Logical Methods in Computer Science
Vol. 7 (3:04) 2011, pp. 1–43
www.lmcs-online.org

Submitted Apr. 26, 2010
Published Aug. 24, 2011

MONOTONICITY CONSTRAINTS FOR TERMINATION IN THE INTEGER DOMAIN

AMIR M. BEN-AMRAM

School of Computer Science, The Tel-Aviv Yaffo Academic College, Israel
e-mail address: benamram.amir@gmail.com

Abstract. Size-Change Termination (SCT) is a method of proving program termination based on the impossibility of infinite descent. To this end we use a program abstraction in which transitions are described by *monotonicity constraints* over (abstract) variables. When only constraints of the form $x > y'$ and $x \geq y'$ are allowed, we have size-change graphs. In the last decade, both theory and practice have evolved significantly in this restricted framework. The crucial underlying assumption of most of the past work is that the domain of the variables is well-founded. In a recent paper I showed how to extend and adapt some theory from the domain of size-change graphs to general monotonicity constraints, thus complementing previous work, but remaining in the realm of well-founded domains. However, monotonicity constraints are, interestingly, capable of proving termination also in the integer domain, which is not well-founded.

The purpose of this paper is to explore the application of monotonicity constraints in this domain. We lay the necessary theoretical foundation, and present precise decision procedures for termination; finally, we provide a procedure to construct explicit global ranking functions from monotonicity constraints in singly-exponential time, and of optimal worst-case size and dimension (ordinal).

1. Introduction

This paper concerns automated termination analysis—deciding whether a program terminates, and possibly generating a *global ranking function.* The termination problem is well-known undecidable for Turing-complete programming languages; one of the ways in which this obstacle may be circumvented is to study, for this purpose, a class of *abstract programs* that are expressive enough to allow many concrete programs to be represented, so that termination of the abstraction implies termination of the concrete program. An important point is that the abstract programs do not constitute a Turing-complete programming language and *can* have a decidable termination problem. The abstraction studied in this paper is *monotonicity constraint transition systems* [12, 6]. The term will be usually abbreviated to MCS.

1998 ACM Subject Classification: D.2.4; F.3.1.
Key words and phrases: program analysis, SCT, termination, ranking functions.

DOI:10.2168/LMCS-7 (3:04) 2011

The MCS abstraction is an extension of the SCT (Size-Change Termination [23]) abstraction, which has been studied quite extensively during the last decade[1]. represent a program as a transition system with states. Abstraction of a program consists of the formation of a *control-flow graph* for the program, identification of a set of *state variables*, and formation of a finite set of abstract transitions (i.e., abstract descriptions of program steps, where the notion of step can be tuned to different needs).

In the SCT abstraction, an abstract transition is specified by a set of inequalities, that relate variable values in the target state to those of the source state (these inequalities are often represented by a *size-change graph*). Extending this notion, a *monotonicity constraint* (MC) allows for any conjunction of order relations, including equalities as well as strict and non-strict inequalities, and involving any pair of variables from the source state and target state. The *Monotonicity Constraint Transition Systems* treated in this paper also allow constraints to be associated with a point in the control-flow graph (these are called *state invariants*).

The size-change technique was conceived to deal with well-founded domains, where infinite descent is impossible. Termination is deduced by proving that any (hypothetical) infinite run would decrease some value monotonically and endlessly, so that well-foundedness would be contradicted.

Monotonicity constraints generalize the SCT abstraction and are clearly more expressive, a fact that was highlighted by Codish, Lagoon and Stuckey [12]. They made the intriguing observation that pre-existing termination analyzers based on monotonicity constraints [24, 13, 25] apply a termination test which is sound and complete for SCT, but incomplete for general monotonicity constraints, even if one does not change the underlying model—namely that "data" are from an unspecified well-founded domain. In addition, they pointed out that monotonicity constraints can imply termination under a different assumption—that the data are integers. Integers, not being well-founded, cannot be handled by the SCT abstraction.

In practice, the integers are already the predominant domain for monotonicity constraints and size-change termination. Often—in particular in functional and logic programming—they represent the size of a list or tree (whence the term *size-change termination*), and are necessarily non-negative, which allows the well-founded model to be used. In contrast, in certain application domains, and typically in imperative programming, the crucial variables are of integer type and might be negative (by design or by mistake). But MCs can still imply termination, as witnessed by the loop `while(x<y) x=x+1`. The value of `x` does not descend but grow; however the constraint `x` $<$ `y` (along with the fact that `y` does not change) tells us that this cannot go on forever.

In a previous paper [6], the theory of monotonicity constraint transition systems in the well-founded model was investigated. Main results include:

- The syntax and semantics of the abstraction are presented.
- A *combinatorial* termination criterion is formulated in terms of the representation of monotonicity constraints as graphs (briefly: the existence of an infinite

[1] References are too numerous to cite here, but see the author's summary web page `http://www2.mta.ac.il/~amirben/sct.html`).

descending path, or walk, in every infinite multipath). This is an adaptation of the SCT criterion from [23].
- It is proved that satisfaction of this criterion is equivalent to the termination of transition system that satisfies the constraints—in logical terms, the criterion is sound and complete.
- Termination of MCSs is shown decidable, and more precisely, PSPACE-complete. Two decision procedures are given: a direct *closure-based* algorithm, and a reduction of the problem to SCT, which provides an alternative algorithm, along with insight into the relationship between the two constraint domains.
- An algorithm is given to construct explicit global ranking functions given any terminating MCS (recall that a global ranking function is a function from program states into a well-founded domain, that decreases on every transition). The algorithm has optimal time complexity ($2^{O(n \log n)}$) and produces a ranking function whose values are tuples, under the lexicographic ordering. It is also optimal in the dimension (length of the tuples, implying the ordinal of the codomain), at least in a worst-case sense.

The contribution of this paper is to obtain some similar results in the integer model. This development is more complicated than in the well-founded model, but is certainly worthwhile due to the practical importance of this domain (many published termination analyses target the integers specifically; a few, closest to this work, are referenced in the related-work section).

An intuitive reduction of the integer case back to the well-founded case is to create a new variable for every difference $x_i - x_j$ which can be deduced from the constraints to be non-negative, and also deduce relations among these new variables to obtain a new abstract program over the natural numbers. But this solution may square the number of variables, which is bad because this number is in the exponent of the complexity, and in general is not complete (see Example 2.9) in the next section). We tackle the problem directly instead, but as an interesting corollary we shall find that the above reduction is, in fact, a correct solution given a certain preprocessing of the program.

This paper is organized as follows. In the next section we formally introduce monotonicity constraint transition systems and their semantics. After this quick technical introduction a sample of examples are given, witnessing the range of termination arguments captured by this framework. Section 3 recalls the notion of a *stable* MCS (introduced in [6]). Stabilization propagates invariants around the abstract program and makes it more amenable to local analysis. Section 4 gives a combinatorial termination criterion (in terms of graphs), similar to what has been known for SCT and for MCS in previous work. The criterion is proved sound and complete and decision algorithms are discussed. In Section 5, we recall another notion from [6], that of *elaboration.* Elaborating a systems makes some information which implicit in it, explicit, and further simplifies its algorithmic processing. It is used in Section 6, presenting the algorithm to construct ranking functions. Section 7 complements the previous sections by briefly explaining the role of reachability (sometimes a program may seem to contain an infinite loop, but it is not reachable). Section 8 discusses related work and Section 9 concludes.

This paper is intended to be self-contained, so that it can be read independently of its predecessor [6]. However, a reader may be interested to know what the challenges were in handling the integer domain in constrast with the well-founded domain, where the definitions or techniques are the same, when they differ and how. The paper is interspersed with special comments marked with the symbol $\mathbb{Z}$. These comments are meant to answer the above questions. They can be skipped without compromising the integrity of the text.

The central results of this work have been presented in the 20th International Conference on Computer Aided Verification conference (2009) and are stated, very briefly, in the proceedings [5] .

2. Basic Definitions and Motivating Examples

$\mathbb{Z}$: The basic definitions (Sect. 2.1–2.2) are essentially as in [6], except for the notation π-termination, introduced for distinction between the integer interpretation of an MCS and the well-founded interpretation. Section 2.3 includes examples to show the expressiveness of the integer MCS model and contrast it with the usage of SCT to analyze the same programs.

2.1. **Monotonicity constraint transition systems.**

A monotonicity constraint transition system is an abstract program. An abstract program is, essentially, a set of *abstract transitions*. An abstract transition is a relation on (abstract) program states. When describing program transitions, it is customary to mark the variables in the *resulting* state with primes (e.g., x'). For simplicity, we will name the variables $x_1, \dots, x_n$ (regardless of what program variables they represent). This notation also suggests that the same number of variables (n) is used to represent all states. Of course, in actual programs this is not necessarily the case (for example due to different scopes) and it will be more efficient to maintain only the necessary variables at each program point.

Definition 2.1 (MCS)**.** A *monotonicity constraint transition system*, or MCS, is an abstract program representation that 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.*
- A *monotonicity constraint* (MC) is a conjunction of order constraints $x \bowtie y$ where $x, y \in \{x_1, \dots, x_n, x'_1, \dots, x'_n\}$, and $\bowtie \in \{>, \geq, =\}$.
- Every CFG arc $f \to g$ is associated with a monotonicity constraint G. We write $G : f \to g$.
- For each $f \in F$, there is an *invariant* I_f, which is a conjunction of order constraints among the variables.

The terms "abstract program" and "MCS" are used interchangeably, when context permits. The letter $\mathcal{A}$ is usually used to denote such a program; $F^{\mathcal{A}}$ will be its flow-point set. When notions of connectivity are applied to $\mathcal{A}$ (such as, "$\mathcal{A}$ is strongly connected"), they concern the underlying CFG.

2.2. **Semantics.**

Definition 2.2 (states)**.** A *state* of $\mathcal{A}$ (or an abstract state) is $s = (f, \sigma)$, where $f \in F^{\mathcal{A}}$ and $\sigma : \{1, \dots, n\} \to \mathbb{Z}$ represents an assignment of values to the variables.

Satisfaction of a predicate e with free variables $x_1, \dots, x_n$ (for example, $x_1 > x_2$) by an assignment σ is defined in the natural way, and expressed by $\sigma \models e$. If e is a predicate involving the $2n$ variables $x_1, \dots, x_n, x'_1, \dots, x'_n$, we write $\sigma, \sigma' \models e$ when e is satisfied by setting the unprimed variables according to σ and the primed ones according to σ'.

Definition 2.3 (transitions)**.** A transition is a pair of states, a *source state* s and a *target state* s'. For $G : f \to g \in \mathcal{A}$, we write $(f, \sigma), (g, \sigma') \models G$ if $\sigma \models I_f$, $\sigma' \models I_g$ and $\sigma, \sigma' \models G$. We say that transition $s \mapsto s'$ satisfies G.

Note that we may have unsatisfiable MCs, such as $x_1 > x_2 \wedge x_2 > x_1$; it is useful to view all such MCs as synonyms and use the common notation $\bot$ for them (as one would typically do in an Abstract Interpretation domain).

Definition 2.4 (transition system)**.** The *transition system* associated with $\mathcal{A}$ is the binary relation

$$T_{\mathcal{A}} = \{(s, s') \mid s, s' \models G \text{ for some } G \in \mathcal{A}\}.$$

Note that a program representation may also be called a "transition system." I am using the unqualified term for a semantic object. The program representation is referred to as an MC transition system (or sometimes just MC system).

Definition 2.5 (run)**.** A *run* of $\mathcal{T}_{\mathcal{A}}$ is a (finite or infinite) sequence of states $\tilde{s} = s_0, s_1, s_2 \dots$ such that for all i, $(s_i, s_{i+1}) \in \mathcal{T}_{\mathcal{A}}$.

Note that by the definition of $\mathcal{T}_{\mathcal{A}}$, a run is associated with a sequence of CFG arcs labeled by $G_1, G_2, \dots$ where $s_{i-1}, s_i \models G_i$. This sequence constitutes a (possibly non-simple) path in the CFG.

Definition 2.6 (termination)**.** Transition system $\mathcal{T}_{\mathcal{A}}$ is *uniformly terminating* if it has no infinite run. MCS $\mathcal{A}$ is said to be π*-terminating* if $\mathcal{T}_{\mathcal{A}}$ is uniformly terminating.

The prefix π (a symbol for the order type of the integers [26]) is included since the same MCS can be interpreted in the well-founded model of [6] and may possibly be non-terminating there (though this would require the value domain to be of an order type greater than ω). The term *uniform* refers to termination that is independent of the initial state. In Section 7 we discuss the consequences of specifying an initial flow-point.

Remark. It may be tempting to abstract away from the integers and, just as the well-founded case was treated generally (so that any well-founded domain could be used), give a general definition of the property that the domain has to satisfy for our termination arguments to apply. The property is that for any two elements there are only finitely many elements strictly between them. However, this abstraction buys us no generality, as every total order with this property is isomorphic to a subset of the integers. On the other hand, the fact that we assume the underlying set to be, specifically, the integers, has the satisfactory outcome that *completeness* of our deduction means that specific knowledge about the integers could not have been used in any better way.

2.3. **Examples.** Here are a few small examples that illustrate the transformation of programs into MCS. The reader is invited to verify that the abstract programs created are indeed terminating. To contrast the expressiveness of the MC abstraction with that of the better-known SCT, comments are included regarding how the examples might be treated by SCT-based tools. Recall that in the SCT framework, only well-founded domains are handled.

Example 2.7. Consider the following program, in a self-explanatory functional programming language with integer data.

```
f(m,n) = if m <= 0 then n
    else if n <= 0 then f(m-1, n-1)
                   else f(m, n-1)
```

In this example the recursive calls depend on some variable being positive, and therefore the proof can somehow be embedded in the well-founded framework. This requires placing the control-flow points at the call sites, where we can rely on the guards; it is also necessary to deal with the fact that n, while being pertinent to the termination proof, is only known to be positive (and thus can be admitted into the SCT abstraction) in the second call. When using MCs, we need not worry about all of this. We can place a single flow-point at the function entry (as done in the simple-minded abstraction of [23]). When analysing the body of the function, we create an abstract transition for each recursive branch, taking into account the branch condition. The abstract variables will be m, n and 0 (the analyser should recognize that these three quantities are involved in the comparisons). The resulting MCS thus consists of two abstract transitions:

$$G_1 : \quad \mathtt{m} > \mathtt{0} \land \mathtt{n} > \mathtt{0} \land \mathtt{n} > \mathtt{n}' \land \mathtt{m} = \mathtt{m}' \land \mathtt{0} = \mathtt{0}'$$

$$G_2 : \quad \mathtt{m} > \mathtt{0} \land \mathtt{n} \leq \mathtt{0} \land \mathtt{n} > \mathtt{n}' \land \mathtt{m} > \mathtt{m}' \land \mathtt{0} = \mathtt{0}'$$

Example 2.8. This simple example involves descent in a difference.

```
s(m,n) = if m > n then 1 + s(m-1, n)
                  else 0
```

Here, it is possible to use the SCT framework by placing the control-flow point at the call site, where we know that the difference $\mathtt{m} - \mathtt{n}$ is positive, and using the difference as the abstract variable. With an MCS, we can use a straight-forward translation:

$$G_1 : \quad \mathtt{m} > \mathtt{n} \land \mathtt{n} = \mathtt{n}' \land \mathtt{m} > \mathtt{m}'$$

As noted, both of the above examples could be proven terminating by SCT using an abstraction that relies on invariants and judicious placement of the flow-points; such techniques have been implemented, for example, in the theorem prover ACL2 [27]. The next programs challenge to the simple invariant-based technique, and therefore illustrate more significantly the advantage of monotonicity constraints.

Example 2.9. Consider the following program:

```
g(m,n) = if m <= 0 then n
                   else g(n, m-1)
```

Both parameters are pertinent to termination, but only m is known to be positive at the call site. But as $\mathtt{m}'$ is unrelated to m, this does not suffice to prove termination. Neither is the difference $\mathtt{n} - \mathtt{m}$ useful as an abstract variable, since there is no information on its sign and it does not change monotonically. Nonetheless, it is not hard to prove termination with the straight-forward MCS abstraction:

$$G_1 : \quad \mathtt{m} > 0 \land \mathtt{n} = \mathtt{m}' \land \mathtt{m} > \mathtt{n}'$$

Example 2.10. In this example, unsatisfiability of certain paths is crucial to the termination proof.

```
while (0<x<n)
   if b then
    x := x+1
   else
    x := x-1
```

A way to handle this program with SCT might be the calling-context method implemented in ACL2; the theorem prover is used to discover the fact that an increment cannot be followed by a decrement (or vice versa). It produces an abstract program into which this information is already coded. It also produces the abstract variable $\mathtt{n}-\mathtt{x}$. In contrast, with monotonicity constraints, a naïve translation of the program is sufficient. Since only the integer domain is treated in this work, we represent the Boolean variable b as an integer with the test interpreted as b>0. This yields:

$$G_1 : \quad 0 < \mathtt{x} \land \mathtt{x} < \mathtt{n} \land \mathtt{b} > 0 \land \mathtt{x} < \mathtt{x}' \land \mathtt{n} = \mathtt{n}' \land \mathtt{b} = \mathtt{b}' \land 0 = 0'$$

$$G_2 : \quad 0 < \mathtt{x} \land \mathtt{x} < \mathtt{n} \land \mathtt{b} \le 0 \land \mathtt{x} > \mathtt{x}' \land \mathtt{n} = \mathtt{n}' \land \mathtt{b} = \mathtt{b}' \land 0 = 0'$$

Example 2.11. Consider the following program.

```
g(m,n) = if m <= 0 then n
                   else g(m+n, n-1)
```

The apparent difficulty here is that the sign of n is not known at the point where it is added to m. Hence, one cannot adequately represent the effect of such an update by a monotonicity constraint. A similar problem arises with other operations (notably subtraction). However, this nut is easily cracked, since one *can* represent the effect

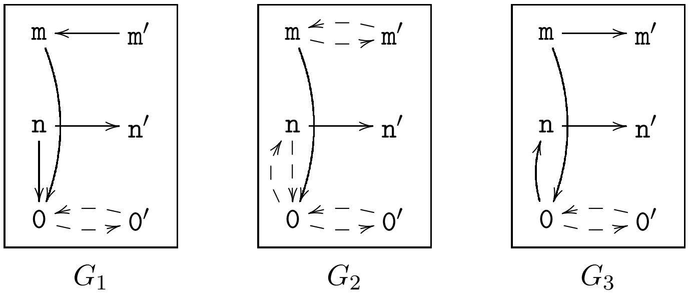


Figure 1: MCs of Example 2.11 as graphs. The left-hand side is the source. Broken arcs represent non-strict descent.

of addition as a disjunction of three monotonicity constraints. Thus, for the above program, we have the following (terminating) abstraction:

$$\begin{aligned} G_1: &\quad \mathtt{m} > 0 \wedge \mathtt{n} > 0 \wedge \mathtt{m} < \mathtt{m}' \wedge \mathtt{n} > \mathtt{n}' \wedge 0 = 0' \\ G_2: &\quad \mathtt{m} > 0 \wedge \mathtt{n} = 0 \wedge \mathtt{m} = \mathtt{m}' \wedge \mathtt{n} > \mathtt{n}' \wedge 0 = 0' \\ G_3: &\quad \mathtt{m} > 0 \wedge \mathtt{n} < 0 \wedge \mathtt{m} > \mathtt{m}' \wedge \mathtt{n} > \mathtt{n}' \wedge 0 = 0' \end{aligned}$$

Note that when n is initially positive, the first parameter grows at first, until n reaches zero and then the first parameter begins to shrink. Programs with such "phase shift" have attracted the attention of termination researchers, and appear in several publications, e.g., [11].

2.4. **A comment regarding state invariants.** None of the above examples used state invariants (associated with a flow-point rather than a transition), and in fact it is easy to see that one can always do without them, as it is possible to include the constraints I_f in every MC that transitions from f. However, it may be convenient to make the association of certain assertions with a flow point, rather than a specific transition, explicit, and our algorithms make significant use of such invariants.

$\mathbb{Z}$: The next two sections deal, respectively, with the representation and transformation of MCSs; the ideas are not essentially different from those used in [6] but there are some new details which are important for the sequel.

2.5. **Weighted graph representation.** An MC can be represented by a labeled digraph (directed graph). This representation enables a style of reasoning, using graph properties like paths, which has been very useful in most, if not all, previous work on SCT and monotonicity constraints. An arc $x \xrightarrow{r} y$ (r is the label) represents a relation $x > y$ or $x \geq y$. One only needs two labels, $>$ and $\geq$. It is convenient to use integers for the labels, and apply techniques from the world of weighted graphs. Hence, we employ the term *weighted graph representation.*

Definition 2.12. The weighted graph representation of a monotonicity constraint is a weighted digraph with node set $\{x_1, \dots, x_n, x'_1, \dots, x'_n\}$ and for each constraint $x > y$ (respectively $x \geq y$), an arc $x \xrightarrow{-1} y$ ($x \overset{0}{\to} y'$). The arcs are referred to, verbally, as *strict* (label -1) or *non-strict* (label 0).

The notation $x \to y$ may be used to represent an arc from x to y (of unspecified label). In diagrams, to avoid clutter, we distinguish the types of arcs by using a dashed arrow for the weak inequalities (see Figure 1). Note that an equality constraint $x = y$ is represented by a pair of non-strict arcs. In certain algorithms, it is convenient to assume that the such arcs are distinguished from "ordinary" non-strict arcs. We refer to them as *no-change arcs*.

Definition 2.13 ($\vdash$)**.** Let $P(s, s')$ be any predicate over states s, s', possibly written using variable names, e.g., $x_1 > x_2 \wedge x_2 < x'_2$. We write $G \vdash P$ if $\forall s, s' : s, s' \models G \Rightarrow P(s, s')$.

Definition 2.14. A monotonicity constraint G, in graph representation, is *closed under logical consequence* (or just closed) if, whenever $G \vdash x > y$, or $G \vdash x \geq y$, for $x, y \in \{x_1, \dots, x_n, x'_1, \dots, x'_n\}$, the stronger of the implied relations is explicitly included in the graph.

Note that for $G : f \to g$, the condition $G \vdash P$ takes the invariants I_f and I_g into account (consider Definitions 2.3 and 2.13). Thus, a closed MC subsumes the invariants in its source and target states.

Definition 2.15. An MC H *is at least as strong as* G if whenever $G \vdash P$, also $H \vdash P$. The (consequence) *closure* of a monotonicity constraint G, denoted $\overline{G}$, is the weakest MC that is at least as strong as G and is consequence-closed.

The closure of a graph be computed efficiently ($O(n^3)$ time) by a DFS-based algorithm. First the graph is divided in strongly connected components. Within every component, only non-strict arcs should appear; otherwise the graph is unsatisfiable and should be immediately replaced by the fixed object $\bot$. Otherwise, the components represent groups of variables constrained to be equal, and the acyclic graph of components can be processed to determine the constraints relating each pair.

Henceforth, we identify a MC with its consequence-closed graph representation, and assume that this is how our algorithms will maintain them. In graph representation, the restriction of the SCT framework is that only arcs of the form $x \to y'$ are admitted. Graphs of this form are called *size-change graphs* in the SCT literature and so will they be called in this paper.

2.6. **Transforming MC systems.** A key tool in processing MC systems is the idea of transforming them for the purposes of analysis. We employ two kinds of transformations: the first kindcreates a new system that is "equivalent" in the sense that they have the same runs, up to renaming of flow-points and possibly variables—the precise notion is *bisimulation*, defined below. The second kind of transformation creates a system that represents *part* of the runs of the given one—see the definition of *restriction* below. These definitions are given here for later reference. The first (bisimulation) is cited verbatim from [6].

Definition 2.16. Let $\mathcal{A}$, $\mathcal{B}$ be transition systems, with flow-point sets $F^{\mathcal{A}}$, $F^{\mathcal{B}}$ respectively, and both having states described by n variables. We say that $\mathcal{A}$ *simulates* $\mathcal{B}$ if there is a relation $\phi \subseteq F^{\mathcal{B}} \times F^{\mathcal{A}}$ ("correspondence of flow-points") and, for all $(f, g) \in \phi$, a bijection $\psi_{g,f} : \{1, \dots, n\} \to \{1, \dots, n\}$ ("variable renaming") such that for every (finite or infinite) state-transition sequence $(f_1, \sigma_1) \mapsto (f_2, \sigma_2) \mapsto (f_3, \sigma_3) \mapsto \dots$ of $\mathcal{B}$ there is a corresponding sequence $(g_1, \sigma'_1) \mapsto (g_2, \sigma'_2) \mapsto (g_3, \sigma'_3) \mapsto \dots$ of $\mathcal{A}$ with $(f_i, g_i) \in \phi$ and $\sigma'_i = \sigma_i \circ (\psi_{g_i,f_i})$. We say that $\mathcal{A}$ *bisimulates* $\mathcal{B}$ if, in addition, for every (finite or infinite) state-transition sequence $(g_1, \sigma'_1) \mapsto (g_2, \sigma'_2) \mapsto (g_3, \sigma'_3) \mapsto \dots$ of $\mathcal{A}$ there is a corresponding sequence $(f_1, \sigma_1) \mapsto (f_2, \sigma_2) \mapsto (f_3, \sigma_3) \mapsto \dots$ of $\mathcal{B}$, also with $(f_i, g_i) \in \phi$ and $\sigma'_i = \sigma_i \circ (\psi_{g_i,f_i})$.

Thus, $\mathcal{A}$ bisimulates $\mathcal{B}$ if they simulate each other via the same pair of mappings.

Definition 2.17. We say that an abstract program $\mathcal{A}$ (bi-)simulates an abstract program $\mathcal{B}$ if $\mathcal{T}_{\mathcal{A}}$ (bi-)simulates $\mathcal{T}_{\mathcal{B}}$, via mappings ϕ and ψ, as above.

We say that $\mathcal{A}$ simulates $\mathcal{B}$ *deterministically* if for every $f \in F^{\mathcal{B}}$ and assignment σ satisfying I_f there is a unique $g \in F^{\mathcal{A}}$ with $(f, g) \in \phi$ such that, letting $\sigma' = \sigma \circ (\psi_{g,f})$, assignment σ' satisfies I_g.

If $\mathcal{A}$ bisimulates $\mathcal{B}$, and $\mathcal{A}$ simulates $\mathcal{B}$ deterministically, we say (for brevity) that $\mathcal{A}$ bisimulates $\mathcal{B}$ deterministically.

Determinism means that the invariants of different $\mathcal{A}$ flow-points that simulate a given $\mathcal{B}$ flow-point have to be mutually exclusive.

Definition 2.18. The notation $\mathcal{A} \sim_{\phi,\psi} \mathcal{B}$ means that $\mathcal{A}$ simulates $\mathcal{B}$ deterministically via flow-point correspondence ϕ and the variable renaming function ϕ. We omit ϕ when it is the identity, and omit both if it is not important to specify them.

Definition 2.19. Let $\mathcal{A}$, $\mathcal{B}$ be transition systems, with same flow-point sets $F^{\mathcal{A}} = F^{\mathcal{B}}$, and both having states described by n variables. We say that $\mathcal{A}$ *is a restriction of* $\mathcal{B}$, and write $\mathcal{A} \Subset \mathcal{B}$, if every (finite or infinite) state-transition sequence $(f_1, \sigma_1) \mapsto (f_2, \sigma_2) \mapsto (f_3, \sigma_3) \mapsto \dots$ of $\mathcal{B}$ is also a transition sequence of $\mathcal{A}$.

A restriction results, obviously, from tightening the constraints at certain flow-points or transitions.

Definition 2.20. We write $\mathcal{A} \Subset_{\phi} \mathcal{B}$, if there exists $\mathcal{C}$ such that $\mathcal{A} \Subset \mathcal{C}$ and $\mathcal{C} \sim_{\phi} \mathcal{B}$.

3. Stable Systems

The notion of *stable* MC systems is from [6]. This section recalls the definition and states some consequences of stability that are used in forthcoming proofs.

$\mathbb{Z}$: There is no novelty here.

Definition 3.1. An MCS $\mathcal{A}$ is *stable* if (1) all MCs in $\mathcal{A}$ are satisfiable; (2) For all $G : f \to g$ in $\mathcal{A}$, whenever $G \vdash x_i \rhd x_j$ (some relation between source variables), also $I_f \vdash x_i \rhd x_j$. (4) Similarly, if $G \vdash x'_i \rhd x'_j$, also $I_g \vdash x_i \rhd x_j$.

Note that while consequence-closure requires that all information from I_f and I_g be present in G, stability requires that G cannot add information to I_f and I_g.

Stabilizing a given system may require flow-points to be duplicated, since two MCs coming out of f may disagree on the conditions that must be placed in I_f.

It is always possible to tranform an MC system $\mathcal{B}$ into a stable $\mathcal{A}$ such that $\mathcal{A} \sim \mathcal{B}$; this transformation is called *stabilization*, and an algorithm is described in [6]. Stabilization does not rename variables, but may require flow-points to be duplicated, since two MCs coming out of f may disagree on the conditions that must be placed in I_f. In the worst case, it can multiply the size of the system by a factor exponential in the number of variables n (specifically by the *Ordered Bell Number* B_n which is between $n!$ and $2^{n-1}n!$ [30, Seq. A670]). A brute-force solution, which always reaches the worst case, was also described. This solution turned out useful in constructing global ranking functions and will also be used for this purpose in this paper.

3.1. **Properties of stable systems.**

Lemma 3.2. *Let $M = G_1 G_2 \dots G_\ell$ be a finite multipath of a stable MCS. Suppose that there is a path in M from $x[0, s]$ to $x[\ell, t]$. Then there is a thread with such endpoints.*

The lemma works for down-paths (and down-threads) as well as for up-paths (and up-threads), which can be seen (for more convenient argument) as paths in the transposed graph.

Proof. We consider the shortest path among the given endpoints. It has to be a thread, for otherwise it has two consecutive arcs among the nodes of a single graph G_i. Stability and consequence-closure of the graphs imply that these two arcs can always be replaced by a single one, contradicting the choice of a shortest path. □

Lemma 3.3. *Let $M = G_1 G_2 \dots G_\ell$ be a finite multipath of a stable MCS. Suppose that there is a path in M from $x[t, i]$ to $x[t, j]$ for some t, i, j with $i \neq j$. Then there is an arc with such endpoints.*

The argument is very similar to the previous one and we omit a detailed proof.

Lemma 3.4. *In a stable MCS, every finite multipath is satisfiable.*

The proof, again, is rather similar to the previous ones and is given in full in [6].

4. Termination

A central contribution of [23] was the definition of a "path based" termination condition for systems of size-change graphs, dubbed *the SCT criterion.* The condition hinges on the presence of certain *infinite paths* in infinite sequences of size-change graphs (such sequences represent hypothetic execution histories). This criterion was generalized in [6] to monotonicity constraints, where the paths have become walks (possibly cyclic). It has also been shown that assuming stability, the condition reduces to the SCT criterion, which is easier to reason about as well as to test for.

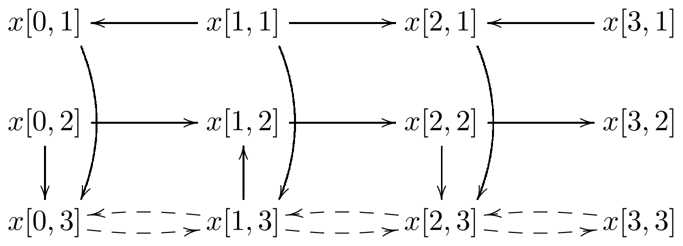


Figure 2: A multipath.

In this section we prove that for π-termination too, termination is captured by a path-based criterion. We consider in particular the case of a stable system, but also the general case. This yields a proof that π-termination is decidable in PSPACE.

$\mathbb{Z}$: The path-based criterion, and consequently the algorithm, enhance the criteria and algorithm of [6] in a way which seems, at least after the fact, very natural. The completeness proof was the challenging part as it differs significantly from [6]; it is inspired by [12]. This section also includes a discussion of the algorithm proposed in [12].

4.1. **Some definitions.**

Definition 4.1 (multipath)**.** Let $\mathcal{A}$ be an n-variable MCS, and let $f_0 \overset{G_1}{\to} f_1 \overset{G_2}{\to} f_2 \ldots$ be an MC-labeled path in the CFG (either finite or infinite). 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 $1 \le i \le n$. Its arcs are obtained by merging the following sets: for all $t \ge 1$, 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 $G_1G_2\ldots$; if M_1, M_2 are finite multipaths, M_1 corresponding 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 $(G)^x$ represent a multipath made of x copies of G.

Figure 2 depicts multipath $G_1G_3G_1$, based on the MCs from Figure 1.

Clearly, a multipath can be interpreted as a conjunction of constraints on a set of variables associated with its nodes. We consider assignments σ to these variables, where the value assigned to $x[t,i]$ are denoted $\sigma[t,i]$. A multipath can be seen as an execution trace of the abstract program, whereas a satisfying assignment constitutes a (concrete) run of $\mathcal{T}_{\mathcal{A}}$. Conversely: every run of $\mathcal{T}_{\mathcal{A}}$ constitutes a satisfying assignment to the corresponding multipath.

Definition 4.2. A path in the graph representation of an MC or a multipath is *strict* if it includes a strict arc.

Observation 4.3. A finite multipath is satisfiable if and only if it does not contain a strict cycle.

We next define down-paths and up-paths. The definition of a down-path is just the standard definition of path (which has already been used in this paper) but it is renamed in order to accommodate the notion of an up-path.

Definition 4.4. A *down-path* in a graph is a sequence $(v_0, e_1, v_1, e_2, v_2, \dots)$ 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 $(v_0, e_1, v_1, e_2, v_2, \dots)$ 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 is a down-path in the *transposed* graph.

Definition 4.5. Let $M = G_1 G_2 \dots$ be a multipath. A *down-thread* in M is a down-path that only includes arcs in a forward direction $(x[t,i] \to x[t+1,j])$.

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

A *thread* is either.

4.2. **Combinatorial criteria for π-termination.**

We formulate necessary and sufficient conditions for the termination of an MCS in terms of its multipaths. The first criterion is called *Condition S* as it assumes a stable MCS. The second, *Condition G*, works with any MCS.

4.2.1. *The Stable Case.*

Definition 4.6. A stable MCS $\mathcal{A}$ satisfies *Condition S* if in any infinite multipath M (with variables $x[t,i]$) there are an infinite down-thread $(x[k,h_k])_{k=k_0,k_0+1,\dots}$ and an up-thread $(x[k,l_k])_{k=k_0,k_0+1,\dots}$, such that all the constraints $x[k,l_k] \le x[k,h_k]$ are present in M. In addition, at least one of the threads has infinitely many strict arcs.

We may later refer to the up-thread $(x[k,l_k])$ as the *low thread*, while the down-thread $(x[k,h_k])$ is the *high thread*. This naming is meant to stress the relation $x[k,l_k] \le x[k,h_k]$: the values assumed by the variables on the low thread are always lower than those assumed on the high thread. The fact that the values in the high thread descend, while those in the low thread ascend, suggests that the process cannot go on forever, which is what we want. A down-thread/up-thread pair satisfying this condition will be called *an approaching pair*. If the condition of infinitely many strict arcs is not guaranteed, we will use the term *a weakly approaching pair*.

Theorem 4.7. *Condition* S *is a sufficient condition for π-termination.*

Proof. Suppose that Condition S is satisfied and that $\mathcal{A}$ does not terminate. Thus, some infinite multipath M is satisfiable. The threads postulated in Condition S imply that the sequence of differences $\sigma[k,h_k] - \sigma[k,l_k]$ is infinitely descending, while consisting of non-negative integers. This is impossible, so we conclude that $\mathcal{A}$ is π-terminating. □

Condition S is also a necessary condition for termination of stable systems (i.e., the criterion is sound and complete), but we will prove this later.

Readers familiar with SCT will surely notice that if, for each flow-point f, for every pair of variables such that $I_f \vdash x_i \leq x_j$, we create a variable $x_{(i,j)}$ to represent $x_j - x_i$ (guaranteed to be non-negative) and we connect such variables with the obvious size-change arcs (if $x_i \leq x'_l$ and $x_j \geq x'_h$ then $x_{(i,j)} \geq x'_{(l,h)}$, etc), Condition S becomes equivalent to ordinary (well-founded) size-change termination in the new variables. The SCT condition requires every infinite multipath to include an infinitely-descending thread (a down-thread which is infinitely often strict). Our approaching pairs correspond precisely to threads in the difference variables.

Thus, given a stable system, π-termination reduces to SCT, and this immediately provides us with a decision algorithm. Nevertheless, we dedicate the next couple of subsections to new decision algorithms, based directly on Condition S. The motivation for doing so is threefold. First, it is theoretically interesting to see how such a direct algorithm would work. Secondly, the SCT algorithms are exponential in the number of variables. Creating difference variables in advance squares the exponent, which is bad. Finally, we shall employ one version of the direct algorithm in proving the *completeness* of our termination criterion.

4.2.2. *The General Case.* Dropping the stability does not complicate the statement of the termination condition very much. Instead of threads, we have to consider paths of any form (even such that repeat arcs).

Definition 4.8. MCS $\mathcal{A}$ satisfies *Condition G* if in any infinite multipath M (with variables denoted by $x[i,j]$) there is an up-path $(x[L_j, l_j])_{j=0,1,\dots}$ and a down-path $(x[H_j, h_j])_{j=0,1,\dots}$ such that at least one of the paths has infinitely many strict arcs. Moreover, for infinitely many values of k, there is a path from $x[H_k, h_k] \leq x[L_k, l_k]$.

Theorem 4.9. *Condition* G *is a sound condition for $\mathbb{Z}$-termination.*

Proof. Suppose that Condition G is satisfied and that some infinite multipath M is satisfiable. Then the paths described in Condition G exist and imply that the sequence of differences $x[H_j, h_j] - x[L_j, l_j]$ is infinitely descending, while consisting of non-negative integers (note that if $x[L_k, l_k] \leq x[H_k, h_k]$, then also for all $j \leq k$, $x[L_j, l_j] \leq x[H_j, h_j]$). This is impossible, hence no infinite multipath is satisfiable, or equivalently, the MCS $\mathbb{Z}$-terminates. □

Theorem 4.10. *Conditions* G *and* S *are equivalent for a stable system.*

Proof. Condition S is a special caes of Condition G, so all we have to prove is that if Condition G is satisified, so is Condition S. Suppose, then, that Condition G holds, and we have the paths $(x[L_j, l_j])_{j=0,1,\dots}$ and $(x[H_j, h_j])_{j=0,1,\dots}$, as above. It is easy to see that since for infinitely many values of k, there is a path from $x[H_k, h_k]$ to $x[L_k, l_k]$, it is also true that for *any* k and j, there is a path from $x[H_k, h_k]$ to $x[L_j, l_j]$ (the path may use parts of the up-path and down-path themselves).

To each path, we can apply Lemma 3.2 to obtain a *thread* whose nodes are a subset of the nodes of the path; an up-thread $(x[t, l_{j_t}])_{t \geq t_0}$ and a down-thread $(x[t, h_{k_t}])_{t \geq t_0}$. At least one of the threads is infinitely often strict.

From the previous observation, we know that there is, for every t, a path from $x[t, h_{k_t}]$ to $x[t, l_{j_t}]$. Applying Lemma 3.3, we cam show that the relation $x[t, h_{k_t}] \geq x[t, l_{j_t}]$ must be included in the invariant for the corresponding flow-point. □

Condition G is also a complete criterion for termination; the proof is based on stabilizing the system and the equivalence of Conditions G with Condition S in the stable one. The details are omitted as they are tedious and give no new insight.

4.3. **A Closure Algorithm for Stable Systems.**

We next present an algorithm to decide termination of stable Integer MCS by computing a composition-closure and applying a certain test to the elements of the closure set. This algorithm is based on the Closure Algorithm for the well-founded model [24, 13, 17, 23, 6], and is theoretically related to the Disjunctive Well-Foundedness principle [19]. An algorithm *sans* stability is presented later for completeness; but our analysis of the algorithms and their correctness makes substantial use of stability (note the previous paragraph).

The following definitions are essentially from [6]:

Definition 4.11 (composition)**.** The *composition* of MC $G_1 : f \to g$ with $G_2 : g \to h$, written $G_1; G_2$, is a MC with source f and target h, which includes all the constraints among s, s' implied by $\exists s'' : s, s'' \models G_1 \wedge s'', s' \models G_2$.

Composition is similar to logical closure and can be implemented by a DFS-based algorithm in $O(n^3)$ time.

Definition 4.12 (collapse)**.** For a finite multipath $M = G_1 \dots G_\ell$, Let $\overline{M} = \overline{G_1; \cdots ; G_\ell}$. This is called the *collapse* of M (if $\ell = 1$, it is just the consequence-closure of G_1).

Applying composition together with logical closure, we can easily compute of $\overline{M}$ for a given M.

Definition 4.13. Given an MCS $\mathcal{A}$, its closure set $cl(\mathcal{A})$ is

$$\{\overline{M} \mid M \text{ is a satisfiable finite } \mathcal{A}\text{-multipath}\}.$$

The closure set can be computed by a routine least-fixed-point procedure, as in [5].

Definition 4.14 (cyclic MC)**.** We say that a MC G is *cyclic* if its source and target flow-points are equal. This is equivalent to stating that GG is a valid multipath.

Definition 4.15 (circular variant)**.** For a cyclic G, the *circular variant* G° is a weighted graph obtained by adding, for every variable x_i, an edge $x_i \leftrightarrow x'_i$. This edge is treated as a pair of no-change arcs, but is distinguished from edges already present in G. These additional edges are called *shortcut edges*.

Definition 4.16 (types of cycles)**.** Let G be a cyclic MC. A cycle in G° is a path commencing and ending at the same node. It is a *forward cycle* if it traverses shortcut edges only in the backward direction (from x'_i to x_i)[2]. It is a *backward* cycle if it traverses shortcut edges only in the forward direction.

[2] this naming may seem strange, but will will later see that such a cycle is "unwound" into a forward-going thread in the multipath $(G)^\omega$.

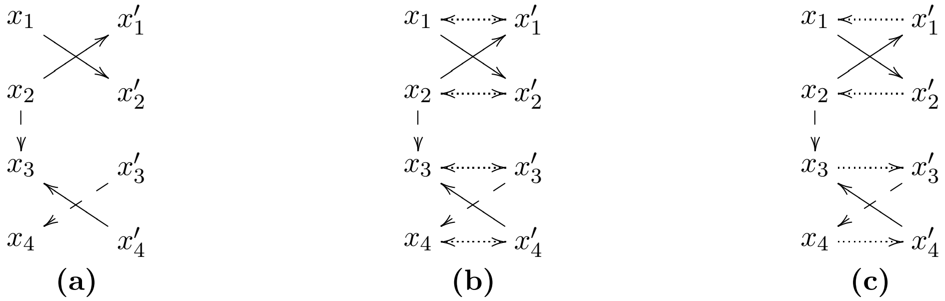


Figure 3: (a) An MC G, (b) its circular variant G°; (c) the directions of shortcut edges are set to highlight the two cycles required by the LTTS.

Definition 4.17 (Local π-Termination Test for Stable Systems)**.** For a cyclic G, we say that G passes the *Local π-Termination Test for Stable Systems*, or LTTS, if G° includes a forward cycle F and a backward cycle B, at least one of which is strict, and an arc from a node of F to a node of B.

This test resembles Sagiv's local test for the well-founded case [24], and likewise can be implemented as a DFS-based algorithm in linear time, on which we will not elaborate. Figure 3 illustrates the test.

Algorithm 4.18. (The Closure Algorithm)

(1) Build $cl(\mathcal{A})$.
(2) For each cyclic G in $cl(\mathcal{A})$, apply the LTTS.
Pronounce failure (non-termination) if a graph that fails the test is found.
(3) If the previous step has completed, the MCS terminates.

Theorem 4.19 (closure algorithm—soundness)**.** *Let $\mathcal{A}$ be a stable MCS. If every cyclic MC in $cl(\mathcal{A})$ passes the Local π-Termination Test for Stable Systems, $\mathcal{A}$ satisfies Condition* S*.*

The proof will use the next lemma.

Lemma 4.20. *Consider an infinite multipath M of a stable system, represented as the concatenation of finite segments $M_1 M_2 \ldots$, and let $M' = (\overline{M}_1)(\overline{M}_2)\ldots$. Then M' satisfies Condition* S *if and only if M does.*

Proof. Suppose first that M' satisfies Condition S. The down-thread in M' can be mapped back to M, where it consists of variables $x[k_i, h_i]$, such that k_i is the length of $M_1 \ldots M_i$. The arc $x[i, h_i] \to x[i+1, h_{i+1}]$ in M' is actually an arc of $\overline{M}_{i+1}$, and reflects (by the definition of composition) a path in M_{i+1}. By Lemma 3.2, this path can be assumed to be a thread. We thus obtain an infinite down-thread that passes through the variables $x[k_i, h_i]$. Similarly, we obtain an infinite up-thread through the variables $x[k_i, l_i]$. Condition S also implies that the constraint $x[k_i, h_i] \geq x[k_i, l_i]$ is present in M for every i. For $k_{i-1} < k < k_i$, we have a down-thread from some $x[k, h]$ to $x[k_i, h_i]$, and an up-thread from some $x[k, l]$ to $x[k_i, l_i]$, so we have a path from $x[k, h]$ to $x[k, l]$. By Lemma 3.3, $x[k, h]$ and $x[k, l]$ must be explicitly related in M.

For the converse implication, we assume that M satisfies Condition S and we are to prove it for M': this is symmetric, but simpler than the above argument, and so is left to the interested reader. □

Proof. (of Theorem 4.19) We suppose that every cyclic MC in $cl(\mathcal{A})$ passes the LTTS. Let $\mathcal{M} = G_1 G_2 \ldots$ be any infinite $\mathcal{A}$-multipath.

Consider the set of positive integers, and label each pair (t, t') , where $t < t'$, by

$$G = G_t; G_{t+1}; \cdots G_{t'-1}$$

which is included in $cl(\mathcal{A})$, since this multipath is satisfiable (Lemma 3.4). By Ramsey's theorem (in its infinite version), there is an infinite set of positive integers, I, such that all pairs (t, t') with $t, t' \in I$ carry the same label G_I.

Thus for any $t, t' \in I$ with $t < t'$, $G_t; G_{t+1}; \cdots G_{t'-1} = G_I$. By Lemma 4.20, it now suffices to show that multipath $(G_I)^\omega$ (infinite sequence of G_I's) satisfies Condition S.

By assumption, G_I passes the LTTS. Let F be the forward cycle. Due to consequence-closure, we can shrink each segment of F that consists of ordinary arcs (not shortcuts) into a single arc, so that without loss of generatlity we may assume that the cycle alternates source (unprimed) with target (primed) variables. We can thus choose indices $h_1, h_2, \ldots$ such that the nodes of the cycle are x_{h_0}, x'_{h_1}, x_{h_1}, x'_{h_2} and so on up to x'_{h_s} for some $s > 0$, with $h_s = h_0$. We can do the same for the backward cycle B, denoting the variables by x_{l_j} and x'_{l_j}, for $0 \le j < \hat{s}$, where $\hat{s}$ is the length of that cycle. By LTTS, there is also an arc from a node of F to a node of B; suppose that they are both source nodes; then, without loss of generality (a cycle can be begun at any point), we can assume that $G \vdash x_{h_0} \ge x_{l_0}$. In fact, there is no loss of generality in assuming that the related nodes are source nodes, either; suppose, for example, that we have $G \vdash x'_{h_1} \ge x_{l_0}$; then $G \vdash x_{h_0} \ge x_{l_0}$ is implied. Other situations can be handled similarly.

To show that Condition S is satisfied, we map the cycles onto infinite threads in $(G_I)^\omega$. The kth node of the down-thread is $x[k, h_{k \bmod s}]$. The kth node of the up-thread is $x[k, l_{(\hat{s}-k) \bmod \hat{s}}]$ (indexing the backward cycle with $\hat{s} - k$ rather than k yields an up-thread). Either the infinite down-thread or the infinite up-thread (or both) is infinitely often strict.

According to our assumptions, for all q we have the constraint $x[qs\hat{s}, h_0] \ge x[qs\hat{s}, l_0]$. This implies $x[k, h_{k \bmod s}] \ge x[k, l_{k \bmod \hat{s}}]$ since we have the following path: traversing part of the up-thread from $x[k, h_{k \bmod s}]$ to $x[qs\hat{s}, h_0]$ for some q such that $qs\hat{s} \ge k$, then using $x[qs\hat{s}, h_0] \ge x[qs\hat{s}, l_0]$ which we have by assumption, and then traversing the down-thread in the reverse direction from $x[qs\hat{s}, l_0]$ to $x[k, l_{k \bmod \hat{s}}]$. By Lemma 3.3, the deduced constraint must explicitly appear in in the multipath.

We conclude that Condition S is satisfied by $(G_I)^\omega$. □

4.4. **The Role of Idempotence.**

We call G *idempotent* if G is cyclic and $G; G = G$. Following [12] (and similar results in [17, 23]), we claim:

(1) In the closure algorithm, it suffices to test only idempotent members of $cl(\mathcal{A})$.
(2) The local test becomes simpler with idempotent graphs.

The first claim is easy to justify by studying the argument in the soundness proof: the graph G_I whose existence is established there is clearly idempotent. For the second, let us describe the simple test and justify it.

Definition 4.21 (Local π-Termination Test for Idempotent MCs)**.** We say that G passes the *Local π-Termination Test for Idempotent MCs*, or LTT1, if it is idempotent, and for some $1 \le l, h \le n$, $G \vdash x_l \le x_h \wedge x_l \le x'_l \wedge x_h \ge x'_h$, where at least one of the last two inequalities is strict.

Lemma 4.22. *Let $\mathcal{A}$ be a stable MCS and G an idempotent, cyclic MC in $cl(\mathcal{A})$. Then G passes the LTTS if and only if it passes the LTT1.*

Proof. One direction of this equivalence is easy: if G passes the LTT1, it has the cycles $x_h \to x'_h \to x_h$ and $x_l \to x'_l \to x_l$, that satisfy the LTTS.

For the other direction, suppose that G passes the LTTS. Repeating the analysis in the proof of Theorem 4.19, and re-using the notation, we let $2s$ be the length of the forward cycle (assuming it alternates shortcut and ordinary arcs) and $2\hat{s}$ the length of backward cycle (under a similar assumption), and consider the multipath $\mathcal{M} = (G)^{s\hat{s}}$. It has a down-thread from $x[0, h_0]$ to $x[s\hat{s}, h_0]$, so $\overline{(G)^{s\hat{s}}} \vdash x_h \ge x'_h$. But by idempotence, $\overline{(G)^{s\hat{s}}} = G$. Similarly, we deduce that $G \vdash x_l \le x'_l$, that one of these relations is strict, and that $G \vdash x_l \le x_h$. Therefore, G passes the LTT1. □

We obtain the following version of the closure algorithm:

Algorithm 4.23. (Closure Algorithm with Idempotence)

(1) Build $cl(\mathcal{A})$.
(2) For each cyclic G in $cl(\mathcal{A})$, if it is idempotent apply the LTT1 to G. Pronounce failure (non-termination) if a graph that fails the test is found.
(3) If the previous step has completed, the MCS terminates.

Though this algorithm makes fewer local tests, it does not seem to be practically better than the previous, since the local test is quite efficient. On the contrary, applying the test to every graph in the closure allows for early discovery of failure. More importantly, it allows for reducing the size of the set by subsumption (consider [8, 18] and others).

4.5. **Complexity.** The closure algorithm, regardless of minor savings based on idempotency or subsumption, has an exponential worst-case for time and space. This follows from the fact that the closure set can be exponential. An easy upper bound is this: for any pair of flow points f, g, an MC relates the $2n$ variables in the source and target states. The number of such MCs is easily bounded by 3^{4n^2} as an MC is completely specified by choosing either $\le$, $<$ or "nothing" for every ordered pair of variables. The complexity of the whole data structure, and the time of the algorithm, are thus $\text{poly}(m) \cdot 2^{O(n^2)}$ where m is the number of flow-points[3].

In Section 5, a polynomial-space algorithm will be described, just to make the theoretical classification of the problem in PSPACE. Practically, the polynomial-space version is not attractive because its running time gets worse. On the other

[3] There are instances of size $O(n)$ that actually generate a closure of $2^{\Theta(n^2)}$ MCs. The exponent is thus not an overestimate.

hand, in Section 4 we show that the exponent in the running time can be brought down to $O(n \log n)$. But this, too, seems to be an improvement only in a highly theoretical sense.

4.6. **Completeness.** This section proves the completeness of Condition S, as well as the closure algorithm, by proving that if an idempotent MC in a stable system does not satisfy the LTTS, the system fails to terminate.

To show that the system does not terminate, we exhibit a satisfiable infinite multipath.

Lemma 4.24. *Let M be a finite, cyclic, satisfiable multipath and $G = \overline{M}$. If $(G)^\omega$ is satisfiable, so is $(M)^\omega$.*

Proof. Suppose that $(G)^\omega$ is satisfiable. That is, a satisfying assignment σ exists such that $\sigma[t,i]$ is the value of $x[t,i]$. Consider $(M)^\omega$. For distinction, let us denote its variables by $x'[\_,\_]$. Let ℓ be the length of M; the variables $x'[\ell t, i]$ represent the borders of the copies of M, and correspond to the source and target variables of G. Recall that i varies between 1 and n, the number of variables. Set $\sigma'[\ell t, i] = n(\ell+1)\sigma[t,i]$. We claim that σ' can be extended to a satisfying assignment for $(M)^\omega$. The construction is illustrated in Figure 4.

When considering the assignment to a particular copy of M, it is possible to ignore all others, since all nodes on its boundary are already assigned. Thus, suppose that we are given a copy of M, say the pth copy, with an assignment to its boundary nodes given by $\sigma'[p\ell, \_]$ and $\sigma'[(p{+}1)\ell, \_]$. Consider M as a weighted graph; the graph has no negative-weight cycles, or M would be unsatisfiable.

Let μ be the biggest value assumed by σ' over the boundary nodes of M. Add an auxiliary node z to M and arcs from z to all M's nodes, weighted as follows: an arc to a boundary node $x'[(p+j)\ell, i]$ is weighted by $\sigma'[(p+j)\ell, i]$. All other arcs have weight $\mu + n(\ell+1)$ (note that $n(\ell+1)$ bounds the length of any simple path in M). Let $\delta_z(j,i)$ be the weight of the lightest path from z to $x'[j,i]$. This is well-defined, due to the absence of negative cycles.

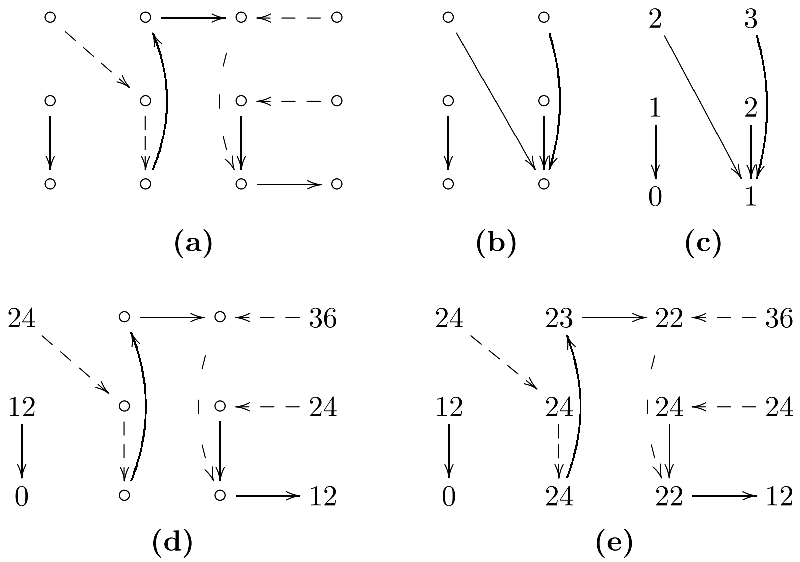
If there is an arc of weight w from $x'[j,i]$ to $x'[j',i']$, then $\delta_z(j',i') \le \delta_z(j,i) + w$. Recalling that non-strict arcs have a weight of 0 and strict arcs, of -1, it is easy to see that assigning $\delta_z(j,i)$ to $x'[j,i]$ is a satisfying assignment for M.

It rests to show that this assignment agrees with σ'. To this end, let $x[(p{+}j)\ell, i]$ be a boundary node. By construction, $\delta_z((p+j)\ell, i) \le \sigma'[(p+j)\ell, i]$. A strict inequality can only arise if there is a path from z to $x'[(p+j)\ell, i]$, lighter than the immediate arc. It can be assumed to be a simple path; it begins with an arc $z \to x'[r,s]$ and then rests within M. Let P be the segment within M; since it is simple, its weight w is bigger than $-n(\ell+1)$. Using this, one can easily eliminate the possibility that the path begins with an arc $z \to x'[r,s]$ where $x'[r,s]$ is not a boundary node. Suppose, then, that $x'[r,s]$ *is* a boundary node, and can be written as $x'[(p+k)\ell, s]$. Then, we have

$$\sigma'[(p+k)\ell, s] + w < \sigma'[(p+j)\ell, i]\,. \tag{4.1}$$

We now distinguish two cases. If $\sigma[p+k, s] > \sigma[p+j, i]$, we get (multiplying by $n(\ell+1)$):

$$\sigma'[(p+k)\ell, s] \ge n(\ell+1) + \sigma'[(p+j)\ell, i] \tag{4.2}$$



(a) (b) (c)

(d) (e)

Figure 4: An illustration for the proof of Lemma 4.23: (a) A multipath M; (b) $G = \overline{M}$; (c) G with an assignment σ; (d) M with its boundary values set (note that $n(\ell + 1) = 3 \cdot 4 = 12$); (e) the assignment to M completed by a lightest-path computation.

contradicting Inequality (4.1).

If $\sigma[p+k, s] = \sigma[p+j, i]$, there can be no strict path from $x[p+k, s]$ to $x[p+j, i]$, and hence no strict path from $x'[(p+k)\ell, s]$ to $x'[(p+j)\ell, i]$. Thus $w \geq 0$; and again a contradiction ensues.

If $\sigma[p+k, s] < \sigma[p+j, i]$, there can be no path at all from $x[p+k, s]$ to $x[p+j, i]$, and hence no path from $x'[(p+k)\ell, s]$ to $x'[(p+j)\ell, i]$.

We conclude that $\delta_z((p+j)\ell, i) = \sigma'[(p+j)\ell, i]$, so we have a satisfying assignment that extends σ', as desired. □

Lemma 4.25. *Let $\mathcal{A}$ be a stable MCS and $G \in cl(\mathcal{A})$ be an idempotent MC that does not pass the Local π-Termination Test for Stable Systems. Then $\mathcal{A}$ has a satisfiable, infinite multipath.*

Proof. We will show that $(G)^\omega$ is a satisfiable multipath. Then, the conclusion will follow by Lemma 4.24.

The first step is to add a new variable x_0 ("the zero line") with constraint $x_0 = x'_0$. Next, we make sure that every other variable is related to x_0. We do this as follows:

(1) For each x_i such that $G \vdash x_i < x'_i$, we add $x_i > x_0$ and $x'_i > x'_0$.
(2) For each x_i such that $G \vdash x_i > x'_i$, we add $x_i < x_0$ and $x'_i < x'_0$.
(3) If there is a variable x_i which is still unrelated to x_0, we add $x_i > x_0$ and $x'_i > x'_0$. This is repeated until all variables are related to x_0 or x'_0.

It is not hard to verify that G remains idempotent throughout this process. G also remains satisfiable, since the above additions are non-contradictory: It is easily seen that in the first stage, the additions cannot be contradictory—they cannot form a directed cycle; the additions in the second stage cannot be contradictory among themselves, and they cannot form a contradiction with the additions of the first stage, because, by assumption, we cannot have $x_i < x'_i$ in G together with $x_j > x'_j$ and $x_j \geq x_i$ (or $x'_j > x'_i$). In the third stage, it is obvious that no contradictions can arise.

Let $\widehat{G}$ denote the extended MC; we will show that $(\widehat{G})^\omega$ is satisfiable. Let $\mathcal{V}$ be the set of variables of $(\widehat{G})^\omega$. We define an assignment $\sigma : \mathcal{V} \to \mathbb{Z}$ as follows:

- For all t, $\sigma(x[t, 0]) = 0$. We shall refer to these variables collectively as $\mathbf{0}$ (note that they are all related by equalities so if a variable is related to one of them, it is related to all).
- For all $v \in \mathcal{V}$ such that there is a directed path from v to $\mathbf{0}$, let $\sigma(v)$ be the maximum number of strict arcs on such a path (note that only simple paths need be considered, since there are no strict cycles).
- For all $v \in \mathcal{V}$ such that there is a directed path from $\mathbf{0}$ to v, let $\sigma(v)$ be the negation of the maximum number of strict arcs on such a path.

Note that since every variable in G is related to x_0 (or x'_0), we have covered all cases. Why are the assignments well-defined? The only problem to worry about is the existence of a node v such that its distance from $\mathbf{0}$ is unbounded.

Suppose that such v exists and that there is a path from v to $\mathbf{0}$ (the other case, a path from $\mathbf{0}$ to v, is symmetric). Thus, there are infinitely many paths P_i from v to $\mathbf{0}$, such that P_i has at least i strict arcs. For each such path, we can assume that only one $\mathbf{0}$ appears on the path; otherwise we can cut the path at the first occurrence (there can be no strict arcs among occurrences of $\mathbf{0}$).

The pigeon-hole principle shows that for sufficiently large i, P_i must visit a certain state-variable x_j twice, say as $x[t, j]$ first and $x[t', j]$ later, with a strict arc in-between. Moreover, by choosing i big enough, we can enforce the condition $t' > t$. Then, by idempotence of $\widehat{G}$, we have $\widehat{G} \vdash x_j > x'_j$. But then, $x_j < x_0$ is in $\widehat{G}$; the rest of P_i implies that there is a path from $x[t', j]$ to $\mathbf{0}$, so we also have $G \vdash x'_j \geq x_0$, a contradiction.

We conclude that our assignment σ is well defined; and the fact that it satisfies all constraints is easy to prove by its construction. Since it satisfies $(\widehat{G})^\omega$, it also satisfies $(G)^\omega$. □

Theorem 4.26. *If a stable MCS does not satisfy Condition* S*, it is not π-terminating.*

Proof. Let $\mathcal{A}$ be such a system; suppose that every idempotent MC in $cl(\mathcal{A})$ passes the LTT1. The Condition S is satisfied, contradictory to assumption. So there has to be an idempotent MC which fails the LTTS, and by the last lemma, a non-terminating run exists. □

Corollary 4.27. *Condition* S *is a equivalent to π-termination of a stable MCS, and the closure algorithm is a sound and complete decision procedure.*

4.7. **Codish, Lagoon and Stuckey's Algorithm.** Codish, Lagoon and Stuckey [12] also gave (albeit implicitly) an algorithm to decide termination of an integer MCS, under the assumption that the data represented are non-negative integers (so termination follows either by descent towards zero, or by approaching variables). The algorithm is closely related to Algorithms 4.18 and 4.23.

Definition 4.28 (balanced constraint, [12])**.** A cyclic monotonicity constraint G is *balanced* if $G \vdash x_i \rhd x_j \;\Leftrightarrow\; G \models x'_i \rhd x'_j$ (where $\rhd$ is $>$ or $\geq$). The *balanced extension* G_B of G is the weakest monotonicity constraint which is at least as strong as G and is balanced.

To compute the balanced extension, Codish, Lagoon and Stuckey suggest a fixed-point computation: Define $\mathit{bal}(G) = G \wedge \{x_i \rhd x_j \mid G \vdash x'_i \rhd x'_j\} \wedge \{x'_i \rhd x'_j \mid G \vdash x_i \rhd x_j\}$. They observe that there always is a p such that $\mathit{bal}^{(p)}(G)$ is balanced and equals G_B (it is then a fixed point of *bal*). Since there are at most $4n(n-1)$ constraints that can be added by *bal*, $p \leq 4n(n-1)$ (according to [12], there are tighter bounds). Hence, G_B can be computed from G in polynomial time.

Algorithm 4.29. (CLS Algorithm)

Input: an MCS $\mathcal{A}$, *not necessarily stable.*

(1) Build $cl(\mathcal{A})$.
(2) For each cyclic G in $cl(\mathcal{A})$, apply the LTTS to G_B.
Pronounce failure (non-termination) if a graph that fails the test is found.
(3) If the previous step has completed, the MCS terminates.

The CLS algorithm relies on the balancing procedure to compensate for lack of stability in $\mathcal{A}$. In fact, the significance of G_B can be related to stability. Consider the MCS $\{G\}$, consisting of a single cyclic MC. Let $S\{G\}$ be the smallest stable system bisimulating $\{G\}$ It is not hard to prove that in $S\{G\}$ there is a single cyclic graph, namely G_B.

Soundness and completeness of Algorithm 4.29 are quite similar to those given for Algorithms 4.18 and 4.23. Codish et al. give a partial completeness proof, specifically they prove that if G_B fails the test than $\{G\}$ does not terminate. The piece missing for proving non-termination of $\mathcal{A}$ is precisely Lemma 4.24.

4.8. **A General Closure Algorithm.** We next present an algorithm which computes the composition-closure and applies a local test, which does not rely on stability. In essence, it corresponds to Algorithm 4.18 in the way that Condition G (Section 4.2.2) corresponds to Condition S; as Condition G refers to general paths instead of (up or down) threads, so will this test refer to cycles in the graph that may have a more complicated form than those used by Algorithm 4.18. Since this is a strict generalization and does not conflict with the prior definitions, we use the same terminology for cycles, redefined as follows.

Definition 4.30 (types of cycles, generalized)**.** Let G be a cyclic MC. A cycle in G° is a *forward cycle* if it traverses shortcut edges more often in the backward direction (from x'_i to x_i) than it does in the forward direction; a *backward* cycle, if it traverses shortcut edges more often in the forward direction; and a *balanced cycle* if it traverses shortcut edges equally often in both directions.

Figure 5 (a–c) illustrates a forward cycle in the extended sense.

Definition 4.31 (Local Termination Test, General Case)**.** For a cyclic G, we say that G passes the *Local π-Termination Test*, or LTT, if G° either has a balanced strict cycle, or both a forward cycle $\mathfrak{f}$ and a backward cycle $\mathfrak{b}$, at least one of which is strict, and a path from a node of $\mathfrak{f}$ to a node of $\mathfrak{b}$.

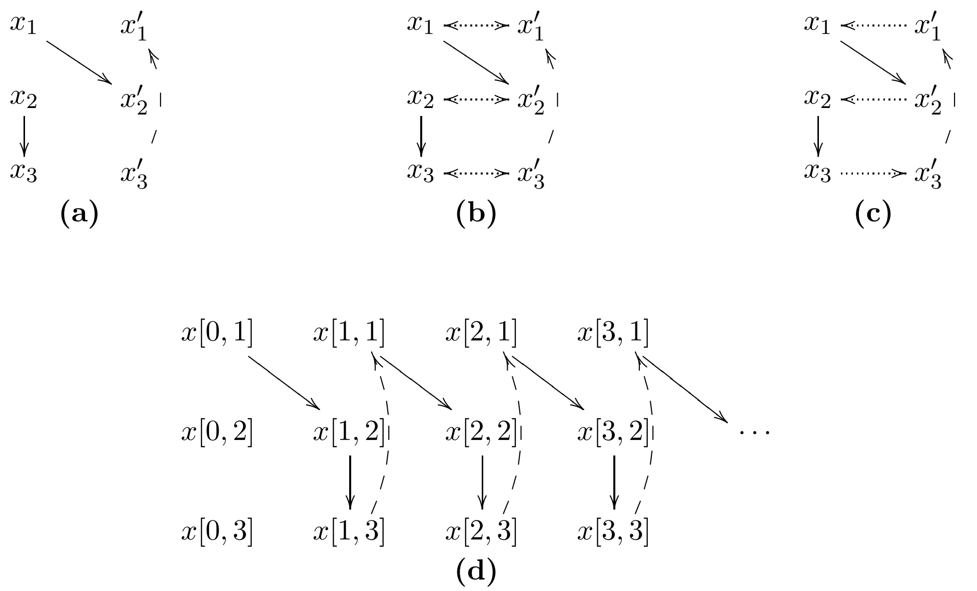


Figure 5: (a) An MC G, (b) its circular variant G°; (c) the directions of shortcut edges are set to form a forward descending cycle, (d) a walk in a prefix of G^{ω}, corresponding to the cycle. The notation x_i^t is a shorthand for $x[t,i]$. (Example after Codish, Lagoon and Stuckey)

Algorithm 4.32. (Closure Algorithm not presuming stability)

(1) Build $cl(\mathcal{A})$.
(2) For each cyclic G in $cl(\mathcal{A})$, apply the Local π-Termination Test.
Pronounce failure (non-termination) if a graph that fails the test is found.
(3) If the previous step has completed, the MCS terminates.

The proof of this algorithm much alike the proof of a similar algorithm for the well-founded model [6], but rather tedious and is omitted.

4.9. **Summary.** In this section we have presented a path-based condition for termination, its soundness and completeness, and sound and complete decision procedures based on it. The condition comes in two flavours: there is a version for general MCSs and a version for stable ones. We concentrated on the latter. Stabilization may be costly (the worst case incurs exponential blow-up), and hence the algorithms that do not require this preprocessing may be more efficient in practice (at least, if all we require is to decide termination). We also included Codish, Lagoon and Stuckey's algorithm, which is similar to the new algorithms proposed here. The contribution of the new algorithms may be mostly theoretical—a contribution to the understanding of MCS termination, as they are directly related to the path-based criteria. In addition, we have closed a gap in the completeness proof of [12]. In the sequel, an entirely different algorithm will be proposed, which constructs an explicit global ranking function. If only the decision problem is of interest, the last (without stabilization) is probably the most efficient, except for simple cases (such a simple case is SCT transition systems, since they are always stable).

5. Full and Partial Elaboration

In [6], some of the results were proved using the following observation: 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 is called full elaboration of an MCS. It served two purposes in [6] (and so it will in the current paper): first, since it generates a stable system, it gives an easy route to a decision procedure—one which is inefficient regarding time but can be used to prove that the problem is decidable in polynomial space. The second purpose was the algorithm to construct a global ranking function; the fully-elaborated system has some structural properties that the construction was based on. In this paper, too, we use full elaboration in that way. The current section presents full elaboration as defined in [6], and infers PSPACE-completeness of the termination problem; then, partial elaboration is introduced, as a tool to be used in the subsequent section.

$\mathbb{Z}$: The idea of full elaboration is taken from [6] , and Theorem 5.4 is a straight-forward application, as it was in the previous paper. The idea and details of *partial elaboration* are new definitions that are necessary for the new algorithm in the following section.

Definition 5.1 (full elaboration)**.** An MCS $\mathcal{A}$ is *fully elaborated* if the following conditions hold:

(1) Each state invariant fully specifies the relations among all variables. That is, for $i, j \leq n$, one of the relations $x_i = x_j$, $x_i < x_j$ or $x_i > x_j$ is specified by I_f.
(2) Each MC is closed under logical consequence.
(3) Each MC in $\mathcal{A}$ is satisfiable.

Since the state invariant fully determines the relations among all variables, we can *re-index* the variables into sorted order, so that the invariant becomes

$$x_1 \left\{ \begin{smallmatrix} < \\ = \end{smallmatrix} \right\} x_2 \left\{ \begin{smallmatrix} < \\ = \end{smallmatrix} \right\} \dots \left\{ \begin{smallmatrix} < \\ = \end{smallmatrix} \right\} x_n. \tag{5.1}$$

Of course, the re-indexing has to be incorporated also in MCs incident to this flow-point, but this is straight-forward to do. Indexing the variables in sorted order has some convenient consequences, such as the having the property:

Definition 5.2. G has the *downward closure property* if for all $k < j$, $G \vdash x_i \geq x'_j$ entails $G \vdash x_i \geq x'_k$.

The number of possible orderings of n variables, hence the maximum number of copies we may need to make of any flow-point to achieve full elaboration, is the *the nth ordered Bell number B_n*, already mentioned. We denote the set of these orderings by Bell_n, and assume that we fix some convenient representation so that "orderings" can be algorithmically manipulated.

The algorithm of full elaboration follows almost immediately from the definitions, and here it is, quoted from [6]:

Algorithm 5.3. (full elaboration) Given an MCS $\mathcal{B}$, this algorithm produces a fully-elaborated MCS $\mathcal{A}$, along with mappings ϕ and ψ, such that $\mathcal{A} \sim_{\phi,\psi} \mathcal{B}$.

(1) For every $f \in F^{\mathcal{B}}$, generate flow-points f_π where π ranges over Bell_n. Define the variable renaming function $\psi_{f_\pi,f}$ so that $\psi_{f_\pi,f}(i)$ is the ith variable in sorted order, according to π. Thus, I_{f_π} will have exactly the form (5.1).

(2) Next, for every MC $G : f \to g$ in $\mathcal{B}$, and every pair f_π, g_ϖ, create a size-change graph $G_{\pi,\varpi} : f_\pi \to g_\varpi$ as follows:
   (a) For every arc $x \to y \in G$, include the corresponding arc in $G_{\pi,\varpi}$, according to the variable renaming used in the two $\mathcal{A}$ flow-points.
   (b) Complete $G_{\pi,\varpi}$ by closure under consequences; unsatisfiable graphs (detected by the closure computation) are removed from the constructed system.

Since a fully elaborated system is stable, full elaboration allows termination to be checked by a closure algorithm as previously described. However, full elaboration will often be costlier than stabilization by fixed-point computation. Therefore, it is not worthwhile—unless we are interested in worst-case complexity! The complexity of the closure algorithm has an exponent of n^2, while full elaboration only introduces an exponent of $n \log n$ and can subsequently be tested for termination by an algorithm of the same exponent, as the next section will show. Full elaboration is also a convenient means to justify the PSPACE upper bound.

Theorem 5.4. *The MCS π-termination problem is PSPACE-complete.*

Proof. It is known [23] that the SCT Termination problem is PSPACE-hard, which also applies to MCS because SCT is a special case (to reduce an SCT problem, which assumes well-foundedness, to a π-termination problem, include a "bottom" variable and constrain all others to be bigger. This is like restricting the domain to the positive integers).

To show that the problem is in PSPACE, we will outline a non-deterministic polynomial-space algorithm for the complement problem, that is, non-termination. The result will follow since (by Savitch's theorem) coNPSPACE = NPSPACE = PSPACE.

Algorithm 4.18 can be seen as a search for a counter-example—a cycle in the CFG that fails the test. The non-deterministic algorithm guesses such a cycle. In each step, it adds a transition to the cycle while composing the transition's MC with an MC that represents the CFG path traversed so far. Only this MC, along with the initial and current flow-point, have to be maintained in memory. Whenever the current flow-point is the same as the initial one, the local termination test is applied. If at some point, an unsatisfiable MC results, the algorithm has failed to find a counter-example. Otherwise it continues until finding one.

Given an input MCS $\mathcal{B}$, we apply the Closure Algorithm to the fully elaborated system $\mathcal{A}$ equivalent to $\mathcal{B}$. To achieve the polynomial space bound, $\mathcal{A}$ is never constructed explicitly. In fact, to represent a flow-point f_π of $\mathcal{A}$, we just maintain the $\mathcal{B}$ flow-point f along with the ordering π (we do not re-index the variables). If the next flow-point is chosen to be g_ϖ (both the flow-point and the ordering are chosen non-deterministically), the MC $G_{\pi,\varpi}$ is computed on the spot.

It should be easy to see that the algorithm only needs access to the original MCS $\mathcal{B}$ and to an additional linear amount of memory. □

In the next section we will have a situation where we require information about the ordering of variables which is not present in the given flow-point invariant, but we do not need the whole ordering. We gain efficiency by only performing a *partial elaboration*, defined generally as follows:

Definition 5.5. Let f be a flow-point in an MCS and let $I_1, \dots, I_k$ be conjunctions of order constraints that are mutually exclusive and satisfy $I_1 \vee I_2 \vee \cdots I_k \equiv I_f$.

A partial elaboration step applied to $f, I_1, \dots, I_f$ splits f into k flow-points, f_i having the invariant I_i, and accordingly replicates every MC from f (or into f). Every such MC is consequence-closed under the new source (or target) invariant, and eliminated if unsatisfiable.

Again, it should be obvious that partial elaboration only refines the abstract program; it does not lose or add possible runs, up to the renaming of flow-points. But it makes the program easier to analyse. As described, it makes the system possibly unstable. Stabilizing the system again may require additional flow-point splitting. For example, if we split f according to the three possible relations among x_1 and x_2, and there is an MC $f \to g$ with constraints $x = x' \wedge y = y'$, we will be forced to split g as well. An important case in which we avoid this complication is SCT constraints, and in fact we can allow a bit more:

Definition 5.6. A *semi-SCT* system is one in which there are no constraints of the form $x < y'$ or $x \le y'$.

This definition generalizes SCT because flow-point invariants are allowed.

Lemma 5.7. *If $\mathcal{A}$ is semi-SCT and stable, applying a partial elaboration step to $\mathcal{A}$ results in a system $\mathcal{A}'$ which is stable as well.*

Proof. It is not hard to see that changing the flow-point invariant of f in a semi-SCT system does not imply any consequences for the ordering of variable values at other flow-points. □

In the next section we have to handle a situation where it is necessary to know which variable has smallest value among k unrelated variables, say $x_1, \dots, x_k$. The reader may verify that it is possible to create k mutually exclusive invariants such that in I_i, x_i is minimum (with some arbitrary breaking of ties).

6. Ranking Functions for Integer MCSs

In this section we develop the algorithm to construct global ranking functions for Integer MCSs. The algorithm will process an MCS and either report that it is non-terminating, or provide an explicit global ranking function. This problem has been solved in the well-founded setting in [6], improving on a previous solution for SCT [22]. Assuming stability, it is possible to solve the problem in the integer domain by a reduction to the well-founded setting (creating difference variables). This solution is not satisfactory, due to the potential squaring of the number of variables, figuring in the exponent of the complexity. This motivates the development of a specialized algorithm. The algorithm will achieve optimal results—in fact, a complexity similar to that obtained in the well-founded case. Achieving this results required dealing with some complications that will be pointed out in the sequel.

In preparation, let us recall the definitions and result of [6].

Definition 6.1. A *global ranking function* for a transition system $\mathcal{T}$ with state space St is a function $\rho : St \to W$, where W is a well-founded set, such that $\rho(s) > \rho(s')$ for every $(s, s') \in \mathcal{T}$.

A ranking function for a MCS $\mathcal{A}$ is a ranking function for $T_{\mathcal{A}}$. Namely, it satisfies $G \vdash \rho(s) > \rho(s')$ for every $G \in \mathcal{A}$.

Remarks: (1) The qualifier *global* may be omitted in the sequel, since we do not deal with the notion of local ranking functions. (2) The definition highlights the role of the transition system $T_{\mathcal{A}}$. In fact, to find a ranking function for an MCS, we transform it into other MC systems that represent the transition system in a refined manner. Therefore the ranking function obtained will be correct with respect to the original MCS.

Definition 6.2 (vectors)**.** Let V be a set of variables. We define $\langle\!\langle V \rangle\!\rangle$ to be the set of tuples $\mathbf{v} = \langle v_1, v_2, \dots \rangle$ of even length, where every even position is a variable of V, such that every variable appears at most once; and every odd position is a non-negative integer constant.

Definition 6.3. The value of $\mathbf{v} \in \langle\!\langle V \rangle\!\rangle$ in program state (f, σ), denoted $\mathbf{v}\sigma$, is a tuple of integers obtained by substituting the values of variables in $\mathbf{v}$ according to σ. Tuples are compared lexicographically.

Theorem 6.4 ([6])**.** *Suppose that MCS $\mathcal{B}$ is terminating in the well-founded model, and has variables $V = \{x_1, \dots, x_n\}$. There is a ranking function ρ for $\mathcal{B}$ where $\rho(f, \sigma)$ is described by a set of elements of $\langle\!\langle V \rangle\!\rangle$, each one associated with certain inequalities on variables, which define the region where that vector determines the function value. The complexity of constructing ρ is $O(|\mathcal{B}| \cdot n^{2n+1})$.*

Here is an example, just to illustrate the form of the function:

$$\rho(f, \sigma) = \begin{cases} \langle 1, x_1, 1, x_3 \rangle\sigma & \text{if } x_1 > x_2 \\ \langle 1, x_2, 1, x_4 \rangle\sigma & \text{if } x_1 \leq x_2. \end{cases}$$

Later, to simplify the presentation and the manipulation of such functions, we can omit σ and write, for example

$$\rho(f) = \begin{cases} \langle 1, x_1, 1, x_3 \rangle & \text{if } x_1 > x_2 \\ \langle 1, x_2, 1, x_4 \rangle & \text{if } x_1 \leq x_2. \end{cases}$$

We are thus dealing with functions that associate a symbolic tuple (or set of tuples selected by order constraints) to a flow-point. Under the assumption of a well-founded domain, the set of tuples is also well-founded by the lexicographic order.

The theorem can be easily translated to a theorem for Integer MCS, based on using pairs of variables whose difference is non-negative and forming a tuple over $\mathbb{N}$ (so again we have well-foundedness). This is formalized next, and yields our first (non-optimal, but simple to describe) solution.

6.1. **The Difference MCS.**

Let $D = \{(i, j) \mid 1 \leq i < j \leq n\}$. We introduce a variable $x_{(i,j)}$, with $(i, j) \in D$, to represent each difference $x_j - x_i$.

Definition 6.5 (difference MCS)**.** Let $\mathcal{A}$ be a fully elaborated system, with n variables in each flow point, indexed in ascending order of value. The *initial difference MCS* of $\mathcal{A}$ is an MCS $\mathcal{A}_0^{\Delta}$ where:

(1) The set of flow-points is as in $\mathcal{A}$. The variables are $V \cup W$, where $V = \{x_1, \dots, x_n\}$ and $W = \{x_{(i,j)} : (i, j) \in D\}$.

(2) For every flow point f, the state invariant I_f^{Δ} includes I_f plus any constraints that can be deduced from the constraints in $\mathcal{A}$, using the rule:

$$x_i \le x_\ell \le x_u \le x_j \Rightarrow x_{(\ell,u)} \le x_{(i,j)}.$$

(3) To every MC $G \in \mathcal{A}$, there is a corresponding MC, G^{Δ}, in $\mathcal{A}_0^{\Delta}$. It includes the constraints in G plus any constraints that can be deduced from the constraints of G using the rules

$$\begin{aligned} x_i \le x'_\ell \le x'_u \le x_j &\Rightarrow x'_{(\ell,u)} \le x_{(i,j)} \\ x_i < x'_\ell \le x'_u \le x_j &\Rightarrow x'_{(\ell,u)} < x_{(i,j)} \\ x_i \le x'_\ell \le x'_u < x_j &\Rightarrow x'_{(\ell,u)} < x_{(i,j)}. \end{aligned}$$

Observe that $\mathcal{A}_0^{\Delta}$ can be derived from $\mathcal{A}$ by a straight-forward, polynomial-time algorithm. It is called *initial* because in the algorithm presented later it will be iteratively refined. But at this stage it suffices. We reproduce the observation from Section 4.2.1:

Observation 6.6. Satisfaction of Condition S by $\mathcal{A}$ is equivalent to satisfaction of the SCT condition by $\mathcal{A}_0^{\Delta}$, restricted to the difference variables.

This allows us to find a ranking function based on the difference variables. The original variables (V) will not be used in it. We keep them in the system, however, as an aid to the forthcoming, more efficient, algorithm. Applying Theorem 6.4 yields

Corollary 6.7. *Suppose that MCS $\mathcal{B}$ is π-terminating, and has variables $x_1, \dots, x_n$. There is a ranking function ρ for $\mathcal{B}$ where $\rho(f)$ is given as a set of elements of $\langle\langle W \rangle\rangle$, each one associated with certain inequalities on differences of variables. These inequalities define the region where that vector determines the function value. There are at most $B_{n(n-1)/2}$ different vectors for any flow-point. The complexity of constructing ρ is $O(|\mathcal{B}| \cdot (n(n-1)/2)^{n(n-1)+1}) = O(|\mathcal{B}| \cdot n^{2n^2})$.*

To present the function in a readable way, we will replace the difference variables by expressions $x_i - x_j$, so we obtain a function like this: (the example is only meant to illustrate the form)

$$\rho(f) = \begin{cases} \langle 1, x_2 - x_4, 1, x_3 - x_4 \rangle & \text{if } x_2 - x_4 > x_2 - x_3 \\ \langle 1, x_2 - x_4, 0, x_3 - x_4 \rangle & \text{if } x_2 - x_4 \le x_2 - x_3 \end{cases}$$

But this is an unsatisfactory result, because of the n^2 exponent and the fact that tuples may include up to $n(n-1)/2$ variable positions (or at least, this is the bound that the theorem gives). The challenge tackled in the rest of this section is how to reduce the complexity to $n^{O(n)}$ and the number of variables in each tuple to $n-1$, by solving the problem specifically rather than reducing it to the well-founded case. This brings the upper bound close to lower bounds (for the length of the tuples, the worst-case lower bound is matched exactly) based on the results of [9].

6.2. **Overview.** The construction below follows the same basic outline as the one in [6], which is also similar to other global ranking-function algorithms such as [7, 2]. The construction is iterative, based on the notions of a *quasi-ranking function* and *a residual transition system.*

Definition 6.8. Let $\mathcal{T}$ be a transition system with state space St. A *quasi-ranking function* for $\mathcal{T}$ is a function $\rho : St \to W$, where W is a well-founded set, such that $\rho(s) \geq \rho(s')$ for every $(s, s') \in \mathcal{T}$.

The *residual transition system* relative to ρ, denoted $\mathcal{T}/\rho$, includes all (and only) the transitions of $\mathcal{T}$ which do *not* decrease ρ.

The outline of the algorithm is: find a quasi-ranking function, generate a representation of the residual system, repeat as long as the system is not vacant (i.e., has transitions). If the quasi-ranking functions found are tuple-valued functions $\rho_1, \rho_2, \dots, \rho_k$, it is easy to see that $\rho_1 \cdot \rho_2 \cdot \ldots \cdot \rho_k$ is a ranking function, where $\cdot$ is tuple concatenation, extended naturally to functions (it is tacitly assumed that care is taken to maintain the correct structure regarding even and odd positions in the tuples).

What are the quasi-ranking functions and how are they found? We have two major cases, depending on the connectivity of the control-flow graph. If it is not strongly connected, there are flow-points that can only occur in a particular order, and we obtain the quasi-ranking function just by assigning suitable numbers to flow-points. This is the easy case. The case of a strongly-connected system is the important one. In [6], it was shown that, thanks to full elaboration, it is always possible to identify a single variable x_{i_f} for each flow-point f so that the function $\rho(f) = x_{i_f}$ is quasi-ranking. The proof of the existence of such a variable, and the algorithm to find it, make essential use of the fact that the n variables are totally ordered by the flow-point invariant. In the current setting, we have a total order on the original n variables, but this does not induce a total order on the $n(n-1)/2$ *difference variables*, and it is them that will constitute the quasi-ranking functions. So, to obtain the results below, the main effort (given the results of [6]) has been to find a way to deal with the partially-ordered set of difference variables without resorting to fully ordering them (as that would create an n^2 exponent again).

Throughout the rest of this section, $\mathcal{A}$ denotes a fully-elaborated MCS that satisfies Condition S, and $\mathcal{A}_0^\Delta$ the initial difference MCS. The algorithm begins with $\mathcal{A}_0^\Delta$, and then, iteratively, refines and restricts it, while constructing quasi-ranking functions, so that in general, the sub-algorithm for finding a q.r.f. works on an MCS that has already changed from the initial one. We define *a difference MCS* to be any MCS $\mathcal{A}^\Delta$ obtained from $\mathcal{A}_0^\Delta$ by a series of partial elaboration steps, as defined in Section 5, and restrictions. Thus $\mathcal{A}^\Delta$ is one of a sequence of MC systems $\mathcal{A}_j^\Delta$, starting with $\mathcal{A}_0^\Delta$, so that

$$\mathcal{A}_{2i+1}^\Delta \simeq_{\phi_i} \mathcal{A}_{2i}^\Delta$$
$$\mathcal{A}_{2i+2}^\Delta \Subset \mathcal{A}_{2i}^\Delta$$

So, for all j, $\mathcal{A}_j^\Delta \Subset_\phi \mathcal{A}_0^\Delta$ for an appropriate ϕ. The goal of each elaboration step is to allow for finding a q.r.f., and the subsequent restriction represents the residual transition system relative to the current quasi-ranking function. Note that the elaboration steps do not involve a re-indexing of variables—this is important since we rely on the indexing of the original elaborated MCS $\mathcal{A}$.

6.3. **Preparations for the construction.** We begin with a few definitions and properties that the algorithm relies on. These definitions are taken (with some adaptation) from previous work on SCT, and therefore refer as *threads* to what we have named *down-threads*. Since our analysis of the difference variables is SCT analysis—only down-threads are considered—let it be understood that in the context of difference variables, a "thread" is a down-thread.

Definition 6.9 (thread preserver)**.** Given difference MCS $\mathcal{A}^{\Delta}$, a mapping $P : F^{\mathcal{A}} \to \mathcal{P}(D)$ is called a *thread preserver* of $\mathcal{A}^{\Delta}$ if for every $G^{\Delta} : f \to g$ in $\mathcal{A}^{\Delta}$ it holds that whenever $\alpha \in P(f)$, there is $\beta \in P(g)$ such that $G^{\Delta} \vdash x_\alpha \geq x'_\beta$.

It is easy to see that the set of thread preservers is closed under union. Hence, there is a unique maximal thread preserver, which we denote by $\mathrm{MTP}(\mathcal{A}^{\Delta})$. Given a standard representation of $\mathcal{A}^{\Delta}$, $\mathrm{MTP}(\mathcal{A}^{\Delta})$ can be computed in linear time [8].

We also need the following definition and results:

Definition 6.10 (complete thread)**.** A thread in a given multipath is *complete* if it starts at the beginning of the multipath, and is as long as the multipath.

Lemma 6.11. *Let M_∞ be an infinite multipath. If every finite prefix M of M_∞ contains a complete thread, then M_∞ contains an infinite thread.*

Proof. The proof is a straight-forward application of König's Lemma. □

Lemma 6.12. *If a strongly connected MCS satisfies SCT, every finite multipath includes a complete thread.*

The proof is straight-forward and left out.

Definition 6.13 (thread-safe)**.** We call a variable *thread-safe* at flow-point f if every finite multipath, starting at f, includes a complete thread starting at that variable.

6.4. **Finding a singleton thread-preserver.** The role of this part of the algorithm is to single out a variable x_{α_f} for every flow point, such that these variables constitute a singleton thread-preserver (i.e., $P(f) = \{\alpha_f\}$ be a thread preserver). Observe that this induces a quasi-ranking function, namely $\rho(f) = \langle x_{\alpha_f} \rangle$. We assume, until further notice, that we are dealing with a strongly-connected MCS; the complete algorithm will provide for the non-strongly-connected case.

Definition 6.14 (ordering of D)**.** We write $(i,j) \subseteq (i',j')$ for containment of the interval $\{i,\ldots,j\}$ in the interval $\{i',\ldots,j'\}$.

Containment is a partial order, and moreover a semi-lattice with join operation $(i,j) \sqcup (i',j') = (\min(i,j), \max(i',j'))$.

Recall that indexing the variables in ascending order of value, during full elaboration, provided a useful *downward closure* property. In particular, the lowest-numbered variable among a set of variables has the smallest value. This was crucial in the construction of [6], and here, we give as a substitute the following (more subtle) lemma for the difference variables.

Lemma 6.15. *Let $\mathcal{A}^\Delta$ be a difference MCS for a fully elaborated $\mathcal{A}$. Each MC, G^Δ, of A^Δ has the following properties:*

1. *If $G^\Delta \vdash x_\alpha \geq x'_\beta$, then $G^\Delta \vdash x_\gamma \geq x'_\beta$ for every $\gamma \supseteq \alpha$.*
2. *If $G^\Delta \vdash x_\alpha \geq x'_\beta$, and $G^\Delta \vdash x_\alpha \geq x'_\gamma$, then $G^\Delta \vdash x_\alpha {\geq} x'_\delta$ for every $\delta \subseteq \beta \sqcup \gamma$.*

Proof. We prove only the second claim (being a bit more involved). Let $\alpha = (a_l, a_h)$, $\beta = (b_l, b_h)$ and $\gamma = (c_l, c_h)$. The inequality $x_\alpha \geq x'_\beta$ means $x_{a_l} \leq x'_{b_l}$ and $x_{a_h} \geq x'_{b_h}$ and similarly for the second inequality, so we have:

$$x_{a_l} \leq x_{\min(b_l,c_l)}; \quad x_{a_h} \geq x_{\max(b_h,c_h)}.$$

That is, $x_\alpha \geq x_{\beta\sqcup\gamma}$, which entails the conclusion. □

Lemma 6.16. *Assume that $\mathcal{A}^\Delta$ is strongly connected and satisfies SCT in the difference variables. For every f, let $S(f)$ be the set of indices of difference variables that are thread-safe at f in $\mathcal{A}^\Delta$. Then $S(f)$ is not empty for any $f \in F^{\mathcal{A}^\Delta}$ and $S = MTP(\mathcal{A}^\Delta)$*[4].

Proof. Let M be any finite $\mathcal{A}^\Delta$-multipath starting at f. Since $\mathcal{A}^\Delta$ satisfies SCT and is strongly connected, there must be a complete thread in M, say starting at x_α. But then $x_{(1,n)}$ can also start a thread. It follows that $(1, n) \in S(f)$, so $S(f)$ is not empty.

We now aim to show that S is a thread preserver. Let $\alpha \in S(f)$, and let $G^\Delta : f \to g$. Let H^Δ be any MC with source point g, M a finite multipath starting with H^Δ and consider the multipath $G^\Delta M = G^\Delta H^\Delta \dots$. By definition of $S(f)$, it has a complete thread that begins with an arc from x_α, say $x_\alpha \to x'_{\beta_M}$, followed by an arc of H^Δ, say $x_{\beta_M} \to x'_{\gamma_M}$. Let B be the set of all such indices β_M (where M ranges over all multipaths starting with H^Δ), and $\kappa = \bigsqcup B$. Then $x_\alpha \to x_\kappa$ is an arc of G^Δ by Lemma 6.15(2); and by 6.15(1) one can see that, for each M, H^Δ has an arc $x_\kappa \to x'_{\gamma_M}$. Thus, M has a complete thread beginning with x_κ, and κ is, by definition, in $S(g)$. We have proved that S has the thread-preservation property. On the other hand, it is easy to see that if P is a thread preserver such that $P(f) \neq \emptyset$, then the variables indexed by $P(f)$ are thread-safe at f. Thus every thread preserver is contained in S, and we conclude that $S = \mathrm{MTP}(\mathcal{A}^\Delta)$. □

Lemma 6.17. *Assume that $\mathcal{A}^\Delta$ is strongly connected and satisfies SCT in the difference variables. Let P be a thread preserver (in difference variables) and suppose that for every $f \in F^{\mathcal{A}^\Delta}$, invariant I^Δ_f implies that a certain variable x_{α_f} has smallest value among the variables of $P(f)$. Then every $G^\Delta : f \to g$ includes $x_{\alpha_f} \to x'_{\alpha_g}$. In other words, $\{\alpha_f\}$ constitutes a singleton thread preserver.*

Proof. By the definition of a thread-preserver, each MC $G^\Delta : f \to g$ must have an arc $x_{\alpha_f} \to x'_\beta$ for some $\beta \in P(g)$; so by Lemma 6.15(2), G^Δ includes $x_{\alpha_f} \to x'_{\alpha_g}$. □

[4] This notation refers to a maximal thread preserver involving the difference variables only.

Since the lemma applies to any thread preserver, the easiest implementation is to use the MTP, just because it can be found efficiently. The difficulty in applying the lemma is that the difference variables are only partially ordered, so in a given set of variables there may be no variable that is necessarily of smallest value. In order to make the lemma applicable, we shall use partial elaboration. Specifically, observe that given $P(f)$, there can be at most $n-1$ candidates for the variable of smallest value. This is so because in the poset of intervals there is no antichain bigger than $n-1$. Suppose that for flow-point f, the set of potential smallest variables in $P(f)$, $P_{\min}(f)$, has more than a single element. We will then create a duplicate f_α of f for each $\alpha \in P_{\min}(f)$, and add different constraints to each one, such that in f_α, x_α will be minimum, as required by the lemma. This is a case of partial elaboration, as described in Section 5, and due to the fact that the constraints on difference variables are semi-SCT, the duplication does not trigger a cascade of duplications of other flow-points (Lemma 5.7); we thus increase the size of the system at most $(n-1)$-fold. After this processing, Lemma 6.17 can be applied.

In the following lemma we formulate the conclusion in a somewhat generalized manner, which will be useful later. Recall that D is the set of indices of all difference variables.

Lemma 6.18. *Let $\mathcal{A}^\Delta$ be a strongly connected difference MCS, that satisfies SCT restricted to a certain subset of the difference variables at each flow-point f, given by $T(f) \subseteq D$, such that $T(f)$ is closed under $\sqcup$. Procedure* SingletonTP$(\mathcal{A}^\Delta, T)$ *below finds a singleton thread preserver within T, while (possibly) modifying the MCS by partial elaboration. The partial elaboration involves duplicating each point f at most $n-1$ times.*

Algorithm 6.19. SingletonTP$(\mathcal{A}^\Delta, T)$

(1) Compute $P = \mathrm{MTP}(\mathcal{A}^\Delta, T)$, the MTP of $\mathcal{A}^\Delta$ restricted to the variables $T(f)$ for each f. If empty, the procedure fails.

(2) For each $f \in F^{\mathcal{A}^\Delta}$, identify the minimal elements of $P(f)$, based on the flow-point invariants. Suppose that there are $k > 1$ minima, $x_{\alpha_1}, \dots, x_{\alpha_k}$. Let I be the invariant of f; create mutually exclusive invariants $I_1, \dots, I_k$ such that I_i includes I with added constraints $x_{\alpha_i} < x_{\alpha_j}$ for $j < i$, and $x_{\alpha_i} \le x_{\alpha_j}$ for $i \le i \le k$. Perform partial elaboration, replacing f by k copies with the respective invariants. Choose x_{α_i} for the thread preserver at the i'th copy of f.

Note that the results of the procedure (unless it fails) are a possibly modified MCS and its thread preserver.

6.5. **Systems with frozen threads.** Having found a quasi-ranking function, our next step is to modify to $\mathcal{A}^\Delta$ so that it expresses the residual system. This includes the following changes

(1) Transitions in which the quasi-ranking function strictly decreases (that is, we have $G^\Delta \vdash x_{\alpha_f} > x_{\alpha_g}$) are removed.

(2) In transitions where the quasi-ranking function was not known to decrease strictly (that is, we only had $G^\Delta \vdash x_{\alpha_f} \ge x_{\alpha_g}$) it is now required not to change.

We now recall another idea from [6], *freezers*. A freezer is a singleton thread-preserver where the values are “frozen,” that is, constrained not to change.

Definition 6.20 (freezer)**.** Let $\mathcal{A}$ be an MCS and $C : F^{\mathcal{A}} \to \{1, \dots, n\}$ a function that associates one original variable (technically, the index of one such variable) to to each flow-point. Such C is called a *freezer* if for every $G \in \mathcal{A}$, $G \vdash x_{C(f)} = x'_{C(g)}$.

In our case, we have a thread of difference variables x_{α_f} that we wish to constrain not to change. Suppose that $G^{\Delta} \vdash x_{\alpha_f} \geq x_{\alpha_g}$, where $\alpha_f = (l, h)$ and $\alpha_g = (i, j)$. In order that the differences before and after the transition be the same, we must have both $x_l = x'_i$ and $x_h = x'_j$. Therefore, every arc like that, participating in the singleton thread-preserver found, implies that two inequalities in *original* variables are restricted to equalities, creating two freezers.

Assuming that there still are cycles in the MCS, we need to look for an additional quasi-ranking function. We have to restrict the search in order not to find the same quasi-ranking function again (which, albeit "frozen," is still a q.r.f.). This requires a bit further analysis.

The following observation comes easily from full elaboration:

Observation 6.21. Freezers are consistently ordered by the relations among their variables. That is, if C_L and C_H are freezers and $I_f \vdash x_{C_L(f)} < x_{C_H(f)}$ for some flow-point f, then this relation holds in every flow-point and we write $C_L < C_H$.

Note that due to the re-indexing in full elaboration, the order relation among the freezers matches the relation among the *indices* $C_L(f)$ and $C_H(f)$. The case of variables related by an equality constraint at a flow-point is less obvious but, in fact, if variable x_1, say, is "frozen" and $I_f \vdash x_1 = x_2$, then x_2 is also "frozen". To make a long story short, the algorithm below has the property that if two variables are constrained by equality, once one of them has been used for a thread-preserver and thereafter put into a freezer, the other can safely (and advantageously) be ignored (yet we ignore this situation in the description of the algorithm, to simplify presentation).

Lemma 6.22. *Suppose that $\mathcal{A}^{\Delta}$ is strongly connected, and has some freezers, among which C_L is lowest and C_H highest. And suppose that $\mathcal{A}^{\Delta}$ terminates. Divide the indices of original variables, for every flow-point f, into three* regions*: the lower region $V_0(f) = \{1, \dots, C_L(f)\}$, the middle region $V_1(f) = \{C_L(f), \dots, C_H(f)\}$, and the upper region $V_2(f) = \{C_H(f), \dots, n\}$. Then every infinite $\mathcal{A}^{\Delta}$-multipath has an approaching pair confined to one of the regions.*

Proof. Let M be an infinite multipath of $\mathcal{A}$, $M = f_0 \overset{G_1}{\to} f_1 \overset{G_2}{\to} f_2 \dots$ Suppose that M has an approaching pair: the low up-thread $(x[k, l_k])_{k=0,1,\dots}$ and the high down-thread $(x[k, h_k])_{k=0,1,\dots}$.

At least one of the low and high threads has to be infinitely often strict; let us suppose that it is the high thread (the other case is similar). This thread cannot intersect any of the the frozen threads infinitely often (or we would have an unsatisfiable section of the multipath, contradicting Lemma 3.4); so in an infinite tail of the multipath (which is all that matters) it lies either always above C_H, or below C_L, or between them; to avoid a trite case analysis, let us pick just one of the cases, and suppose that the thread uses variables above C_H, that is, from V_2. Now, we can let C_H play the part of the low thread, to obtain an approaching pair within the upper region (V_2). □

Corollary 6.23. *Under the assumptions of the last lemma, define regions of difference variables:*

$$D_0(f) = \{(i,j) \mid i < j \le C_L(f)\}; \tag{6.1}$$

$$D_1(f) = \{(i,j) \mid C_L(f) \le i < j \le C_H(f)\}; \tag{6.2}$$

$$D_2(f) = \{(i,j) \mid C_H(f) \le i < j\}; \tag{6.3}$$

then in every infinite $\mathcal{A}^\Delta$-multipath there is an infinitely-descending thread of difference variables within one of these regions.

Lemma 6.24. *Under the assumptions of the last lemma, there is a region D_r that contains an infinite thread (corresponding to a weakly approaching pair in V_r) in every infinite $\mathcal{A}^\Delta$-multipath.*

Note the change in the order of quantification: the region is selected before the multipath, at the expense of not guaranteeing strict descent in every multipath.

Proof. Assume to the contrary that for each of $r = 0, 1, 2$ there is an infinite multipath in which Region D_r contains no infinite thread. By Lemma 6.11, there are finite multipaths M_0, M_1, M_2 such that M_i has no complete thread in region i. Since our MCS is strongly connected, one can form a multipath $(M_0 \ldots M_1 \ldots M_2 \ldots)^\omega$, which will have no infinite thread in any of the regions, contradicting Corollary 6.23. □

Let r be such that there is always an infinite thread in D_r. We can use the strategy of the previous subsection to find a singleton thread preserver, thus making progress in our construction. But is it progress? There is a pitfall: in the case of the middle region, our procedure may find a pair of variables that was already frozen (they constitute a weakly approaching pair, but they really never approach). However, the middle region can be treated in a special way that avoids the pitfall and is also more efficient.

Lemma 6.25. *Under the assumptions of Lemma 6.22, An infinite multipath M contains an approaching pair within the middle region (V_1) if and only if this region contains a thread—either a down-thread or an up-thread—which is infinitely often strict. Such a thread is disjoint from the two freezers delimiting this region.*

Proof. The non-trivial implication is the "if," but it is also quite easy: suppose that the region contains an infinitely-often strict down-thread (up-thread). It can be complemented to an approaching pair by using C_L (C_H) for the other thread. □

From this lemma we conclude that the middle region can be reduced to the following two subsets of difference variables:

$$D_1^L(f) = \{(C_L(f), j) \mid C_L(f) < j < C_H(f) \text{ such that } x_j \text{ is not frozen in } f\} \tag{6.4}$$

$$D_1^H(f) = \{(i, C_H(f)) \mid C_L(f) < i < C_H(f) \text{ such that } x_i \text{ is not frozen in } f\}. \tag{6.5}$$

Both sets of intervals are closed under $\sqcup$ and Lemma 6.18 applies. Moreover, these sets of intervals are totally ordered, which means that no elaboration steps will be needed when looking for a singleton thread-preserver among them.

6.6. **Putting it all together.**

Algorithm 6.26. (ranking function construction for $\mathcal{A}^\Delta$)

Assumes that $\mathcal{A}^\Delta$ is a difference MCS. The system may also be adorned with any number of freezers. If $\mathcal{A}^\Delta$ terminates, a ranking function will be returned. Otherwise, the algorithm will fail.

(1) List the SCCs of $\mathcal{A}^\Delta$ in reverse-topological order. For each $f \in F^{\mathcal{A}^\Delta}$, let κ_f be the position of the SCC of f. Define $\rho(f) = \langle \kappa_f \rangle$.
 If all SCCs are vacant (contain no transitions), return ρ.

(2) For each SCC C, compute a ranking function ρ_C by applying the next algorithm to the component (separately). Let $\rho' = \bigcup \rho_C$. Return $\rho \cdot \rho'$.

Algorithm 6.27. (for a strongly connected $\mathcal{A}^\Delta$)

(1) If no freezers are associated with $\mathcal{A}^\Delta$, run SingletonTP($\mathcal{A}^\Delta, D$) and proceed to Step 3.

(2) If some freezers are associated with $\mathcal{A}^\Delta$, let C_L and C_H be the lowest and highest freezers[5]. For every flow-point f, create the sets of difference variables $D_0(f)$, $D_1^L(f)$, $D_1^H(f)$, $D_2(f)$, as defined in Equations (6.1–6.5).
   (1) Run SingletonTP($\mathcal{A}^\Delta, D_1^x$) for $x = L$ and then H. If one of these calls succeeds, proceed to Step 3.
   (2) Run SingletonTP($\mathcal{A}^\Delta, D_0$). If successful, proceed to Step 3.
   (3) Run SingletonTP($\mathcal{A}^\Delta, D_2$). If successful, proceed to Step 3. If not, the algorithm fails.

(3) (Create residual system) Let P be the thread preserver found. For every graph $G^\Delta : f \to g$, if it includes $x_{P(f)} > x'_{P(g)}$, delete the graph from $\mathcal{A}^\Delta$. Otherwise, retain the graph. Also include [6] $C_1(f) = \pi_1 P(f)$, $C_2(f) = \pi_2 P(f)$ as freezers associated with $\mathcal{A}^\Delta$.

(4) For every f, let $\rho(f) = \langle x_{P(f)} \rangle$.

(5) If $\mathcal{A}^\Delta$ is now vacant, return ρ. Otherwise, compute a ranking function ρ' recursively for (what remains of) $\mathcal{A}^\Delta$, using Algorithm 6.26, and return $\rho \cdot \rho'$.

Recall that $\tilde{O}(f)$ is a shorthand for $O(f \cdot \log^{O(1)} f)$. Thus $\tilde{O}(n^n)$ is asymptotically dominated by n^n times a polynomial in n. For an MCS $\mathcal{A}$, let $|\mathcal{A}|$ denote the number of abstract transitions (MCs) in $\mathcal{A}$ (without loss of generality, $|\mathcal{A}| \geq |F^{\mathcal{A}}|$).

Theorem 6.28. *Let $\mathcal{A}$ be a fully-elaborated, π-terminating MCS, with n variables per point. Algorithm 6.26, applied to $\mathcal{A}^\Delta$, produces a ranking function $\rho : F^{\mathcal{A}} \to \langle\langle W \rangle\rangle$ where each vector includes at most $n-1$ difference variables. The complexity of construction of ρ is $\tilde{O}(|\mathcal{A}| \cdot n!)$.*

Proof. Assuming that the correctness of the algorithm has been justified convincingly enough, we now discuss the complexity. One should consider the effect of partial elaborations by SingletonTP. Such a step may make up to $n-1$ copies of every flow-point. However, each time it is performed, at least one new freezer is subsequently created (in the first time, two freezers). Therefore, this multiplication

[5] A case in which there is only one freezer is acceptable. C_L and C_H are then the same thread and Step (2a) is skipped.

[6] We use the projections π_1, π_2 to map indices from D to their components.

of flow-points can occur at most $n-1$ times, leading to the upper bound on the length of the vectors. Since the number of variables participating in the search for quasi-ranking functions diminishes in each iteration, we obtain a bound of $|F^{\mathcal{A}}| \cdot n!$ on the size of the resulting expression, which associates a vector with each of the flow points of the mostly elaborated system obtained. The running time is further multiplied by a (low order) polynomial expressing the complexity of the procedures at each level of the recursion. □

Theorem 6.29. *Let $\mathcal{B}$ be a π-terminating MCS, with n variables per point. A ranking function ρ for $\mathcal{B}$ where $\rho(f)$ is given by a case expression with inequalities among differences for guards; the value in each case is given by a vector in $\langle\!\langle W \rangle\!\rangle$, that includes at most $n-1$ difference variables. The complexity of construction of ρ, as well as the size of the expression, are $\tilde{O}(|\mathcal{B}| \cdot 2^n (n!)^2)$.*

Proof. First, fully elaborate $\mathcal{B}$, yielding an MCS of size at most $|\mathcal{B}| \cdot B_n \le |\mathcal{B}| \cdot 2^n (n!)$, then use the last theorem. □

The algorithm could, in principle, be used just to determine if a system is terminating, and with an exponent of $O(n \log n)$, better than the $O(n^2)$ exponent given in Section 4. However, the use of full elaboration makes it unattractive in practice because it blindly generates the worst case (all possible ordering of variables) for every input instance. In contrast, the algorithms of Section 4 will often perform much better than their worst-case behaviour.

7. Rooted Versus Uniform Termination

Up to this point, the notion of termination used was *uniform termination*, which means that there must be no cycles in the whole state space of the modeled transition system. Practically, what we usually require is *rooted termination*, when only computation paths beginning at a given initial point f_0 are considered.

$\mathbb{Z}$: There is no difference between the treatment of this subject in the well-founded case and here, but the examples below should illustrate that the issue is doubly important in the current setting.

Here is a little C example to show the importance of rooted termination:

```
    if (x<0)
        while (y > 0)  y = y+x
```

Consider an abstraction that represents the command `y=y+x` by three parallel MCs, as discussed in Example 2.11. It can be shown to terminate when only paths from the top of the program are considered. The example could also be solved by a preprocessing that calculates state invariants, such as `x` <0 inside the `while`, as is often done in program analysers. But this can get complicated: the following example would require an invariant that specifies the dependence of variable `b` on `x`. This kind of invariant that is not found in common invariant generators (namely those that describe a state by a conjunction of linear constraints). However with rooted termination, no invariants are necessary other than the direct translation of the conditionals to monotonicity constraints.

```
if (x<0)  b=1
    else  b=0;
if (x<>0)
    while (y > 0)
        if (b) y = y+x;
          else y = y-x;
```

We conclude that it is desirable, practically, to account for rooted termination. Up to this point, this was avoided only in favor of simplicity of presentation. However, it is very easy to do: with a stable system, unreachable states will be represented by flow-points that are inaccessible from f_0, due to Lemma 3.4.

Thus, all that is necessary is to remove inaccessible parts of the CFG, or better yet, never generate them in the first place, by creating the stabilized (or fully elaborated) system in the manner of a graph exploration (say, DFS) starting at the initial point, only covering the reachable state space. This can occasionally have a significant effect on efficiency (in particular with full elaboration), as confirmed by our experience with implementing full elaboration [31].

Example 7.1. To conclude this section, here is another example which illustrates the effectiveness of the MCS abstraction in expressing disjunctions, besides the need for rooted termination. Disjunction is used in expressing the condition `x != 0`, as well as representing the command `x := x-1`.

```
assert x > 0
while (x != 0)  x := x-1
```

Abstraction: unlike previous examples, we will not merge this time the loop guard with the loop body, in order to clarify the transformation. We will thus have three flow-points, 0 (initial), 1 (loop header) and 2 (loop body).

$$\begin{aligned}
G_1 : 0 \to 1 : &\quad \mathsf{x} > \mathsf{0} \wedge \mathsf{x} = \mathsf{x}' \wedge \mathsf{0} = \mathsf{0}' \\
G_2 : 1 \to 2 : &\quad \mathsf{x} > \mathsf{0} \wedge \mathsf{x} = \mathsf{x}' \wedge \mathsf{0} = \mathsf{0}' \\
G_3 : 1 \to 2 : &\quad \mathsf{x} < \mathsf{0} \wedge \mathsf{x} = \mathsf{x}' \wedge \mathsf{0} = \mathsf{0}' \\
G_4 : 2 \to 1 : &\quad \mathsf{x} > \mathsf{0} \wedge \mathsf{x} > \mathsf{x}' \wedge \mathsf{x}' \geq \mathsf{0}' \wedge \mathsf{0} = \mathsf{0}' \\
G_5 : 2 \to 1 : &\quad \mathsf{x} \leq \mathsf{0} \wedge \mathsf{x} > \mathsf{x}' \wedge \mathsf{x}' < \mathsf{0}' \wedge \mathsf{0} = \mathsf{0}'
\end{aligned}$$

8. Some Related work

The field of termination analysis is well developed and it is infeasible to survey it extensively here. This section will point out some works that are related inasmuch as they base a termination analysis on the behavior of integer variables. The following questions are asked when considering such works:

(1) Does the proposed algorithm or tool work on an abstract transition system, or on concrete programs? Clearly the case that can be best related to the current

work is the former. Therefore, much work of the second kind is ignored here; but it is not hard to find.

(2) What abstraction is used? In particular, are monotonicity constraints used?

(3) What is the main technique? Is it complete for the given abstraction? (It is also possible to ask if there would have been completeness, had the method been applied to MCS. I have tried to answer this question, though some of the methods are not fully described in the publications, so it is difficult to be precise.)

We start with those works which mostly resemble the current paper.

(1) Manolios and Vroon [27] describe a termination analysis implemented in the ACL2 theorem prover. It works on concrete programs using the SCT abstraction. It handles termination arguments involving integers by introducing the difference of two concrete integer variables as an abstract variable, when it can be determined (using theorem proving techniques applied to the source program) to be non-negative. Of course, it may also include a single integer variable if it is determined to be lower-bounded. Since concrete programs are the subject, no completeness claim is made. It would be possible to represent an MCS as a program and apply the tool, but completeness is still unlikely because, as previously remarked, this reduction to SCT has to be combined with stabilization to achieve completeness.

(2) Avery [3] describes a tool to analyze C programs by first abstracting them to a constraint transition system of the following form: transition constraints use inequalities ($\geq$, $>$, $\leq$, $<$) to relate source and target variables; flow-point invariants are polyhedral ones, that is, conjunctions of linear inequalities in state variables and constants. This is clearly a generalization of MC transition systems, however one expressive enough to represent counter programs, which means that termination is undecidable. The (sound but incomplete) algorithm is based on closure computation, where the composition operation used to form the closure takes into account only the variants (source-to-target relations, which are MCs) and not the invariants. For each idempotent graph in the closure, the invariants are taken into account when deciding which variables would imply termination if they descend.

Clearly, the algorithm could be applied to MCs. What bars it from achieving completeness in this case is the fact that the control-flow graph is not refined (i.e., no stabilization). Here is an example to illustrate this limitation (the program consists of a single loop with the following description)

$$\mathtt{y} > \mathtt{w} \land \mathtt{x} > \mathtt{x}' \land \mathtt{x} > \mathtt{y}' \land \mathtt{w} \leq \mathtt{z}' \land \mathtt{w} \leq \mathtt{w}'$$

Note that the only invariant which is valid whenever the transition is entered is $\mathtt{y} > \mathtt{w}$, which is not very helpful.

(3) Termilog [24] was a termination analyzer for Prolog, that made use of an abstraction to monotonicity constraints and a closure computation plus a local test, which is sound but incomplete for the MC constraint domain, as pointed out in [12]. The variables of the abstraction represent certain *norms* of symbolic terms in the Prolog program. They are, therefore, non-negative integers and the termination proof only looks for descent towards zero. We should remark, however that they do not use "abstract compilation" as in our examples. Instead,

an abstract interpreter is used to compute the closure set. This may be more precise with respect to the semantics of the subject program, see [19].

In [17], Dershowitz et al. reformulated the principles underlying the usage of monotonicity constraints in Termilog, and also proposed an extension to handle programs with integer variables and arithmetics. Their proposal is based on creating a "custom-tailored" domain for abstraction of the integer variables, based on constraints extracted from the program. Their algorithms are Prolog-specific, and do not analyze a contraint-based abstract program, but it seems that it could be applied to MC transition systems (appropriately represented) and that the proposed abstraction process may actually compute a (partial) elaboration and obtain a stable system. It will further attempt to prove termination for every cyclic MC by "guessing" a local ranking function. The functions they propose to "guess" are differences $(x_i - x_j)$, which we know to suffice for Idempotent cyclic MCs (See Definition 4.21: the function is $x_h - x_l$). However, such functions do not suffice for all cyclic MCs [12]. We conlcude that it would be desirable to apply the idempotence-based algorithm with their framework (which, in fact, they do in the part that deals with symbolic variables and norms).

(4) Terminweb [13] is a termination analyzer for Prolog. It uses a procedure which tries to prove termination of an abstract program, and there are two kinds of abstractions used. In the first, transitions are described by polyhedra. In the second, by monotonicity constraints. In both cases, the data are non-negative integers and the termination proof sought is based on descent towards zero. As for the methods, closure computation and a local termination test are used; in the polyhedral case, the closure is approximated (using widening) for otherwise it might be infinite. For the monotonicity-constraint abstraction, the closure computation is precise, but the local termination test is incomplete, as pointed out in [12].

(5) Mesnard and Serebrenik [28] show that for abstract programs with transitions defined by polyhedra (conjunctions of linear inequalities), when the data are rationals or real numbers, it is possible to determine in polynomial time (using linear programming) whether there is a global ranking function that associates an affine combination of the variables with each flow-point. This is an extension of the idea previously presented by Sohn and van-Gelder [32]. The existence of such a function is, of course, a sound (but incomplete) criterion for termination, and restricting the data to integers maintains its soundness.

(6) The BinTerm analyzer of Spoto, Mesnard and Payet analyzes abstract programs and serves as a back-end to the Java Bytecode termination analyzer Julia [33]. The abstraction that it uses is a transition system with polyhedral constaints, and it applies a selection of strategies, which it tries one by one. The first two correspond to the two methods of Terminweb, slightly modified since the domain is now the integers. Thus, both for polyhedral transitions and for MCs, its local test is based on a search for an affine ranking function. In the MC domain, such a test would be complete in the stable case, but is not complete in general. The third method used is the method of Mesnard and Serebrenik.

(7) Colón and Sipma [14] is representative of a series of works that ostensibly target imperative programs, but work, in fact, on a constraint-transition system with linear (affine) constraints (the domain may be assumed to be the rationals, the reals or the integers) and search for global ranking functions of the lexicographic-linear type, using linear programming techniques.

Alias et al. [2] use the same general approach, but their class of ranking functions is more general (specifically, each q.r.f. associates a linear expression with every flow-point, whereas in [14], a q.r.f. is a single expression throughout a SCC). The lexicographic approach is, of course, more general than just looking for a single affine global ranking function (as in [28]), but still does not guarantee completeness for MCSs, where the ranking function sometimes has to depend on the order relations of the variables, and so is not linear (a simple example is a ranking function $\min(x, y)$).

Both methods rely on polyhedral invariants associated with a flow-point (in [2], they are part of the abstract program, and would be generated by a front-end; in [14], they are recomputed in each stage of the algorithm, which may improve its precision).

(8) Noting that monotonicity constraints are a special case of polyhedral constraints, it is natural to look for other interesting subclasses, possibly richer than monotonicity constraints. The class of *difference constraints* is defined by constraints of the form $x - y \leq c$. Termination of such constraint transition systems is shown undecidable in [4]. Decidability in PSPACE is proved for a restricted subclass, called *fan-in free δSCT*. This class is incomparable to MCSs (their intersection is fan-in free SCT).

(9) There are several published works that address a special subclass of constraint transition systems: simple loops, namely transition systems with only one control location (flow point). Moreover, some of them consider a single-path loop, consisting of one abstract transition only. This appears like a far-fetched restriction, but nonetheless, such simple loops can be complex enough to merit theoretical interest, and practically, an algorithm to decide termination of such a loop can be used as the local test in a closure-based algorithm, or an algorithm based on counter-example based search (see [16] for a well-known example of the approach).

Single-path loops, represented with polyhedral constraints, are handled using linear ranking functions in [29]. Bradley, Manna and Sipma [10] extend this to a multi-path loop, and show how to find a lexicographic-linear ranking function. In [11], the approach is generalized so that the components of the ranking tuple are not required to be quasi-ranking functions, but only "eventually" quasi-ranking, that is, they may ascend initially but must eventually descend (as in Example 2.11).

A few of the works mentioned also generate global ranking functions. [2] generates lexicographic-linear ranking functions. This is also the case with [14], although they are not explicit about it, and their class of functions is more restricted. In [28], the class is restricted to affine functions. All the works mentioned for analyzing simple loops are ranking-function based, but only those in [10, 11] may be truly called "global" since they apply to all the paths of a multi-path loop.

9. Conclusion and Research Questions

We studied the MCS abstraction, an appealing extension of the Size-Change Termination framework, that can be used for termination analysis in the integer domain. We showed how several elements of the theory developed in the well-founded model can be achieved in the integer case: sound and complete termination criteria, closure-based algorithms and the construction of ranking functions in singly-exponential time. Global ranking functions may be useful for certified termination [21, 15, 20] and cost analysis [1, 2], and the complexity achieved here is better than what has been published before for SCT.

Hopefully, this paper will trigger further research, moving towards the practical application of the theory presented. Some of the systems mentioned in the last section can gain an increase in precision by incorporating a complete decision procedure for monotonicity constraints, and it is encouraging that abstraction of a concrete program to monotonicity constraints either exists already in these systems or can be added with very little effort, typically because *richer* domains are already used (such as affine relations).

The algorithms in this article were aimed at getting the theoretical results with a minimum of complications. They can certainly be improved in practice (as discussed in the conclusion of [6]).

Cases like Example 2.10 suggest that it may be worthwhile to treat Boolean variables as such, so that they do not get entangled with the integer variables in the course of elaboration, creating an unnecessary combinatorial explosion. A better idea is to extend the MCS abstraction to include Boolean variables and extend the termination criteria and algorithms to account for them precisely. This may be a useful extension in practice, and moreover, it allows for adding information that does not come straight-forwardly from the program, in the form of "invented" Boolean variables—leveraging abstraction techniques used in the area of model checking.

Here are a few other directions for extension of this work:

(1) Investigating extensions of the constraint domain, particular to the integers (i.e., not appropriate for general SCT). An example is difference constraints (mentioned in the last section).
(2) Proceeding from termination analysis to analysis of a program's complexity [1, 2].
(3) The idea of using multiple abstractions (one may speak of abstractions of varying refinement) in a single tool in quite enticing. One can also consider an abstraction-refinement loop [16] which allows for eliminating spurious counterexamples by specialized tools, while using size-change analysis as a backbone.

Acknowledgments. The author thanks the APL group at DIKU (the Computer Science department at the University of Copenhagen), where part of this work was done, for hospitality, and the anonymous referees, whose thorough reviews and suggestions contributed significant improvements to this paper.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] Amir M. Ben-Amram. Size-change termination, monotonicity constraints and ranking functions. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 20th International Conference, CAV 2009, Grenoble, France*, volume 5643 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2009.

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

[7] 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.

[8] Amir M. Ben-Amram and Chin Soon Lee. Size-change analysis in polynomial time. *ACM Transactions on Programming Languages and Systems*, 29(1), 2007.

[9] Amir M. Ben-Amram and Chin Soon Lee. Ranking functions for size-change termination II. *Logical Methods in Computer Science*, 5(2), 2009.

[10] Aaron Bradley, Zohar Manna, and Henny Sipma. Linear ranking with reachability. In Kousha Etessami and Sriram Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 247–250. Springer Berlin / Heidelberg, 2005.

[11] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Proc. 32nd International Colloquium on Automata, Languages and Programming*, volume 3580 of *Lecture Notes in Computer Science*, pages 1349–1361. Springer Verlag, 2005.

[12] 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.

[13] 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).

[14] Michael Colón and Henny Sipma. Practical methods for proving program termination. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 442–454. Springer, 2002.

[15] Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Certification of automated termination proofs. In Boris Konev and Frank Wolter, editors, *Frontiers of Combining Systems, 6th International Symposium, FroCoS 2007, Liverpool, UK, Proceedings*, volume 4720 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2007.

[16] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI), Ottawa, Canada*, pages 415–426. ACM, 2006. Terminator.

[17] Nachum Dershowitz, Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1–2):117–156, 2001.

[18] Seth Fogarty and Moshe Y. Vardi. Büchi complementation and Size-Change Termination. 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 16–30. Springer, 2009.

[19] Matthias Heizmann, Neil D. Jones, and Andreas Podelski. Size-change termination and transition invariants. In Radhia Cousot and Matthieu Martel, editors, *Proc. 17th Int'l Static Analysis Symposium (SAS), Perpignan, France*, volume 6337 of *Lecture Notes in Computer Science*, pages 22–50. Springer-Verlag, 2010.

[20] Adam Koprowski and Hans Zantema. Certification of proving termination of term rewriting by matrix interpretations. In Viliam Geffert, Juhani Karhumäki, Alberto Bertoni, Bart Preneel, Pavol Návrat, and Mária Bieliková, editors, *SOFSEM 2008: Theory and Practice of Computer Science, 34th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, Proceedings*, volume 4910 of *Lecture Notes in Computer Science*, pages 328–339. Springer, 2008.

[21] 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.

[22] Chin Soon Lee. Ranking functions for size-change termination. *ACM Transactions on Programming Languages and Systems*, 31(3), April 2009.

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

[24] 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.

[25] Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. Proving termination for logic programs by the query-mapping pairs approach. In Maurice Bruynooghe and Kung-Kiu Lau, editors, *Program Development in Computational Logic: A Decade of Research Advances in Logic-Based Program Development*, volume 3049 of *Lecture Notes in Computer Science*, pages 453–498. Springer, 2004.

[26] Seymour Lipschutz. *Schaum's outline of theory and problems of set theory and related topics.* McGraw-Hill, 1998.

[27] 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.

[28] Frédéric Mesnard and Alexander Serebrenik. Recurrence with affine level mappings is p-time decidable for CLP(R). *TPLP*, 8(1):111–119, 2008.

[29] Andreas Podelski and Andrey Rybalchenko. A complete method for synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI 2003: Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.

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

[31] Ariel Snir. Termination and ranking function generation for monotonicity constraints systems. Technical report, Tel-Aviv Yaffo Academic College, 2010.

[32] Kirack Sohn and Allen Van Gelder. Termination detection in logic programs using argument sizes (extended abstract). In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SOGART Symposium on Principles of Database Systems (PODS), May 1991, Denver, Colorado*, pages 216–226. ACM Press, 1991.

[33] 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.