Computing Termination Conditions of Iterative Programs

Wided Ghardallou\textsuperscript{a}, Lamia L. Jilani\textsuperscript{b}, Fairouz Tchier\textsuperscript{c}, Ali Jaoua\textsuperscript{d}, Marcelo Frías\textsuperscript{e}, Jules Desharnais\textsuperscript{f}, Nafi Diallo\textsuperscript{g}, Ali Mili\textsuperscript{g}

\textsuperscript{a}University of Tunis El Manar, Tunis, Tunisia
\textsuperscript{b}Graduate Institute of Management, Bardo, Tunisia
\textsuperscript{c}King Saud University, Riyadh, KSA
\textsuperscript{d}Qatar University, Doha, Qatar
\textsuperscript{e}Institute of Technology of Buenos Aires, Argentina
\textsuperscript{f}Laval University, Quebec City, Canada
\textsuperscript{g}New Jersey Institute of Technology, Newark, NJ

Abstract

It is common to use invariant assertions to analyze functional properties of loops (such as partial correctness) and to use variant functions to analyze operational properties of loops (such as termination). In this paper, we explore an orthogonal approach, based on invariant relations, which enables us to reason about functional properties as well as termination in a uniform manner; also, our definition of termination encompasses not only the property that the loop executes a finite number of iterations, but also the property that every single iteration of the loop executes without raising an exception and/or causing an abort. We present a general theorem that provides a necessary condition of termination (in the sense of having a finite number of iterations, and in the sense of avoiding aborts), and discuss under what circumstances the given condition can be deemed sufficient, in addition to being provably necessary.

Key words: while loops, invariant assertions, invariant relations, termination conditions, abort-freedom, a logic for loop termination.

1. Introduction: Modeling Termination

1.1. Analyzing Loops with Invariant Relations

In the broad picture of the analysis of software artifacts, loops mobilize the lion’s share of attention, because they are typically the focus of product complexity, and (consequently) the locus of most programming faults. In [53], we propose a theory for computing...
or approximating the function of a loop, using the concept of invariant relation, whose properties we analyze in [56]. In [27], we expand the application of invariant relations by showing that in addition to being useful for computing and approximating loop functions, they can also be used to compute several other relevant attributes of while loops, such as: weakest preconditions, strongest postconditions, invariant assertions, invariant functions, necessary conditions of correctness, sufficient conditions of correctness, and necessary conditions of termination. The use of invariant relations to generate necessary conditions of termination is the focus of this paper. We define termination in the broadest sense possible, to encompass not only the condition that the number of iterations is finite, but also the condition that each individual iteration completes its execution without causing an abort. In particular, we consider in turn the following conditions, each of which can be modeled by a different instance of a generic invariant relation (which we present in this paper).

- Array references out of bounds.
- Illegal operations, such as division by zero, logarithm of a non-positive number, square root of a negative number, etc.
- Arithmetic overflow (whenever the result of some operation on two representable values is not representable in the computer).
- Illegal pointer reference (whenever the program attempts to reference the information addressed by a nil pointer).

While different programming languages may have different policies on how to handle these conditions, we view them uniformly as situations whose outcome is not defined, and we refer to them as abort conditions; also, we refer to the property of avoiding these conditions as abort-freedom. Other authors refer to these conditions as safety properties, but we adhere to the terminology introduced by Laprie et. al. [45, 44, 46, 2], in which safety refers to correctness with respect to a safety-critical requirement, rather than freedom from abort conditions; needless to say, safety (in the sense of Laprie et. al.) is neither necessary nor sufficient for abort-freedom.

1.2. Merging Termination and Abort-Freedom

It is customary to study loop termination and abort-freedom as two separate properties; in this paper, we study them jointly, for the following reasons:

- From the standpoint of an analyst, termination and abort-freedom are indistinguishable in the sense that they both represent the absence of a well-defined final state. If execution of the program on some initial state fails to produce a final state, then we declare that that the program does not terminate normally for state s; whether it fails to terminate normally because it enters an infinite loop or because it attempts an operation whose outcome is undefined makes no difference if we are interested in the existence of a well-defined final state.

- We take a denotational semantic approach to the analysis of programs, where the semantics of a program is captured by the function that the program defines on its state space. In such a semantic model, the set of states to which the program associates a final state is merely the domain of the program’s function. The domain of this function excludes any initial state for which the program fails to associate a final state, regardless of why it fails to do so (whether it fails due to an infinite loop or due to an illegal operation).
The method we use to characterize initial states that the program maps into final states is based on a theorem that converts invariant relations into necessary conditions of termination [27]. While we may, for the sake of illustration, generate invariant relations that are geared specifically toward abort-freedom or toward bounded iterations, invariant relations cannot be neatly classified according to which aspect they capture, and most typically invariant relations capture both aspects to varying degrees (consider that the intersection of invariant relations is an invariant relation).

Perhaps the most compelling argument in favor of an integrated approach to the analysis of termination and abort-freedom is merely this: most typically, the condition under which a loop terminates after a finite number of abort free iterations is not the conjunction of the condition of termination (in the sense of bounded number of iterations) and the condition of abort-freedom. As we will see in multiple examples throughout this paper, the conditions we generate is not the conjunction of two separate clauses, but rather weaves clauses of abort-freedom and clauses of bounded iterations in non-trivial ways.

Not only do we believe that it is advantageous to model termination and abort-freedom in the same framework, but we must recognize that from the standpoint of our semantic model, we could not separate them if we wanted to, since they are both modeled by the domain of the program’s function.

1.3. Characterizing our Approach

Our approach is in sharp contrast with traditional approaches [21, 14, 10, 15, 61, 68], which analyze the functional properties of iterative programs by means of invariant assertions and analyze termination by means of variant functions. In our approach, both of these aspects are modeled by means of invariant relations; the same artifact that we use to compute or approximate the function of the loop is used to compute or approximate the domain of this function. This is only fitting, given that the domain of the loop function is an attribute of the function, rather than being a separate/ orthogonal attribute of the loop. The difference between computing the function of a loop (which we discuss in [53]) and computing its termination condition (i.e. the domain of its function) is that the former requires that we compute all the invariant relations of the loop, whereas the latter requires only that we compute those invariant relations that are relevant to termination (as we shall see throughout the paper).

We can model termination (in the traditional sense, the property that the number of iterations is bounded) by deriving invariant relations that involve program variables that appear in the loop condition, as well as invariant relations that involve variables that affect the value of these (through assignment statements, for example). As for modeling abort-freedom, we present an original theorem, Theorem 3, which gives the general form of invariant relations that capture the condition under which a loop executes without causing an abort. This theorem links in effect the domain of the loop with the domain of the loop body, in the following sense: in order for an initial state to be in the domain of the loop, it has to ensure that all intermediate states generated from it by the execution of loop fall in the domain of the loop body; this ensures that the execution proceeds without aborts.
1.4. Characterizing our Results

In addition to highlighting how our approach is distinct from other approaches (as we did cursorily in the previous section), we highlight, in this section, how our results are distinct from those of other research and development efforts. To this effect, we consider the following loop on integer variables $i$, $x$, and $y$, and we wish to compute the condition under which this loop terminates without attempting a division by zero; in other words, we want the condition under which this loop terminates after a finite number of iteration, and such that no single iteration will fail to execute properly.

```
while (i!=0) {i=i+2; x=x-5; y=y-y/x;}
```

Application of Theorems 2 and 3 yields the following condition:

$$(i = 0) \lor (i < 0 \land i \mod 2 = 0) \land (x < 5 \lor (5 < x < -\frac{5 \times i}{2}) \land x \mod 5 \neq 0) \lor x > -\frac{5 \times i}{2}).$$

If we analyze this condition, we find that it stipulates that either $(i = 0)$ (in which case the loop does not iterate at all) or $(i < 0 \land i \mod 2 = 0)$ (in which case the number of iterations is finite —note that if $i$ is odd, then it will skip over zero and never terminate) then either $x < 5$ (in which case $x$ never takes value 0 as it is decremented by 5 at each iteration) or $(5 < x < -\frac{5 \times i}{2})$ (in which case $x$ flies over zero on its way down but does not hit zero) or $(x > -\frac{5 \times i}{2})$ (in which case $i$ reaches 0 and terminates the loop before $x$ gets near zero). If $(i > 0)$ or if $(i < 0 \land i \mod 2 \neq 0)$ then this loop does not terminate since $i$ never hits 0 as it is incremented by 2 at each iteration.

We know of no other approach or tool that performs the same analysis (this matter will be discussed in greater detail in Section 7.2): Some tools compute the condition of termination of while loops; some tools prove the termination of iterative programs for a given precondition; other tools analyze iterative programs and alert users to the possibility of non-termination or the possibility that the program may cause an abort. But no tool that we know of computes the condition under which an iterative program terminates after a finite number of iterations without causing an abort. The purpose of this paper is to develop mathematical foundations that can serve as a blueprint for such a tool; we do have a prototype that we use for the purpose of experimentation, but it is not a tool by any stretch; in Section 7.1.2 we discuss our plan for developing and evolving such a tool, along with the challenges that arise in the process. Some sample demos of the current prototype can be seen at: https://selab.njit.edu/tools/fxloop.php.

1.5. Outline

In section 2 we briefly introduce elements of relational mathematics that we use throughout this paper, and in section 3 we introduce the concept of invariant relation, and discuss how this concept can be used to analyze loops, and what is its relation to the more widely used concept of invariant assertion [32]. In section 4 we discuss a general framework for analyzing the termination of programs, which we then specialize to iterative programs, by means of a necessary condition of termination. In section 5, we consider several conditions of abort avoidance and apply the necessary condition of termination to them, then we discuss in section 6 under what condition the computed necessary conditions can be deemed sufficient. Finally in section 7 we summarize our findings, compare them to related work, and sketch directions of future research.
2. Mathematical Background

We assume the reader familiar with relational mathematics; the purpose of this section is to introduce some definition and notations, inspired from [6].

2.1. Definitions and Notations

We consider a set $S$ defined by the values of some program variables, say $x$ and $y$; we denote elements of $S$ by $s$, and we note that $s$ has the form $s = (x, y)$. We denote the $x$-component and (resp.) $y$-component of $s$ by $x(s)$ and $y(s)$. For elements $s$ and $s'$ of $S$, we may use $x$ to refer to $x(s)$ and $x'$ to refer to $x(s')$. We refer to $S$ as the space of the program and to $s \in S$ as a state of the program. A relation on $S$ is a subset of the cartesian product $S \times S$. Constant relations on some set $S$ include the universal relation, denoted by $I$, the identity relation, denoted by $I$, and the empty relation, denoted by $\phi$.

2.2. Operations on Relations

Because relations are sets, we apply set theoretic operations to them: union ($\cup$), intersection ($\cap$), and complement ($\overline{R}$). Operations on relations also include: The converse, denoted by $R$, and defined by $\overline{R} = \{(s, s')|(s', s) \in R\}$. The product of relations $R$ and $R'$ is the relation denoted by $R \circ R'$ (or $RR'$) and defined by $R \circ R' = \{(s, s')|\exists s'' : (s, s'') \in R \land (s'', s') \in R\}$. The nucleus of relation $R$ is the relation denoted by $\mu(R)$ and defined by $\mu(R) = RR$. The $n$th power of relation $R$, for natural number $n$, is denoted by $R^n$ and defined by $R^0 = I$, and $R^n = R \circ R^{n-1}$, for $n \geq 1$. The transitive closure of relation $R$ is the relation denoted by $R^+$ and defined by $R^+ = \{(s, s')|\exists i > 0 : (s, s') \in R^i\}$. The reflexive transitive closure of relation $R$ is the relation denoted by $R^*$ and defined by $R^* = I \cup R^+$. We admit without proof that $R^* R^* = R^*$ and that $R^* R^+ = R^+ R^* = R^+$. The pre-restriction (resp. post-restriction) of relation $R$ to predicate $t$ is the relation $\{(s, s')|t(s) \land (s, s') \in R\}$ (resp. $\{(s, s')|(s, s') \in R \land t(s')\}$). Given a predicate $t$, we denote by $T$ the relation defined as $T = \{(s, s')|t(s)\}$. The domain of relation $R$ is defined as $\text{dom}(R) = \{s|\exists s' : (s, s') \in R\}$, and the range of $R$ ($\text{rng}(R)$) is the domain of $\overline{R}$. We apply the usual conventions for operator precedence: unary operators are applied first, followed by product, then intersection, then union.

2.3. Properties of Relations.

We say that $R$ is deterministic (or that it is a function) if and only if $\overline{RR} \subseteq I$, and we say that $R$ is total if and only if $I \subseteq R$, or equivalently, $RL = L$. A vector $V$ is a relation that satisfies $VL = V$; in set theoretic terms, a vector on set $S$ has the form $C \times S$, for some set $C$ of $S$; we use vectors as a relational representation of sets. We note that for a relation $R$, $RL$ represents the vector $\{(s, s')|s \in \text{dom}(R)\}$; we use $RL$ as the relational representation of the domain of $R$. A relation $R$ is said to be reflexive if and only if $I \subseteq R$, transitive if and only if $RR \subseteq R$ and symmetric if and only if $R = \overline{R}$. We admit without proof that the transitive closure of a relation $R$ is the smallest transitive superset of $R$ and that the reflexive transitive closure of $R$ is the smallest reflexive transitive superset of $R$. A relation that is reflexive, symmetric and transitive is called an equivalence relation. The nucleus of a deterministic relation $f$ can be written as: $\mu(f) = \{(s, s')|f(s) = f(s')\}$ and is an equivalence relation. A relation $R$ is said to be antisymmetric if and only if $RR \subseteq \phi$, i.e. it has no pairs of the form $(s, s)$. A relation $R$ is said to be inductive if and only there exists a vector $A$ such that $R = A \cup A$; inductive relations can be written as $R = \{(s, s')|a(s) \Rightarrow a(s')\}$ for some predicate $a$ on $S$. 
3. Invariant Relations

Informally, an invariant relation of a while loop of the form \( w : \text{while } (t) \{ b \} \) is a relation that contains all (but not necessarily only) the pairs of program states that are separated by an arbitrary number of iterations of the loop. Invariant relations are introduced in [55, 53], their relation to invariant assertions is explored in detail in [56], and their applications are explored in [27]. Among their main attributes, we cite:

- Unlike invariant assertions, they depend exclusively on the loop, and do not depend on the context in which the loop is used.
- Whereas invariant assertions can only be used to prove the partial correctness of a loop, invariant relations can be used to prove the total correctness of a loop.
- Whereas invariant assertions can only be used to prove that a loop is correct, invariant relations can be used to prove that a loop is correct (if the invariant relation subsumes the candidate specification), but can also be used to prove that a loop is incorrect (if the invariant relation is incompatible with the candidate specification).
- Invariant relations enable us to model the termination of while loops in a broad sense, in such a way as to encompass not only the condition that the number of iterations is finite, but also the condition that every single iteration executes without causing an abort.
- In [56] we find that invariant relations are a more general concept than invariant assertions, in the following sense: all invariant assertions stem from invariant relations, but only a small class of invariant relations stem from (can be derived from) invariant assertions.

Before we introduce a formal definition of invariant relations, we present some definitions and notations pertaining to loop semantics.

3.1. Program Semantics

Given a program \( g \) on space \( S \), we let the function of \( g \) be denoted by \( G \) and defined as the set of pairs \((s, s')\) such that if \( g \) starts execution on state \( s \) then it terminates normally in state \( s' \). By terminates normally, we mean that the program terminates after a finite number of operations, without causing an abort (resulting from an illegal operation), and returns a well-defined final state. From this definition it stems that \( \text{dom}(G) \) is the set of states \( s \) such that if execution of \( g \) starts in state \( s \) then it terminates (normally). The termination condition of program \( g \) is the predicate \( s \in \text{dom}(G) \); note that we talk about the termination condition of any program, not exclusively of iterative programs. As a convention, we represent programs by lower case letters and their function by the same letter in upper case.

We consider while loops written in some C-like programming language, and we quote the following theorem, due to [53], which we use as the semantic definition of a while loop.

**Theorem 1.** We consider a while statement of the form \( w : \text{while } (t) \{ b \} \). Then its function \( W \) is given by:

\[
W = (T \cap B)^* \cap T,
\]

where \( B \) is the function of \( b \), and \( T \) is the vector defined by: \( \{(s, s')|t(s)\} \).

The main difficulty of analyzing while loops is that we cannot, in general, compute the reflexive transitive closure of \((T \cap B)\) for arbitrary values of \( T \) and \( B \).
3.2. Definitions

If we knew how to compute reflexive transitive closures of arbitrary functions and relations, then we would apply Theorem 1 to derive the function of the loop, and do away with invariant relations, invariant assertions, and all the other loop artifacts [27]; but in general we do not. The interest of invariant relations is two-fold:

- First, they enable us to compute the function of a loop in a stepwise manner, by successive approximations; this is explored in [53].
- Second, perhaps more interestingly, they enable us to answer many questions about the loop without having to compute its function; this is explored in [27]. In particular, they enable us to compute termination conditions of while loops, which we explore in this paper.

We define invariant relations formally as follows.

**Definition 1.** Given a while loop of the form \( w : \text{while (t)} \{ b \} \) on space \( S \), we say that relation \( R \) is an invariant relation for \( w \) if and only if it is a reflexive and transitive superset of \((T \cap B)\).

The interest of invariant relations is that they are approximations of \((T \cap B)^*\), the reflexive transitive closure of \((T \cap B)\); smaller invariant relations are better, because they represent tighter approximations of the reflexive transitive closure; the smallest invariant relation is \((T \cap B)^*\). The following proposition stems readily from the definition.

**Proposition 1.** Given a while loop of the form \( w : \text{while (t)} \{ b \} \) on space \( S \), we have the following results:

1. The relation \((T \cap B)^*\) is an invariant relation for \( w \).
2. If \( R \) is an invariant relation for \( w \), then \((T \cap B)^* \subseteq R\).
3. If \( R_0 \) and \( R_1 \) are invariant relations for \( w \) then so is \( R_0 \cap R_1 \).

To illustrate the concept of invariant relation, we consider the following while loop on integer variables \( n, f, \) and \( k \):

\[ w: \text{while (k!=n)} \{ k=k+1; f=f*k; \}. \]

We consider the following relation:

\[ R = \left\{ (s, s') | \frac{f}{k!} = \frac{f'}{k'!} \right\}. \]

This relation is reflexive and transitive, since it is the nucleus of a function; to prove that it is a superset of \((T \cap B)\) we compute the intersection \( R \cap (T \cap B) \) and easily find that it equals \((T \cap B)\). Other invariant relations include \( R' = \{(s, s')| n' = n\} \), and \( R'' = \{(s, s')| k \leq k'\} \).

3.3. Invariant Relations and Invariant Assertions

In [56], we have analyzed the relationships between invariant assertions [32], invariant functions [54], and invariant relations [55, 53]. In this section, we briefly present and illustrate the results of [56] as they pertain to the relationship between invariant assertions and invariant relations. First, we present a relational definition of invariant assertions [32], by representing assertions in relational form as vectors.
Definition 2. A vector \( A \) is said to be an invariant assertion for the while loop \( w: \texttt{while} \ t \{ b \} \) with respect to precondition \( \phi \) and postcondition \( \psi \) if and only if it satisfies the following conditions:

- \( \phi \subseteq A \),
- \( A \cap (T \cap B) \subseteq \hat{A} \),
- \( A \cap T \subseteq \psi \).

Before we compare invariant assertions and invariant relations, it is important to remember that invariant relations are intrinsic to the loop whereas invariant assertions depend not only on the loop but also on its context, as defined by its precondition and postcondition. With this clarification in mind, we can summarize the relationship between invariant relations and invariant assertions by the following propositions, which are due to [56]:

- If \( R \) is an invariant relation for the while loop \( w \) and \( \phi \) is an arbitrary vector on \( S \) (that represents a precondition of the loop), then the vector \( A = \hat{R}\phi \) is an invariant assertion for \( w \) with respect to precondition \( \phi \) and postcondition \( \hat{R}\phi \cap T \). This Proposition shows how we can map an invariant relation and precondition into an invariant assertion.

- If \( A \) is an invariant assertion for the while loop \( w \) then the relation \( R = A \cup \hat{A} \) is an invariant relation for \( w \). This Proposition shows how we can map an invariant assertion into an invariant relation.

- Given an invariant assertion \( A \) for the while loop \( w \), there exists a vector \( \phi \) (precondition) and an invariant relation \( R \) such that \( A = \hat{R}\phi \). In other words, all invariant assertions stem from invariant relations, and all invariant assertions are the product of the inverse of an invariant relation (a factor that is intrinsic to the loop) with a vector (that represents the precondition of the loop, and reflects the context in which the loop is used). This structure can be used to streamline the generation of invariant assertions [8, 20, 22, 25, 35, 39, 40, 61, 64, 33, 41, 48, 70, 42, 34, 26].

- Given an invariant relation \( R \) for the while loop \( w \), if \( R \) is an inductive relation then there exists an invariant assertion \( A \) such that \( R = \overline{A} \cup \hat{A} \). In other words, all inductive invariant relations stem from invariant assertions.

Because all invariant assertions stem from invariant relations, but only inductive invariant relations (a small class of invarian relations) stem from invariant assertions, we argue that invariant relations are a more general concept than invariant assertions.

As an illustration of this discussion, we consider the sample factorial program presented above (in section 3.2), and we consider the invariant relation

\[
R = \left\{ (s, s') \middle| \frac{f}{k!} = \frac{f'}{k'!} \right\}.
\]

If we take the precondition represented by the vector

\[
\phi = \{(s, s')|f = 1 \land k = 0\}
\]

then we can compute the corresponding invariant assertion and postcondition constructively, as follows:

\[
A = \hat{R}\phi = \{(s, s')|f = 1 \}\cup \{(s, s')|f = k!\} = \{(s, s')|f = k!\},
\]

\[
\psi = \hat{R}\phi \cap T = A \cap T = \{(s, s')|f = k! \land k = n\} = \{(s, s')|f = n! \land k = n\}.
\]
The interested reader may choose another (arbitrary) precondition \( \phi \) and be assured that by applying the proposed formulae, she/he will find the corresponding invariant assertion and postcondition.

### 3.4 Generating Invariant Relations

In order to put our research into practice, we have developed a prototype tool that generates invariant relations of loops written in C-like languages (C, C++, Java). The design and operation of this tool is beyond the scope of this paper, and is discussed in other sources [53, 37, 27, 56, 36]; in this paper, we briefly present some details of this tool, for the purpose of making this paper self-contained.

**Proposition 2.** Let \( w: \textbf{while} \ (t) \ \{ b \} \) be a while loop on space \( S \). The relation \( R = I \cup T(T \cap B) \) is an invariant relation for \( w \).

This relation can be computed constructively from \( T \) and \( B \), and includes pairs \((s, s')\) such that \( s' = s \) (case when no iterations are executed) and pairs \((s, s')\) such that \( s \) verifies \( t \) and \( s' \) is in the range of \((T \cap B)\) (case when one or more iterations are executed). We refer to it as the elementary invariant relation of \( w \), and in practice we generate it systematically whenever we analyze a loop. To generate other relations, we proceed by pattern matching: We map the source code of loops (in C, C++, or Java) onto relational notation, then we match clauses of their relational representation against code patterns for which we know invariant relation patterns. Whenever a match is successful, we generate an invariant relation by instantiating the matching invariant relation pattern with the variable substitutions of the match.

The aggregate made up of a code pattern and the corresponding invariant relation pattern is called a recognizer. We distinguish between 1-recognizers, whose code pattern includes a single statement, 2-recognizers, whose code pattern includes two statements, and 3-recognizers, whose code pattern includes three statements; to keep combinatorics under control, we seldom use recognizers of more than 3 statements. The machinery that maps source code into internal relational notation is in place, as is the machinery that maintains the database of recognizers and matches the relational representation of a loop against recognizers to generate invariant relations. What determines the capability of our tool is the set of recognizers that are stored in its database. In the remainder of this paper, whenever we talk about an invariant relation that would fulfill some role (e.g. identify exceptional conditions that preclude normal termination), it is understood that we can deploy this invariant relation in practice by including its recognizer in the database of our tool.

One may argue that our approach lacks generality because it depends on a pre-coded database of recognizers. We put forth the following observations:

- It is impossible to build a system to analyze programs without codifying the programming knowledge and the domain knowledge that are needed for this task; we argue that the recognizers are our way to capture the relevant programming knowledge and domain knowledge.
- We are currently exploring ways to do away with pre-coded recognizers for simple numeric calculations; indeed, many of our numeric invariant relations can be generated automatically from the source code by converting the code to recurrence relations (according to the work of Janicki and Carrette [9]) and eliminating the recurrence variable.
• The focus of this paper is the generation of termination/ abort-freedom conditions from invariant relations; we deploy a prototype as a proof-of-concept artifact, but automation is not the focus of this work.

4. A Logic for Loop Termination

The purpose of this section is to lay a foundation for the analysis of loop termination by means of two theorems: the first gives a general formula for mapping any invariant relation into a necessary condition of termination; and the second theorem gives guidance on how to generate invariant relations to target specific abort freedom properties.

4.1. A Necessary Condition of Termination

We consider a while loop \( w \) of the form \( \text{while } \{ t \} \{ b \} \) on space \( S \), and we are interested in computing its domain, which we represent by the vector \( WL \) (where \( W \) is the function of \( w \) and \( L \) is the universal relation). The following theorem, due to [27], gives a necessary condition of termination.

**Theorem 2.** We consider a while loop \( w \) of the form \( \text{while } \{ t \} \{ b \} \) on space \( S \), and we let \( R \) be an invariant relation for \( w \). Then

\[
WL \subseteq R^T.
\]

This Theorem converts an invariant relation of \( w \) into a necessary condition of termination; we seek to derive the smallest possible invariant relations, in order to approximate or achieve the necessary and sufficient condition of termination. The proof of this Theorem is given in [27]; it stems readily from Theorem 1, and from relational identities. In practice, we compute the termination condition of a loop by means of the following steps:

- Using the invariant relation generator, we generate all the invariant relations we can recognize; whenever a code pattern of the loop matches a recognizer pattern from our recognizer database, we generate the corresponding invariant relation. These relations are represented in Mathematica syntax (©Wolfram Research).
- We compute the intersection of the invariant relations we are able to generate, by merely taking the conjunct of their Mathematica representation.
- Given \( R \) the aggregate invariant relation computed above, we simplify the following logical formula, which is the logical representation of the formula of Theorem 2.

\[
\exists s' : (s, s') \in R \land \neg t(s').
\]

The result is a logical expression in \( s \), which represent a necessary condition of termination of the loop.

As an illustration of this Theorem, we consider the sample factorial loop discussed earlier, namely:

\[
w: \text{while } \{ k!=n \} \{ k=k+1; f=f*k; \}.
\]

We consider the following invariant relation of \( w \): \( R = \{(s, s')|k \leq k'\} \). Application of Theorem 2 to this invariant relation yields the following necessary condition: \( k \leq n \). Indeed, this condition is necessary to ensure that the number of iterations of the loop is finite.

As a less trivial example, consider the following loop on integer variables \( i \), \( j \), and \( k \).
\[ \text{while } (i > 1) \{ j = j + 1; i = i + 2j - 1; k = k - 1; \} \]

The parameters of this loop are:
- \( T = \{(s, s')|i > 1\} \).
- \( B = \{(s, s')|j' = j + 1 \land i' = i + 2j + 1 \land k' = k - 1\} \).

We derive the following invariant relations (using recognizers from our existing database [37]):
- The elementary invariant relation, \( R_0 = I \cup T(T \cap B) \).
- Symmetric invariant relations: \( R_1 = \{(s, s')|j + k = j' + k'\} \), \( R_2 = \{(s, s')|i - j^2 = i' - j'^2\} \).
- Antisymmetric invariant relations (one of them suffices, given that we already have \( R_1 \), but we write them both): \( R_3 = \{(s, s')|j' \geq j\} \), \( R_4 = \{(s, s')|k' \leq k\} \).

Taking their intersection \( R = R_0 \cap R_1 \cap R_2 \cap R_3 \cap R_4 \), and applying Theorem 2 to \( R \), we find the following termination condition:

\[ (i \leq 1) \lor (i > 1 \land j \leq -\sqrt{i - 1}). \]

This condition is provably a necessary condition of termination; we believe that it is also a sufficient condition of termination, because the invariant relations we have used to generate it capture all the relevant information for termination: relation \( R_0 \) captures relevant boundary conditions; relation \( R_1 \) captures the progression of the program state; relation \( R_2 \) links variable \( j \) which counts the number of iterations and variable \( i \), which is used in the loop condition. Note that relations \( R_3 \) and \( R_4 \) were redundant for our purposes, and are not needed to compute the termination condition, if we have \( R_0 \), \( R_2 \) and \( R_3 \). As an illustration, we consider a data sample that satisfies the termination condition, e.g. \( i = 10 \land j = -5 \) and a data sample that does not satisfy the condition, e.g. \( i = 10 \land j = 0 \), and verify that the first sample yields to termination and the second leads to an infinite loop.

4.2. \textit{Abort Freedom}

Theorem 2 converts any invariant relation into an approximation of (more precisely: a superset of) the domain of the while loop; in logical terms, this produces a necessary condition of termination. The domain of \( W \) is limited by failure of the loop to terminate, as well as failure of abort-prone statements to execute successfully; Theorem 2 applies equally well to either of these circumstances. Depending on our choice of invariant relations, we can capture one aspect of non-termination or the other, or a combination thereof. In this subsection, we present a general format of invariant relations that enable us to capture arbitrary aspects of abort-freedom (freedom from: array reference out of bounds, nil pointer reference, division by zero, arithmetic overflow, etc).

The following discussion builds an intuitive argument for the proposed theorem, and explains how we derived it. As a general rule, a program terminates whenever it is applied to a state within its domain, and fails to terminate otherwise. Hence, at a macro-level, the condition of termination of program \( g \) can merely be written as:

\[ s \in \text{dom}(G). \]

If \( g \) is a sequence of two subprograms, say \( g = g_1; g_2 \) then this condition can be rewritten as:

\[ s \in \text{dom}(G_1) \land G_1(s) \in \text{dom}(G_2). \]
We can prove by induction that if \( g \) is written as a sequence of arbitrary length, say \( g = (g_1; g_2; g_3; \ldots; g_n) \), then the condition of termination can be written as:

\[
s \in \text{dom}(G_1) \land G_1(s) \in \text{dom}(G_2) \land G_2(G_1(s)) \in \text{dom}(G_3) \land \ldots \land G_{n-1}(G_{n-2}(\ldots(G_3(G_2(G_1(s))))\ldots)) \in \text{dom}(G_n),
\]

or, equivalently, as:

\[
\forall h: 0 \leq h < n : G_h(G_{h-1}(\ldots(G_3(G_2(G_1(s)))))) \in \text{dom}(G_{h+1}). 
\]

If we specialize this equation to while loops, where all the \( G_i \)'s are instances of the loop body, we find the following equation:

\[
\forall h: 0 \leq h < n : (T \cap B)^h(s) \in \text{dom}(B). 
\]

In practice it is difficult to compute \((T \cap B)^h\) for arbitrary values of \( h \); fortunately, it is not necessary to compute them either, as usually only a small set of program variables is involved in characterizing abort freedom. Hence, we substitute in the above equation the term \((T \cap B)\) by a superset thereof (which we call \( B' \)), that captures only the transformation of abort-relevant variables. This equation can then be written as:

\[
\forall h: 0 \leq h < n : B'^h(s) \in \text{dom}(B). 
\]

We want to change this formula from a quantification on the number of iterations to a quantification on intermediate states; to this effect, we use the change of variables: \( u = (T \cap B)^h(s) \), and we represent the initial state (that corresponds to \( h = 0 \)) by \( s \) and the final state (that corresponds to \( h = n \)) by \( s' \). With these change of variables, the inequality \( 0 \leq h \) can be written as \((s, u) \in B'^*\), and the inequality \((h < n)\) can be written as \((u, s') \in B'^*\). Equation 3 can then be written as:

\[
\forall u : (s, u) \in B'^* \land (u, s') \in B'^+ \Rightarrow u \in \text{dom}(B). 
\]

Interestingly, this equation defines an invariant relation between \( s \) and \( s' \); this is the object of Theorem 3. Before we present this theorem and its proof, we write the proposed invariant relation in algebraic form.

\[
R = \{ \text{denotation} \}
\{ (s', s) | \forall u : (s, u) \in B'^* \land (u, s') \in B'^+ \Rightarrow u \in \text{dom}(B) \}
\{ \text{rewriting} u \in \text{dom}(B) \}
\{ (s', s) | \forall u : (s, u) \in B'^* \land (u, s') \in B'^+ \Rightarrow (u, s') \in BL \}
\{ \text{De Morgan} \}
\{ (s', s) | \exists u : (s, u) \in B'^* \land (u, s') \in B'^+ \land (u, s') \notin BL \}
\{ \text{Associativity} \}
\{(s', s) | \exists u : (s, u) \in B'^* \land (u, s') \in (B'^+ \cap BL) \}
\{ \text{Relational Product} \}
\{ B'^*(B'^+ \cap BL) \}.
\]

12
This discussion introduces, though it does not prove, the following theorem; its proof is given below.

**Theorem 3.** We consider a while loop \( w \) of the form \( \textbf{while} \ (t) \ \{ b \} \) on space \( S \), and we let \( B' \) be a superset of \((T \cap B)\). If \( B' \) satisfies the following conditions:
- \( B'^+ \) is anti-reflexive.
- The following relation \( Q = B'^+(B'^+ \cap V) \) is transitive, for an arbitrary vector \( V \).
- \( T \cap B \cap B'^+ B' = \phi \).

then \( R = (B'^+(B'^+ \cap BL)) \) is an invariant relation for \( w \).

This theorem provides, in effect (when applied in conjunction with Theorem 2), that if the loop terminates on initial state \( s \) (i.e. \( s \) is in \( \text{dom}(W) \)), then any intermediate state \( s' \) generated from \( s \) by an arbitrary number of iterations of the loop causes no abort at the next iteration (i.e. \( s' \) is in \( \text{dom}(B) \)). It is in this sense that this theorem links \( \text{dom}(W) \) and \( \text{dom}(B) \).

**Proof.** We have to show three properties of \( R \), namely reflexivity, transitivity, and invariance (i.e. that \( R \) is a superset of \((T \cap B)\)).

**Reflexivity.** In order to show that \( I \) is a subset of \( R \), we show that \( I \cap \overline{R} = \phi \). We find:

\[
I \cap \overline{R} = \phi
\]

**Transitivity.** Transitivity is a trivial consequence of the second condition of the theorem, by taking \( V = BL \).

**Invariance.** In order to prove that \((T \cap B) \subseteq R\), it suffices (by set theory) to prove that \((T \cap B) \cap \overline{R} = \phi \). To this effect, we analyze the expression \((T \cap B) \cap \overline{R} \). But first, we introduce a lemma to the effect that for any relation \( C \), \( C^+ C = C^+ C^+ \). Indeed, \( C^+ C^+ \) can be written \( CC^+ C \) by decomposing \( C^+ \) as \( CC^+ \) then as \( C^+ C \). Now, \( C^+ C^+ \) is equal to \( C^+ \cdot C^+ C^+ \subseteq C^+ C \) because of transitivity, and \( C^+ \subseteq C^+ C^+ \) (because \( I \subseteq C^+ \)). Hence \( C^+ C^+ = CC^+ C = C^+ C \). Now, we consider the expression \((T \cap B) \cap \overline{R} \).

\[
(T \cap B) \cap \overline{R} = \phi
\]
\[(T \cap B) \cap B^+(B^+ \cap BL) \subseteq \{\text{monotonicity}\}\]
\[(T \cap B) \cap B^+ B^+ = \{\text{lemma above}\}\]
\[(T \cap B) \cap B^+ B^+ = \{\text{by hypothesis}\}\]
\[\phi.\]  

qed

The first condition of this theorem ensures that \(B^\prime\) captures variant properties of \((T \cap B)\), hence does not revisit the same state after a number of iterations; we refer to this as the anti-reflexivity condition. The second condition ensures that the resulting relation is transitive (a necessary condition to be an invariant relation); this condition involves \(B^\prime\) and the structure of \(R\), but does not involve \(B\); we refer to this as the transitivity condition. The third condition ensures that \(B^\prime\), while approximating \((T \cap B)\), remains in unison with it, i.e. does not iterate faster than \((T \cap B)\); this condition is needed to ensure that \(R\) is a superset of \((T \cap B)\); we refer to it as the concordance condition. Note that there is a one-to-one correspondence between the properties of \(B^\prime\) and the resulting properties of \(R\): The anti-reflexivity of \(B^\prime^+\) yields the reflexivity of \(R\); the transitivity of \((B^\prime^+(B^+ \cap V))\) yields the transivity of \(R\) and the concordance of \(B^\prime\) yileds the invariance of \(R\) (i.e. the property that \((T \cap B)\) is a subset of \(R\)).

The interest of this theorem is that it appears to capture (in the form of an invariant relation) the property of abort-freedom of a while loop. To understand how it does that, consider the logical form of such invariant relations:

\[R = \{(s, s') | \forall u: (s, u) \in B^\prime^+ \land (u, s') \in B^\prime^+ \Rightarrow u \in \text{dom}(B)\},\]

where \(B^\prime\) is a superset of \(B\). In practice, we use \(B^\prime\) to approximate \(B\), by focusing on the variables that are of interest to us (that are involved in abort-prone statements) and recording how \(B\) transforms them. As for \(\text{dom}(B)\), we use it to record the abort condition of interest: for example, if we want to model the condition that arithmetic operations in the loop body do not cause overflow, then we let \(\text{dom}(B)\) include a clause to the effect that all operations produce a result within the range of representable values; if we want to model the condition that no division by zero arises in the execution of the loop body, then we include a condition in \(\text{dom}(B)\) that ensures that all divisors in \(B\) are non-zero; etc. So that relation \(R\), as written above, provides that all intermediate states generated by successive iterations of \(B\) cause no abort conditions. When we apply Theorem 2 using invariant relations generated by Theorem 3 (for various choices of \(B^\prime\) and various possible characterizations of \(\text{dom}(B)\)), we find conditions on the initial states of the loop, that ensure a terminating abort-free execution.

As we have discussed in section 3, smaller invariant relations are better. If we consider the template of invariant relations generated by Theorem 3,

\[R = B^\prime^+(B^+ \cap BL),\]

we find that \(R\) grows smaller (better) when \(B^\prime\) grows larger (i.e. provides a looser approximation of \(B\)) and when \(BL\) (i.e. the domain of \(B\)) grows smaller (i.e. we capture more and more abort conditions).
5. Computing Termination Conditions

In the previous section we presented two theorems: Theorem 2 converts any invariant relation into a necessary condition of termination of a while loop; and Theorem 3 proposes a general form for invariant relations that capture abort freedom properties. In this section we combine these two theorems to capture termination conditions that arise from different abort-prone statements. Note that nothing in Theorem 2 indicates whether the necessary condition of termination produced therein is putting a bound on the number of iterations or ensuring the absence of aborts; hence this theorem can be used with any invariant relation to enhance our estimate of the necessary and sufficient condition of termination. What Theorem 3 does is to give some indication on how to derive invariant relations that capture abort-freedom; in practice, we use a mix of invariant relations to obtain necessary conditions of termination, culminating (if our invariant relation is small enough) into a necessary and sufficient condition of termination.

5.1. Array Reference Out of Bounds

The first corollary of Theorem 3 (Proposition 3, below) is so trivial as to make the theorem look like an overkill; but it also helps the reader better understand the theorem. We consider a while loop $w$ on space $S$ and we assume that space $S$ includes an array $a$ of index range $[low...high]$. We assume that space $S$ also includes an index variable, say $k$, which is used to address the array. Then one issue of concern is to ensure that the array is not referenced outside of its bounds; the following proposition provides the appropriate invariant relation for this purpose.

**Proposition 3.** Let $w$ be a while loop of the form $w$: while $(t)$ { $b$ } on space $S$, where $S$ includes an array $a$ of index range $[low..high]$, and an index $k$ that is incremented by 1 at each iteration. Then the following relation is an invariant relation for $w$:

$$R = \{(s,s') | \forall h : k \leq h < k' \implies low \leq h \leq high\}.$$

**Proof.** This proposition is a special case of Theorem 3, in which we take we define $B'$ as $\{(s,s') | k' = k + 1\}$ and we let the domain of $B$ be defined as: $dom(B) = \{s | low \leq k \leq high\}$. We find that the transitive closure of $B'$ is $B'^+ = \{(s,s') | k \leq k'\}$, and that the reflexive transitive closure of of $B'$ is $B'^* = \{(s,s') | k \leq k'\}$. We must check the three conditions of Theorem 3: $B'^+$ is indeed anti-reflexive, since its intersection with identity is empty. To verify the transitivity condition, we consider a relation of the form $Q = B'^*(B'^+ \cap V)$ for some vector $V$, and we write it in logical form:

$$Q = \{(s,s') | \forall h : k \leq h < k' \implies v(h)\},$$

for some predicate $v$. From this representation, it is plain that $Q$ is transitive: if predicate $v$ holds for any $h$ between $k$ (inclusive) and $k'$ (exclusive) and for any $h$ between $k'$ (inclusive) and $k''$ (exclusive) then it holds for any $h$ between $k$ (inclusive) and $k''$ exclusive. Finally, to verify the concordance condition, we compute $T \cap B \cap B'^+B'$ and show it to be the empty relation:

$$T \cap B \cap B'^+B' \subseteq \{ \text{ by hypothesis } \}$$
\[ B' \cap B' + B' = \{ \text{substitutions} \} \]
\[ = \{(s, s')|k' = k + 1\} \cap \{(s, s')|k < k'\} \circ \{(s, s')|k = k + 1\} \]
\[ = \{ \text{performing the relational product} \} \]
\[ = \{(s, s')|k' = k + 1\} \cap \{(s, s')|k + 1 < k'\} \]
\[ = \{ \text{contradiction} \} \]
\[ = \phi. \]

\text{qed}

We illustrate this proposition on a simple example:

\[ \text{while} \ (i!=0) \ \{i=i-1; \ x=x+a[k]; \ k=k+1;\}. \]

Then, we write \( T \) and \( B \) as follows:

\[ T = \{(s, s')|i \neq 0\} \]
\[ B = \{(s, s')|\text{low} \leq k \leq \text{high} \land i' = i - 1 \land x' = x + a[k] \land k' = k + 1 \land a' = a\}. \]

This program meets the condition of Proposition 3, with \( B' = \{(s, s')|k' = k + 1\} \).

Application of this Proposition yields the following invariant relation:

\[ R = \{(s, s')|\forall h : k \leq h < k' \Rightarrow \text{low} \leq h \leq \text{high}\}. \]

In addition to this invariant relation, we also generate the elementary invariant relation \( R' \) provided by Proposition 2, and the following invariant relation, which links \( i \) (the loop counter) and \( k \) (the array index):

\[ R'' = \{(s, s')|i + k = i' + k'\}. \]

Applying Theorem 2 to the intersection of these three relations yields the following necessary condition of termination:

\[ (i = 0) \lor (i \geq 1 \land \text{low} \leq k \leq \text{high} - i + 1). \]

The reader may verify that this is indeed a necessary condition of abort-free termination; we believe it is sufficient as well (i.e. our combined invariant relation is small enough as to produce the necessary and sufficient condition of termination).

As another array example, we consider the following program on real variables \( x \) and \( y \), array variable \( a \) and \( b \) (of type real), index (integer) variables \( i \) and \( j \), and constant \( N \).

\[ \text{while} \ (i<N) \ \{x=x+a[i]; \ y=y+b[j]; \ j=j+i; \ i=i+1; \ j=j-i;\}. \]

By combining the invariant relation provided by Proposition 3 with invariant relations we maintain about array manipulation, we find the following condition of termination, which we believe to be sufficient:

\[ (i \geq N) \lor (i < N \land \text{low} \leq i \leq \text{high} \land \text{low} \leq j \leq \text{high} \land \text{low} \leq N \leq \text{high} \land \text{low} \leq (i+j-N) \leq \text{high}). \]

The first disjunct of this formula represents the case when the loop does not iterate at all (in which case it terminates readily); the second disjunct is long and complex but can in fact be interpreted easily. The first conjunct is the condition under which the loop iterates at least once; the four subsequent conjuncts impose the condition (\( \text{low} \leq ... \leq \text{high} \)) for the initial values and the final values of variables \( i \) and \( j \). Because \( i \) increases monotonically and \( j \) decreases monotonically through the execution of the loop, ensuring
that their initial value and final value are both within range is sufficient to ensure that all their intermediate values are also within range.

In this proposition we have assumed, for the sake of simplicity, that the loop has an index variable $i$ that is incremented at each iteration; but this is not necessary, as Theorem 3 gives us much broader latitude in choosing relation $B'$. Any relation that satisfies the conditions of anti-reflexivity, transitivity, and concordance is an adequate choice for our purposes; this includes not only a relation that increments (or decrements) an integer variable by a non-zero constant amount, but any relation that beats the tempo of the iteration (by depleting a data structure, popping a stack, progressing through a sequence of pointers, navigating a graph, etc).

5.2. Illegal Arithmetic Operation

Let $w$ be a while loop on space $S$ of the form $w$: while (t) {b}, and let $f$ be an arithmetic function that is evaluated in the loop body $b$; if function $f$ involves evaluating a square root, then the expression given in the argument must be non-negative; if it involves evaluating a fraction, then the expression given in the denominator must be non-zero; if it involves evaluating a logarithm, then the expression given in the argument must be positive; etc. We assume that execution of function $f(s)$ in state $s$ is prone to cause an abort, and we are interested to characterize the initial states on which the loop $w$ may execute without causing $f$ to abort. The following proposition is a corollary of Theorem 3.

**Proposition 4.** Let $w$ be a while loop on space $S$ of the form $w$: while (t) {b}, and let $f$ be an arithmetic function that is evaluated in $b$, and let $B'$ be a superset of $B$ that satisfies the conditions of anti-reflexivity, transitivity, and concordance. Then the following relation is an invariant relation for $w$:

$$R = \{(s, s') | \forall s'' : (s, s'') \in B' \land (s'', s') \in B'^+ \Rightarrow s'' \in \text{def}(f)\}.$$  

where $\text{def}(f)$ is the set of states for which function $f(s)$ is defined (can be evaluated).

**Proof.** This proposition is a corollary of Theorem 3, in which we let $\text{dom}(B)$ be defined as $\text{def}(f)$.

As an illustration of this Proposition, we consider the following loop on integer variables $i$, $x$, and $y$.

```plaintext
while (i!=0) {i=i-1; x=x+1; y=y-y/x;}
```

and we propose to apply Proposition 4 using the following relation as a superset of $B$:

$$B' = \{(s, s') | x' = x + 1\}.$$  

As a result of this choice, we find:

$$B'^+ = \{(s, s') | x < x'\}, B'^* = \{(s, s') | x \leq x'\}.$$  

We write the parameters of this loop ($T$ and $B$) as follows:

$$T = \{(s, s') | i \neq 0\}$$  

$$B = \{(s, s') | x + 1 \neq 0 \land i' = i - 1 \land x' = x + 1 \land y' = y - \frac{y}{x + 1}\}.$$
From this definition of $B$, we infer that $\text{dom}(B)$ is defined as follows:

$$\text{dom}(B) = \{ s | x + 1 \neq 0 \}.$$ 

By Proposition 4, we find the following invariant relation:

$$R = \{(s, s') \mid \forall s'' : x(s) \leq x(s'') < x(s') \Rightarrow x(s'') + 1 \neq 0 \},$$

which we can rewrite more simply (by the change of variable $x(s'') = h$) as:

$$R = \{(s, s') \mid \forall h : x \leq h < x' \Rightarrow h + 1 \neq 0 \}.$$ 

This invariant relation alone is insufficient to derive a meaningful termination condition, since the loop terminates by testing variable $i$ whereas this invariant relation refers to variable $x$; at a minimum, we need an additional invariant relation that links $i$ and $x$, and an invariant relation that records the variation of $i$ (or, equivalently, the variation of $x$). Finally, we also generate the elementary invariant relation, which captures asymptotic behavior of the loop. This yields the following aditional relations.

- The elementary invariant relation: $R_1 = I \cup T(T \cap B)$,
- The relation that will ensure that the number of iterations is finite: $R_2 = \{(s, s') | i \geq i' \}$,
- The relation that links the loop counter, on which termination depends, to $x$, on which the condition of abort-avoidance applies: $R_3 = \{(s, s') | x + i = x' + i' \}$.

We take the intersection of these four relations, and apply Theorem 2; this yields the following necessary condition of termination (which we believe to also be sufficient),

$$(i = 0) \lor (i \geq 1 \land (x < -i \lor x \geq 0)).$$

Indeed, in order for this loop to terminate after a finite number of iterations without attempting a division by zero, either ($i = 0$) (in which case the loop exits without iterating) or ($i > 0$) in which case either ($x \geq 0$) (then $x + 1$ is initially greater than zero, and increases away from zero at each iteration) or ($x < -i$), in which case $x$ starts negative but the loop exits before ($x + 1$) reaches 0.

As another example, we consider the following loop on the same space $S$ defined by integer variables $i$, $x$ and $y$:

```
while (x!=0) { i=i+2; x=x-5; y=y-y/x; }
```

we find the following termination condition, which we believe to be sufficient in addition to being provably necessary:

$$(x = 0).$$

Indeed, any value of $x$ other than a positive multiple of 5 leads to an unbounded number of iterations; any value of $x$ that is a positive multiple of 5 will iterate $\frac{x}{5}$ times, but cause a division by zero on the last iteration. Hence the only case when this loop terminates is the case when ($x = 0$), i.e. it does not iterate at all.

Using the same invariant relations as the examples above, we were able to derive termination conditions of variations of this loop (with the loop condition being ($i!=0$)), including the following configurations of indices:

$\{i' = i+1 \land x' = x+5\}, \{i' = i-2 \land x' = x+1\}, \{i' = i-2 \land x' = x+5\}, \{i' = i+a \land x' = x+b\}.$
5.3. Arithmetic Overflow

Because computer arithmetic is limited, one may apply an arithmetic operation to two representable arguments, and obtain a result that is not representable in the computer; this is another source of abort conditions. In this section, we consider the condition under which the execution of a loop proceeds without causing an arithmetic overflow; we assume that the abort is caused, not by applying the arithmetic operation (strictly speaking, the ALU makes the necessary provisions to represent the result of its arithmetic operations), but rather by attempting to store the (un-representable) result in memory; hence we assume that strictly speaking, the abort is caused by the assignment statement, not by the expression evaluation. In order to capture freedom from overflow, we consider all the assignment statements of the loop body, and for each statement of the form \( x=E \) where \( x \) is a variable of type \( T \) and \( E \) is an expression that returns a value of type \( T \), we include (in the definition of \( \text{dom}(B) \)) the predicate: \( E \) is a representable value of type \( T \), which we abbreviate as \( \text{rep}_T(E) \).

Because the same expression \( E \) may be evaluated in different states of the program, we denote by \( E(s) \) the value of expression \( E \) at state \( s \). We obtain the following proposition, which is a corollary of Theorem 3.

**Proposition 5.** Let \( w \) be a while loop on space \( S \) of the form \( w: \text{while} \ (t) \ {b} \), and let \( E \) be an arithmetic expression that is assigned to a variable of type \( T \) in the loop body \( b \). Let \( B' \) a superset of \( B \) that satisfies the conditions of anti-reflexivity, transitivity, and concordance. Then

\[
R = \{(s,s')|\forall s'': (s, s'') \in B'' \land (s'', s') \in B'^+ \Rightarrow \text{rep}_T(E(s''))\}.
\]

**Proof.** This proposition is a corollary of Theorem 3, where we let \( \text{dom}(B) \) be defined as \( \text{dom}(B) = \{s|\text{rep}_T(E(s))\} \). qed

As an illustration of this proposition, we consider the following loop on variables \( x, y \) and \( z \) of type integer:

\[
\text{while } (y \neq 0) \ {y = y-1; z = z+x;}
\]

The function of the body of this loop can be written as:

\[
B = \{(s, s')|\text{rep}_{\text{int}}(y - 1) \land \text{rep}_{\text{int}}(z + x) \land x' = x \land y' = y - 1 \land z' = z + x\}.
\]

We apply Proposition 5 to this loop, taking \( B' \) as

\[
B' = \{(s, s')|x' = x \land y' = y - 1 \land z' = z + x\}.
\]

To compute the transitive closure and the reflexive transitive closure of \( B' \), we refer to our database of recognizers, which provide the following relations:

\[
B'^+ = \{(s, s')|x' = x \land y' > y' \land z + xy = z' + x'y'\},
\]

\[
B'^* = \{(s, s')|x' = x \land y \geq y' \land z + xy = z' + x'y'\}.
\]

The condition of anti-reflexivity is trivially verified, since \( B'^+ \cap I \) is a subset of

\[
\{(s, s')|y > y' \land y = y'\},
\]

which is empty. Also, the condition of concordance is verified, since \( (T \cap B) \cap B'^+ B' \) is a subset of

\[
\{(s, s')|y > y' + 1 \land y' = y - 1\},
\]
which is also empty. As for the transitivity condition, its proof can be established in the same way as we did for Proposition 3. Hence we may apply Proposition 5, yielding the following invariant relation:

$$ R = \{(s, s') | \forall s'' : y \geq y'' > y' \land x = x'' \land z'' = x' \land z + xy = z'' + x''y' \land z'' + x''y'' = z' + x'y' \}

\Rightarrow repInt(z'' + x'') \land repInt(y'' - 1) \}.$$  

We further derive the following invariant relations:

- The elementary invariant relation, that captures loop behavior in border cases: $$ R_0 = I \cup T(T \cap B). $$
- The invariant relation that records the decrease of $$ y $$ (to bind the number of iterations): $$ R_1 = \{(s, s') | y \geq y' \}.$$
- The invariant relation that records the relation between the index variable $$ y $$, and the arithmetic expression whose overflow we want to model ($$ z + x $$): $$ R_2 = \{(s, s') | z + xy = z' + x'y' \}.$$

When we take the intersection of all these relations and apply Proposition 5, we find the following termination condition:

$$ (y = 0) \lor (y > 0 \land MinInt \leq z + xy \leq MaxInt) \}.$$  

In addition to being provably a necessary condition of termination, we believe that this logical formula is also a sufficient condition of termination: In order for this loop to terminate without causing an abort, $$ y $$ has to be zero, or it has to be positive, then $$ z + xy $$ (which is the expression that the loop computes into $$ z $$) has to be representable (i.e. included between MinInt and MaxInt).

As a second illustrative example, we consider the following loop on integer variables $$ x $$ and $$ y $$:

```
while (y!=N) { y = y+1; x = x+y; }
```

The function of the loop body can be written as:

$$ B = \{(s, s') | repInt(y + 1) \land repInt(x + y + 1) \land y' = y + 1 \land x' = x + y + 1 \} , $$

whence the domain of $$ B $$ can be written as:

$$ \text{dom}(B) = \{s | repInt(y + 1) \land repInt(x + y + 1) \}.$$  

For $$ B' $$, we choose the following relation:

$$ B' = \{(s, s') | y' = y + 1 \land x' = x + y + 1 \}.$$  

We proceed the same way as the previous example, and we find:

$$ B'^+ = \{(s, s') | y < y' \land 2x - y(y + 1) = 2x' - y'(y' + 1) \}, $$

$$ B'^* = \{(s, s') | y \leq y' \land 2x - y(y + 1) = 2x' - y'(y' + 1) \}.$$  

Using the same argument as in the previous example, we can establish that $$ B' $$ satisfies the conditions of anti-reflexivity, transitivity, and concordance. Hence, by Proposition 5, the following relation is an invariant relation for $$ w $$:

$$ R = \{(s, s') | \forall s'' : y \leq y'' < y' \land 2x - y(y + 1) = 2x'' - y''(y'' + 1) \}

\wedge 2x'' - y''(y'' + 1) = 2x' - y'(y' + 1) \Rightarrow repInt(y + 1) \land repInt(x + y + 1) \}.$$  

This relation by itself is not adequate; we add to it the following invariant relations:

- The elementary invariant relation, $$ R_0 = I \cup T(T \cap B) $$.
The invariant relation that records the increase of \( y \):
\[
R_1 = \{(s, s')|y \leq y'\}.
\]

The invariant relation that links program variables to each other:
\[
R_2 = \{(s, s')|2x - y(y + 1) = 2x' - y'(y' + 1)\}.
\]

If we take the intersection of these invariant relations and apply Proposition 5, we find the following necessary of termination:
\[
(y = N) \lor (y < N \land \text{MinInt} \leq N \leq \text{MaxInt} \land \text{MinInt} \leq x - \frac{y(y + 1)}{2} + \frac{N(N + 1)}{2} \leq \text{MaxInt}).
\]

We believe that this condition is sufficient, in addition to being provably necessary. This condition provides that the loop terminates without causing an abort if and only if \((y = N)\) (in which case the loop terminates instantly) or \((i < N)\), in which case the number of iterations is finite, but then we also have conditions that ensure that the loop causes no arithmetic overflow of (respectively) the two assignment statements of the loop body. Note that the two conjuncts that follow \((y < N)\) provide (respectively) that the final value of \(y\) and the final value of \(x\) are both representable (i.e. included between \(\text{MinInt}\) and \(\text{MaxInt}\)). Because the initial values of these variables are representable by hypothesis, and because both variables evolve monotonically, ensuring that the initial and final values of these variables are representable is sufficient to ensure that all intermediate values are as well; which ensures that execution of this loop causes no arithmetic overflow.

5.4. Illegal Pointer Reference

The investigation of illegal pointer references in the general case, under general assumptions about the structure of the data, the heap management policies, data sharing, aliasing, etc is very complex, and is beyond the scope of this paper. The only goal of this section is to show that the generic model introduced in Theorem 3 applies to pointer-caused aborts as well as it applies to other abort conditions.

We consider a while loop \(w\) on space \(S\) and we assume that space \(S\) includes a pointer variable \(p\). We assume that pointer \(p\) refers to a record structure that has several pointer fields that point to the same structure. Whenever a pointer is referenced, we must ensure that it is not nil, to avoid an abort. The following proposition is a corollary of Theorem 3, and applies to loops that are prone to cause an illegal pointer reference.

**Proposition 6.** Let \(w\) be a while loop on space \(S\) of the form \(w: \text{while } (t) \{b\}\), and let \(p\) be a pointer variable that is referenced in \(b\) in a statement of the form \(p=*p.f\) for some field \(f\). We assume that pointer \(p\) points to a record type \(P\), which contains one or more fields that point to records of type \(P\). If the data structure (graph) so defined does not have loops (\(p\) points to itself) nor cycles (\(P\) is reachable from \(p\)) then then the following relation is an invariant relation for \(w\):
\[
R = \{(s, s')|\forall p'' : reach(p, p'') \land reach(p'', p') \land p'' \neq p' \Rightarrow p'' \neq \text{nil}\},
\]
where \(reach(p, p')\) means that pointer \(p'\) can be reached from pointer \(p\) by an arbitrary number (possibly zero) of pointer references.

**Proof.** Given that \(b\) has a statement of the form \(p=*p.f\), where \(*p\) designates the record pointed to by \(p\) and \(*p.f\) designates the pointer addressed by field \(f\), we let \(B'\) be the relation defined by:
\[
B' = \{(s, s')|p' = *p.f\}.
\]
The anti-reflexivity and concordance of $B'$ can be established by virtue of the absence of cycles and the absence of loops in the data structure (by hypothesis). The condition of transitivity can be established in the same way as the proof of Proposition 3. Hence we may apply Theorem 3, letting $dom(B)$ be defined as $\{s|p \neq \text{nil}\}$. To this effect, we compute $B'^{+}$ and $B'^{*}$, which we find to be as follows:

$$B'^{+} = \{(s, s')|\text{reach}(p, p') \land p \neq p'\},$$

$$B'^{*} = \{(s, s')|\text{reach}(p, p')\}.$$

Combining $dom(B)$ with $B'^{+}$ and $B'^{*}$ as indicated by Theorem 3, we find:

$$R = B'^{*}(B'^{+} \cap BL).$$

This is exactly the invariant relation proposed by the proposition. \hspace{1cm} \text{qed}

As we have seen repeatedly in previous sections, the invariant relation provided by Theorem 3 are typically insufficient to compute a meaningful termination condition, and must be supplemented with other (functional) invariant relations, that capture relevant functional properties of the loop at hand. The same applies for the invariant relation proposed in Proposition 6. To supplement this proposition, we introduce relevant invariant relations that pertain to data structures defined by pointers; to this effect, we introduce some functions. As we recall, we assume that the data structure has no cycle; we let $\text{Roots}$ be the set of nodes that have no pointer pointing to them, and $\text{Leaves}$ be the set of nodes that are pointing to no other nodes (all their pointer fields are nil). We introduce a fictitious node that has links to all the roots, which we denote by $\text{Root}$, and a fictitious node to which all leaves, which we denote by $\text{Leaf}$.

- Given a node represented by its pointer $p$, we let $\text{MaxDepth}(p)$ be the length of the longest path from the Root to $p$, and $\text{MinDepth}(p)$ be the length of the shortest path from the Root to $p$.
- Given a node represented by its pointer $p$, we let $\text{maxHeight}(p)$ be the length of a longest path from $p$ to the Leaf, and we let $\text{minHeight}$ be the length of a shortest path from $p$ to the Leaf.

These functions enable us to derive some general invariant relations, which we present in the following Proposition.

**Proposition 7.** Let $w$ be a while loop on space $S$ of the form $w: \text{while } (t) \{b\}$, and let $p$ be a pointer variable that is referenced in $b$. We assume that the record that $p$ points to has several pointer fields, say $f_1, f_2, .. f_n$. If the function of $b$ is a subset of $B' = \{(s, s')|p' = *p.f_i\}$ for some pointer field $f_i$ of $p$ then the following relations are invariant relations for $w$:

$$R_0 = \{(s, s')|\text{maxDepth}(p) \leq \text{maxDepth}(p')\}.$$

$$R_1 = \{(s, s')|\text{minDepth}(p) \leq \text{minDepth}(p')\}.$$

$$R_2 = \{(s, s')|\text{maxHeight}(p) \geq \text{maxHeight}(p')\}.$$

$$R_3 = \{(s, s')|\text{minHeight}(p) \geq \text{minHeight}(p')\}.$$

22
Proof. Reflexivity and transitivity stem readily from the structure of the relations; invariance can be proved readily by considering that the inequalities that characterize each relation are logical conclusions of the formula: $p' = *p.f$ for any pointer field $f$. qed

In tree-like structures, where there is a single path from the root to every node, functions minDepth and maxDepth are identical, and are denoted by depth, affording us smaller invariant relations, as shown below.

**Proposition 8.** Let $w$ be a while loop on space $S$ of the form $w$: while $(t) \{ b \}$, and let $p$ be a pointer variable that is referenced in $b$. We assume that the record that $p$ points to has several pointer fields, say $f_1, f_2, \ldots f_n$, and that the resulting data structure is tree-like. If the function of $b$ is a subset of $B' = \{(s, s')|p' = *p.f_i\}$ for some pointer field $f_i$ of $p$ then the following relations are invariant relations for $w$:

$$R_0 = \{(s, s')|\text{depth}(p) \leq \text{depth}(p')\}.$$

$$R_1 = \{(s, s')|\text{depth}(p) + \text{maxHeight}(p) \geq \text{depth}(p') + \text{maxHeight}(p')\}.$$

$$R_2 = \{(s, s')|\text{depth}(p) + \text{minHeight}(p) \leq \text{depth}(p') + \text{minHeight}(p')\}.$$

$$R_3 = \{(s, s')|\forall h : \text{depth}(p) \leq h < \text{depth}(p') \Rightarrow (*p.f^h \neq \text{nil})\}.$$

**Proof.** Relation $R_0$ is reflexive and transitive; it is a superset of $B'$ (hence a superset of $B$) because the unique path from the root to $p'$ necessarily goes through $p$. Relation $R_1$ is reflexive and transitive. As for being a superset of $B$, it suffices to prove that it is a superset of $B'$. Let $(s, s')$ be a pair of $B'$. Then, by definition, $\text{depth}(p') = \text{depth}(p) + 1$. Now, if $p'$ is on the path from $p$ to the farthest leaf, then $\text{maxHeight}(p) = 1 + \text{maxHeight}(p')$. Whence $\text{depth}(p) + \text{maxHeight}(p) = \text{depth}(p') + \text{maxHeight}(p')$. If $p'$ is not on the path from $p$ to the farthest leaf, then $\text{maxdepth}(p) > 1 + \text{maxHeight}(p')$. Whence $\text{depth}(p) + \text{maxHeight}(p) > \text{depth}(p') + \text{maxHeight}(p')$. The same argument can be used (with some duality) for relation $R_2$. The proof that relation $R_3$ is invariant is similar to the proof of Proposition 3. qed

A trivial corollary of this Proposition is that if $p$ has a single pointer field, then there is a single path from any node to a leaf, hence $\text{maxHeight}()$ is the same as $\text{minHeight}()$; we refer to this function as $\text{height}()$, and we have the following Proposition.

**Proposition 9.** Let $w$ be a while loop on space $S$ of the form $w$: while $(t) \{ b \}$, and let $p$ be a pointer variable that is referenced in $b$. We assume that the record that $p$ points to has a single pointer field (say, $f$) and that it defines a structure without cycles. If the function $(T \cap B)$ is a subset of $B' = \{(s, s')|p' = *p.f\}$ then the following relation is an invariant relation for $w$:

$$R = \{(s, s')|\text{depth}(p) + \text{height}(p) = \text{depth}(p') + \text{height}(p')\}.$$

If an integer variable is incremented or decremented alongside a pointer reference in a rooted tree structure, then we can link the depth of a node to the integer variable, as shown in the following Proposition.
Proposition 10. Let \( w \) be a while loop on space \( S \) of the form \( \text{while } (t) \{ b \} \), and let \( p \) be a pointer variable in \( S \) and \( i \) be an integer variable in \( S \). If the function \((T \cap B)\) is a subset of \( B' = \{(s, s') | i' = i + c \land p' = *p.f \} \) for some non-zero constant \( c \), then the following relation is an invariant relation for \( w \):

\[
R = \{(s, s') | i - c \times \text{depth}(p) = i' - c \times \text{depth}(p') \}.
\]

Proof. This relation is reflexive and transitive, as it is the nucleus of a function. That it is a superset of \( B' \) can be readily established by considering that if \( p' = *p.f \) then \( \text{depth}(p') = \text{depth}(p) + 1 \).

We consider a number of examples to illustrate the results of this subsection. If we consider the following program on pointer \( p \), where the record of \( p \) has a single pointer field \( \text{next} \),

\[
\text{while } (p \neq \text{nil}) \{ \text{\texttt{p=*p.next;}} \},
\]

then we find the termination condition true.

If we consider the following program on integer variable \( i \) and pointer variable \( p \), where the record of \( p \) has a single pointer field \( \text{next} \),

\[
\text{while } (i < N) \{ \text{\texttt{p=*p.next; i=i+1;}} \},
\]

then we find the following termination condition

\[
(i \geq N) \lor (i < N \land \text{depth}(p) \geq N - i).
\]

If we consider the following program on integer variable \( i \) and pointer variable \( p \), where the record of \( p \) has two pointer fields \( \text{left} \) and \( \text{right} \),

\[
\text{while } (i < N) \{ \text{\texttt{i=i+1; if ((i% 2)==0)\{p=*p.right;\} else \{p=*p.left;\}}},
\]

then we find the following termination condition

\[
(i \geq N) \lor (i < N \land N - i \leq \text{maxHeight}(p)).
\]

Note that this is a necessary condition of termination, but not a sufficient condition of termination; we conjecture that a sufficient condition of termination would have \( \text{minHeight()} \) rather than \( \text{maxHeight()} \).

6. Condition of Sufficiency

Throughout this paper we have considered several examples of programs for which we have given a necessary condition of termination, and claimed that we thought the condition was sufficient, in addition to being provably necessary. In this section, we discuss two questions, namely: why can’t we derive a provably sufficient condition of termination? How can we claim that our necessary conditions are sufficient? We address these questions in turn, below.

- **Why can’t we derive a sufficient condition?** It is hardly surprising that arbitrary (arbitrarily large) invariant relations can only generate necessary conditions, since they capture arbitrarily partial information about the loop, hence cannot be used to make claims about a global property of the loop. Yet strictly speaking, we can formulate a sufficient condition of termination, but it is of little use in practice. A sufficient condition of termination would read as follows: Given a while loop of the form \( w: \text{while } (t) \{ b \} \), and given the invariant relation \( R = (T \cap B)^* \), then \( R \subseteq WL \).

As we recall from Proposition 1, \( R = (T \cap B)^* \) is an invariant relation of the loop, and is in fact the smallest invariant relation of the loop. In practice, it is very difficult to compute this reflexive transitive closure for arbitrary \( T \) and \( B \). One of the
main interests of invariant relations is in fact that: First they enable us to compute or approximate the reflexive transitive closure of \((T \cap B)\). Second and perhaps most importantly, they enable us to dispense with the need to compute the reflexive transitive closure of \((T \cap B)\); in particular, one of the main motivations for using invariant relations is that they enable us, with relatively little scrutiny of the loop, to answer many questions pertaining to the loops; hence requiring that we compute the strongest possible invariant relation to secure a sufficient condition of termination defeats the purpose of using invariant relations.

- How can we claim sufficiency? We are currently developing heuristics that enable us to recognize when an invariant relation is small enough to ensure that the formula of Theorem 2 provides a sufficient condition of termination. As far as ensuring that the number of iterations is finite, we can proceed by identifying the variables that intervene in the loop condition, and generating all the invariant relations that involve these variables, and any variable that affects their value (through assignment statements). As for ensuring freedom from aborts, we also want to include any invariant relation that links the variables identified above with the variables that are involved in the abort condition (array indices, denominators of fractions, arithmetic expressions, etc).

Another heuristic that we are considering is to define a set of recognizers that specialize in computing a sufficient condition of termination, by focusing on termination-related details; for example, if the loop body includes a clause of the form \(x' = x + a[i]\) for some real variable \(x\), real array \(a\), and index (integer) variable \(i\), then the complete recognizer would generate the invariant relation \(\{(s, s')|x + \Sigma a = x' + \Sigma a'\}\) whereas the termination-related recognizer would merely record that array \(a\) has been accessed at index \(i\). A final heuristic, invoked in [37] for the purpose of minimizing the number of invariant relations generated by our tool, involves generating just enough invariant relations to link all the statements of the loop body into a connected graph.

All the heuristics discussed herein are intended to enable us to claim sufficiency of our termination condition without having to generate all the invariant relations of the loop; we envision to organize these heuristics into a cohesive algorithm, as part of our future research plans.

7. Conclusion

7.1. Summary

7.1.1. Conceptual Results

In this paper, we have introduced an approach to computing the termination condition of programs, which can be characterized by the following premises:

- Our definition of termination refers to the property that the number of steps in the program's execution is finite, and that each individual step can be completed without causing an illegal operation (to which we refer as an abort).

- Focusing on while loops, we present a theorem (Theorem 2) that transforms each invariant relation into a necessary condition of termination. Because the theorem makes no reference to what may preclude normal termination, it can be used to model any possible cause that may preclude normal termination.

- We find that when we apply Theorem 2 with invariant relations that are antisymmetric (in addition to being reflexive and transitive), we generate bounds on the number of iterations, i.e. conditions to the effect that the number of iterations is finite.
We present a theorem (Theorem 3) that provides a general format for invariant relations that capture the property of abort freedom; when Theorem 2 is applied using invariant relations generated by Theorem 3 combined with invariant relations that capture other relevant functional information, they produce a necessary condition of termination that encompasses bounds on the number of iterations as well as conditions that ensure that no single iteration causes an abort at run-time. Put together, Theorems 2 and 3 provide a necessary condition of termination which says in effect that if a while loop terminates on an initial state \( s \) without causing an abort then all its intermediate states are in \( \text{dom}(B) \), where \( B \) is the function of the loop body.

We have generated a number of corollaries for Theorem 3, which apply its general formula to special abort conditions, generating appropriate termination conditions for each type of abort. The adequacy of the results that we have obtained from the corollaries gives us further confidence in the original theorem.

We argue that while invariant relations enable us to compute provably necessary conditions of termination, we can use them to also attain sufficient conditions of termination, provided we generate a sufficient number of them. Also, we are exploring heuristics that enable us to recognize when we have collected enough invariant relations to be able to claim sufficiency.

7.1.2. Automated Support

The transformation of source code (C++ for now) into our internal relational notation is a simple compiler transformation. The step that generates invariant relations from the relational representation of the loop, by matching clauses of the relational representation against recognizers is currently operational; and our current database includes 89 recognizers, covering a number of data structures (scalar data types, structured data types such as arrays, abstract data types such as lists, etc). Whereas this component currently operates by syntactic matching, we are replacing it with a semantic matching algorithm. Semantic matching determines whether a formal pattern matches an actual pattern by instantiating the formal pattern with actual variable names and checking the validity of the theorem that results from the equality of the two patterns. Semantic matching offers two significant advantages over syntactic matching: first it enables us to match patterns across a wide range of variance in form; second, it enables us to achieve much broader scope with fewer recognizers. This transformation is currently under way. The third step, of transforming invariant relations into necessary conditions of termination, is a trivial step, since it involves submitting a precoded formula, in which a placeholder is replaced by the current invariant relation, to an algebraic system, to have the formula simplified. Our plan calls for completing the semantic matching step, then integrating the three steps within a single tool, with a user interface to manage interactions with the user.

7.2. Related Work

7.2.1. Loop Termination

Analysis of termination is a very active research area for which there is a vast bibliography. Boyer and Moore [4] propose a technique based on semi-automatic theorem proving where termination arguments have to be user-supplied. The work of Gupta et al. [30] uses templates to identify recurrent sets, but for the sole purpose of characterizing infinite loops; also focused on non termination is the work of Velroyen and Ruemmer [69]. In these two cases, the analysis is restricted to linear programs. Linear programs are
also the focus of other researchers, such as [21, 5, 67, 11]. In [7], Burnim et al. propose a
dynamic approach to detecting infinite loops, based on concolic executions (a combination
of concrete execution and symbolic analysis); the technique is generally incomplete,
in the sense that the iterative analysis may lack the resources needed to solve complex
constraints. In [23] Falke et al. critique existing approaches to the analysis of termination
of iterative program, on the grounds that treating bitvectors and bitvector arithmetic
as integers and integer arithmetic is unsound and incomplete; also, they propose a novel
method for modeling the wrap-around behavior of bitvector arithmetic, and analyze loop
termination within this model.

In [61], Podelski and Rybalchenko propose a complete method for computing linear
ranking functions; their approach is complete in the sense that if the loop can be bound
by a linear ranking function, one such a function will be found by their method; Lee et.
al. [47] use the results of Podelski and Rybalchenko [61, 14] and propose an approach
based on algorithmic learning of Boolean formula in order to compute disjunctive, well
founded, transition invariants; the technique appears to be particularly effective when
dealing with simple programs dealing with linear arithmetic. In [15], Cook et al. give
a comprehensive survey of loop termination, in which they discuss transition invariants;
whereas invariant relations are approximations of $(T \cap B)^*$, transition invariants are in
fact approximations of $(T \cap B)^+$. This slight difference of form has a significant impact on
the properties and uses of these distinct concepts. Whereas transition invariants are used
by Cook et al. to characterize the well-founded property of $(T \cap B)^+$, we use invariant
relations to approximate the function of a loop, and its domain.

In [10], Chawdhary et al. use abstract interpretation to synthesize ranking functions;
their technique is subsequently improved by Tsitovitch et al. [68], where loop summaries
allow them to increase the scalability of the technique. In [12], Cook et al. propose to
underapproximate weakest liberal preconditions in order to synthesize simpler predicates
that still enable them to prove termination in cases where other tools would return a
spurious warning of possible non-termination. In [69], Velroyen and Ruemmer propose to
synthesize invariants from a set of recorded invariant templates, and deploy a theorem
prover to prove that the final states characterized by the invariants is unreachable, hence
proving termination; because it provides a necessary condition of termination, our
work can be used to disprove termination: whenever the necessary condition is violated,
the loop does not terminate.

Abstract interpretation [16, 18, 17] is a broad scoped technique that aims to infer
properties of programs by successive approximations of their execution traces; as such,
it bears some resemblance to our invariant relations-based approach (which infer properties
of while loops by approximations of the transitive closure $(T \cap B)^*$). Also, abstract
interpretation has been used to, among others, analyze the properties of abort freedom
of arbitrary programs [38]. The work on abstract interpretation has given rise to a widely
used automated tool that analyzes programs and issues reports pertaining to their cor-
rectness, termination, abort-freedom, etc [19, 3]. In [1], Ancourt et al. analyze loops by
some form of abstract interpretation, but they dispense with the fixpoint semantics of
loops by attempting to approximate the transitive closure of the loop body abstraction.
While the calculation of transitive closures is complex in general, the authors attempt
it using affine approximations of the loop body transformations, which they define in
terms of affine equalities and inequalities of state variables. Using techniques of discrete
differentiation and integration, they derive an algorithm that computes affine invariant
assertions from this analysis, and use the generated assertions to monitor abort-freedom conditions on the state of the program. They illustrate their algorithm by running it on many published sample loops.

Overall, it is fair to say, perhaps, that all the work on ensuring termination by means of ranking functions and well founded orderings is an attempt to approximate (i.e. find a superset of) the transitive closure of the loop body, i.e. \((T \cap B)^+\). By contrast, our work attempts to compute or approximate the domain of the function of the loop, hence takes a broader interpretation of the concept of termination; to do so, we approximate the reflexive transitive closure of the loop body, i.e. \((T \cap B)^*\).

7.2.2. Pointer Semantics

Heap data structures manipulate potentially unbounded data structures, which do not lend themselves to simple modeling; as such, they represent one of the biggest challenges to scalable and precise software verification. In order to model the property that a loop causes no illegal pointer reference, we have to capture some aspects of pointer semantics; in our work, we use invariant relations to represent unbounded pointer references, and to reason about them. In this section, we review some of the alternative approaches to pointer semantics, and compare them to ours; we have been able to classify it into five broad categories, which we review in turn below.

- **Shape Analysis.** These approaches proceed by identifying some structure into the pattern of pointers between nodes. In [63] Sagiv et. al. use three-valued logic as a foundation for a parameterized framework for carrying out shape analysis; the framework is instantiated by supplying predicates that capture different relationships between nodes, and by supplying the functions that specify how the predicates are updated by particular assignments. In [28], Bhargav et. al. propose a new shape analysis algorithm, which is presented as an inference system for computing Hoare triplets summarizing heap manipulation programs. These inference rules are used as a basis for a bottom-up shape analysis of data structures.

- **Path-Length Analysis.** In [65], Spoto et al. prove the termination of programs written in Java Bytecode by mapping them into a constraint logic program which is built on the basis of a path-length analysis of the original program. The proof is based on the proposition that the termination of the logic constraint program is a sufficient condition to the termination of the original program. The path-length analysis of a Java bytecode program derives an upper bound of the maximal length of a path of pointers that can be followed from each variable of the program; the concepts of maxDepth and maxHeight presented in this paper bear some resemblance to Spoto et al.’s path-length function, and the overapproximations derived for the path-length function bears some resemblance to the type of approximations that are produced by invariant relations. But while Spoto et al. are interested in proving program termination, we are interested in computing termination conditions; while Spoto et al. are interested in termination as the property that the program executes a finite number of steps, we are interested to model termination as well as abort freedom; while Spoto et al.’s approach is focused primarily on the data structure of the program, our approach is focused primarily on its control structure.

- **Alias Analysis.** This approach focuses on determining whether two pointers refer to the same heap cell [57]. In [31], Hackett and Aiken use a combination of predicate abstraction, bounded model checking, and procedure summarization to compute a precise
path-sensitive and context-sensitive pointer analysis. Alias analysis is only useful for reasoning about explicitly named heap cells, and cannot model general unbounded data structures.

- **Separation Logic.** This approach makes it possible to reason about heap manipulation programs [62] by extending Hoare logic [32] with two operators, namely separation conjunction and separation implication; these operators are used to formulate assertions over disjoint parts of the heap. In [59], O’hearn et. al. define a logic for reasoning about programs that alter data structures; to this effect they define a low-level storage model based on a heap with associated access operations, along with axiomatizations for these operations. The resulting model supports local reasoning, whereby only those cells that a program accesses are referenced in specifications and proofs.

- **Reachability Predicates.** This approach defines and uses predicates that characterize reachable nodes in an arbitrary data structure [58]. Indexed predicate abstraction [43] and Boolean heaps [60] generalize the predicate abstraction domain so that it enables the inference of universally quantified invariants. In [29], Gulwani et. al. show how to combine different abstract domains to obtain universally quantified domains that capture properties of linked lists. Craig interpolation has also been used to find universally quantified invariants for linked lists [49]. In [50], Mehta and Nipkow model heaps as mappings from addresses to values, and pointer structures are mapped to higher level data types for the verification of inductively defined data types like lists and trees. In [24], Filliatre and Marche introduce a method for proving that a program satisfies its specification and is free of null pointer referencing and out-of-bounds array access. Their approach is based on Burstall’s model for structures extended to arrays and pointers. Similar tools have been developed for C-like languages, including Astree [3], Caveat [66], and SDV [13], but they are bounded to specific provers. In [51, 52], Meyer presents a comprehensive theory for modeling pointer-rich object structures and proving their properties; the model proposed by Meyer comes in two versions, a coarse-grained version that supports the analysis of the overall properties of the object structures, and a fine-grained version, that analyzes object structures at the level of individual fields. Meyer’s approach is represented in Eiffel syntax, and uses simple discrete mathematics.

Our interest in pointer semantics is much more recent than all these authors, and is driven by (and limited to) our interest in capturing conditions of abort avoidance as they pertain to illegal pointer references. Whereas we had thought initially that we could produce invariant relations that represent the scope equation of pointer references in loops for arbitrary data structures, we have subsequently resolved to generate invariant relations for well known data structures instead, for several reasons: First, generating invariant relations for the general case is very difficult; second, many authors whose work we have reviewed above appear to focus on well-known data structures rather than to arbitrary pointer-based structures; third, existing algorithms of shape analysis give us confidence that we can proceed by first analyzing the shape of our data, then deploying specialized invariant relations according to the shape that has been identified.

### 7.3. Assessment

To the best of our knowledge, our work is the only approach to computing termination conditions that interprets termination in the general sense of: ending in a well-defined final state. We say that a program terminates for an initial state $s$ if and only if the
program can produce a final state \( s' \) as an image of \( s \) by the program function. Whether the program fails to produce a final state because it fails to terminate or because it fails to apply an intermediate function in its finite execution sequence does not matter to us. In keeping with this premise, our definition of termination applies to iterative programs as much as it applies to non-iterative programs; also, as far as while loops are concerned, our approach provides a way to map any given invariant relation of the loop onto a necessary condition of termination. We can generate many invariant relations for the loop, each capturing a specific aspect of termination, and obtain a termination condition that ensures freedom from all causes of non-termination; to the best of our knowledge, our approach is unique in this feature.

Traditionally, the analysis of loop termination is studied separately from the analysis of its functional properties, with the latter relying on invariant assertions and the former relying on variant functions. By contrast, we use the same concept, namely invariant relations, to characterize the termination conditions and the functional properties of loops. From a conceptual viewpoint, we find it appealing to use the same approach/means to analyze the function of the loop and the termination condition of the loop, as the domain of a function is an integral part of the function, rather than an orthogonal attribute.

References


