Bounded Termination of Monotonicity-Constraint Transition Systems

Amir M. Ben-Amram and Michael Vainer *

February 20, 2011

Abstract

Intuitively, if we can prove that a program terminates, we expect some conclusion regarding its complexity. But the passage from termination proofs to complexity bounds is not always clear. In this work we consider Monotonicity Constraint Transition Systems, a program abstraction where termination is decidable (based on the size-change termination principle). We show that these programs also have a decidable complexity property: one can determine whether the length of all transition sequences can be bounded in terms of the initial state. This is the bounded termination problem. Interestingly, if a bound exists, it must be polynomial. We prove that the bounded termination problem is PSPACE-complete and, moreover, if a bound exists then a symbolic bound which is constant-factor tight (in the univariate case) can be computed in PSPACE. We present this computation in the form of computing a reachability bound, a bound on the number of visits to a given program location. This presentation is inspired by the practical usefulness of this problem formulation.

We also discuss, theoretically, the use of bounds on the abstract program to infer conclusions on a concrete program that has been abstracted. The conclusion maybe a polynomial time bound, or in other cases polynomial space or exponential time. We argue that the monotonicity-constraint abstraction promises to be useful for practical complexity analysis of programs.

1 Introduction

On Complexity Analysis of programs. Automatically inferring complexity properties of computer programs is a well-established subfield of static analysis (the related work section will provide bibliographic references). The topic received renewed attention from static analysis researchers in recent years, sometimes called cost analysis, bound analysis or growth-rate analysis. The overall goal is to develop algorithms that can process a subject program and answer questions about its complexity, where complexity may refer to various measures of resource usage such as running time, memory usage, stack usage, etc.

It is well-known that in the analysis of algorithms, questions about precise running time (in physical units) are usually abandoned, since studying this measure involves many properties of complex hardware systems as well as the software platform, which shift the focus from the algorithm itself. In program analysis, one can distinguish works that concentrate on the realtime dimension (often going by the keyword WCET—worst-case execution time analysis), and works that concentrate on more robust (and abstract) program-based measures such as number

*School of Computer Science, Tel-Aviv Academic College. Part of the work of the first author was carried out while visiting DIKU, the Computer Science Department at the University of Copenhagen, Denmark.
of instructions executed or just the number of iterations of a loop. Naturally, some works involve both, to varying degrees, however our work only addresses the program-based analysis.

A typical question that a program analyzer for complexity may be asked to answer is: give an expression of the cost (say, execution time—which we shall understand as the number of program steps) in terms of (some designated) input values.

Since we are not measuring real time anyway, it seems reasonable, as in algorithm textbooks, to neglect input-independent constants and use the $O$-notation. This simplifies the problem, but does not change the basic challenge. Even if we only ask for a complexity class, for example to separate polynomial-time programs from super-polynomial ones, this problem is still undecidable in every Turing-complete programming language. This means that there is no hope to solve the problem! How can an algorithm designer overcome such an obstacle? We list a few alternative approaches (in the context of complexity analysis).

- Focus on specially-designed languages. Such works often grew out of the research on Implicit Computational Complexity (ICC). In fact, a typical result in this field is the proof that a complexity class is precisely captured by a particular sub-recursive (Turing incomplete) programming language. But these languages force the user to program in a particular way, often too unnatural. Other works show that for suitably restricted languages, the complexity classification is not predetermined but is decidable. This is an advantage, as it means that the language is less restricted and a more natural programming style should be possible.

- Give up a complete solution to the problem. This is actually the common approach in the field of static analysis, since research in this field often takes the programming language for given. One then produces analyses that can have “false negatives” or “false positives”; in complexity analysis, the most common goal is to provide an upper bound, thus the question “is the program polynomial?” will occasionally be answered by a false negative, resulting of an overshot upper bound.

- A third approach—perhaps a middle road—may be described as abstract and conquer. The idea is to first translate a program from its original language into an abstract form, and then analyze the abstract form; a useful abstraction captures important aspects of the source program, but it is in the nature of abstraction to lose some precision. One may hope, then, that for abstract programs one really can solve the problem of interest. This may require the development of a good definition of the analysis goals in the abstract world. This approach can already be seen in different fields of program analysis, including complexity analysis, as we will mention in more detail below. It has several benefits, in particular, theoretically, the abstract program model may be sufficiently simple to develop a firm theoretical understanding; as problems may be decidable, one can may be able to progress to proving their computational complexity. Practically, the approach suggests a separation of concerns among a front end and a back end, and promotes modularity in tool construction.

**Termination Analysis.** Termination Analysis is another much-studied topic in program analysis. Intuitively, a termination proof seems likely to reveal something about the complexity of the subject program, since proving termination means proving that the complexity is bounded. It is, therefore, natural to try to extend work on termination proofs to obtain complexity bounds. In fact, some works on complexity analysis have already exploited techniques from termination analysis (polynomial interpretation of terms in [BCMT01, CL92]; ranking functions in [AAGP08, ADFG10a]). In this work too our goal was to examine certain theoretical and al-
constraint from termination analysis and evolve them to obtain results in complexity
analysis. Specifically, we study the monotonocity constraint abstraction.

**Constraint Transition Systems.** A Constraint Transition System (CTS) is an abstract
program which is based on viewing the semantics of the program as an infinite-state transition
system which has a finite description. The components of this description are: first, a control
flow graph (CFG), which is a finite directed graph; we refer to its nodes as flow points. Typically,
they represent concrete locations in the source code of the subject program. Second, a finite set
of variables associated with every flow point; a state is specified by \((\ell, x_1, \ldots, x_n)\) where \(\ell\) is a
flow point and \(x_1, \ldots, x_n\) the values of the variables. The variables may represent actual program
data, abstractions like the size of an object (a list, a tree, a set etc.), in some cases program
constants, and in some cases “invented” variables (created by the analysis tool). Finally, every
arc of the CFG, to which we refer as an abstract transition, is associated with a formula that
represents a relation on source states and target states (the transition relation). We refer to
this formula as a constraint. A common notation for constraints is to denote the target state
variables by primed identifiers. So, for example, \(x > x'\) means that the new value of variable
\(x\) is smaller than the old one. Figure 1 shows a small program and a possible abstraction to
constraints (in fact, to monotonicity constraints, as defined below. The reader should be able
to see that the constraints suffice for deducing that the loop always terminates). Additional
examples appear in later sections.

So far, the definition has been very general, and practically any program representation or
computational model of finite description can be represented in this way. However, certain kinds
of CTS are more frequent in program analysis. To specify a particular kind of CTS, we have to
specify the kind of constraints allowed and over what carrier set they work. In this paper, we
employ the notation \((C, D)\)-CTS for a CTS that applies constraints of type \(C\) to the domain \(D\).

Monotonicity constraints were introduced to termination analysis as early as 1991 [Sag91].
These are constraints that only use order relations \(>\) and \(\geq\), and their use in termination
analysis stems from the idea of proving termination by identifying a descending sequence—a
pattern typical to Logic and Functional programming, where one often recurses on values such
as terms, trees or lists while shrinking them. Hence size-change termination, a name given to
this approach in [LJBA01]. The precise abstraction used in the latter work is this: Constraints
are conjunctions of relations of the form \(x > y'\) or \(x \geq y'\). They are referred to as size-change
graphs (SCG). Thus, the abstraction employed by size-change termination (à-la [LJBA01]) may
be expressed as \((SCG, Ord)\)-CTS, where \(Ord\) stands for “any well-ordered set.”

When one looks at earlier papers using monotonicity constraints (e.g., [Sag91, LS97]), one
may notice that their constraint formulae are not restricted to size-change graphs—there was
no prohibition of constraints such as \(x < x'\) (an increase, rather than decrease) or \(x < y\) (a
constraint on source-state variables) or \(x' < y'\). We refer to this constraint domain as \(MC\). It
also is clear that the intended domain is the non-negative integers. In 2005, Codish, Lagoon
and Stuckey [CLS05] began the extension of size-change termination theory to monotonicity
constraints and the integers. To illustrate the need for refining the theory, note that a loop
described by the constraint \(x < x' \land x < y \land y = y'\), a common pattern in imperative programs,
does not satisfy size-change termination (there are well-ordered sets in which this can be repeated
forever), but terminates over the integers. Note also that when arguing for its termination over
the integers, the assumption \(x, y \geq 0\) is redundant, and in fact in imperative programs the
**Program 1**

\[
\text{while } x > z \text{ do} \\
(x, y) := (y, x-1)
\]

**CFG and constraints**

\[
\begin{aligned}
x &> z \\
y & = x' \\
x &> y' \\
z & = z'
\end{aligned}
\]

Figure 1: CTS abstraction of a simple program (1).

Important variables for loop control are often of integer type, and can be (by design or by mistake) negative too. Note also that the usage of constraints which is not of the “size-change graph” type. This motivated the study of \((\mathcal{M}, \mathbb{Z})\)-CTS in [BA11]. Two significant results of this study are: (1) termination of \((\mathcal{M}, \mathbb{Z})\)-CTS is decidable; it is PSPACE-complete. (2) There is an algorithm for constructing global ranking functions for terminating \((\mathcal{M}, \mathbb{Z})\)-CTS instances.

Other types of CTS have also appeared in termination analysis as well of complexity analysis; more on this below.

**Complexity Analysis of Abstract Programs.** Stated succinctly, a CTS represents a transition relation (relation on the set of states) and the goal of termination analysis is to prove that this relation is well-founded. A natural notion of complexity for the abstract program is the (worst-case) number of transitions starting from an initial state (a state where the program is at its designated point of entry), which we would like to bound in terms of the variables at that initial state (or a few designated variables).

Our research on complexity analysis of \((\mathcal{M}, \mathbb{Z})\)-CTS has been inspired by two earlier works on the complexity analysis of programs, which are both based on a CTS abstraction: the COSTA system of Albert et al. [AAGP08, AAGP10], which targets Java bytecode programs, and the WTC analyzer of Alias et al. [ADFG10a], targeting C programs. For the purpose of this presentation, we follow the latter (more on the former in Section 2). The abstraction used is \((\text{Aff}, \mathbb{Z})\)-CTS where Aff denotes a constraint language where a constraint is a conjunction of linear (affine) inequalities, for example: \(x < 1 \land x + y \leq z\). It should be clear that \((\mathcal{M}, \mathbb{Z})\)-CTS is a sub-model of \((\text{Aff}, \mathbb{Z})\)-CTS. As for analysis of the abstract program, the method is to search for a lexicographic linear ranking function. Roughly speaking, this is a function of the form \(\rho_{\ell}(x_1, \ldots, x_n) = (f_{\ell,1}(x), \ldots, f_{\ell,d}(x))\) where each \(f_{\ell,i}\) is an affine function on \(\mathbb{Z}^n\) whose values in reachable program states \((\ell, x)\) are guaranteed to be non-negative. Moreover, the value of this function decreases (lexicographically) in every transition. It is easy enough to see that this proves termination; it also permits one to bound the running time. The bound will be a polynomial of degree \(d\) (the length of the longest tuple used, also referred to as the dimension). Interestingly, among all functions that satisfy the conditions which [ADFG10a] poses in the search for ranking functions, the algorithm provably finds one of smallest dimension.

Both of the above works were accompanied by front-ends that abstracted programs, demonstrating the applicability of the approach to analysis of concrete programs in the respective languages.

The \((\mathcal{M}, \mathbb{Z})\)-CTS abstraction has, previous to our work only been used for proving termi-
nation. Thus, our first contribution is to define the property of *bounded termination* in this particular context. This may seem a trivial step, but introducing this definition was important as it expressed our realization that *not for every terminating CTS can a complexity bound be obtained* (this will be shown precisely in Section 3). Hence, the class of bounded-terminating instances is a subset of the terminating ones. Now we can ask about the decidability and complexity of this set. Our fundamental result is a proof that bounded termination is decidable.

Moreover: we prove that it is PSPACE-complete. This is the same complexity as for termination; and indeed we re-use some techniques from the work on \((\mathcal{M}, \mathbb{Z})\)-CTS termination in both the upper bound proof and the hardness proof. Unlike [AAGP08, ADFG10a, ZGSV11], we do not use ranking functions.

An interesting consequence of our proofs was the discovery that bounded termination implies that the bound obtained is always polynomial (in terms of the initial values). Note that this is an inherent property—not an artifact of the analysis algorithm.

Given this result, the natural next step is to ask how hard it is to obtain the precise degree of the bounding polynomial (which, for univariate polynomials, determines the bound up to a constant factor). Our theorem, proved in Section 5, shows that this too is decidable—and, naturally, we also determine its complexity class, which is still PSPACE-complete.

While turning attention from bounded termination in general to precise degree bounds, we also change, in Section 5, the object of our study from a bound on the length of a computation to the number of visits to a prescribed point in the program. For this measure we adopt the term *reachability bound* (RB), introduced in [GZ10]. Defining the problem in terms of the reachability bound has some advantages, as a procedure to determine the RB can be easily put to different uses. If the property of interest is the total length of a computation, it is possible to bound it by computing the RB for selected *cut points* in the program (for a “big O” bound, it suffices to ensure that every cycle includes a cut point). The RB problem is more general, since different flow-points may have different bounds. If we are interested in consumption of some resource, consumed at specific points in the program, we can compute the RB for those points. Moreover, computation of flow-point RBs helps modularity in the following sense: suppose that certain flow-points \(f_1, f_2, \ldots\) are in fact procedure calls and we have computed costs of these procedures, \(C_i\). We then obtain an overall bound (though not always tight) by multiplying the RB of each \(f_i\) by its associated cost.

To state our theorem, we define a decision problem, RBD (for *reachability bound degree*): for a given flow-point and degree \(k\), is the RB for the flow-point a polynomial of degree at least \(k\)? This decision problem is proved PSPACE-complete. The algorithm proving the upper bound is based on the notion of a *fully-elaborated* CTS from [BA10b], where it was introduced for constructing global ranking functions. Here, however, it is combined with closure computation (a well-known technique in size-change termination) and uses a new result (Theorem 5.10) that shows how the RB degree is determined by the closure set. A notable corollary of our analysis is that a constant-factor tight reachability bound is always a polynomial (not just polynomially bounded). Here it is assumed that the bound is expressed as a function of a single independent variable \(N\) (this may actually turn out to be the maximum of several actual inputs, a difference of inputs, etc), so that by a tight bound is meant a bound \(f(N)\) such that the actual worst-case bounds in terms of \(N\) is \(\Theta(f(N))\).

1 Concurrently to our work, it was also put to use in complexity analysis by Zuleger et al. [ZGSV11].
Program 2

\[
\begin{align*}
&i = N; \\
&\text{while } (i > 0) \{ \\
&\quad \text{if } (j > 0) \ j --; \\
&\quad \text{else } \{ j = N; \ i --; \} \\
&\}\end{align*}
\]

\[\begin{array}{c}
\circ \\
\downarrow \\
\circ \\
\downarrow \\
\circ \\
\end{array}\]

- (1) \( i > 0 \land \text{Same}(N, 0, i, j) \)
- (2) \( j > 0 \land j > j' \land \text{Same}(N, 0, i) \)
- (3) \( j \leq 0 \land j' = N' \land i > i' \land \text{Same}(N, 0) \)

Figure 2: CTS abstraction of a simple program. The notation Same\((x, y, \ldots)\) is syntactic sugar for indicating abstract variables that are constant in the transition (see Section 3).

\((\mathcal{MC}, \mathbb{Z})\)-CTS as a back-end. Our paper can be viewed as a theoretical study of \((\mathcal{MC}, \mathbb{Z})\)-CTS. However, we argue that such constraint transition systems are useful as an abstraction of “real” programs. To support this claim, we have to discuss the manner in which a concrete program is modeled by a \((\mathcal{MC}, \mathbb{Z})\)-CTS.

In termination analysis, the concrete-abstract connection is always based on the following principle: If the concrete program has an infinite execution, the abstract program will have one. This is achieved in different ways depending on the nature of the concrete program (e.g., imperative versus pure-functional). Complexity analysis complicates this relationship: the above principle clearly does not suffice. It is therefore necessary to discuss what conclusions on the concrete program may be drawn from bounded termination, or a reachability bound, for the abstraction.

Section 6 is dedicated to this discussion. Our choice is to keep this paper concentrated on the theory of \((\mathcal{MC}, \mathbb{Z})\)-CTS; therefore this discussion is quite informal. The support for our arguments here is not theorems and proofs, but the practical experience of researchers who, previous to this work, have already used a CTS abstraction for complexity analysis. We discuss how this abstraction has been done in [ADFG10a] and [AAGP08]. The fact that they used a richer constraint language has no consequence for this question.

Briefly, the simplest case is of an imperative program, without procedure calls. The CFG of the \((\mathcal{MC}, \mathbb{Z})\)-CTS is essentially the flow-chart of the program, and the length of the computation is related to time complexity.

Next, we consider programs with recursive functions. We argue that for such programs, bounded termination most naturally yields a bound on stack height. Depending on the program’s use of “heap space,” we may be able to conclude that it runs in polynomial space, or just deduce an exponential time bound.

The fact that our abstraction is coarser than the one used in the cited works is relevant to another concern: the loss of information due to abstraction. Section 6.4 discusses the impact of relaxing the \((\text{Aff}, \mathbb{Z})\)-CTS abstraction (or possibly even stronger ones) to \((\mathcal{MC}, \mathbb{Z})\)-CTS. Such relaxation, which may suffice for termination, does not always suffice for complexity analysis. An example can be seen in Figure 2: for termination, we could do with a simpler abstraction, eliminating all constraints involving the variable \(N\). But then we would not obtain a bounded-terminating CTS.

Our thesis is that, despite its relative simplicity, the monotonicity constraint abstraction stands a good chance of being effective in practice (when used judiciously). The ultimate test
would, of course, be the construction of an industrial-strength tool; this is far beyond the scope of our work, but existing related work (see the next section and Section 6.4) makes the prospects seem encouraging.

As an additional informal argument to the interest in this abstraction, we include in Section 7 a few additional examples, collected from previous papers on complexity analysis, that illustrate different loop behaviours which are still all captured by our model.

A comment in order is that practical cost-analysis tools typically generate explicit constants, for example, they would generate a bound such as $n^3 - n + 2$ rather than just $O(n^3)$. However, the real bound may possibly be $O(n^2)$, since no tightness is guaranteed. In contrast, we chose to relax the expression of the bound to a big-Oh one but we show that the precise degree is decidable. Our algorithms can provide explicit constants, but they will be definitely over-approximative. Bounds that have precise explicit constants may be computable, too. We leave this as a challenge for further research.

2 Related Work

There is a surprisingly large body of work related to the topics of this paper. Most pertinent is the work in program analysis, directed at obtaining symbolic, possibly asymptotic, complexity bounds for programs (in a high-level language or an intermediate language) under generic cost models (either unit cost or a more flexible, parametrized cost model). In this section, to put our work in context, we cite some of these works and indicate what approaches were employed. The first subsection is an overview and cites various approaches. The second one elaborates on the works most directly related to ours. There are many other works in this area which have been left out; a complete survey would be an article in itself.

2.1 Approaches in Complexity Analysis

Seminal works. Wgbreit [Weg75] presented the first, and very influential, system for automatically analysing a program’s complexity. His system analyzes first-order LISP programs; Broadly speaking, the system instruments a program to obtain a function that returns the desired complexity measure, and then attempts to simplify the program until a closed form for the function can be found. Possibly, the program becomes a set of recurrence equations for the complexity which have to be solved. Subsequent works along similar lines included [LM88, Ros89] and more recently [Ben01, Ben04] for functional programs and [DwL93, DLGHL94] for logic programs. The latter describe static analyses to deal with complications particular to the semantics of logic programs, where programs compute sets of answers and involve backtracking.

Studies of restricted languages. Our approach in this paper involves the study of complexity properties of a simplified, abstract program. Research in Implicit Computational Complexity (ICC) has produced numerous examples of programming languages that are so restricted that they capture an intended complexity class, that is, compute all, and only, functions of that class. Early examples include [Cob64, KA80, BC92]. Many of these restrictions (e.g., [Cob64, BC92]) may be seen (or are even explicitly presented) as imposing a certain type system on a language which, otherwise, could also compute outside the intended complexity class; but this is not an automated analysis in the sense that the programmer has to supply the “types” (in [Cob64],
and also some later works like [CW00], these are explicit resource bounds). In these cases one might describe the technique more as certification than analysis. However, ICC research has also developed some methods that were later put to effective use in automated analysis. Two notable examples are the method of term interpretations (see the paragraph on Term Rewriting Systems below), and the method of linear types [Hof03], which yielded strong analysis techniques as described, e.g., in [HH10, JHLH10].

SPEED is an ambitious project from Microsoft Research to create a complexity analysis tool using a variety of techniques, focusing on C programs [GG08, GMC09, GJK09, Gul09, GZ10]. In [GG08, GMC09], the essence of the technique is to instrument the program with a counter, so that the desired resource usage becomes an output value, and bound this value using invariant-generation methods. In [GJK09], the techniques are program transformation (called control-flow refinement) and “progress invariants,” which are used for obtaining more precise bounds for nested loops. In [GZ10], the term reachability bound was coined, which we also employ in this paper.

Abstract interpretation techniques. While abstract interpretation [Cou96] is the de-facto standard way of presenting many program analyses, in the realm of complexity analysis its role has mostly been confined to supporting analyses (finding the ranges of values etc). As mentioned above, complexity analysis is sometimes reduced to computing a bound on computed values, and this is done by the traditional kind of abstract interpretation (invariant generation). However, there are a few works where abstract interpretations have been developed that directly result in complexity properties. In [MPS10] it was done for space complexity of a functional language. In [NW06, JK09], simple imperative programming languages have been analysed for complexity; interestingly, because of the background in ICC rather than in static analysis, the terminology of abstract interpretation is not used. These works were followed by [BJK08, BA10a] were it was shown that for languages of a similar style (imperative structure, very restricted in the usage of data, and non-deterministic in control flow except for bounded loops), an abstract-interpretation based analysis is actually a decision procedure: for example, one can decide whether a program is polynomial-time. In this paper, we are also interested in abstract programs whose properties of interest are decidable. However, the nature of the abstract programs is very different.

Term Rewriting Systems are an elementary computational model that may be used to represent programs from a variety of source languages. There is already much work on complexity analysis for TRSs. We mention two of the directions taken. [HM08b, HM08a, AM09, NEG11] employ the dependency pair method, which like the model we are studying, was originally conceived for termination, and in fact has been effectively combined with size-change termination [TG05, GTSKF06, CFGSk10].

Another method that has extended its scope from proving termination to proving complexity bounds in the context of Term Rewriting Systems is the polynomial interpretation method [BCMT01], later extended to other kinds of interpretation functions [MSW08, MP09, BD10, NZM10, Wal10]. The method has some resemblance to the analysis of transition systems with ranking functions, since the value of an interpretation has to decrease as computation progresses, but interpretations have a particular structure which is related to the structure of the terms in the system. Different interpretation methods have very different structures and it is beyond the
scope of this work to survey this line of work in greater detail. It should be pointed out that, basically, interpretations are proof methods and it is not always clear how to turn them into automatic analyses (in other words: how to *synthesize* suitable interpretations), but this issue is discussed in the literature, for example in [Ama05, MSW08] and many others.

2.2 Analysis of Constraint Transition Systems.

We have already described [ADFG10a], where \((\text{Aff}, \mathbb{Z})\)-CTS was used as an abstract program and analysed using lexicographic linear ranking functions.

The COSTA project [AAGP10, AAG+12] targets symbolic analysis of Java bytecode programs. It is a big project, in which involved methods of abstracting the concrete programs were implemented, but this is unrelated to our topic. Our interest begins where they reach an abstract program representation, which they call CRS (for *cost relation system*). An example of a CRS (liberally modified from [AAGP10]) is:

\[
\begin{align*}
E(a, j) &= k_1 + E(a', j') + F(a, j, j', a') \\
F(a, j, j', a') &= k_2 + E(a, j + 1)
\end{align*}
\]

where \(k_1, k_2\) represents costs (and can be non-constant expressions depending on the variables); essentially, this can be understood as a non-deterministic sort of recursive program whose result is the desired cost bound. As a central part in the algorithm to bound this result, the system is simplified to eliminate indirect recursion (which is not possible for all systems, but is argued to work well in practice) and then the height of the recursion tree is bounded by looking at individual (multiple-path) loops, e.g., all the “calls” from \(E\) to \(E\), and finding a linear ranking function for each such loop. In a structured program with nested loops, each loop will turn into this kind of a recursive cost relation and will therefore have to be bounded using a linear ranking function. This implies that a global ranking function of the lexicographic linear kind exists, but the technique is more restricted than [ADFG10a] which finds a lexicographic linear ranking function by analysing the transition system globally (that is, the lexicographic structure does not have to follow the loop nesting).

In comparison to our work, it is important to note that affine relations are expressive enough to make their termination problem undecidable (the simple argument is that counter machines can be represented). Thus, a complete solution cannot be achieved. One could try to relate our works by considering \((\text{MC}, \mathbb{Z})\)-CTS as a special case of \((\text{Aff}, \mathbb{Z})\)-CTS; if we do so, we find that their solutions do not encompass ours as a special case. Indeed, not every \((\text{MC}, \mathbb{Z})\)-CTS which is bounded terminating has a lexicographic linear ranking function (not even systems with a single program point). This is probably well known but will also be demonstrated by an example in Section 7.

Monotonicity constraint transition systems. As mentioned earlier, monotonicity constraint transition systems have been first used (with different terminology) for termination analysis of logic programs [Sag91, LS97, CT99]. In addition to this successful application, they have also been applied in the termination analysis of functional programs [LJBA01, MV06a, Kra07, SJ05] and imperative programs [SMP10, Ave06, CGBA+11]. Some works on the theory of \((\text{MC}, \mathbb{Z})\)-CTS and their decision problems are [CLS05, MT09, BA09, BA10b, BA11]; decision procedures for extensions of the model have been discussed in [BA08, BP12].
While this paper was in preparation, we learnt of the work of Zuleger et al. [ZGSV11], who also applied $(\mathcal{MC}, \mathbb{Z})$-CTS (more specifically, size-change graphs) in the context of cost analysis. The cited conference paper does not provide all details, however even a superficial look confirms that their use of the abstraction is essentially different from our work, since they do not employ an “abstract and conquer” approach where an abstract program becomes an object in itself. Instead, the abstraction is just one tool in a complex algorithm that processes source programs.

3 Preliminaries

The results in this paper build on previous research on the termination problem of $(\mathcal{MC}, \mathbb{Z})$-CTS. To make the paper self-contained, we repeat in this section the basic definitions and certain results from previous work. Readers familiar with [BA11] will find little news here but should read Section 3.4.

3.1 Monotonicity Constraint Systems and their semantics

**Definition 3.1.** A $(\mathcal{MC}, \mathbb{Z})$-CTS consists of a control-flow graph (CFG), monotonicity constraints and state invariants, all defined below.

- A control-flow graph is a directed graph (allowing parallel arcs) over the set $F$ of flow points. Every flow-point $f$ is associated with a fixed list of variables\(^2\) The number of variables is called the arity of $f$ and may be denoted by $\text{ar}(f)$; the variables themselves are usually denoted in the text by $x_1, \ldots, x_{\text{ar}(f)}$, though in examples we may use other identifiers, most naturally the names of variables of the source program.

- A non-empty set of flow points, $F_{\text{init}} \in F$, is designated as the initial flow points of the CFG.

- Every CFG arc $f \to g$ is associated with a monotonicity constraint (MC), being a conjunction of order constraints $x \succ y$ where $x, y \in \{x_1, \ldots, x_{\text{ar}(f)}, x'_1, \ldots, x'_{\text{ar}(g)}\}$, and $\succ$ is either $>$ or $\geq$; for uniform notation, we also use $\succ 0$ for $\geq$ and $\succ -1$ for $>$. Note that $<, \leq, =$ can be used as syntactic sugar.

We write $G : f \to g$ to indicate the association of an MC $G$ with its source and target flow-points.

A calligraphic-style letter (typically $\mathcal{A}$, for abstract program) is used to denote a $(\mathcal{MC}, \mathbb{Z})$-CTS. $F^\mathcal{A}$ ($F^\mathcal{A}_{\text{init}}$) will be its flow-point (initial flow-point) set. A monotonicity constraint will often be denoted by $G$ because it is typically represented by a graph (as explained below). However, when graph-theoretic notions are applied to $\mathcal{A}$ (such as, “$\mathcal{A}$ is strongly connected”), they concern the underlying CFG. In the text, a $(\mathcal{MC}, \mathbb{Z})$-CTS may be succinctly referred to as “a system” when the meaning should be clear.

\(^2\)Called parameters or arguments in some publications—depending on the programming paradigm the authors has in mind. Similarly, flow points may be called program points or locations.
State Invariants. Our representation of a \((\mathcal{MC}, \mathcal{Z})\)-CTS also includes, for each \(f \in F\), an invariant \(I_f\), which is a conjunction of order constraints among the variables. An example is \((x_1 > x_2) \land (x_3 = x_4)\). It is assumed that these constraints are also included in the MCs entering or leaving \(f\) (note that for an MC entering \(f\), the variables will be primed, as they belong to the target state). This assumption implies that the invariants are only a convenience, a way to indicate that some constraints will hold whenever \(f\) is visited, irrespective of which of the incoming and outgoing transitions are taken. The reader will see later that our algorithms make significant use of this information.

Semantics. Semantically, a \((\mathcal{MC}, \mathcal{Z})\)-CTS represents a transition relation over a set of (abstract) program states. In a state, every variable has a specific value. In this paper, all values are integers (as in [BA11] and unlike [BA10b, LJBA01, etc.], which dealt with well-founded sets).

**Definition 3.2 (states).** A state of \(\mathcal{A}\) is \(s = (f, \sigma)\), where \(f \in F^A\) and \(\sigma : \{1, \ldots, n\} \to \mathbb{Z}\) represents an assignment of values to the variables, where \(n = \text{ar}(f)\). The state is initial if \(f \in F^A_{\text{INIT}}\).

Satisfaction of a predicate \(e\) with free variables \(x_1, \ldots, x_n\) (for example, \(x_1 > x_2\)) by an assignment \(\sigma\) is defined in the natural way, and expressed by \(\sigma \models e\). If \(e\) is a predicate involving the \(n + n'\) variables \(x_1, \ldots, x_n, x'_1, \ldots, x'_{n'}\), we write \(\sigma, \sigma' \models e\) when \(e\) is satisfied by setting the unprimed variables according to \(\sigma\) and the primed ones according to \(\sigma'\).

**Definition 3.3 (transitions).** A transition is a pair of states, a source state \(s\) and a target state \(s'\). For \(G : f \rightarrow g \in \mathcal{A}\), we write \((f, \sigma), (g, \sigma') \models G\) if \(\sigma, \sigma' \models G\).

Note that we may have unsatisfiable MCs, such as \(x_1 > x_2 \land x_2 > x_1\); our algorithms will identify such MCs and ignore them.

**Definition 3.4 (transition system).** The transition system associated with \(\mathcal{A}\) is the binary relation

\[
T_A = \{(s, s') \mid s, s' \models G \text{ for some } G \in \mathcal{A}\}.
\]

Note that some authors refer to a program representation as a “transition system.” We use this term for a semantic object. Our view of a \((\mathcal{MC}, \mathcal{Z})\)-CTS is declarative: a set of constraints that describe the transition system \(T_A\). It is also possible to interpret a \((\mathcal{MC}, \mathcal{Z})\)-CTS operationally, as a kind of program. Every MC, \(G : f \rightarrow g\), then represents a step that the program may take when in program location (label) \(f\). The step consists of non-deterministically choosing values for the primed variables such that \(G\) is satisfied by the current state plus the chosen new values. The new values are then assigned to the variables, and the program location changed to \(g\). While we hope that this view may be useful to some readers, our formal development will use the declarative viewpoint.

**Definition 3.5 (run, height).** A run of \(T_A\) is a (finite or infinite) sequence of states \(\bar{s} = s_0, s_1, s_2, \ldots\) such that for all \(i > 0\) (up to the end of the sequence), \((s_{i-1}, s_i) \in T_A\). For a finite run \(s_0, s_1, s_2, \ldots, s_\ell\) we refer to \(\ell\) as its length. The height of a state is the length of the longest run beginning at the state.
Note that by the definition of \( T_A \), a run is associated with a sequence of CFG arcs labeled by \( G_1, G_2, \ldots \) where \( s_{i-1}, s_i \models G_i \). This sequence constitutes a (possibly non-simple) path in the CFG. As a slight abuse of definition, we may associate the run with \( A \) rather than explicitly mentioning \( T_A \).

**Definition 3.6 (termination).** A transition system is **terminating** if it has no infinite run from an initial state. A \((MC, Z)\)-CTS \( A \) is **terminating** if \( T_A \) is terminating.

This notion of termination was called **rooted termination** in [BA11], which also considered uniform termination—where reachability from an initial state is not taken into account. In the context of work on bounded termination, rooted termination is essential, and therefore the unqualified term will refer, in this paper, to rooted termination.

**Definition 3.7 (bounded termination).** A transition system satisfies **bounded termination** if it is terminating and the height of every initial state is finite. We say that a \((MC, Z)\)-CTS \( A \) satisfies bounded termination if \( T_A \) does (we also say that \( A \) is **bounded-terminating**).

Ben-Amram [BA11] proved that \((MC, Z)\)-CTS termination is decidable, and, more precisely, PSPACE-complete. We shall prove the same for bounded termination. It is important to note that a terminating program is not necessarily bounded-terminating, as in the next example.

**Example 3.1.** A classic example of termination analysis is the Ackermann function, here in pure-functional style:

\[
\text{ack}(m, n) = \begin{cases} 
\text{if } m \leq 0 \text{ then } n + 1 \\ 
\text{else if } n \leq 0 \text{ then } \text{ack}(m-1, 1) \\ 
\text{else } \text{ack}(m-1, \text{ack}(m, n-1))
\end{cases}
\]

The straightforward abstraction to a \((MC, Z)\)-CTS, has a single-node control-flow graph (the node represents the function \text{ack}), with three self-loops representing the recursive calls (here in the order of the call sites in the program text):

\[
\begin{align*}
& m > 0 \land n \leq 0 \land m > m' \land n' > 0' \land 0 = 0' \quad (1) \\
& m > 0 \land n > 0 \land m > m' \land 0 = 0' \quad (2) \\
& m > 0 \land n > 0 \land m = m' \land n > n' \land 0 = 0' \quad (3)
\end{align*}
\]

Note the constraints \( 0 = 0' \); these are included since in our constraint language there is no notion of constant. Technically, \( 0 \) is a state variable, hence the need for explicitly stating that it is constant\(^3\). The need for constraints like that also arises because of the “frame problem” (as it is called in Artificial Intelligence), that is, the need to state explicitly that variables not affected by a transition do not lose their value. In order to make the writing of these constraints more concise, we use the notation \text{Same}(x, y, \ldots) for \( x = x' \land y = y' \land \ldots \) (as in Figure 2).

Returning to the Ackermann example, it is easy to verify that this constraint transition system terminates; in fact, it has the global ranking function \((m, n)\). However, it is not bounded-terminating. Indeed, for any (arbitrary large) number \( N \), it has a transition sequence of length \( N + 1 \) from the initial state \((1, 2)\):

\[(2, 1) \rightarrow (1, N) \rightarrow (1, N - 1) \rightarrow \ldots \rightarrow (1, 0)\]

\(^3\)Constants can also be explicitly added to the constraint language, see [BP12].
The concrete program is, of course, bounded-terminating, because it is deterministic. Thus the length of the run is a function of the initial state. This information is lost because the abstraction is non-deterministic, and only super-approximates the semantics of the concrete program. To be more precise, it is the fact that we have unbounded non-determinism that causes the problem; if the abstraction had been non-deterministic, but finitely branching, by König’s lemma it would still be bounded-terminating.

3.2 MC graphs and multipaths

It is convenient for reasoning, and practical for algorithms, to represent MCs as directed graphs. These graphs have nodes \( x_1, \ldots, x_n, x'_1, \ldots, x'_n \) for the appropriate arities \( n, n' \) and represent each relation \( x \succ y \) by an arc; an arc representing a strict inequality is called a strict arc. A path in the graph is called strict if it includes at least one strict arc.

Standard graph algorithms can be used to perform operations such as path-finding and ensuring that the representation is closed under logical consequence, which is a simple reachability closure in the graph. In the process, we also identify (and remove) unsatisfiable MCs. Clearly, an MC is unsatisfiable if and only if there is a strict cycle.

Example 3.2. Figure 3 shows MCs extracted from the program below. The flow-points are \( w \) (entry to the \textbf{while} command) and \( i \) (entry to the \textbf{if} statement).

```plaintext
while (m<n)
    if (m>0) n := n-1
    else m := m+1
```

Some publications use the term MC graph (MCG); we, however, identify an MC with its graph representation. This should not cause any problems. We also use set notation, such as \((x > y) \in G\). We define the notation \((x, y) \in G\) to mean that \(x\) and \(y\) are related in \(G\) (without indicating the relation, which may be \(>, \geq, <\) or \(\leq\)). We employ the same notations with respect to state invariants, e.g., \((x > y) \in I_f\). Note that the above example does not have any state invariants (in the given abstraction), because the transitions that enter and exit each point do not agree on any relation among the state variables.

Notation. Whenever graphs are considered, the notation \(u \leadsto v\) means that there is a path from \(u\) to \(v\). The notation \(p : u \leadsto v\) names the path.
**Definition 3.8** (multipath). Let $A$ be a $(\mathcal{MC}, \mathbb{Z})$-CTS. Let $f_0, f_1, f_2, \cdots \in F^A$ be a (finite or infinite) list of flow-points connected by MCs $G_t : f_{t-1} \rightarrow f_t$ (clearly, this constitutes a path in the CFG). The *multipath* $M$ that corresponds to this path is a (finite or infinite) graph with nodes $x[t, i]$, where $t$ ranges from 0 up to the length of the path (which we also refer to as the length of $M$), and $1 \leq i \leq ar(f_t)$. Its arcs are obtained by merging the following sets: for all $t \geq 1$, $M$ includes the arcs of $G_t$, with source variable $x_i$ renamed to $x[t-1, i]$ and target variable $x'_j$ renamed to $x[t, j]$.

The multipath may be written concisely as $G_1G_2\ldots$; for example, Figure 4 illustrates a multipath $G_2G_1G_2$, based on the MCs from Figure 3. The term *multipath* (originating in [LJBA01]) hints at the multiple paths that may exist in the graph representation of $M$ (the importance of these paths is further discussed below). We use the expression $A$-multipath when it is necessary to name the CTS that $M$ is formed from.

If $M_1, M_2$ are finite multipaths, and $M_1$ corresponds to a CFG path that ends where $M_2$ begins, we denote by $M_1M_2$ the result of concatenating them in the obvious way. The notation $M : f \leadsto g$ indicates the initial and final flow-points of $M$.

Clearly, a multipath can be interpreted as a conjunction of constraints on a set of variables associated with its nodes. We consider assignments $\sigma$ to these variables, where the value assigned to $x[t, i]$ is denoted $\sigma[t, i]$.

A multipath may be seen as an execution trace of the abstract program, whereas a satisfying assignment constitutes a (concrete) run of $T_A$. Conversely: every run of $T_A$ constitutes a satisfying assignment to the corresponding multipath. Multipaths that start at an initial flow-point are called *rooted*. Termination can thus be expressed as non-existence of satisfiable, rooted infinite multipaths.

As for single MC graphs, we have

**OBSERVATION 3.9.** A finite multipath is satisfiable if and only if it does not contain a strict cycle.

We next consider down-paths and up-paths. The definition of a down-path is just the standard definition of a graph path, but it is renamed in order to accommodate the notion of an up-path.

**Definition 3.10.** A *down-path* in a graph is a sequence $(v_0, e_1, v_1, e_2, v_2, \ldots)$ where for all $i, e_i$ is an arc from $v_{i-1}$ to $v_i$ (in the absence of parallel arcs, it suffices to list the nodes). An *up-path* is a sequence $(v_0, e_1, v_1, e_2, v_2, \ldots)$ where for all $i, e_i$ is an arc from $v_i$ to $v_{i-1}$.

The term path may be used generically to mean either a down-path or an up-path (such usage should be clarified by context).
Semantically, in an MC or a multipath, a down-path represents a descending chain of values, whereas an up-path represents an ascending chain. Note also that an up-path listed backwards is a down-path in the transposed graph.

**Definition 3.11.** Let \( M = G_1 G_2 \ldots \) be a multipath. A **down-thread** in \( M \) is a down-path that only includes arcs of the form \((x[t, i] \rightarrow x[t+1, j])\).

An **up-thread** in \( M \) is an up-path that only includes arcs of the form \((x[t, i] \leftarrow x[t+1, j])\).

A **thread** is either.

**Definition 3.12** (cyclic). We say that a transition, or a multipath, is **cyclic** if its source and target flow-points are equal.

The next lemma and the following definitions, all from [BA11], are only used in Section 5.

**Lemma 3.13.** If a strongly connected \((\mathcal{MC}, \mathcal{Z})\)-CTS satisfies SCT, every finite multipath includes a strict, complete thread.

**Definition 3.14** (composition). The **composition** of \( MC G_1 : f \rightarrow g \) with \( G_2 : g \rightarrow h \), written \( G_1 ; G_2 \), is a \( MC \) with source \( f \) and target \( h \), which includes all the constraints among \( s, s' \) implied by \( \exists s'' : s, s'' \models G_1 \land s'', s' \models G_2 \).

**Definition 3.15** (collapse). For a finite multipath \( M = G_1 \ldots G_\ell \), Let \( \overline{M} = G_1 ; \cdots ; G_\ell \). This is called the **collapse** of \( M \).

**Definition 3.16** (reachability). A flow-point \( f \in \mathcal{F}_A \) is **reachable** if there is a satisfiable finite multipath \( M : f_0 \leadsto f \) such that \( f_0 \) is initial.

**Definition 3.17.** Given a \((\mathcal{MC}, \mathcal{Z})\)-CTS \( A \), its closure set \( cl(A) \) is

\[ \{ \overline{M} \mid M \text{ is a satisfiable finite } A\text{-multipath starting at a reachable flow-point} \} \]

**3.3 Stability**

**Definition 3.18** (stability). A \((\mathcal{MC}, \mathcal{Z})\)-CTS \( A \) is **stable** if (1) all MCs in \( A \) are satisfiable; (2) in the CFG of \( A \), all flow-points are reachable from an initial flow-point; (3) to every \( f \in \mathcal{F}_A \) is associated an invariant \( I_f \) such that for all \( G : f \rightarrow g \) in \( A \), \((x_i \succ x_j) \in G \iff (x_i \succ x_j) \in I_f \); similarly, \((x'_i \succ x'_j) \in G \iff (x_i \succ x_j) \in I_f \).

**Lemma 3.19.** [BA10b] Suppose that \((\mathcal{MC}, \mathcal{Z})\)-CTS \( A \) is stable. Then every finite multipath is satisfiable.

Note that in stable systems the flow-point invariants play an essential role since they are supposed to contain all the information that can be deduced from the adjacent MCs. The process of stabilizing a \((\mathcal{MC}, \mathcal{Z})\)-CTS involves splitting flow-points in the CFG whose original invariants were not precise enough. Algorithms for stabilization are described in [BA10b]. Such an algorithm transforms a \((\mathcal{MC}, \mathcal{Z})\)-CTS \( A \) into an equivalent stable system, which we denote by \( S(A) \) (“equivalent” means that they have the same runs, up to renaming of flow-points or possibly variables). We say that \( S(A) \) is a refinement of \( A \), since it explicitly separates states that in \( A \) are not explicitly separated.
Figure 5 shows how the CFG of Example 3.2 (which originally had two nodes) is transformed by stabilization. The \( w \) node has been split in four and the \( i \) node in two. There are also several CFG arcs that represent the same original transition, for example \( G_1 \) appears twice. The MCs annotating these arcs will not be identical to \( G_1 \), since the source and target invariants are merged into each MC. Note also that there are now several initial flow-points (namely all the nodes labeled \( w \)).

In the worst case, such a transformation can multiply the size of the system by a factor exponential in the number of variables \( n \) (bounded by the Ordered Bell Number \( B_n \) which is between \( n! \) and \( 2^{n-1}n! \) [Slo, Seq. A670]).

### 3.4 Full elaboration

Full elaboration [BA10b] may be seen as a brute-force way of obtaining a stable version of a program. The key observation is that for a finite number of variables, there are only finitely many orderings of their values. It is thus possible to exhaustively list all possibilities and create an explicit representation of how transitions will affect each one. This transformation is useful for proving that certain problems are decidable in polynomial space, since one does not actually need to list all possibilities; they can be created “on the fly.” This should become clear from the explanations below.

In this paper we make subtle adjustments to the presentation of full elaboration in [BA10b], which are important for its use in Section 5.

**Definition 3.20** (full elaboration). A \((\mathcal{MC}, \mathbb{Z})\)-CTS \( \mathcal{A} \) is fully elaborated if the following conditions hold:

1. Each state invariant fully specifies the relations among all variables, so that they are in ascending order by index. Namely, for all \( i < j \leq ar(f) \), \( I_f \) includes \( x_i < x_j \).

2. Each MC is satisfiable.

3. In the CFG of \( \mathcal{A} \), all flow-points are reachable from an initial flow-point.

Indexing the variables in sorted order has some convenient consequences.

**Lemma 3.21.** In a fully-elaborated system, every MC, \( G \), has the downward closure property: for all \( k < j \), \((x_i \succ^b x_j') \in G\) entails \((x_i \succ^d x_k') \in G\), where \( b \geq d \) (that is, the latter relation is at least as strict).
Since equalities are not allowed in the state invariants, creating a fully-elaborated version of a given \((\mathcal{MC}, \mathbb{Z})\)-CTS may require the \textit{coalescing} of variables related by an equality constraint. Otherwise, it is a process similar to stabilization, as described previously.

\textbf{Definition 3.22.} For a \((\mathcal{MC}, \mathbb{Z})\)-CTS \(\mathcal{A}\), its full elaboration \(E(\mathcal{A})\) is a fully-elaborated system that simulates \(\mathcal{A}\) in the following sense (see also example below):

- A flow-point of \(E(\mathcal{A})\) is specified by a pair \((f, \psi)\) where \(f\) is an \(\mathcal{A}\) flow-point and \(\psi_f : [1, \text{ar}(f)] \rightarrow [1, k]\) maps variables of \(f\) to variables of \((f, \psi)\), whose arity \(k\) lies between 1 and \(\text{ar}(f)\). This mapping \(\psi\) is required to be surjective, but is not necessarily injective; thus, two \(\mathcal{A}\) variables may be coalesced in a corresponding flow-point of \(E(\mathcal{A})\). The variable mapping, and the invariant of \((f, \psi)\), are consistent with the invariants of \(f\) (in particular, variables are coalesced if only if they are constrained by an equality).

- Given two such points \((f, \psi)\) and \((g, \psi')\), and an MC \(G : f \rightarrow g\) from \(\mathcal{A}\), there is a corresponding MC \(G' : (f, \psi) \rightarrow (g, \psi')\) which is obtained by renaming the variables in \(G\) according to \(\psi, \psi'\) and adding the elaborated flow-point invariants, provided that the result is satisfiable.

- An initial point of \(E(\mathcal{A})\) is any point \((f, \psi)\) such that \(f\) is initial in \(\mathcal{A}\).

- \(E(\mathcal{A})\) contains all possible combinations \((f, \psi)\) as above, that are reachable from an initial point.

\textit{Example 3.3.} Consider the system of Example 3.2. One of the initial flow-points of the elaborated system will be \((w, [0 \mapsto 1, m \mapsto 2, n \mapsto 3])\) (we are using identifiers for the variables at \(w\), for legibility). This flow-point represents all initial states in which \(0 < m < n\). A transition, corresponding to \(G_1\), takes this flow-point to a flow-point corresponding to \(i\) and having the same variable mapping, since the order among the values is unchanged.

Consider now the elaborated flow-point \((w, [0 \mapsto 1, m \mapsto 1, n \mapsto 2])\). It represents states in which \(0 = m < n\). A transition, corresponding to \(G_1\), takes this flow-point to a flow-point corresponding to \(i\) and having the same variable mapping. From this flow-point, there will be an out-going transition representing \(G_3\), but no one representing \(G_2\), since that would be unsatisfiable. (End of example)

Note that given two elaborated flow-points \((f, \psi)\) and \((g, \psi')\), and an MC \(G : f \rightarrow g\) from \(\mathcal{A}\), a simple polynomial-time procedure computes the elaborated transition \(G_E : (f, \psi) \rightarrow (g, \psi')\), or possibly rejects it as unsatisfiable, assuming (and not verifying) that \((f, \psi)\) is itself reachable in \(E(\mathcal{A})\). This is why it is not necessary, in the algorithms where we shall use full elaboration, to pre-compute a full representation of \(E(\mathcal{A})\) and keep it in memory.

When presenting an algorithm that processes a fully-elaborated transition system, we will not use the notation \((f, \psi)\) for flow-points since we do not care about the correspondence with the original system. We will use simple identifiers instead.

\section{The Bounded Termination Problem}

This section gives our first theoretical result: decidability and complexity of the bounded termination problem, and the corollary that height bounds are polynomial.
4.1 Discovering Bounded Variables

To establish bounds on transition-sequence length we need bounds on the values of variables throughout the execution, in terms of the initial values. So, we are looking for invariants of the kind \( x_i \leq x_j \) where \( x_j \) is the initial value of \( x_j \). The inequality relates values at two different points in execution, not a property of a state, which can be captured by a state invariant. This apparent difficulty is easily solved by instrumenting the program. Specifically, we make a copy of the initial variables. The copies are never modified but carried over to every subsequent state and turn the relationship of current values to initial values into a property of states. In this paper we will, for simplicity, create only two such variables: \( x_{\text{max}} \) to represent the maximum among initial values, and \( x_{\text{min}} \) to represent the minimum. This will allow us to determine whether a subsequently-computed value is upper-bounded by at least one initial value (which is the same as being bounded by \( x_{\text{max}} \)) or lower-bounded by at least one initial value (same as lower-bounded by \( x_{\text{min}} \)). Note that this instrumentation is part of the algorithm whose input is the constraint transition system; we do not deal with concrete programs. We find it more legible to avoid using numeric indices for these variables, though technically they will just be \( x_{n+1} \) and \( x_{n+2} \) where \( n \) is the original arity.

**Definition 4.1.** For a given \( (\mathcal{MC}, Z)\)-CTS \( A \), the instrumented version \( I(A) \) is obtained by the following steps.

1. Add two new variables \( x_{\text{max}}, x_{\text{min}} \) to every flow-point.
2. Add a new initial point \( f_0 \) with an invariant \( I_{f_0} \) that expresses the intended relationship of \( x_{\text{max}} \) and \( x_{\text{min}} \) to the initial value of \( x_j \) for \( 1 \leq j \leq \text{ar}(f_0) \), namely \( x_{\text{max}} \geq x_j \) and \( x_{\text{min}} \leq x_j \).
3. Add a transition from \( f_0 \) to each of the original initial points, with constraints \( x_i = x'_i \), for all \( i \), in addition to constraints inherited from \( I_{f_0} \).
4. Add constraints \( x_{\text{max}} = x'_{\text{max}} \) and \( x_{\text{min}} = x'_{\text{min}} \) to all transitions.

The reader may note that the constraints cannot express that \( x_{\text{max}} \) is precisely the maximum among initial values—but the effect is the same. If for some flow-point we can deduce the invariant \( x_{\text{max}} \geq x_j \), then \( x_j \) must be bounded by one of the initial values—since \( x_j \) is related to \( x_{\text{max}} \) only by paths passing through \( x_1, \ldots, x_{\text{ar}(f_0)} \) at \( f_0 \). More formally:

**Lemma 4.2.** Let \( M \) be a rooted multipath of \( I(A) \). Suppose that \( M \) is satisfiable. Then there is a satisfying assignment for \( M \) such that \( \sigma[0, \text{max}] \) (the assignment of \( x_{\text{max}} \) in the initial flow-point) is exactly \( \max_{1 \leq i \leq n} \sigma[0, i] \); and \( \sigma[0, \text{min}] \) is exactly \( \min_{1 \leq i \leq n} \sigma[0, i] \).

Such an assignment will be called tight.

The next step will be to compute the stable program \( S(I(A)) \). Then we proceed to identifying bounded variables. To see why stabilization is necessary, consider the program in Figure 6, shown together with its control-flow graph. The notation \( x : = \ast \) represents the assignment of a value unrelated to the program inputs.

It is easy to see that at point \( \mathcal{W} \), there is no invariant that bounds one of the variables (or both) in terms of the input values. The closest we might come is to establish a disjunctive invariant of the form "either the value of \( x \) or the value of \( y \) is bounded by the input \( b \)," but such an invariant is not useful for our approach, as will be seen below. Stabilization solves this problem: in a stable system, each of the possible cases (\( x \leq b \) and \( y \leq b \)) is represented by a distinct flow-point.
input a, b  
if (*) then x := b; y := *  
else y := b; x := *  
while ( x>a and y>a )  
  x := x-1; y := y-1

Figure 6: A program and its CFG (with flow-points for the If, the Then and Else branches, and the While header). Which variable is bounded at W?

Definition 4.3. For every flow-point $f$ of $S(I(A))$, let

$$B(f) = \{ j \mid (x_j \succ^b x_{\text{min}}), (x_{\text{max}} \succ^d x_j) \in I_f \text{ for some } b,d \}.$$  

We call $B(f)$ the set of bounded variables at $f$.

4.2 Deciding Bounded Termination

We next provide a decision algorithm for bounded termination. The idea, in a nutshell: ignore all non-bounded variables, and check for termination. The example given earlier illustrates that the role of stability in the definition of $B$ is crucial for the correctness of this algorithm.

If $M$ is a multipath, a $B$-restricted assignment for $M$ is an assignment $\sigma$ to the variables of $M$ that assigns integers to the bounded variables and the special value $\perp$ to all others. This value satisfies any constraint (including $\perp > \perp$) which means that the non-bounded variables do not influence the satisfaction of constraints. If such an assignment exists, $M$ is called $B$-satisfiable.

Definition 4.4. We say that $S(I(A))$ $B$-terminates if there is no infinite, rooted multipath $M$ which is $B$-satisfiable.

Algorithm 4.1. (Bounded Termination) Input: $(M^C, Z)$-CTS $A$.

1. Build $S(I(A))$.
2. Perform a decision procedure for termination on $S(I(A))$, taking only bounded variables into account.
3. Return the result of the termination procedure.

Clearly, the algorithm checks for $B$-termination of $S(I(A))$. We next prove that this is a sufficient and necessary condition for bounded termination.

**Theorem 4.5** (extended soundness). If $S(I(A))$ $B$-terminates, then $A$ bounded-terminates. Moreover, let $\max x^i$ and $\min x^i$ the maximum and minimum values among the variables of the initial state. The height of the initial state is $O((\max x^i - \min x^i)^n)$, where the constant factor depends on the size of $S(I(A))$, and $n$ is the maximum of the flow-point arities in $A$.  

19
Proof. The fact that $A$ is terminating is obvious—$B$ termination is a weaker notion than termination. To justify bounded termination, consider any rooted $S(I(A))$-multipath, and a satisfying, tight $B$-restricted assignment. All bounded variables will be assigned values between $\min x^i$ and $\max x^i$. The variables $x_{\max}$ and $x_{\min}$ are constant throughout. Thus there are at most $m(\max x^i - \min x^i)^n$ different states that can potentially appear in a satisfying assignment to this multipath, where $m$ is the number of flow-points in $S(I(A))$. There can be no repeated states, since otherwise one can use an obvious “cut and paste” argument and exhibit an infinite sequence of $L$’s) is satisfiable.  

LEMMA 4.6. If $A$ is a non-terminating $(\mathcal{M},\mathcal{C},\mathcal{Z})$-CTS with initial point $f_0$, there is a cyclic multipath $L : f \leadsto f$, and a rooted multipath $H : f_0 \leadsto f$, such that $HL^\infty$ (H followed by an infinite sequence of $L$’s) is satisfiable.  

And we add a new lemma.

LEMMA 4.7. Let $M$ be a finite multipath in $S(I(A))$ and $\sigma$ a $B$-restricted assignment for $M$. It is possible to extend $\sigma$ to an assignment $\sigma'$ that satisfies $M$.

Proof. We treat $M$ as a directed graph, with arcs weighted by 0 for a non-strict arc and $-1$ for a strict one. Since $M$ is satisfiable (Lemma 3.19), there is no negative-weight cycle; hence, for all nodes $u$ and $v$, if $v$ is reachable from $u$, there is a minimum-weight path from $u$ to $v$. We define the \textit{minimum-weight distance} $\delta(u,v)$ to be the weight of such a path, or $+\infty$ if $v$ is unreachable from $u$.

We define sets of nodes $U_0, D_1, U_1, D_2, U_2 \ldots$ as follows:

- $U_0$ consists of all nodes which represent \textit{bounded} variables.
- For all $i \geq 0$, let $P_{i+1}^D = U_0 \cup D_1 \cup \cdots \cup U_i$; then, $D_{i+1}$ is the set of nodes $v \notin P_{i+1}^D$ such that $u \leadsto v$ for some $u \in P_{i+1}^D$.
- For all $i \geq 1$, let $P_i^U = U_0 \cup D_1 \cup \cdots \cup D_i$; then, $U_i$ is the set of nodes $u \notin P_i^U$ such that $u \leadsto v$ for some $v \in P_i^U$.

We extend $\sigma$ from the nodes of $U_0$, on which it is initially defined, to nodes of every set $D_i$ and $U_i$, inductively. Suppose that $U_0, D_1, \ldots, U_i$ have already been treated; let $P_{i+1}^D$ be the union of these sets. Then for every $v \in D_{i+1}$ we extend $\sigma$ by letting

$$\sigma(v) = \min_{u \in P_{i+1}^D} \{ \sigma(u) + \delta(u,v) \}$$

Note that $\sigma(v)$ is finite since, by definition of $D_{i+1}$, there are nodes in $P_{i+1}^D$ such that $\delta(u,v)$ is finite, and they are already assigned.

\footnote{Actually, the corresponding lemma in [BA11] does not consider rooted termination and therefore neglects the stem $H$. But if the termination test is modified to test for rooted termination (so that only reachable cycles are considered), the lemma stated here ensues.}
Alternatingly, we extend $\sigma$ to $U_i$, assuming that all nodes in $P_i^{U_i} = U_0 \cup D_1 \cup \cdots \cup D_i$ have been assigned. For $u \in U_i$ we let

$$\sigma(u) = \max_{v \in P_i^U} \{ \sigma(v) - \delta(u, v) \}$$

As above, $\sigma(v)$ is well-defined and finite.

We claim that the assignments to $\sigma$ are consistent with the constraints in $M$. To prove this, consider an assignment to $v \in D_{i+1}$. There are three possible types of constraints involving $v$ and another assigned variable:

1. A constraint $v > b v'$ with both $v$ and $v'$ in $D_{i+1}$. Thus, both are reachable from $P_{i+1}^D$, and, by the definition of $\delta$, we have for all $u \in P_{i+1}^D$, $\delta(u, v) + b \geq \delta(u, v')$. In particular, choose $u_L \in P_{i+1}^D$ such that $\sigma(u_L) + \delta(u_L, v)$ is minimum (and, hence, this is the value assigned to $v$); then

$$\sigma(u_L) + \delta(u_L, v') \leq \sigma(u_L) + \delta(u_L, v) + b$$

so our definition of $\sigma(v)$ and $\sigma(v')$ satisfies

$$\sigma(v') \leq \sigma(v) + b$$

and the constraint is satisfied.

2. A constraint $v > b v'$ with $v$ in $D_{i+1}$ and $v'$ in $P_{i+1}^D$. Here the case $i = 0$ is special. So consider first $i > 0$. By examining the definition of the sets, the reader may verify that $P_{i+1}^D$ is closed under reverse-reachability. Hence, our assumptions imply $v \in P_{i+1}^D$, and $v \in D_{i+1}$ is impossible. Next, let $i = 0$; so $v$ is in $D_1$ and $v'$ in $U_0$ which is the set of bounded variables. By the definition of $D_1$, there is a $U_0$ variable which upper-bounds $v$, while $v'$ lower-bounds it, so $v$ too is a bounded variable and cannot be in $D_1$.

3. A constraint $v > b v$ with $v$ in $D_{i+1}$ and $v'$ in $P_{i+1}^D$. Clearly,

$$\min_{u \in P_{i+1}^D} \{ \sigma(u) + \delta(u, v) \} \leq \sigma(v') + b$$

so this constraint will be satisfied.

A similar case analysis justifies the assignments in $U_i$.

Finally, there may remain nodes not in any of the above sets. These nodes are not connected to any node already assigned. So an assignment may be chosen for them freely, only having to satisfy relations among themselves, which is possible since $M$ is satisfiable. □

**THEOREM 4.8** (Completeness). A $(MC, \mathbb{Z})$-CTS A bounded-terminates only if $S(I(A))$ $B$-terminates.

*Proof.* Suppose that $S(I(A))$ does not $B$-terminate. Our goal is to show that for a particular initial state, there is a run, starting at this state, of any length.

We use Lemma 4.6. It provides us with a cyclic multipath $L$ and a rooted multipath $H$, such that $HL^\omega$ is $B$-satisfiable, say be $\sigma$. For every $p \geq 0$, multipath $HL^p$ is $B$-satisfied by the corresponding part of $\sigma$. Since it is a finite multipath in a stable system, by Lemmas 3.19 and 4.7 we can extend $\sigma$ to a complete assignment $\sigma_p$ that satisfies $HL^p$.

Next, we note that all the variables of the initial point $f_0$ of $I(A)$ are clearly bounded, which means that $\sigma$ valuates them. So, all the assignments $\sigma_p$ agree on the initial state. This concludes the proof. □
Finally we consider the complexity of the decision problem.

**Theorem 4.9.** Deciding whether a \((\mathcal{MC}, \mathbb{Z})\)-CTS \(A\) bounded-terminates is PSPACE-complete (and is PSPACE-hard even for stable systems that have a single flow-point).

*Proof. Upper bound.* The algorithm as described calls for constructing \(S(I(A))\) and stabilization may, in general, increase the size of a \((\mathcal{MC}, \mathbb{Z})\)-CTS exponentially. However, as shown in [BA10b], it is possible to implement the decision procedure for termination (or, more precisely, for non-termination) as a non-deterministic PSPACE algorithm. The problem is then in PSPACE thanks to Savitch’s theorem. The trick is to use full elaboration (which yields a stable system).

Given the \((\mathcal{MC}, \mathbb{Z})\)-CTS program \(A\), our algorithm constructs flow-points and transitions of the elaborated system \(E = E(I(A))\) on the fly. First, it non-deterministically walks through \(E\) to find a reachable flow-point \(f\) that it guesses will start the loop (the part denoted above by \(L\)). From that point on, it maintains a summary of the multipath traversed, proceeding with the random walk through \(E\) until a counter-example to bounded termination (a cyclic multipath which is not B-terminating) has been found.

*Lower bound.* We reduce from the SCT problem [LJBA01], a simple case of \((\mathcal{MC}, \mathbb{Z})\)-CTS termination. In an SCT instance, the only type of constraints that appear in the input is \(x_i > x_j\) (a source variable bounds a target variable).

Let \(S\) be an SCT instance, with a single flow-point and with \(n\) variables. Add variables \(x_b\) (bottom) and \(x_t\) (top), and the constraints: \(x_t \geq x_i \geq x_b\), for every \(i\). To every MC add the constraints \(x_b = x_b'\) and \(x_t = x_t'\). We claim that the resulting system, \(A\), bounded-terminates if and only if \(S\) terminates. Indeed, it is obvious that we made all the variables of \(S\) bounded. So if \(S\) terminates, \(A\) satisfies the condition for bounded-termination. For the other direction, suppose that \(S\) does not terminate: In [CLS05] is is shown, that in such a case, there is a loop in \(T_S\). That is, there is a run which reaches a certain state \(s\) and then repeats forever a certain finite run from \(s\) to \(s\). Such an infinite run clearly refutes bounded termination.

This reduction proves the hardness result, since the SCT problem is PSPACE-hard even for instances with a single flow-point [BA09]. It is easy to verify that the \((\mathcal{MC}, \mathbb{Z})\)-CTS created is also stable.

\[\square\]

## 5 The Reachability-Bound Problem

This section presents our second theoretical result: decidability and complexity of the reachability-bound problem, defined more precisely below.

### 5.1 Problem Definition

The reachability-bound problem is defined by Gulwani and Zuleger [GZ10] as the problem of computing a worst-case symbolic bound on the number of visits to a given control location. They discuss practical motivations for evaluating bounds for specific flow-points, rather than a global bound on the length of runs. If the property of interest is the total length of a computation, it is possible to obtain a bound by computing the RB bound for selected cut points in the program (for a “big O” bound, it suffices to ensure that every cycle includes a cut point). Therefore, the RB problem subsumes the problem of computing height bounds (as posed in the previous section). The RB problem is more general, since different flow-points may have different bounds.
We refine the definition for our specific setting as follows.

**Definition 5.1.** Let $A$ be a $(\mathcal{MC}, \mathbb{Z})$-CTS, and $f \in F^A$. Consider the instrumented program $I(A)$ and let $f_0$ be its initial flow-point.

The *reachability bound* (RB) for $f$ is a function $T_f : \mathbb{N} \to \mathbb{N}$ such that $T_f(N)$ is the maximal number of occurrences of the flow-point $f$ in a run that begins with a state $(f_0, \sigma)$, that $\sigma(x_{\text{min}}) = 0$ and the value $\sigma(x_{\text{max}}) = N$. The *unrooted* reachability bound is defined analogously, however for runs beginning at $f$.

The *reachability bound problem* is to explicitly find a function $\beta$ such that $T_f(N)$ above is in $\Theta(\beta(N))$.

The decision problem $RBD$ (reachability-bound degree) is the set of tuples $(A, f, k)$ such that $T_f(N) \in \Omega(N^k)$. The unrooted problem $URB$ is defined analogously.

Remarks:

1. If the number of visits to $f$ is *unbounded*, the reachability bound is undefined. We point out below how boundedness can be verified, but otherwise we assume, throughout this section, that a bound is known to exist.

2. The choice of $(0, N)$ as the initial values for $(x_{\text{min}}, x_{\text{max}})$ is not restrictive; any initial state with $x_{\text{max}} - x_{\text{min}} = N$ would give rise to the same runs, up to a shift in the values, but the expression of the bound would be more cumbersome.

3. We shall show that the RBD problem is in PSPACE. This means that one can also search for the tightest degree in polynomial space. We will further prove that once the tight degree is found, one actually has $T_f$ up to a constant factor. Since $O$, $\Omega$ and $\Theta$ play an essential role in this section, let us recall the definitions: let $F, G : \mathbb{N} \to \mathbb{N}$.

\[
F \in O(G) \iff (\exists c > 0)(\exists d \geq 0)(\forall x) F(x) \leq cG(x) + d; \\
F \in \Omega(G) \iff (\exists c > 0)(\exists d \geq 0)(\forall x) F(x) \geq cG(x) + d; \\
\Theta(G) = O(G) \cap \Omega(G).
\]

We will commit the common abuse of language and write “terminate in $O(n)$ steps” instead of “terminate in $S(n)$ steps for $S \in O(n)$.”

For our analysis, it is convenient to pose the question in an inverted manner. Instead of asking for a bound in terms of $N$, we ask for a value of $N$ that gives rise to a certain number of visits.

**Definition 5.2.** Let $M$ be a multipath of $I(A)$. We say that it is $N$-satisfiable if it has a satisfying assignment with initial values $(0, N)$ in $(x_{\text{min}}, x_{\text{max}})$.

We formulate a simple lemma:

**Lemma 5.3.** Let $T_f(N)$ be the reachability bound for $f$. Function $\beta : \mathbb{N} \to \mathbb{N}$ satisfies $\beta \leq T_f$ if and only if every multipath that starts with $f_0$ and visits $f$ at most $\beta(N)$ times is $N$-satisfiable.

The words “at most” can be omitted in the lemma, since if multipath $M$ is $N$-satisfiable, then so is every prefix of $M$. Another important observation for our algorithms is contained in the following lemma:
LEMMA 5.4. A multipath $M$ of $I(A)$ is $N$-satisfiable if and only if it is satisfiable, and in the weighted-graph representation of $M$, every directed path from $x_{\text{max}}$ to $x_{\text{min}}$ has weight at least $(-N)$.

Proof. If $N$ is not satisfiable, it is clearly not $N$-satisfiable. If it is satisfiable, but there is a path from $x_{\text{max}}$ to $x_{\text{min}}$ of weight below $(-N)$, this implies $x_{\text{min}} < x_{\text{max}} - N$ and contradicts $N$-satisfiability. Thus, the condition given in the lemma is necessary.

To show sufficiency, let $M$ be a multipath satisfying both conditions. We construct an assignment $\sigma$ for $M$ by first defining $\sigma(x_{\text{min}}) = 0$ (in all occurrences of $x_{\text{min}}$) and $\sigma(x_{\text{max}}) = N$. Note that if any other variable $v$ is related to both $x_{\text{min}}$ and $x_{\text{max}}$, it must be lower-bounded by $x_{\text{min}}$ and upper-bounded by $x_{\text{max}}$. Let $B(M)$ be the set of such variables of $M$. For each $v \in B(M)$, define $\sigma(v) = N + w(v)$ where $w(v)$ is the lowest weight of a path from $x_{\text{max}}$ to $v$. Clearly, $\sigma(v)$ will always be a number between 0 and $N$, and it is also easy to see that it satisfies any constraint involving the variables of $B(M)$. Based on the fact that $M$ is satisfiable (and hence includes no negative-weight cycles), it is possible to extend the assignment consistently to the rest of the variables as done in the proof of Lemma 4.7.

5.2 Set-up for the algorithms

We present algorithms to decide $RBD$, and solve the RB problem, for the constraint transition system $E = E(I(A))$, obtained by fully elaborating the instrumented version of the input system $A$. Note that a solution to the RB problem for $E$ implies a solution for the original system (with due attention to the fact that several flow-points in $E$ represent a single one in $A$; one has, therefore, to sum their bounds).

LEMMA 5.5. In $E$, the reachability bound for $f$ differs from the unrooted reachability bound for $f$ by at most a constant factor.

Proof. Let $T_f$ be the RB for $f$, and let $T_f^u$ be the unrooted reachability bound. As $f$ is reachable, every multipath $M$ from $f_0$ can be extended to a multipath $M'$ from $f_0$ by the adjunction of a fixed finite prefix $H$, leading from $f_0$ to $f$. It follows from Lemma 5.4 and Lemma 3.19 that if $M$ is $N$-satisfiable, then $M'$ is $N + h$ satisfiable where $h$ is a constant depending on $H$. The number of occurrences of $f$ in $M$ and $M'$ is the same. Hence, $T_f(N + h) \geq T_f^u(N)$.

Conversely, every multipath $M$ from $f_0$ can be trimmed at the beginning so that it begins with $f$. If $M$ was $N$-satisfiable, it is also $N$-satisfiable after trimming. Hence, $T_f^u(N) \geq T_f(N)$. Using the fact that $T_f$ has polynomial growth, we conclude that $T_f^u \in \Theta(T_f)$. In fact, $T_f^u \in (1 + o(1))T_f$.

We have thus reduced the general problem to the unrooted one. Next, we develop a decision procedure for the URB problem. For our next steps, we assume that we are given $E$ and the degree $k$. Our goal is then the following: given $E$, $f$ and $k$, we wish to determine if the unrooted RB for $f$ in $E$ is in $\Omega(N^k)$. We consider $A$, $E$ and $f$ fixed up to Section 5.6.

5.3 Deciding URB

We describe a decision procedure for URB which is based on inspecting the cyclic transitions $G : f \rightarrow f$ that appear in the closure $cl(E)$. Thus, we reduce the problem to one that involves a degenerate CFG with a single flow-point. This is sound by the following easy lemma:
Lemma 5.6. Let $T_f^r(N)$ be the unrooted reachability bound for $f$. Let $cl(E)$ be the composition-closure of $E$; let $T_f^*(N)$ be the maximal length of a run of $cl(E)$ that uses only the flow-point $f$ and starts with a state where $(x_{min}, x_{max}) = (0, N)$. Then $T_f^*(N) \in \Theta(T_f^u(N))$.

The justification for the lemma is that the closure is finite, and every transition in the closure represents a fixed, finite sequence of transitions of $E$.

Since $E$ is stable, we can define the subset $B$ of bounded variables as in Section 4. Then, in looking for the reachability bound, we can ignore unbounded variables, as Lemma 4.7 shows that the satisfiability of finite multipaths is only affected by the bounded variables.

Definition 5.7. $cl(E) \mid f$ is the subset of $cl(E)$ that consists only of transitions $f \rightarrow f$, and where the unbounded variables are omitted.

We are thus going to analyze $cl(E) \mid f$. Note that $cl(E) \mid f$ itself is a fully-elaborated transition system, where we define $f$ to be initial. For convenience, we denote the bounded variables of $cl(E) \mid f$ by $x_1, \ldots, x_n$, in the order determined during full elaboration (thus $I_f$ is $x_1 < x_2 < \cdots < x_n$).

Note that, based on the previous section, the existence of a reachability bound for $f$ may be verified by simply checking $cl(E) \mid f$ for termination. We remark that since it is a fully-elaborated system with a single flow-point, the termination test is simpler than for a general ($\mathcal{MC}, \mathcal{Z}$)-CTS; it is actually polynomial-time [BA11].

Definition 5.8. Let $0 < L \leq n$. A level partition of depth $L$ and width $w$, denoted $\mathcal{LP}$, consists of a disjoint partition of $\{1, \ldots, n\}$ into intervals $VI_1, \ldots, VI_w$, where $VI_1 = \{1, \ldots, i_1\}$, $VI_2 = \{i_1 + 1, \ldots, i_2\}, \ldots, VI_w = \{i_{w-1} + 1, \ldots, n\}$, and every interval is associated with a level from 0 to $L$, so that every level, except possibly level 0, has at least one associated interval, and adjacent intervals have different levels.

In addition, every variable $x_i$ is associated with a direction $d_i \in \{-1, 1\}$.

Informally, we may view each interval $VI_j$ as a set of variables rather than indices of variables. We use $iv(i)$ for the index of the interval to which $x_i$ belongs. We use $level(i)$ for the level associated with the interval $iv(i)$.

For a binary relation $\triangleright$, and an integer exponent $d$ the notation $\triangleright^d$ has a standard meaning; in what follows we only use the cases $\triangleright^1 (= \triangleright)$ and $\triangleright^{-1}$ (the inverse of $\triangleright$).

Definition 5.9. Let $G \in cl(E) \mid f$. We say that $G$ is consistent with $\mathcal{LP}$ at level $h$, where $0 < h \leq L$, if for every $1 \leq i \leq n$ we have:

\[
\text{level}(i) > h \Rightarrow (x_i \geq x_i') \in G \\
\text{level}(i) = h \Rightarrow (x_i \geq x_i') \in G \lor (x_i \geq x_i') \in G \\
\text{level}(i) < h \Rightarrow (x_i, x_i') \notin G
\]

and $\exists i : level(i) = h \land (x_i > x_i') \in G$.

An $\mathcal{LP}$-consistent set of MCs is an indexed set $G = \{G_1, \ldots, G_L\}$ such that each $G_h$ is consistent with $\mathcal{LP}$ at level $h$.

The conditions listed in the definition may be verbally referred to by saying that $G$ is still at levels above $h$, active at level $h$, and disconnected at levels below $h$.

We can now state the essence of the decision procedure:
**Theorem 5.10.** The reachability bound for \( f \) in \( \text{cl}(E)|f \) is in \( \Omega(N^L) \) if and only if an \( \mathcal{LP} \)-consistent set of MCs exists, for some \( \mathcal{LP} \) of depth \( L \).

For intuition, observe that if such MCs exist then, for the subsystem of \( \text{cl}(E)|f \) that consists only of these MCs, we have a lexicographic ranking function \( \rho \) (with codomain \([0, N]_L\)) constructed as follows: let \( |x|_1 = x - x_{\text{min}} \) and \( |x|_+ = x_{\text{max}} - x \). Then

\[
\rho(x_1, \ldots, x_n) = (\rho_L(\bar{x}), \rho_{L-1}(\bar{x}), \ldots, \rho_1(\bar{x}))
\]

where

\[
\rho_h(\bar{x}) = \sum_{\text{level}(i)=h} |x_i|_d_i.
\]

Therefore, for this subsystem, an \( O(N^L) \) upper bound is clear. We still have to prove that it is tight by proving an \( \Omega(N^L) \) lower bound: this will justify the if part of the theorem. To establish the only if part, we later show how the existence of a level partition and a consistent set of MCs follow from a lower bound on the RB.

### 5.4 Level partition implies a lower bound

Let \( \mathcal{G} \) be an \( \mathcal{LP} \)-consistent set of MCs, as specified in the above theorem. Let \( M \) be a \( \mathcal{G} \)-multipath (at this stage, unspecified). We associate the nodes (variables) of \( M \) to intervals and levels according to the given level partition.

**Lemma 5.11.** No \( G \in \mathcal{G} \) contains an arc of the form \( x_i \rightarrow x'_j \), or \( x'_i \rightarrow x_j \), where \( \text{iv}(j) > \text{iv}(i) \).

**Proof.** Suppose that \( G_h \in \mathcal{G} \) contains an arc \( x_i \rightarrow x'_j \) (the case \( x'_i \rightarrow x_j \) is symmetric). Such an arc implies the constraint

\[
x_i \geq x'_j.
\]

Due to the upward-indexing in full elaboration, and since \( \text{iv}(j) > \text{iv}(i) \) implies \( j > i \), we have \( x'_j > x'_i \). Hence, \( (x_i > x'_i) \in G \). Thus, \( h = \text{level}(i) \). Now, from \( j > i \), we also have \( x_j > x_i \), and hence (by (4)) \( (x_j > x'_j) \in G \). This implies \( h = \text{level}(j) \), that is, \( \text{level}(j) = \text{level}(i) \). However, \( i \) and \( j \) are not in the same interval. This means that there is a \( j' \) such that \( i < j' < j \) and \( \text{level}(j') \neq h \). From (4) and the upward-indexing, we obtain \( x_i > x'_j \) and consequently \( x'_j > x_j \), implying \( \text{level}(j') = h \), a contradiction.

**Definition 5.12.** A block in \( M \) is a maximal connected set of nodes that belong to the same interval. The level of a block is the level of the interval.

Due to the form of graphs consistent with \( \mathcal{LP} \), a block consists of nodes \( x[t, i] \) where \( i \) ranges over one of the intervals \( VI_i \), and \( t \) ranges over an interval of “time coordinates” (see Figure 7). We refer to the interval number as the height of the block, so a block is higher if it corresponds to a higher-numbered interval. The last lemma indicates that there is no arc that connects a block to a higher one. By the length of the block we mean the number of MCs it spans.

An in-situ arc, mentioned in the proof below, is an arc (of an MC graph) connecting \( x_i \) to \( x'_j \) for some \( i \).

**Lemma 5.13.** Suppose that \( \text{cl}(E)|f \) has an \( \mathcal{LP} \)-consistent set of MCs where \( \mathcal{LP} \) is a level partition of depth \( L \). Then \( T_f^1(N) \in \Omega(N^L) \).
Figure 7: A multipath with a highlighted block (the black nodes), corresponding to interval number 2. The whole of the lowest row is also a block.

**Proof.** The idea of the proof is to construct a multipath \( M \) of length \( \ell(N) \in \Omega(N^L) \), more precisely \( \ell(N) = \lfloor N/n \rfloor L - 1 \), that is \( N \)-satisfiable. The multipath is constructed by the following formula. Let \( b = \lfloor N/n \rfloor \) and let \( 1 \leq t < b^L \). Denote by \( \eta(t) \) the maximal number \( h \) such that \( b^h - 1 \) divides \( t \); clearly, \( 1 \leq \eta(t) \leq L \). Let

\[
M = G_{\eta(1)}G_{\eta(2)} \cdots G_{\eta(b^L-1)}.
\]

Observe that a block of level \( h \) is delimited on its left and right sides either by the end of the multipath, or by an occurrence of \( G_k \) for some \( k > h \). By the definition of \( \eta \), the length of each level \( h \) block is \( b^h - 1 \).

To show that \( M \) is \( N \)-satisfiable, we use Lemma 5.4. Consider a directed path \( \tau \) in \( M \). Suppose that \( \tau \) enters a block associated with interval \( j \), at level \( h \). The block is a rectangular array of nodes of dimensions \( b^h \times |VI_j| \), and contains \( b - 1 \) occurrences of \( G_h \), which is the MC active at this level, that is, including strict in-situ arcs within the block; other MCs in the block have a lower level and therefore have non-strict in-situ arcs. It is not hard to see that the lightest path in the block is one that starts at the upper left corner of the block and ends in the lower right corner, and its weight cannot be smaller than \( -(b \cdot |VI_j|) \).

Once \( \tau \) exits a block, it must proceed to a lower block (by Lemma 5.11), so it visits a block at each height at most once. We conclude that the weight of \( \tau \) is lower-bounded by

\[
\sum_{i=1}^{w} -(b \cdot |VI_i|) = -b(\sum_{i=1}^{w} |VI_i|) = -bn \geq -(N/n)n = -N
\]

which proves that \( M \) is \( N \)-satisfiable.

\[\square\]

5.5 **Lower bound implies a level partition**

We now take the converse direction, and prove that if a sufficiently long multipath exists, the condition in Theorem 5.10 holds.
Recall that a thread is either a down-thread or an up-thread. A thread is in-situ if it consists only of in-situ arcs. Below, we reason on the structure of threads. The treatment of down-threads and up-threads is essentially the same, and to avoid cumbersome notation, we will restrict the reasoning to down-threads.

**Definition 5.14.** Let $M = G_1G_2\ldots G_m$ be a multipath. A segment of $M$ is a multipath $G_bG_{b+1}\ldots G_e$ for some $1 \leq b \leq e \leq m$. A sub-multipath of $M$ is a multipath $M' = H_1H_2\ldots H_{m'}$ such that there is a subsequence $\gamma = (i_1,\ldots, i_{m'})$ of $(1,\ldots, m+1)$, where for all $j$, $H_j = G_{i_j-1};\ldots;G_{i_j-1}$. We write this as $M' = M_\gamma$.

**Lemma 5.15.** If $M$ is an $N$-satisfiable $cl(\mathcal{E})|f$-multipath, so is any sub-multipath $M'$.

**Proof.** $M'$ is a $cl(\mathcal{E})|f$-multipath because $cl(\mathcal{E})|f$ is closed under composition. It is $N$-satisfiable because every path in $M'$ is obtained from a path of $M$ by contracting some segments of the path due to MC composition. A strict segment is contracted to a strict arc, and a non-strict segment to a non-strict arc; at any case, the weight of the contracted path is at least the weight of the original one. $\square$

**Lemma 5.16.** Let $M$ be a $cl(\mathcal{E})|f$-multipath of length at least $(n+1)m$, for some $m > 0$. There is a sub-multipath $M'$ of $M$, of length $m$, that has an in-situ strict thread.

**Proof.** Since $cl(\mathcal{E})|f$ is terminating, by Lemma 3.13, it has a strict complete thread $\tau$. For $0 \leq t \leq m$, let $v(t)$ denote the index such that $x[t,v(t)]$ is on $\tau$. Since the length of $M$ is at least $(n+1)m$, there must be an $i$ such that $v(t) = i$ at least for $m+1$ different indices $t = i_1, i_2, \ldots, i_{m+1}$. Hence, $M_{\langle i_1, i_2, \ldots, i_{m+1} \rangle}$ is the desired sub-multipath. $\square$

**Lemma 5.17.** Suppose that for some $N \geq 1$, $cl(\mathcal{E})|f$ has an $N$-satisfiable multipath longer than $(n+1)^{n+L}(nN+1)^{-L}$. Then there is an $\mathcal{LP}$-consistent set of MCs where $\mathcal{LP}$ is a level partition of depth $L$.

**Proof.** Let $M_0$ be an $N$-satisfiable multipath of length $(n+1)^{n+L}(nN+1)^{-L}$. We will construct a sequence of multipaths, starting with $M_0$, each multipath being a sub-multipath of the former. This sequence will help us find the desired set of MCs. Let $k_0 = 0$. By last lemma, there is a sub-multipath $M_1$ of length $(n+1)^{n+L-1}(nN+1)^{-L}$ that includes an in-situ strict (down-)thread $\tau_0$, say at $x_{i_0}$. Suppose that $M_1$ has another complete thread, disjoint from $\tau_0$. Then arguing as in the proof of Lemma 5.16, we get a sub-multipath $M_2$, of length $(n+1)^{n+L-2}(nN+1)^{-L}$, that includes two in-situ threads, $\tau_0$ and $\tau_1$. We continue this way until we reach a multipath $M_{k_1}$, of length $(n+1)^{n+L-k_1}(nN+1)^{-L}$, that does not include any additional complete thread. All the threads $\tau_0, \ldots, \tau_{k_1-1}$ are complete and in-situ, and at least one of them must be strict (otherwise, Lemma 3.13 is contradicted).

Since $M_0$ is $N$-satisfiable, so is $M_{k_1}$, and each of the threads $\tau_i$ above includes at most $N$ strict arcs. Altogether, they include at most $nN$ strict arcs. Suppose $L > 1$. We are now looking for a long segment of $M_{k_1}$ avoiding these strict arcs. If we drop from $M_{k_1}$ the MCs where such
a strict arc appear, we get at most $|M_{k_1}| - nN$ segments. Hence one segment is at least

$$\frac{(M_{k_1}| - nN}{nN + 1} \geq \frac{(n + 1)^{n+L-k_1}(nN + 1)^{L-1} - nN}{nN + 1} \geq \frac{(n + 1)^{n+L-k_1-1}(nN + 1)^{L-1}}{nN + 1} \geq (n + 1)^{n+L-k_1-1}(nN + 1)^{L-2},$$

Call this segment $M'_{k_1}$. Note that all the arcs of $\tau_0, \ldots, \tau_{k_1-1}$ are non-strict in $M'_{k_1}$. We treat $M'_{k_1}$ as we treated $M_0$ and generate additional sub-multipaths $M_{k_1} + 1, \ldots, M_{k_2}$ such that $M_{k_2}$ has precisely $k_2$ complete in-situ threads, including the $k_1$ present in $M'_{k_1}$; and we argue, as before, that at least one of the new $k_2-k_1$ threads is strict (those inherited from $M'_{k_1}$ are not, due to the choice of $M'_{k_1}$). The length of $M_{k_2}$ is at least $(n + 1)^{n+L-k_2-1}(nN + 1)^{L-2}$. If $L > 2$, we find $M'_{k_2}$ of length at least $(n + 1)^{n+L-k_2-2}(nN + 1)^{L-3}$, and so on, until we arrive at a multipath $M_{k_L}$, in which no further in-situ threads can be discovered (the reader is invited to check that the length of every $M_{k_j}$, for $j < L$, is large enough to allow the construction to continue).

Now, we construct a level partition $LP$ of depth $L$. We let $level(i)$ be $h$ if the first $j$, such that $M_{k_j}$ includes an in-situ thread at $x_i$, is $L - h + 1$. We thus assign levels from $L$ to 1. We assign level 0 to any remaining variables. Sets $VI_1, \ldots, VI_w$ are now defined as maximal intervals of variables that have the same level.

For $1 \leq h \leq L$, let $G_h = \overline{M_{k_j}}$ for $j = L - h + 1$. Then $G_h$ is consistent with $LP$ at level $h$. \hfill \Box

### 5.6 Wrap-up

By combining Lemmas 5.13 and 5.17, Theorem 5.10 is proved. Moreover, we obtain

**Corollary 5.18.** The reachability bound for $f$ in $cl(E)|f$ must be in $\Theta(N^L)$ for some $L$, namely the largest $L$ such that an $LP$-consistent set of MCs exists, where $LP$ has depth $L$.

**Proof.** Let $L$ be as above; then by Lemma 5.13, the reachability bound is in $\Omega(N^L)$. However if it is not in $O(N^L)$, then for sufficiently large $N$ the bound must exceed $(n + 1)^{n+L+1}(nN + 1)^{L}$; now Lemma 5.17 implies that there is a level partition of depth $L + 1$ such that an $LP$-consistent set of MCs exists, contradicting our assumption. \hfill \Box

### 5.7 A Decision Algorithm and its Complexity

From Theorem 5.10, we immediately get the following decision procedure for RBD. The algorithm that follows naturally calls for constructing $cl(E)|f$ and testing for the existence of a suitable level partition and a consistent set of graphs. We define the algorithm’s input to be the instrumented abstract program, since one can in fact modify the instrumented problem in ways that may be useful (for example, we may want to establish bounds not in terms of all initial variables, but only a selected few; this only requires a slight modification of the instrumentation).
Algorithm 5.1. (Reachability Bound Degree) Input: Instrumented abstract program $I(A)$, flow-point $f \in F^A$ and degree $d$.

1. Build $cl(E)|f$.

2. Search for a level partition $LP$ of depth $d$ and an $LP$-consistent set of MCs.

3. Report the conclusion according to Theorem 5.10.

Since the size of $cl(E)|f$ is exponential in the size of $A$, this algorithm is exponential in both time and space, even if we implement the search for $LP$ very cleverly (which we don’t). Instead, we tighten the complexity bound to PSPACE in the following theorem.

Theorem 5.19. The RBD problem is PSPACE-complete (and is PSPACE-hard even for stable systems that have a single flow-point, and guaranteed to be bounded-terminating).

Proof. Upper bound. We give a non-deterministic polynomial-space algorithm for recognizing RBD. By Savitch’s theorem, the problem lies in PSPACE.

As described in Section 3.4, it is possible to generate transitions of $cl(E)|f$ in polynomial space, given access to $A$. Our algorithm uses this procedure to sample, non-deterministically, a set of MCs $G_1, \ldots, G_L$. The algorithm then attempts to construct a level partition that these graphs are consistent with—a simple polynomial-time procedure (though, since the algorithm is already non-deterministic, it could as well just guess the level partition and then verify consistency). The algorithm accepts if a level partition and a consistent set of graphs have been found.

Lower bound. To prove PSPACE-hardness, we reduce from the bounded termination problem for stable systems with a single flow-point. Let $A$ be a $(MC, Z)$-CTS program with $n$ variables. Add new variables $y_1$ through $y_{n+1}$. Add constraints to ensure that these variables will be bounded in $I(A)$. From every MC $G$, create $n + 1$ copies, and in copy $i$ include the relations: $y_i > y_j'$ and $y_j \geq y_j'$ for $j > i$. These variables ensure termination in $O(N^{n+1})$ steps. However, if $A$ is in itself bounded-terminating, the system will terminate in $O(N^n)$ steps. Thus, the bounded termination of $A$ can be decided by testing the new system for a reachability bound in $\Omega(N^{n+1})$.

6 Significance for Concrete Programs

$(MC, Z)$-CTSs may be considered as an abstract computational model and its analysis as a goal in itself, which is interesting since such systems, despite the relative simplicity of the constraints, may exhibit a complex behaviour. However, we would like to promote the view that such systems are useful as an abstraction of concrete programs, to facilitate their analysis.

In this section we consider the question: What does the fact that a $(MC, Z)$-CTS has polynomially bounded height tell us about the program it represents? We discuss this question in three settings, which we first present informally; secondly, we give a formal example using a toy programming language defined for this sake only; and finally relate our discussion to the implementation of two published analysis tool which use the abstract and conquer approach.
6.1 The three settings, informally

Flat imperative programs  We first consider imperative programs without any procedure calls. Figures 1 and 2 are examples of flat imperative programs abstracted in the natural way. The control-flow graph corresponds to the flow-chart of the program; transitions correspond to program instructions, or—more effectively—basic blocks. Often, the assumption is that such a block takes a constant time to execute.

In this setting, the height of the transition system represents the time complexity of the program. In terms of complexity classes, this allows us to identify a program as polynomial-time in the selected input parameters. When basic blocks have associated costs which are not uniform, the Reachability Bound analysis may allow for inferring a bound on the cost of a computation based on the formula \( \sum_f \text{cost}(f) \times T_f \) where \( f \) ranges over flow-points and \( T_f \) is a reachability bound for \( f \).

In practice, the control-flow graph of the program may be transformed during abstraction. Suppose that we select a set of cut points in the program’s flow-chart such that any cycle must traverse a cut point, and the program entry is a cut point. Any such set of cut points may be chosen as the set of flow-points as long as any (finite) path between two cut points is represented by an abstract transition. The conclusions on the concrete program’s complexity remain valid.

For programs that contain procedure calls, but not recursion, a bottom-up analysis may be applicable. The results of analyzing a procedure \( p \) will be plugged into the summation for its caller, using reachability bounds, as shown above ([AAGP10] also describes a bottom-up process, however their analysis is not based on the RB approach).

Pure-functional programs  [LJBA01] showed the simplest way in which a (first-order, eager) pure-functional program may be abstracted. The control-flow graph is the call-graph of the program; flow-points are function names and transitions correspond to functions calls. Hence, every call chain of the program corresponds to a particular run of the transition system.

It should be clear that in this setting, the height of the transition system represents the stack height of the program. This is a resource of practical importance in itself. What can we infer in terms of the traditional resources, space and time? The pure functionality suggests that there is no iteration but recursion, so it may be possible to bound the execution time of a function body, or the space it consumes, outside any calls it performs; often, this bound will be a constant. If functions cannot allocate “heap space” at all, the stack height corresponds to space usage. If the functions can allocate space outside the stack, exponential space may be consumed for a polynomial stack height. Because the call tree is a tree of bounded degree, we obtain an exponential time bound (that is, a constant to a polynomial power).

In terms of complexity classes, we may conclude that the program is polynomial-space or only the weaker result that it has the class \( \text{EXPTIME} \).

Also in this setting, we note that the abstraction may create the CFG in different ways, which are sometimes useful. For example, [MV06b] chose call sites to be flow-points rather than function names.

6.2 Analysing a simple programming language

We demonstrate the ideas more formally by defining three variants of a simple (but Turing complete) functional programming language SFPL and a simple-minded, conservative abstraction \( A \)
mapping SFPL programs to \((\mathcal{MC}, \mathbb{Z})\)-CTSs. Since the language has functional style, imperative programs are represented by tail recursion.

The syntax of SFPL is defined in Table 1, and further explained below. Semantically, SFPL programs operate on strings over a finite alphabet \(\Sigma = \{0, 1, \ldots\}\). The expression \(a:x\), where \(a \in \Sigma\), evaluates to \(a\) followed by the value of \(x\).

A program is a collection of definitions which leaves no undefined identifiers. A function is defined by a set of definitional patterns. To avoid ambiguity, a first-match disambiguation rule can be used. If there is no match, the program halts. A wildcard ‘?’ can be introduced in patterns as syntactic sugar. For simplicity, all functions have the same arity \(n\). A function \(f^\chi\) is indicated as the entry point.

Example 6.1. Here is a short SFPL program that tests two strings for equality, where \(\Sigma = \{0, 1\}\). For some complication, it occasionally swaps its arguments. We use the first-match rule for pattern matching.

\[
\begin{align*}
f(\varepsilon, \varepsilon) &= 1 \\
f(0:x_1, 0:x_2) &= f(x_1, x_2) \\
f(1:x_1, 1:x_2) &= f(x_2, x_1) \\
f(x_1, x_2) &= \varepsilon
\end{align*}
\]

The specification of function bodies and their return values differs in the three language variants:

**SFPL\(_1\)** Allows only the simple and the tail-recursive expressions as function bodies. Hence, it represents imperative programs. The return value of functions is \(\Sigma^*\).

**SFPL\(_2\)** Allows, in addition, the conditional expression (see Table 1). The condition \(g_1(\ldots)\) is evaluated first; if it is a non-empty string, the value of the conditional is obtained by evaluating \(g_2(\ldots)\), and otherwise, \(g_3(\ldots)\).

**SFPL\(_3\)** Also includes the nested (‘let’) expression.

**Definition 6.1.** Abstraction \(A\) maps an SFPL program to a \((\mathcal{MC}, \mathbb{Z})\)-CTS as follows: the flow-point set \(F\) is the set of defined functions. There is an abstract transition \(G : f \rightarrow g\) for every call expression \(g(\beta_1, \ldots, \beta_n)\) in a definition \(f(\alpha_1, \ldots, \alpha_n) = \ldots\); a relation among \(x_i\) and \(x'_j\) is included in \(G\), dependent on the patterns \(\alpha_i\) and \(\beta_j\), as specified in the following table (the cases missing in the table contribute no constraint).

| \(\alpha_i\) | \(\beta_j\) | relation |
|-------------|-------------|----------|
| \(\varepsilon\) | \(\varepsilon\) | = |
| \(x_i\) | \(\varepsilon\) | \(\geq\) |
| \(x_i\) | \(x_i\) | = |
| \(a:x_i\) | \(\varepsilon\) | > |
| \(a:x_i\) | \(a'\) | \(\geq\) |
| \(a:x_i\) | \(a':x_i\) | = |
We can now state our observations in this formal setting.

We assume a typical RAM implementation of SFPL, using a stack for function calls, and a heap memory to keep the strings, which are implemented as linked lists, so that removing or adding an element at the front takes constant time and space. We also assume immediate garbage collection so that garbage does not accumulate (this is easy for such a language, e.g., by reference counting).

CLAIM 6.2. If $A(P)$ satisfies bounded termination, and $P$ is an SFPL$_i$ program, then, for all $i$, the stack height is polynomially bounded in the size of the input strings. For $i = 1$, the program runs in polynomial time; for $i = 2$, its space usage is polynomial; and for $i = 3$, its running time is bounded by $2^{\text{poly}(n)}$.

A formal proof of this claim is skipped as it is uninteresting and tedious (demanding a formalization of semantics and complexity, currently left informal). The time bound in the case of SFPL$_1$ is straightforward and that of SFPL$_3$ follows almost as easily since the height of the recursion tree is polynomial. As to the space bound for SFPL$_2$, note that a branch in the recursion tree only occurs in this language when a conditional is evaluated, and that heap space allocated by the evaluation of the condition ($g_1$) can be discarded once it is determined whether the return value is $\varepsilon$ or not. Thus, for the purpose of bounding the space, it is possible to consider the stack height.

Note that our language is, in fact, a Turing-complete one. It is possible to extend Claim 6.2 to a proposition of class capture: every decision problem in PTIME (resp., PSPACE, EXPTIME) may be represented by an SFPL$_1$ (resp., SFPL$_2$, SFPL$_3$) program. We find that this result is of little consequence to the main goals of our work, and have decided to omit the proof.

Table 1: Syntax of SFPL
6.3 A discussion of two analyzers for real-world languages

We compare our informal statements at the beginning of this section to the way abstraction is used in the WTC project by Alias et al. [ADFG10a] and the COSTA project by Albert et al. [AAG+09, AAG+12]. As described in Section 1, both works use a constraint language richer than monotonicity constraints, but this issue is independent of the current discussion (it may affect precision of the abstraction—see the next section).

In [ADFG10a], C language programs are abstracted to affine constraint transition systems. They have implemented two forms of abstraction. One represents a basic block as a transition, another only places a flow-point at a loop header and expands the loop body so that every path through the loop is abstracted to one abstract transition. This means that exponentially more transitions may be generated, but the abstraction will be more precise. In both cases, our informal description for “flat imperative programs” applies.

In [AAG+09, AAG+12], Java Bytecode programs are abstracted to transition systems which express a sequential transition (from a block in the flow-chart to the next) and a procedure call in essentially the same way. Thus a sequential computation is treated as tail recursion—much like in our toy language. The analysis described in [AAGP10] distinguishes the case of tail recursion from the case where a recursion tree is involved and an exponential bound may result. This is again similar to the framework we have described. Their abstract programs are annotated with cost expressions, used in computing a closed formula for a cost bound. As stated earlier, in our framework this may require the computation of reachability bounds and a (symbolic) summation, and possibly also another static analysis to bound the cost expressions in terms of input parameters.

There are other tools that translate real-world languages to some kind of contraint transition systems, for example [SMP10] analyze Java Bytecode and [MV06b] analyze the ACL2 programming language, both for the purpose of termination analysis. Since the correspondence of the abstract program to the concrete one is still essentially as in our discussion, we conclude that the generated abstract programs could be used, perhaps with some adaptation, for cost analysis as well.

6.4 Reflections on Effective Abstraction

Both of the tools we cited in the last section use a more expressive abstraction—an affine-constraint CTS (also known as a CTS with polyhedral constraints). This constraint language is strictly more expressive, as monotonicity constraints form a simple special case of affine constraints. So there is reason to fear that by abstracting a program to a \((MC, Z)-CTS\) we might lose crucial information. We would like to argue that this consideration should not discourage researchers from employing this abstraction.

One reason for our optimism is the existing empirical evidence for the effectiveness of the size-change technique in termination analysis [LS97, CT99, TG05, MV06b, BAC08, SMP10, KH09, CGBA+11]. As shown in our theoretical sections, the complexity analysis is a refinement of termination analysis and reuses its methods. Nonetheless, we argue that for bounded termination analysis, it is necessary to transfer more information to the \((MC, Z)-CTS\) than one does for termination, in particular if one wants to analyze it as a stand-alone abstract program. The main reason is the necessity for bounding variables. Consider Program 2 in Figure 2 on Page 6. If the initial assignment is changed from \(i = N\) to \(i = 2*N\), and the abstract variables still correspond
to the program variables in a one-to-one fashion, we will lose the bound on \( i \) in terms of \( N \), since it is not a monotonicity constraint. Note that this relation is not necessary for the termination proof, but is crucial for deducing \textit{bounded} termination.

We think that this problem may be mitigated by the use of an auxiliary bound analysis, one which attempts to bound expressions in the program in terms of the designated input variables. Such an analysis can be performed by, for example, polyhedral analysis \cite{CH78} or one of its many variants. When an expression \( exp \) is found to be bounded by a bound \( B_{\text{exp}} \) in terms of the input, an abstract variable representing \( B_{\text{exp}} \) may be added to the abstraction. In order to avoid combinatorial explosion, one may decide to add such variables only when necessary for changing an unbounded variable in the \((MC,Z)\)-CTS into a bounded one; one may also opt to keep only a representative of the maximum among such expressions, in the same way we used \( x_{\text{max}} \) in Section 4. Note that if we have an analysis that (unlike polyhedral analysis) may ascertain a non-polynomial bound on \( exp \) we may end up with complexity bounds that are polynomial functions of that bound, hence possibly non-polynomial as a function of the input parameters.

We also invite the reader to note that \((MC,Z)\)-CTSs can capture rather complex behaviours. The examples in the next section illustrate a few. This should be at least a reason to consider the model interesting.

7 Additional Examples

To illustrate the variety of loop structures that can be represented and analysed, we have selected a few examples, shown in this section as C program fragments; see also examples on pages 4, 6, 19. In all these examples, it is pretty simple to verify that the associated constraints systems are indeed bounded terminating.

\textit{Example 7.1.} This is a quadratic-time example (similar to Figure 2, but counting up rather than down), from \cite{GMC09}, where it is analysed by means of counter instrumentation and bounding.

\begin{verbatim}
SimpleMultipleDep(int n, int m) {
    x = 0; y = 0;
    while (x < n)
        if (y < m) y++;
        else { y = 0; x++; }
}
\end{verbatim}

Here is its MC representation:

\begin{equation}
\begin{array}{c}
\begin{array}{c}
(1) \\
(2) \\
(3) \\
(4)
\end{array}
\end{array}
\end{equation}

\begin{itemize}
\item (1) \( x' = 0' \land y' = 0' \)
\item (2) \( x < n \land \text{Same}(m, n, y, 0) \)
\item (3) \( y < m \land y < y' \land \text{Same}(m, n, x, 0) \)
\item (4) \( y' = 0' \land x < x' \land \text{Same}(m, n, 0) \)
\end{itemize}
Example 7.2. The next example is from [GMC09]. They explain that their algorithm does not handle it because of the lack of path-sensitive information. Alain et al. report in [ADFG10b] that their tool solved this instance.

```c
void pathSensitive2(int n, int b, int x) {
    int t;
    if (b>=1) t=1; else t = -1;
    while (x<=n) {
        if (b>=1)
            x=x+t;
        else
            x=x-t;
    }
}
```

In its MC representation, we represent the effect of addition and subtraction disjunctively: for example, we use the knowledge that \( x = x+t \) is a command that increases \( x \) if \( t \) is positive, decreases \( x \) if \( t \) is negative, etc. Thus we have three MCs for each command of this form. In this particular program, two of those represent transitions that will never be taken in an actual run, but we do not assume our “front end” to do such an analysis.

\[
\begin{align*}
(1) & \quad b > 0 \land t' > 0' \land \text{Same}(b, n, 0) \\
(2) & \quad b \leq 0 \land t' < 0' \land \text{Same}(b, n, 0) \\
(3) & \quad x \leq n \land \text{Same}(x, b, n, t, 0) \\
(4) & \quad b > 0 \land t > 0 \land x' > x \land \text{Same}(b, n, t, 0) \\
(5) & \quad b > 0 \land t < 0 \land x' < x \land \text{Same}(b, n, t, 0) \\
(6) & \quad b > 0 \land t = 0 \land x' = x \land \text{Same}(b, n, t, 0) \\
(7) & \quad b \leq 0 \land t > 0 \land x' > x \land \text{Same}(b, n, t, 0) \\
(8) & \quad b \leq 0 \land t < 0 \land x' < x \land \text{Same}(b, n, t, 0) \\
(9) & \quad b \leq 0 \land t = 0 \land x' = x \land \text{Same}(b, n, t, 0)
\end{align*}
\]

Example 7.3. The next program does not have a lexicographic-linear global ranking function, an obstacle for tools that, explicitly or implicitly, require functions of this kind (this class includes [ADFG10a], by their own description, and also COSTA, though the fact is implicit—see Section 2. The class also includes the algorithm of [GMC09], according to a discussion in [ADFG10a]). We omit the transition system this time, which the reader would be able to create at ease (for assignments like \( y = y+x \) it suffices, in this case, to consider \( y \) as being unconstrained, although a disjunctive representation of the effect, as in the previous example, could be harmlessly included).

```c
void min(int x, int y) {
    while (y > 0 && x > 0) {
        if (x>y) z = y;
        else z = x;
        if (*){ y = y+x; x = z-1; z = y+z }
        else { x = y+x; y = z-1; z = x+z }
    }
}
```
Another instance where lexicographic linear global ranking functions do not suffice is given in Figure 6 (Page 19).

**Example 7.4.** The following example from [GZ10] shows the weakness of a straight-forward abstraction to monotonicity constraints.

```plaintext
e = 0;
while (i < n) {
    j = i + 1;
    while (j < n) {
        if (A[j])
            j--; n--;
        j++;
    }
    i++;
}
```

The problem is that abstracting the effect of the `if`-block on `j` to `j' < j` does not allow a later analysis to figure out that `j++` “undoes” this decrement. There are, of course, multiple ways to handle this issue. For example, one could use a more expressive abstraction—say, \((\text{Aff}, \mathbb{Z})\)-CTS—and use it for computing a composition in the closure algorithm, widening to monotonicity constraints only at the level of cycles. This still allows the use of \((\text{MC}, \mathbb{Z})\)-CTS algorithms for the bound analysis.

### 8 Conclusion

The Monotonicity Constraint abstraction came into being specifically for the purpose of termination analysis [CT99, LS97, Sag91]. It is natural to wish to extend termination proofs into complexity bounds. This work does it for the MC framework. For abstract programs, the complexity problem is to bound the length of transition sequences. Pleasantly, we find that the problem is decidable, and its computational complexity is the same as termination. An interesting conclusion is that a bound exists if and only if a polynomial one does (a different kind of statement than stating that a certain analysis tool only finds polynomial bounds!). We have investigated the problem of obtaining polynomials of minimum degree, and found that to be also computable and in PSPACE.

Since we are dealing with abstract programs, the question of relating these bounds to complexity of the concrete program arises. We illustrate how the polynomial bound may mean polynomial time, space or a polynomial exponent. In fact, classes PTIME, PSPACE and EXP-TIME may all be captured by very simple abstraction of programs to constraint systems.

We have not yet been able to perform an empirical evaluation, but at least theoretically, our results sustain the claim that, just as they proved quite useful for termination, MCs can contribute to complexity analysis.

In this research, we chose to relax the expression of the bound to a big-O one so we can show that the precise degree is decidable. We propose as an open problem the question whether bounds that have precise explicit constants can also be computed (in polynomial space?).
References

[AAG+09] 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, volume 5705 of Lecture Notes in Computer Science, pages 258–288. Springer Berlin / Heidelberg, 2009.

[AAG+12] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. Theoretical Computer Science, 413(1):142–159, 2012. special issue: Quantitative Aspects of Programming Languages (QAPL 2010).

[AAGP08] 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, Proceedings, volume 5079 of Lecture Notes in Computer Science, pages 221–237. Springer, 2008.

[AAGP10] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-form upper bounds in static cost analysis. Journal of Automated Reasoning, pages 1–43, 2010.

[ADFG10a] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In Static Analysis, Proceedings of the 17th International Symposium, Perpignan, France, 2010.

[ADFG10b] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. WTC toolsuite benchmarks. http://www.ens-lyon.fr/LIP/COMPSYS/Tools/Ranking/, September 2010.

[AM09] M. Avanzini and G. Moser. Dependency pairs and polynomial path orders. In Proc. RTA ’09, LNCS 5595, pages 48–62, 2009.

[Ama05] Roberto M. Amadio. Synthesis of max-plus quasi-interpretations. Fundam. Inform, 65(1-2):29–60, 2005.

[Ave06] James Avery. Size-change termination and bound analysis. In M. Hagiya and P. Wadler, editors, Functional and Logic Programming: 8th International Symposium, FLOPS 2006, volume 3945 of Lecture Notes in Computer Science. Springer, 2006.

[BA08] Amir M. Ben-Amram. Size-change termination with difference constraints. ACM Trans. Program. Lang. Syst., 30(3):1–31, 2008.

[BA09] Amir M. Ben-Amram. A complexity tradeoff in ranking-function termination proofs. Acta Informatica, 46(1):57–72, February 2009.
Amir M. Ben-Amram. On decidable growth-rate properties of imperative programs. In Patrick Baillot, editor, *International Workshop on Developments in Implicit Computational complexity (DICE 2010)*, volume 23 of *EPTCS*, pages 1–14. ArXiv.org, 2010.

Amir M. Ben-Amram. Size-change termination, monotonicity constraints and ranking functions. *Logical Methods in Computer Science*, 6(3), 2010.

Amir M. Ben-Amram. Monotonicity constraints for termination in the integer domain. *Logical Methods in Computer Science*, 7:1–43, 2011.

Amir M. Ben-Amram and Michael Codish. A SAT-based approach to size change termination with global ranking functions. In C.R. Ramakrishnan and Jakob Rehof, editors, *14th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5028 of *LNCS*, pages 46–55. Springer, 2008.

Stephen Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the polycase functions. *Computational Complexity*, 2:97–110, 1992.

G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11:33–53, 2001.

Guillaume Bonfante and Florian Deloup. Complexity invariance of real interpretations. In Jan Kratochvíl, Angsheng Li, Jirí Fiala, and Petr Kolman, editors, *Theory and Applications of Models of Computation*, volume 6108 of *Lecture Notes in Computer Science*, pages 139–150. Springer Berlin / Heidelberg, 2010.

Ralph Benzinger. Automated complexity analysis of nuprl extracted programs. *Journal of Functional Programming*, 11(1):3–31, 2001.

Ralph Benzinger. Automated higher-order complexity analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, 2004.

Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. Linear, polynomial or exponential? complexity inference in polynomial time. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *LNCS*, pages 67–76. Springer, 2008.

Laura Bozzi and Sophie Pinchinat. Verification of gap-order constraint abstractions of counter systems. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation, 13th International Conference, VMCAI 2012*, volume 7148 of *Lecture Notes in Computer Science*, pages 88–103. Springer, 2012.

Michael Codish, Carsten Fuhs, Jürgen Giesl, and Peter Schneider-kamp. Lazy abstraction for size-change termination. In Christian G. Fermüller and Andrei Voronkov, editors, *17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2010)*, volume 6397 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2010.
Michael Codish, Igor Gonopolskiy, Amir M. Ben-Amram, Carsten Fuhs, and Jürgen Giesl. SAT-based termination analysis using monotonicity constraints over the integers. *Theory and Practice of Logic Programming, 26th Int’l. Conference on Logic Programming (ICLP’11) Special Issue, 11(Special Issue 4-5):503–520, July 2011.*

Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth annual ACM Symposium on Principles of Programming Languages*, pages 84–96. ACM, ACM, January 1978.

Adam Cichon and Pierre Lescanne. Polynomial interpretations and the complexity of algorithms. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, pages 139–147. Springer-Verlag, 1992.

Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. Testing for termination with monotonicity constraints. In Maurizio Gabbrielli and Gopal Gupta, editors, *Logic Programming, 21st International Conference, ICLP 2005*, volume 3668 of *Lecture Notes in Computer Science*, pages 326–340. Springer, 2005.

A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proc. 1964 International Congress for Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam, 1964.

Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.

Michael Codish and Cohavit Taboch. A semantic basis for termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999. Preliminary (conference) version in LNCS 1298 (1997).

Karl Crary and Stephnie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’00, pages 184–198, New York, NY, USA, 2000. ACM.

S. K. Debray, P. López-García, M. V. Hermenegildo, and N.-W. Lin. Estimating the computational cost of logic programs. In *Proc. SAS 94*, pages 255–265. LNCS 864, 1994.

Saumya Debray and Nai wei Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15:48–62, 1993.

Bhargav S. Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2008.
[GJK09] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In Michael Hind and Amer Diwan, editors, Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009, pages 375–385. ACM, 2009.

[GMC09] 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, pages 127–139. ACM, 2009.

[GTSKF06] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. Journal of Automated Reasoning, 37(3):155–203, 2006.

[Gul09] Sumit Gulwani. SPEED: Symbolic complexity bound analysis. In Ahmed Bouajjani and Oded Maler, editors, Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, volume 5643 of Lecture Notes in Computer Science, pages 51–62. Springer, 2009.

[GZ10] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In Benjamin G. Zorn and Alexander Aiken, editors, Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010, pages 292–304. ACM, 2010.

[HH10] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In Andrew Gordon, editor, Programming Languages and Systems, volume 6012 of Lecture Notes in Computer Science, pages 287–306. Springer Berlin / Heidelberg, 2010.

[HM08a] N. Hirokawa and G. Moser. Complexity, graphs, and the dependency pair method. In Proc. LPAR ’08, volume 5330 of LNAI, pages 652–666, 2008.

[HM08b] Nao Hirokawa and Georg Moser. Automated complexity analysis based on the dependency pair method. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings, volume 5195 of Lecture Notes in Computer Science, pages 364–379. Springer, 2008. Full version available on-line (under review).

[Hof03] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. Information and Computation, 183(1):57–85, 2003. International Workshop on Implicit Computational Complexity (ICC’99).

[JHLH10] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In The 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, (POPL), pages 223–236, New York, NY, USA, 2010. ACM.
[JK09] Neil D. Jones and Lars Kristiansen. A flow calculus of mwp-bounds for complexity analysis. *ACM Trans. Computational Logic*, 10(4):1–41, 2009.

[KA80] Takumi Kasai and Akeo Adachi. A characterization of time complexity by simple loop programs. *Journal of Computer and System Sciences*, 20(1):1–17, February 1980.

[KH09] Alexander Krauss and Armin Heller. A mechanized proof reconstruction for SCNP termination. Presented in the Tenth International Workshop on Termination WST’09, Leipzig, 2009.

[Kra07] Alexander Krauss. Certified size-change termination. In Frank Pfenning, editor, *11th International Conference on Automated Deduction (CADE)*, volume 4603 of *LNAI*, pages 460–475. Springer-Verlag, July 2007.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the Twenty-Eighth ACM Symposium on Principles of Programming Languages*, January 2001, volume 28, pages 81–92. ACM press, January 2001.

[LM88] Daniel Le Métayer. Ace: an automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, 1988.

[LS97] Naomi Lindenstrauss and Yehoshua Sagiv. Automatic termination analysis of Prolog programs. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 64–77, Leuven, Belgium, Jul 1997. MIT Press.

[MP09] Jean-Yves Marion and Romain Péchoux. Sup-interpretations, a semantic method for static analysis of program resources. *ACM Transactions on Computational Logic*, 10(4), 2009.

[MPS10] Manuel Montenegro, Ricardo Peña, and Clara Segura. A space consumption analysis by abstract interpretation. In Marko van Eekelen and Olha Shkaravska, editors, *Foundational and Practical Aspects of Resource Analysis*, volume 6324 of *Lecture Notes in Computer Science*, pages 34–50. Springer Berlin / Heidelberg, 2010.

[MSW08] G. Moser, A. Schnabl, and J. Waldmann. Complexity analysis of term rewriting based on matrix and context dependent interpretations. In Ramesh Hariharan, Madhavan Mukund, and V Vinay, editors, *Proc. FSTTCS ’08*, LIPIcs 2, pages 304–315. Dagstuhl Publishing, 2008.

[MT09] Panagiotis Manolios and Aaron Turon. All-Termination(T). In *Proceedings of the 15th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2009.

[MV06a] Panagiotis Manolios and Daron Vroon. Integrating static analysis and general-purpose theorem proving for termination analysis. In *Proceedings of ICSE’06: International Conference on Software Engineering*, pages 873–876. ACM, 2006.
[MV06b] Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In Proceedings, Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, volume 4144 of LNCS, pages 401–414. Springer-Verlag, 2006.

[NEG11] Lars Noschinski, Fabian Emmes, and Jürgen Giesl. A dependency pair framework for innermost complexity analysis of term rewrite systems. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, Automated Deduction, 23rd International Conference, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings, volume 6803 of Lecture Notes in Computer Science, pages 422–438. Springer, 2011.

[NW06] Karl-Heinz Niggl and Henning Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. SIAM J. Comput, 35(5):1122–1147, 2006.

[NZM10] F. Neurauter, H. Zankl, and A. Middeldorp. Revisiting matrix interpretations for polynomial derivational complexity of term rewriting. In Proc. LPAR ’10, volume 6397 of LNCS, pages 550–564, 2010.

[Ros89] M. Rosendahl. Automatic complexity analysis. In Proceedings of the Conference on Functional Programming Languages and Computer Architecture FPCA’89, pages 144–156. ACM, 1989.

[Sag91] Yehoshua Sagiv. A termination test for logic programs. In Vijay Saraswat and Kazunori Ueda, editors, Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, pages 518–532. MIT Press, 1991.

[SJ05] Damien Sereni and Neil D. Jones. Termination analysis of higher-order functional programs. In Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS 2005), volume 3780 of Lecture Notes in Computer Science, pages 281–297. Springer-Verlag, 2005.

[Slo] N. J. A. Sloane. The on-line encyclopedia of integer sequences. Published electronically at www.research.att.com/~njas/sequences/.

[SMP10] Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for Java bytecode based on path-length. ACM Trans. Program. Lang. Syst., 32(3):1–70, 2010.

[TG05] René Thiemann and Jürgen Giesl. The size-change principle and dependency pairs for termination of term rewriting. Applicable Algebra in Engineering, Communication and Computing, 16(4):229–270, September 2005.

[Wal10] Johannes Waldmann. Polynomials bounded matrix interpretations. In Christopher Lynch, editor, Proceedings of the 21st International Conference on Rewriting Techniques and Applications, volume 6 of Leibniz International Proceedings in Informatics (LIPIcs), pages 357–372, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
[Weg75] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, September 1975.

[ZGSV11] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In Eran Yahav, editor, *Proceedings of the 8th International Symposium on Static Analysis, SAS 2011, Venice, Italy*, volume 6887 of *Lecture Notes in Computer Science*, pages 280–297, 2011.