer Science, Darmstadt, Germany, 2020.

[45] Ben Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[46] Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

[47] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.

[48] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction (extended version). *CoRR*, abs/1203.5303, 2012.

Part II

Papers of the thesis

Chapter 6

List of Publications

The list of papers, in chronological order, on which the thesis is based, is:

1. Elvira Albert, Reiner Hähnle, Alicia Merayo, and Dominic Steinhöfel. Certified Abstract Cost Analysis. *In Esther Guerra and Mariëlle Stoelinga, editors, Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Proceedings, volume 12649 of Lecture Notes in Computer Science, pages 24–45. Springer, 2021.*

2. Elvira Albert, Samir Genaim, Enrique Martín-Martín, Alicia Merayo, and Albert Rubio. Lower Bound Synthesis using Loop Specialization and Max-SMT. *In Alexandra Silva and K. Rustan M. Leino, editors, Computer Aided Verification - 33rd International Conference, CAV 2021, Proceedings, Part II, volume 12760 of Lecture Notes in Computer Science, pages 863–886. Springer, 2021.*

3. Elvira Albert, Reiner Hähnle, Alicia Merayo, and Dominic Steinhöfel. Certified Cost Bounds for Abstract Programs. *Under review in ACM Transactions on Software Engineering and Methodology (TOSEM).*

Certified Abstract Cost Analysis

Elvira Albert[1,2], Reiner Hähnle[3], Alicia Merayo[2](✉), and Dominic Steinhöfel[3,4]

[1] Instituto de Tecnología del Conocimiento, Madrid, Spain
[2] Complutense University of Madrid, Madrid, Spain. (✉ amerayo@ucm.es)
[3] Technische Universität Darmstadt, Darmstadt, Germany
[4] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Abstract. A program containing placeholders for unspecified statements or expressions is called an abstract (or schematic) program. Placeholder symbols occur naturally in program transformation rules, as used in refactoring, compilation, optimization, or parallelization. We present a generalization of automated cost analysis that can handle abstract programs and, hence, can analyze the impact on the cost of program transformations. This kind of relational property requires provably precise cost bounds which are not always produced by cost analysis. Therefore, we certify by deductive verification that the inferred abstract cost bounds are correct and sufficiently precise. It is the first approach solving this problem. Both, abstract cost analysis and certification, are based on quantitative abstract execution (QAE) which in turn is a variation of abstract execution, a recently developed symbolic execution technique for abstract programs. To realize QAE the new concept of a cost invariant is introduced. QAE is implemented and runs fully automatically on a benchmark set consisting of representative optimization rules.

1 Introduction

We present a generalization of automated cost analysis that can handle programs containing placeholders for unspecified statements. Consider the program $Q \equiv$ "$\mathsf{i}=0$; **while** $(\mathsf{i} < \mathsf{t})$ $\{\mathsf{P};\ \mathsf{i}{+}{+};\}$", where P is any statement not modifying i or t. We call P an *abstract statement*; a program like Q containing abstract statements is called *abstract program*. The (exact or upper bound) cost of executing P is described by a function $\mathtt{ac}_{\mathsf{P}}(\overline{x})$ depending on the variables $\overline{x}$ occurring in P. We call this function the *abstract cost* of P. Assuming that executing any statement has unit cost and that $t \geq 0$, one can compute the (abstract) cost of Q as $2 + t \cdot (\mathtt{ac}_{\mathsf{P}}(\overline{x}) + 2)$ depending on $\mathtt{ac}_{\mathsf{P}}$ and t. For any concrete instance of P, we can derive its concrete cost as usual and then obtain the concrete cost of Q simply by instantiating $\mathtt{ac}_{\mathsf{P}}$. In this paper, we define and implement an abstract cost analysis to infer abstract cost bounds. Our implementation consists of an automatic abstract cost analysis tool and an automatic certifier for the correctness of inferred abstract bounds. Both steps are performed with an approach called *Quantitative Abstract Execution* (QAE).

Fine, but what is this good for? Abstract programs occur in program transformation rules used in compilation, optimization, parallelization, refactoring, etc.: Transformations are specified as rules over *program schemata* which are nothing but abstract programs. If we can perform cost analysis of abstract programs, we can *analyze the cost effect of program transformations.* Our approach is the *first method to analyze the cost impact of program transformations.*

Automated Cost Analysis. Cost analysis occupies an interesting middle ground between termination checking and full functional verification in the static program analysis portfolio. The main problem in functional verification is that one has to come up with a functional specification of the intended behavior, as well as with auxiliary specifications including loop invariants and contracts [21]. In contrast, termination is a generic property and it is sufficient to come up with a suitable term order or ranking function [6]. For many programs, termination analysis is vastly easier to automate than verification.[1]

Computation cost is not a generic property, but it is usually schematic: One fixes a class of cost functions (for example, polynomial) that can be handled. A cost analysis then must come up with parameters (degree, coefficients) that constitute a valid bound (lower, upper, exact) for all inputs of a given program with respect to a cost model (# of instructions, allocated memory, etc.). If this is performed bottom up with respect to a program's call graph, it is possible to *infer* a cost bound for the top-level function of a program. Such a cost expression is often *symbolic*, because it depends on the program's input parameters.

A central technique for inferring symbolic cost of a piece of code with high precision is *symbolic execution* (SE) [9, 25]. The main difficulty is to render SE of loops with symbolic bounds finite. This is achieved with *loop invariants* that generalize the behavior of a loop body: an invariant is valid at the loop head after arbitrarily many iterations. To infer sufficiently strong invariants automatically is generally an unsolved problem in functional verification, but much easier in the context of cost analysis, because invariants do not need to characterize functional behavior: it suffices that they permit to infer schematic cost expressions.

Abstract Execution. To infer the cost of program transformation *schemata* requires the capability of analyzing abstract programs. *This is not possible with standard SE*, because abstract statements have no operational semantics. One way to reason about abstract programs is to perform structural induction over the syntactic definition of statements and expressions whenever an abstract symbol is encountered. Structural induction is done in interactive theorem proving [7, 31] to verify, e.g., compilers. It is labor-intensive and not automatic. Instead, here we perform cost analysis of abstract programs via a recent generalization of SE called abstract execution (AE) [37,38]. The idea of AE is, quite simply, to symbolically execute a program containing abstract placeholder symbols for expressions and statements, just as if it were a concrete program. It might seem

[1] In theory, of course, proving termination is as difficult as functional verification. It is hard to imagine, for example, to find a termination argument for the Collatz function without a deep understanding of what it does. But automated termination checking works very well for many programs in practice.

counterintuitive that this is possible: after all, nothing is known about an abstract symbol. But this is not quite true: one can equip an abstract symbol with an *abstract* description of the behavior of its instances: a set of memory locations its behavior may depend on, commonly called *footprint* and a (possibly different) set of memory locations it can change, commonly called *frame* [21].

Cost Invariants. In automated cost analysis, one infers cost bounds often from loop invariants, ranking functions, and size relations computed during SE [3,11,16,40]. For *abstract* programs, we need a more general concept, namely a loop invariant expressing a *valid abstract cost bound* at the beginning of any iteration (e.g., $2 + i * (\mathtt{ac}_{\mathsf{P}}(\overline{x}) + 2)$ for the program Q above). We call this a *cost invariant.* This is an important technical innovation of this paper, increasing the modularity of cost analysis, because each loop can be verified and certified separately.

Relational Cost Analysis. AE allows specifying and verifying *relational* program properties [37], because one can express rule schemata. This extends to QAE and makes it possible, for the first time, to infer and to prove (automatically!), for example, the impact of program transformation on performance.

Certification. Cost annotations inferred by abstract cost analysis, i.e., cost invariants and abstract cost bounds, are automatically *certified* by a deductive verification system, extending the approach reported in [4] to abstract cost and abstract programs. This is possible because the specification (i.e., the cost bound) and the loop (cost) invariants are inferred by the cost analyzer—the verification system does not need to generate them.

To argue correctness of an abstract cost analysis is complex, because it must be valid for an infinite set of concrete programs. For this reason alone, it is useful to certify the abstract cost inferred for a given abstract program: during development of the abstract cost analysis reported here, several errors in abstract cost computation were detected—analysis of the failed verification attempt gave immediate feedback on the cause. We built a test suite of problems so that any change in the cost analyzer can be validated in the future.

Certification is crucial for the correctness of quantitative relational properties: The inferred cost invariants might not be precise enough to establish, e.g., that a program transformation does not increase cost for any possible program instance and run. This is only established at the certification stage, where relational properties are formally verified. *A relational setting requires provably precise cost bounds.* This feature is not offered by existing cost analysis methods.

2 QAE by Example

We introduce our approach and terminology informally by means of a motivating example: *Code Motion* [1] is a compiler optimization technique moving a statement not affected by a loop from the beginning of the loop body to before the loop. This code transformation should preserve behavior provided the loop is executed at least once, but can be expected to improve computation effort, i.e. *quantitative* properties of the program, such as execution time and memory

Program Before

```
int i = 0;
//@ loop_invariant i ≥ 0 && i ≤ t;
//@ cost_invariant
      i · (ac_P (t, w) + ac_Q (t, z) + 2) ;
//@ decreases t − i;
while (i < t) {
      //@ assignable x;
      //@ accessible t, w;
      //@ cost_footprint t, w;
      \abstract_statement P;
      //@ assignable y;
      //@ accessible i, t, y, z;
      //@ cost_footprint t, z;
      \abstract_statement Q;
      i++;
}
//@ assert \cost == 2 +
      t · (ac_P (t, w) + ac_Q (t, z) + 2) ;
```

Program After

```
int i = 0;
//@ assignable x;
//@ accessible t, w;
//@ cost_footprint t, w;
\abstract_statement P;
//@ loop_invariant i ≥ 0 && i ≤ t;
//@ cost_invariant
      i · (ac_Q (t, z) + 2) ;
//@ decreases t − i;
while (i < t) {
      //@ assignable y;
      //@ accessible i, t, y, z;
      //@ cost_footprint t, z;
      \abstract_statement Q;
      i++;
}
//@ assert \cost == 2 +
      ac_P (t, w) + t · (ac_Q (t, z) + 2) ;
```

Preconditions and Postconditions

Inputs: t, w, x, y, z Precondition: $t > 0$ Postcondition: $\backslash$**cost_1** $\geq$ $\backslash$**cost_2**

Fig. 1: Motivating example on relational quantitative properties.

consumption: The moved code block is executed just once in the transformed context, leading to less instructions (less energy consumed) and, in case it allocates memory, less memory usage. In the following we subsume any quantitative aspect of a program under the term *cost* expressed in an unspecified *cost model* with the understanding that it can be instantiated to specific cost measures, such as number of instructions, number of allocated bytes, energy consumed, etc.

To formalize code motion as a transformation rule, we describe in- and output of the transformation *schematically*. Fig. 1 depicts such a schema in a language based on JAVA. An *Abstract Statement* (AS) with identifier *Id*, declared as "\abstract_statement *Id*;", represents an arbitrary concrete statement. It is obviously unsafe to extract arbitrary, possibly non-invariant, code blocks from loops. For this reason, the AS P in question has a *specification* restricting the allowed behavior of its instances. For compatibility with JAVA we base our specification language on the *Java Modeling Language* (JML) [27]. Specifications are attached to code via structured comments that are marked as JML by an "@" symbol. JML keyword "**assignable**" defines the memory locations that may occur in the frame of an AS; similarly, "**accessible**" restricts the footprint. Fig. 1 contains further keywords explained below.

Input to QAE is the abstract program to analyze, including annotations (highlighted in light gray in Fig. 1) that express restrictions on the permitted instances of ASs. In addition to the frame and footprint, the *cost footprint* of an AS, denoted with the keyword "**cost_footprint**", is a subset of its footprint listing locations the cost expressions in AS instances may depend on. In Fig. 1, the cost footprint of AS Q excludes accessible variables i and y. Annotations highlighted in dark gray are *automatically inferred* by abstract cost analysis and are input for the certifier. As usual, loop invariants (keyword "**loop_invariant**") are needed to describe the behavior of loops with symbolic bounds. The loop invariant in Fig. 1 allows inferring the final value t of loop counter i after loop termination. To prove termination, the loop *variant* (keyword "**decreases**") is inferred.

So far, this is standard automated cost analysis [3]. The ability to *infer automatically* the remaining annotations represents our main contribution: Each AS P has an associated *abstract cost* function parametric in the locations of its footprint, represented by an abstract cost symbol ac_P. The symbol $\mathsf{ac}_\mathsf{P}(\mathsf{t}, \mathsf{w})$ in the "**assert**" statement in Fig. 1 can be instantiated with any concrete function parametric in t, w being a valid cost bound for the instance of P. For example, for the instantiation "$\mathsf{P} \equiv$ x=t+1;" the constant function $\mathsf{ac}_\mathsf{P}(\mathsf{t}, \mathsf{w}) = 1$ is the correct *exact* cost, while $\mathsf{ac}_\mathsf{P}(\mathsf{t}, \mathsf{w}) = \mathsf{t}$ with $\mathsf{t} \geq 1$ is a correct *upper bound* cost.

As pointed out in Sect. 1 we require *cost invariants* to capture the cost of each loop iteration. They are declared by the keyword "**cost_invariant**". To generate them, it is necessary to infer the *cost growth* of abstract programs that bounds the number of loop iterations executed so far. In Sect. 4 we describe automated inference of cost invariants including the generation of cost growth for all loops. Our technique is compositional and also works in the presence of nested loops.

The QAE framework can express and prove quantitative relational properties. The assertions in the last lines in Fig. 1 use the expression \cost referring to the total accumulated cost of the program, i.e., the quantitative *postcondition*. We support quantitative relational postconditions such as \cost_1 $\geq$ \cost_2, where \cost_1, \cost_2 refer to the total cost of the original (on the left) and transformed (on the right) program, respectively. To prove relational properties, one must be able to deduce *exact* cost invariants for loops such that the comparison of the invariants allows concluding that the programs from which the invariants are obtained fulfill the proven relational property. Otherwise, over-approximation introduced by cost analysis could make the relation for the postconditions hold, while the relational property does not necessarily hold for the programs.

To obtain a formal account of QAE with correctness guarantees we require a mathematically rigorous semantic foundation of abstract cost. This is provided in the following section.

3 (Quantitative) Abstract Execution

Abstract Execution [37,38] extends symbolic execution by permitting abstract statements to occur in programs. Thus AE reasons about an *infinite* set of concrete programs. An abstract program contains at least one AS. The semantics of an AS is given by the set of concrete programs it represents, its set of *legal instances*. To simplify presentation, we only consider normally completing Java code as instances: an instance may not throw an exception, break from a loop, etc. Each AS has an *identifier* and a specification consisting of its frame and footprint. Semantically, instances of an AS with identifier P may at most write to memory locations specified in P's frame and may only read the values of locations in its footprint. All occurrences of an AS with the *same identifier* symbol have the same legal instances (possibly modulo renaming of variables, if variable names in frame and footprint specifications differ). For example, by

```
//@ assignable x,y;
//@ accessible y, z;
\abstract_statement P;
```

we declare an AS with identifier "P", which can be instantiated by programs that write at most to variables x and y, while only depending on variables y and z. The program "x=y; y=17;" is a legal instance of it, but not "x=y; y=w;", which accesses the value of variable w not contained in the footprint.

We use the shorthand $\mathsf{P}(\mathsf{x},\mathsf{y} :\approx \mathsf{y},\mathsf{z})$ for the AS declaration above. The left-hand side of ":≈" is the frame, the right-hand side the footprint. Abstract programs allow expressing a second-order property such as "all programs assigning at most x, y while reading at most y, z leave the value of i unchanged". In *Hoare triple* format (where i_0 is a fresh constant not occurring in P):

$$\{\mathsf{i} \doteq i_0\}\, \mathsf{P}(\mathsf{x},\mathsf{y} :\approx \mathsf{y},\mathsf{z});\, \{\mathsf{i} \doteq i_0\} \qquad (*)$$

3.1 Abstract Execution with Abstract Cost

We extend the AE framework [37,38] to QAE by adding *cost specifications* that extend the specification of an AS with an annotated *cost expression*. An abstract cost expression is a function whose value may depend on any memory location in the footprint of the AS it specifies. This location set is called the *cost footprint*, specified via the **cost_footprint** keyword (see Fig. 1), and must be a subset of the footprint of the specified AS. The cost footprint for the program in (∗) might be declared as "$\{z\}$". It implicitly declares the abstract function $\mathsf{ac}_\mathsf{P}(z)$ that could be instantiated to, say, quadratic cost "z^2".

Definition 1 (Abstract Program). *A pair $\mathcal{P} = (abstrStmts, p_{abstr})$ of a set of AS declarations $abstrStmts \neq \emptyset$ and a program fragment p_{abstr} containing exactly those ASs is called* abstract program. *Each AS declaration in abstrStmts is a pair $(\mathsf{P}(frame :\approx footprint), \mathsf{ac}_\mathsf{P}(costFootprint))$, where* P *is an identifier; frame, footprint, and $costFootprint \subseteq footprint$ are location sets.*

A concrete program fragment p is a legal instance *of $\mathcal{P}$ if it arises from substituting concrete cost functions for all* ac_P *in abstrStmts, and concrete statements for all* P *in abstrStmts, where (i) all ASs are instantiated legally, i.e., by statements respecting their frame, footprint, and cost function, and (ii) all ASs with the same identifier are instantiated with the same concrete program. The semantics $[\![\mathcal{P}]\!]$ consists of all its legal instances.*

The abstract program consisting of only AS P in (∗) with cost footprint "{z}" is formally defined as: $\big(\{(\mathsf{P}(\mathsf{x},\mathsf{y} :\approx \mathsf{y},\mathsf{z}), \mathsf{ac}_\mathsf{P}(\mathsf{z}))\}, \mathsf{P};\big)$. The program "$\mathsf{P}^0 \equiv$ i =0; **while** (i <z) {x = z; i ++;}" with cost function "$\mathsf{ac}_\mathsf{P}(\mathsf{z}) = 3 \cdot z + 2$" is a legal instance: it respects frame, footprint, and cost footprint, as well as the cost function, that (assuming $\mathsf{z} \geq 0$) can be obtained by static cost analysis of P^0.

By encoding the semantics of abstract programs in a program logic [38, Sect. 4.2] one can statically verify whether an instance is legal. It may require auxiliary specifications (invariants, contracts) of the concrete code. The property is undecidable, but can be proven automatically in many cases, see [38] for a discussion. A first implementation of such a check is part of the REFINITY tool (see [36], also https://www.key-project.org/REFINITY/).

3.2 Cost of Abstract Programs

Finitely executing a concrete program p starting in a state $s_0 = (p, \sigma_0)$ with an initial assignment σ_0 of p's program variables results in a finite trace of the form $t \equiv s_0 \xrightarrow{c_1} \dots \xrightarrow{c_n} s_n$. Each state $s_i = (p_i, \sigma_i)$ consists of a program counter p_i (the remaining program to execute) and a store σ_i (the current variable assignment); each transition $s_i \xrightarrow{c_{i+1}} s_{i+1}$ updates s_i to s_{i+1} according to the effect of executing command c_{i+1} defined in the semantics of the programming language. A *complete* trace corresponds to a terminating execution, i.e., $s_n = (\epsilon, \sigma_n)$, where ϵ is the empty program and σ_n the resulting final variable assignment.

The cost of a program can be computed based on execution traces. To allow arbitrary quantitative properties, we work on a generic *cost model* $\mathcal{M}$ that assigns cost values to programming language instructions. We will compute the cost of a trace t, denoted $\mathcal{M}(t)$, by summing up the costs of the executed instructions. A straightforward measure is the number of executed instructions $\mathcal{M}_{\mathsf{instr}}$: In this cost model, instructions like "x=1;", the evaluation of the loop guard, etc., all are assigned cost 1. For example, the cost of the complete trace of "**while** (i >0) i−−;" when started with an initial store assigning the value 3 to i is 7, because "i −−;" is executed three times and the guard is evaluated four times. This can be generalized to *symbolic* execution: Executing the same program with a *symbolic* store assigning to i a symbolic initial value $i_0 \geq 0$ produces traces of cost $2 \cdot i_0 + 1$. The cost of *abstract programs*, i.e., the generalization to QAE, is defined similarly: By generalizing not merely over all initial stores, but also over all concrete instances of the abstract program.

Definition 2 (Abstract Program Cost). *Let $\mathcal{M}$ be a cost model. Let an integer-valued expression $c_\mathcal{P}$ consist of scalar constants, program variables, and abstract cost symbols applied to constants and variables. Expression $c_\mathcal{P}$ is the* cost of an abstract program $\mathcal{P}$ w.r.t. $\mathcal{M}$ *if for all concrete stores σ and instances $p \in [\![\mathcal{P}]\!]$ such that p terminates with a complete trace t of cost $\mathcal{M}(t)$ when executed in σ, $c_\mathcal{P}$ evaluates to $\mathcal{M}(t)$ when interpreting variables according to σ, and abstract cost functions according to the instantiation step leading to p. The instance of $c_\mathcal{P}$ using the concrete store σ is denoted $c_\mathcal{P}(\sigma)$.*

Example 1. We test the cost assertion in the last lines of the left program in Fig. 1 by computing the cost of a trace obtained from a fixed initial store and instances of P, Q. We use the cost model $\mathcal{M}_{\mathsf{instr}}$ and an initial store that assigns 2 to t and 0 to all other variables. We instantiate P with "x=2∗t;" and Q with "y=i; y++;". Consequently, the abstract cost functions $\mathsf{ac_P}\,(\mathsf{t},\mathsf{w})$ and $\mathsf{ac_Q}\,(\mathsf{t},\mathsf{z})$ are instantiated with 1 and 2, respectively. Evaluating the postulated abstract program cost $2 + \mathsf{t} \cdot (2 + \mathsf{ac_P}\,(\mathsf{t},\mathsf{w}) + \mathsf{ac_Q}\,(\mathsf{t},\mathsf{z}))$ for the concrete store and AS instantiations results in $2+2\cdot(2+1+2) = 12$. Consequently, the execution trace should contain 12 transitions, which is the case.

3.3 Proving Quantitative Properties with QAE

There are two ways to realize QAE on top of the existing functional verification layer provided by the AE framework [37, 38]: (i) provide a "cost" extension to the program logic and calculus underlying AE; (ii) translate non-functional (cost) properties to functional ones. We opt for the second, as it is less prone to introduce soundness issues stemming from the addition of new concepts to the existing framework. It is also faster to realize and allows early testing.

The translation consists of three elements: (a) A global "ghost" variable "cost" (representing keyword "**cost**") for tracking accumulated cost; (b) explicit encoding of a chosen cost model by suitable ghost setter methods that update this variable; (c) functional loop invariants and method postconditions expressing cost invariants and cost postconditions.

Regarding item (c), we support three kinds of cost specification. These are, descending in the order of their strength: *exact*, *upper bound*, and *asymptotic* cost. At the analysis stage, it is usually impossible to determine the best match. For this reason, there is merely one **cost_invariant** keyword, not three. However, when translating cost to functional properties, a decision has to be made. A natural strategy is to start with the strongest kind of specification, then proceed towards the weaker ones when a proof fails.

An exact cost invariant has the shape "cost == *expr*", an upper bound on the invariant cost is specified by "cost <= *expr*"; asymptotic cost is expressed by the idiom "asymptotic(cost) <= asymptotic(*expr*)". The function "asymptotic" abstracts from constant symbols in the argument. For example, the (exact) cost postcondition of the abstract program on the right in Fig. 1 is:

$$\mathsf{cost} \ == 2 + \mathsf{ac_P}\,(\mathsf{t},\mathsf{w}) + \mathsf{t}\cdot(\mathsf{ac_Q}\,(\mathsf{t},\mathsf{z}) + 2) \qquad (\dagger)$$

Asymptotic cost would be expressed as $\mathsf{asymptotic(cost)} \ <= \mathsf{asymptotic}(2 + \mathsf{ac_P}\,(\mathsf{t},\mathsf{w}) + \mathsf{t}\cdot(\mathsf{ac_Q}\,(\mathsf{t},\mathsf{z}) + 2))$ where the right-hand side of the equation is equivalent to $\mathsf{asymptotic}(\mathsf{ac_P}\,(\mathsf{t},\mathsf{w}) + \mathsf{t}\cdot(\mathsf{ac_Q}\,(\mathsf{t},\mathsf{z})))$.

Listing 2 shows the result of translating the cost invariant in Fig. 1 to a functional loop invariant (highlighted lines), using cost model $\mathcal{M}_{\mathsf{instr}}$ in ghost setters and postconditions of AS ("**ensures**" clauses). ASs P, Q must include the ghost variable "cost" in their frame, because they update its value. The keyword **before** in the postcondition of an AS refers to the value a variable had just before executing the AS. In loops we use "inner" cost variables "iCost" tracking the cost inside the loop. When the loop terminates, we add the final value of "iCost" to "cost". After every evaluation of the guard of the loop, the cost is incremented accordingly. Using the translation in Listing 2 of the inferred annotations in Fig. 1, the AE system proves cost postcondition (†) automatically.

Apart from the translation of inferred quantitative annotations to functional AE specifications, we implemented the axiomatization of the asymptotic function and extended the AE system's *proof script* language. This made it possible to define a highly automated proof strategy for non-linear arithmetic problems generated by some cost analysis benchmarks.

4 Abstract Cost Analysis

Recall from Sect. 2 that for automatic cost certification we need to infer annotations for abstract cost invariants and cost postconditions. To achieve this, we

```
//@ ghost int cost = 0;
int i = 0;
//@ set cost = cost + 1;

//@ assignable x, cost;
//@ accessible t, w;
//@ ensures cost == \before(cost)
//@     + ac_P (t, w);
\abstract_statement P;

//@ ghost int iCost = 0;
//@ loop_invariant i ≥ 0 && i ≤ t
//@  && iCost == i · (ac_Q (t, z) + 2);
//@ decreases t − i;
while (i < t) {
    //@ set iCost = iCost + 1;
    //@ assignable y, cost;
    //@ accessible i, t, y, z;
    //@ ensures cost ==
    //@    \before(cost) + ac_Q (t, z);
    \abstract_statement Q;
    i++;
    //@ set iCost = iCost + 1;
}
//@ set cost = cost + 1;
//@ set cost = cost + iCost;
```

Listing 2: Translation of cost model and cost invariants to AE.

leverage a cost analysis framework for concrete programs to the abstract setting. The presentation is structured as follows: Sect. 4.1 defines the notion of an abstract cost relation system (ACRS) used in cost analysis for the abstract setting. Sect. 4.2 details how to generate automatically inductive cost invariants for abstract programs from ACRSs. Sect. 4.3 tells how to generate cost postconditions used to prove relational properties and required to handle nested loops.

4.1 Inference of Abstract Cost Relations

There are two main cost analysis approaches: those using recurrence equations in the style of Wegbreit [39], and those based on type systems [14, 24]. Our formalization is based on the first kind, but the main ideas for extending the framework to abstract programs would be also applicable to the second. The key issue when extending a recurrences-based framework to the abstract setting is the notion of *abstract cost relation* for loops which generalizes the concept of cost recurrence equations for a loop to an abstract setting. We start with notation for loops and technical details on assumed size relations.

Loops. In our formalization we consider while-loops containing n abstract statements and m non-abstract statements. Non-abstract statements include any concrete instruction of the target language (arithmetic instructions, conditionals, method calls, ...). We assume loops L have the general outline displayed on the right. Each abstract statement has a frame specification, abstract and non-abstract statements may appear in any order, either might be empty.

```
while (G) {
  //@ accessible r_{1,1}, ..., r_{1,h_{r1}}
  //@ assignable w_{1,1}, ..., w_{1,h_{w1}}
  //@ cost_footprint c_{1,1}, ..., c_{1,h_{c1}}
  \abstract_statement A_1;
  non_abstract_statement N_1;
  ...
}
```

Size relations. We assume that for each loop sets of *size constraints* have been computed. These sets capture the size relation among the variables in the loop upon exit (called *base case*, denoted φ_B), and when moving from one iteration to the next (denoted φ_I). ASs are ignored by the size analysis. While this would be unsound in general, it will be correct under the requirements we impose in Def. 4 and with the handling of ASs in Def. 3. Size relations are available from any cost analyzer by means of a static analysis [13] that records the effect of concrete program statements on variables and propagates it through each loop iteration. In our examples, since we work on integer data, size analysis corresponds to a value analysis [10] tracking the value of the integer variables.[2]

Example 2. The size relations for the loop on the left in Fig. 1 are $\varphi_B = \{\mathsf{i} \geq \mathsf{t}\}$ and $\varphi_I = \{\mathsf{i} < \mathsf{t},\ \mathsf{i}' = \mathsf{i} + 1\}$. φ_B is inferred from the loop guard and φ_I from the guard and the increment of i (primed variables refer to the value of the variable after the loop execution).

Based on pre-computed size relations, we define the cost of executing a loop by means of an *abstract cost relation system* (ACRS). This is a set of cost equations characterizing the abstract cost of executing a loop for any input with respect to a given cost model $\mathcal{M}$. Cost equations consist of a cost expression governed by size constraints containing applicability conditions for the equation (like $\mathsf{i} < \mathsf{t}$ in φ_I above) and size relations between loop variables (like $\mathsf{i}' = \mathsf{i} + 1$ in φ_I).

Definition 3 (Abstract Cost Relation System). *Let L be a loop as above with n abstract and m non-abstract statements. Let $\overline{x}$ be the set of variables accessed in L. Let φ_I, φ_B be sound size relations for L, and $\mathcal{M}$ a cost model. The ACRS for L is defined as the following set of cost equations:*

$$\begin{array}{ll} C(\overline{x}) = \mathtt{C_B} & ,\ \varphi_B \\ C(\overline{x}) = \sum_{j=1}^{n} \mathtt{ac_j}\left(c_{j,1},\ldots,c_{j,h_{cj}}\right) + \sum_{i=1}^{m} \mathtt{C_{N_i}} + C(\overline{x}'), & \varphi_I \end{array}$$

where:

(1) $\mathtt{C_B} \geq 0$ is the cost of exiting the loop (executing the base case) w.r.t. $\mathcal{M}$.
(2) Each $\mathtt{ac_j}(\cdot) \geq 0$ represents the abstract cost for the abstract statement A_j in L w.r.t. to $\mathcal{M}$. Each $\mathtt{ac_j}$ is parameterized with the variables in the cost footprint of the corresponding A_j, as it may depend on any of them.
(3) Each $\mathtt{C_{N_i}} \geq 0$ is the cost of the non-abstract statement N_i w.r.t. to $\mathcal{M}$.
(4) C is a recursive call.
(5) $\overline{x}'$ are variables $\overline{x}$ when renamed after executing the loop.
(6) The assignable variables $w_{j,*}$ in the $\mathtt{ac_j}$ get an unknown value in $\overline{x}'$ (denoted with "_" in the examples below).

Ignoring the abstract statements, one can apply a complete algorithm for cost relation systems [6] to an ACRS to obtain automatically a *linear*[3] ranking function f for loop L: f is a linear, non-negative function over $\overline{x}$ that decreases strictly at every loop iteration. Function f yields directly the "`//@` **decreases** f;" annotation required for QAE.

As in Sect. 3, the definition of ACRS assumes a generic cost model $\mathcal{M}$ and uses $\mathtt{C}$ to refer in a generic way to cost according to $\mathcal{M}$. For example, to infer the number of executed steps, $\mathtt{C}$ is set to 1 per instruction, while for memory usage $\mathtt{C}$ records the amount of memory allocated by an instruction.

[2] For complex data structures, one would need heap analyses [35] to infer size relations.
[3] There exist (more expensive) algorithms to obtain also polynomial ranking functions [5] but for the sake of efficiency we are not using them in our system.

General Case of ACRS. The definition of ACRS was simplified for presentation. The following generalizations, not requiring any new concept, are possible: (1) We assume an ACRS for a loop has only two equations, one for the base case (the guard G does not hold) and one for the iterative case (G holds). In general, there might be more than one equation for the base case, e.g., if the guard involves multiple conditions and the cost varies depending on the condition that holds on the exit. Similarly, there might be multiple equations in the iterative case, e.g., if the loop body contains conditional statements and each iteration has different cost depending on the taken branch. This issue is orthogonal to the extension to abstract cost. (2) A loop might contain method calls that in turn contain ASs. In absence of recursion, such calls can be inlined. For recursive methods, it is possible to compute the call graph and solve the equations in reverse topological order such that the abstract cost of the (inner) method calls is obtained first and then inserted into the surrounding equations. (3) The cost of code fragments not part of any loop (before, after, and in between loops) is defined as well by abstract cost equations accumulating the cost of all instructions these fragments include, just as for concrete programs. This aspect does not require changes to the framework for concrete programs, so we do not formalize it, but just illustrate it in the next example.

Example 3. The ACRSs of the programs in Fig. 1 are (left program above line, right program below):

$$
\begin{array}{rlr}
C_{\text{before}}(\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) & = c_{\text{before}} + C_{w_0}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}), & \{\mathsf{i}=0\}\\
C_{w_0}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) & = c_{B_{w_0}}, & \{\mathsf{i}\geq\mathsf{t}\}\\
C_{w_0}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) & = c_{w_0} + \mathsf{ac_P}\,(\mathsf{t},\mathsf{w}) + \mathsf{ac_Q}\,(\mathsf{t},\mathsf{z}) + C_{w_0}(\mathsf{i}',\mathsf{t},\_,\mathsf{w},\_,\mathsf{z}), & \{\mathsf{i}'=\mathsf{i}+1,\mathsf{i}<\mathsf{t}\}\\
\hline
C_{\text{after}}(\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) & = c_{\text{after}} + \mathsf{ac_P}\,(\mathsf{t},\mathsf{w}) + C_{w_1}(\mathsf{i},\mathsf{t},\_,\mathsf{w},\mathsf{y},\mathsf{z}), & \{\mathsf{i}=0\}\\
C_{w_1}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) & = c_{B_{w_1}}, & \{\mathsf{i}\geq\mathsf{t}\}\\
C_{w_1}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) & = c_{w_1} + \mathsf{ac_Q}\,(\mathsf{t},\mathsf{z}) + C_{w_1}(\mathsf{i}',\mathsf{t},\mathsf{x},\mathsf{w},\_,\mathsf{z}), & \{\mathsf{i}'=\mathsf{i}+1,\mathsf{i}<\mathsf{t}\}
\end{array}
$$

Notation c refers to the generic cost that can be instantiated to a chosen cost model $\mathcal{M}$. Cost equation C_{before} for the first program is composed of the instructions appearing before the loop is c_{before} plus the cost of executing the while loop C_{w_0}. The size constraint fixes the initial value of i. Following Def. 3, there are two equations corresponding to the base case of the loop and executing one iteration, respectively. Observe that assignable variables in ASs have unknown values in the ACRS (according to item (6) in Def. 3). Program *after* has a similar structure. A ranking function for both loops is $\mathsf{t}-\mathsf{i}$ which is used to generate the annotation "`//@ decreases t−i;`" inserted just before each loop in Fig. 1.

To guarantee soundness of abstract cost analysis, it is mandatory that (i) no AS in the loop modifies any of the variables that influence loop cost, i.e., they do not *interfere with cost*, and (ii) the cost of the AS in the loop is independent of the variables modified in the loop. We call the latter ASs *cost neutral*. The first requirement is guaranteed by item (6) in Def. 3, because the value of assignable variables is "forgotten" in the equations. It is implemented, as usual in static analysis, by using a name generator for *fresh* variables. If cost depends on assignable variables in an AS, then the ACRS will not be solvable (i.e., the analysis returns "unbound cost"). The ACRS in the example contains "_" in equations that do not prevent solvability of the system nor its evaluation, because they do not interfere with cost. However, if we had "forgotten" a cost-relevant variable (such as $\mathtt{t}$), we would be unable to solve or evaluate the equations: without knowing $\mathtt{t}$ the equation guard is not evaluable. Requirement (ii) is ensured by the following definition ensuring that variables in the cost footprint are not modified by other statements in the loop.

Definition 4 (Cost neutral AS). *Given a loop L, where*

- *$W(L)$ is the set of variables written by the non-abstract statements of L.*
- *$\mathtt{Abstr}(L)$ is the set of all ASs in loop L.*
- *$\mathit{Frame}(\mathtt{Abstr}(L))$ is the set of variables assigned by any AS $A \in \mathtt{Abstr}(L)$.*
- *$\mathit{CostFootprint}(A)$ is the set of variables which the cost of an A depends on.*

L is a loop with cost neutral *ASs if, for all $A \in \mathtt{Abstr}(L)$, it is the case that $(W(L) \cup \mathit{Frame}(\mathtt{Abstr}(L))) \cap \mathit{CostFootprint}(A) = \emptyset$.*

The definition above constitutes a sufficient, but not necessary criterion that could be tightened by a more expensive analysis. For instance, our framework easily extends to allow conditions in the cost footprint that the concretizations of the AS must fulfill. In our example, the cost footprint might include condition $i' \geq i$, where i' is the value of i after executing the AS. This permits the abstract statement to modify i provided it does not decrease its value. Thus, the AS is not cost neutral, but the upper bound remains sound. The formalization of this generalization is left to future work.

Example 4. It is easy to check that both loops in Fig. 1 have cost neutral ASs. On the left: $W(L) = \{\mathsf{i}\}$, $\mathit{Frame}(\{P, Q\}) = \{\mathsf{x}, \mathsf{y}\}$, $\mathit{CostFootprint}(P) = \{\mathtt{t}, \mathsf{w}\}$, and $\mathit{CostFootprint}(Q) = \{\mathtt{t}, \mathsf{z}\}$, so $(W(L) \cup \mathit{Frame}(\{P, Q\})) \cap \mathit{CostFootprint}(P) = \emptyset$, and $(W(L) \cup \mathit{Frame}(\{P, Q\})) \cap \mathit{CostFootprint}(Q) = \emptyset$. The program on the right is checked analogously.

Given a program $\mathcal{P}$ with variables $\overline{x}$ and ACRS with initial equation $C_{ini}(\overline{x})$. We denote by $\mathit{eval}(C_{ini}(\overline{x}), \sigma_0)$ the evaluation of the ACRS for a given initial assignment σ_0 of the variables. This is a standard evaluation of recurrence equations performed by instantiating the right-hand side of the equations with the values of the variables in σ_0 and checking the satisfiability of the size constraints (if the expression being checked or accumulated contains "_", the evaluation returns "unbound"). As usual, the process is repeated until an equation without calls is reached.

Example 5. Consider the ACRS of the left program in Fig. 1 with variables $(\mathtt{t}, \mathsf{x}, \mathsf{w}, \mathsf{y}, \mathsf{z})$, initial state $\sigma_0 = (2, 0, 0, 0, 0)$, and cost model $\mathcal{M}_{\text{inst}}$ (thus c_{before}, $c_{B_{w_0}}$ and c_{w_0} take values 1, 1 and 2 respectively). The evaluation of the ACRS results in $\mathit{eval}(C_{ini}(\mathtt{t}, \mathsf{x}, \mathsf{w}, \mathsf{y}, \mathsf{z}), (2, 0, 0, 0, 0)) = 6 + 2 \cdot \mathtt{ac}_{\mathsf{P}}(2, 0) + 2 \cdot \mathtt{ac}_{\mathsf{Q}}(2, 0)$.

The following theorem states soundness of the ACRS obtained by applying Def. 3 provided that all loops satisfy Def. 4.

Theorem 1 (Soundness of ACRS). *Let $\mathcal{M}$ be a cost model and $\mathcal{P}$ an abstract program whose loops satisfy Def. 4. Let $c_{\mathcal{P}}$ be the abstract cost of $\mathcal{P}$ defined as in Definition 2. Let C_{ini} be the initial equation for the ACRS obtained by Def. 3. For any initial state of the variables $\sigma_0 \in \mathbb{Z}^{n_m}$, it holds that $c_{\mathcal{P}}(\sigma_0) \leq eval(C_{ini}(\overline{x}), \sigma_0)$.*

4.2 From ACRS to Abstract Cost Invariants

Example 5 shows that ACRSs are evaluable for concrete instances. However, to enable automated QAE, we need to obtain from them *closed-form* cost invariants and postconditions, i.e., non-recursive expressions. We introduce the novel concept of *abstract cost invariant* (ACI) that enables automated, inductive proofs over cost in a deductive verification system. The crucial difference to (non-inductive) cost postconditions as inferred by existing cost analyzers is that ACIs can be proven inductively for each loop iteration. Hence, they integrate naturally into deductive verification systems that use loop invariants [21].

In contrast to ACIs, postconditions provide a bound for the cost *after* execution of the *whole* loop they refer to. Typically, a postcondition bound for a loop has the form $max\_iter * max\_cost + max\_base$, where max_iter is the maximal number of iterations of the loop, max_cost is the maximal cost of any loop iteration, and max_base is the maximal cost of executing the loop with no iterations. Instead, an ACI has the form $growth * max\_cost + max\_base$, where $growth$ counts how many times the loop has been executed and hence provides a bound after *each* loop iteration. The challenge is to design an automated technique that infers *growth*. We propose to obtain it from the ranking function:

Definition 5 (Growth). *Given a loop with ranking function $F = c + \sum_i a_i \cdot v_i$, where c and v_i are the constant and variable parts of the function, respectively, and a_i are constant coefficients. If we denote with v_i^0 the initial value of variable v_i before entering the loop, then $growth = \sum_i a_i \cdot \left(v_i^0 - v_i\right)$.*

Example 6. We look at four simple loops with ranking function *decreases* and the *growth* inferred automatically by applying Def. 5:

int i = 0;	**int** i = t;	**int** i = 0;	**int** i = t;
while (i < t)	**while** (i > 0)	**while** (i < t)	**while** (i > 0)
i++;	i−−;	i += 2;	i −= 2;
decreases t − i	*decreases* i	*decreases* $\frac{t-i+1}{2}$	*decreases* $\frac{i+1}{2}$
growth i	*growth* t − i	*growth* $\frac{i}{2}$	*growth* $\frac{t-i}{2}$

We can now define the concept of ACI that relies on abstract cost relations defined in Sect. 4.1 and growth as defined above.

Definition 6 (Abstract Cost Invariant). *Given an ACRS as in Def. 3 and its growth as in Def. 5, an* abstract cost invariant *is defined as follows:* $\texttt{cinv}(\overline{x}) = \texttt{C}_\texttt{B}{}^{\texttt{max}} + growth \cdot \left(\sum_{j=1}^{n} \texttt{ac}_\texttt{j}\left(c_{j,1}, \ldots, c_{j,h_{cj}}\right) + \sum_{i=1}^{m} \texttt{C}_{\texttt{N}_\texttt{i}}{}^{\texttt{max}}\right)$ *where* $\texttt{C}_\texttt{B}{}^{\texttt{max}}$ *stands for the maximal value that the expression* $\texttt{C}_\texttt{B}$ *can take under the constraints* φ_B*, and* $\texttt{C}_{\texttt{N}_\texttt{i}}{}^{\texttt{max}}$ *the maximal value of* $\texttt{C}_{\texttt{N}_\texttt{i}}$ *under* φ_I*. We generate the annotation "*//@ **cost_invariant** cinv$(\overline{x})$;*".*

To obtain the maximal cost of a cost expression under a set of constraints, we use existing maximization procedures [5].

From Def. 6 we obtain ACIs as closed-form abstract cost expressions of the form $\texttt{abexpr} = \texttt{cexpr} \mid \texttt{ac} \mid \texttt{abexpr}_1 + \texttt{abexpr}_2 \mid \texttt{abexpr}_1 * \texttt{abexpr}_2$ where `ac` represents an abstract cost function as defined in Sect. 3.1 and `cexpr` is a concrete cost expression. The definition above yields linear bounds, however, the extension to infer postconditions in the subsequent section leads to polynomial expressions (of arbitrary degree).[4]

Example 7 (Abstract Cost Invariant). Consider the first loop in Example 6 (where *growth* = i) with the following frame and footprint:

```
//@ assignable j;  accessible i,t,j,k;  cost_footprint k;
```

Using $\mathcal{M}_{\mathsf{instr}}$, the evaluation of the loop guard and the increase of i both have unit cost, so the ACRS is:

$$\begin{array}{ll} C(\mathsf{i},\mathsf{t},\mathsf{j},\mathsf{k}) = 1 & \{\mathsf{i} \geq \mathsf{t}\} \\ C(\mathsf{i},\mathsf{t},\mathsf{j},\mathsf{k}) = \mathsf{ac_P}(\mathsf{k}) + 2 + C(\mathsf{i}',\mathsf{t},\_,\mathsf{k}) & \{\mathsf{i}' = \mathsf{i}+1,\ \mathsf{i} < \mathsf{t}\} \end{array}$$

The value of the assignable variable j in the recursive call is "forgotten" (item (6) in Def. 3), but this information loss does not affect solvability of the ACRS. We obtain the following ACI: "`//@ cost_invariant 1 + i * (2 + acP(k));`".

Example 8 (Upper Bound Abstract Cost Invariant). Sometimes an ACI is over-approximating cost, resulting in an *upper bound ACI.* To illustrate this, we add an instruction that creates an array of non-constant size "i" to the program in Example 7 and measure memory consumption instead of instruction count.

```
while (i < t) {
    a = new int[i];
    //@ assignable j;
    //@ accessible i,t,j,a,k;
    //@ cost_footprint k;
    \abstract_statement P;
    i++;
}
```

The resulting ACRS thus accumulates cost "i" at each iteration, plus the memory consumed by the abstract statement:

$$\begin{array}{ll} C(\mathsf{i},\mathsf{t},\mathsf{j},\mathsf{k}) = 0, & \{\mathsf{i} \geq \mathsf{t}\} \\ C(\mathsf{i},\mathsf{t},\mathsf{j},\mathsf{k}) = \mathsf{ac_P}(\mathsf{k}) + \mathsf{i} + C(\mathsf{i}',\mathsf{t},\_,\mathsf{k}), & \{\mathsf{i}' = \mathsf{i}+1, \mathsf{i} < \mathsf{t}\} \end{array}$$

Now, maximizing the expression $\texttt{C}_{\texttt{N}_1} = \mathsf{i}$ under $\{\mathsf{i}' = \mathsf{i}+1, \mathsf{i} < \mathsf{t}\}$ results in $\texttt{C}_{\texttt{N}_1}{}^{\texttt{max}} = \mathsf{t}-1$ and upper bound ACI "`//@ cost_invariant i * (t - 1 + acP(k));`".

Let c_L denote the abstract cost of executing a loop L (in analogy to $c_{\mathcal{P}}$ in Def. 2, but considering only loop L rather than the whole program $\mathcal{P}$). We denote by c_I the portion of the cost in c_L up to the execution of iteration I.

Proposition 1. *Let L be a loop with variables $\overline{x}$ satisfying Def. 4, $\texttt{cinv}(\overline{x})$ its ACI, and $\sigma_I \in \mathbb{Z}^{n_m}$ be the store after performing iteration I of L. Then the following holds: (1) $\texttt{cinv}(\overline{x})$ is true on entering the loop; (2) $c_I(\sigma_I) \leq \texttt{cinv}(\sigma_I)$.*

[4] As our approach is based on a recurrences-based framework [39] that works for exponential and logarithmic expressions, the results in this section generalize to these expressions. However, the AE deductive verification system is not able to deal with them automatically at the moment, so we skip these expressions in our account.

4.3 From Cost Invariants to Postconditions

To handle programs with nested loops and to prove relational properties it is necessary to infer *cost postconditions* for abstract programs. For nested loops the cost postcondition states the abstract cost after complete execution of the inner loop and it is used to compute the invariant of the outer loop. For relational properties, the cost postconditions of two abstract programs are compared. Cost postconditions for concrete programs are obtained by upper bound solvers (e.g., COSTA [3], CoFloCo [16], AProVE [17]) that compute *max_iter*, an upper bound on the number of iterations that a loop performs. To do so, one relies on ranking functions. We do this as well, but generalize the computation of postconditions to abstract programs. The cost postcondition is obtained by substituting `growth` by `max_iter` in the formula of $\mathtt{cinv}(\overline{x})$ in Def. 6 as follows.

Definition 7 (Cost Postcondition). *Let L be a loop, max_iter be an upper bound on the number of iterations of L. Given the ACRS for L in Def. 3, we infer the cost postcondition for L as*

$$post(\overline{x}) = \mathtt{C_B}^{\mathtt{max}} + max\_iter(\overline{x}) \cdot \left(\sum_{j=1}^{n} \mathtt{ac_j}\left(c_{j,1},\ldots,c_{j,h_{c_j}}\right) + \sum_{i=1}^{m} \mathtt{C_{N_i}}^{\mathtt{max}}\right)$$

*and generate the annotation "//@ **assert** cost $== post(\overline{x})$;".*

To infer the postcondition for a complete abstract program, we take the sum of all *cost postconditions* of its top-level loops plus the cost of the non-iterative fragments. Fig. 1 shows the cost postconditions for our running example obtained by replacing the growth `i` of the invariant with the bound `t` on the loop iterations and requiring $\mathtt{t} \geq 0$. The generation of inductive ACIs for nested loops uses the cost postcondition of inner loops to compute the invariants of the outer ones. The following theorem states soundness of cost postconditions:

Theorem 2. *Let L be a loop over variables $\overline{x}$ satisfying Def. 4 and $post(\overline{x})$ its cost postcondition. Let $\sigma_L \in \mathbb{Z}^{m_n}$ be the store upon termination of L. Then $c_L(\sigma_L) \leq post(\sigma_L)$.*

5 Experimental Evaluation

We implemented a prototype of our approach downloadable from https://tinyurl.com/qae-impl (including required libraries). The archive contains the benchmarks of this section and additional examples as well as build and usage instructions. The prototype is a command-line implementation backed by an existing cost analysis library for (non-abstract) Java bytecode as well as the deductive verification system KeY [2] including the AE framework [37,38]. Our implementation consists of three components: (1) An extension of a cost analyzer (written in Python) to handle abstract Java programs, (2) a conversion tool (written in Java) translating the output of the analyzer to a set of input files for KeY, (3) a bash script orchestrating the whole tool chain, specifically, the interplay between item (1), item (2) and the two libraries. In case of a failed certification attempt, our script offers the choice to open the generated proof in KeY for further debugging. In total, our implementation (excluding the libraries) consists of 1,802 lines of PYTHON, 703 lines of JAVA, and 389 lines of bash code (without blank lines and comments).

To assess effectiveness and efficiency of our approach, we used our QAE implementation to analyze seven typical code optimization rules using cost models $\mathcal{M}_{\mathsf{instr}}$ (rows "1∗"–"6∗" in Table 1) and $\mathcal{M}_{\mathsf{heap}}$ (rows "7∗"). While $\mathcal{M}_{\mathsf{instr}}$ counts the number of instructions, $\mathcal{M}_{\mathsf{heap}}$ measures heap consumption. The first column identifies the benchmark ("a" refers to the original program, "b" to the transformed one), the second **P** refers to the kind of proven cost result (asymptotic "a", exact "e", upper "u"), column three shows the inferred growth function for each loop in the program (separated by "," if there are two or more loops), in the fourth column we list the cost postcondition obtained by the analysis (expressions indicating the number of loop iterations are highlighted), and columns five to eight display performance metrics. Time $t_{\texttt{cost}}$, given in milliseconds, is the time needed to perform the cost analysis. The proof generation time $t_{\texttt{proof}}$ is given in seconds. We also display the time $t_{\texttt{check}}$ needed for checking integrity of an already generated proof certificate. Finally, $s_{\texttt{proof}}$ is the size of the generated KeY proof in terms of number of proof steps. Even though the time needed for certification is significantly higher than for cost analysis (which is to be expected), each analysis can be performed within one minute. The time to *check* a proof certificate amounts to approximately one fourth to one third of the time needed to *generate* it. We stress that all analyses are *fully automatic*.

We briefly describe the nature of each experiment: **1** is a *loop unrolling* transformation duplicating the body of a loop: each copy of the body is put inside an **if** -statement conditioned by the loop guard. Here, we had to switch to *asymptotic* cost invariants: The cost analyzer over-approximates the number of iterations of the unrolled loop, since there are different possible control flows in the body. This was automatically detected by the certifier which failed to find a proof when exact cost invariants are conjectured and succeeds with asymptotic ones. **2** is the *CodeMotion* example from Sect. 2. The result reflects the cost *decrease* in the sense that less instructions need to be executed by the transformed program. **3** implements a *LoopTiling* optimization at compiler level in which a single loop with $n \cdot m$ iterations is transformed into two nested loops, an outer one looping until n and an inner one until m. Since our cost analyzer only handles linear size expressions, the first program is written using an auxiliary parameter t that is then instantiated to value $n \cdot m$. **4** is a *SplitLoop* transformation splitting a loop with two independent parts into two separate loops. We prove that this transformation does not affect the cost up to a constant factor. **5** is an optimization combining *two loops* with the same body structure into one loop. **6** is a *three loops* example, one nested and one simple. The optimization combines the bodies of the outer loop in the nested structure and the simple loop. **7** is an *array* optimization, where an array declaration is moved in front of a loop, initializing it with an auxiliary parameter that is the sum of all the initial sizes.

	P	**Cost analysis results**		t_{cost}	t_{proof}	t_{check}	s_{proof}
		Growth	**Postcondition**	[ms]	[s]	[s]	#nodes
1a	a	i	$t{\cdot}\mathsf{ac_P}(x)$	45.0	12.9	4.3	1,784
1b	a	i	$t{\cdot}\mathsf{ac_P}(x)$	53.4	23.8	5.0	3,472
2a	e	i	$2{+}t{\cdot}(7 + \mathsf{ac_P}(t,w) + \mathsf{ac_Q}(t,z))$	50.0	23.3	5.7	3,692
2b	e	i	$3 + \mathsf{ac_P}(t,w){+}t{\cdot}(6 + \mathsf{ac_Q}(t,z))$	42.0	19.7	5.7	3,243
3a	e	i	$2{+}t{\cdot}(6 + \mathsf{ac_P}(k))$	49.1	18.7	5.1	2,821
3b	e	$i\ ,\ j$	$6{+}n \cdot m{\cdot}(6 + \mathsf{ac_P}(k))$	49.5	23.3	5.7	3,794
4a	e	$i+1$	$2{+}(l+1){\cdot}(7 + \mathsf{ac_{Q1}}(t,w) + \mathsf{ac_{Q2}}(t,z))$	49.5	23.8	5.7	3,933
4b	e	$i+1\ ,\ i+1$	$2{+}(l+1){\cdot}(12 + \mathsf{ac_{Q1}}(t,w) + \mathsf{ac_{Q2}}(t,z))$	48.5	29.4	7.3	5,137
5a	e	$i\ ,\ j$	$2{+}n{\cdot}(6 + \mathsf{ac_P}(y)){+}\mathrm{m}{\cdot}(6 + \mathsf{ac_P}(y))$	55.1	25.3	7.1	4,795
5b	e	i	$2{+}(n+m){\cdot}(8 + \mathsf{ac_P}(y))$	48.2	14.1	4.7	2,492
6a	e	$k\ ,\ j\ ,\ n-i$	$6{+}n{\cdot}(m{\cdot}(6 + \mathsf{ac_P}(y)){+}n{\cdot}(5 + \mathsf{ac_Q}(y))$	49.8	32.0	8.1	7,078
6b	e	$k\ ,\ j$	$7{+}n{\cdot}(m{\cdot}(6 + \mathsf{ac_P}(y)) + \mathsf{ac_Q}(y))$	49.6	24.9	6.4	4,995
7a	u	$i-1$	$(t-1){\cdot}(4 \cdot (t-1) + \mathsf{ac_P}(y))$	51.2	15.6	5.3	2,578
7b	u	$i-1$	$4 \cdot m{+}(t-1){\cdot}\mathsf{ac_P}(y)$	43.3	13.0	4.2	1,793

Table 1: Results of the experiments.

6 Related Work

The present paper builds on the original AE framework [37,38], which we extend to *Quantitative* AE. At the moment no other approach or tool is able to analyze and certify the cost of schematic programs, specifically relational properties, so a direct comparison is impossible.

Cost Analysis. There are many resource analysis tools, including: [20], based on introducing counters and inferring loop invariants; [23], based on an analysis over the depth of functional programs formalized by means of type systems. Approaches that bound the number of execution steps include [19,29], working at the level of compilers. Systems such as APROVE [17] analyze the complexity of JAVA programs by transforming them to integer transition systems; COSTA [3] and COFLOCO [16] are based on the generation of cost recurrence equations from which upper bounds can be inferred. That is also the basis of the approach we pursue to infer abstract upper bounds in Sect. 4.1, hence our technique can be viewed as a generalization of these systems. Approaches based on type systems could also be generalized to work on abstract programs by introducing abstract cost as in Sect. 4.1.

For our work it is crucial to use ranking functions to infer growth of cost invariants. Ranking functions were used to generate bounds on the number of loop iterations in several systems, but none used them to define growth: [10] obtain runtime complexity bounds via symbolic representation from ranking functions, likewise PUBS [3], LOOPUS [40], and ABC [8]. PUBS analyses all loop transitions at once, LOOPUS uses an iterative procedure where bounds are propagated from inner to outer loops, ABC deals with nested, but not sequential loops. In our work, when inferring upper bounds, we solve all transitions at once and handle nested as well as sequential loops.

Certification. Several general-purpose deductive software verification [21] tools exist, including VeryFast [34], Why [15], Dafny [28], KIV [33], and KeY [2]. We use KeY, the currently only system to implement AE. *Interactive* proof assistants like Isabelle [31] or Coq [7] also support more or less expressive abstract program fragments, but lack full automation. There are dedicated approaches involving schematic programs for *specific* contexts, like regression verification [18], compilation [22, 26, 30] or derived symbolic execution rules [12].

Regarding the combination of deductive verification and cost analysis, the closest approach to ours is the integration of COSTA and KeY [4] which was realized for concrete, not abstract programs. They verify upper bounds on the cost of concrete programs by decomposing them into ranking functions and size relations which are then verified separately. Here we use the novel concept of cost invariant that allows verification of quantitative properties without decomposition. Paper [4] deals only with the global number of iterations as is common in worst-case cost analysis. Our cost invariants are designed to be inductive and propagate cost through all loop iterations. Radiček et al. [32] devise a formal framework for analyzing the relative cost of different programs (or the same program with different inputs). Compared to our approach, they target purely functional programs extended with monads representing cost, while we work with an industrial programming language. Moreover, we generally reason about the cost of *transformations*, not of a transformation applied to one *particular* program.

7 Conclusion and Future Work

We presented the first approach to analyze the cost of schematic programs with placeholders. We can infer and verify cost bounds for a potentially infinite class of programs once and for all. In particular, for the first time, it is possible to analyze and prove changes in efficiency caused by program transformations—for all input programs. Our approach supports exact and asymptotic cost and a configurable cost model. We implemented a tool chain based on a cost analyzer and a program verifier which analyzes and formally certifies abstract cost bounds in a fully automated manner. Certification is essential, because only the verifier can determine whether the bounds inferred by the cost analyzer are exact.

Our work required the new concept of an (abstract) cost invariant. This is interesting in itself, because (i) it renders the analysis of nested loops modular and (ii) provides an interface to backends (such as verifiers) that characterizes the cost of code in iterations.

Obvious future work involves extending the analyzed target language. Cost analysis and deductive verification (including AE) are already possible for a large Java fragment [3, 37]. More interesting—and more challenging—is the analysis of program transformations that parallelize code. The extension to larger classes of cost functions, such as logarithmic or exponential, could be realized by integrating non-linear SMT solvers into the tool chain.

Acknowledgments. 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 EU and by the UCM CT42/18-CT43/18 grant.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *LNCS*. Springer, 2016.
3. Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.
4. Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, Germán Puebla, and Guillermo Román-Díez. A formal verification framework for static analysis - as well as its instantiation to the resource analyzer COSTA and formal verification tool KeY. *Software and Systems Modeling*, 15(4):987–1012, 2016.
5. Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.
6. Roberto Bagnara, Fred Mesnard, Andrea Pescetti, and Enea Zaffanella. A new look at the automatic synthesis of linear ranking functions. *Inf. Comput.*, 215:47–67, 2012.
7. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
8. Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. ABC: algebraic bound computation for loops. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *LNCS*, pages 103–118. Springer, 2010.
9. Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, June 1975.
10. Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th Intl. Conf., TACAS, Grenoble, France*, volume 8413 of *LNCS*, pages 140–155. Springer, 2014.
11. Marc Brockschmidt, Richard Musiol, Carsten Otto, and Jürgen Giesl. Automated termination proofs for Java programs with cyclic data. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *LNCS*, pages 105–122. Springer, 2012.
12. Richard Bubel, Andreas Roth, and Philipp Rümmer. Ensuring the Correctness of Lightweight Tactics for JavaCard Dynamic Logic. *Electr. Notes Theor. Comput. Sci.*, 199:107–128, 2008.
13. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978.

14. Karl Crary and Stephanie Weirich. Resource bound certification. In Mark N. Wegman and Thomas W. Reps, editors, *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, pages 184–198. ACM, 2000.
15. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th Intl. Conf., CAV, Berlin, Germany*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.
16. Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In Jacques Garrigue, editor, *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, volume 8858 of *LNCS*, pages 275–295. Springer, 2014.
17. Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th Intl. Joint Conf., IJCAR, Vienna, Austria*, volume 8562 of *LNCS*, pages 184–191. Springer, 2014.
18. Benny Godlin and Ofer Strichman. Regression Verification: Proving the Equivalence of Similar Programs. *Softw. Test., Verif. Reliab.*, 23(3):241–258, 2013.
19. Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, and Kerstin Eder. Static energy consumption analysis of LLVM IR programs. *CoRR*, abs/1405.4565, 2014.
20. Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 127–139. ACM, 2009.
21. Reiner Hähnle and Marieke Huisman. Deductive verification: from pen-and-paper proofs to industrial tools. In Bernhard Steffen and Gerhard Woeginger, editors, *Computing and Software Science: State of the Art and Perspectives*, volume 10000 of *LNCS*, pages 345–373. Springer, 2019.
22. Reiner Hähnle and Dominic Steinhöfel. Modular, correct compilation with automatic soundness proofs. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques, 8th Intl. Symp., Proc. Part I, ISoLA, Cyprus*, volume 11244 of *LNCS*, pages 424–447. Springer, 2018.
23. Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP, Paphos, Cyprus*, volume 6012 of *LNCS*, pages 287–306. Springer, 2010.
24. John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 410–423, New York, NY, USA, 1996. Association for Computing Machinery.
25. James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.
26. Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving Optimizations Correct Using Parameterized Program Equivalence. In *Proc. PLDI 2009*, pages 327–337, 2009.

27. Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. JML Reference Manual, May 2013. Draft revision 2344.
28. Rustan Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference, LPAR-16, Dakar, Senegal*, pages 348–370. Springer Berlin Heidelberg, April 2010.
29. Umer Liqat, Kyriakos Georgiou, Steve Kerrison, Pedro López-García, John P. Gallagher, Manuel V. Hermenegildo, and Kerstin Eder. Inferring parametric energy consumption functions at different software levels: ISA vs. LLVM IR. In Marko C. J. D. van Eekelen and Ugo Dal Lago, editors, *Foundational and Practical Aspects of Resource Analysis - 4th Intl. Workshop, FOPARA, London, UK, Revised Selected Papers*, volume 9964 of *LNCS*, pages 81–100, 2015.
30. Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Practical Verification of Peephole Optimizations with Alive. *Commun. ACM*, 61(2):84–91, 2018.
31. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
32. Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. Monadic refinements for relational cost analysis. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.
33. Wolfgang Reif. The KIV-approach to software verification. In *KORSO - Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *LNCS*, pages 339–370. Springer, 1995.
34. Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for Java-like programs based on dynamic frames. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th Intl. Conf., FASE, Budapest, Hungary*, volume 4961 of *LNCS*, pages 261–275. Springer, 2008.
35. Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for Java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.*, 32(3):8:1–8:70, 2010.
36. Dominic Steinhöfel. REFINITY to Model and Prove Program Transformation Rules. In Bruno C. d. S. Oliveira, editor, *Proc. 18th Asian Symposium on Programming Languages and Systems (APLAS)*, LNCS. Springer, 2020.
37. Dominic Steinhöfel and Reiner Hähnle. Abstract execution. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *LNCS*, pages 319–336. Springer, 2019.
38. Dominic Steinhöfel. *Abstract Execution: Automatically Proving Infinitely Many Programs*. PhD thesis, Technical University of Darmstadt, Department of Computer Science, Darmstadt, Germany, 2020.
39. Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.
40. Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction (extended version). *CoRR*, abs/1203.5303, 2012.

Lower-Bound Synthesis Using Loop Specialization and Max-SMT

Elvira Albert[1,2], Samir Genaim[1,2], Enrique Martin-Martin[1], Alicia Merayo[1]( ), 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.

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 *w*orst-case and *b*est-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 under-estimate 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.

```
while (x >= 0 && y > 0) {
  if (*) {
    x = x - 1;
  } else {
    y = y - 1;
  }
}
```

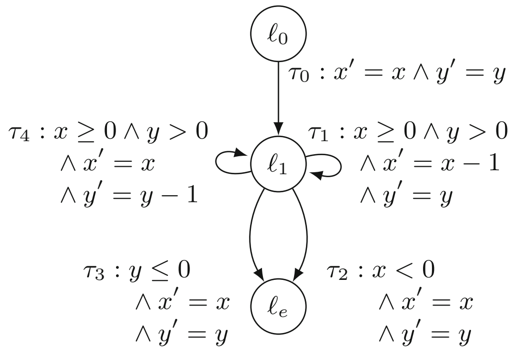

ℓ_0
$\tau_0 : x' = x \wedge y' = y$
$\tau_4 : x \geq 0 \wedge y > 0 \wedge x' = x \wedge y' = y - 1$
ℓ_1
$\tau_1 : x \geq 0 \wedge y > 0 \wedge x' = x - 1 \wedge y' = y$
$\tau_3 : y \leq 0 \wedge x' = x \wedge y' = y$
ℓ_e
$\tau_2 : x < 0 \wedge x' = x \wedge y' = y$

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:=nondet();”, where x is a program variable and 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:=nondet(); if (u≥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

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

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 ($\Theta = true$):

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

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 $\wedge_{i=1}^{n} x_i = \sigma(x_i)$, 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 $(\ell_0, \sigma_0), (\ell_1, \sigma_1), \ldots$ 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

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

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 $len(t)$ 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 $Traces_{\mathcal{S}}(C)$, 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 $wc_{\mathcal{S}} : \mathcal{C} \to \mathbb{R}_{\geq 0}$ is $wc_{\mathcal{S}}(C) = sup\ \{len(t) \mid t \in Traces_{\mathcal{S}}(C)\}$ and its best-case cost function $bc_{\mathcal{S}} : \mathcal{C} \to \mathbb{R}_{\geq 0}$ is $bc_{\mathcal{S}}(C) = inf\ \{len(t) \mid t \in Traces_{\mathcal{S}}(C)\}$.*

Clearly, $wc_{\mathcal{S}}$ and $bc_{\mathcal{S}}$ are not computable. Our goal in this paper is to automatically find a lower-bound function $\rho : \mathbb{Z}^n \to \mathbb{R}_{\geq 0}$ 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 $\rho : \mathbb{Z}^n \to \mathbb{R}_{\geq 0}$ 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 $\|\rho(\bar{x})\| = \lceil \max(0, \rho(x)) \rceil$ to map all negative valuations of ρ to zero.

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

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

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

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):

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

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

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

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

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

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

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

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):

$$\begin{aligned}
&\tau_1 = (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x - 1 \wedge y' = y)\\
&\tau_4 = (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x \wedge y' = y - 1)\\
&\tau_2 = (\ell_1, \ell_2, x < 0 \wedge x' = x \wedge y' = y)\\
&\tau_3 = (\ell_1, \ell_2, y \leq 0 \wedge x' = x \wedge y' = y)
\end{aligned}$$

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

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

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

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

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

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 $x \geq 5$:

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

With this specialized guard and considering $\mathcal{Q}_{\ell_1} \equiv \mathit{true}$, 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

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

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

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

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:

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

$$\forall \bar{x}.\ \mathcal{Q}_{\ell_0} \rightarrow \Theta \tag{16}$$

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 well[1].

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

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

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

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

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

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 $\texttt{cone}(\mathcal{P})$ 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

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

where $\texttt{cone}(\mathcal{Q}_\ell)$ 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 *LO*wer-*B*ound synthesiz*ER*, 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 $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ 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. $\mathcal{Q} \wedge \overline{-n \leq x_i \leq n}$, 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 $f_1(n) = k_1 n + d_1$ and $f_2(n) = k_2 n + d_2$, 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>=0 and y>=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 $y >= x$ in the if-condition (and hence $x > y$ in the else), we can prove that $x >= 0 \wedge y >= 0 \wedge x + y = 1$ 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)

Certified Cost Bounds for Abstract Programs

ELVIRA ALBERT, Instituto de Tecnología del Conocimiento and Complutense University of Madrid, Spain
REINER HÄHNLE, Technical University Darmstadt, Germany
ALICIA MERAYO, Complutense University of Madrid, Spain
DOMINIC STEINHÖFEL, CISPA Helmholtz Center for Information Security, Germany

A program containing placeholders for unspecified statements or expressions is called an abstract (or schematic) program. Placeholder symbols occur naturally in program transformation rules, as used in refactoring, compilation, optimization, or parallelization. Static cost analysis derives the *precise* cost –or *upper* and *lower* bounds for it– of executing programs, as functions in terms of the program's input data size. We present a generalization of automated cost analysis that can handle abstract programs and, hence, can analyze the impact on the *cost effect of program transformations*. This kind of relational property requires provably precise cost bounds which are not always produced by cost analysis. Therefore, we certify by deductive verification that the inferred abstract cost bounds are correct and sufficiently precise. It is the first approach solving this problem. Both, abstract cost analysis and certification, are based on quantitative abstract execution (QAE) which in turn is a variation of abstract execution, a recently developed symbolic execution technique for abstract programs. To realize QAE the new concept of a cost invariant is introduced. QAE is implemented and runs fully automatically on a benchmark set consisting of representative optimization rules.

ACM Reference Format:
Elvira Albert, Reiner Hähnle, Alicia Merayo, and Dominic Steinhöfel. 2021. Certified Cost Bounds for Abstract Programs. 1, 1 (March 2021), 25 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

We present a generalization of automated cost analysis that can handle programs containing placeholders for unspecified statements. Consider the program $Q \equiv$ "i =0; **while** (i < t) {P; i ++;}", where P is any statement not modifying i or t. We call P an *abstract statement*; a program like Q containing abstract statements is called *abstract program*. The (exact or upper/lower bound) cost of executing P is described by a function $\mathsf{ac}_{\mathsf{P}}(\overline{x})$ depending on the variables $\overline{x}$ occurring in P. We call this function the *abstract cost* of P. Assuming that executing any statement has unit cost and that $t \geq 0$, one can compute the (abstract) cost of Q as $2 + t \cdot (\mathsf{ac}_{\mathsf{P}}(\overline{x}) + 2)$ depending on ac_{P} and t. For any concrete instance of P, we can derive its concrete cost as usual and then obtain the concrete cost of Q simply by instantiating ac_{P}. In this article, we define and implement an abstract cost analysis to infer abstract cost bounds. Our implementation consists of an automatic abstract cost analysis tool and an automatic certifier for the correctness of inferred abstract bounds. Both steps are performed with an approach called *Quantitative Abstract Execution* (QAE).

Authors' addresses: Elvira Albert, elvira@fdi.ucm.es, Instituto de Tecnología del Conocimiento and Complutense University of Madrid, Madrid, Spain; Reiner Hähnle, reiner.haehnle@tu-darmstadt.de, Technical University Darmstadt, Darmstadt, Germany; Alicia Merayo, amerayo@ucm.es, Complutense University of Madrid, Madrid, Spain; Dominic Steinhöfel, dominic.steinhoefel@cispa.de, CISPA Helmholtz Center for Information Security, Saarbrücken, Germany.

Fine, but what is this good for? Abstract programs occur in program transformation rules used in compilation, optimization, parallelization, refactoring, etc. [29, 31, 45]: Transformations are specified as rules over *program schemata* which are nothing but abstract programs. If we can perform cost analysis of abstract programs, we can *analyze the cost effect of program transformations*. Our approach is the *first method to analyze the cost impact of program transformations*.

Automated Cost Analysis. Cost analysis occupies an interesting middle ground between termination checking and full functional verification in the static program analysis portfolio. The main problem in functional verification is that one has to come up with a functional specification of the intended behavior, as well as with auxiliary specifications including loop invariants and contracts [30]. In contrast, termination is a generic property and it is sufficient to come up with a suitable term order or ranking function [10]. For many programs, termination analysis is vastly easier to automate than verification.[1]

Computation cost is not a generic property, but it is usually schematic: One fixes a class of cost functions (for example, polynomial) that can be handled. A cost analysis then must come up with parameters (degree, coefficients) that constitute a valid bound (lower, upper, exact) for all inputs of a given program with respect to a cost model (# of instructions, allocated memory, etc.). If this is performed bottom up with respect to a program's call graph, it is possible to *infer* a cost bound for the top-level function of a program. Such a cost expression is often *symbolic*, because it depends on the program's input parameters.

A central technique for inferring symbolic cost of a piece of code with high precision is *symbolic execution* (SE) [13, 34]. The main difficulty is to render SE of loops with symbolic bounds finite. This is achieved with *loop invariants* that generalize the behavior of a loop body: an invariant is valid at the loop head after arbitrarily many iterations. To infer sufficiently strong invariants automatically is generally an unsolved problem in functional verification, but much easier in the context of cost analysis, because invariants do not need to characterize functional behavior: it suffices that they permit to infer schematic cost expressions.

Upper and Lower Bounds. Cost analysis techniques traditionally focused on three types of bounds: (1) The vast majority concentrated on inferring *upper bounds on the worst-case* cost, because the assurance that no execution of a program will exceed the inferred amount of resources has important applications in safety-critical contexts. (2) *Lower bounds on the best-case* cost [7, 19, 21] characterize the minimal cost of any program execution and are useful in task parallelization, where a task is not parallelized unless its best-case cost is larger than the overhead from parallelization. Moreover, such bounds are used in performance debugging, verification and optimization. (3) *Lower bounds on the worst-case* cost that bound the worst-case cost from below. Their main usage is, when combined with upper bounds on the worst-case, to infer tighter worst-case cost bounds. Further applications include: the gain on accuracy on the analysis of smart contracts, detection of performance bugs such as non-termination and of safety-critical vulnerabilities such as denial-of-service, see [24] for details on applications.

Lower bounds were studied less often than upper bounds, yet there are recent papers on lower bounds on the worst-case cost [6, 24]. The present paper adopts the approach of [6]: By means of specializing the loops of a program, it is able to obtain lower bounds on the worst-case cost, using techniques developed for lower bounds on the best-case cost. In our implementation, we use the LOBER tool [6]. While that system was initially designed to compute lower bounds on the worst-case cost, it allows to disable loop specialization, so as to provide lower bounds on the best-case

[1]In theory, of course, proving termination is as difficult as functional verification. It is hard to imagine, for example, to find a termination argument for the Collatz function without a deep understanding of what it does. But automated termination checking works very well for many programs in practice.

cost instead. We use the system in that way to compute lower bounds on the best-case cost. Combined with upper bounds on the worst-case cost this bounds the cost of any execution from both, above and below.

Abstract Execution. To infer the cost of program transformation *schemata* requires the capability of analyzing abstract programs. *This is not possible with standard SE*, because abstract statements have no operational semantics. One way to reason about abstract programs is to perform structural induction over the syntactic definition of statements and expressions whenever an abstract symbol is encountered. Structural induction is routinely used in interactive theorem proving [11, 40] to verify, e.g., compilers. It is labor-intensive and not automatic. Instead, here we perform cost analysis of abstract programs via a generalization of SE called *abstract execution* (AE) [46, 47]. The idea of AE is, quite simply, to symbolically execute a program containing abstract placeholder symbols for expressions and statements, just as if it were a concrete program. It might seem counterintuitive that this is possible: after all, nothing is known about an abstract symbol. But this is not quite true: one can equip an abstract symbol with an *abstract* description of the behavior of its instances: a set of memory locations its behavior may depend on, commonly called *footprint* and a (possibly different) set of memory locations it can change, commonly called *frame* [30].

Cost Invariants. In automated cost analysis, one infers cost bounds frequently from loop invariants, ranking functions, metering functions, and size relations computed during SE [4, 15, 22, 49]. In particular, ranking functions are used to bound the worst-case execution cost from above (upper bound on worst-case cost) and metering functions [24] are used to bound the cost from below (lower bounds on worst and best-case cost). To deal with loops containing *abstract* programs, we need a more general concept: loop invariants that express a *valid abstract cost bound* at the beginning of any loop iteration (e.g., $2 + i * (\mathsf{ac}_{\mathsf{P}}(\overline{x}) + 2)$ for program Q above). We call this a *cost invariant*. This is an important technical innovation of our paper. Cost invariants increase modularity of a cost analysis, because each loop can be analyzed and certified in isolation.

Relational Cost Analysis. AE allows specifying and verifying *relational* program properties [46], because one can express rule schemata. This extends to quantified AE and makes it possible, for the first time, to infer and to prove (automatically!), for example, the impact of program transformations on performance.

Certification. Cost annotations inferred by abstract cost analysis, i.e., cost invariants and abstract cost bounds, are automatically *certified* in our approach by a deductive verification system. This constitutes an extension of earlier work on cost certification [5] to abstract cost and abstract programs. As before, the division of labor is as follows: the specification (i.e., the cost bound) and the loop (cost) invariants are inferred by the cost analyzer, while the verification system formally proves that the bounds are correct. This ensures that the overall process is fully automatic, because the verification system obtains complete specifications from the cost analyzer.

To argue for correctness of an abstract cost analysis is complex, because it must be valid for an infinite set of concrete programs. For this reason alone, it is useful to certify the abstract cost inferred for a given abstract program: during development of the abstract cost analysis reported here, several errors in abstract cost computation were detected—analysis of the failed verification attempt gave immediate feedback on the cause. We built a regression test suite of problems so that any change in the cost analyzer can be validated in the future.

Certification is crucial for the correctness of quantitative relational properties: The inferred cost invariants might not be precise enough to establish, e.g., that a program transformation does not increase cost for any possible program instance and run. This is only established at the certification stage, where relational properties are formally verified. *A relational setting requires provably precise cost bounds.* This feature is not offered by existing cost analysis methods.

Summary of Contributions

This article makes the following main contributions:

(1) We extend the abstract execution framework [46, 47] to *quantitative abstract execution* by adding cost specifications that extend the standard specification of an abstract statement with an annotated cost expression.
(2) We leverage a cost analysis framework [4] for concrete programs to the abstract setting by means of the key notions of *inductive cost invariant* and *cost postcondition* for abstract programs.
(3) We report on an implementation of our approach that is publicly available and assess its effectiveness and efficiency on representative code optimization patterns.

This work extends and improves on a conference paper published in the proceedings of FASE 2021 [8]. Relative to that paper the present article contains the following *further* contributions:

(4) The initial approach was focused on the generation and verification of *upper bounds* on the cost of abstract programs. We extended it to generate and certify also *lower bounds* on the abstract cost.
(5) We relaxed the conditions to guarantee soundness of the abstract cost analysis framework. In particular, the concept of *cost neutral* abstract statement in Section 4 is less restrictive than in [8]. This allows to handle a larger class of programs.
(6) We extended the implementation and evaluation to include the two novel features.

Organization of the Article

This article is organized as follows.

- Section 2 introduces our approach informally by means of an example, and sets up the terminology that is going to be used.
- Section 3 introduces quantitative abstract execution. We begin by introducing the idea of abstract execution and the semantics of abstract statements. Then, we add to this basic notion of AE the new concept of a cost specification that leads to the QAE framework based on the concept of abstract cost for an abstract statement.
- Section 4 is focused on the automatic inference of cost invariants. This is achieved by extending an existing cost analysis framework for concrete programs [4] to abstract ones. We first define the notion of an *abstract cost relation system* (ACRS) that extends the well-known concept of a cost relation system [7]. Then, ACRS are used to obtain abstract cost invariants and, as the final step in the analysis, cost postconditions. Soundness is proven at the end of each subsection.
- Section 5 contains the experimental evaluation. Representative program transformation examples are considered, for which we summarize the most important results.
- Section 6 describes the state-of-art in our field, comparing the novel aspects of our work to existing results.
- Section 7 concludes this article, highlighting our most important achievements.

2 QAE BY EXAMPLE

We introduce our approach and terminology informally by means of a motivating example: *Code Motion* [1] is a compiler optimization technique moving a statement not affected by a loop from the beginning of the loop body in front of the loop. This code transformation should preserve behavior provided that the loop is executed at least once, but it can be expected to improve computation effort, i.e. *quantitative* properties of the program, such as execution time and memory consumption: The moved code block is executed exactly once, independently of the number of loop

Program Before

```
int i = 0;
// @ loop_invariant i ≥ 0 && i ≤ t;
// @ cost_invariant_upper
      i · (ac^u_P(t, w) + ac^u_Q(t, z) + 2) ;
// @ cost_invariant_lower
      i · (ac^l_P(t, w) + ac^l_Q(t, z) + 2) ;
// @ decreases t − i;
while (i < t) {
    // @ assignable x;
    // @ accessible t, w;
    // @ cost_footprint t, w;
    \abstract_statement P;
    // @ assignable y;
    // @ accessible i , t , y, z;
    // @ cost_footprint t, z;
    \abstract_statement Q;
    i ++;
}
// @ assert   t · (ac^l_P(t, w) + ac^l_Q(t, z) + 2)
       <=\cost
// @ assert \cost  <=
       2 + t · (ac^u_P(t, w) + ac^u_Q(t, z) + 2) ;
```

Program After

```
int i = 0;
// @ assignable x;
// @ accessible t, w;
// @ cost_footprint t, w;
\abstract_statement P;
// @ loop_invariant i ≥ 0 && i ≤ t;
// @ cost_invariant_upper
      i · (ac^u_Q(t, z) + 2) ;
// @ cost_invariant_lower
      i · (ac^l_Q(t, z) + 2) ;
// @ decreases t − i;
while (i < t) {
    // @ assignable y;
    // @ accessible i , t , y, z;
    // @ cost_footprint t, z;
    \abstract_statement Q;
    i ++;
}
// @ assert 2 + ac^l_P(t, w) + t · (ac^l_Q(t, z) + 2)
       <=\cost
// @ assert \cost  <=
       2 + ac^u_P(t, w) + t · (ac^u_Q(t, z) + 2) ;
```

Preconditions and Postconditions

Inputs: t, w, x, y, z   Precondition: t > 0   Postcondition: **\cost_1** ≥ **\cost_2**

Fig. 1. Motivating example on relational quantitative properties.

iterations, in the transformed context, leading to less executed instructions (less energy consumed) and, in case it allocates memory, reduced memory usage. In the following we subsume any quantitative aspect of a program under the term *cost* expressed in a parametric *cost model*, with the understanding that it can be instantiated to specific cost measures, such as number of instructions, number of allocated bytes, energy consumed, etc.

To formalize code motion as a transformation rule, we describe in- and output of the transformation *schematically*. Figure 1 depicts such a schema in a simple language based on JAVA. An *abstract statement* (AS) with identifier *Id*, declared as "**\abstract_statement** *Id*;", represents an arbitrary concrete statement. It is obviously unsafe to move arbitrary, possibly non-invariant, code blocks out of a loop. For this reason, the AS P to be moved, has a *specification* restricting the permitted behavior of its instances. For compatibility with JAVA we base our specification language on the *Java Modeling Language* (JML) [36]. Specifications are attached to code via structured comments marked as JML by an "@" symbol. JML keyword "**assignable**" defines the memory locations that may occur in the frame of an AS; similarly, "**accessible**" restricts the footprint. Figure 1 contains further keywords explained below.

Input to QAE is the abstract program to analyze, including annotations (highlighted in light gray in Figure 1) that express restrictions on the permitted instances of ASs. In addition to the frame and footprint, the *cost footprint* of an AS, denoted with the keyword "**cost_footprint**", is a subset of its footprint listing locations the cost expressions in

AS instances may depend on. It can also involve conditions on the variables. In Figure 1, the cost footprint of AS Q excludes accessible variables i and y. Annotations highlighted in dark gray are *automatically inferred* by abstract cost analysis and are input for the certifier. As usual, loop invariants (keyword "**loop_invariant**") are needed to describe the behavior of loops with symbolic bounds. The loop invariant in Figure 1 permits to determine the final value t of loop counter i after loop termination. To prove termination, the loop *variant* (keyword "**decreases**") is inferred. This keyword keeps information of an upper bound over the number of iterations of the loop.

So far, this is standard automated cost analysis [4]. The ability to *infer automatically* the remaining annotations represents our main contribution: Each AS P has an associated *abstract cost* function parametric in the locations of its footprint, represented by an abstract cost symbol ac_P. As our framework works for both upper and lower bounds, we distinguish the upper and lower bounds of the abstract cost of an abstract statement P using the cost symbols $\mathsf{ac}^{l}_{\mathsf{P}}(t, w)$ and $\mathsf{ac}^{u}_{\mathsf{P}}(t, w)$, respectively. These symbols can be instantiated with any concrete function parametric in t, w being a valid cost bound for the instance of P. For example, for the instance "P ≡ x=t+1;" the constant function 1 is the correct *exact* cost, and t with t ≥ 1 is a correct *upper bound* cost. On the other hand, for the instance "P ≡ k=0; **while** (k<t) k++;" the exact cost is t, while correct (but not tight) upper and lower bound costs are t^2 and 1, respectively.

As pointed out in Section 1, we need *cost invariants* to capture the cost of each loop iteration. These are declared with the keywords "**cost_invariant_upper**" and "**cost_invariant_lower**". To generate them, it is necessary to infer the *cost growth* of abstract programs that bounds the number of loop iterations executed so far. In the program of Figure 1 the number of loop iterations is determined exactly by the increase of variable i until it reaches t, but we can handle more general programs, where the number of performed iterations is not deterministic. For example, consider the following program, where "*" randomly returns true or false, causing a different increase of i in each case:

```
while (i < t) {
  if ( * ) i++; else i+=2;
}
```

Now the most precise information obtainable on the number of loop iterations is that they are between $\frac{t}{2}$ and t. The lower bound inference is reflected in the growth and cost invariants. In Section 4 we describe automated inference of cost invariants including the generation of cost growth for all loops. Our technique is compositional and works in presence of nested loops.

The QAE framework can express and prove quantitative relational properties. The assertions in the last lines in Figure 1 contain the expression **\cost** referring to the total accumulated cost of a program: its quantitative *postcondition.* We support quantitative relational postconditions such as **\cost_1** ≥ **\cost_2**, where **\cost_1**, **\cost_2** refer to the total cost of the original (on the left) and transformed (on the right) program, respectively. To prove relational properties, it is necessary to infer matching upper and lower bounds for the number of loop iterations. Only then the comparison of the invariants allows to conclude that the programs from which they derive satisfy the stipulated relational property. Otherwise, over- and under-approximation introduced by the cost analysis can make the relation hold for the postconditions, but the relational property does not necessarily hold for the programs.

Figure 2 shows a schema of our toolchain for proving relational properties. Given two abstract programs (the original and the transformed one), we obtain the corresponding cost-specified programs by means of the cost analyzer. If both upper and lower bounds of the loop match, then the tool verifies that the relational property holds. Proving quantitative properties is explained further in Section 3.4.

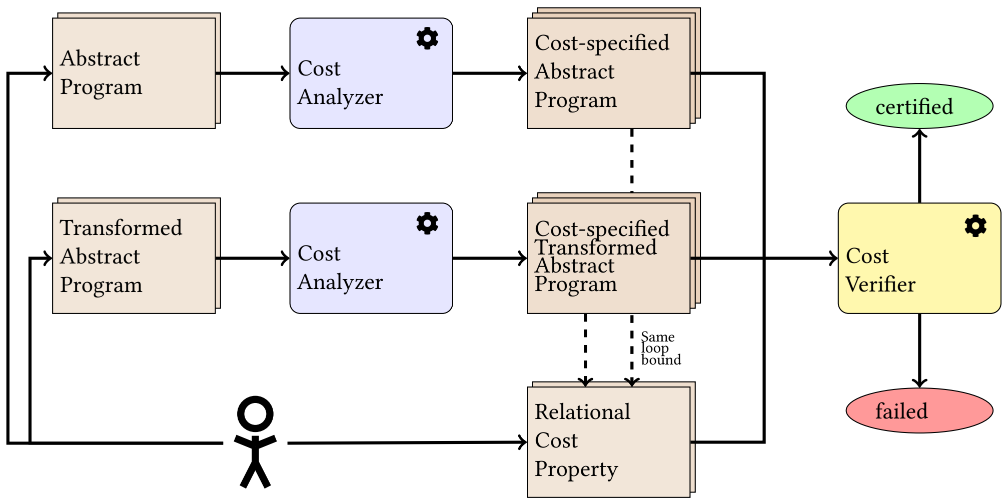


Fig. 2. Schema of tool chain for cost certification of abstract programs

3 FROM ABSTRACT EXECUTION TO QUANTITATIVE ABSTRACT EXECUTION

To obtain a formal account of QAE with correctness guarantees we require a mathematically rigorous semantic foundation of abstract cost. This is provided in the following.

3.1 Abstract Execution

Abstract Execution [46, 47] extends symbolic execution by permitting abstract statements to occur in programs. Thus AE reasons about an *infinite* set of concrete programs. An abstract program contains at least one AS. The semantics of an AS is given by the set of concrete programs it represents, its set of *legal instances*. To simplify the presentation, we only consider normally completing Java code as instances: an instance may not throw an exception, break from a loop, etc.[2] Each AS has an *identifier* and a specification consisting of its frame and footprint. Semantically, instances of an AS with identifier P may at most write to memory locations specified in P's frame and may only read the values of locations in its footprint. All occurrences of an AS with the *same identifier* symbol have the same legal instances (possibly modulo renaming of variables, if variable names in frame and footprint specifications differ). For example, by

```
// @ assignable x,y;
// @ accessible y,z;
\abstract_statement P;
```

we declare an AS with identifier "P", which can be instantiated by programs that write at most to variables x and y, while only depending on variables y and z. The program "x=y; y=17;" is a legal instance of it, but not "x=y; y=w;", which accesses the value of variable w not contained in the footprint.

We use the shorthand $\mathsf{P}(\mathsf{x},\mathsf{y} :\approx \mathsf{y},\mathsf{z})$ for the AS declaration above. The left-hand side of ":≈" is the frame, the right-hand side the footprint. Abstract programs allow to express certain second-order properties such as "all programs assigning at most x, y while reading at most y, z leave the value of i unchanged". In *Hoare triple* format (where i_0 is a fresh constant not occurring in P):

$$\{\mathsf{i} \doteq i_0\}\ \mathsf{P}(\mathsf{x},\mathsf{y} :\approx \mathsf{y},\mathsf{z});\ \{\mathsf{i} \doteq i_0\} \qquad (*)$$

[2]The AE framework [47] can also deal with exceptional and abrupt termination.

3.2 Abstract Execution with Abstract Cost

We extend the AE framework [46, 47] to QAE by adding *cost specifications* that extend the specification of an AS with an annotated *cost expression*. An abstract cost expression is a function whose value may depend on any memory location in the footprint of the AS it specifies. This location set is called the *cost footprint*, specified via the **cost_footprint** keyword (see Figure 1), and must be a subset of the footprint of the specified AS. If the cost footprint for the program in (∗) is defined to be "$\{z\}$" then this implicitly declares abstract functions $\mathsf{ac}^l_\mathsf{P}(z)$, $\mathsf{ac}^u_\mathsf{P}(z)$ that might be instantiated to, for example, quadratic cost "z^2".

Definition 3.1 (Abstract Program). A pair $\mathcal{P} = (abstrStmts, p_{abstr})$ of a set of AS declarations $abstrStmts \neq \emptyset$ and a program fragment p_{abstr} containing exactly those ASs is called *abstract program*. Each AS declaration in *abstrStmts* is a triple $\Big(\mathsf{P}(frame :\approx footprint),\ \mathsf{ac}^l_\mathsf{P}(costFootprint)\ ,\ \mathsf{ac}^u_\mathsf{P}(costFootprint)\Big)$, where P is an identifier; *frame*, *footprint*, and $costFootprint \subseteq footprint$ are location sets.

A concrete program fragment p is a *legal instance* of $\mathcal{P}$ if it arises from substituting concrete cost functions for all ac_P in *abstrStmts*, and concrete statements for all P in *abstrStmts*, where (i) all ASs are instantiated legally, i.e., by statements respecting their frame, footprint, and cost function, and (ii) all ASs with the same identifier are instantiated with the same concrete program. The semantics $[\![\mathcal{P}]\!]$ consists of all its legal instances.

The abstract program consisting of only AS P in (∗) with cost footprint "{z}" is formally defined as:

$$\left(\left\{\Big(\mathsf{P}(\mathsf{x},\mathsf{y} :\approx \mathsf{y},\mathsf{z}),\ \mathsf{ac}^l_\mathsf{P}(\mathsf{z})\ ,\ \mathsf{ac}^u_\mathsf{P}(\mathsf{z})\Big)\right\}\ ,\ \mathsf{P};\right)$$

The program "P^0 ≡ i =0; **while** (i <z) {x = z; i ++;} " with cost functions "$\mathsf{ac}^l_\mathsf{P}(\mathsf{z}) = \mathsf{ac}^u_\mathsf{P}(\mathsf{z}) = 3 \cdot z + 2$" is a legal instance: it respects frame, footprint, and cost footprint, as well as the cost functions, that (assuming $\mathsf{z} \geq 0$) can be obtained by static cost analysis of P^0.

By encoding the semantics of abstract programs in a program logic [47, Sect. 4.2] one can statically verify whether an instance is legal. It may require auxiliary specifications (invariants, contracts) of the concrete code. The property is undecidable, but can be proven automatically in many cases, see [47] for a discussion. A first implementation of such a check is part of the REFINITY tool (see [45], also https://www.key-project.org/REFINITY/).

In the following we will write ac_P instead of ac^l_P, ac^u_P to mean either, whenever appropriate.

3.3 Cost of Abstract Programs

Finitely executing a concrete program p starting in a state $s_0 = (p, \sigma_0)$ with an initial assignment σ_0 of p's program variables results in a finite trace of the form $t \equiv s_0 \xrightarrow{c_1} \ldots \xrightarrow{c_n} s_n$. Each state $s_i = (p_i, \sigma_i)$ consists of a program counter p_i (the remaining program to execute) and a store σ_i (the current variable assignment); each transition $s_i \xrightarrow{c_{i+1}} s_{i+1}$ updates s_i to s_{i+1} according to the effect of executing command c_{i+1} defined in the semantics of the programming language. A *complete* trace corresponds to a terminating execution, i.e., $s_n = (\epsilon, \sigma_n)$, where ϵ is the empty program and σ_n the resulting final variable assignment.

The cost of a program can be computed based on execution traces. To allow arbitrary quantitative properties, we work on a generic *cost model* $\mathcal{M}$ that assigns cost values to programming language instructions. We compute the cost of a trace t, denoted $\mathcal{M}(t)$, by summing up the cost of the individual instructions. A straightforward measure is the total number of executed instructions $\mathcal{M}_{\mathsf{instr}}$: In this cost model, instructions like "x=1;", the evaluation of a loop guard, etc., all are assigned cost 1. For example, the cost of the complete trace of "**while** (i >0) i −−;" when started with an initial store assigning the value 3 to i is 7, because "i −−;" is executed three times and the guard is evaluated four times. This can be generalized to *symbolic* execution: Executing the same program with a *symbolic* store assigning to i a symbolic initial value $i_0 \geq 0$ produces traces of cost $2 \cdot i_0 + 1$. The cost of *abstract programs*, i.e., the generalization to QAE, is defined similarly: By generalizing not merely over all initial stores, but also over all concrete instances of the abstract program.

Definition 3.2 (Abstract Program Cost). Let $\mathcal{M}$ be a cost model. Let an integer-valued expression $c_{\mathcal{P}}$ consist of scalar constants, program variables, and abstract cost symbols applied to constants and variables. Expression $c_{\mathcal{P}}$ is the *cost of an abstract program $\mathcal{P}$ w.r.t. $\mathcal{M}$* if for all concrete stores σ and instances $p \in [\![\mathcal{P}]\!]$ such that p terminates with a complete trace t of cost $\mathcal{M}(t)$ when executed in σ, $c_{\mathcal{P}}$ evaluates to $\mathcal{M}(t)$ when interpreting variables according to σ, and abstract cost functions according to the instantiation step leading to p. The instance of $c_{\mathcal{P}}$ using the concrete store σ is denoted $c_{\mathcal{P}}(\sigma)$.

Example 3.3. We test the cost assertion in the last lines of the left program in Figure 1 by computing the cost of a trace obtained from a fixed initial store and instances of P, Q. We use the cost model $\mathcal{M}_{\mathsf{instr}}$ and an initial store that assigns 2 to t and 0 to all other variables. We instantiate P with "x=2*t; " and Q with "y=i; y++;". Consequently, the abstract cost functions $\mathsf{ac_P}(\mathsf{t},\mathsf{w})$ and $\mathsf{ac_Q}(\mathsf{t},\mathsf{z})$ are instantiated with 1 and 2, respectively. Evaluating the postulated abstract program cost $2+\mathsf{t}\cdot\big(2+\mathsf{ac_P}(\mathsf{t},\mathsf{w})+\mathsf{ac_Q}(\mathsf{t},\mathsf{z})\big)$ for the concrete store and AS instantiation results in $2+2\cdot(2+1+2) = 12$. Consequently, the execution trace should contain 12 transitions, which is the case.

3.4 Proving Quantitative Properties with QAE

There are two ways to realize QAE on top of the existing functional verification layer provided by the AE framework [46, 47]: (i) provide a "cost" extension to the program logic and calculus underlying AE; (ii) translate non-functional (cost) properties to functional ones. We opt for the second, as it is less prone to introduce soundness issues stemming from the addition of new concepts to the existing framework. It is also faster to realize and allows early testing.

The translation consists of three elements: (a) A global "ghost" variable "cost" (representing keyword "**\cost** ") for tracking accumulated cost; (b) explicit encoding of a chosen cost model by suitable ghost setter methods that update this variable; (c) functional loop invariants and method postconditions expressing cost invariants and cost postconditions.

Regarding item (c), we support three types of cost specifications. These are, in descending order of strictness: *exact*, *upper and lower bound*, and *upper bound* cost. During the analysis stage, it is usually impossible to determine which of them is applicable. For this reason, there are merely two **cost_invariant** keywords, not three, referring only to the cost invariants related to the upper and lower bounds, respectively. However, when translating cost to functional properties, a decision has to be made. A natural strategy is to start with the *strictest* type, then proceed towards the more liberal ones whenever a proof fails.

An exact cost invariant has the shape "cost == *expr*", an upper bound is specified by "cost <= *expr*", a lower bound on the invariant cost is specified by "*expr* <= cost". For example, the (exact) cost postcondition of the abstract program on the right in Figure 1 is:

$$\mathsf{cost}\ \ == 2+\mathsf{ac_P}(\mathsf{t},\mathsf{w})+\mathsf{t}\cdot(\mathsf{ac_Q}(\mathsf{t},\mathsf{z})+2) \qquad (\dagger)$$

Figure 3 shows the result of translating the cost invariant in Figure 1 to a functional loop invariant (highlighted lines), using cost model $\mathcal{M}_{\mathsf{instr}}$ in ghost setters and postconditions of AS ("**ensures**" clauses). Recall that an expression such as "cost == **\before**(cost) + $\mathsf{ac_P}(\mathsf{t},\mathsf{w})$;" is a shorthand for two equations, one with ac^l_P and one with ac^u_P. ASs P, Q must include the ghost variable "cost" in their frame, because they update its value. The keyword **\before** in the

```
//@ ghost int cost = 0;
int i = 0;
//@ set cost = cost + 1;

//@ assignable x, cost;
//@ accessible t, w;
//@ ensures cost == \before(cost) + acP(t, w);
\abstract_statement P;

//@ ghost int iCost = 0;
//@ loop_invariant i ≥ 0 && i ≤ t
//@                 && iCost == i · (acQ(t, z) + 2);
//@ decreases t − i;
while (i < t) {
  //@ set iCost = iCost + 1;
  //@ assignable y, cost;
  //@ accessible i, t, y, z;
  //@ ensures cost == \before(cost) + acQ(t, z);
  \abstract_statement Q;
  i++;
  //@ set iCost = iCost + 1;
}
//@ set cost = cost + 1;
//@ set cost = cost + iCost;
```

Fig. 3. Translation of cost model and cost invariants to AE.

postcondition of an AS refers to the value a variable had just before executing the AS. In loops we use "inner" cost variables "iCost" tracking the cost inside the loop. When the loop terminates, we add the final value of "iCost" to "cost". After every evaluation of the guard of the loop, the cost is incremented accordingly. Using the translation in Figure 3 of the inferred annotations in Figure 1, the AE system proves cost postcondition (†) automatically.

Apart from the translation of inferred quantitative annotations to functional AE specifications, we extended the AE system's *proof script* language. This made it possible to define a highly automated proof strategy for non-linear arithmetic problems generated by some cost analysis benchmarks.

4 ABSTRACT COST ANALYSIS

Recall from Section 2 that for automatic cost certification we need to infer annotations for abstract cost invariants and cost postconditions. To achieve this, we leverage a cost analysis framework for concrete programs to the abstract setting. The presentation is structured as follows: Section 4.1 defines the notion of an abstract cost relation system (ACRS) used in cost analysis for the abstract setting. Section 4.2 defines the concept of cost neutral ASs that ensure independence of ASs in a loop from variables modified in the loop. Section 4.3 details how to generate automatically inductive cost invariants for abstract programs from ACRSs. Section 4.4 tells how to generate cost postconditions used to prove relational properties and required to handle nested loops.

4.1 Inference of Abstract Cost Relations

There are two major cost analysis approaches: those using recurrence equations in the style of Wegbreit [48], and those based on type systems [18, 33]. Our formalization is based on the first kind, but the main ideas for extending the framework to abstract programs are as well applicable to the second. The key issue when extending a recurrences-based framework to the abstract setting is the notion of *abstract cost relation* for loops which generalizes the concept of cost recurrence equations for a loop to an abstract setting. We start with notation for loops and technical details on assumed size relations.

Loops. In our formalization, we consider while-loops containing n abstract statements and m non-abstract statements. Non-abstract statements include any concrete instruction of the target language (arithmetic instructions, conditionals, method calls, …). We assume loops L have the general outline displayed below. Each abstract statement has a specification as shown, abstract and non-abstract statements may appear in any order, either might be empty.

```
while (G) {
  // @ accessible $r_{1,1}, \ldots, r_{1,h_{r1}}$
  // @ assignable $w_{1,1}, \ldots, w_{1,h_{w1}}$
  // @ cost_footprint $c_{1,1}, \ldots, c_{1,h_{c1}}$, $\mathsf{cond}_{1,1}$, ...
  \abstract_statement $A_1$;
  non_abstract_statement $N_1$;
  ...
}
```

Size relations. We assume that for each loop sets of *size constraints* have been computed. These sets capture the size relation among the variables in the loop upon exit (called *base case*, denoted φ_B), and when moving from one iteration to the next (denoted φ_I). ASs are ignored by the size analysis. While this would be unsound in general, it will be correct under the requirements we impose in Definition 4.5 and with the handling of ASs in Definition 4.2. Size relations are available from any cost analyzer by means of a static analysis [17] that records the effect of concrete program statements on variables and propagates it through each loop iteration. In our examples, since we work on integer data, size analysis corresponds to a value analysis [14] tracking the value of the integer variables.[3]

Example 4.1. The size relations for the loop on the left in Figure 1 are $\varphi_B = \{\mathsf{i} \geq \mathsf{t}\}$ and $\varphi_I = \{\mathsf{i} < \mathsf{t},\ \mathsf{i}' = \mathsf{i} + 1\}$. φ_B is inferred from the loop guard and φ_I from the guard and the increment of i (primed variables refer to the value of the variable after the loop execution). □

Based on pre-computed size relations, we define the cost of executing a loop by means of an *abstract cost relation system* (ACRS). This is a set of cost equations characterizing the abstract cost of executing a loop for any input with respect to a given cost model $\mathcal{M}$. Cost equations consist of a cost expression governed by size constraints containing applicability conditions for the equation (like $\mathsf{i} < \mathsf{t}$ in φ_I above) and size relations between loop variables (like $\mathsf{i}' = \mathsf{i} + 1$ in φ_I).

Definition 4.2 (Abstract Cost Relation System). Let L be a loop as above with n abstract and m non-abstract statements. Let $\overline{x}$ be the set of variables accessed in L. Let φ_I, φ_B be sound size relations for L, and $\mathcal{M}$ a cost model. The ACRS for L is defined as the following set of cost equations:

$$\begin{array}{ll} C(\overline{x}) = \mathsf{C_B} & ,\ \varphi_B \\ C(\overline{x}) = \sum_{j=1}^{n} \mathsf{ac_j}\left(c_{j,1}, \ldots, c_{j,h_{cj}}\right) + \sum_{i=1}^{m} \mathsf{C_{N_i}} + C(\overline{x}'), & \varphi_I \end{array}$$

where:

(1) $\mathsf{C_B} \geq 0$ is the cost of exiting the loop (executing the base case) w.r.t. $\mathcal{M}$.
(2) Each $\mathsf{ac_j}(\cdot) \geq 0$ represents the abstract cost for abstract statement A_j in L w.r.t. to $\mathcal{M}$. Each $\mathsf{ac_j}$ is parameterized with the variables in the cost footprint of the corresponding A_j, as it may depend on any of them.
(3) Each $\mathsf{C_{N_i}} \geq 0$ is the cost of the non-abstract statement N_i w.r.t. to $\mathcal{M}$.
(4) C on the right-hand side is a recursive call.
(5) $\overline{x}'$ are variables $\overline{x}$ when renamed after executing the loop.
(6) The assignable variables $w_{j,*}$ in the $\mathsf{ac_j}$ get an unknown value in $\overline{x}'$ (denoted with "_" in the examples below).

Ignoring the abstract statements, one can apply a complete algorithm for cost relation systems [10] to an ACRS to obtain automatically a *linear*[4] ranking function f for loop L: f is a linear, non-negative function over $\overline{x}$ that decreases

[3]For complex data structures, one would need heap analyses [44] to infer size relations.
[4]There exist (more expensive) algorithms to obtain also polynomial ranking functions [9] but for the sake of efficiency we are not using them.

strictly at every loop iteration. Function f yields directly the "// @ **decreases** f;" annotation required for QAE. It gives us an upper bound on the number of iterations of loop L. In analogy to ranking functions, we use metering functions to obtain lower bounds: a *metering function* is a function that decreases at most by one in each iteration and that, when exiting the loop, is zero or negative. It provides a lower bound on the number of iterations of loop L.

As in Section 3, the definition of ACRS assumes a generic cost model $\mathcal{M}$ and uses `C` to refer in a generic way to cost according to $\mathcal{M}$. For example, to infer the number of executed steps, `C` is set to 1 per instruction, while for memory usage `C` records the amount of memory allocated by an instruction.

General Case of an ACRS. The definition of ACRS was simplified for presentation. The following generalizations, not requiring any new concept, are possible: (1) We assume an ACRS for a loop has only two equations, one for the base case (the guard G does not hold) and one for the iterative case (G holds). In general, there might be more than one equation for the base case, e.g., if the guard involves multiple conditions and the cost varies depending on the condition that holds on the exit. Similarly, there might be multiple equations in the iterative case, e.g., if the loop body contains conditional statements and each iteration has different cost depending on the taken branch. This issue is orthogonal to the extension to abstract cost. (2) A loop might contain method calls that in turn contain ASs. In absence of recursion, such calls can be inlined. For recursive methods, it is possible to compute the call graph and solve the equations in reverse topological order such that the abstract cost of the (inner) method calls is obtained first and then inserted into the surrounding equations. (3) The cost of code fragments not part of any loop (before, after, and in between loops) is defined as well by abstract cost equations accumulating the cost of all instructions these fragments include, just as for concrete programs. This aspect does not require changes to the framework for concrete programs, so we do not formalize it, but just illustrate it in the next example.

Example 4.3. The ACRSs of the programs in Figure 1 are (left program above line, right program below):

$$\begin{array}{rll}
C_{\text{before}}(\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) = & c_{\text{before}} + C_{w_0}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}), & \{\mathsf{i}=0\}\\
C_{w_0}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) = & c_{B_{w_0}}, & \{\mathsf{i}\geq\mathsf{t}\}\\
C_{w_0}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) = & c_{w_0} + \mathsf{ac}_{\mathsf{P}}(\mathsf{t},\mathsf{w}) + \mathsf{ac}_{\mathsf{Q}}(\mathsf{t},\mathsf{z}) + C_{w_0}(\mathsf{i}',\mathsf{t},\_,\mathsf{w},\_,\mathsf{z}), & \{\mathsf{i}'=\mathsf{i}+1,\mathsf{i}<\mathsf{t}\}\\
\hline
C_{\text{after}}(\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) = & c_{\text{after}} + \mathsf{ac}_{\mathsf{P}}(\mathsf{t},\mathsf{w}) + C_{w_1}(\mathsf{i},\mathsf{t},\_,\mathsf{w},\mathsf{y},\mathsf{z}), & \{\mathsf{i}=0\}\\
C_{w_1}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) = & c_{B_{w_1}}, & \{\mathsf{i}\geq\mathsf{t}\}\\
C_{w_1}(\mathsf{i},\mathsf{t},\mathsf{x},\mathsf{w},\mathsf{y},\mathsf{z}) = & c_{w_1} + \mathsf{ac}_{\mathsf{Q}}(\mathsf{t},\mathsf{z}) + C_{w_1}(\mathsf{i}',\mathsf{t},\mathsf{x},\mathsf{w},\_,\mathsf{z}), & \{\mathsf{i}'=\mathsf{i}+1,\mathsf{i}<\mathsf{t}\}
\end{array}$$

Notation c refers to the generic cost that can be instantiated to a chosen cost model $\mathcal{M}$. Cost equation C_{before} for the first program is composed of the cost of the instructions appearing in front of the loop as c_{before} plus the cost C_{w_0} of executing the loop. The size constraint fixes the initial value of i. Following Definition 4.2, there are two equations corresponding to the base case of the loop and executing one iteration, respectively. Observe that assignable variables in ASs have unknown values in the ACRS (according to item (6) in Definition 4.2). Program *after* has a similar structure. A ranking and a metering function for each loop is $\mathsf{t}-\mathsf{i}$, which leads to an exact cost estimation as the bounds match. The ranking function is used to generate annotation "// @ **decreases** t−i;" inserted before each loop in Figure 1. □

Example 4.4. In the following example, a tight ranking function is t, while a tight metering function is $\frac{t}{2}$:

```
while (t > 0) {
  if (*) t = t−2; else t = t−1;
}
```

□

For soundness of abstract cost analysis it is essential that (i) no AS in the loop modifies any of the variables that influence loop cost, i.e., they do not *interfere with cost*, and (ii) the cost of the AS in the loop is independent of the variables modified in the loop. We call the latter ASs *cost neutral*. The first requirement is guaranteed by item (6) in Definition 4.2, because the value of assignable variables is "forgotten" in the equations. It is implemented, as usual in static analysis, by using a name generator for *fresh* variables. If cost depends on assignable variables in an AS, then the ACRS will not be solvable (i.e., the analysis returns "unbound cost"). The ACRS in the example contains "_" in equations that do not prevent solvability of the system nor its evaluation, because they do not interfere with cost. However, if we had "forgotten" a cost-relevant variable (such as t), we would be unable to solve or evaluate the equations: without knowing t the equation guard is not evaluable. To fulfill requirement (ii) we introduce the new notion of a *cost neutral AS* to ensure that variables in the cost footprint are not modified by other statements in the loop that decrease the cost.

4.2 Cost Neutral Abstract Statements

Due to the number of involved components of cost specifications, the following definition is a little long-winded, but conceptually straightforward.

Definition 4.5 (Cost Neutral Abstract Statement). Given a loop L, where:

- $W(L)$ is the set of variables written to by the non-abstract statements of L;
- $\texttt{Abstr}(L)$ is the set of all ASs in loop L;
- $\mathit{Frame}(\texttt{Abstr}(L))$ is the set of variables assigned by any AS $A \in \texttt{Abstr}(L)$;
- $\mathit{CostFootprint}(A)$ is the set of variables on which the cost of an A depends;
- $\mathit{CostFootprint}(\texttt{Abstr}(L))$ is the set of variables on which the cost of any AS A of L depends;
- $\mathit{Conditions}(\texttt{Abstr}(L))$ is the set of conditions defined in the cost footprint of any AS $A \in \texttt{Abstr}(L)$;
- $\mathit{Vars}(\texttt{Conditions}(\texttt{Abstr}(L)))$ is the set of variables occurring in $\texttt{Conditions}(\texttt{Abstr}(L))$;
- $R(L)$ is the ranking function of loop L;
- $m(L)$ is the metering function of loop L;
- $\texttt{ac}^u_{\texttt{P}}$ and $\texttt{ac}^l_{\texttt{P}}$ are the upper and lower bounds, respectively, on the abstract cost of abstract statement P.

L is a loop with *cost neutral* ASs if, for any of its ASs A either (1) any variable not occurring in conditions of the cost footprint and that is assigned in the loop does not occur in the cost footprint; or (2) if a variable occurs in a cost footprint condition, then the changes in the cost referring to that variable are safe. Formally:

(1) **(Unconditioned)** For all $x \notin \mathit{Vars}(\mathit{Conditions}(\texttt{Abstr}(L)))$, it holds that

$$x \in \mathit{Frame}(\texttt{Abstr}(L)) \cup W(L) \quad \Longrightarrow \quad x \notin \mathit{CostFootprint}(A)$$

(2) **(Safe)** For all $x \in \mathit{Vars}(\texttt{Conditions}(\texttt{Abstr}(L)))$ the following conditions must hold. Let x' refer to the update of variable x.

(a) **(SafeAS)** If $x \in \mathit{Frame}(\texttt{Abstr}(L)) \cup \mathit{CostFootprint}(\texttt{Abstr}(L))$, then

$$\texttt{Conditions}(\texttt{Abstr}(L)) \Longrightarrow \texttt{ac}^l_{\texttt{A}}[x'/x] \leq \texttt{ac}^l_{\texttt{A}} \quad \wedge \quad \texttt{ac}^u_{\texttt{A}} \leq \texttt{ac}^u_{\texttt{A}}[x'/x]$$

(b) If $x \in W(L) \cup \mathit{Frame}(\texttt{Abstr}(L))$, then

(b)-(i) **(SafeRanking)**

$$\texttt{Conditions}(\texttt{Abstr}(L)) \Longrightarrow R(L)[x'/x] \leq R(L)$$

(b)-(ii) **(SafeMetering)** If $R(L) \neq m(L)$, it holds that

$$\texttt{Conditions(Abstr}(L)) \implies m(L) \leq m(L)[x'/x]$$

□

Condition (2)(a) ensures that changes in the abstract statement costs preserve upper and lower bounds of the corresponding costs. Conditions (2)(b)-(i), (2)(b)-(ii) guarantee that the upper and lower bounds on the iterations remain valid. Metering needs merely to be checked if it does not coincide with the ranking function.

Example 4.6. Both loops in Figure 1 have cost neutral ASs. No conditions are declared in the cost footprint, so Definition 4.5(1) applies. In the left program $Frame(\{P, Q\}) \cup W(L) = \{\mathsf{x}, \mathsf{y}\}, \mathsf{i}$, $CostFootprint(P) = \{\mathsf{t}, \mathsf{w}\}$, and $CostFootprint(Q) = \{\mathsf{t}, \mathsf{z}\}$, so the condition is fulfilled. Similar for the program on the right. □

int i = 0;
while (i < t) {
  // @ **assignable** x,w;
  // @ **accessible** t, w;
  // @ **cost_footprint** t, w,
    $\mathsf{ac_P}(t, w') \geq \mathsf{ac_P}(t, w)$;
  \abstract_statement P;
 	
  // @ **assignable** y;
  // @ **accessible** i, t, y, z;
  // @ **cost_footprint** t, z;
  \abstract_statement Q;
 	
  i ++;
}
	int i = 0;
	while (i < t) {
	  // @ **assignable** x;
	  // @ **accessible** t, w;
	  // @ **cost_footprint** t, w;
	  \abstract_statement P;
	
	  // @ **assignable** y,i;
	  // @ **accessible** i, t, y, z;
	  // @ **cost_footprint** t, z, $i' \geq i$;
	  \abstract_statement Q;
	
	  i ++;
	}
		int i = 0;
		while (i < t) {
		  // @ **assignable** x;
		  // @ **accessible** t, w;
		  // @ **cost_footprint** t, w;
		  \abstract_statement P;
		
		  // @ **assignable** y,t;
		  // @ **accessible** i, t, y, z;
		  // @ **cost_footprint** t, z, $t' \geq t$;
		  \abstract_statement Q;
		
		  i ++;
		}

(2)(a) holds, because $\mathsf{ac_P}(t, w') \geq \mathsf{ac_P}(t, w)$.	Ranking and metering functions are both $t - i$. (2)(b)-(i) holds, because $i' \geq i \implies t - i' \leq t - i$.	Ranking function is $t - i$. (2)(b)-(i) does not hold, because $t' \geq t \;\not\Longrightarrow\; t' - i \leq t - i$.

Fig. 4. Loops illustrating Definition 4.5.

Example 4.7. Figure 4 contains further examples and their justification. We only discuss conditions that need to be checked. For example, in the first loop, (2)(b)-(i) and (2)(b)-(ii) trivially hold, because their conditions do not contain variables that occur in the ranking function, so the only condition that needs to be checked is (2)(a). □

Given a program $\mathcal{P}$ with variables $\overline{x}$ and ACRS with initial equation $C_{ini}(\overline{x})$. We denote by $eval(C_{ini}(\overline{x}), \sigma_0)$ the evaluation of the ACRS for a given initial store σ_0. This is a standard evaluation of recurrence equations performed by instantiating the right-hand side of the equations with the values of the variables in σ_0 and checking the satisfiability of the size constraints (if the expression being checked or accumulated contains "_", the evaluation returns "unbound"). As usual, the process is repeated until an equation without calls is reached. Moreover, we denote by $eval(C_{ini}(\overline{x}), \sigma_0)^u$ the *upper bound of the evaluation* of the ACRS, obtained by substituting each abstract statement $\mathsf{ac_P}$ in $eval(C_{ini}(\overline{x}), \sigma_0)$ by its upper bound ac^u_P. Similarly, we define the *lower bound of the evaluation* $eval(C_{ini}(\overline{x}), \sigma_0)^l$ by substituting each abstract statement $\mathsf{ac_P}$ by its lower bound ac^l_P. It is trivial to see that $eval(C_{ini}(\overline{x}), \sigma_0)^l \leq eval(C_{ini}(\overline{x}), \sigma_0) \leq eval(C_{ini}(\overline{x}), \sigma_0)^u$.

Example 4.8. Consider the ACRS of the left program in Figure 1 with variables $\overline{\mathsf{x}} = (\mathsf{t}, \mathsf{x}, \mathsf{w}, \mathsf{y}, \mathsf{z})$, initial state $\sigma_0 = (2, 0, 0, 0, 0)$, and cost model $\mathcal{M}_{\text{inst}}$ (i.e., c_{before}, $c_{B_{w_0}}$, and c_{w_0} assume values 1, 1, and 2, respectively). The evaluation of the ACRS is $eval(C_{ini}(\overline{\mathsf{x}}), \sigma_0) = 6{+}2{\cdot}\mathsf{ac}_{\mathsf{P}}(2, 0){+}2{\cdot}\mathsf{ac}_{\mathsf{Q}}(2, 0)$. Its upper bound is $eval(C_{ini}(\overline{\mathsf{x}}), \sigma_0)^u = 6{+}2{\cdot}\mathsf{ac}^u_{\mathsf{P}}(2, 0){+}2{\cdot}\mathsf{ac}^u_{\mathsf{Q}}(2, 0)$, its lower bound is $eval(C_{ini}(\overline{\mathsf{x}}), \sigma_0)^l = 6 + 2 \cdot \mathsf{ac}^l_{\mathsf{P}}(2, 0) + 2 \cdot \mathsf{ac}^l_{\mathsf{Q}}(2, 0)$. □

Theorem 4.9 (Soundness of ACRS). *Let $\mathcal{M}$ be a cost model and $\mathcal{P}$ an abstract program with cost neutral ASs in loops (Definition 4.5), $c_{\mathcal{P}}$ the abstract cost of $\mathcal{P}$ as in Definition 3.2, and C_{ini} the initial equation for the ACRS obtained by Definition 4.2. Then, for any initial state of the variables $\sigma_0 \in \mathbb{Z}^{n_m}$, it holds that $eval(C_{ini}(\overline{x}), \sigma_0)^l \leq c_{\mathcal{P}}(\sigma_0) \leq eval(C_{ini}(\overline{x}), \sigma_0)^u$.*

Proof. First we prove $c_{\mathcal{P}}(\sigma_0) \leq eval(C_{ini}(\overline{x}), \sigma_0)^u$. As $c_{\mathcal{P}}$ satisfies Definition 3.2, in particular, for the concrete store σ_0 it is the case that $c_{\mathcal{P}}(\sigma_0)$ evaluates to $\mathcal{M}(t)$ for any complete trace t beginning in σ_0. Hence, it is sufficient to check that $\mathcal{M}(t) \leq eval(C_{ini}(\overline{x}), \sigma_0)^u$. We focus on soundness for the abstract statements—soundness of concrete statements is covered by previous results [4]: Definition 4.5(2)(b)-(i) ensures that in case the variables that affect the cost of the loop are updated, the resulting ranking function does not become greater than the one used in the evaluation of the ACRS, that is, the ranking function inferred at the beginning remains a valid upper bound.

Turning to abstract cost, let A be an abstract statement of $\mathcal{P}$ and σ_1 the store in t when A is executed. It is sufficient to check that cost $\mathsf{ac}^u_{\mathsf{A}}(\sigma_1)$ in $\mathcal{M}(t)$ contributes at least the same to $eval(C_{ini}(\overline{x}), \sigma_0)^u$ as A. If the abstract cost is unchanged by the abstract statement, by definition of upper bound, we obtain $\mathsf{ac}^u_{\mathsf{A}}(\sigma_1) \geq \mathsf{ac}_{\mathsf{A}}(\sigma_1)$. If the abstract cost is changed, then by Definition 4.5(2)(a) we have that the new abstract cost is greater than the previous upper bound on the abstract cost. If $\mathsf{ac}^{u'}_{\mathsf{A}}(\sigma_1)$ is the new abstract cost upper bound then $\mathsf{ac}^{u'}_{\mathsf{A}}(\sigma_1) \geq \mathsf{ac}^u_{\mathsf{A}}(\sigma_1) \geq \mathsf{ac}_{\mathsf{A}}(\sigma_1)$.

Proving $eval(C_{ini}(\overline{x}), \sigma_0)^l \leq c_{\mathcal{P}}(\sigma_0)$ is similar, but we need to check $eval(C_{ini}(\overline{x}), \sigma_0)^l \leq \mathcal{M}(t)$ and take into account 4.5(2)(b)-(ii), This ensures that in case variables that affect the cost of the loop are updated, the resulting metering function does not become lower than the one used to underestimate the length of the trace of the abstract cost. As before, it is sufficient to show that the cost $\mathsf{ac}^l_{\mathsf{A}}(\sigma_1)$ in $\mathcal{M}(t)$ contributes at most the same to $eval(C_{ini}(\overline{x}), \sigma_0)^l$ as A. This holds, because Definition 4.5(1) enforces that variables affecting the cost of A are unmodified by other statements inside the loop, hence $\mathsf{ac}^l_{\mathsf{A}}(\sigma_1) \leq \mathsf{ac}_{\mathsf{A}}(\sigma_1)$ or else, in case of being modified, 4.5(2)(a) ensures that the cost when updating the variables according to conditions defined in the cost footprint remains the same or becomes a lower bound. That is, if $\mathsf{ac}^{l'}_{\mathsf{A}}(\sigma_1)$ is the new abstract cost lower bound, then $\mathsf{ac}^{l'}_{\mathsf{A}}(\sigma_1) \leq \mathsf{ac}^l_{\mathsf{A}}(\sigma_1) \leq \mathsf{ac}_{\mathsf{A}}(\sigma_1)$. □

4.3 From ACRS to Abstract Cost Invariants

Example 4.8 shows that ACRSs are evaluable for concrete instances. However, to enable automated QAE, we need to obtain from them *closed-form* cost invariants and postconditions, i.e., non-recursive expressions. We introduce the novel concept of *abstract cost invariant* (ACI) that enables automated, inductive proofs over cost in a deductive verification system. The crucial difference to (non-inductive) cost postconditions as inferred by existing cost analyzers is that ACIs can be proven inductively for each loop iteration. Hence, they integrate naturally into deductive verification systems that use loop invariants [30].

In contrast to ACIs, postconditions provide a bound for the cost *after* execution of the *whole* loop they refer to. Typically, an upper bound postcondition for a loop has the form $max\_iter * max\_cost + max\_base$, where max_iter is the maximal number of iterations of the loop, max_cost is the maximal cost of any loop iteration, and max_base is the maximal cost of executing the loop with no iterations. A lower bound postcondition is defined similarly, using a lower bound on the number of iterations, lower bounds on the abstract cost and the minimal concrete cost (i.e., $min\_iter * min\_cost + min\_base$).

An upper bound ACI has the form $growth_u * max\_cost + max\_base$, where $growth_u$ counts how many times the loop has been executed at most and hence provides an upper bound after *each* loop iteration. Similarly, a lower bound ACI has the form $growth_l * min\_cost + min\_base$, where $growth_l$ counts how many times the loop has been executed at least and provides a lower bound after *each* loop iteration. The challenge is to design automated techniques that infer upper and lower bounds $growth_u$, $growth_l$.

Definition 4.10 (Upper and Lower Growth). Given a loop with ranking function $R = c + \sum_i a_i \cdot v_i$, metering function $m = d + \sum_i b_i \cdot v_i$, where c, d are the constant parts, v_i the variable parts, and a_i, b_i constant coefficients. If we denote with v_i^0 the initial value of variable v_i before entering the loop, then the *upper growth* and the *lower growth* are defined as

$$growth_u = \sum_i a_i \cdot \left(v_i^0 - v_i\right) \qquad\qquad growth_l = \sum_i b_i \cdot \left(v_i^0 - v_i\right)$$

For any loop execution, $growth_l \leq growth \leq growth_u$, where *growth* is the exact number of performed iterations.

Example 4.11. In Example 4.4, let t_0 be the initial value of t. Recall that a ranking function was t and a metering function was $\frac{t}{2}$. Then upper and lower growths are

$$growth_u = t_0 - t \qquad growth_l = \frac{t_0 - t}{2}$$

□

Example 4.12. We look at four simple loops, where ranking and metering functions coincide. Functions $growth_u$, $growth_l$ are inferred automatically by applying Definition 4.10. They coincide as well and thus provide exact *growth*.

int i = 0;
while (i < t)
i ++;
	ranking/*metering*
	growth
		t − i
		i
			int i = t;
			while (i > 0)
			i −−;
				ranking/*metering*
				growth
					i
					t − i

int i = 0;
while (i < t)
i += 2;
	ranking/*metering*
	growth
		$\frac{\mathsf{t-i+1}}{2}$
		$\frac{\mathsf{i}}{2}$
			int i = t;
			while (i > 0)
			i −= 2;
				ranking/*metering*
				growth
					$\frac{\mathsf{i+1}}{2}$
					$\frac{\mathsf{t-i}}{2}$

We can now define the concept of an ACI that relies on abstract cost relations as defined in Section 4.1 and growth as defined above.

Definition 4.13 (Abstract Cost Invariant). Given an ACRS as in Definition 4.2 with growth as in Definition 4.10, its upper and lower *abstract cost invariants* are defined as follows:

- $\mathsf{cinv}_u(\overline{x}) = \mathsf{C_B}^{\mathtt{max}} + growth_u \cdot \left(\sum_{j=1}^{n} \mathsf{ac}_\mathsf{j}^u\left(c_{j,1},\ldots,c_{j,h_{cj}}\right) + \sum_{i=1}^{m} \mathsf{C_{N_i}}^{\mathtt{max}}\right)$
- $\mathsf{cinv}_l(\overline{x}) = \mathsf{C_B}^{\mathtt{min}} + growth_l \cdot \left(\sum_{j=1}^{n} \mathsf{ac}_\mathsf{j}^l\left(c_{j,1},\ldots,c_{j,h_{cj}}\right) + \sum_{i=1}^{m} \mathsf{C_{N_i}}^{\mathtt{min}}\right)$

where $\mathsf{C_B}^{\mathtt{max}}$ and $\mathsf{C_B}^{\mathtt{min}}$ stand for the maximal and minimal value, respectively, the expression $\mathsf{C_B}$ can assume under the constraints φ_B; $\mathsf{C_{N_i}}^{\mathtt{max}}$ and $\mathsf{C_{N_i}}^{\mathtt{min}}$ are the maximal and minimal value, respectively, of $\mathsf{C_{N_i}}$ under φ_I.

The maxima and minima can be provided by cost analyzers and they give rise to the automatically generated annotations "// @ **cost_invariant_upper** $\mathsf{cinv}_u(\overline{x})$;" and "// @ **cost_invariant_lower** $\mathsf{cinv}_l(\overline{x})$;".

Example 4.14. Consider the first loop in Example 4.12, where $growth_u = growth_l =$ i, with the abstract statement:

```
// @ assignable j;
// @ accessible i , t , j , k;
// @ cost_footprint k;
\abstract_statement P;
```

Under $\mathcal{M}_{\mathsf{instr}}$ the evaluation of the loop guard and the increase of i both have unit cost, so the ACRS is:

$$\begin{array}{ll} C(\mathsf{i},\mathsf{t},\mathsf{j},\mathsf{k}) = 1 & \{\mathsf{i} \geq \mathsf{t}\} \\ C(\mathsf{i},\mathsf{t},\mathsf{j},\mathsf{k}) = \mathsf{ac_P}(\mathsf{k}) + 2 + C(\mathsf{i}',\mathsf{t},\_,\mathsf{k}) & \{\mathsf{i}' = \mathsf{i}+1,\ \mathsf{i} < \mathsf{t}\} \end{array}$$

As usual, the value of the assignable variable j in the recursive call is "forgotten". We obtain the following ACIs: "// @ **cost_invariant_upper** 1 + i * (2 + acP(k));" and "// @ **cost_invariant_lower** 1 + i * (2 + acP(k));". □

Example 4.15. Consider Example 4.11, where $growth_u = t_0 - t$ and $growth_l = \frac{t_0 - t}{2}$, with the abstract statement:

```
// @ assignable w;
// @ accessible y,t , w;
// @ cost_footprint t;
\abstract_statement P;
```

Under $\mathcal{M}_{\mathsf{instr}}$ the evaluation of the loop guard and the decreasing of y both have unit cost, so the ACRS is:

$$\begin{array}{ll} C(\mathsf{y},\mathsf{t},\mathsf{w}) = 1 & \{\mathsf{y} \leq 0\} \\ C(\mathsf{y},\mathsf{t},\mathsf{w}) = \mathsf{ac_P}(\mathsf{t}) + 2 + C(\mathsf{y}',\mathsf{t},\_) & \{\mathsf{y}' = \mathsf{y}-1,\ \mathsf{y} > 0\} \\ C(\mathsf{y},\mathsf{t},\mathsf{w}) = \mathsf{ac_P}(\mathsf{t}) + 2 + C(\mathsf{y}',\mathsf{t},\_) & \{\mathsf{y}' = \mathsf{y}-2,\ \mathsf{y} > 0\} \end{array}$$

Again, the value of assignable variable w is "forgotten", but it does not affect ACRS solvability. We obtain the ACIs: "// @ **cost_invariant_upper** $1 + (t_0 - t) * (2 + \mathsf{ac_P}(k))$;", "// @ **cost_invariant_lower** $1 + \frac{t_0-t}{2} * (2 + \mathsf{ac_P}(k))$;". □

To obtain the maximal and minimal cost of a cost expression under a set of constraints, we use existing maximization procedures [9] and lower bound techniques [6], respectively.

From Definition 4.13 we obtain ACIs as closed form abstract cost expressions of the shape

$$\mathtt{abexpr} = \mathtt{cexpr} \mid \mathtt{ac} \mid \mathtt{abexpr}_1 + \mathtt{abexpr}_2 \mid \mathtt{abexpr}_1 * \mathtt{abexpr}_2 \ ,$$

where ac represents an abstract cost function as defined in Section 3.2 and cexpr is a concrete cost expression. The definition above yields linear bounds, however, the extension to infer postconditions in the subsequent section leads to polynomial expressions (of arbitrary degree).[5]

Example 4.16. Sometimes the cost of a single iteration cannot be computed exactly by cost analyzers, resulting in maximized and minimized cost expressions for concrete instructions. To illustrate this phenomenon, we add an instruction that creates an array of non-constant size "i" to the program in Example 4.14 and measure memory consumption instead of instruction count:

[5]Our approach being based on a recurrences-based framework [48], it works for exponential and logarithmic expressions. The results in this section generalize accordingly. However, the AE deductive verification system is not able to deal with them automatically at the moment, so we exclude them.

```
while (i < t) {
  a = new int[i];
  //@ assignable j;
  //@ accessible i, t, j, a, k;
  //@ cost_footprint k;
  \abstract_statement P;
  i++;
}
```

The resulting ACRS accumulates cost “i” at each iteration, plus the memory consumed by the abstract statement:

$$\begin{array}{ll} C(\mathsf{i},\mathsf{t},\mathsf{j},\mathsf{k}) = 0, & \{\mathsf{i} \geq \mathsf{t}\} \\ C(\mathsf{i},\mathsf{t},\mathsf{j},\mathsf{k}) = \mathsf{ac}_\mathsf{P}(\mathsf{k}) + \mathsf{i} + C(\mathsf{i}',\mathsf{t},\_,\mathsf{k}), & \{\mathsf{i}' = \mathsf{i}+1, \mathsf{i} < \mathsf{t}\} \end{array}$$

Now, maximizing the expression $\mathsf{C}_{\mathsf{N}_1} = \mathsf{i}$ under $\{\mathsf{i}' = \mathsf{i}+1, \mathsf{i} < \mathsf{t}\}$ results in ${\mathsf{C}_{\mathsf{N}_1}}^{\mathtt{max}} = 4 \cdot (\mathsf{t}-1)$ and minimizing it results in ${\mathsf{C}_{\mathsf{N}_1}}^{\mathtt{min}} = 4$, as we are working with 32 bit integers.[6] Accordingly, we obtain the following cost invariants: “//@ **cost_invariant_upper** $i * (4 * (t-1) + \mathsf{ac}_\mathsf{P}(k))$;” and “//@ **cost_invariant_lower** $i * (4 + \mathsf{ac}_\mathsf{P}(k))$;”. □

While the maximized expression is computed automatically, the minimized expression was manually calculated, because LOBER [6] computes lower bounds on the number of iterations of a given loop, but it does not minimize the cost of single iterations (by default it assumes each iteration has unit cost). For this reason, in the current implementation, the tool only infers upper bounds for the *heap* cost model.

Let c_L denote the abstract cost of executing a loop L (in analogy to $c_\mathcal{P}$ in Definition 3.2, but considering only loop L rather than the whole program $\mathcal{P}$). We denote by c_I the portion of the cost in c_L up to the execution of iteration I.

Proposition 4.17. *Let L be a loop with variables $\overline{x}$ satisfying Definition 4.5, and c_I the abstract cost of L in iteration I, $\mathtt{cinv}_u(\overline{x})$ and $\mathtt{cinv}_l(\overline{x})$ its ACIs, and $\sigma_I \in \mathbb{Z}^{n_m}$ the store after performing iteration I of L. Then the following holds:*

(1) $\mathtt{cinv}_u(\overline{x})$ and $\mathtt{cinv}_l(\overline{x})$ are true on entering the loop;
(2) $\mathtt{cinv}_l(\sigma_I) \leq c_I(\sigma_I) \leq \mathtt{cinv}_u(\sigma_I)$.

Proof Sketch. By a straightforward induction, relying on the soundness of the ACRS of L (Theorem 4.9) and assuming soundness of the ranking function, metering function and maximization and minimization procedures used in Definitions 4.10, 4.13. □

4.4 From Cost Invariants to Postconditions

To handle programs with nested loops and to prove relational properties it is necessary to infer *cost postconditions* for abstract programs. For nested loops the cost postcondition states the abstract cost after complete execution of the inner loop and it is used to compute the invariant of the outer loop. For relational properties, the cost postconditions of two abstract programs are compared. Cost postconditions for concrete programs are obtained by upper and lower bound solvers (e.g., COSTA [4], CoFloCo [22], AProVE [25], LoAT [24], Lober [6]) that compute *max_iter*, an upper bound on the number of iterations and *min_iter*, a lower bound on the number of iterations that a loop performs, by relying on ranking and metering functions. The cost postcondition is obtained by replacing $\mathtt{growth}_u$ with `max_iter` and $\mathtt{growth}_l$ with `min_iter` in the formulas of $\mathtt{cinv}_u(\overline{x})$ and $\mathtt{cinv}_l(\overline{x})$ in Definition 4.13:

[6]Better maximization and minimization results can with techniques such as in [7].

Definition 4.18 (Cost Postcondition). Let L be a loop, *max_iter* an upper bound, *min_iter* a lower bound on the number of iterations of L. From the ACRS for L (Definition 4.2) we infer the *cost postconditions* for L:

- $post_l(\overline{x}) = \mathtt{C_B}^{\mathtt{min}} + min\_iter(\overline{x}) \cdot \left(\sum_{j=1}^{n} \mathtt{ac_j}^{l}\left(c_{j,1}, \ldots, c_{j,h_{cj}}\right) + \sum_{i=1}^{m} \mathtt{C_{N_i}}^{\mathtt{min}}\right)$
- $post_u(\overline{x}) = \mathtt{C_B}^{\mathtt{max}} + max\_iter(\overline{x}) \cdot \left(\sum_{j=1}^{n} \mathtt{ac_j}^{u}\left(c_{j,1}, \ldots, c_{j,h_{cj}}\right) + \sum_{i=1}^{m} \mathtt{C_{N_i}}^{\mathtt{max}}\right)$

and generate the annotations

```
// @ assert post_l(x̄) ≤ cost
// @ assert cost  ≤  post_u(x̄)
```

To infer the postcondition for a complete abstract program, we take the sum of all *cost postconditions* of its top-level loops plus the cost of the non-iterative fragments. Figure 1 shows the cost postconditions for our running example obtained by replacing the growth i of the invariant with the bound t on the loop iterations and requiring $\mathtt{t} \geq 0$. The generation of inductive ACIs for nested loops uses the cost postcondition of inner loops to compute the invariants of the outer ones.

Theorem 4.19 (Soundness of Cost Postconditions). *Let L be a loop over variables $\overline{x}$ satisfying Definition 4.5, and c_L the abstract cost of L. Let $\sigma_L \in \mathbb{Z}^{m_n}$ be the state when L terminates. Then $post_l(\sigma_L) \leq c_L(\sigma_L) \leq post_u(\sigma_L)$.*

Proof Sketch. Analogous to Proposition 4.17. □

5 EXPERIMENTAL EVALUATION

5.1 Threats to Validity

Concerning *internal* threats to validity, there is the question of whether our results for a small, abstract programming language generalize to realistic languages. This is the case, because (i) the cost model is completely parametric and can be instantiated as required, and (ii) both the used cost analyzers as well as the AE framework are implemented for large subsets of Java (although to extend the QAE framework is future work). Checking correctness of experimental outcomes and their interpretation is straightforward. Also the outcome, up to minor variations in observed user time, is deterministic.

The main *external* threat is the relatively small number and size of the performed experiments. However, this threat is mitigated for *methodological* reasons: (i) Since we analyze *abstract* programs, each experiment covers infinitely many concrete instances that can be arbitrarily complex; (ii) because our approach is *compositional*, there is no need to analyze large programs, rather small code fragments containing loops can be analyzed in isolation. In consequence, we only need to make sure that the experiments contain a representative selection of the phenomena encountered in program transformation. We ensured this by using typical examples from optimization [1, 31], parallelization [29], and refactoring [23, 45]

In addition, we need to ensure that our analysis is indeed *automatic*, as claimed.

5.2 Experiments

We implemented our approach downloadable and provide the code at https://tinyurl.com/qae-impl (including required libraries). The archive contains the benchmarks of this section, additional examples, as well as build and usage instructions. The tool is command-line and built on top of an existing cost analysis library for (non-abstract) Java bytecode as well as the deductive verification system KeY [2] including the AE framework [46, 47]. Our implementation consists of three components:

(1) An extension of a cost analyzer (written in PYTHON) to handle abstract JAVA programs, ;
(2) a conversion tool (written in JAVA) translating the output of the analyzer to a set of input files for KeY, and
(3) a `bash` script orchestrating the toolchain, specifically, the interplay between items (1), (2) and the two libraries.

In case of a failed certification attempt, our script offers the choice to open the generated proof in KeY for further debugging. In total, our implementation (excluding the libraries) consists of 1,802 lines of PYTHON, 703 lines of JAVA, and 389 lines of bash code (without blank lines and comments).

To assess effectiveness and efficiency of our approach, we used our QAE implementation to analyze seven representative code optimization rules using cost models $\mathcal{M}_{\mathsf{instr}}$ (rows "1∗"–"6∗" in Table 1) and $\mathcal{M}_{\mathsf{heap}}$ (rows "7∗"). While $\mathcal{M}_{\mathsf{instr}}$ counts the number of instructions, $\mathcal{M}_{\mathsf{heap}}$ measures heap consumption. Rows 8–9 show two examples where the lower and upper bound on the number of iterations of the analyzed loops do not coincide, so the estimation of the number of iterations is not exact. The cost model used in these last two examples is $\mathcal{M}_{\mathsf{instr}}$.

The first column identifies the benchmark ("a" refers to the original program, "b" to the transformed one after application of an optimization rule), the second column named **P** refers to the type of cost result that was proven (exact "e", upper "u", upper-lower "ul"). Column three contains the type of bound (lower or upper) for which results are displayed in the row. Column four shows the inferred growth function for each loop in the program (separated by "," if there is more than one loop), in the fifth column we list the cost postconditions obtained by the analysis (expressions indicating the number of loop iterations are highlighted), and columns six to nine display performance metrics. Time $t_{\mathtt{cost}}$, given in milliseconds, is the time needed to perform the cost analysis. The proof generation time $t_{\mathtt{proof}}$ is given in seconds. We also display the time $t_{\mathtt{check}}$ needed for checking integrity of an already generated proof certificate. Finally, $s_{\mathtt{proof}}$ is the size of the generated KeY proof in terms of number of proof steps.

Even though the time needed for certification is significantly higher than for cost analysis (which is to be expected), each analysis can be performed within one minute. The time to *check* a proof certificate amounts to approximately one fourth to one third of the time needed to *generate* it. We stress that all analyses are *fully automatic*. We briefly describe the nature of each experiment:

1 is a *loop unrolling* transformation duplicating the body of a loop: each copy of the body is put inside an **if**-statement conditioned by the loop guard. In (1b), the cost analyzer over-approximates the number of iterations of the unrolled loop, since there are different possible control flows in the body. This was automatically detected by the certifier which failed to find a proof when exact cost invariants are conjectured. Performing an asymptotic abstraction over both bounds [3] would lead us to the same asymptotic function in both examples. The automation of a sound asymptotic technique is left for future work.

2 is the *CodeMotion* example from Section 2. The result reflects a cost *decrease* in the sense that less instructions need to be executed by the transformed program.

3 implements a *LoopTiling* optimization at compiler level, where a single loop with $n \cdot m$ iterations is transformed into two nested loops, an outer one counting to n and an inner one to m. Since our cost analyzer handles only linear size expressions, the first program uses an auxiliary parameter t, which is then instantiated to value $n \cdot m$.

4 is a *SplitLoop* transformation that splits a loop with two independent parts into two separate loops. We prove that this transformation does not affect the cost up to a constant factor.

5 is an optimization combining *two loops* with the same body structure into one loop.

6 is a *three loops* example, one nested and one simple. The optimization combines the bodies of the outer loop in the nested structure and the simple loop.

	P	Cost analysis results			t_{cost}	t_{proof}	t_{check}	s_{proof}
		Bound	**Growth**	**Bound**	[ms]	[s]	[s]	#nodes
1a	e	Lower	i	$2+t\cdot(6+\mathsf{ac_P}^l(x))$	88.1	22.3	6.6	2,815
		Upper	i	$2+t\cdot(6+\mathsf{ac_P}^u(x))$				
1b	u	Upper	i	$2+t\cdot(14+2\cdot\mathsf{ac_P}^u(x))$	113.7	36.9	5.0	4,990
2a	e	Lower	i	$2+t\cdot(7+\mathsf{ac_P}^l(t,w)+\mathsf{ac_Q}^l(t,z))$	138.4	30.3	7.5	4,037
		Upper	i	$2+t\cdot(7+\mathsf{ac_P}^u(t,w)+\mathsf{ac_Q}^u(t,z))$				
2b	e	Lower	i	$3+\mathsf{ac_P}^l(t,w)+t\cdot(6+\mathsf{ac_Q}^l(t,z))$	87.4	26.9	7.1	3,711
		Upper	i	$3+\mathsf{ac_P}^u(t,w)+t\cdot(6+\mathsf{ac_Q}^u(t,z))$				
3a	e	Lower	i	$2+t\cdot(6+\mathsf{ac_P}^l(k))$	82.8	23.2	6.6	2,994
		Upper	i	$2+t\cdot(6+\mathsf{ac_P}^u(k))$				
3b	e	Lower	i , j	$6+n\cdot m\cdot(6+\mathsf{ac_P}^l(k))$	148.5	30.5	4.0	3,998
		Upper	i , j	$6+n\cdot m\cdot(6+\mathsf{ac_P}^u(k))$				
4a	e	Lower	$i+1$	$2+(l+1)\cdot(7+\mathsf{ac_{Q1}}^l(t,w)+\mathsf{ac_{Q2}}^l(t,z))$	90.3	32.1	7.7	4,439
		Upper	$i+1$	$2+(l+1)\cdot(7+\mathsf{ac_{Q1}}^u(t,w)+\mathsf{ac_{Q2}}^u(t,z))$				
4b	e	Lower	$i+1$, $i+1$	$2+(l+1)\cdot(12+\mathsf{ac_{Q1}}^l(t,w)+\mathsf{ac_{Q2}}^l(t,z))$	140.6	42.5	10.0	5,943
		Upper	$i+1$, $i+1$	$2+(l+1)\cdot(12+\mathsf{ac_{Q1}}^u(t,w)+\mathsf{ac_{Q2}}^u(t,z))$				
5a	e	Lower	i , j	$2+n\cdot(6+\mathsf{ac_P}^l(y))+\mathrm{m}\cdot(6+\mathsf{ac_P}^l(y))$	146.2	32.0	8.5	4,827
		Upper	i , j	$2+n\cdot(6+\mathsf{ac_P}^u(y))+\mathrm{m}\cdot(6+\mathsf{ac_P}^u(y))$				
5b	e	Lower	i	$2+(n+m)\cdot(8+\mathsf{ac_P}^l(y))$	91.6	19.8	6.8	2,759
		Upper	i	$2+(n+m)\cdot(8+\mathsf{ac_P}^u(y))$				
6a	e	Lower	k , j , $n-i$	$6+n\cdot(m\cdot(6+\mathsf{ac_P}^l(y))+n\cdot(5+\mathsf{ac_Q}^l(y))$	187.0	39.7	10.1	6,081
		Upper	k , j , $n-i$	$6+n\cdot(m\cdot(6+\mathsf{ac_P}^u(y))+n\cdot(5+\mathsf{ac_Q}^u(y))$				
6b	e	Lower	k , j	$7+n\cdot(m\cdot(6+\mathsf{ac_P}^l(y))+\mathsf{ac_Q}^l(y))$	143.7	30.9	8.6	4,534
		Upper	k , j	$7+n\cdot(m\cdot(6+\mathsf{ac_P}^u(y))+\mathsf{ac_Q}^u(y))$				
7a	u	Upper	$i-1$	$(t-1)\cdot(4\cdot(t-1)+\mathsf{ac_P}^u(y))$	92.4	21.2	7.0	2,597
7b	u	Upper	$i-1$	$4\cdot m+(t-1)\cdot\mathsf{ac_P}^u(y)$	97.0	18.2	6.2	2,097
8	ul	Lower	$\frac{t-y}{2}$	$3+\frac{t}{2}\cdot(16+\mathsf{ac_P}^l(t,w))$	119.0	46.6	9.8	6,802
		Upper	$t-y$	$3+t\cdot(16+\mathsf{ac_P}^u(t,w))$				
9	ul	Lower	$z-x$	$5+z\cdot(15+\mathsf{ac_P}^l(t))$	200.0	53.6	11.2	8,574
		Upper	$t-y+z-x$	$5+t+z-1\cdot(15+\mathsf{ac_P}^u(t))$				

Table 1. Results of the experiments.

7 is an *array* optimization, where an array declaration is moved in front of a loop, initializing it with an auxiliary parameter that is the sum of all the initial sizes.

8 corresponds to Example 4.4, where the loop is executed between $\frac{t}{2}$ and t times.

9 is a loop example with two variables x and y with initial values z and t. In each loop iteration, x or y is decreased by one, and the loop exits when one of them becomes negative. In this case, given that $z < t$, z is a lower bound while $z + t - 1$ is an upper bound on the number of iterations.

6 RELATED WORK

The present article builds on the AE framework [46, 47], which we extend to *Quantitative* AE. At the moment no other approach or tool is able to analyze and certify the cost of schematic programs, specifically relational properties, so a direct comparison is impossible.

Cost Analysis. There are many resource analysis tools, including: [28], based on introducing counters and inferring loop invariants; [32], based on an analysis over the depth of functional programs formalized by means of type systems. Approaches that bound the number of execution steps include [27, 38], working at the level of compilers. Systems such as APROVE [25] analyze the complexity of JAVA programs by transforming them to integer transition systems; COSTA [4] and COFLOCO [22] are based on the generation of cost recurrence equations from which upper bounds can be inferred. This is also the basis of the approach we pursue to infer abstract upper bounds in Section 4.1, hence our technique can be viewed as a generalization of the latter systems. Approaches based on type systems could also be generalized to work on abstract programs by introducing abstract cost as in Section 4.1.

For our work it is crucial to use ranking and metering functions to infer growth of cost invariants. Ranking functions were used to generate bounds on the number of loop iterations in several systems, but none used them to define growth: [14] obtain runtime complexity bounds via symbolic representation from ranking functions, likewise PUBS [4], LOOPUS [49], and ABC [12]. PUBS analyses all loop transitions at once, LOOPUS uses an iterative procedure where bounds are propagated from inner to outer loops, ABC deals with nested, but not sequential loops. In our work, when inferring upper bounds, we solve all transitions at once and handle nested as well as sequential loops. Metering functions [6, 24] have been less used than ranking functions and hitherto were never applied to growth computation.

Certification. A variety of general purpose deductive software verification tools [30] exist, including VERYFAST [43], WHY [20], DAFNY [37], KIV [42], and KeY [2]. We use KeY, the currently only system to implement AE. *Interactive* proof assistants like Isabelle [40] or Coq [11] also support more or less expressive abstract program fragments, but lack full automation. There are dedicated approaches involving schematic programs for *specific* contexts, like regression verification [26], compilation [31, 35, 39] or derived symbolic execution rules [16].

Regarding the combination of deductive verification and cost analysis, the closest approach to ours is the integration [5] of COSTA [4] and KeY [2] which was realized for concrete, not abstract programs. They verify upper bounds on the cost of concrete programs by decomposing them into ranking functions and size relations which are then verified separately. Here we use the novel concept of cost invariant that allows verification of quantitative properties without decomposition. Paper [5] deals only with the global number of iterations as is common in worst-case cost analysis. Our cost invariants are designed to be inductive and propagate cost through all loop iterations. Radiček et al. [41] devise a formal framework for analyzing the relative cost of different programs (or the same program with different inputs). Compared to our approach, they target purely functional programs extended with monads representing cost, while we work with an industrial programming language. Moreover, we generally reason about the cost of *transformations*, not of a transformation applied to one *particular* program.

7 CONCLUSION AND FUTURE WORK

We presented the first approach to analyze the cost of schematic programs with placeholders. We can infer and verify cost bounds for a potentially infinite class of programs once and for all. In particular, for the first time, it is possible to analyze and prove changes in efficiency caused by program transformations—for all input programs. Our approach supports upper/lower and exact cost and a configurable cost model. We implemented a toolchain based on a cost analyzer and a program verifier which analyzes and formally certifies abstract cost bounds in a fully automated manner. Certification is essential, because only the verifier can determine whether the bounds inferred by the cost analyzer are exact.

Our work required the new concept of an (abstract) cost invariant. This is interesting in itself, because (i) it renders the analysis of nested loops modular and (ii) provides an interface to backends (such as verifiers) that characterizes the cost of code in iterations.

Obvious future work involves extending the analyzed target language. Cost analysis and deductive verification (including AE) are already possible for a large Java fragment [4, 46]. More interesting—and more challenging—is the analysis of program transformations that parallelize code. The extension to larger classes of cost functions, such as logarithmic or exponential, could be realized by integrating non-linear SMT solvers into the toolchain. Finally, sound abstraction of cost bounds to *asymptotic* bounds would allow to compare the asymptotic behavior of program transformations in cases when lower and upper bounds differ.

Acknowledgments. 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 EU and by the UCM CT42/18-CT43/18 grant.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *LNCS*. Springer, 2016.

[3] Elvira Albert, Diego Esteban Alonso-Blas, Puri Arenas, Samir Genaim, and German Puebla. Asymptotic resource usage bounds. In Zhenjiang Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer, 2009.

[4] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.

[5] Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, Germán Puebla, and Guillermo Román-Díez. A formal verification framework for static analysis - as well as its instantiation to the resource analyzer COSTA and formal verification tool KeY. *Software and Systems Modeling*, 15(4):987–1012, 2016.

[6] Elvira Albert, Samir Genaim, Enrique Martin-Martin, Alicia Merayo, and Albert Rubio. Lower-bound synthesis using loop specialization and max-smt. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 863–886. Springer, 2021.

[7] Elvira Albert, Samir Genaim, and Abu Naser Masud. On the inference of resource usage upper and lower bounds. *ACM Transactions on Computational Logic*, 14(3):1–35, aug 2013.

[8] Elvira Albert, Reiner Hähnle, Alicia Merayo, and Dominic Steinhöfel. Certified abstract cost analysis. In Esther Guerra and Mariëlle Stoelinga, editors, *Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12649 of *Lecture Notes in Computer Science*, pages 24–45. Springer, 2021.

[9] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[10] Roberto Bagnara, Fred Mesnard, Andrea Pescetti, and Enea Zaffanella. A new look at the automatic synthesis of linear ranking functions. *Inf. Comput.*, 215:47–67, 2012.

[11] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[12] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. ABC: algebraic bound computation for loops. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *LNCS*, pages 103–118. Springer, 2010.

[13] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, June 1975.

[14] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th Intl. Conf., TACAS, Grenoble, France*, volume 8413 of *LNCS*, pages 140–155. Springer, 2014.

[15] Marc Brockschmidt, Richard Musiol, Carsten Otto, and Jürgen Giesl. Automated termination proofs for Java programs with cyclic data. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *LNCS*, pages 105–122. Springer, 2012.

[16] Richard Bubel, Andreas Roth, and Philipp Rümmer. Ensuring the Correctness of Lightweight Tactics for JavaCard Dynamic Logic. *Electr. Notes Theor. Comput. Sci.*, 199:107–128, 2008.

[17] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978.

[18] Karl Crary and Stephanie Weirich. Resource bound certification. In Mark N. Wegman and Thomas W. Reps, editors, *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, pages 184–198. ACM, 2000.

[19] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower bound cost estimation for logic programs. In *Logic Programming*. The MIT Press, 1997.

[20] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th Intl. Conf., CAV, Berlin, Germany*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[21] Antonio Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In *FM 2016: Formal Methods*, pages 254–273. Springer International Publishing, 2016.

[22] Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In Jacques Garrigue, editor, *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, volume 8858 of *LNCS*, pages 275–295. Springer, 2014.

[23] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Object Technology Series. Addison-Wesley, June 1999. With contributions by Kent Beck, John Brant, Willima Opdyke, and Don Roberts.

[24] Florian Frohn, Matthias Naaf, Marc Brockschmidt, and Jürgen Giesl. Inferring lower runtime bounds for integer programs. *ACM Transactions on Programming Languages and Systems*, 42(3):1–50, dec 2020.

[25] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th Intl. Joint Conf., IJCAR, Vienna, Austria*, volume 8562 of *LNCS*, pages 184–191. Springer, 2014.

[26] Benny Godlin and Ofer Strichman. Regression Verification: Proving the Equivalence of Similar Programs. *Softw. Test., Verif. Reliab.*, 23(3):241–258, 2013.

[27] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, and Kerstin Eder. Static energy consumption analysis of LLVM IR programs. *CoRR*, abs/1405.4565, 2014.

[28] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 127–139. ACM, 2009.

[29] Reiner Hähnle, Asmae Heydari Tabar, Arya Mazaheri, Mohammed Norouzi, Dominic Steinhöfel, and Felix Wolf. Safer parallelization. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 9th Intl. Symp., ISoLA, Proc. Part II*, volume 12477 of *LNCS*, pages 117–137, Cham, October 2020. Springer.

[30] Reiner Hähnle and Marieke Huisman. Deductive verification: from pen-and-paper proofs to industrial tools. In Bernhard Steffen and Gerhard Woeginger, editors, *Computing and Software Science: State of the Art and Perspectives*, volume 10000 of *LNCS*, pages 345–373. Springer, 2019.

[31] Reiner Hähnle and Dominic Steinhöfel. Modular, correct compilation with automatic soundness proofs. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques, 8th Intl. Symp., Proc. Part I, ISoLA, Cyprus*, volume 11244 of *LNCS*, pages 424–447. Springer, 2018.

[32] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP, Paphos, Cyprus*, volume 6012 of *LNCS*, pages 287–306. Springer, 2010.

[33] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 410–423, New York, NY, USA, 1996. Association for Computing Machinery.

[34] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[35] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving Optimizations Correct Using Parameterized Program Equivalence. In *Proc. PLDI 2009*, pages 327–337, 2009.

[36] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. JML Reference Manual, May 2013. Draft revision 2344.

[37] Rustan Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference, LPAR-16, Dakar, Senegal*, pages 348–370. Springer Berlin Heidelberg, April 2010.

[38] Umer Liqat, Kyriakos Georgiou, Steve Kerrison, Pedro López-García, John P. Gallagher, Manuel V. Hermenegildo, and Kerstin Eder. Inferring parametric energy consumption functions at different software levels: ISA vs. LLVM IR. In Marko C. J. D. van Eekelen and Ugo Dal Lago, editors, *Foundational and Practical Aspects of Resource Analysis - 4th Intl. Workshop, FOPARA, London, UK, Revised Selected Papers*, volume 9964 of *LNCS*, pages 81–100, 2015.

[39] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Practical Verification of Peephole Optimizations with Alive. *Commun. ACM*, 61(2):84–91, 2018.

[40] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[41] Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. Monadic refinements for relational cost analysis. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[42] Wolfgang Reif. The KIV-approach to software verification. In *KORSO - Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *LNCS*, pages 339–370. Springer, 1995.

[43] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for Java-like programs based on dynamic frames. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th Intl. Conf., FASE, Budapest, Hungary*, volume 4961 of *LNCS*, pages 261–275. Springer, 2008.

[44] Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for Java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.*, 32(3):8:1–8:70, 2010.

[45] Dominic Steinhöfel. REFINITY to Model and Prove Program Transformation Rules. In Bruno C. d. S. Oliveira, editor, *Proc. 18th Asian Symposium on Programming Languages and Systems (APLAS)*, LNCS. Springer, 2020.

[46] Dominic Steinhöfel and Reiner Hähnle. Abstract execution. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *LNCS*, pages 319–336. Springer, 2019.

[47] Dominic Steinhöfel. *Abstract Execution: Automatically Proving Infinitely Many Programs.* PhD thesis, Technical University of Darmstadt, Department of Computer Science, Darmstadt, Germany, 2020.

[48] Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

[49] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction (extended version). *CoRR*, abs/1203.5303, 2012.