Invariant Functions and Invariant Relations: An Alternative to Invariant Assertions

Olfa Mraihi

Institut Superieur de Gestion, Bardo, Tunisia

Asma Louhichi

Faculte des Sciences de Tunis, El Manar, Tunisia

Lamia Labed Jilani

Institut Superieur de Gestion, Bardo, Tunisia

Khaled Bsaies

Faculte des Sciences de Tunis, El Manar, Tunisia

Ali Mili

New Jersey Institute of Technology, Newark, NJ

Abstract

Whereas the analysis of loops in imperative programs is, justifiably, dominated by the concept of invariant assertion, we submit a related but different concept, of invariant relation, and show how it can be used to compute or approximate the function of a while loop. We also introduce the concept of invariant function, which is used to generate a broad class of invariant relations.

Key words: Invariant assertions, invariant relations, invariant functions, while loops, reasoning about loops, invariant generation, loop functions.

Email addresses: olfa.mraihi@yahoo.fr (Olfa Mraihi), louhichiasma@yahoo.fr (Asma Louhichi), lamia.labed@isg.rnu.tn (Lamia Labed Jilani), khaled.bsaies@fst.rnu.tn (Khaled Bsaies), mili@cis.njit.edu (Ali Mili).

1. Invariant Assertions, Invariant Relations, and Invariant Functions

Since its introduction by Tony Hoare in (Hoare (1969)), the concept of invariant assertion has, understandably, played a central role in the static analysis of iterative programs. It has influenced much of the research on program analysis and design since its introduction (Dijkstra (1976); Gries (1981)), and continues to do so to this day; the generation of invariant assertions has remained a topic of great interest during the seventies and early eithies (Cousot and Cousot (1977); Cheatham and Townley (1976); Cousot and Halbwachs (1978)), then has emerged again in the last decade (Carbonnell and Kapur (2004); Colon et al. (2003); Denney and Fischer (2006); Ernst et al. (2006); Fahringer and Scholz (2003); Fu et al. (2008); Jebelean and Giese (2007); Hu et al. (2004); Kovacs and Jebelean (2004, 2005); Podelski and Rybalchenko (2004); Sankaranarayana et al. (2004); Hoder et al. (2010); Kovacs and Voronkov (2009); Maclean et al. (2010); Zuleger and Sinn (2010); Kroening et al. (2010); Iosif et al. (2010); Furia and Meyer (2010)). In this paper, we present invariant relations, a related but distinct concept for the analysis of while loops, and show how this concept can be used to provide complementary insights into the functional attributes of a while loop. In (Jilani et al. (2010)), we discuss the relationships between invariant assertions, invariant relations, invariant functions and loop functions; by contrast with (Jilani et al. (2010)), the theme of this paper is focused primarily on invariant relations and invariant functions, and how they can be used to compute or approximate loop functions.

We briefly and informally present the concepts of invariant relation and invariant function, then discuss the main attributes of our approach to the analysis of while loops, and contrast it to the traditional invariant assertion-based approach.

- An *invariant assertion* of a loop is a predicate that holds after any number of iterations of the loop body.
- An *invariant relation* of a loop is a relation that holds between any two states s and s' of the program such that s and s' are separated by an arbitrary number (0 or more) of iterations of the loop body.
- An *invariant function* of a loop is a function that takes the same value before and after execution of the loop body, assuming the loop condition holds.

As an illustration, we consider the following program on natural variables n, f and k such that $k \leq n+1$:

```
{k:=1; f:=1; while (k!=n+1) {f:=f*k; k:=k+1;}},
```

and we propose the following invariants for this loop:

- *Invariant Assertion, A*: $f = (k-1)!$.
- *Invariant Relation, R*:

$$\left\{ \begin{pmatrix} n & n' \\ f & , f' \\ k & k' \end{pmatrix} | \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!} \right\}.$$

- *Invariant Function, V*:

$$V \begin{pmatrix} n \\ f \\ k \end{pmatrix} = \frac{f}{(k-1)!}.$$

To motivate our approach, we answer in turn the questions raised in the submission guidelines of the special issue.

- *What is the Problem?* The problem we are addressing is that of analyzing while loops in C-like imperative languages. While most researchers approach this problem in the context of proving program correctness with respect to a pre-condition/ post-condition pair, and generate an invariant assertion to this effect, we approach this problem by trying to compute or approximate the function of the loop, and we use invariant relations to this effect.
- *Why is the Problem Important?* This problem is important because C-like imperative languages represent the vast majority of code being developed and maintained nowadays, and loops are the main locus of algorithmic complexity in such programs.
- *What has been done so far on the Problem?* We have developed the theoretical basis that allows us: First, to generate and reason about invariant relations; second, to use invariant relations to compute approximations of loop functions. Also, we have developed and are evolving an automated tool that computes or approximates the function of a loop from a static analysis of its source code.
- *What is the Main Contribution of the Paper to the Problem?* This paper is a progress report on an ongoing research effort, and reports the following new results:
  - A new algorithm for computing the function of while loops whose loop body includes if-then-else statements.
  - An algorithm for identifying missing invariant relation templates whenever the tool fails to compute the loop function in full (and provides an approximation thereof instead).
  - A new set of invariant relation templates derived from Taylor series.
  - An illustration of improved tool functionality, made possible by the recently added invariant relation templates.
- *Is the Contribution Original?* The research direction is original in the sense that it pursues unique research goals, that are distinct from most other teams. The results of this paper are original in the sense that they are presented for the first time in journal form.
- *Is the Contribution Non-Trivial?* We illustrate our approach on non trivial control structures, non trivial data structures (though we have not automated this part yet) and non trivial size.

In the next section, we introduce some mathematical definitions and notations that are needed for the purpose of our discussion, and in section 3 we introduce invariant relations and discuss some of their salient properties. In section 4 we discuss the design and implementation of our algorithm for computing or approximating loop functions by means of invariant relations; and in section 5 we discuss how we enhance our algorithm to deal with non-trivial control structures and non-trivial data structures. In section 6 we discuss to what extent our approach scales up to larger programs, and illustrate our discussion with a program of non-trivial size. We conclude in section 7 by summarizing our results, discussing related work, and discussing prospects of future research.

2. Mathematical Background: A Refinement Lattice

2.1. Relational Mathematics

Most of the material of this section is supposed to be a reminder of simple mathematical concepts, and a presentation of adopted notation, rather than a tutorial; the reader is assumed to be familiar with the concepts discussed herein.

Given a set S, we let a *relation* R on S be a subset of $S \times S$. Among the special relations on S, we mention: the *universal* relation $L = S \times S$, the *empty* relation $\phi = \{\}$, and the *identity* relation $I = \{(s,s)|s \in S\}$. Among the operations on relations, we mention:

- The usual set theoretic operations of *union*, denoted by $R \cup R'$, *intersection*, denoted by $R \cap R'$, *difference*, denoted by $R \setminus R'$, and *complement*, denoted by $\overline{R}$.
- In addition, we mention the following relation-specific operations: the *inverse*, denoted by $\widehat{R}$, and defined by $\widehat{R} = \{(s,s')|(s',s) \in R\}$; the *product*, denoted by $R \circ R'$, or RR' when no ambiguity arises, and defined by $R \circ R' = \{(s,s')|\exists t : (s,t) \in R \wedge (t,s') \in R'\}$; the *nucleus* of R, denoted by $\mu(R)$ and defined by $\mu(R) = R\widehat{R}$.
- The n^{th} *power* of relation R, for n natural, is denoted by R^n and defined as follows: if $n = 0$ then I else $R^{n-1} \circ R$. The *transitive closure* of relation R is denoted by R^+ and defined by: $R^+ = \{(s,s')|\exists n > 0 : (s,s') \in R^n\}$. The *reflexive transitive closure* of relation R is denoted by R^* and defined by: $R^* = \{(s,s')|\exists n \geq 0 : (s,s') \in R^n\}$.
- Given a subset A of S, we let $I(A)$ be defined by $I(A) = \{(s,s)|s \in A\}$; also, we define the *pre-restriction* of relation R to set A as $I(A) \circ R$ and the *post-restriction* of relation R to set A as $R \circ I(A)$.
- Given a relation R on S, we let the *domain* of relation R be the set denoted by $dom(R)$ and defined by $\{s|\exists t : (s,t) \in R\}$; we let the *range* of R be denoted by $rng(R)$ and defined by $dom(\widehat{R})$.

Among the properties of relations, we mention: a relation R is said to be *total* if and only if $RL = L$; a relation R is said to be *deterministic* (or to be a *function*) if and only if $\widehat{R}R \subseteq I$; a relation R is said to be a *vector* if and only if $RL = R$.

A relation R is said to be *reflexive* if and only if $I \subseteq R$; a relation is said to be *transitive* if and only if $RR \subseteq R$; a relation is said to be *symmetric* if and only if $\widehat{R} = R$; a relation R is said to be an *equivalence* if and only if it is reflexive, symmetric and transitive. We admit without proof that the transitive closure of relation R is the smallest transitive relation that is a superset of or equal to R, and that the reflexive transitive closure of relation R is the smallest reflexive transitive relation that is a superset of or equal to R. We also admit that if R is total and deterministic then its nucleus can be written as: $\{(s,s')|R(s) = R(s')\}$. This is an equivalence relation, whose equivalence classes share the same image by R.

A relation R is said to be *antisymmetric* if and only if $R \cap \widehat{R} \subseteq I$; a relation R is said to be a *partial ordering* if and only if it is reflexive, transitive, and antisymmetric. A partial ordering is said to be a *lattice* if and only if any two elements of S, say x and y, have a unique least upper bound (also called a *join*), which we denote by $x \sqcup y$ and a greatest lower bound (also called a *meet*), which we denote by $x \sqcap y$.

2.2. Relational Semantics

In the Mills's logic of program verification and analysis, specifications are represented by relations and programs are represented by functions (Mills (1975); Linger et al. (1979)); program correctness is determined by matching the function defined by the program against the relation that represents the specification. We introduce an ordering relation between relational specification: we say that specification R *refines* specification R' if and only if:

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

This relation is denoted by $R \sqsupseteq R'$ (R refines R') or $R' \sqsubseteq R$ (R' is refined by R). Intuitively, this ordering means that R represents a stronger specification than R'. The following two propositions, which we present without proof, convey this idea:

- Program p is correct with respect to specification R (in the sense of total correctness (Dijkstra (1976); Gries (1981); Manna (1974)) if and only if the function of p refines specification R.
- Specification R refines specification R' if and only if any program correct with respect to R is correct with respect to R'.

In (Boudriga et al. (1992)), we find that the refinement relation is a partial ordering, and explore its lattice properties, which we summarize as follows:

- Any two specifications have a greatest lower bound, which is given by the formula:

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

  The specification $R \sqcap R'$ captures the requirements that are represented simultaneously by R and R'.
- Two specifications R and R' have a least upper bound if and only if they satisfy the following condition, which we call the *consistency condition*:

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

  Because the existence of least upper bound is conditional (not all pairs have a least upper bound), the refinement ordering is not a lattice; nevertheless, we will continue to refer to this structure as the *lattice of specifications* or the *lattice of refinement*, in reference to its lattice-like properties.
- The least upper bound (join) of two specifications that satisfy the consistency condition is given by the following formula:

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

  The specification $R \sqcup R'$ captures all the requirements that are in R and all the requirements that are in R'. It is possible to capture the requirements of R and R' in a specification only if they do not contradict each other, which is what the consistency condition represents.
- The lattice of specifications has a unique universal lower bound, which is the empty relation.
- The lattice of specifications has no universal upper bound.
- Maximal elements of the lattice of specifications are total deterministic relations.
- Any two specifications have a least upper bound if and only they have a (common) upper bound; this result is known in general in lattice theory (Davey and Priestley (1990)), but because our refinement structure is not, strictly speaking, a lattice, it bears checking specifically.

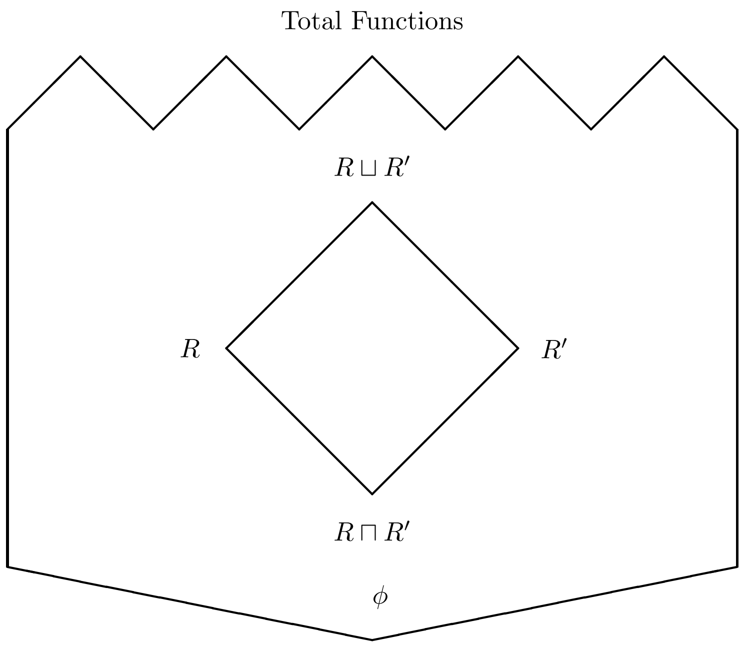

Fig. 1. Lattice Structure of Refinement

See Figure 1. We use this lattice structure as a basis for our stepwise approach to the derivation of loop functions; this will be discussed in section 4.

3. Invariant Relations and Invariant Functions

For the sake of readability, the presentation of this material is meant to be intuitively appealing rather than strictly formal; interested readers are referred to (Mili et al. (2009)) for details. Given a set of program variables, say x, y, z of types X, Y, Z, we let a *state* s of the program be defined as any aggregate of values that these variables may take, and we let $x(s)$, $y(s)$ and $z(s)$ denote the $x-$ (respectively $y-$, $z-$) component of s. Hence for example, if $s = \langle 2, 6, 0 \rangle$, then $x(s) = 2$, $y(s) = 6$, $z(s) = 0$. We let the *space* of a program be the set of its states, and we usually denote the space by S. We consider a program p on space S, and we let P be the function that p computes on its space, defined as follows:

$$P = \{(s, s')|\text{if program } p \text{ starts execution in state } s \text{ then it terminates in state } s'\}.$$

It stems from this definition that the domain of relation P can be written as:

$$dom(P) = \{s|\text{if program } p \text{ starts execution in state } s \text{ then it terminates}\}.$$

We focus specifically on while loops in C-like programming languages, i.e. programs of the form: `w: while t {b}`, where t is a predicate and b is a block of code that represents the loop body. We let W be the function of the while loop (w), we let B be the function of

b, and we let T be the vector defined by $T = \{(s,s')|t(s)\}$. We assume that w terminates for all states in S, from which we infer that W is total; we discuss in (Mili et al. (2009)) in what sense and to what extent this hypothesis is justified. We present the following definition.

Definition 1 (Invariant Relations)**.** We consider a while loop of the form w: `while t {b}` on space S that terminates for all initial states, and we let B be the function of b and T be defined as $T = \{(s,s')|t(s)\}$. A relation R on S is said to be an *invariant relation* for w if and only if it satisfies the following conditions:
- *Inductive Condition*: R is reflexive and transitive.
- *Invariance Condition*: $T \cap B \subseteq R$.
- *Convergence Condition*: $R\overline{T} = L$.

To illustrate this definition, we consider again the factorial program, whose loop is: w = `while (k!=n+1) {f=f*k; k=k+1;}`. We consider again the relation

$$R = \{(s,s')|\frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\},$$

where we use f, k as shorthands for $f(s)$, $k(s)$, and f', k' as shorthands for $f(s')$, $k(s')$, and we check that this relation satisfies the conditions of the definition:
- Inductive condition: R is clearly reflexive and transitive.
- Invariance condition: if $k \neq n+1$ and $f' = f \times k$ and $k' = k+1$, then $\frac{f'}{(k'-1)!} = \frac{f \times k}{k!} = \frac{f}{(k-1)!}$. Hence $T \cap B$ is a subset of R.
- Convergence condition: $R\overline{T} = \{(s,s')|\frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\} \circ \{(s,s')|k = n+1\} = \{(s,s')|\exists s" : \frac{f}{(k-1)!} = \frac{f"}{(k"-1)!} \wedge k" = n" + 1\} = \{(s,s')|\exists s" : f" = \frac{f \times n"!}{(k-1)!} \wedge k" = n" + 1\} = L$.

To give an intuitive understanding of invariant relations, we offer below two possible interpretations, ignoring for a moment the convergence condition (which is related to the hypothesis that w terminates for all initial states):
- *The Invariant Relation as an Approximation of* $(T \cap B)^*$. We have proven in (Mili et al. (2009)) that the function of the loop can be written as $W = (T \cap B)^* \cap \widehat{\overline{T}}$. In practice, we cannot use this formula to compute the function of a loop because it is generally impossible to derive the reflexive transitive closure of $(T \cap B)$. Now, an invariant relation is a reflexive transitive superset of $(T \cap B)$; as such, it is an approximation of the reflexive transitive closure of $(T \cap B)$, which is by definition the *smallest* reflexive transitive superset of $(T \cap B)$.
- *The Invariant Relation as an Inductive Argument.* The reflexivity and transitivity of R can be used as (respectively) the basis of induction and the inductive step of an inductive proof to the effect that any pair of states (s,s') such that s' is obtained from s by application of an arbitrary number of iterations, is in R. While this holds for any pair (s,s'), it holds in particular for the pair made up of the initial state s and the final state s'.

The following theorem explains why invariant relations are important for our purposes.

Theorem 2 (Mili et al. (2009))**.** *We consider a while loop of the form w:* `while t {b}` *on space S that terminates for all initial states, and we let W be the function of this loop, and R be an invariant relation for w. Then W refines $R \cap \widehat{\overline{T}}$.*

We refer to $U = R \cap \widehat{\widehat{T}}$ as a *lower bound* of W; note that $R \cap \widehat{\widehat{T}}$ is nothing but the post-restriction of R to $\neg t$. As we recall, the lattice of refinement has no universal upper bound, and the maximal elements of this lattice are total deterministic relations. Also, because we are dealing with deterministic languages, and because we assume that the while loop terminates for all its initial states, we can infer that W is total and deterministic, hence it is maximal in the refinement lattice. As such, it can be approximated by means of lower bounds; in other words, this theorem is interesting because it enables us to derive a lower bound of W from any invariant relation of the while loop. Furthermore, if we can find invariant relations R_1, R_2, R_3, ..., R_k, and use them to derive lower bounds U_1, U_2, U_3, ..., U_k, then U_1, U_2, U_3, ..., U_k have an upper bound, namely W, hence they do have a least upper bound, which we write as

$$U = U_1 \sqcup U_2 \sqcup U_3 \sqcup ... \sqcup U_k.$$

Because W is an upper bound of all the U_i's, it is an upper bound of their join, i.e. $W \sqsupseteq U$. If U is total and deterministic, then $W = U$, since the only upper bound to a maximal element is the element itself. If U is not total and deterministic, then it is the best approximation we could compute for W, given the invariant relations we are able to find.

To illustrate this process, we consider again the factorial example, and we let R_1 be the invariant relation we had already identified, i.e.

$$R_1 = \{(s, s')|\frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\}.$$

From this relation, we derive a lower bound for W, which is

$$\begin{array}{ll}
 & U_1 \\
= & \qquad \{ \text{ Theorem 2 } \} \\
 & R_1 \cap \widehat{\widehat{T}} \\
= & \qquad \{ \text{ substitution } \} \\
 & \{(s, s')|\frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\} \cap \{(s, s')|k' = n' + 1\} \\
= & \qquad \{ \text{ taking the intersection } \} \\
 & \{(s, s')|\frac{f}{(k-1)!} = \frac{f'}{(k'-1)!} \wedge k' = n' + 1\}
\end{array}$$

This relation is not deterministic, since we have three variables to determine, n', f', and k', but only two equations. Hence we must find other invariant relations; we propose,

$$R_2 = \{(s, s')|n = n'\},$$

$$R_3 = \{(s, s')|k \leq k'\}.$$

Note that these relations are reflexive and transitive, and they are both supersets of the loop body function; we do not detail the convergence condition, though it too is verified. From these invariant relations we derive lower bounds U_1 and U_2, and find:

$$U_2 = \{(s, s')|n = n' \wedge k' = n' + 1\},$$

$$U_3 = \{(s, s')|k \leq k' \wedge k' = n' + 1\}.$$

Taking the join of U_1, U_2, and U_3, which in this case is the intersection (because the three relations have the same domain), we find:

$$U = \{(s,s')|n = n' \wedge k \leq k' \wedge \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!} \wedge k' = n'+1\},$$

which can be simplified to

$$U = \{(s,s')|k \leq n+1 \wedge n' = n \wedge f' = \frac{f \times n!}{(k-1)!} \wedge k' = n+1\}.$$

Given that we have defined the space of our program by natural variables n, f, k, such that $k \leq n+1$, that clause needs not be cited in the definition of U, hence we write:

$$U = \{(s,s')|n' = n \wedge f' = \frac{f \times n!}{(k-1)!} \wedge k' = n+1\}.$$

Because this is a total deterministic relation, it is maximal in the refinement lattice, hence from $W \sqsupseteq U$, we infer $W = U$, whence

$$W = \{(s,s')|n' = n \wedge f' = \frac{f \times n!}{(k-1)!} \wedge k' = n+1\}.$$

Hence, given enough (and sufficiently refined) invariant relations, we can approximate or compute the function of the loop —which raises the question: How do we compute invariant relations? Invariant functions, discussed below, are an important part of the answer to this question.

Definition 3 (Invariant Functions)**.** We consider a while loop of the form w: `while t {b}` on space S that terminates for all initial states, and we let B be the function of b and T be defined as $T = \{(s,s')|t(s)\}$. A function V on S is said to be an *invariant function* for w if and only if it is total and satisfies the condition: $T \cap V = T \cap BV$.

Intuitively, an invariant function is one that takes the same value before and after application of $(T \cap B)$, i.e. application of B when t holds.[1] To illustrate the concept of invariant functions, we offer the following examples:

$$V_1 \begin{pmatrix} n \\ f \\ k \end{pmatrix} = \begin{pmatrix} \frac{f}{(k-1)!} \\ 0 \\ 0 \end{pmatrix},$$

$$V_2 \begin{pmatrix} n \\ f \\ k \end{pmatrix} = \begin{pmatrix} n \\ 0 \\ 0 \end{pmatrix}.$$

[1] Note incidentally that an invariant assertion, in the sense of (Hoare (1969)), is not a boolean-valued invariant function, since an invariant assertion may be false before application of the loop body and become true afterwards: consider the assertion $x > 0$ in the loop `while (x<100) do {x=x+1}`. This leads us to suggest that a better name for invariant assertions is: *monotonic assertions*.

To check that these are invariant functions, we write function B:

$$B\begin{pmatrix} n \\ f \\ k \end{pmatrix} = \begin{pmatrix} n \\ f \times k \\ k+1 \end{pmatrix}.$$

We leave it to the reader to check, easily, that under the condition $t(s)$, we have $V_1(B(s)) = V_1(s)$ and $V_2(B(s)) = V_2(s)$, whence V_1 and V_2 are invariant functions.

The question that we raise now is: why do we need invariant functions? The following theorem answers this question.

Theorem 4 (Jilani et al. (2010))**.** *We consider a while loop of the form w:* `while t {b}` *on space S that terminates for all initial states.*

- *If V is an invariant function for w, then the nucleus of V (i.e. $\mu(V) = V\widehat{V}$) is an invariant relation for w.*
- *If R is a symmetric invariant relation for w, then there exists an invariant function V of w whose nucleus is R.*

According to this theorem, we can derive an invariant relation from any invariant function; this is interesting, since in practice it is easy to find invariant functions (Mili et al. (1985)). But the question that arises then is: do invariant functions allow us to cover the set of all the invariant relations? This theorem provides that all invariant relations that are symmetric (in addition to being reflexive and transitive) are the nuceli of some invariant function, which is fair since the nucleus of a function is necessarily symmetric. Hence by focusing on finding invariant functions we can derive all the symmetric invariant relations of a loop; we will need alternative means to derive antisymmetric invariant relations. As we recall, an invariant relation contains pairs of the form (s, s') such that s' follows s by an arbitrary number of iterations; symmetric relations make no ordering between s and s', either one could precede the other; whereas antisymmetric relations do specify that s precedes s'.

As an illustration, we consider again the factorial program and we submit that invariant relation R_1 (presented above) is the nucleus of invariant function V_1, and that invariant relation R_2 is the nucleus of invariant function V_2. As for invariant relation R_3, it is not symmetric, hence cannot be written as the nucleus of an invariant function. In practice, symmetric invariant relations carry a lot more information about the function of the loop (as illustrated by relations R_1, R_2, and R_3 above); nevertheless, we will deploy both types, as needed.

To summarize the foregoing discussion, we can derive the function of a while loop by proceeding through the following sequence: we derive invariant functions (by inspection), from which we derive invariant relations (by taking the nuclei of invariant functions), from which we derive lower bounds (by post-restricting invariant relations to $\neg t$), from which we derive the function of the loop or an approximation thereof (by taking the join of all the lower bounds and checking for totality and determinacy).

4. Computing Loop Functions

In this section we discuss the design and implementation of an automated tool that computes or approximates the function of a loop from an analysis of its source code. Our tool proceeds in three steps:

- *From Source Code to Relational Form: cpp2cca.* The purpose of this step is two-fold: First, to map the source code into a uniform internal notation, that is independent of the programming language, so that subsequent steps can be reused for any programming language. Second, to prepare the loop for the generation of invariant relations, which takes place in the next step. As we recall, an invariant relation of a loop is a superset of the function of its loop body. In order to facilitate the search for invariant relations, we rewrite the function of the loop body as an intersection of relations, say

  $$B = B_1 \cap B_2 \cap B_3 \cap ... \cap B_m.$$

  Once B is written in this form, we can find supersets of B by taking a superset of B_1, or by taking a superset of $B_1 \cap B_2$, or by taking a superset of $B_1 \cap B_2 \cap B_3$, etc. If we consider a loop body that is made up of a sequence of assignment statements, we can rewrite it as an intersection by eliminating the sequential dependencies between statements and writing, for each program variable, the cumulative effect of all relevant assignment statements. We obtain what is called *concurrent assignments*, or more generally *conditional concurrent assignments* (abbreviated: CCA) (Collins et al. (2005); Hevner et al. (2005)), although conditions will appear only when we consider conditional statements, in section 5. This step maps source code (in, say, C++) into CCA code (file extention: .cca), and is carried out by a simple compiler, using standard compiler generation technology.
- *From CCA code to Lower Bounds: cca2mat.* Once the loop body is written as an intersection, we can derive an invariant relation, hence a lower bound for W, by considering one term of the intersection at a time, or two terms of the intersection at a time, or three terms of the intersection at a time. This stepwise strategy is at the heart of our algorithm, in that it allows us to handle potentially large loops (see section 6) by looking at no more than a few CCA statements at a time. We use a pattern matching procedure to generate invariant relations, whereby statements or combinations of statements from the CCA code are matched against pre-stored templates of CCA code for which we also store an invariant relation template. When a match is successful, we instantiate and generate the corresponding invariant relation. The aggregate made up of a template of variable types, code patterns and corresponding invariant relation template is called a *recognizer*; we distinguish between 1-recognizers, whose code template includes a single CCA statement, 2-recognizers, whose code template includes two CCA statements, and 3-recognizers, whose code template includes three CCA statements. There is nothing magical about the number 3; we stopped at 3 because we found that for the applications we have considered so far, there is no need to look at more than three CCA statements at a time, but we expect to introduce recognizers of size 4 or more as we broaden the scope of application of our tool. With larger recognizer sizes, we expect the algorithm to be more slightly more complex, and significantly more time-consuming. For the sake of illustration, we present below some simple recognizers, that will be needed to solve the factorial example.

ID	State Space	CCA Code Pattern	Invariant Relation
1R1	int n;	n=n	$\{(s,s')\|n(s)=n(s')\}$
1R2	int k;	k=k+1	$\{(s,s')\|k(s)\leq k(s')\}$
2R1	int k, f;	f=f*k, k=k+1	$\{(s,s')\|\frac{f}{(k-1)!}=\frac{f'}{(k'-1)!}\}$

It may look to the reader that we produce custom-tailored recognizers for each new loop we encounter; of course, this is not the case. Each recognizer we introduce can be used in any program where its code pattern occurs; recognizers are our way of conveying to the tool the necessary programming knowledge and domain knowledge that it needs to analyze loops. Because the code patterns of the recognizers are small (1 to 3 CCA statements, so far), it is very easy to match them in large programs. In a recent experiment, we ran our tool with its current database of 40 recognizers, to analyze all the loops that we encounter in papers on invariant generation; we were able to compute the function of all the programs we encountered, except two. For the two that we missed, we identified the missing recognizers and are planning to add them.

As to the question of how do we find the invariant relation corresponding to each pattern of CCA statements, this is where we use invariant functions. The question we ask is: what function is preserved when the transformation represented by the CCA statements is applied; once we find the function, say V, we write the invariant relation R as $\{(s,s')|V(s)=V(s')\}$. Another method that we may use if our intuition fails us is to write the recurrence relations represented by the CCA statements then try to solve them by eliminating the recurrence variable. As for antisymmetric invariant relations, they are usually very easy to generate, as shown with recognizer 1R2.

- *Solving the Equations that Characterize Invariant Relations: mat2bn.* The step cca2mat generates invariant relations in the form of equations between initial states (represented by unprimed variable names) and final states (represented by primed variable names). All we have to do now is solve these equations in the primed variables, as a function of the unprimed variables. We carry out this step using *Mathematica*, ©Wolfram Research. If Mathematica produces an expression for each primed program variable, we infer that the relation herein defined is deterministic; if it does so without imposing any condition on unprimed variables, we infer that the relation is also total. Being total and deterministic, the relation is maximal in the refinement lattice, hence it is the function of the loop. Else what we have produced is not the function of the loop but an approximation thereof, the best approximation we can muster with the database of recognizers we have; in that case, the tool offers suggestions for missing recognizers, as we discuss in section 5.

The derivation of the loop function by successive approximations is illustrated in Figure 2; as we identify more and more lower bounds of the loop function, the resulting join climbs in the lattice of refinement; if the join hits the ceiling, we obtain the function of the loop, if not we only obtain an approximation thereof. To illustrate this algorithm, we apply it to the factorial example, where we show the various forms of the loop from its source code form to its functional form.

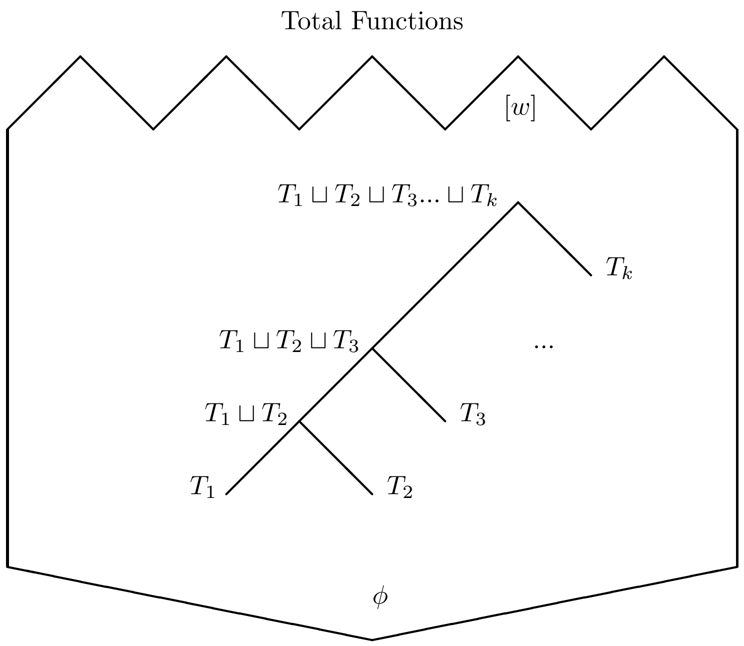
Total Functions
$[w]$
$T_1 \sqcup T_2 \sqcup T_3 ... \sqcup T_k$
T_k
$T_1 \sqcup T_2 \sqcup T_3$
...
$T_1 \sqcup T_2$
T_3
T_1
T_2
ϕ

Fig. 2. Successive Approximations of the Loop Function

- **Source code, C++**:
  `int n, f, k; while (k!=n+1) {f=f*k; k=k+1;}`.
  We assume in fact that n, f, k are natural and that $k \leq n+1$.
- **Internal representation, CCA**:
  `int n, f, k; while (k!=n+1) {n=n, f=f*k, k=k+1}`.
  We make the same assumptions as above.
- **Invariant Relations, MAT**:
  · Application of recognizer 2R2 to second and third statements yields:
  $$R_1 = \{(s,s')|\frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\}.$$
  · Application of recognizer 1R1 to first statement yields:
  $$R_2 = \{(s,s')|n' = n\}.$$
  · Application of recognizer 1R2 to third statement yields:
  $$R_2 = \{(s,s')|k \leq k'\}.$$
  Generating Lower Bounds from the invariant relations, and taking the join, we find the following compound lower bound:
  $$U = \{(s,s')|\frac{f}{(k-1)!} = \frac{f'}{(k'-1)!} \wedge n' = n \wedge k \leq k' \wedge k' = n'+1\}.$$

- **Loop Function or Approximation**: Solving these equations in the primed variables yields, we find the following formula for the lower bound:

$$U = \{(s, s')|n' = n \wedge f' = f \times \frac{n!}{(k-1)!} \wedge k' = n + 1\}.$$

  Because U is total and deterministic, it is the function of the loop. Hence, we find:

$$W = \{(s, s')|n' = n \wedge f' = f \times \frac{n!}{(k-1)!} \wedge k' = n + 1\}.$$

5. Non Trivial Program Structures

5.1. Non Trivial Control Structures

One of the key ideas of our algorithm, and the basis of its stepwise analysis strategy, is that we write the function of the loop body as an intersection (which is the CCA form) so that we can derive invariant relations for the loop by looking at arbitrarily partial information (i.e. arbitrarily few CCA statements) at a time. The trouble with this approach is that we do not usually get to choose what form the function of the loop body: for example, if the loop body includes if-then-else statements, then the outermost structure of its function is a union, not an intersection. We have two approaches to this problem, which we discuss in turn.

5.1.1. Merging Invariant Relations

We consider a while loop whose loop body is written in the CCA notation as a set of conditional concurrent assignments:

```
while t
    {(cond1) -> {a11, a12, a13, ..., a1k},
     (cond2) -> {a21, a22, a23, ..., a2k},
     (cond3) -> {a31, a32, a33, ..., a3k},
      ... ... ... ,
     (condh) -> {ah1, ah2, ah3, ..., ahk}
    }
```

The first thing we need to acknowledge is that the conditions can be used to generate stronger invariant relations than we would generate by looking only at the concurrent assignment statements. Consider for example the conditional concurrent assignment statements:

```
(x%2==0) -> {x=x/2, y=2*y}
```

where x and y are natural variables. Given that the division of x by 2 takes place for even values of x, it produces no remainder, hence we can say that the product of x by y is preserved by these conditional concurrent assignments; i.e. we could generate the invariant relation $\{(s, s')|xy = x'y'\}$. Without the condition that x is even we cannot ensure the preservation of xy, as one can easily verify with $x = 5$ and $y = 2$: before the concurrent assignments, $xy = 10$, whereas after the assignments, $x = 2$ and $y = 4$, hence $xy = 8$.

Whereas the intersection structure allowed us to generate invariant relations locally, by looking at few terms of the intersection at a time, the union structure does not allows such luxury: in order to prove that a relation R is a superset of the union, we have to prove that it is a superset of each term of the union, which precludes the stepwise approach that we have taken before. Hence we proceed as follows:

- We define conditional recognizers, which include, in addition to the variable declarations and statement templates, a condition under which the statements are applied. The corresponding invariant relation template is then generated according to the statements at hand, and the condition under which the statements are applied. These recognizers are applied to statements of the same branch, not to statements across branches.
- We consider that there is a match between a conditional recognizer and an actual code instance if and only if the variable declarations and concurrent statements of the CCA code match the variable declarations and statement template of the recognizer, and if the condition of the CCA code logically implies the condition of the recognizer.
- We match each branch of the conditional loop body separately, using the conditional recognizers, and end up with a reflexive transitive superset of each branch. Let's call these relations R_1, R_2, R_3, ..., R_h; the union of these relations is a superset of the loop body, and it is reflexive, but it is not transitive (the union of transitive relations is not necessarily transitive). We discuss below how to derive an invariant relation of the loop from relations R_1, R_2, R_3, ..., R_h.

We have devised a method for extracting an invariant relation from a set of relations R_1, R_2, R_3, ..., R_h obtained from applying the conditional recognizers to the h branches of the loop body. We discuss this method for two branches, then generalize its application to a larger number of branches. Let the loop body be composed of two CCA branches, and let B_1 and B_2 be the functions of these branches. Let R_1 and R_2 be derived from these two branches, respectively; by construction, we know that R_1 is a superset of B_1 and R_2 is a superset of B_2; also, since they have been generated by recognizers, relations R_1 and R_2 are the intersections of several reflexive and transitive relations, which we write as,

$$R_1 = R_{11} \cap R_{12} \cap R_{13} \cap ... \cap R_{1m},$$
$$R_2 = R_{21} \cap R_{22} \cap R_{23} \cap ... \cap R_{2n}.$$

We consider the following derivation, which produces a reflexive transitive superset of $T \cap B$:

$$\begin{array}{ll}
 & T \cap B \\
\subseteq & \qquad \{ \text{ lifting the restriction of } B \ \} \\
 & B \\
= & \qquad \{ \text{ structure of } B \text{ as the union of two branches } \} \\
 & B_1 \cup B_2 \\
\subseteq & \qquad \{ \text{ by construction of } R_1 \text{ and } R_2\text{, as discussed above } \} \\
 & R_1 \cup R_2 \\
= & \qquad \{ \text{ idempotence } \} \\
 & (R_1 \cup R_2) \cap (R_1 \cup R_2) \\
= & \qquad \{ \text{ substitution } \} \\
 & ((R_{11} \cap R_{12} \cap R_{13} \cap ... \cap R_{1m}) \cup R_2) \cap \\
 & (R_1 \cup (R_{21} \cap R_{22} \cap R_{23} \cap ... \cap R_{2n})) \\
= & \qquad \{ \text{ distributivity } \} \\
 & (R_{11} \cup R_2) \cap (R_{12} \cup R_2) \cap (R_{13} \cup R_2) \cap ... \cap (R_{1m} \cup R_2) \cap \\
 & (R_1 \cup R_{21}) \cap (R_1 \cup R_{22}) \cap (R_1 \cup R_{23}) \cap ... \cap (R_1 \cup R_{2n})
\end{array}$$

$=$ { assuming, e.g. $R_2 \subseteq R_{11}$, $R_2 \subseteq R_{12}$, $R_1 \subseteq R_{21}$, $R_1 \subseteq R_{22}$ }

$R_{11} \cap R_{12} \cap (R_{13} \cup R_2) \cap ... \cap (R_{1m} \cup R_2) \cap$

$R_{21} \cap R_{22} \cap (R_1 \cup R_{23}) \cap ... \cap (R_1 \cup R_{2n})$

$\subseteq$ { An intersection grows larger as we delete terms thereof }

$R_{11} \cap R_{12} \cap R_{21} \cap R_{22}$

Let R be defined as $R = R_{11} \cap R_{12} \cap R_{21} \cap R_{22}$. According to the derivation above, R is a superset of $T \cap B$; on the other hand, R is the intersection of several relations that are generated by recognizers, hence are by construction reflexive and transitive; the intersection of reflexive and transitive relations is likewise reflexive and transitive. We admit that R also satisfies the convergence condition (all relations generated by recognizers do); hence R thus constructed is an invariant relation of the loop.

We need to explain some steps above: after the distributivity step, we obtain the intersection of several terms of the form $(R_1 \cup R_{2j})$ or $(R_2 \cup R_{1j})$; we review each one of these terms and check whether the term of the union that represents a complete branch (R_1 or R_2) is a subset of the other term (that represents a clause of the other branch); if it is, then the union can be simplified to the second term. In the derivation above, we assumed the following inclusions, $R_2 \subseteq R_{11}$, $R_2 \subseteq R_{12}$, $R_1 \subseteq R_{21}$, $R_1 \subseteq R_{22}$, and we simplified the unions accordingly (when one term of the union is a subset of the other, the union equals the larger set). In the subsequent step, we simply delete all the terms of the intersection that contain a union, because we cannot ensure that they are transitive. Once we delete all the terms that have a union, we are left with the intersection of terms that are known to be reflexive and transitive.

We have automated this step by writing a Mathematica program (*Merge*) that merges R_1 and R_2 into an invariant relation; this program does all the list processing involved, and checks automatically for the inclusion relations. It produces a set of equations in s and s', that are then resolved in s'. If we have more than two branches, the algorithm proceeds two by two: for example, it handles three branches R_1, R_2, R_3 by calling: $Merge(Merge(R_1, R_2), R_3)$. To illustrate this algorithm, we consider a simple while loop on integer variables x and z and natural variable y:

```
while (y!=1) {if (y%2==0) {y=y/2; x=2*x;} else {y=y-1; z=z+x}}.
```

The CCA version of this loop is written as:

```
while (y!=1)
  { (y mod 2==0) -> {y=y/2, x=2*x, z=z},
     (y mod 2!=0) -> {y=y-1, z=z+x, x=x}
  }
```

The relations generated by the recognizers for the two branches are:

$$R_1 = \{(s, s') | z = z' \wedge xy = x'y' \wedge x \times 2^{\lfloor log_2(y) \rfloor} = x' \times 2^{\lfloor log_2(y') \rfloor}$$

$$R_2 = \{(s, s') | x = x' \wedge z + xy = z' + x'y' \wedge \lfloor log_2(y) \rfloor = \lfloor log_2(y') \rfloor\}.$$

We let

- $R_{11} = \{(s, s') | z = z'\}$,
- $R_{12} = \{(s, s') | xy = x'y'\}$,
- $R_{13} = \{(s, s') | x \times 2^{\lfloor log_2(y) \rfloor} = x' \times 2^{\lfloor log_2(y') \rfloor}\}$,
- $R_{21} = \{(s, s') | x = x'\}$,
- $R_{22} = \{(s, s') | z + xy = z' + x'y'\}$ and

- $R_{23} = \{(s,s')|\lfloor log_2(y)\rfloor = \lfloor log_2(y)\rfloor\}$.

We then write $R_1 \cup R_2$ as:

$$\begin{array}{ll}
 & R_1 \cup R_2 \\
= & \qquad \{ \text{ substitution, factorization } \} \\
 & (R_1 \cup R_{21}) \cap (R_1 \cup R_{22}) \cap (R_1 \cup R_{23}) \\
 & (R_{11} \cup R_2) \cap (R_{12} \cup R_2) \cap (R_{13} \cup R_2) \\
= & \qquad \{ \text{ because } R_1 \subseteq R_{22} \ \} \\
 & (R_1 \cup R_{21}) \cap (R_{22}) \cap (R_1 \cup R_{23}) \\
 & (R_{11} \cup R_2) \cap (R_{12} \cup R_2) \cap (R_{13} \cup R_2) \\
= & \qquad \{ \text{ because } R_2 \subseteq R_{13} \ \} \\
 & (R_1 \cup R_{21}) \cap (R_{22}) \cap (R_1 \cup R_{23}) \\
 & (R_{11} \cup R_2) \cap (R_{12} \cup R_2) \cap (R_{13}) \\
\subseteq & \qquad \{ \text{ deleting terms that contain a union } \} \\
 & R_{22} \cap R_{13}.
\end{array}$$

Whence we find the following invariant relation:

$$R = R_{22} \cap R_{13} = \{(s,s')|z + xy = z' + x'y' \wedge x \times 2^{\lfloor log_2(y)\rfloor} = x' \times 2^{\lfloor log_2(y')\rfloor}\}.$$

From this invariant relation, we derive a lower bound for the loop function by taking the post restriction to $(y = 1)$, which yields:

$$U = \{(s,s')|z + xy = z' + x'y' \wedge x \times 2^{\lfloor log_2(y)\rfloor} = x' \times 2^{\lfloor log_2(y')\rfloor} \wedge y' = 1\}.$$

We rewrite this relation in such a way as to highlight the expressions of primed variables:

$$U = \{(s,s')|x' = x \times 2^{\lfloor log_2(y)\rfloor} \wedge y' = 1 \wedge z' = z + xy - x \times 2^{\lfloor log_2(y)\rfloor}\}.$$

Because this is a total deterministic relation, it is actually the function of the loop. Hence we write:

$$W = \{(s,s')|x' = x \times 2^{\lfloor log_2(y)\rfloor} \wedge y' = 1 \wedge z' = z + xy - x \times 2^{\lfloor log_2(y)\rfloor}\}.$$

We have tested this loop against an oracle formed from this function, varying each of the variables x, y, z over a range of 100 values, for a total test size of a million; all the tests were successful.

As a second example, we consider a slightly more complex example, with more complicated arithmetic, and three branches:

```
int main ()
   {
     int x, y, z, t, v, w;
     while (x!=1)
        {if (x%4==0) {x=x/4; y=y+2; t=t*4; v=pow(v,4);}
          else if (x%2==0) {x=x/2; y=y+1; t=t*2; v=pow(v,2);}
                else {x=x-1; z=z+t; w=w*v;}
   }
```

Using the conditional recognizers listed in table 1, our tool produce the following function for this loop.

ID	State Space	Condition	Code Pattern	Invariant Relation
1R3	int x;	x mod 2 =1	x=x-1	$\{(s,s')\|\lfloor log_2(x)\rfloor = \lfloor log_2(x')\rfloor\}$

2R12	int x,y;	x mod 2 =0	x=x/2
			y=y+1
				$\{(s,s')\|y + log_2(x) = y' + log_2(x')\}$

2R13	int x,y;	x mod 4 =0	x=x/4
			y=y+2
				$\{(s,s')\|y + log_2(x) = y' + log_2(x')\}$

2R14	int x,y;	x mod a =0	x=x/a $y = y^a$	$\{(s,s')\|xlog_2(y) = x'log_2(y')\}$

2R15	int x; int y;
	const int a,b;
		true	$x = x^a$
			y=y+b
				$\{(s,s')\|b.log_a(ln(x)) - y =$
				$b.log_a(ln(x')) - y'\}$

2R16	int x,y;
	const int a,b;
		true	x=x+a
			y=by,
				$\{(s,s')\|\frac{y}{b^{\frac{x}{a}}} = \frac{y'}{b^{\frac{x'}{a}}}\}$

3R4	int x,y,z;
	const int a,b;
		true	x=x-a
			y=y+b*z
			z=z
				$\{(s,s')\|z = z' \wedge$
				$ay + bxz = ay' + bx'z'\}$

3R5	int x,y,z;
	const int a;
		true	x=x-a
			y=y*z
			z=z
				$\{(s,s')\|z = z' \wedge$
				$a.log_2(y) + xlog_2(z) =$
				$a.log_2(y') + x'log_2(z')\}$

Table 1. Conditional Recognizers

$$W = \left\{ (s,s')| \begin{array}{c} x' = 1 \wedge y' = y + \lfloor log_2(x) rfloor \wedge t' = t \times 2^{\lfloor log_2(x)\rfloor} \wedge \\ v' = v^{2^{\lfloor log_2(x)\rfloor}} \wedge w' = w \times v^{x - 2^{\lfloor (log_2(x)\rfloor}} \wedge \\ z' = z + tx - t \times 2^{\lfloor log_2(x)\rfloor} \end{array} \right\}.$$

5.1.2. Composing Invariant Relations

One issue in the algorithm presented above concerns us: when we delete all the terms of the form $(R_i \cup R_{jh})$ that do not simplify to R_{jh}, we may be inadvertently deleting too much information, hence making it impossible to compute the function of the loop (or as good an approximation as we could). Interestingly, this has not happened to us so far: in all the examples we have worked through so far, this algorithm was able to derive an invariant relation of the loop without loss of information. But we are not prepared to assume that we should always be so lucky; in this section, we consider an alternative venue, articulated in the following theorem.

Theorem 5. *We consider a while loop of the form w:* `while t {b}` *on space S that terminates for all initial states, and we let B be the function of b; we assume that B can be written as the union of two terms, say $B = B_1 \cup B_2$. If we let R_1 be a reflexive transitive superset of B_1 and R_2 a reflexive transitive superset of $B_2 R_1$ then $R_1 R_2$ is an* *invariant relation for w, provided it is transitive and satisfies the convergence condition, i.e.* $R_1R_2\overline{T} = L$.

As an illustration of this theorem, we consider the first loop discussed in the previous section and derive an invariant relation for it using the formula proposed therein.

```
    while (y!=1) {if (y%2==0) {x=x/2; y=2*y ;} else {y=y-1; z=z+x;}}
```

We find that the function of the loop body can be written as the union of two terms,

$$B_1 = \{(s,s')|y\ mod\ 2 = 0 \wedge x' = x/2 \wedge y' = 2 \times y \wedge z' = z\}$$

$$B_2 = \{(s,s')|y\ mod\ 2 \neq 0 \wedge x' = x \wedge y' = y - 1 \wedge z' = z + x\}$$

We apply theorem 5 to this loop, whose first step involves finding a reflexive transitive superset of B_1. Using the conditional recognizers listed in table 1, we find the following relation that is a reflexive transitive superset of B_1:

$$R_1 = \{(s,s')|z = z' \wedge xy = x'y' \wedge x \times 2^{\lfloor Log_2 y\rfloor} = x' \times 2^{\lfloor Log_2 y'\rfloor}\}$$

According to Theorem 5, we must now find a relation R_2 that is a reflexive transitive superset of B_2R_1; to this effect, we compute B_2R_1.

B_2R_1

$=$ { Substitutions }

$\{(s,s')|y\ mod\ 2 \neq 0 \wedge x' = x \wedge y' = y - 1 \wedge z' = z + x\}$

$\circ\{(s,s')|z = z' \wedge xy = x'y' \wedge x \times 2^{\lfloor log_2 y\rfloor} = x' \times 2^{\lfloor log_2 y'\rfloor}\}$

$\subseteq$ { We apply conditional recognizers to B_2, which yields }

$\{(s,s')|\lfloor log_2(y)\rfloor = \lfloor log_2(y')\rfloor \wedge x' = x \wedge z + xy = z' + x'y' \wedge y \geq y'\}$

$\circ\{(s,s')|z = z' \wedge xy = x'y' \wedge x \times 2^{\lfloor log_2 y\rfloor} = x' \times 2^{\lfloor log_2 y'\rfloor}\}$

$=$ { Definition of Relational Product }

$\{(s,s')|\exists s" : \lfloor log_2(y)\rfloor = \lfloor log_2(y")\rfloor \wedge x" = x \wedge z + xy = z" + x"y" \wedge y \geq y" \wedge$
$z" = z' \wedge x"y" = x'y' \wedge x" \times 2^{\lfloor log_2 y"\rfloor} = x' \times 2^{\lfloor log_2 y'\rfloor}\}$

$=$ { Logical Inference }

$\{(s,s')|z + xy = z' + x'y' \wedge x \times 2^{\lfloor log_2 y\rfloor} = x' \times 2^{\lfloor log_2 y'\rfloor} \wedge$
$\exists s" : \lfloor log_2(y)\rfloor = \lfloor log_2(y")\rfloor \wedge x" = x \wedge z + xy = z" + x"y" \wedge y \geq y" \wedge$
$z" = z' \wedge x"y" = x'y' \wedge x" \times 2^{\lfloor log_2 y"\rfloor} = x' \times 2^{\lfloor log_2 y'\rfloor}\}$

$\subseteq$ { Simplification }

$\{(s,s')|z + xy = z' + x'y' \wedge x \times 2^{\lfloor log_2 y\rfloor} = x' \times 2^{\lfloor log_2 y'\rfloor}\}$

Interestingly, we find the same invariant relation as before; as we recall, this invariant relation will yield the function of the while loop once we post restrict it to $(y = 1)$.

5.2. Non Trivial Data Structures

We consider the following C++ program that operates on some user defined data structure, called `itemtype`, and some aggregate data structure, which is a list of such itemtypes; it is on purpose that we make the code unclear, for reasons that we elucidate below.

```
#include <iostream>
#include <list>
using namespace std;
```

```
typedef struct
   {int x; float y;}
   itemtype;
int main ()
  {list<itemtype> l1;  list<itemtype> l2 ;  list<itemtype> l3;
   l3.clear();
   while( !l1.empty() && !l2.empty())
      {if(l1.front().x>l2.front().x)
          {l3.push_back(l1.front());l1.pop_front();}
       else
          {if (l2.front().x>l1.front().x)
             {l3.push_back(l2.front()); l2.pop_front();}
           else
              {itemtype val;
               val.y = l1.front().y + l2.front().y;
               val.x = l1.front().x; l3.push_back(val);
               l1.pop_front(); l2.pop_front();}}
       }
   while(!l1.empty())
      {l3.push_back(l1.front());  l1.pop_front();}
   while (!l2.empty())
      {l3.push_back(l2.front());  l2.pop_front();}
  }
```

We use this example to illustrate the premise that computing the function of a loop is a *domain-dependent* task, in the following sense: computing the function of any program, written in some programming language (say, C++), consists in mapping the program from a domain-neutral representation (the code source) to a domain-specific representation, which expresses the function of the program in terms that are meaningful in the application domain, and that refer to domain concepts, domain abstractions, and domain axiomatizations, more generally to domain knowledge. As long as we were computing the function of numeric programs, this dichotomy between the programming notation and the domain notation was not warranted, because numeric types are a native data type of the programming languages. But for user defined data types, the program must be interpreted in domain-specific terms before its function can be computed. Consider for example that the program written above can be interpreted in three different ways, depending on how we interpret the user defined data type called *itemtype*.

- *Student Records*. The data structure *itemtype* represents student records, where x represents the student's ID and y represents the student's grade point average. Then lists $l1$ and $l2$ represent two class rolls, and the program places in $l3$ the combined student transcripts.
- *Merchandise Descriptor*. The data structure *itemtype* represents merchandise, where x represents the article's ID and y represents the article's available quantity (we assume that some merchandise may be tallied in volume or in quantity, rather in number of articles). Then lists $l1$ and $l2$ represent the stock of two warehouses, and the program places in $l3$ the combined available stock.
- *Monomial*. The data structure *itemtype* represents a monomial, where x is the exponent and y is the coefficient of the monomial. Then $l1$ and $l2$ represent two polynomials and the program computes the sum of $l1$ and $l2$ into $l3$.

- *Tuple*. The data structure *itemize* represents a pair (x, y) of a function from integers to reals, and the program computes in $l3$ the deterministic union of $l1$ and $l2$ (where the deterministic union of two functions takes the value of each function outside the domain of the other function, and takes the sum of their values in the intersection of their domains).

In order for an automated tool to begin analyzing this program and deriving its function, it must first determine (with the help of the user) which interpretation to use, out of these four and possibly others. Once an interpretation is chosen, we must:

- Map the program data structures onto the selected data types.
- Reinterpret the program in terms of the selected data type, using domain-specific notations, concepts, and assumptions.
- Upload all the axiomatisations pertaining to the selected data type, to be used for subsequent analysis and transformations.

We refer to this phase as *reverse modeling*, because it maps data structures to data types, thereby reversing the modeling phase of software design, which maps domain-specific data types to data structures. We envision this phase as taking place early in the stepwise derivation of program functions, whereby the first step is cpp2dom, where the DOM notation is a representation of the program in terms of the selected domain. This step is followed by dom2cca, that maps the code into CCA notation, and the stepwise transformation proceeds as discussed in section 4, only with a domain-specific notations (involving domain-specific data types, and associated operations, axiomatizations, abstractions, notations, etc) rather than programming notations (involving native data structures and native operations). As an illustration of the cpp2dom transformation, we consider again the cryptic program given above, and we reinterpret it on the basis that `itemtype` represents monomials and `list<itemtype>` represents polynomials. Also, we introduce the following notations:

- The *degree* of a polynomial is the highest exponent of the polynomial.
- The *top* of a polynomial is the monomial of highest exponent.
- The *tail* of a polynomial is the polynomial we obtain by deleting the monomial of highest degree.
- Function *addmonomial* appends a monomial of lower exponent to a polynomial whose monomials all have higher exponents.

```
#include <iostream>
#include <list>
using namespace std;
typedef struct
   {int exponent; float coefficient;}
   monomial;
typedef list<monomial> polynomial;
int main ()
  {polynomial l1;  polynomial l2 ;  polynomial l3;
   l3.clear();
   while( !l1.empty() && !l2.empty())
      {if (degree(l1)>degree(l2))
          {addmonomial(l3,top(l1)); l1=tail(l1);}
       else
          {if (degree(l2)>degree(l1))
             {addmonomial(l3,top(l2)); l2=tail(l2);}
          else
             {monomial val;
              val.coefficient = top(l1).coefficient + top(l2).coefficient;
              val.exponent = top(l1).coefficient;
      addmonomial(l3,val);
              l1=tail(l1); l2=tail(l2);}}
      }
   while(!l1.empty())
      {addmonomial(l3,top(l1));  l1=tail(l1);}
   while (!l2.empty())
      {addmonomial(l3,top(l2));  l2=tail(l2);}
  }
```

Subsequent steps involve mapping this program to CCA format by eliminating sequential dependencies, then generating invariant relations by means of polynomial-specific recognizers, then solving the equations that result from the invariant relations by a combination of a theorem prover and an algebraic system, using domain knowledge as needed. We are currently exploring the automation of these steps for selected application domains.

6. Non Trivial Size

Whereas in the previous section we have considered complex programs, in this section we consider large programs. Two obvious issues arise when we are dealing with large programs:

- First, the combinatorial explosion that results from trying to match p (p=1, 2, 3) statements of the loop body out of N statements, against K recognizers, trying in turn all permutations of the statements. The number of operations required for this task is bound by $O(\frac{N!}{(N-p)!} \times K \times p!)$. For small values of p (currently 3), $p!$ is a small constant, and the expression above is linear in K; hence the factor that we must focus on is $\frac{N!}{(N-p)!}$. A simple observation enables us to scale that factor down from N^p to N: We observe that recognizers reflect the property that the statements they involve are executed an equal number of times; for example, a recognizer that links statements st1 and st2 is merely saying that these two statements are executed the same number of times. We let G be the graph whose nodes are the CCA statements of the loop body, and we let there be an arc between two nodes of this graph if and only if we have matched a recognizer against the statements representing these nodes. Because the relation "being executed the same number of times" is transitive, it is not necessary to build a complete graph on the nodes representing CCA statements (by matching all the pairs of statements or triplets of statements by a recognizer); rather, it is sufficient to build a connected graph on this set of nodes. For a set of N nodes, $N-1$ arcs are sufficient to make the graph connected.

  Hence, as we try to match combinations of CCA statements against recognizers, we maintain a graph G that represents direct connections between statements and a graph G^* that represents transitive links between statements. We use these graphs as follows: whenever a match between between a 2-recognizer and two statements, say i and j, is successful, we place 1 in $G(i,j)$ and $G(j,i)$; also, whenever a match between a 3-recognizer and three statements, say i, j, and k is successful, we place 1 in $G(i,j)$, $G(j,k)$, $G(j,i)$, and $G(k,j)$. Also, after each successful match, we update G^* as the transitive closure of G, and we stop as soon as G^* is full. On the other hand, if G^* is not full, we only try to match those pairs of statements that have 0 in G^*. This simple device reduces the computing time of our algorithm for generating invariant relations, and makes it easier and faster for Mathematica to solve the equations we generate, since we generate significantly fewer equations.

- Second, When we exhaust all our available recognizers and still cannot fill graph G^*, we conclude that it is because we are missing recognizers for the loop at hand. In that case, our tool uses matrix G^* to offer suggestions for the statements that ought to be linked by recognizers. Whenever G^* has a zero in some entry, say $G^*(i,j)$, the system proposes the pair of statements (i,j) as a candidate for a 2-recognizer; also, whenever G^* has two zeros in the same row or the same column, say $G^*(i,j)$ and $G^*(i,k)$, then the system proposes the triplet (i,j,k) as a possible candidate for a 3-recognizer. Note that the system may propose a large number of candidates for recognizers; it does not mean that we have to generate one recognizer for each; the minimal number of required recognizers is determined by the number of connected components in G^*: if this graph has 3 connected components, e.g., then one 3-recognizer or two 2-recognizers may be enough to make it connected.

To illustrate the ability of our system to handle loops of non-trivial size, we consider the following C++ program; this program has 5 integer constants, 9 integer variables, 18 real scalar variables, two real arrays, and two lists; the loop body of this program has 35 assignment statements; we let `Fact()` be defined as the factorial function, and let `pow` be defined as the power function.

```
#include <iostream>
#include <list>
#include<math.h>
using namespace std;
int Fact (int z)
int main()
  {
   const int ca; const int cb; const int cd; const int ce; const int cN;
   int i, j, k, h, y, m, n, q, w;
   float ma, st, ut, x1, x2, t, p, n, g, r, s, u, v, z, ta, ka, la, uv;
   list <int> l1; list <int> l2;
   float aa[]; float ab[];

   while (l2.size()!=0)
      { r=pow(i,5)+r; s=s+2*u; k=ca*h+k; la=pow(x1,j)/Fact(j) + la;
        l1.push_back(l2.front()); h=h+j; m=m+1; j=j+i; g=g-15*cd;
        q=1+2*i+q; ma=ka-ma; y=y+i; n=n-i; i=i+1; st=st+aa[i]; x2=x2-8;
        j= j-i; w=4*w; ut=ut+ab[j]; n=n+y; ma=(cd+1)*ka - ma; ka=ka-1;
        ta=pow(ta,3); x2=2+x2/4; y=3+y-i; z=8*t+z; t=t+2*j; ka=3+3*ka;
        w=cd+w/2; p=2*pow(p,3); m=2*m-2; n=2+(n-y)/2; s=(cb-2)*u+s;
        h=h-1+cb-j; g=3*cd+g/5; v=pow(v,4); u=ca+u; uv=pow(uv,5);
        t= 4*t-8*j; l2.pop_front();
      }
  }
```

Table 2 represents the recognizers that were used to analyze this program. The resulting function is given in Figure 3, where $sl2$ represents the size of list $l2$, and Γ is Euler's

Gamma function, defined as follows:

$$\Gamma[z] = \int_0^\infty t^{z-1} e^{-t} dt.$$

$$\Gamma[a, z] = \int_z^\infty t^{a-1} e^{-t} dt.$$

Recognizers 2R9 and 2R10 are due to a benchmark of Aligator's Demo (Group (2010)). Recognizer 3R3 is generated using Taylor series of the exponential function; we initially write the invariant relation corresponding to the code pattern of 3R3 as:

$$y + \sum_{k=1}^{i} \frac{x^k}{k!} = y' + \sum_{k=1}^{i'} \frac{x'^k}{k!},$$

which we rewrite as

$$y + \sum_{k=1}^{\infty} \frac{x^k}{k!} - \sum_{k=i+1}^{\infty} \frac{x^k}{k!} = y' + \sum_{k=1}^{i'} \frac{x'^k}{k!},$$

then we replace the term $\sum_{k=i+1}^{\infty} \frac{x^k}{k!}$ by its Taylor formula; so, while the invariant relation of recognizer 3R3 does not appear to be, it is actually reflexive and transitive (as well as symmetric). This explains where the exponential function and the Gamma function come from in the function of this loop.

7. Concluding Remarks

7.1. Summary

In this paper we have presented the concept of invariant relations, and shown how this concept can help us in analyzing the functional properties of while loops in a deterministic C-like language. In particular we have shown how we can use invariant relations to derive or approximate the function of a while loop, and discussed the issue of scaling up by condiering how to deal with complex control structures, complex data structures, and large size programs. A key pillar of our approach is the artifact of recognizer, which generates invariant relations by matching source code configurations against pre-catalogued code patterns for which we know an invariant relation. In our approach, recognizers are the artifact that we use to codify programming knowledge and domain knowledge, without which no analysis of source code is possible. The scope of application of our approach depends critically, for better or for worst, on the contents of our database of recognizers. To deploy our approach in a particular domain, we need to enter domain-specific recognizers, that capture domain knowledge in the form of domain abstractions, domain concepts, domains axioms, etc; in addition to a core set of domain independent recognizers that capture general programming language attributes.

It is noteworthy that the effectiveness of our approach does not increase linearly with the size of our recognizer database, but combinatorically: each new recognizer can be combined with all the existing recognizers to support an ever increasing range of code configurations. Also, because our approach proceeds by pattern matching, it can deal

ID	State Space	Code Pattern	Invariant Relation
1R1	int x;	x=x+1	$\{(s,s')\|x \leq x'\}$
1R2	int x;	x=x-1	$\{(s,s')\|x \geq x'\}$

2R1	int x,y;
	const int a,b;
		x=x+a
		y=y+b
			$\{(s,s')\|ay - bx = ay' - bx'\}$

2R2	int x,y;
	const int a,b;
		x=x+a
		y=by
			$\{(s,s')\|\frac{y}{b^{\frac{x}{a}}} = \frac{y'}{b^{\frac{x'}{a}}}\}$

2R3	int x,y;
	const int a,b,c;
		x = ax + b
		y = y + c
			$\{(s,s')\|\frac{(1-a)x-b}{a^{\frac{y}{c}}} = \frac{(1-a)x'-b}{a^{\frac{y'}{c}}}\}$

2R4	int x,y;
	const int a,b;
		x=x+a
		y=y+bx
			$\{(s,s')\|y - \frac{bx(x-a)}{2a} = y' - \frac{bx'(x'-a)}{2a}\}$

2R5	int x; real y;
	const int a;
		x=$\frac{x}{a}$
		y=y+1
			$\{(s,s')\|y + log_a(x) = y' + log_a(x')\}$

2R6	real x; int y;
	const int a,b,c;
		x=bx^a
		y=y+c
			$\{(s,s')\|\frac{(1-a)log_b(x)-1}{a^{\frac{y}{c}}} = \frac{(1-a)log_b(x')-1}{a^{\frac{y'}{c}}}\}$

2R7	real x; int y;
	const int a,b;
		$x = x^a$
		y=y+b
			$\{(s,s')\|b.log_a(ln(x)) - y = b.log_a(ln(x')) - y'\}$

2R8	real x,y;	x=2x
		y=$\frac{y}{2} + b$
			$\{(s,s')\|x(y-2) = x'(y'-2)\}$

2R9	int x,y;	x=x+2y+1
		y=y+1
			$\{(s,s')\|x - y^2 = x' - y'^2\}$

2R10	int x,y;	x=$x + y^5$
		y=y+1
			$\{(s,s')\|12x - 2y^6 + 6y^5 - 5y^4 + y^2 =$
			$12x' - 2y'^6 + 6y'^5 - 5y'^4 + y'^2\}$

2R11	int x,y;	x=ax
		y=bx+y
			$\{(s,s')\|y + \frac{bx}{1-a} = y' + \frac{bx'}{1-a}\}$

3R1	int x,i;
	int a[N];
		i=i-1
		x=x+a[i-1]
		a=a
			$\{(s,s')\|a' = a \wedge$
			$x + \sum_{k=1}^{i-1} a[k] = x' + \sum_{k=1}^{i'-1} a'[k]\}$

3R2	int x,i;
	int a[N]
		i=i+1
		x=x+a[i+1]
		a=a
			$\{(s,s')\|a' = a \wedge$
			$x + \sum_{k=i+1}^{N} a[k] = x' + \sum_{k=i+1}^{N} a'[k]\}$

3R3	real x,y;
	int i;
		$x = \frac{x^i}{i!}$
		i=i-1
		x=x
			$\{(s,s')\|x' = x \wedge$
			$y + Exp[x] - 1 - \sum_{k=i+1}^{\infty} \frac{x^k}{k!} = y' + \sum_{k=1}^{i'} \frac{x^k}{k!}\}$

Table 2. Sample Recognizers

$$W = \left\{ (s's) \middle| \begin{array}{c} aa' = aa \wedge ab' = ab \wedge st' = st + \sum_{ai=i+1}^{i+sl2} aa[ai] \\ \wedge ut' = ut + \sum_{ai=j-sl2}^{j-1} ab[ai] \wedge l1' = Concat[l1, l2] \wedge l2 = () \\ \wedge size(l1') = size(l1) + sl2 \wedge size(l2') = 0 \wedge x1' = x1 \wedge \\ la' = \frac{la\Gamma(j+1)\Gamma(2+j-sl2)+e^{x1}\Gamma(2+j-sl2)\Gamma(1+j,x1)}{\Gamma(1+j)\Gamma(2+j-sl2)} \\ + \frac{-e^{x1}\Gamma(1+j)\Gamma(1+j-sl2,x)-e^{x1}j\Gamma(1+j)\Gamma(1+j-sl2,x1)+e^{x1}\Gamma(1+j)\Gamma(1+j-sl2,x1)sl2}{\Gamma(1+j)\Gamma(2+j-sl2)} \\ ma' = ma + cd \times ka \times \frac{3^{sl2}-1}{2} \wedge i' = i + sl2 \wedge j' = j - sl2 \wedge ta' = ta^{3^{sl2}} \\ \wedge ka' = ka \times 3^{sl2} \wedge h' = h - cb\ sl2 \wedge k' = k + \frac{2ca\ h\ sl2 + ca\ cb\ sl2^2 - ca\ cb\ sl2}{2} \\ \wedge x2' = x24^{-sl2} \wedge y' = y + 2sl2 \wedge t' = t4^{sl2} \wedge w' = w2^{sl2} + cd(2^{sl2} - 1) \\ \wedge (p' = 2^{\frac{3^{sl2}-1}{2}} p^{3^{sl2}} \vee sl2 = 0 \wedge p' = p) \wedge (m' = m2^{sl2} \vee m = 0 \wedge m' = 0) \\ \wedge (n' = 2^{-sl2}(n - 2 + 2^{1+sl2}) \wedge n \neq 2 \vee n = 2 \wedge n' = 2) \\ \wedge q' = q + 2\ i\ sl2 + sl2^2 \wedge s' = \frac{1}{2}(2s - ca\ cb\ sl2 + 2cb\ u\ sl2 + ca\ cb\ sl2^2) \\ \wedge r' = \frac{12r-2\ i\ sl2+20\ i^3 sl2-3-i^4 sl2+12\ i^5 sl2-12sl2^2+30\ i^2 sl2^2-60i^3 sl2^2+30\ i^4 sl2^2}{12} \\ + \frac{20isl2^3-60i^2 sl2^3+40\ i^3 sl2^3+5sl2^4-30\ i\ sl2^4+30i^2 sl2^4-6sl2^5+12\ i\ sl2^5+2sl2^6}{12} \\ \wedge g' = g5^{-sl2} \wedge v' = v^{4^{sl2}} \wedge z' = z + \frac{8t}{3}(2^{2sl2} - 1) \wedge uv' = uv^{5^{sl2}} \\ \wedge u' = u + ca\ sl2 \end{array} \right\}.$$

Fig. 3. Loop Function

equally easily with any application domain, provided we store the relevant recognizers, and incorporate the necessary domain knowledge to make inferences; it can also deal with any programming construct, any data structure, any control structure, provided we include the appropriate recognizers, and associated axiomatizations to make it possible to reason about them.

7.2. Prospects

As far as theoretical extentions of our work, we envision to explore further the relation between invariant relations, invariant functions and invariant assertions; we believe this is important not only because these concepts give complementary perspectives on the analysis of loops, but also because the study of these concepts is giving us insights on the structure of invariant assertions, which may help the research being done on invariant generation. Another theoretical extention we are considering is to lift the hypothesis that while loops terminate for all initial states; the fact of the matter is that it is difficult to separate the task of computing the function of a loop from the task of computing the domain over which the loop terminates normally. Lifting the hypothesis $dom(W) = S$ means that now the loop functions that we are seeking to compute are no longer maximal in the refinement lattice, hence lower bounds are no longer sufficient; we may have to approximate loop functions by means of lower bounds and upper bounds. This promises to be very challenging, and also very interesting.

As for practical extensions, we envision to augment the database of recognizers with domain-independent programming knowledge. Concurrently, we envision to specialize our tool to specific domains by storing a database of domain-specific recognizers, along with related domain knowledge, and experiment with the capability of the resulting tool.

7.3. Related Work

We know of few researchers who work deals with invariant relations, though they do not refer to them as such; we discuss these in section 7.4. The research literature on invariant assertions is so vast that it is impossible to do justice to all the work being carried out in this area; in section 7.5, we briefly discuss some of the work we feel is most related to ours, or has influenced ours.

7.4. Invariant Relations

Invariant relations were introduced by Mili et al. (2009) as a tool to compute or approximate loop functions. In Carette and Janicki (2007), Carrette and Janicki derive properties of numeric iterative programs by modeling the iterative program as a recurrence relation and solving the recurrence relation. Typically, the equations obtained from the recurrence relations by removing the recurrence variable are nothing but the reflexive transitive relations that we are talking about. However, whereas the method of Carrette and Janicki is applicable only to numeric programs, ours is applicable to arbitrary programs, provided we have an adequate axiomatization of their data structure.

In Podelski and Rybalchenko (2004), Podelski and Rybalchenko introduce the concept of *transition invariant*, which is a transitive superset of the loop body's function. Of special interest are transitive invariants which are well-founded; these are used to characterize termination properties or liveness properties of loops. Transitive invariants are related to the invariant relations that we introduce in Mili et al. (2009), in the sense that they are both transitive supersets of the loop body. Though they are both transitive supersets of the loop body function, transition invariants and invariant relations differ on a very crucial attribute: whereas the latter are reflexive, and are used to capture functional properties of the loop, the latter are (typically) not reflexive, and are used to capture what changes from one iteration to the next.

In Group (2010), Kovacs et. al. deploy an algorithm they have written in Mathematica to generate invariant assertions of while loops under some conditions. They proceed by formulating then resolving the recurrence relations that are defined by the loop body, much in the same way as Carette and Janicki advocate (Carette and Janicki (2007)); once they resolve the recurrence relations, they obtain a binary relation beween the initial states and the *current* state, from which they derive an invariant relation by imposing the pre-conditions on the initial state. In many instances, the binary relation they obtain prior to adding the pre-conditions is nothing but our invariant relations. Some of our recognizers stem from examples solved by Aligator.

7.5. Invariant Assertion

In (Kovacs (2007)), Kovacs combines techniques from algorithmic combinatorics and polynomial algebra to generate polynomial equations as loop invariants for a class of so-called P-solvable loops with ignored loop conditions. Implemented in the Mathematica software package Aligator, the approach was successfully tested on many examples.

Moreover, in (Henzinger et al. (2008)), Aligator is extended by implementing an approach for automatically inferring polynomial equalities and inequalities by treating loop conditions as invariants from the polynomial closed form of the loop by imposing bound constraints on the number of loop iterations. In (Kovacs and Voronko (2009)), a method for automatic generation of loop invariants for programs containing arrays is presented. Many properties of update predicates can be extracted automatically from the loop description and loop properties obtained by other methods such as a simple analysis of counters occurring in the loop, recurrence solving and quantifier elimination over loop variables.

Ernst et al. (Ernst et al. (2006)) discuss a system named *Daikon*, which is a full-featured, scalable, robust tool for dynamic detection of likely invariants. Daikon runs candidate programs and observes their behaviors at user-selected points, and reports properties that were true over the observed executions, using machine learning techniques. Because these are empirical observations, the system produces probabilistic claims of invariance.

LOOPFROG (Kroening et al. (2008)) is an automatic bug-finding tool for ANSI C programs. It is an Abstract Interpretation based static analyzer for C programs. It works on binary model files generated by the compiler goto-cc that compiles programs given in a C and C++ into GOTO-programs (i.e., control-flow graphs). LOOPFROG is based on the concept of Loop Summarization using Abstract Transformers. Because Iterative fixpoint computation is expensive, this approach avoids iterative computation of an abstract fixpoint. Instead, it builds summaries which are symbolic transformers. This tool heuristically provides invariant candidates to use as summary.

In (Gulwani et al. (2009)), a challenging aspect of computing complexity bounds is undertaken by calculating a bound on the number of iterations of a given loop. Control-flow refinement enables standard invariant generators to reason about mildly complex control-flow, which would otherwise require impractical disjunctive invariants. Progress invariants are introduced and used to compute precise procedure bounds.

In (Furia and Meyer (2010)) Furia and Meyer propose techniques for deriving invariant assertions from an analysis of the post-condition. To this effect, they explore several techniques for generalizing the post-condition for the purpose of obtaining a formula that is weak enough to be invariant, yet strong enough to logically imply the post-condition upon exiting the iteration. This approach is fairly orthogonal to ours, in that we analyze the loop without regard to its context/ its specification, and do so by inspecting the loop itself, exclusively.

In (Cousot and Cousot (1977)), invariant assertions of programs are explored in light of the fixpoint approach (lattice theory) in the static analysis of programs. The resolution of a fixpoint system of equations by Jacobi's successive approximations method is undertaken. Under continuity hypothesis any chaotic iterative method converges to the optimal solution. Therefore these optimal invariants can be used for total correctness proofs. Usually a system of inequations is used as a substitute for the system of equations. Hence the solutions to this system of inequations are approximate invariants which can only be used for proofs of partial correctness.

References

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

Carbonnell, E. R., Kapur, D., 2004. Program verification using automatic generation of invariants. In: Proceedings, International Conference on Theoretical Aspects of Computing 2004. Vol. 3407. Lecture Notes in Computer Science, Springer Verlag, pp. 325–340.

Carette, J., Janicki, R., March 2007. Computing properties of numeric iterative programs by symbolic computation. Fundamentae Informatica 80 (1-3), 125–146.

Cheatham, T. E., Townley, J. A., 1976. Symbolic evaluation of programs: A look at loop analysis. In: Proc. of ACM Symposium on Symbolic and Algebraic Computation. pp. 90–96.

Collins, R. W., Walton, G. H., Hevner, A. R., Linger, R. C., December 2005. The CERT function extraction experiment: Quantifying FX impact on software comprehension and verification. Tech. Rep. CMU/SEI-2005-TN-047, Software Engineering Institute, Carnegie Mellon University.

Colon, M. A., Sankaranarayana, S., Sipma, H. B., 2003. Linear invariant generation using non linear constraint solving. In: Proceedings, Computer Aided Verification, CAV 2003. Vol. 2725 of Lecture Notes in Computer Science. Springer Verlag, pp. 420–432.

Cousot, P., Cousot, R., 1977. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In: Proceeding Proceedings of the 1977 symposium on Artificial intelligence and programming languages. ACM.

Cousot, P., Halbwachs, N., 1978. Automatic discovery of linear restraints among variables of a program. In: Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages. pp. 84–97.

Davey, Priestley, 1990. Introduction to Lattices and Order. Cambridge University Press.

Denney, E., Fischer, B., 2006. 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.

Dijkstra, E., 1976. A Discipline of Programming. Prentice Hall.

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

Fahringer, T., Scholz, B., 2003. Advanced Symbolic Analysis for Compilers. Springer Verlag, Berlin, Germany.

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

Furia, C. A., Meyer, B., August 2010. Inferring loop invariants using postconditions. In: Dershowitz, N. (Ed.), Festschrift in honor of Yuri Gurevich's 70th birthday. Lecture Notes in Computer Science. Springer-Verlag.

Gries, D., 1981. The Science of programming. Springer Verlag.

Group, M., 2010. Demo of aligator. Tech. rep., Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland.

Gulwani, S., Jain, S., Koskinen, E., June 2009. Control-flow refinement and progress invariants for bound analysis. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. Dublin, Ireland.

Henzinger, T. A., Hottelier, T., Kovacs, L., November 2008. Valigator: Verification tool with bound and invariant generation. In: Proceedings, LPAR08: International Conferences on Logic for Programming, Artificial Intelligence and Reasoning. Doha, Qatar.

Hevner, A. R., Linger, R. C., Collins, R. W., Pleszkoch, M. G., Prowell, S. J., Walton, G. H., July 2005. The impact of function extraction technology on next generation software engineering. Tech. Rep. CMU/SEI-2005-TR-015, Software Engineering Institute.

Hoare, C., Oct. 1969. An axiomatic basis for computer programming. Communications of the ACM 12 (10), 576 – 583.

Hoder, K., Kovacs, L., Voronkov, A., 2010. Interpolation and symbolic elimination in vampire. In: Proceedings, IJCAR. pp. 188–195.

Hu, L., Harman, M., Hierons, R., Binkley, D., 2004. Loop squashing transformations for amorphous slicing. In: Proceedings, 11th Working Conference on Reverse Engineering. IEEE Computer Society.

Iosif, R., Bozga, M., Konecny, F., Vojnar, T., July 2010. Tool demosntration for the flata counter automata toolset. In: Proceedings, Workshop on Invariant Generation: WING 2010. Edimburg, UK.

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

Jilani, L. L., Louhichi, A., Mraihi, O., Desharnais, J., Mili, A., 2010. Invariant assertions, invariant relations and invariant functions. Tech. rep., NJIT, http://web.njit.edu/m̃ili/tse.pdf.

Kovacs, L., october 2007. Automated invariant generation by algebraic techniques for imperative program verification in theorema. Tech. rep., University of Linz, Linz, Austria.

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

Kovacs, L., Jebelean, T., 2005. An algorithm for automated generation of invariants for loops with conditionals. In: Petcu, D. (Ed.), 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). Department of Computer Science, West University of Timisoara, Romania, pp. 16–19.

Kovacs, L., Voronko, A., September 2009. Finding loop invariants for programs over arrays using a theorem prover. In: Proceedings, 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. Timisoara, Romania.

Kovacs, L., Voronkov, A., 2009. Finding loop invariants for programs over arrays using a theorem prover. In: Proceedings, FASE 2009. LNCS 5503, Springer Verlag, pp. 470–485.

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

Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C. M., 2008. Loop summarization using abstract transformers. In: Automated Technology for Verification and Analysis. Vol. 5311/2008 of Lecture Notes in Computer Science. pp. 111–125.

Linger, R., Mills, H., Witt, B., 1979. Structured Programming. Addison Wesley.

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

Manna, Z., 1974. A Mathematical Theory of Computation. McGraw Hill.

Mili, A., Aharon, S., Nadkarni, C., 2009. Mathematics for reasoning about loop. Science of Computer Programming, 989–1020.

Mili, A., Desharnais, J., Gagne, J. R., April 1985. Strongest invariant functions: Their use in the systematic analysis of while statements. Acta Informatica.

Mills, H., January 1975. The new math of computer programming. Communications of the ACM 18 (1).

Podelski, A., Rybalchenko, A., 2004. Transition invariants. In: Proceedings, 19th Annual Symposium on Logic in Computer Science. pp. 132–144.

Sankaranarayana, S., Sipma, H. B., Manna, Z., 2004. Non linear loop invariant generation using groebner bases. In: Proceedings, ACM SIGPLAN Principles of Programming Languages, POPL 2004. pp. 381–329.

Zuleger, F., Sinn, M., July 2010. Loopus: A tool for computing loop bounds for c programs. In: Proceedings, Workshop on Invariant Generation: WING 2010. Edimburg, UK.