Verifying While Loops with Invariant Relations*

Asma Louhichi, Wided Ghardallou and Khaled Bsaies
Faculty of Sciences, University of Tunis El Manar, 1080 Tunisia
wided.ghardallou@gmail.com, louhichiasma@yahoo.fr, Khaled.Bsaies@fst.rnu.tn
Lamia Labed Jilani and Olfa Mraihi
Institut Superieur de Gestion, Bardo, 2000 Tunisia
lamia.labed@planet.tn, olfa.mraihi@yahoo.fr
Ali Mili
New Jersey Institute of Technology University Heights
Newark NJ 07102-1982, USA mili@cis.njit.edu

March 12, 2013

Abstract

Traditionally, invariant assertions are used to verify the partial correctness of while loops with respect to pre/post specifications. In this paper we discuss a related but distinct concept, namely invariant relations, and show how invariant relations are a more potent tool in the analysis of while loops: whereas invariant assertions can only be used to prove partial correctness, invariant relations can be used to prove total correctness; also, whereas invariant assertions can only be used to prove correctness, invariant relations can be used to prove correctness and can also be used to prove incorrectness; finally, where traditional studies of loop termination equate termination with iterating a finit number of times, we broaden the definition of termination to also capture the condition that each individual iteration proceeds without raising an exception.

Keywords

while loops, invariant assertions, invariant relations, invariant functions, sufficient conditions of correctness, necessary conditions of correctness.

1 Introduction: Conclusive Proofs of Correctness

Ever since their introduction by Hoare in 1969 [19], invariant assertions have played a central role in the analysis of while loop, most notably in the verification of partial correctness with respect to pre/post specifications. Variant functions were subsequently introduced to characterize termination, thereby complementing the analysis afforded by invariant assertions. In this paper we explore the use of an alternative concept, invariant relations (due to [32]) to verify the total correctness of while loops with respect to relational specifications.

*Acknowledgement: This publication was made possible by a grant from the Qatar National Research Fund NPRP04-1109-1-174. Its contents are solely the responsability of the authors and do not necessarily represent the official views of the QNRF.

Invariant relations offer a number of theoretical and practical advantages with respect to invariant assertions, most notably:

- **Total Correctness vs Partial Correctness**. Whereas invariant assertions are used to prove partial correctness and variant functions are used to prove termination, invariant relations can be used to prove partial correctness and termination; in relational terms, the analysis of termination amounts to characterizing the domain of the loop function, whereas the analysis of partial correctness amounts to characterizing the input/ output mapping defined by the loop.

- **Broader Definition of Termination**. Whereas the analysis of termination using variant functions equates loop termination with having a finite number of iterations, our analysis of termination includes not only the condition that the number of iterations is finite, but also the condition that each individual iteration terminates normally, without causing an exception (such as division by zero, array reference out of bounds, etc).

- **Proving Correctness or Incorrectness**. Whereas invariant assertions may be used to prove the (partial) correctness of a loop but may not be used to prove its incorrectness, invariant relations may be used to prove either. Indeed, when one attempts to prove the correctness of a while loop using an invariant assertion, and the proof fails, one cannot determine whether the proof failed because the loop is incorrect or because the invariant assertion is inadequate. By contrast, invariant relations enable us to positively rule on correctness (if they subsume the candidate specification) or on incorrectness (if they are incompatible with the candidate specification).

Furthermore, the distinction between invariant assertions and invariant relations can be characterized by the following premises:

- **Context Freedom**. Whereas invariant assertions depend on the loop and also on its context (pre/post conditions), invariant relations are intrinsic to the loop, hence remain unchanged regardless of where the loop is used.

- **Subsumption**. Whereas any invariant assertion can be mapped onto an invariant relation and any invariant relation can be mapped to an invariant assertion, invariant relations are more general than invariant assertions in the following sense: all invariant assertions stem from invariant relations, but the converse is not true (only a small class of invariant relations stem from invariant assertions).

Invariant relations were introduced in [32]; in [33] we analyze the relationships between invariant assertions, invariant functions and invariant relations; in [15] we discuss how invariant relations can be used to answer a wide range of questions about a loop (computing its invariant assertions, its weakest preconditions, its strongest postconditions, its termination condition, its function, etc), in [23] we discuss scaling up of our invariant relation-based analysis, and in [24] we compare the performance of our anallysis tool against other tools (that generate invariant assertions or termination conditions).

In this paper we use the background presented herewith to achieve two modest goals:

- Show how invariant relations can be used to capture loop termination in a way that reflects not only the condition that the number of iterations is finite, but also the condition that each individual iteration executes without raising an exception.

- Show how invariant relations can be used to determine whether a loop is correct, whether it is incorrect, or whether the invariant relations that we have derived for the loop give insufficient information (relative to the specification) to make a decision.

This paper is an extension of [30], in which we had shown how invariant relations can be used to support a wide range of queries about the functional attributes of the loop, most notably: they can be used to compute or approximate the function of the loop; to compute the weakest precondition of the loop, for a given postcondition; to compute or approximate the strongest postcondition of the loop, for a given precondition; to compute an invariant assertion of the loop, for a given precondition; to compute or approximate the termination condition of the loop; to generate a necessary condition of correctness of the loop; and to generate a sufficient condition of correctness of the loop. In this paper we consider the last three uses of invariant relations, and we combine them to derive an integrated algorithm that takes a loop and a candidate specification, and determines, by generating a succession of invariant relations, whether the loop is correct with respect to the specification. Our algorithm is designed in such a way that we can rule on correctness (if the invariant relations subsume the candidate specification) or incorrectness (if the invariant relations are incompatible with the candidate specification) without necessarily analyzing all the loop's functional details.

In section 2 we briefly introduce some relational mathematics, which we use in section 3 to introduce invariant relations as well as a relational form of invariant assertions. In section 4 we discuss how invariant relations can be used to compute the termination condition of a while loop, and in section 5 we discuss propositions that give (respectively) a necessary condition of correctness, and a sufficient condition of correctness of a loop with respect to a relational specification. In section 6 we use the results of the previous three sections to derive an algorithm that analyzes while loops to determine their correctness (or incorrectness) with respect to a given relational specification. We conclude in section 8 with a discussion of our main findings, our research prospects, and related work.

2 Relational Mathematics

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 $s = \langle x, y \rangle$. 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 L, the *identity* relation, denoted by I, and the *empty* relation, denoted by $\emptyset$.

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 $\widehat{R}$, and defined by $\widehat{R} = \{(s, s') | (s', s) \in R\}$. The *product* of relations R and R' is the relation denoted by $R \circ R'$ (or RR') and defined by $R \circ R' = \{(s, s') | \exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\}$. The *pre-restriction* (resp. *post-restriction*) of relation R to predicate t is the relation $\{(s, s') | t(s) \wedge (s, s') \in R\}$

(resp. $\{(s,s')|(s,s') \in R \wedge t(s')\}$). 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 $dom(R) = \{s|\exists s' : (s,s') \in R\}$, and the *range* of R ($rng(R)$) is the domain of $\widehat{R}$. We note that for a relation 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 $\widehat{R}R \subseteq I$, and we say that R is *total* if and only if $I \subseteq R\widehat{R}$, 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 subset C of S; we use vectors as a relational representation of sets. 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 $R = \widehat{R}$. A relation that is reflexive, symmetric and transitive is called an *equivalence relation*.

2.4 Refinement Lattice

In [1], we had introduced a refinement ordering between relational specifications, and explored its lattice properties; in this section, we briefly present the main results of [1], which we need for our subsequent discussions. The following definition introduces an ordering relation between relational specifications.

Definition 1 *Relation R on S is said to* refine *relation R' on S if and only if $RL \cap R'L \cap (R \cup R') = R'$.*

We admit without proof that this is a partial ordering; we denote it by: $R \sqsupseteq R'$ or $R' \sqsubseteq R$. Among the lattice-like properties of this ordering, we cite the following:

- Any two relations R and R' have a greatest lower bound (a *meet*), denoted by $R \sqcap R'$, and given by the following expression:

$$R \sqcap R' = RL \cap R'L \cap (R \cup R').$$

- Relations R and R' have a least upper bound if and only if they satisfy the following condition, which we call the *consistency condition*:

$$(R \cap R')L = RL \cap R'L.$$

- Given two relations R and R' that satisfy the consistency condition, their least upper bound (referred to as their *join*) is denoted by $R \sqcup R'$ and given by the following expression:

$$R \sqcup R' = \overline{RL} \cap R' \cup \overline{R'L} \cap R \cup R \cap R'.$$

- Any two relations have a least upper bound (join) if and only if they have an upper bound.
- The refinement ordering has a universal lower bound, which is the empty relation, but has no universal upper bound.
- Maximal elements of this refinement ordering are total deterministic relations.

Intuitively, the join of two relational specifications represents the functional requirements that they represent in common, whereas the meet of two relational specifications represents the sum of the functional requirements that each represents; this sum is defined only if the specifications do not contradict each other (whence the consistency condition).

3 Invariant Relations

3.1 Loop 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'. From this definition it stems that $dom(G)$ is the set of states s such that if execution of g starts in state s then it terminates (normally). As a convention, we represent programs by lower case letters and their function by the same letter in upper case. Specifications on a space S may be represented by relations on S; while programs are assumed to be deterministic, specifications may be arbitrarily non-deterministic (associate more than one correct final state for a given initial state). The following definition characterizes the correctness of a program with respect to a relational specification as a match between the specification and the program function.

Definition 2 *We consider a program g on space S, and a specification R on S. We say that g is* correct *with respect to R if and only if $G \sqsupseteq R$, where G is the function of g.*

We consider while loops written in some C-like programming language, and we quote the following theorem, due to [32], which we use as the semantic definition of a while loop.

Theorem 1 *(Mili et. al. 2009 [32]) We consider a while statement of the form $w =$* `while t {b}`*. Then its function W is given by:*

$$W = (T \cap B)^* \cap \widehat{\overline{T}},$$

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.

To illustrate our subsequent discussions, we use a simple while loop on natural variables n, f, k: `while (k!=n) {k=k+1; f=f*k;}`.

3.2 Definitions

Intuitively, an invariant relation is a relation that contains all (but not necessarily only) the pairs of states (s, s') such that s' can be derived from s by application of an arbitrary number (including zero) of iterations of the (guarded) loop body. We define it formally as follows.

Definition 3 *Given a while loop of the form $w =$* `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. The following proposition stems readily from the definition.

Proposition 1 *Given a while loop of the form* $w =$ `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* $R_0 \cap R_1$*.*

Proof. We readily prove these results in turn, below:

- Relation $(T \cap B)^*$ is reflexive and transitive, by construction. It is a superset of $(T \cap B)$, by definition.
- The reflexive transitive closure of $(T \cap B)$ is the *smallest* superset of $(T \cap B)$ that is reflexive and transitive; hence it is a subset of any reflexive transitive superset of $(T \cap B)$.
- The intersection of two reflexive transitive relations is reflexive and transitive; the intersection of two supersets of $(T \cap B)$ is a superset of $(T \cap B)$.

qed

To illustrate this concept, we consider the loop of the running example, and we submit the following relation:

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

This relation is clearly reflexive and transitive; we leave it to the interested reader to check that it is a superset of $(T \cap B)$. Other invariant relations include $R' = \{(s, s') | n' = n\}$, and $R'' = \{(s, s') | k \leq k'\}$.

Invariant relations are prone to be confused with invariant assertions, since they look similar to them. To help the reader distinguish between them, we briefly present a relational definition of the latter, and discuss some relevant comparative results. In keeping with our relational approach, we represent invariant assertions by relations, more specifically by vectors.

Definition 4 *A vector* A *is said to be an* invariant assertion *for the while loop* w*:* `while t {b}` *with respect to precondition* ϕ *and postcondition* ψ *if and only if it satisfies the following conditions:*

- $\phi \subseteq A$,
- $A \cap (T \cap B) \subseteq \widehat{A}$,
- $A \cap \overline{T} \subseteq \psi$.

We discuss below how invariant relations and invariant assertions maps to each other.

3.3 From Invariant Relations to Invariant Assertions

The following proposition, due to [15], elucidates how invariant relations can be used to generate invariant assertions.

Proposition 2 *Given an invariant relation R of the while loop w:* `while t {b}`*, the vector $A = \widehat{R}\phi$ is an invariant assertion with respect to precondition ϕ and postcondition $\psi = \widehat{R}\phi \cap \overline{T}$.*

As an illustration of this Proposition, we consider again the sample factorial program, and we consider the invariant relation

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

If we take the precondition represented by the vector

$$\phi = \{(s, s') | f = 1 \wedge k = 0\}$$

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

$$A = \widehat{R}\phi = \{(s, s') | \frac{f}{k!} = \frac{1}{0!}\} = \{(s, s') | f = k!\},$$

$$\psi = \widehat{R}\phi \cap \overline{T} = A \cap \overline{T} = \{(s, s') | f = k! \wedge k = n\} = \{(s, s') | f = n! \wedge k = n\}.$$

The interested reader may choose another (arbitrary) precondition ϕ and compute the corresponding invariant assertion and postcondition.

Proposition 2 provides that any invariant relation can be used to generate an invariant assertion, as the product of the inverse of the invariant relation by the (vector that represents the) precondition. A more interesting question is: are all invariant assertions derived from invariant relation? The answer is provided by the following proposition, due to [33].

Proposition 3 *Given an invariant assertion A of a while loop $w =$* `while t {b}` *on space S, there exists an invariant relation R and a vector P such that $A = \widehat{R}P$.*

In [33], we give a constructive proof of this Proposition, where $R = \overline{A} \cup \widehat{A}$ and $P = A$. This Proposition unveils a structure that is shared by all invariant assertions, as the combination of a loop-specific term (the invariant relation) with a context-dependent term (the precondition), and may be used to gain some insights into the generation of invariant assertions [5, 10, 11, 13, 14, 20–22, 25–28, 31, 34, 39, 43].

3.4 From Invariant Assertions to Invariant Relations

The following Proposition, due to [33], shows how we can derive an invariant relation from an invariant assertion.

Proposition 4 *Given an invariant assertion A of $w =$*`while t {b}`*, the relation $R = \overline{A} \cup \widehat{A}$ is an invariant relation for w.*

On the factorial example, given the following invariant assertion $A = \{(s, s')|f = k!\}$, we can generate the invariant relation:

$$R = \{(s, s')|f = k! \Rightarrow f' = k'!\}.$$

Interestingly, this relation is a superset of the invariant relation we had identified earlier,

$$R' = \{(s, s')|\frac{f}{k!} = \frac{f'}{k!}\}.$$

Hence the latter is a better (more useful) invariant relation than the former. As to the question of whether any invariant relation can be derived from an invariant assertion, the answer is negative: only invariant relations of the form $\overline{A} \cup \widehat{A}$ can; these can be written as $\{(s, s')|\alpha(s) \Rightarrow \alpha(s')\}$, for some unary predicate α on S.

The discussions of this and the previous subsection highlight in what sense we claim that invariant relations are a more general concept than invariant assertions.

3.5 Generating Invariant Relations

The following Proposition (due to [15]) provides a constructive formula for an invariant relation, which can be computed from the function of the loop body and the vector that represents the loop condition.

Proposition 5 *Let w be a while loop of the form* `while t { b}` *on space S. Then $R = I \cup T(T \cap B)$ is an invariant relation for w.*

This invariant relation contains pairs of states (s, s') such that either $s' = s$ (in case s and s' are separated by zero iterations) or s satisfies t and s' is in the range of $(T \cap B)$ (in case s and s' are separated by at least one iteration). We refer to this invariant relation as the *elementary invariant relation* of the loop. The elementary invariant relation is the only invariant relation we get *for free*: we build it constructively from the parameters of the loop. For all other invariant relations, we have to analyze the source code in detail. As a divide-and-conquer discipline, we resolve to structure the function of the loop body in such a way as to streamline the derivation of invariant relations; because invariant relations are supersets of the loop function, it is judicious to structure the function of the loop as an intersection of relations. Once the function of the loop body is written as an intersection, say:

$$(T \cap B) = B_1 \cap B_2 \cap B_3 \cap ... \cap B_k,$$

then any superset of B_1 is a superset of $(T \cap B)$, any superset of $B_1 \cap B_2$ is a superset of $(T \cap B)$, any superset of $B_1 \cap B_2 \cap B_3$ is a superset of $(T \cap B)$, etc. We maintain templates of loop body terms (B_i's), along with their corresponding invariant relations, which we call *recognizers*; we distinguish between 1-recognizers, whose template includes a single term of the intersection, 2-recognizers, whose template includes two terms, and 3-recognizers, whose template includes three terms (we could consider more than three terms, but we have not felt much need for it so far). Whenever the formal template of a recognizer matches an actual term of the intersection (or combination of terms), the corresponding invariant relation is instantiated with the proper variable substitutions, producing an actual invariant relation. Figure 1 shows three examples of recognizers:

Our pattern matching algorithm matches all its 1-recognizers against individual terms of the intersection, then all its 2-recognizers against pairs of terms, then all its 3-recognizers against

ID	variables	constants	condition	loop body	invariant
1R1	x: int;	a: int>0;	true	x = x+a;	$\{(s,s')\|x \ mod \ a = x' \ mod \ a\}$.
2R1	x, y: int	a, b: int	true	x=x+a; y=y+b;	$\{(s,s')\|ay - bx = ay' - bx'\}$
2R2	x, y: int	a: int>0;	x mod a=0	x=x div a; y=a*y;	$\{(s,s')\|xy = x'y'\}$

Figure 1: Sample Recognizers

triplets of terms. If the algorithm is able to connect all the terms of the intersection by means of 2-recognizers or 3-recognizers, then we normally have the necessary information to capture all the functional details of the loop. If not, the algorithm identifies the terms that remain isolated, so that the user can determine what recognizers may be missing from the database, and precluding a complete analysis.

This process works well when the function of the loop body can be written as an intersection of terms: a loop of arbitrary size and complexity can be analyzed by means of 1-, 2-, or 3-recognizers provided we cover (with 1-recognizers) and connect (with 2-recognizers and 3-recognizers) all the terms of the intersection. But we cannot always decide what form the function of the loop body has: if the loop body has if-then-else statements, nested if-then-else statements, or sequences of if-then-else statements, then the outermost structure of its function is a union, not an intersection[1]; we write it as

$$(T \cap B) = B_1 \cup B_2 \cup B_3 \cup ... \cup B_k.$$

In that case, we deploy the pattern matching process discussed above for each term of the union; this results in the union of relations, where each term of the union is reflexive, transitive and is a superset of the corresponding term of $(T \cap B)$; we write it as

$$R = R_1 \cup R_2 \cup R_3 \cup ... \cup R_k.$$

Relation R is a superset of $(T \cap B)$; in addition, it is reflexive, since it is the union of reflexive relations. But we can not claim that it is transitive, because the union of transitive relations is not necessarily transitive. We have written a Mathematica program that takes such a union, and attempts to find the smallest (or a sufficiently small) transitive superset thereof, using the structure of each R_i as an intersection of transitive relations. The process of generation an invariant relation from a union of invariant relations is prone to cause a loss of information (we may not find a sufficiently small invariant relation); the merits and limitations of this approach are discussed in [24], and compared to alternative approaches.

4 Computing Termination Conditions

Now that we know how to generate invariant relations of a loop, we discuss how to use invariant relations to verify the (total) correctness of while loops with respect to a relational specification. We proceed in two steps; first, we focus on computing termination conditions.

[1]If the loop body has loops, then we analyze the inner loops first to determine their function, as shown in [32], and replace them by their closed form function before proceeding to analyze the outer loop.

4.1 A Necessary Condition of Termination

We consider a while loop w on space S, whose function we denote with W. In [32] we have a proposition to the effect that we can, without loss of generality, restrict the space of the loop to $dom(W)$; in other words, if originally the domain of W is a proper subset of S, we may let S be redefined as $dom(W)$; we justify this claim in [32] by showing that all relevant states (initial, intermediate, final states) of the execution of the loop fall within $dom(W)$ (hence we have no reason to look outside). This move affords us the advantage that W is a total function on (the newly defined) S. From a theoretical standpoint, the totality of W is important, because in conjunction with determinacy, it makes W a maximal element in the lattice of refinement (re: Section 2.4); as such, W can be approximated using nothing but lower bounds [32]. This result means that in practice, we compute the domain of W, redefine S as this domain, then apply the results of [32]. The question that arises then is: how do we compute the domain of W. The following Proposition, due to [15], maps invariant relations into necessary conditions of termination.

Proposition 6 *We consider a while loop w on space S, defined by $w =$*`while t {b}`*, and we let R be an invariant relation of w. Then, $WL \subseteq R\overline{T}$, where W is the function of w and T is the vector defined by $T = \{(s, s')|t(s)\}$.*

This proposition provides a necessary condition of termination for the loop, using an invariant relation; any invariant relation yields a necessary condition of termination. Of course, in practice we are interested in necessary conditions that are also sufficient conditions of termination. These can be obtained by choosing invariant relations as small as possible. In practice, we proceed by applying our recognizers, which generate successive invariant relations, and we take the intersection of these invariant relations to which we apply Proposition 6. Two issues arise as we generate successive invariant relations:

- *Target invariant relations that are relevant to termination.* Let R be the current invariant relation (intersection of previously generated invariant relations) and let R_1 be the latest invariant relation. If $(R \cap R_1)\overline{T} = R\overline{T}$, then R_1 is not improving our estimate of the necessary condition of termination. Ideally, we want to be able to recognize such invariant relations, and exclude them from consideration.

- *Recognize when we have identified enough invariant relations to ensure sufficiency.* Proposition 6 ensures that the generated condition is a necessary condition of termination; to ensure that it is also a sufficient condition of termination, we need to generate all the invariant relations that pertain to termination.

In other words, we must endeavor to generate all the invariant relations that are relevant to termination (to ensure sufficiency), and nothing but the relevant invariant relations (for the sake of efficiency and parsimony). We do not have an algorithmic solution to these two requirements, but we are using heuristics, which we are in the process of codifying, organizing, and validating. Examples of such heuristics include:

- Always include the elementary invariant relation, given in Proposition 5, as it characterizes cases when the loop terminates without iterations, and the case when the loop terminates after at least one iteration.

- Generate invariant relations that represent ordering relations ($\leq$, $\geq$) between values of all the variables that are used in the loop condition; as such ordering relations may be key to the termination condition.

As an illustrative example, we consider the factorial program introduced in section 3.1, and we apply this proposition using the intersection of invariant relations R' and R'' introduced in section 3.2 (the intersection of invariant relations is an invariant relation). We find: $R = \{(s, s')|n = n' \wedge k \leq k'\}$. We compute the necessary condition of termination as follows:

$$
\begin{array}{ll}
 & WL \\
\subseteq & \quad \{\text{Proposition 6}\} \\
 & R\overline{T} \\
= & \quad \{\text{substitutions, product}\} \\
 & \{(s, s')|\exists s'' : n = n'' \wedge k \leq k'' \wedge k'' = n''\} \\
= & \quad \{\text{substitutions, simplification}\} \\
 & \{(s, s')|k \leq n\}.
\end{array}
$$

Though we have no proof, we believe that in addition to being provably necessary, this condition is sufficient to ensure termination. We redefine the space S as the set of natural variables n, f, k such that $k \leq n$. On the newly defined space, function W is vacuously total.

4.2 Avoiding Abort

Proposition 6 makes no distinction between whether we are modeling the condition that the number of iterations is finite or the condition that no iteration causes an abort. That distinction stems from the invariant relations that we choose in practice, or equivalently, from the recognizers that we deploy to generate invariant relations. Considering the factorial example above, if we assume a perfect arithmetic, then the only condition under which the loop might not terminate, is that the number of iterations is not bound; this is reflected in the termination condition that was generated.

In this section we discuss what kind of invariant relation we need to generate to reflect the additional condition that successive executions of the loop body terminate normally, without raising exceptions. As a general rule we consider that programs abort when they apply a function to an argument that is outside their domain. Hence as a condition of termination, we want to write:

$$\forall x : App(f, x) \Rightarrow x \in dom(f),$$

where $App(f, x)$ means that function f is applied to argument x. Once this condition is written, the challenge is to generate predicates of the form $App(f, x)$ for all applicable functions and all applicable arguments throughout the program.

As an illustrative example, we consider the case where the function in question is an array reference. Let a be an array of index range $low..high$ and let $ref(h)$ be the predicate: array a is referenced at index h. Then the above formula can be rewritten as:

$$\forall h : ref(h) \Rightarrow low \leq h \leq high.$$

This condition can be generated upon encountering the array declaration, and is included as part of the characterization of the termination condition. What remains to be done is to identify the index values that are indexed through the loop, and generate predicate $ref()$ for them. By the Modus Ponens rule, the condition $low \leq h \leq high$ will be generated for those indices, and only

those. We propose to use invariant relations to characterize those indices in loops. For the sake of readability, we illustrate our idea on a simple loop:

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

Then, we write T and B as follows:

$$T = \{(s, s')|i \neq 0\}$$
$$B = \{(s, s')|ref(k) \wedge i' = i - 1 \wedge x' = x + a[k] \wedge k' = k + 1 \wedge a' = a\}.$$

Our challenge now is to characterize the indices of a that are referenced by the loop, using an invariant relation (i.e. a reflexive transitive superset of $(T \cap B)$). We propose the following relation

$$R = \{(s, s')|\forall h : k \leq h < k' \Rightarrow ref(h)\}.$$

This relation is clearly reflexive since for $k = k'$, the premise of the implication is false. It is also transitive since if $ref(h)$ holds for all h between k and $k' - 1$ and between k' and $k'' - 1$ then it holds between k and $k'' - 1$. Finally, it is a subset of $T \cap B$ since

$$\begin{array}{ll} & T \cap B \\ = & \quad \{\text{substitution}\} \\ & \{(s, s')|i \neq 0 \wedge ref(k) \wedge i' = i - 1 \wedge k' = k + 1 \wedge x' = x + a[k] \wedge a' = a\} \\ \subseteq & \quad \{\text{substitution, rewriting, simplification}\} \\ & \{(s, s')|(\forall h : k \leq h < k' \Rightarrow ref(h)) \wedge k' = k + 1\} \\ \subseteq & \quad \{\text{simplification}\} \\ & \{(s, s')|\forall h : k \leq h < k' \Rightarrow ref(k)\} \\ = & \quad \{\text{substitution}\} \\ R. & \end{array}$$

We consider this invariant relation, along with the elementary invariant relation proposed by Proposition 5, take their intersection, then we apply Proposition 6, yielding the following necessary condition of termination:

$$(i = 0) \vee (i \geq 1 \wedge low \leq k \leq high - i + 1).$$

The reader may verify that this is indeed a necessary condition of abort-free termination; we believe it is sufficient as well.

As a second illustrative example, we consider the following loop on integer variables i, x, and y, where the risk of *abort* stems from a division by zero:

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

We let predicate *divby(x)* be defined as: x is used as a divisor. Then we propose the following generic law (which could be generated implicitly whenever a numeric variable is declared):

$$\forall x : divby(x) \Rightarrow x \neq 0.$$

The challenge, now, is to characterize those values of x for which predicate *divby* holds, in the course of a loop execution. To this effect, we begin by writing the parameters of the loop:

$$T = \{(s, s')|i \neq 0\}$$
$$B = \{(s, s')|divby(x + 1) \wedge i' = i - 1 \wedge x' = x + 1 \wedge y' = y - \tfrac{y}{x+1}\}.$$

In addition to the elementary invariant relation,

$$R_0 = I \cup T(T \cap B),$$

we propose the following invariant relations:

$$R_1 = \{(s, s') | \forall h : x < h \leq x' \Rightarrow divby(h)\},$$
$$R_2 = \{(s, s') | i \geq i'\},$$
$$R_3 = \{(s, s') | x + i = x' + i'\}.$$

We take the intersection of these four relations, and apply Proposition 6; this yields the following necessary condition of termination (which we have reason to believe is also sufficient),

$$(i = 0 \vee (i \geq 1 \wedge (x < i \vee x \geq 0))).$$

In light of these two examples, we further present two more heuristics (for generating invariant assertions that aim to produce a sufficient condition of termination):

- If a function f is applied to some variable x (or, more generally, some expression E), then we must generate an invariant relation that characterizes all the values of x (E) on which function f is applied.
- If a variable x is used in the loop condition and a variable y is used as an argument to a function f that is prone to cause an abort (division by zero, array reference out of bound, etc), then generate an invariant relation that links x and y (to produce the condition under which the loop terminates before the abort-prone function is applied to an illegal argument).

So far we have assumed that we have perfect arithmetic, and have focused our attention on what we refer to as *abort-prone* operations, such as array references out of bounds, divisions by zero, and other similar operations, and have explored what form the corresponding invariant relations take to model these aspects of termination. An extension of this work, which is currently under investigation, is to abandon the hypothesis of perfect arithmetic, and to model arithmetic overflow and underflow; this matter is currently under investigation.

There is a fairly rich literature dealing with proving loop termination, proving loop non-termination, or characterizing necessary or sufficient conditions of termination [2–4, 6–9, 12, 16–18, 29, 34–37, 40–42]. The vast majority of this work centers on the generation of ranking functions of some sort, which tend to equate termination with having a provably finite number of iterations. In this regard, our work is more general, as it also encompasses aborts as a possible cause of non-termination. Another characteristic of existing research on termination is that it deals exclusively with numeric programs, whereas our approach is not restricted to any application domain; whatever we can model with recognizers (and we can subsequently reason about with Mathematica) can be analyzed by our approach.

5 Conditions of Correctness

Once we have computed the termination condition of the loop, we redefine the space of the loop to be the subset of S for which this condition holds, thereby making W vacuously total. Under the hypothesis of totality, we present a necessary condition of correctness and a sufficient condition of correctness, which we use in section 6 to outline an algorithm that analyzes the correctness of while loops.

5.1 Necessary Condition

The following proposition provides a necessary condition of correctness of a loop with respect to a relational specification.

Proposition 7 *Let w be a while loop of the form $w =$* `while t {b}` *that terminates for all states in S, let R be an invariant relation for w, and let V be a specification on S. If w is correct with respect to V then*

$$(V \cap R)\overline{T} = VL.$$

Proof. By hypothesis, and by definition of correctness, we know that W (the function of the loop) refines V. On the other hand, because R is an invariant relation for w, we know by virtue of a theorem due to [32], that W refines $R \cap \widehat{\overline{T}}$. Hence V and $R \cap \widehat{\overline{T}}$ have an upper bound (viz. W); according to a proposition due to [1] (and cited in section 2.4), whenever two relations have an upper bound, they have a least upper bound. Hence, according to a theorem due to [1] (and cited in section 2.4), they satisfy the consistency condition, i.e.

$$(V \cap R \cap \widehat{\overline{T}})L = VL \cap (R \cap \widehat{\overline{T}})L.$$

By virtue of a vector identity, the left hand side can be simplified into: $(V \cap R)\overline{T}$. On the other hand, if we consider the second term of the right hand side, we can simplify it as follows:

$$\begin{array}{ll} & (R \cap \widehat{\overline{T}})L \\ \supseteq & \quad \{(T \cap B)^* \text{ is the smallest invariant relation}\} \\ & ((T \cap B)^* \cap \widehat{\overline{T}})L \\ = & \quad \{\text{semantic definition: Theorem 1}\} \\ & WL \\ = & \quad \{\text{totality of } W\} \\ & L. \end{array}$$

From $(R \cap \widehat{\overline{T}})L \supseteq L$ we infer $(R \cap \widehat{\overline{T}})L = L$, whence the right hand side become CL and the condition becomes: $VL \cap (R \cap \widehat{\overline{T}})L = VL$. **qed**

Even though this is a necessary condition of correctness, we propose to use it (more precisely: its negation) as a sufficient condition of incorrectness; this a similar goal to [38], though we use a different approach. For illustration, we consider the factorial program, and the following specification:

$$V = \{(s, s') | f' = n! \wedge n' = n\}.$$

We let R be the following invariant relation for this loop:

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

and we check the condition of Proposition 8:

$$\begin{array}{ll} & (R \cap V)\overline{T} \\ = & \quad \{\text{substitutions, product}\} \\ & \{(s, s') | \exists s'' : k'' = n \wedge f'' = n! \wedge n'' = n \wedge \frac{f}{k!} = \frac{f''}{k''!} \wedge k'' = n''\} \end{array}$$

$=$ {substitutions, simplifications}
$\{(s, s')|f = k!\}$
$\neq$ {by inspection}
L
$=$ {since V is total}
VL.

Hence the while loop is not correct with respect to V. We do not need to know what function w computes: knowing that R is an invariant relation for w is sufficient to exclude the possibility that w could be correct with respect to V.

5.2 Sufficient Condition

The following proposition uses invariant relations to generate a sufficient condition of correctness of the loop with respect to a relational specification.

Proposition 8 *Given a while loop w of the form* `while t {b}` *that terminates for all states in its space S, and given a specification U on S, if an invariant relation R of w satisfies the condition*

$$R\overline{T} \cap UL \cap (U \cup R \cap \widehat{\overline{T}}) = U$$

then w is correct with respect to U.

Proof. By virtue of a vector identity (for a relation R and a vector v, $((R \cap \widehat{v})L = Rv)$, the hypothesis of the Proposition can be written as:

$$(R \cap \widehat{\overline{T}})L \cap UL \cap (U \cup R \cap \widehat{\overline{T}}) = U.$$

By virtue of the definition of refinement, this can be rewritten as:

$$R \cap \widehat{\overline{T}} \sqsupseteq U.$$

Because R is an invariant relation for w, we know (by virtue of a theorem due to [32]) that

$$W \sqsupseteq R \cap \widehat{\overline{T}}.$$

By transitivity of the refinement, we infer that W refines U, hence w is correct with respect to U. **qed**

We illustrate this proposition on the sample factorial program by taking the following candidate specification:

$$U = \{(s, s')|f = k! \wedge f' = n'!\}.$$

To prove w correct with respect to U, we use the invariant relation $R = \{(s, s')|\frac{f}{k!} = \frac{f'}{k'!}\}$, and we apply Proposition 8:

$R\overline{T} \cap UL \cap (U \cup R \cap \widehat{\overline{T}})$
$=$ {substitutions, simplifications}
$L \cap \{(s, s')|f = k!\} \cap$

```
diagnosistype = {correct, incorrect, undecided};
CorrectnessVerification (S, w, X) // space, loop, specification
   {
    //  declarations
    relation R, cumulR;           // invariant, cumulative invariant
    diagnosistype diagnosis;      // correctness outcome
    //  initializations
    S = S and termination(w);     // restricting S to dom(w)
    cumulR = L;                   // universal relation
    diagnosis = undecided;        // default option
    //  default option
    while more-inv-relations(w)
       {
        R = get-inv-relation(w);  // new invariant relation
        cumulR = cumulR inter R;  // cumulative invariant relation
        if not necessary(R,X)     // the necessary condition is not met
           {diagnosis = incorrect;}
        else
           if sufficient(cumulR,X)// the sufficient condition is met
              {diagnosis = correct;}
       }
    return diagnosis;
   }
```

$$
\begin{array}{ll}
 & (\{(s,s') | f = k! \wedge f' = n'!\} \cup \{(s,s') | \frac{f}{k!} = \frac{f'}{k'!} \wedge k' = n'\}) \\
= & \quad \{\text{factoring, simplification}\} \\
 & \{(s,s') | f = k! \wedge f' = n'!\} \cup \{(s,s') | f = k! \wedge f' = n'! \wedge k' = n'\} \\
= & \quad \{\text{set theory, substitution}\} \\
 & U.
\end{array}
$$

The same invariant relation, $R = \{(s,s') | \frac{f}{k!} = \frac{f'}{k'!}\}$, which was sufficient to rule out the correctness of w with respect to V (above), is sufficient to ensure the correctness of w with respect to specification U. In both cases, we did not need to compute the function of the loop, nor to generate all its invariant relations.

6 An Algorithm for Verifying Loops

Using the three propositions above, we propose the following algorithm for assessing the correctness of a loop w with respect to a relational specification. This algorithm depends on our ability to generate successive invariant relations, which we represent by the operation: `get-inv-relation(w)`; we assume available a boolean function `more-inv-relations(w)` that returns false when we have exhausted all the invariant relations we can extract for w (in practice: when we have deployed all the recognizers available to us), true otherwise. We write it in pseudo-code, using relations as a data type (to represent invariant relations), and using predicates `termination`, `sufficient`, and `necessary`, to represent the conditions of Propositions 6, 8, and 7, respectively.

This algorithm is automated but is not yet fully integrated, and is currently under development/ evolution. Note that while functions `necessary(R,X)` and `sufficient(cumulR,X)` return a boolean value, function `termination(w)` returns a boolean expression, which is subsequently

used to redefine space S. Note also, that function `termination(w)` involves its own loop calling invariant relations; we are currently refining our characterization of which invariant relations may be useful to generate termination conditions, and which may not. We envision defining two distinct (but likely overlapping) databases of recognizers: one that we deploy for termination and one that we deploy for correctness; they may have significant overlap (many invariant relations may be used for both).

7 Illustration

As further illustration of the proposed approach, we consider the following loop:

```
#include <iostream> using namespace std;
const int cN=...; int i, j, fb, nc, np; int main () {
while (j!=cN) {j=j+i; nc=fb; fb=np+nc; np=nc; i=i+1; j=j-i;}}
```

and the following relational specification (where F represents the Fibonacci function):

$$V = \{(s,s')|j = cN \wedge s' = s\} \cup \{(s,s')|j > cN \wedge fb' = F(j+2-cN) \wedge$$

$$nc' = F(j+1-cN) \wedge np' = F(j+1-cN) \wedge i' = i+j-cN \wedge j' = cN\}.$$

Application of Proposition 6 with appropriate invariant relations yields the following termination condition: $j \geq cN$, which we consider as part of the space definition; with this new space, we can now apply Propositions 7 and 8. To check whether this loop is correct with respect to V, we consider the following invariant relation of the loop:

$$R = \{(s,s')|fb' = fb \times F(i'-i+1) + np \times F(i'-i)\},$$

and we apply Proposition 7. To this effect, we observe readily that U is total, hence $UL = L$, and we compute $(R \cap V)\overline{T}$:

$(R \cap V)\overline{T}$

$=$ {substitutions, distributivity}

$\{(s,s')|j = cN \wedge s' = s \wedge fb' = fb \times F(i'-i+1) + np \times F(i'-i) \wedge$
$j' = cN\} \circ L \cup \{(s,s')|j > cN \wedge fb' = fb \times F(i'-i+1) + np \times F(i'-i)$
$\wedge fb' = F(j+2-cN) \wedge nc' = F(j+1-cN) \wedge np' = F(j+1-cN) \wedge$
$i' = i+j-cN \wedge j' = cN\} \circ L$

$=$ {simplifications}

$\{(s,s')|j = cN \wedge s' = s\} \circ L$
$\cup\{(s,s')|j > cN \wedge F(j+2-cN) = fb \times F(j+1-cN) + np \times F(j-cN)$
$\wedge fb' = F(j+2-cN) \wedge nc' = F(j+1-cN) \wedge np' = F(j+1-cN) \wedge$
$i' = i+j-cN \wedge j' = cN\} \circ L$

$=$ {relation product}

$\{(s,s')|j = cN\}$
$\cup\{(s,s')|j > cN \wedge F(j+2-cN) = fb \times F(j+1-cN) + np \times F(j-cN)\}$

Because this relation is not L, we can conclude that the loop w is not correct with respect to V; we do not need to know what function this loop computes, it suffices that we find an invariant relation that is incompatible with V.

As an illustration of the sufficient condition of correctness, we consider the same program as the previous example, and the following specification:

$$U = \{(s, s')|j > cN \wedge np = 1 \wedge fb = 1 \wedge fb' = F(j + 2 - cN) \wedge nc' = F(j + 1 - cN)\}.$$

In order to prove that w is correct with respect to this specification, we submit the following invariant relation:

$$R = \{(s, s')|fb' = np \times F(j - cN) + fb \times F(j + 1 - cN)$$

$$\wedge nc' = np \times F(j - 1 - cN) + fb \times F(j - cN)\}.$$

To check the condition of Proposition 8, we proceed as follows:

$R\overline{T} \cap UL \cap (U \cup R \cap \widehat{\overline{T}})$

$=$ {substitutions}

$\{(s, s')|fb' = np \times F(j - cN) + fb \times F(j + 1 - cN) \wedge$
$nc' = np \times F(j - 1 - cN) + fb \times F(j - cN) \wedge j' = cN\} \circ L$
$\cap\{(s, s')|j > cN \wedge np = 1 \wedge fb = 1 \wedge fb' = F(j + 2 - cN) \wedge$
$nc' = F(j + 1 - cN)\} \circ L \cap (U \cup R \cap \widehat{\overline{T}})$

$=$ {simplifications}

$\{(s, s')|j > cN \wedge np = 1 \wedge fb = 1\} \cap (U \cup R \cap \widehat{\overline{T}})$

$=$ {substitutions, distributivity}

$\{(s, s')|j > cN \wedge np = 1 \wedge fb = 1 \wedge fb' = F(j + 2 - cN) \wedge nc' = F(j + 1 - cN)\}$
$\cup\{(s, s')|j > cN \wedge np = 1 \wedge fb = 1 \wedge fb' = np \times F(j - cN) +$
$fb \times F(j + 1 - cN) \wedge nc' = np \times F(j - 1 - cN) + fb \times F(j - cN)\}$

$=$ {simplification, Fibonacci property}

$\{(s, s')|j > cN \wedge np = 1 \wedge fb = 1 \wedge fb' = F(j + 2 - cN) \wedge nc' = F(j + 1 - cN)\}$
$\cup\{(s, s')|j > cN \wedge np = 1 \wedge fb = 1 \wedge fb' = F(j + 2 - cN) \wedge$
$nc' = F(j + 1 - cN)\}$

$=$ {simplification}

$\{(s, s')|j > cN \wedge np = 1 \wedge fb = 1 \wedge fb' = F(j + 2 - cN) \wedge nc' = F(j + 1 - cN)\}$

$=$ {substitution}

U.

Hence w is correct with respect to U.

Interested readers may view the video at http://web.njit.edu/ mili/cst.exe, where the stepwise analysis of this loop is carried out, one invariant relation at a time.

8 Conclusion

8.1 Summary

In this paper, we have explored the use of invariant relations as a means to establish the correctness of a while loop with respect to a relational specification. Our invariant relation-based approach proceeds by computing the termination condition of the loop, then redefining the space of the loop in such a way as to include only those elements for which the loop terminates. Then we generate successive invariant relations of the loop, and use them to check a sufficient condition and a necessary condition of correctness: as soon as we find enough invariant relations to subsume the

candidate the specification or to contradict it, we can terminate conclusively, with the diagnosis that (respectively) the loop is correct or incorrect. If we run out of invariant relations before we reach one or the other of these conclusions, then our system gives us an indication of what recognizers are missing (causing the lack of invariant relations); if we build the missing recognizers and run it again, we may conclude with a decisive diagnosis.

8.2 Assessment

Because the analysis of while loops is dominated by the use of invariant assertions, we compare our approach to common approaches based on invariant assertions.

- As we discussed in the introduction, our original motivation is to remedy the situation where the proof of a while loop using invariant assertions fails, and we do not know whether the proof failed because the loop is incorrect or because the invariant assertion is inadequate.
- Even assuming that we have determined somehow that the proof failed due to the inadequacy of the invariant assertion, it is not always straightforward to determine whether the invariant assertion needs to be strengthened or weakened. By contrast, with invariant relations we have only one way to go: we must find smaller and smaller (cumulative) invariant relations, thereby capturing increasingly detailed functional information.
- Whereas other researchers use invariant assertions to prove partial correctness and variant functions to prove termination, we use invariant relations to prove termination (or to compute termination conditions) and to prove correctness.
- Whereas other researchers equate termination with having a finite number of iteration, we make no such an assumption, and capture the condition that the number of iterations is finite as well as the condition that each individual iteration terminates normally. In appendix A, we show a non trivial program for which we compute a termination condition that reflects both important aspects of termination: that the number of iterations is finite and that no iteration causes an array reference out of bounds.

8.3 Prospects

We envision three venues for future research:

- Change the current program that generates invariant relations by syntactic matching to make it proceed by semantic matching instead. This allows us to generate more general recognizers, and to broaden the scope of the tool for a given size of the recognizer database.
- Build a domain-specific database of recognizers, and use it to analyze software applications in that domain.
- Refine the characterization of invariant relations that may be useful for termination, and expand the study of termination by lifting the hypothesis of perfect arithmetic.

This research is currently under way.

A Computing a Termination Condition

We consider the following program:

```
#include<iostream>
#include<math.h>
using namespace std;
float taylor(int x, int p)
{int fact=1;
 if(p!=0 && p!=1)  {for(int k=p;k>1;k--) fact=fact*k;}
  return pow(x,p)/fact;}
int main()
{const int N; int i,j,k,x; float y,z; float a[N]; float b[N];
while (j!=0)
 { i=i+1; k=k-i-j; j=j+i-8;   a[i-1]=taylor(x,i-1);
   y=y+a[i-1]; k=k+j+7; z=z+b[k+1]; j=j+1-i;  }}
```

The termination condition for this loop, as computed by Mathematica using the formula of Proposition 6, is the following, where *low* and *high* are the indices of arrays a and b. This formula begins by saying that the loop terminates if $j = 0$ (trivially), or if j is a multiple of 7, then a number of conditions follow, that check on the various configurations of the array bounds and array indices, making sure that no array references are out of bound.

(j=0) ∨ (C[1]∈ Integers ∧ C[1]≥ 1 ∧ j=7×C[1] ∧ (high=low ∧ i=low ∧ k=low ∧ j=7) ∨ (high ≥ low+1 ∧ ((i=low ∧ low≤ k≤ high ∧ 7≤ j≤ 7+7× k-7×low) ∨ (low+1≤ i ≤ high ∧ ((low-1< k≤ high-i+low ∧ 0<j≤ 7+7×k-7×low)∨(high+low-i< k≤ high ∧ 0<j≤ 7+7×high-7×i))))))

References

[1] N. Boudriga, F. Elloumi, and A. Mili. The lattice of specifications: Applications to a specification methodology. *Formal Aspects of Computing*, 4:544–571, 1992.

[2] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In *CONCUR*, pages 488–502, 2005.

[3] Jürgen Brauburger and Jürgen Giesl. Approximating the domains of functional and imperative programs. *Sci. Comput. Program.*, 35(2):113–136, 1999.

[4] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *ASE*, pages 161–169, 2009.

[5] E. Rodriguez Carbonnell and D. Kapur. Program verification using automatic generation of invariants. In *Proceedings, International Conference on Theoretical Aspects of Computing 2004*, volume 3407, pages 325–340. Lecture Notes in Computer Science, Springer Verlag, 2004.

[6] Aziem Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, and Hongseok Yang. Ranking abstractions. In *ESOP*, pages 148–162, 2008.

[7] Michael Colón and Henny Sipma. Practical methods for proving program termination. In *Proc. International Conference on Computer Aided Verification*, CAV '02, pages 442–454, London, UK, UK, 2002. Springer-Verlag.

[8] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 328–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 415–426, New York, NY, USA, 2006. ACM.

[10] E. Denney and B. Fischer. A generic annotation inference algorithm for the safety certification of automatically generated code. In *Proceedings, the Fifth International Conference on Generative programming and Component Engineering*, Portland, Oregon, 2006.

[11] M. D. Ernst, J. H Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

[12] Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *VSTTE*, pages 261–277, 2012.

[13] J.C. Fu, F. B. Bastani, and I-L. Yen. Automated discovery of loop invariants for high assurance programs synthesized using ai planning techniques. In *HASE 2008: 11th High Assurance Systems Engineering Symposium*, pages 333–342, Nanjing, China, 2008.

[14] C. A. Furia and B. Meyer. Inferring loop invariants using postconditions. In Nachum Dershowitz, editor, *Festschrift in honor of Yuri Gurevich's 70th birthday*, Lecture Notes in Computer Science. Springer-Verlag, August 2010.

[15] Wided Ghardallou, Olfa Mraihi, Asma Louhichi, Lamia Labed Jilani, Khaled Bsaies, and Ali Mili. A versatile concept for the analysis of loops. *Journal of Logic and Algebraic Programming*, May 2012.

[16] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logic domains. In *35th ACM Symposium on Principles of Programming Languages*, pages 235–246. ACM, january 2008.

[17] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *POPL*, pages 147–158, 2008.

[18] William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. Alternation for termination. In *SAS*, pages 304–319, 2010.

[19] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576 – 583, October 1969.

[20] K. Hoder, L. Kovacs, and A. Voronkov. Interpolation and symbolic elimination in vampire. In *Proceedings, IJCAR*, pages 188–195, 2010.

[21] R. Iosif, M. Bozga, F. Konecny, and T. Vojnar. Tool demonstration for the FLATA counter automata toolset. In *Proceedings, Workshop on Invariant Generation: WING 2010*, Edimburg, UK, July 2010.

[22] T. Jebelean and M. Giese. *Proceedings, First International Workshop on Invariant Generation*. Research Institute on Symbolic Computation, Hagenberg, Austria, 2007.

[23] Lamia Labed Jilani, Asma Louhich, Olfa Mraihi, and Ali Mili. Invariant relations, invariant functions and loop functions. *Innovations in Systems and Software Engineering: A NASA Journal*, 8(3):195–212, 2012.

[24] Lamia Labed Jilani, Olfa Mraihi, Asma Louhichi, Wided Ghardallou, Khaled Bsaies, and Ali Mili. Invariant relations and invariant functions: An alternative to invariant assertions. *Journal of Symbolic Logic*, May 2012.

[25] L. Kovacs and T. Jebelean. Automated generation of loop invariants by recurrence solving in theorema. In D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, editors, *Proceedings of the 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC04)*, pages 451–464, Timisoara, Romania, 2004. Mirton Publisher.

[26] L. Kovacs and T. Jebelean. An algorithm for automated generation of invariants for loops with conditionals. In D. Petcu, editor, *Proceedings of the Computer-Aided Verification on Information Systems Workshop (CAVIS 2005), 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2005)*, pages 16–19, Department of Computer Science, West University of Timisoara, Romania, 2005.

[27] L. Kovacs and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proceedings, FASE 2009*, pages 470–485. LNCS 5503, Springer Verlag, 2009.

[28] D. Kroening, N. Sharygina, S. Tonetta, A. Letychevskyy Jr, S. Potiyenko, and T. Weigert. Loopfrog: Loop summarization for static analysis. In *Proceedings, Workshop on Invariant Generation: WING 2010*, Edimburg, UK, July 2010.

[29] Wonchan Lee, Bow-Yaw Wang, and Kwangkeun Yi. Termination analysis with algorithmic learning. In *CAV*, pages 88–104, 2012.

[30] A. Louhichi, O. Mraihi, W. Ghardallou, L. Labed Jilani, Kh. Bsaies, and A. Mili. Invariant relations: An automated tool to analyze loops. In *Proceedings, Verification and Evaluation of Computer and Communications Systems*, Tunis, Tunisia, 2011.

[31] E. Maclean, A. Ireland, and G. Grov. Synthesizing functional invariants in separation logic. In *Proceedings, Workshop on Invariant Generation: WING 2010*, Edimburg, UK, July 2010.

[32] A. Mili, S. Aharon, and Ch. Nadkarni. Mathematics for reasoning about loop. *Science of Computer Programming*, pages 989–1020, 2009.

[33] Olfa Mraihi, Asma Louhichi, Lamia Labed Jilani, Jules Desharnais, and Ali Mili. Invariant assertions, invariant relations, and invariant functions. *Science of Computer Programming*, 2012.

[34] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.

[35] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS*, pages 32–41, 2004.

[36] Andreas Podelski and Andrey Rybalchenko. Transition invariants and transition predicate abstraction for program termination. In *TACAS*, pages 3–10, 2011.

[37] Andreas Podelski, Andrey Rybalchenko, and Thomas Wies. Heap assumptions on demand. In *CAV*, pages 314–327, 2008.

[38] P Rümmer and M.A. Shah. Proving programs incorrect using a sequent calculus for java dynamic logic. In *Proceedings, First International Conference on Tests and Proofs*, Zurich, Switzerland, 2007.

[39] S. Sankaranarayana, H. B. Sipma, and Z. Manna. Non linear loop invariant generation using Groebner bases. In *Proceedings, ACM SIGPLAN Principles of Programming Languages, POPL 2004*, pages 381–329, 2004.

[40] Ashish Tiwari. Termination of linear programs. In *CAV*, pages 70–82, 2004.

[41] Aliaksei Tsitovich, Natasha Sharygina, Christoph M. Wintersteiger, and Daniel Kroening. Loop summarization and termination analysis. In *Proc.International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 81–95, 2011.

[42] H. Velroyen and Ph. Rümmer. Non-termination checking for imperative programs. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy*, volume 4966 of *lncs*, pages 154–170. spv, 2008.

[43] F. Zuleger and M. Sinn. LOOPUS: A tool for computing loop bounds for C programs. In *Proceedings, Workshop on Invariant Generation: WING 2010*, Edimburg, UK, July 2010.