Invariant Relations: An Automated Tool to Analyze Loops

Asma Louhichi
University of Tunis El Manar
Tunis, Tunisia
louhichiasma@yahoo.fr

Olfa Mraihi
ISG, Universite de Tunis
Tunis, Tunisia
olfa.mraihi@yahoo.fr

Wided Ghardallou
Universite de Tunis el Manar
Tunis, Tunisia
wided.ghardallou@gmail.com

Lamia Labed Jilani
ISG, Universite de Tunis
Tunis, Tunisia
lamia.labed@isg.rnu.tn

Khaled Bsaies
Universite de Tunis El Manar
Tunis, Tunisia
khaled.bsaies@fst.rnu.tn

Ali Mili
NJIT
Newark, NJ USA
mili@cis.njit.edu

Since their introduction more than four decades ago, invariant assertions have, justifiably, dominated the analysis of while loops, and have been the focus of sustained research interest in the seventies and eighties, and renewed interest in the last decade. In this paper, we tentatively submit an alternative concept for the analysis of while loops, explore its attributes, its applications, and its relationship to invariant assertions. Also, we discuss the design, implementation and use of a tool that analyzes while loops using this concept.

Invariant assertions, invariant functions, invariant relations,loop invariants, program analysis, program verification, while loops, loop functions

1. INTRODUCTION

In Hoare (1969), Hoare introduced the concept of an *invariant assertion* as a pivotal tool in the analysis of while loops. The study of invariant assertions, and their use in the analysis of while loops, have been the focus of active research in the seventies and eighties, and the subject of renewed interest in the last few years Fu and Bastani and Yen (2008); Carette and Janicki (2007); Berdine (et al 2007); Jebelean and Giese (2007); Ernst, et al (2006); Carbonnell and Kapur (2004); Colon and Sankaranarayana and Sipma (2003); Sankaranarayana and Sipma and Manna (2004); Kovacs and Jebelean (2004, 2005). In this paper we discuss an alternative concept to the analysis of while loops, namely the concept of *invariant relation* Mili, et al (2009). Also, we explore the attributes of invariant relations, and their relationship to invariant assertions and to loop semantics. Specifically,

- Whereas an invariant assertion is an assertion that holds after any number of iterations of the loop (including zero iterations, i.e. at the precondition), an invariant relation is a relation that holds between two states, say s and s', that are separated by an arbitrary number of iterations (including zero).
- Whereas an invariant assertion is dependent not only on the loop, but also on the context of the loop (its pre-condition and post-condition), an invariant relation is intrinsic to the loop, and does not depend on where the loop is used.
- Given an invariant relation and a precondition, we can use them to generate an invariant assertion; on the other hand, any invariant assertions can be derived from an invariant relation and a pre-condition.
- Given an invariant assertion, we can use it to generate an invariant relation; but not all invariant relations stem from invariant assertions. On the basis of this premise, we submit that invariant relations are a more general concept than invariant assertions.
- At the LIPAH laboratory, University of Tunis, we are evolving an automated tool that generates invariant relations of while loops written in C-like languages, using pre-stored patterns that represent programming knowledge and domain knowledge; we refer to these patterns as *recognizers*.
- We are also evolving and maintaining a tool, programmed in Mathematica (©Wolfram Research), that uses the invariant relations of

a loop to compute or approximate the function of the loop.

In section 2 we briefly introduce the mathematical background that is needed subsequently for our discussions. In section 3 we discuss the concept of *invariant relation* (introduced in Mili, et al (2009)) and briefly present the main results pertaining to the properties of invariant relations and their relationship to invariant assertions Jilani, et al (2011). In section 4 we discuss how invariant relations can be used to compute (or approximate) loop functions, and to compute invariant assertions; and we compare our approach to alternative approaches for generating invariant assertions. In section 5 we discuss the generation of invariant relations from a static analysis of their source code, and in section 6 we summarize and assess our main findings and identify future research directions.

2. RELATIONAL MATHEMATICS

2.1. Elements of Relations

2.1.1. Definitions and Notations

We consider a set S defined by the values of some program variables, say x, y and z; we typically denote elements of S by s, and we note that s has the form $s = \langle x, y, z\rangle$. We use the notation $x(s)$, $y(s)$, $z(s)$ to denote the x-component, y-component and z-component of s. We may sometimes use x to refer to x(s) and x' to refer to $x(s')$, when this raises no ambiguity. We refer to elements s of S as *states* of the program and to S as the *space* 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 $\emptyset$.

2.1.2. Operations on Relations

Because relations are sets, we apply the usual set theoretic operations between relations: union ($\cup$), intersection ($\cap$), and complement ($\overline{R}$). Operations on relations also include:

- The *converse* is denoted by $\widehat{R}$, and defined by $\widehat{R} = \{(s, s')|(s', s) \in R\}$.
- The *product* of relations R and R' is denoted by $R \circ R'$ (or RR'), and defined by $R \circ R' = \{(s, s')|\exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\}$. We admit without proof that $\widehat{RR'} = \widehat{R'}\widehat{R}$ and that $\widehat{\widehat{R}} = R$.
- 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 $T = \{(s, s')|t(s)\}$. We admit without proof that the pre-restriction of a relation R to predicate t can be written as $T \cap R$ or as $(I \cap T) \circ R$, and the post-restriction of relation R to predicate t can be written as $R \cap \widehat{T}$ or as $R \circ (I \cap T)$.
- The *domain* of relation R is defined as *dom*$(R) = \{s|\exists s' : (s, s') \in R\}$.
- The *range* of relation R is denoted by *rng*(R) and defined as *dom*$(\widehat{R})$.
- The n^{th} power of relation R, for natural number n, is denoted by R^n and defined by: $R^0 = I$, For $n > 0$, $R^n = R^{n-1} \circ R$.
- The *transitive closure* of relation R is defined as the relation 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 the relation denoted by R^* and defined by $R^* = R^+ \cup I$.

We apply the usual conventions with regards to operator precedence.

2.1.3. 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 relation R is said to a *vector* if and only if $RL = R$. A vector on set S has the form $C \times S$ for some subset C of S. 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 $R = \widehat{R}$. We admit without proof that the reflexive transitive closure of relation R is the smallest reflexive transitive superset of R. A relation R is said to be *inductive* if and only if it can be written as $R = \overline{A} \cup \widehat{A}$ for some vector A; we admit without proof that if A is written as $\{(s, s')|\alpha(s)\}$, then $\overline{A} \cup \widehat{A}$ can be written as $\{(s, s')|\alpha(s) \Rightarrow \alpha(s')\}$.

2.2. Refinement Ordering

Definition 1 *A relation R is said to* refine *a relation R' if and only if*

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

We denote this relation by $R \sqsupseteq R'$ or $R' \sqsubseteq R$. Given a program g on space S, we let G be the function defined as:

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

We admit that, modulo traditional definitions of total correctness Morgan (1990); Dijkstra (1976); Gries (1981); Manna (1974), the following propositions hold:

- A program g is correct with respect to a specification R if and only if $G \sqsupseteq R$.
- $R \sqsupseteq R'$ implies that any program correct with respect to R is correct with respect to R'.

Intuitively, R refines R' if and only if R represents a stronger requirement than R'.

2.3. Lattice Properties

We admit without proof that the refinement relation is a partial ordering. In Boudriga and Elloumi and Mili (1992) Boudriga et al. analyze the lattice properties of this ordering and find:

- Any two relations R and R' have a greatest lower bound, which we call the *meet*, denote by $\sqcap$, and define by: $R \sqcap R' = RL \cap R'L \cap (R \cup R')$.
- Two relations R and R' have a least upper bound if and only if they satisfy the following condition (which we call the *consistency condition*): $RL \cap R'L = (R \cap R')L$. Under this condition, their least upper bound is referred to as the *join*, is denoted by $\sqcup$, and is defined by: $R \sqcup R' = \overline{RL} \cap R' \cup \overline{R'L} \cap R \cup (R \cap R')$. Intuitively, the join of R and R', when it exists, behaves like R outside the domain of R', behaves like R' outside the domain of R, and behaves like the intersection of R and R' on the intersection of their domain. The consistency condition ensures that the domain of their intersection is identical to the intersection of their domains.
- Two relations R and R' have a least upper bound if and only if they have an upper bound.
- The lattice of refinement admits a *universal lower bound*, which is the empty relation, and admits no *universal upper bound*.
- Maximal elements of this lattice are total deterministic relations.

3. INVARIANT RELATIONS AND INVARIANT ASSERTIONS

In this section, we present relational definitions for invariant relations Mili, et al (2009) and invariant assertions Hoare (1969), to set the stage for our subsequent comparative analysis. We consider a while loop of the form w `= while t do b` on space S, which we assume to terminate for all states in S; in Mili and Aharon and Nadkarni (2009), we explain why, in theory, this hypothesis does not affect the generality of our study; in practice, this hypothesis does pose a problem, and we envision to lift it in future research. The following theorem defines the semantics of while loops.

Theorem 1 *(Mili et al, 2009 Mili and Aharon and Nadkarni (2009)) We consider a while loop* w `= while t do b` *that terminates for all the states in* S*. Then its function* W *is given by (where* T *is the vector defined by* `t` *and* B *is the function of* `b`*):*

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

A proof of this simple theorem is given in Mili and Aharon and Nadkarni (2009). To illustrate our discussions, we use a simple running example, which is the following while loop on natural variables n, f, k, such that $1 \leq k \leq n+1$:

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

We apply the formula of this theorem:

$(T \cap B)^* \cap \widehat{\overline{T}}$
= { definition }
$I \cap \overline{T} \cup (T \cap B)^+ \cap \widehat{\overline{T}}$
= { substitutions }
$\{(s,s') | k = n+1 \wedge s' = s\} \cup$
$\{(s,s') | k \neq n+1 \wedge n' = n \wedge f' = f \times k \wedge$
$k' = k+1\}^+$
$\cap \{(s,s') | k' = n'+1\}$
= { extrapolating the transitive closure }
$\{(s,s') | k = n+1 \wedge s' = s\} \cup$
$\{(s,s') | k \neq n+1 \wedge n' = n \wedge \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!} \wedge$
$k \leq k'\} \cap \{(s,s') | k' = n'+1\}$
= { simplification }
$\{(s,s') | k = n+1 \wedge s' = s\} \cup$
$\{(s,s') | k < n+1 \wedge n' = n \wedge f' = n! \times \frac{f}{(k-1)!} \wedge$
$k' = n+1\}$
= { merging }
$\{(s,s') | k \leq n+1 \wedge n' = n \wedge f' = n! \times \frac{f}{(k-1)!} \wedge$
$k' = n+1\}$.

3.1. Invariant Assertions

Traditionally Manna (1974); Hoare (1969), an invariant assertion α for the while loop w `= while t do b` with respect to a precondition/ postcondition pair (p, q) is defined as a predicate on S that satisfies the following conditions:

- $p \Rightarrow \alpha$,
- $\{\alpha \wedge t\} b \{\alpha\}$, and
- $\alpha \wedge \neg t \Rightarrow q$.

For the sake of uniformity, we recast these conditions in relational terms, representing the precondition p by the vector $P = \{(s,s') | p(s)\}$, the postcondition q by the vector $Q = \{(s,s') | q(s)\}$, and the predicate α by the vector $A = \{(s,s') | \alpha(s)\}$.

Definition 2 *Given a while statement of the form,* w `= while t do b`*, an* invariant assertion *for* w *with respect to precondition* P *and postcondition* Q *is a*

vector A on S that satisfies the following conditions: $P \subseteq A$, $A \cap T \cap B \subseteq \widehat{A}$, and $A \cap \overline{T} \subseteq Q$.

If we consider the sample loop introduced earlier, and take the precondition $f = 1 \wedge k = 1$ and the postcondition $f = n!$, then we find that $f = (k-1)!$ is an adequate invariant assertion for this program (adequate in the sense: sufficiently large to meet the first condition, sufficiently small to meet the third condition).

3.2. Invariant Relations

The concept of invariant relation was introduced by Mili et al. in Mili, et al (2009); it is defined as follows.

Definition 3 *Given a while loop of the form w =* `while t do b` *on space S, we say that a relation R is an* invariant relation *for w if and only if it is a reflexive and transitive superset of $(T \cap B)$.*

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

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

This relation is clearly reflexive and transitive; we leave it to the reader to check that it is a superset of $(T \cap B)$.

3.3. Comparative Analysis

The first question we ask is: can we derive an invariant assertion from an invariant relation?

Proposition 1 *Let w be a while loop on space S and let R be an invariant relation for w. Then $A = \widehat{R}P$ is an invariant assertion for w with respect to the precondition P and the postcondition $Q = \overline{T} \cap \widehat{R}P$.*

Proof. According to Definition 2, we must prove three conditions:

- $P \subseteq \widehat{R}P$,
- $\widehat{R}P \cap T \cap B \subseteq \widehat{\widehat{R}P}$,
- $\widehat{R}P \cap \overline{T} \subseteq \overline{T} \cap \widehat{R}P$.

The first condition stems readily from the reflexivity of R. The second condition is a result from Jilani, et al (2011). The third condition is a tautology. **qed**

As an illustration, we consider the invariant relation we had proposed earlier for the sample loop, and we let P be the following vector, representing possible initial conditions of the loop: $P = \{(s, s') | f = 1 \wedge k = 1\}$. The invariant assertion that we then obtain is the following:

A
= { Proposition 1, symmetry of R }
$\{(s, s') | \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\} \circ \{(s, s') | f = 1 \wedge k = 1\}$
= { relational product }
$\{(s, s') | \exists s' : \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!} \wedge f' = 1 \wedge k' = 1\}$
= { simplification }
$\{(s, s') | \frac{f}{(k-1)!} = 1\}$,

which is the invariant assertion we had proposed earlier.

Because invariant relations depend exclusively on the loop whereas invariant assertions depend on the context of the loop in addition to the loop (the context being defined by the precondition and the postcondition), it is difficult to compare them meaningfully. Hence we consider a context-free version of invariant assertions, which we introduce below.

Definition 4 *Given a while loop on space S of the form w =* `while t do b`*, and given a vector U on S, we say that U is an* inductive assertion *for w if and only if $U \cap T \cap B \subseteq \widehat{U}$.*

An inductive assertion is an assertion that satisfies the second condition of invariant assertions, but not the first nor the third (both of which refer to its context). The first result that we propose is a generalization of Proposition 1.

Proposition 2 *Let w be a while loop on space S and let R be an invariant relation for w and V be an arbitrary vector on S. Then $A = \widehat{R}V$ is an inductive assertion for w.*

This Proposition is due to Jilani, et al (2011), where it is proved; we content ourselves here with illustrating it. We take the invariant relation

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

and the following vectors:

$V_0 = \{(s, s') | f = 1 \wedge k = 1\}$
$V_1 = \{(s, s') | f = 1 \wedge k = 2\}$
$V_2 = \{(s, s') | f = 2 \wedge k = 3\}$
$V_3 = \{(s, s') | f = 6 \wedge k = 4\}$
$V_4 = \{(s, s') | f = (k-1)!\}$
$V_5 = \{(s, s') | \frac{f}{(k-1)!} = 120\}$.

If we let A_0, A_1, ... A_5 be the inductive assertions derived from the selected invariant relation by applying vectors V_0, V_1, ... V_5, we find

$A_0 = A_1 = A_2 = A_3 = A_4 = \{(s, s') | f = (k-1)!\}$
$A_5 = \{(s, s') | \frac{f}{(k-1)!} = 120\}$.

The next question that we address is, naturally: can we generate an invariant relation from any inductive assertion.

Proposition 3 *Let A be an inductive assertion for the while loop w; then $R = \overline{A} \cup \widehat{A}$ is an invariant relation for w.*

Proof. By set theory, the condition $T \cap B \cap A \subseteq \widehat{A}$ is equivalent to $T \cap B \subseteq (\overline{A} \cup \widehat{A})$, i.e. R is a superset of $T \cap B$. By construction, $(\overline{V} \cup \widehat{V})$ is reflexive and

transitive for any vector V. **qed**

We illustrate this proposition by means of a simple example, using the sample factorial loop. We consider the following inductive assertion,

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

and we apply the formula dictated by this Proposition:

R
= { Proposition 3 }
$\overline{A} \cup \widehat{A}$
= { substitution, simplification }
$\{(s, s')|f = (k-1)! \Rightarrow f' = (k'-1)!\}$.

Now that we know how to generate an inductive assertion from an invariant relation (Proposition 1) and an invariant relation from an inductive assertion (Proposition 3), can we infer that invariant relations and inductive assertions are equally powerful? The answer is no, because the question we need to pose instead is whether *all* invariant relations can be derived from inductive assertions, and whether *all* inductive assertions can be derived from invariant relations. The following proposition answers the second question.

Proposition 4 *Let w be a while loop on space S and let A be an inductive assertion for w. Then there exists an invariant relation R and a vector V such that $A = \widehat{R}V$.*

The proof of this proposition is given in Jilani, et al (2011), where R is taken as $\overline{A} \cup \widehat{A}$ and V is taken as A. The interest of this proposition is three-fold: First, it proves that by focusing on generating invariant relations, we cover all possible inductive assertions; Second, it provides a structure for all inductive assertions, as the inverse of an invariant relation composed with a vector; Third, it separates two components of an inductive assertion, namely $\widehat{R}$, which depends exclusively on the loop, and V, which depends on the context (initialization) of the loop.

As to the question of whether any invariant relation can be derived from an inductive assertion, we have reason to believe it is not the case. Consider that according to Proposition 3, the search for inductive assertions is amenable to the search for invariant relations of a certain kind, namely relations that are inductive. Since not all invariant relations are inductive, it is legitimate to infer that invariant relations are a more general concept than inductive assertions.

4. INVARIANT RELATIONS AND LOOP ANALYSIS

Proposition 1 shows how, given an invariant relation and a precondition to a loop, we can constructively generate a postcondition for the loop. While this is an interesting result, it is not the only application of invariant relations, nor the most useful: the following proposition, due to Mili and Aharon and Nadkarni (2009), shows that invariant relations can also be used to compute or approximate the function of a loop.

Proposition 5 *Let w = `while t do b` be a while loop on space S; let R be an invariant relation for w and let W be the function of w. Then $W \sqsupseteq R \cap \widehat{\widehat{T}}$.*

According to this proposition, we can transform an invariant relation of w into a lower bound (in the refinement ordering) of its function W. Because by hypothesis W is total and determninistic, it is maximal in the lattice of refinement, hence it is possible to compute it or approximate it using nothing but lower bounds. To this effect, we gather as many invariant relations as we can, from which we generate lower bounds of the loop's function. Then we take the join of all the lower bounds, which we test for totality and determinacy; if it is total and deterministic, then it is the function of the loop, if not it is the best approximation we can derive from the invariant relations at hand.

As an illustration, we consider the running sample loop, for which we know the following invariant relation: $R = \{(s, s')|\frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\}$; from this, we derive the following lower bound

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

This relation is total but is not deterministic, hence we seek another invariant relation for w, say $R = \{(s, s')|n = n'\}$, from which we derive the following lower bound:

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

Taking the join (in this case, the intersection) of these two lower bounds produces the following relation:

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

This is the function we had found above, by application of Theorem 1. If this loop is initialized by any segment such as: `k=1; f=1;` or `k=2; f=1;` or `k=3; f=2;` or any segment whose postcondition is $\frac{f}{(k-1)!} = 1 \wedge 1 \leq k \leq n + 1$, then the function of the initialized loop is:

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

In the remainder of this section, we consider a number of sample loops; for each loop, we generate the invariant relations derived by our tool, from which we compute the function of the loop (using Proposition 5) and an invariant assertion (using Proposition 1). For the sake of comparison, we show what other tools generate for the same loops, most notably: Aligator Kovacs and Voronkov (2009); Daikon Ernst, et al (2006), and LoopFrog Kroening, et al (2010). The code is written in C++, with slight modifications: while C++ requires all program constants to be assigned values, we merely declare them as constants; also, we may sometimes put several statements on the same line, to save space,

even though C++ compilers do not allow that. We compare the performance of our tool against the other tools at the end of the experiment, rather than after each sample program, because many of our observations apply repeatedly.

4.1. Sequential Loop Body

We consider the following loop, whose loop body consists of a sequence of assignment statements to numeric variables; note that function f is not declared explicitly, as we do not need to know its explicit expression to analyze the loop.

```
#include <iostream> #include <math.h>
using namespace std; float f (float z);
int main ()  {const int e, g, cN;
const float a, m, b, c, d;// constants
int h, i, j, k, l, n, p, xx, yy;
float x, y, z, u, v, w, q,  r, s, t;
int aa[cN]; int ab[cN];  //arrays
while (k>=7)
 {j=j+aa[i]; i=i+n; l=l+ab[n]; n=n-k;
  k=k-7; yy=xx*yy; x=a+m*x; y=a*pow(y,m);
  t=t+b*w; n=n+k+6; s=s+c*z; z=z+b;
  u=a*u+m; v=v-c; w=d*w; q=q+r;r=f(r);
  h=h+g*p; p=p/e; i=i-n;}}
```

Our tool finds the following function for this loop (where $\frac{k}{7}$ designates the integer division of k by 7 and $(k\%7)$ designates the remainder of k by 7):

$$W = \left\{ \begin{array}{c} (s,s')|d \neq 1 \wedge a \neq 0 \wedge a \neq 1 \wedge aa' = aa \wedge \\ e > 0 \wedge e \neq 1 \wedge m \geq 2 \wedge k \geq 0 \wedge k' = k\%7 \wedge \\ d > 0 \wedge r' = f^{\frac{k}{7}}(r) \wedge j' = j + \Sigma_{H=i}^{i-1+\frac{k}{7}} aa[H] \\ \wedge z' = z + b\frac{k}{7} \wedge l' = l + \Sigma_{H=1+n-\frac{k}{7}}^{n} ab[H] \wedge \\ n' = n - \frac{k}{7} \wedge ab' = ab \wedge y' = y^{m^{\frac{k}{7}}} a^{\frac{m^{\frac{k}{7}}-1}{m-1}} \wedge \\ p' = e^{-\frac{k}{7}} p \wedge h' = \frac{h - eh + e(-1+e^{-\frac{k}{7}})gp}{1-e} \wedge \\ q' = q + \Sigma_{H=1}^{n} f^H(r) - \Sigma_{H=1}^{n-\frac{k}{7}} f^H(f^{\frac{k}{7}}(r)) \wedge \\ s' = \frac{1}{2}(2s - bc\frac{k}{7} + 2cz\frac{k}{7} + bc\frac{k}{7}^2) \wedge w' = d^{\frac{k}{7}} w \\ \wedge u' = \frac{-m + a^{\frac{k}{7}}(m+(a-1)u)}{a-1} \wedge i' = i + \frac{k}{7} \wedge \\ xx' = xx \wedge yy' = xx^{\frac{k}{7}} yy \wedge v' = v - c\frac{k}{7} \wedge \\ t' = t + bw\frac{d^{\frac{k}{7}}-1}{d-1} \wedge x' = \frac{-a + m^{\frac{k}{7}}(a+(m-1)x)}{m-1} \end{array} \right\}.$$

Execution of Aligator on this program is possible only after we delete the array statements, the function calls, and the statements `y=a*y**m` and `i=i-n`; when we delete these statements, the execution yields the following invariant assertion:

$$c \times q + r \times v = 5 \times r \wedge b \times q + r = r \times z \wedge$$
$$b \times v + c \times z = 5 \times b + c \wedge$$
$$2 \times s + (v-5) \times (1+z) = b \times (v-5).$$

Execution of Daikon was only possible after we instantiated all the constants, initialized all the variables, and defined function f, which we did as follows:

```
const int e=2, g=1; cN=21;  float a=2.,
m=2., b=1., c=1., d=2.; int i=1, n=20,
j=0, k=150, p=20, h=0, l=0, xx=5, yy=2;
float x=0.,y=2.,z=1.,v=5.,w=1.,s=0.,
t=0.,u=0.,q=0.,r=1.;int aa[cN]={2,8,10,
38,15,0,3,6,23,90,57,14,46,175,23,19,0,
16,22,17,72}; int ab[cN]={12,50,4,9,6,
3,0,22,19,12,15,2,0,0,8,1,42,12,5,3,0};
float f (float z) {return z+1;};
```

Then, Daikon generates the following invariant assertion:

$$i + n = 21 \wedge 7 \times i + k = 157 \wedge 7 \times n + 10 = k \wedge$$
$$z + v = 6 \wedge w - t = 1 \wedge 2 \times w = x + 2 \wedge 2 \times t = x \wedge$$
$$z = r \wedge s = q \wedge x = u \wedge x\%a = 0 \wedge yy\%e = 0 \wedge e \in aa[].$$

As for LoopFrog, it produces the following assertion:

$$r = z \wedge u = x.$$

For the sake of comparison, we generate an invariant assertion from our invariant relation (the same invariant relation we used to generate the loop function, above), using Proposition 1; for the precondition, we take the initialization presented above, and we instantiate the constants to the values presented above. We find the following invariant assertion (represented by a vector), where ϕ represents the fractional part of a number.

$$\begin{array}{c} i \geq 1 \wedge n \leq 20 \wedge l + \Sigma_{H=1}^{n} ab[H] = 225 \wedge \\ \phi(1.443log(p)) = 0.322 \wedge j + \Sigma_{H=i}^{21} aa[H] = 656 \wedge \\ aa = [2,8,10,38,15,0,3,6,23,90,57,14,46,175, \\ 23,19,0,16,22,17,72] \wedge ab = [12,50,4,9,6,3, \\ 0,22,19,12,15,2,0,0,8,1,42,12,5,3,0] \wedge \\ xx = 5 \wedge q + \Sigma_{H=1}^{n} f^H(r) = \Sigma_{H=1}^{20} f^H(1) \wedge \\ 1.4427nlog(xx) + 1.4427log(yy) = 47.4386 \wedge \\ i + n = 21 \wedge 7i + k = 157 \wedge 2^i = x + 2 \wedge \\ 2^i = 1 + 1.4427log(y) \wedge i = z \wedge 2^i = u + 2 \wedge \\ i + v = 6 \wedge w = 0.52^i \wedge f^{21-i}(r) = f^{20}(1) \wedge \\ i + \lfloor \frac{log(p)}{log(2)} \rfloor = 5 \wedge s - 0.5(z-1)z = 0 \wedge \\ t - w = -1 \wedge h + 2p = 40. \end{array}$$

4.2. Non Trivial Calculations

We consider the following loop, which manipulates numeric variables, and includes non trivial numeric computations.

```
#include <iostream> #include <math.h>
using namespace std; // header
int fact (int n); // factorial function
int main () {const int cN, ca;
int i,j,fb,nc, np; float x,x1,x2,x3;
while (j!=cN) {j=j+i; nc=fb; fb=np+nc;
np=nc; x2=x2+pow((x-ca),i)/fact(i);
x3=x3+pow(x,i)/fact(i);
x1=x1+pow(x,j)/fact(j); i=i+1; j=j-i;}}
```

Our method produces the following function for this loop (where F is the Fibonacci function and Γ is

Euler's Gamma function):

$$W = \{(s,s')|j = cN \wedge s' = s\} \cup$$

$$\left\{ \begin{array}{c} (s,s')|j \geq cN \wedge x' = x \wedge j' = cN \wedge \\ x1' = \frac{x1\Gamma(1+cN)\Gamma(1+j) - e^x\Gamma(1+j)\Gamma(1+cN,x)}{\Gamma(1+j)\Gamma(1+cN)} \\ + \frac{e^x\Gamma(1+cN)\Gamma(1+j,x)}{\Gamma(1+j)\Gamma(1+cN)} \wedge \\ x2' = \frac{e^{-ca}(e^{ca}x2\Gamma(i)\Gamma(i+j-cN)}{\Gamma(i)\Gamma(i+j-cN)} \\ + \frac{-e^x\Gamma(i+j-cN)\Gamma(i,x-ca) + e^x\Gamma(i)\Gamma(i+j-ca,x-ca)}{\Gamma(i)\Gamma(i+j-cN)} \wedge \\ x3' = \frac{x3\Gamma(i)\Gamma(i+j-cN) - e^x\Gamma(i+j-cN)\Gamma(i,x)}{\Gamma(i)\Gamma(i+j-cN)} \\ + \frac{e^x\Gamma(i)\Gamma(i+j-cN,x)}{\Gamma(i)\Gamma(i+j-cN)} \wedge \\ fb' = np \times F(j-cN) + fb \times F(j+1-cN) \wedge \\ nc' = np \times F(j-cN-1) + fb \times F(j-cN) \wedge \\ np' = np \times F(j-cN-1) + fb \times F(j-cN) \wedge \\ i' = i + j - cN \end{array} \right\},$$

In order to generate an invariant assertion for this loop, we need to choose a vector P that represents initial conditions (as per Proposition 1). We let P be defined by the following initial conditions: $i = 1 \wedge j = 30 \wedge fb = 1 \wedge nc = 1 \wedge np = 1 \wedge x1 = x2 = x3 = 0$. This yields the following invariant assertion: $A =$

$$\left\{ \begin{array}{c} (s,s')|i \geq 1 \wedge j \leq 30 \wedge i + j = 31 \wedge x = 3 \wedge \\ np = F(i) \wedge fb = F(i+1) \wedge \\ x2 = 2.71828\frac{\Gamma(i,1)}{\Gamma(i)} - 1 \wedge x3 = 20.0855\frac{\Gamma(i,3)}{\Gamma(i)} \wedge \\ x1 = 20.0855(1 - \frac{\Gamma(j+1,3)}{\Gamma(j+1)}) \end{array} \right\}.$$

Aligator could not process this loop; when we removed all the statements that it could not parse, it produced the following assertion:

$$60 + i^2 = i + 2j.$$

Daikon did not object to any statement, and produced $i + j - 31 = 0 \wedge nc = np \wedge x = 3$. LoopFrog produced $nc = np$.

4.3. Non Trivial Control Structures

We consider the following loop, which has nested if then else statements in its loop body:

```
#include <iostream> using namespace std;
int main () {int x, z, t;  float y;
while (x!=1) {if (x%4==0)
 {x=x/4; y=y*4; z=z+2; t=t-2;}
   else {if (x%2==0) {x=x/2;
    y=y*2; z=z+1; t=t-1;}
    else {x=x-1; y=y+y/x;}}}
```

Our tool returns the following function for this loop:

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

We apply Proposition 1 to the invariant relation generated for this loop, using the precondition P defined by $x = 30 \wedge y = 2 \wedge z = 0 \wedge t = 20$, and we find the following invariant assertion (represented by a vector):

$$A = \left\{ (s,s')| \begin{array}{c} xy = 60 \wedge z + \lfloor log_2(x) \rfloor = 4 \\ \wedge t - \lfloor log_2(x) \rfloor = 16 \end{array} \right\}.$$

Aligator and Daikon both find $t + z = 20$.

4.4. Non Integer Data

We consider the following loop, which performs a fixpoint computation on real numbers.

```
#include <iostream> #include <math.h>
int main () {float x, y, z;
while (fabs(y-x)!=0)  {y=x; x=1+z/x;}}
```

Our algorithm finds the following invariant relation:

$$R = \left\{ \begin{array}{c} (x' = x \wedge y' = y \wedge z' = z) \vee \\ (s,s')|\ ((x \neq y) \wedge (\exists s" : x' = 1 + \frac{z"}{x"} \wedge \\ y' = x" \wedge z' = z")) \end{array} \right\}.$$

By generating a lower bound from this invariant relation, we find the following approximation of W:

$$W \sqsupseteq \{(s,s')|x = y \wedge s' = s\}$$

$$\cup \{(s,s')|x \neq y \wedge 1 + 4z \geq 0 \wedge x' = \frac{1 + \sqrt{1+4z}}{2} \wedge$$

$$y' = \frac{1 + \sqrt{1+4z}}{2} \wedge z' = z\}$$

$$\cup \{(s,s')|x \neq y \wedge 1 + 4z \geq 0 \wedge x' = \frac{1 - \sqrt{1+4z}}{2} \wedge$$

$$y' = \frac{1 - \sqrt{1+4z}}{2} \wedge z' = z\}.$$

Indeed, this program has two fixpoints, and it converges to one or the other depending on the initial value of x. Note that because this relation is not deterministic (though it is total), we cannot say that it equals W; we can only say that it is refined by W. Aligator and LoopFrog find no loop invariant. Daikon finds: $x \neq y \wedge x > z \wedge y > z$. The characterization of $x \neq y$ as an invariant assertion is misleading, since this is the condition of the loop (the loop condition is not an invariant assertion, since it is not true at the exit point).

4.5. Non Numeric Data

We consider the following program, which handles data of type list.

```
#include <iostream>  #include <list>
using namespace std; int main ()
{list <int> l1, l2, l3;  int x;
while (!l2.empty())
{l1.push_back(l2.front());
l3.push_front(l2.front());
x=x+l2.front(); l2.pop_front();}}
```

Our tool produces the following function for this loop:

$$W = \left\{ (s,s')| \begin{array}{c} l2' = () \wedge l1' = l1.l2 \wedge l3' = \widehat{l2}.l3 \wedge \\ x' = x + \Sigma(l2) \end{array} \right\},$$

where the dot represents list concatenation, the hat represents list inversion, and Σ represents list summation. Application of Proposition 1 with the precondition $l1 = (12,2,12) \wedge l2 = (8,10,3,6,4) \wedge l3 = (7,8,5,9) \wedge x = 1$ yields the following invariant

assertion:

$$A = \left\{ (s, s') \middle| \begin{array}{c} l1.l2 = (12, 2, 12, 8, 10, 3, 6, 4) \wedge \\ x + \Sigma(l2) = 32 \wedge \\ \widehat{l2}.l3 = (4, 6, 3, 10, 8, 7, 8, 5, 9) \end{array} \right\}.$$

Neither Aligator nor Daikon nor LoopFrog can process this program.

4.6. Comparisons

The deployment of our tool, along with competing tools, on an eclectic sample of loops, leads to make the following observations:

- All of Aligator, Daikon and LoopFrog handle only numeric variables, and often only integer variables. By contrast, our tool can handle any data type, provided the system has the relevant recognizers for it.
- Because it produces invariants of a certain form, Aligator also has restrictions on the kinds of numeric operations we can have in the loop. By contrast, our tool has no limit on the form of invariant relations that it produces; whatever we store in recognizers gets generated upon a successful match.
- Because it focuses exclusively on empirical observations, Daikon has no way to distinguish between properties that stem from the loop versus properties that stem from the data; so that changing the data on which the loop executes changes the invariant that Daikon finds, even though the loop has not changed. By contrast, our tool is focused exclusively on capturing functional properties of the loop, regardless of the value that the data takes at any particular execution.
- Because it tries to generate whatever clause it believes to hold, Daikon tends to produces many clauses that are of little value, i.e. tell us very little about the loop. For large loops, this may lead to very low precision, where only very few clauses are relevant. By contrast, our tool focuses on functional properties that enable us to compute the function of the loop —a rigorous test of relevance.
- Because they are not goal oriented (they are not generating invariants for a specific verification goal), Aligator, Daikon and LoopFrog offer no warranty as to how strong their invariant assertions are; in practice these can be arbitrarily weak/ uninformative, leading to poor recall. By contrast, our tool attempts to generate invariant relations that are sufficiently strong to compute the function of the loop; and it tests whether it has found them or not, by whether or not the corresponding lower bounds produce a total deterministic relation.
- Whenever it is applied to the same loop, our tool generates an invariant assertion that subsumes (logically implies) the invariant assertions generated by the other tools. Note that in the examples above, we find that some of the clauses generated by other tools are not generated by ours, but that is because we had to remove statements from the loop before we applied the other tools, hence we are not dealing with the same loop any more.

In fairness, we must also acknowledge the single most critical weakness of our tool: it can only handle loops for which it has relevant recognizers; hence its superior performance exhibited in the examples above carries over only as far as its database of recognizers allows. But two important qualification must be made:

- Whereas the limitations of the other tools are inherent to their respective algorithms, the limitations of our tool are circumstantial, in the sense that they are dependent on the current status of the recognizer database, and can be readily fixed by adding the missing recognizers. For example, if Daikon or Aligator or LoopFrog cannot handle a program on non numeric data types, there is typically no simple fix that can break the limitation; by contrast, if our tool is unable to handle a particular code configuration, all we have to do is add the missing recognizer and run it again, without changing the core algorithm.
- We justify the dependence of our tool on recognizers by observing that recognizers are merely the way in which we codify the programming knowledge and domain knowledge that is needed to analyze loops adequately. In the other tools, this knowledge is hard-wired into the algorithm.

5. GENERATING INVARIANT RELATIONS

While in the previous section we discussed how to use invariant relations, in this section we discuss how to generate them. The first question we need to address is: do all loops have invariant relations.

Proposition 6 *Let w = `while t do b` be a loop on space S. Then $R = (T \cap B)^*$ is an invariant relation for w.*

Proof. The reflexive transitive closure of $(T \cap B)$ is known to be the smallest reflexive transitive superset of $(T \cap B)$. Hence it is a reflexive transitive superset of $(T \cap B)$, i.e. an invariant relation for w. **qed**

This trivial proposition is useful for two reasons: first, because it assures us of the existence of invariant relations; second, because it highlights the property that smaller invariant relations are better. The following proposition gives us means to obtain small invariant relations.

Proposition 7 *Let w be a while loop on space S and let R and R' be invariant relations for w; then $R \cap R'$ is an invariant relation for w.*

Proof. The intersection of two reflexive relations is reflexive; the intersection of two transitive relations

is transitive; and the intersection of two supersets of $(B \cap T)$ is a superset of $(B \cap T)$. **qed**

Hence we can compute small invariant relations by taking the intersection of arbitrary invariant relations, which are not necessarily small. As to the question of how we generate elementary invariant relations, we first offer the following proposition.

Proposition 8 *Let w be a while loop* `while t do b` *on space S. Then $R = (I \cup T(T \cap B))$ is an invariant relation for w.*

Proof. We must prove that R is reflexive, transitive, and is a superset of $(T \cap B)$. Reflexivity is trivial, by construction. To prove transitivity, we proceed as follows:

RR
= { substitution }
$(I \cup T(T \cap B))(I \cup T(T \cap B))$
= { distributivity, idempotence }
$I \cup T(T \cap B) \cup T(T \cap B)T(T \cap B)$
$\subseteq$ { associativity, $(T \cap B)T \subseteq L$) }
$I \cup T(T \cap B) \cup TL(T \cap B)$
= { because T is a vector }
$I \cup T(T \cap B) \cup T(T \cap B)$
= { idempotence, definition }
R.

To prove that R is a superset of $(T \cap B)$, we write

$T \cap B$
= { idempotence }
$T \cap T \cap B$
= { alternative representation of pre-restriction }
$(T \cap I) \circ (T \cap B)$
$\subseteq$ { set theory }
$T(T \cap B)$
$\subseteq$ { set theory }
$I \cup T(T \cap B)$
= { substitution }
R.

qed

Intuitive interpretation of this proposition: If s' is obtained from s by an arbitrary number of iterations, then either $s' = s$ or $t(s)$ and s' is in the range of $(T \cap B)$.

As for how to generate more general invariant relations, consider that in order to find supersets of $(B \cap T)$, it helps to write it as an intersection, such as:

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

Because then, any superset of B_1 is a superset of $(B \cap T)$; any superset of $B_1 \cap B_2$ is a superset of $(B \cap T)$; any superset of $B_1 \cap B_2 \cap B_3$ is a superset of $(B \cap T)$; etc. This gives us a priceless divide-and-conquer strategy: we can derive invariant relations for an arbitrarily large loop, once the function of its loop body is written as an intersection, by looking at one term at a time, or two at a time, or three at a time, etc. In practice, our algorithm proceeds as follows:

- The source code is mapped into a notation that rewrites the function of the loop body as an intersection; when the loop body is merely a sequence of assignments, this can be done by eliminating sequential dependencies. When the loop body has a more complex control structures, we invoke a more general procedure, which we discuss subsequently.
- We deploy a pattern-matching algorithm that matches the terms of the intersection one a time, then two at a time, then three at a time against pre-stored patterns (the *recognizers*) for which we store the corresponding invariant relation pattern. Whenever a match is successful, we instantiate the invariant relation pattern to obtain an actual invariant relation. So far, we have limited ourselves to looking at no more than three terms at a time in order to control the combinatorics of the pattern-matching step; but as we consider more complex programs, we are envisioning to increase the number of terms that need to be considered.
- We take the intersection of all the invariant relations that are generated, to obtain a smaller invariant relation.

It is easy to write the function of the loop body as an intersection whenever the loop body is made up of a sequence of assignments; we do so by eliminating the functional dependencies between sequential statements, and summarizing the effect of the sequence of statements on each individual variable. When the loop body contains more complex control structures, such as nested if-then-else statements, then the outermost structure of the function of loop body is a union (not an intersection). In that case, we apply the pattern matching algorithm discussed above to each term of the union, to obtain an invariant relation as an intersection of larger invariant relations, of the form:

$R = (R_{1,1} \cap R_{1,2} \cap ... R_{1,n1})$
$\cup (R_{2,1} \cap R_{2,2} \cap ... R_{2,n2}) \cup ...$
$\cup (R_{m,1} \cap R_{m,2} \cap ... R_{m,nm})$.

This relation is a superset of the function of the loop body, and it is reflexive; but it is not transitive, as the union of transitive relations is not transitive. To derive an invariant relation from it, we deploy a routine (written in Mathematica, ©Wolfram Research), to merge the terms of this union into a single term, structured as an intersection. The key idea of the routine is to identify common supersets of the terms of the union, and take their intersection. To explain the merger routine, we consider two terms of the union, where each term is the intersection of two terms:

$$R = (R_{11} \cap R_{12}) \cup (R_{21} \cap R_{22}).$$

If we find, for example, that $(R_{21} \cap R_{22}) \subseteq R_{11}$ then we conclude that R_{11} is an invariant relation, since it is reflexive and transitive (by construction), and it is a superset of each term of the union (hence a superset of the union). If, for example, we find also that $(R_{11} \cap R_{12}) \subseteq R_{22}$ then we can infer (for the

same reasons as above) that R_{22} is an invariant relation. From which we conclude that $R_{11} \cap R_{22}$ is an invariant relation. As an illustration, consider the following simple loop:

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

As a reflexive transitive superset of the first branch (which we call B_1), our tool finds $R_1 = R_{11} \cap R_{12}$, where $R_{11} = \{(s, s')|xy = x'y'\}$ and $R_{12} = \{(s, s')|z = z'\}$. As a reflexive transitive superset of the second branch (which we call B_2), our tool finds $R_2 = R_{21} \cap R_{22}$, where $R_{21} = \{(s, s')|x = x'\}$ and $R_{22} = \{(s, s')|z + xy = z' + x'y'\}$. The relation $R = R_1 \cup R_2$ is a superset of $(T \cap B)$ (by construction); and it is reflexive (as the union of reflexive relations); but it is not necessarily transitive (as the union of transitive relations). However, we note that R_{22} is a superset of R_1 (by inspection); on the other hand, it is also a superset of R_2 (any term of an intersection is a superset of the intersection). Hence R_{22} is a superset of $R_1 \cup R_2$; because by construction $R_1 \supseteq B_1$ and $R_2 \supseteq B_2$ we infer that R_{22} is a superset of $B_1 \cup B_2$, which is $(T \cap B)$. On the other hand, because it is generated by our tool, R_{22} is by construction reflexive and transitive. As a reflexive transitive superset of $(T \cap B)$, R_{22} is an invariant relation for the while loop.

Figure 1 shows a sample of recognizers that we currently have in our database; some of them have been used in the examples shown in this paper.

6. CONCLUDING REMARKS

This paper reports on our ongoing work to analyze while loops by means of invariant relations, a concept we had introduced in Louhichi and Mraihi and Jilani and Mili (2009). The focus of this paper is on the tool that we are developing and evolving to generate invariant relations from a static analysis of its source code. We have presented the following results:

- Invariant relations can be used to compute or approximate the function of a loop.
- Invariant relations subsume invariant assertions, in the sense that any invariant assertion can be derived from an invariant relation.
- Any invariant assertion can be structured as the combination of an invariant relation, which is intrinsic to the loop, with the precondition of the loop, which reflects the loop's initialization.
- Invariant relations can be derived from an analysis of the source code of the loop, using a divide-and-conquer algorithm that enables us to handle large loops in nearly linear time.
- To the extent that it is applied to data structures and control structures that have previously been modeled in recognizers, our tool produces better/ more complete results than the tools we have compared it against.

Our tool for generating invariant relations is based on a pattern matching algorithm that matches pre-stored code patterns, the *recognizers*, against a representation of the loop, and generates corresponding invariant relations whenever a match is successful. We believe that the task of computing the function of a loop is essentially a mapping from a domain neutral notation, namely the programming language, to a domain-specific notation, namely the application domain of the program, with its attendant abstractions, notations, axiomatizations, etc. As long as we are handling only numeric data types (integers, reals, etc), then the distinction between programming notation and domain notation is moot, since numeric data types are native to all (C-like) programming languages. But as soon as we need to analyze programs that handle non native data types, we must be able to codify and integrate domain information in order for the tool to carry out a meaningful analysis. In our approach, domain knowledge is codified in the recognizers, and in the axiomatizations that we use after invariant relations are generated to compute the function of the loop.

The examples that we have shown in section 4 show our tool in a favorable light by comparison with other tools, but that is only because we chose the examples according to our current repository of recognizers. In its current status, our prototype tool includes about 50 recognizers, and our algorithm operates by syntactic matching. We envision three important extensions of it:

- Replace the current syntactic matching algorithm by a semantic matching algorithm; whereas we currently match statements of the loop with recognizer patterns token by token, we want to replace this by semantic match, which declares a match if the actual expression and the formal expression are identical, when instantiated by the same variable names. We envision to use Mathematica for this purpose.
- Replace the current repository of recognizers by a set of more general recognizers, and increase the scope of the repository in size (number of recognizers), as well as in genericity (range of code patterns that semantically match each recognizer).
- Develop domain-specific sub-repositories, that can be deployed for code dealing with specific application domains, and would then use domain-specific abstractions, notations, and axiomatizations.

7. REFERENCES

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

J.Fu, F.B. Bastani, and I.-L. Yen (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, 2008, pp.

ID	State Space	Condition	Code Pattern	Invariant Relation
1R3	int x;	x mod 2 =1	x=x-1	$\{(s,s')\|\lfloor log_2(x)\rfloor = \lfloor log_2(x')\rfloor\}$
1R4	int x	true	x=x+1	$\{(s,s')\|x \leq x'\}$
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=b*y	$\{(s,s')\|\frac{y}{b^{\frac{x}{a}}} = \frac{y'}{b^{\frac{x'}{a}}}\}$
2R17	list l; int i	!empty(l)	l=tail(l), i=i+1	$\{(s,s')\|i + size(l) = i' + size(l')\}$
2R18	anytype l; int i; function f	$l \in dom(f)$	l=f(l), i=i+1	$\{(s,s')\|f^{i'}(l) = f^i(l')\}$
2R19	list l; int i	!empty(l)	l=tail(l), i=i+head(l)	$\{(s,s')\|i + summation(l) = i' + summation(l')\}$
2R20	list l; list m	!empty(l)	l=tail(l), m=head(l).m	$\{(s,s')\|concat(rev(l),m) = concat(rev(l'),m')\}$
2R21	list l; list m	!empty(l)	l=tail(l), m=head(l).m	$\{(s,s')\|size(l) + size(m) = size(l') + size(m')\}$
3R4	int x,y,z; const int a,b;	true	x=x-a, z=z, y=y+b*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	$\{(s,s')\|z = z' \wedge a.log_2(y) + xlog_2(z) = a.log_2(y') + x'log_2(z')\}$

Figure 1: *Sample Recognizers*

333–342, IEEE Computer Society, Washington, DC, USA.

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

J.Berdine, A.Chawdhary, B.Cook, D.Distefano, and P.O'Hearn (2007) Variance analyses from invariance analyses, in *Proceedings of the 34th Annual Symposium on Principles of Programming Languages*, Nice, France, 2007, pp 211–224, ACM, New York, NY, USA.

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

M.D. Ernst, J.H. Perkins, P.J. Guo, S.McCamant, C.Pacheco, M.S. Tschantz, and C.Xiao (2006) The Daikon system for dynamic detection of likely invariants, *Science of Computer Programming* 69, 35-45.

E.R. Carbonnell and D.Kapur (2004) Program verification using automatic generation of invariants, in *Proceedings, International Conference on Theoretical Aspects of Computing* Guiyang, China, September 20-24, 2004, pp.325–340, Lecture Notes in Computer Science, Springer Verlag.

M.A. Colon, S.Sankaranarayana, and H.B. Sipma (2003) Linear invariant generation using non linear constraint solving, in *Proceedings, Computer Aided Verification, CAV 2003*, Colorado, USA, July 8-12, 2003, pp. 420–432, Lecture Notes in Computer Science,Springer Verlag.

S.Sankaranarayana, H.B. Sipma, and Z.Manna (2004) Non linear loop invariant generation using groebner bases, in *Proceedings, ACM SIGPLAN Principles of Programming Languages, POPL 2004*,Venice, Italy, January 2004, pp. 381–329 ACM,New York, NY, USA.

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

Kovacs, Laura Ildiko and Jebelean, Tudor (2005), An algorithm for automated generation of invariants for loops with conditionals, in *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)*, Romania, 2005, pp. 16–19, IEEE Computer Society,Washington, DC, USA.

A.Mili, S.Aharon, C.Nadkarni, O.Mraihi, A.Louhichi, and L.L. Jilani (2009) Reflexive transitive invariant relations: A basis for computing loop functions, *Journal of Symbolic Computation*, 45, 1114–1143.

L.L. Jilani, A.Louhichi, O.Mraihi, J.Desharnais, and A.Mili (2011) Invariant assertions, invariant relations and invariant functions, NJIT, http://web.njit.edu/mili/inv.pdf, Tech. Rep., 2010.

C.Morgan (1990) *Programming from Specifications*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

E.Dijkstra (1976) *A Discipline of Programming*Prentice Hall, .

D.Gries (1981) *The Science of programming* Springer Verlag.

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

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

A.Mili, S.Aharon, and C.Nadkarni, Mathematics for reasoning about loop (2009), *Science of Computer Programming*, 74, 989–1020, .

L.Kovacs and A.Voronkov (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, September 2009, pp 10–, IEEE Computer Society,Washington, DC, USA .

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

A.Louhichi, O.Mraihi, L.L. Jilani, and A.Mili (2009) Invariant assertions, invariant relations and invariant functions, in *Proceedings, 2nd International Workshop on Invariant Generation*, York, UK, 2009.