Reflexive transitive invariant relations: A basis for computing loop functions

Ali Mili a,1, Shir Aharon a, Chaitanya Nadkarni a, Lamia Labeled Jilani b, Asma Louhichi b, Olfa Mraihi b

a New Jersey Institute of Technology, Newark, NJ 07102-1982, USA
b Institut Superior de Gestion, Bardo 2000, Tunisia

A R T I C L E   I N F O

Article history:
Received 15 June 2008
Accepted 11 November 2008
Available online 9 March 2009

Keywords:
Function extraction
Loop functions
Invariant assertions
Invariant relations
Invariant functions
Relational calculus
Refinement calculus
Computing loop behavior

A B S T R A C T

Invariant assertions play an important role in the analysis and verification of iterative programs. In this paper, we introduce a related but distinct concept, namely that of invariant relation. While invariant assertions are useful to prove the correctness of a loop with respect to a specification (represented by a precondition/postcondition pair) in Hoare’s logic, invariant relations are useful to derive the function of the loop in Mills’ logic.

© 2009 Elsevier Ltd. All rights reserved.

1. Introduction

1.1. Invariant assertions and invariant relations

Loop invariants (also known as invariant assertions) have been the subject of extensive research since their introduction by Hoare (1969). Their primary function is to prove the correctness of a while loop with respect to a specification that takes the form of a precondition and postcondition. In this paper, we present a related but different concept, to which we refer as invariant relations, and discuss the application of this concept to the derivation of loop functions in H.D. Mills’ logic of programming as discussed in Linger et al. (1979), Mills and Dyer (1987), Mills (1975), Mills et al. (1986a),...
Mills et al. (1986b) and Basili et al. (1999). Invariant assertions and invariant relations look alike, and are often confused with each other, but they have important differences, most notably:

- Whereas invariant assertions are represented by unary predicates (referring to a current state of the program), invariant relations are binary relations that refer to two states of the program.
- Whereas invariant assertions depend on the loop and its specification, invariant relations depend exclusively on the loop.
- Whereas invariant assertions are used in Hoare’s logic to prove the correctness of an iterative program with respect to a specification (in the form of a precondition/postcondition pair), invariant relations are used in Mills’ logic to compute the function of an iterative program.

In this paper, we investigate the distinctions between these two related concepts, and discuss in particular how invariant relations can be used to derive the function of a while loop. We will revisit this distinction in the conclusion (Section 9), where we highlight it in greater detail using results from our paper.

In Section 1.2, we present a simple while loop, for which we derive an invariant assertion and an invariant relation, to give the reader some intuition for invariant relations, and for the distinction between invariant assertions and invariant relations. In Section 2 we introduce some relational mathematics which we use in Section 3 to define these concepts and in Section 4 to characterize them in relational terms. In Section 5 we introduce a theorem that characterizes the relation between invariant relations and loop functions, and discuss how this theorem can be used to compute or approximate the function of a while loop; because the results of this section are discussed elsewhere (Mili et al., 2008), we do not present any proofs, but keep the discussion at an informal/intuitive level. This section gives us an opportunity to discuss another concept that is related to invariant relations, namely invariant functions (Mili et al., 1985); we briefly discuss this concept, and show in particular how it can be used to derive a wide class of invariant relations. In Section 6 we discuss how we have deployed the results of Section 5 to design and implement a prototype that automatically computes the function of a loop, and illustrate the execution of this prototype on a sample C++ program in Section 7. We briefly discuss and compare related works in Section 8. Finally, in Section 9 we summarize our findings and outline directions of further research.

1.2. An illustrative example

To give the reader some intuition on the difference between invariant assertions and invariant relations, we present (in a Pascal-like notation) a simple example involving a loop that computes the sum of an array.

```pascal
x: xtype;
i: 1..N+1;
a: array [1..N] of xtype;
begin
  x:= 0; i:= 1;
  while (i <> N+1) do
    begin
      x:= x+a[i];
      i:= i+1
    end
end
```

For this loop, we let the precondition and postcondition be defined as:

\[ \phi(s_0, s) \equiv a = a_0 \land x = 0 \land i = 1, \]

\[ \psi(s_0, s) \equiv x = \sum_{k=1}^{N} a_0[k]. \]
An adequate invariant for this specification is:

\[ \chi(s_0, s) \equiv a = a_0 \land x = \sum_{k=1}^{i-1} a[k]. \]

According to Hoare (1969), in order to prove that the while statement is partially correct with respect to the specification \( (\phi(s_0, s), \psi(s_0, s)) \), it suffices to prove the following premises.

1. Initial condition:
   \[ \phi(s_0, s) \Rightarrow \chi(s_0, s). \]
2. Invariance (Inductive) condition:
   \[ \{ \chi(s_0, s) \land t \} B \{ \chi(s_0, s) \}. \]
3. Exit (Final) condition:
   \[ \chi(s_0, s) \land \neg t \Rightarrow \psi(s_0, s). \]

We leave it to the reader to check that these three premises hold for the specification and the invariant assertion at hand.

While invariant assertions capture a property that holds initially, and after an arbitrary number of executions of the loop body, the invariant relations that we illustrate in this section capture a reflexive transitive relation that exists between two states \( s \) and \( s' \) that are separated by an arbitrary number of iterations (i.e. \( s' \) is obtained from \( s \) by application of an arbitrary number of loop body instances, assuming the loop condition returns true whenever is tested). Whereas invariant assertions (which are unary predicates) are geared towards an induction on the trace of execution (if the invariant assertion holds for some intermediate state in the execution trace, then it holds for the next), invariant relations (which are binary relations) are geared towards an induction on the number of iterations that separate some state \( s \) from some state \( s' \) that follows it (if relation \( R \) holds between two states separated by \( n \) iterations, then it holds between two states separated by \( n + 1 \) iterations); as we shall see, invariant relations are reflexive and transitive, but are not necessarily symmetric, hence \( s \) and \( s' \) are not interchangeable. Also, whereas invariant assertions are dependent upon the loop as well as its context (in terms of initial state, precondition/ postcondition, etc), invariant relations depend exclusively on the loop, and remain unchanged regardless of the context in which the loop is embedded.

For illustration, we consider the loop that computes the array sum (above) and we consider the following binary relations.

\[ R_0 = \left\{ (s, s') | a = a' \land x + \sum_{k=i}^{N} a[k] = x' + \sum_{k=i'}^{N} a'[k] \right\}, \]
\[ R_1 = \left\{ (s, s') | x + \sum_{k=i}^{N} a[k] = x' + \sum_{k=i'}^{N} a'[k] \right\}, \]
\[ R_2 = \left\{ (s, s') | a = a' \land x + \sum_{k=i}^{N} a[k] = x' + \sum_{k=i'}^{N} a'[k] \land i \leq i' \right\}, \]
\[ R_3 = \left\{ (s, s') | x + \sum_{k=i}^{N} a[k] = x' + \sum_{k=i'}^{N} a'[k] \land i \leq i' \right\}. \]

We argue that each one of these relations characterizes a link between two states \( s \) and \( s' \) that are separated by an arbitrary number (including zero) of iterations. Notice that some of these relations are symmetric (e.g. \( R_0 \) and \( R_1 \)) and some are not (e.g. \( R_2 \) and \( R_3 \)). Symmetric relations characterize pairs of states \((s, s')\) that are an arbitrary number of iterations apart, without specifying which comes before which. Antisymmetric relations represent pairs \((s, s')\) where \( s \) is known to precede \( s' \).

We further argue that invariant relations are more general than invariant assertions, in the sense that it is possible to obtain an invariant assertion from an invariant relation \( R \) by specializing the first
argument of $R$ to be the initial state of the loop. For the sake of illustration, we let $\chi$ be the predicate defined by

$$\chi(s_0, s) \equiv (s_0, s) \in R,$$

where $s_0$ is some initial state. We find that predicate $\chi(s_0, s)$ satisfies the second premise of Hoare’s rule, i.e.

$$\{\chi(s_0, s) \land t\} B \{\chi(s_0, s)\}.$$

To this effect, we reinterpret the second premise,

$$\{\chi(s_0, s) \land t\} B \{\chi(s_0, s)\}$$

in the following terms, where $[B]$ is the function of $B$:

$$\chi(s_0, s) \land t(s) \Rightarrow \chi(s_0, [B](s)).$$

We proceed by successive implications.

$$\begin{align*}
\Rightarrow & \quad \{\text{substitution of } \chi\} \\
& (s_0, s) \in R \land t \\
\Rightarrow & \quad \{\text{substitution of } R\} \\
& a_0 = a \land \\
& x_0 + \sum_{k=0}^{N} a_0[k] = x + \sum_{k=i}^{N} a[k] \land \\
& i \neq N + 1
\end{align*}$$

$$\Rightarrow \quad \{\text{Shifting one term of the sum into } x\} \\
& a_0 = a \land \\
& x_0 + \sum_{k=0}^{N} a_0[k] = x + a[i] + \sum_{k=i+1}^{N} a[k] \land \\
& i \neq N + 1
\Rightarrow \quad \{\text{deleting unnecessary conjunct}\} \\
& a_0 = a \land \\
& x_0 + \sum_{k=0}^{N} a_0[k] = x + a[i] + \sum_{k=i+1}^{N} a[k]
\Rightarrow \quad \left\{\begin{array}{c}
\text{substitution:} \\
\left(\begin{array}{c}
a' \\
i'
\end{array}\right) = [B]\left(\begin{array}{c}
a \\
i
\end{array}\right)
\end{array}\right\}
\Rightarrow \quad \{\text{reverse substitution of } R\} \\
& a_0 = a' \land \\
& x_0 + \sum_{k=0}^{N} a_0[k] = x' + \sum_{k=i'}^{N} a[k] \land \\
\Rightarrow \quad \{\text{reverse substitution of } [B]\} \\
& (s_0, s') \in R \\
\Rightarrow \quad \{\text{reverse substitution of } R\} \\
& (s_0, [B](s)) \in R,
\end{align*}$$

which is what we wanted to prove.

The main interest of invariant relations, for our purposes, is that they enable us to compute or approximate loop functions, as we briefly illustrate using relation $R_2$: if we know that the pair $(s, s')$ made up of the initial state $s$ and final state $s'$ is in $R$ and further that $s'$ satisfies the condition $i' = N + 1$ (exit condition of the loop), then we can write

$$a = a' \land x + \sum_{k=i}^{N} a[k] = x' + \sum_{k=i'}^{N} a'[k] \land i \leq i' \land i' = N + 1.$$
After substitution and simplification, this becomes:

\[ i \leq N + 1 \land a = a' \land x' = x + \sum_{k=1}^{N} a[k] \land i' = N + 1, \]

which is the function of the while loop.

2. Mathematical background

2.1. Elements of relations

We represent the functional specification of programs by relations; without much loss of generality, we consider homogeneous relations, and we denote by \( S \) the space on which relations are defined. A relation \( R \) on set \( S \) is a subset of the Cartesian product \( S \times S \), hence it is natural to represent general relations as

\[ R = \{(s', s) \mid p(s, s')\}, \]

for some binary predicate \( p \). Typically, set \( S \) is defined by some variables, say \( x, y, z \); hence an element \( s \) of \( S \) has the structure

\[ s = (x, y, z). \]

We use the notation \( x(s), y(s), z(s) \) (resp. \( x(s'), y(s'), z(s') \)) to refer to the \( x \)-component, \( y \)-component and \( z \)-component of \( s \) (resp. \( s' \)). We may, for the sake of brevity, write \( x \) for \( x(s) \) and \( x' \) for \( x(s') \) (and do the same for other variables).

Constant relations include the universal relation, denoted by \( L \), the identity relation, denoted by \( I \), and the empty relation, denoted by \( \emptyset \). Given a predicate \( t \), we denote by \( I(t) \) the subset of the identity relation defined as follows:

\[ I(t) = \{(s, s') \mid s' = s \land t(s)\}. \]

Because relations are sets, we use the usual set theoretic operations between relations. Operations on relations also include the converse, denoted by \( \hat{R} \) or \( R \hat{\imath} \), and defined by

\[ \hat{R} = \{(s, s') \mid (s', s) \in R\}. \]

The product of relations \( R \) and \( R' \) is the relation denoted by \( R \circ R' \) (or \( RR' \)) and defined by

\[ R \circ R' = \{(s, s') \mid \exists (t, t') \in R \land (t, t') \in R'\}. \]

The pre-restriction (resp. post-restriction) of relation \( R \) to predicate \( t \) is the relation \( \{(s, s') \mid t(s) \land (s, s') \in R\} \) (resp. \( \{(s, s') \mid (s, s') \in R \land t(s')\} \)). We admit without proof that the pre-restriction of a relation \( R \) to predicate \( t \) is \( I(t) \circ R \) and the post-restriction of relation \( R \) to predicate \( t \) is \( R \circ I(t) \). The domain of relation \( R \) is defined as \( \text{dom}(R) = \{s \mid \exists s' : (s, s') \in R\} \). The range of relation \( R \) is denoted by \( \text{rng}(R) \) and defined as \( \text{dom}(\hat{R}) \). The nucleus of relation \( R \) is the relation denoted by \( \mu(R) \) and defined as \( RR \). For any \( R \), the nucleus of \( R \) is symmetric and reflexive on \( \text{dom}(R) \). We say that \( R \) is deterministic (or that it is a function) if and only if \( RR \subseteq I \) and we say that \( R \) is total if and only if \( I \subseteq RR \), or equivalently, \( RL = L \).

Given a relation \( R \) on \( S \) and an element \( s \) in \( S \), we let the image set of \( s \) by \( R \) be denoted by \( sR \) and defined by \( sR = \{s' \mid (s, s') \in R\} \). A relation \( R \) is said to be rectangular if and only if \( R = RL \). 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 \( R = \hat{R} \). We will occasionally refer to Tarski’s Identity (Tarski, 1941, 1955), which provides that for any relation \( R, LRL = L \) if and only if \( R \) is non-empty. We are interested in two special types of rectangular relations: rectangular surjective relations are called vectors and satisfy the condition \( RL = R \); rectangular total relations are called inventors (inverse of a vector) and satisfy the condition \( LR = R \). In set theoretic terms, a vector on set \( S \) has the form \( A \times S \), and an inventor has the form \( S \times A \), for some subset \( A \) of \( S \). Vector \( A \times S \) can also be written as \( I(A) \circ L \).
2.2. Refinement ordering

We define an ordering relation on relational specifications under the name refinement ordering:

**Definition 1.** A relation \( R \) is said to refine a relation \( R' \) if and only if
\[
RL \cap R'L \cap (R \cup R') = R'.
\]

In set theoretic terms, this equation means that the domain of \( R \) is a superset of (or equal to) the domain of \( R' \), and that for elements in the domain of \( R' \), the set of images by \( R \) is a subset of (or equal to) the set of images by \( R' \). This is similar to, but different from, refining a pre/postcondition specification by weakening its precondition and/or strengthening its postcondition (Gries, 1981; Morgan, 1998). We abbreviate this property by \( R \sqsubseteq R' \) or \( R' \sqsupseteq R \). We admit that, modulo traditional definitions of total correctness (Dijkstra, 1976; Gries, 1981; Manna, 1974), the following propositions hold.

- A program \( P \) is correct with respect to a specification \( R \) if and only if \([P] \sqsupseteq R \), where \([P]\) is the function defined by \( P \) (we may, by abuse of notation use \( P \) to refer to \([P]\) when the context allows).
- \( R \sqsubseteq R' \) if and only if 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 \( R' \). We admit without proof that any relation \( R \) can be refined by a deterministic relation, i.e. a function.

2.3. Refinement lattice

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

- Any two relations \( R \) and \( R' \) have a greatest lower bound, which we refer to as the meet, denoted by \( \sqcap \), and defined 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:
\[
RL \cap R'L = (R \cap R')L.
\]
Under this condition, their least upper bound is referred to as the join, denoted by \( \sqcup \), and defined by:
\[
R \sqcup R' = RL \cap R' \cup R'L \cap R \cup (R \cap R').
\]
- Two relations \( R \) and \( R' \) have a least upper bound if and only if they have an upper bound; this property holds in general for lattices, but because the refinement ordering is not a lattice (since the existence of the join is conditional), it bears checking for this ordering specifically.
- The lattice of refinement admits a universal lower bound, which is the empty relation.
- The lattice of refinement admits no universal upper bound.
- Maximal elements of this lattice are total deterministic relations.

See Fig. 1. We conclude this section with a simple proposition, which is due to Boudriga et al. (1992), and can easily be established from the relevant definitions. This proposition gives simple expressions for the join and meet of two relations, under some special conditions.

**Proposition 1.** If \( R \) and \( R' \) have the same domain (i.e. \( RL = R'L \)), then
\[
R \sqcup R' = R \cap R' \land R \sqcap R' = R \cup R'.
\]
If \( R \) and \( R' \) have disjoint domains (i.e. \( RL \cap R'L = \phi \)) then
\[
R \sqcup R' = R \cup R' \land R \sqcap R' = R \cap R'.
\]
3. Relational definitions

Because invariant relations involve two states, it is natural to represent them with binary relations; for the sake of uniformity, we will also use binary relations (specifically, vectors) to represent invariant assertions. In this paper we assume implicitly that loops terminate for all elements of their space, i.e. they define total functions. The following proposition, due to Miliet al. (2008), provides that this assumption does not involve any loss of generality.

**Proposition 2.** We consider a while loop \( w \) on space \( S \). We let \( s_0 \) be an element of \( \text{dom}([w]) \) and we denote by \( s_1, s_2, \ldots, s_n \) be the sequence of states obtained by the successive executions of the loop body, where \( s_n \) is the state obtained when the loop terminates. Then, for all \( i, 0 \leq i \leq n, s_i \in \text{dom}([w]) \).

**Proof.** State \( s_0 \) is in \( \text{dom}([w]) \), by hypothesis. State \( s_n \) is also in \( \text{dom}([w]) \) since execution of the while loop on \( s_n \) terminates (instantly, in fact, since \( s_n \) does not satisfy the loop condition, \( t \)). For all \( i, 1 \leq i \leq n - 1 \), execution of the loop on state \( s_i \) terminates; for if it did not, neither would execution of the loop on \( s_0 \). □

Since initial states, intermediate states, and final states are all in \( \text{dom}([w]) \), we can let \( S \) be \( \text{dom}([w]) \) without loss of generality, as then all the states of interest are within \( S \). This choice of state space makes the while statement’s function total by construction. In the following, we implicitly assume this condition throughout, unless otherwise specified. What this means, in practice, is that whenever we are given a while loop on some space \( S’ \), we let \( S \) be the subset of \( S’ \) that represents the domain of \( [w] \), and we discuss the loop extraction of \( w \) on space \( S \). By making this assumption, we are not presuming that the derivation of the domain of \( [w] \) is easy in practice; it is often very difficult, and we are separately exploring means to derive it. But our subsequent discussion holds only for cases where \( [w] \) is total, or, equivalently, where the space is restricted to the domain of \( [w] \).

3.1. Invariant assertions

We consider a while loop of the form

\[
\text{w = while } t \text{ do } B
\]
on space \( S \), and we let \( \chi() \) be an invariant assertion of \( w \). The invariance condition of \( \chi() \) can be written as

\[
\{ \chi() \wedge t \} B \{ \chi() \}.
\]
We interpret this condition in logical terms as:
\[ \forall s, s' : \chi(s) \land t(s) \land (s, s') \in [B] \Rightarrow \chi(s'). \]

If we let \( V \) be the vector defined by
\[ V = \{(s, s')|\chi(s)\} \]
and \( T \) be the vector defined by
\[ T = \{(s, s')|t(s)\}, \]
then we can rewrite this condition as:
\[ \forall s, s' : (s, s') \in V \cap T \cap [B] \Rightarrow (s, s') \in \hat{V}. \]

Given that \( s \) and \( s' \) are arbitrary, this can be written algebraically as,
\[ V \cap T \cap [B] \subseteq \hat{V}. \]

Whence the following definition,

**Definition 2.** Given a while statement of the form, \( w = \text{while } t \text{ do } B \), a invariant assertion is defined as a vector \( V \) on \( S \) that satisfies the following condition:
\[ V \cap T \cap [B] \subseteq \hat{V}, \]
where \( T \) is the vector defined by predicate \( t \).

### 3.2. Invariant relations

Because invariant relations involve two states, a past state and a current state, it is natural to represent them with binary relations.

**Definition 3.** Given a while loop of the form \( w = \text{while } t \text{ do } B \) on some space \( S \), and given a relation \( R \) on \( S \), we say that \( R \) is an invariant relation for \( w \) if and only if \( R \) is reflexive, transitive, and satisfies the following conditions (where \( T \) is the vector defined by predicate \( t \) and \( \overline{T} \) is the complement of \( T \)):

- The Invariance Condition:
  \[ T \cap [B] \subseteq R. \]
- The Convergence condition:
  \[ R \circ \overline{T} = L. \]

To highlight its important properties, we may sometimes refer to an invariant relation as a reflexive transitive invariant relation; these two terms refer to the same concept. Note that unlike the definition of invariant assertions (Definition 2) the definition of invariant relations (Definition 3) is not recursive: the term \( R \) appears on one side only of each equation. What makes it recursive/inductive, nevertheless, is the fact that \( R \) is reflexive and transitive; reflexivity serves the basis of induction, and transitivity serves the inductive step. The following proposition elucidates one aspect of the relation between invariant assertions and invariant relations: given an invariant relation, we can derive an invariant assertion from it.

**Proposition 3.** Given a while statement \( w = \text{while } t \text{ do } B \) on space \( S \), and given an invariant relation \( R \) of \( w \). If we let \( \chi(s_0, s) \) be the predicate defined by:
\[ \chi(s_0, s) \equiv (s_0, s) \in R \]
for some state \( s_0 \) of \( S \), then \( \chi \) satisfies the following Hoare formula:
\[ \{\chi(s_0, s) \land t(s)\} B \{\chi(s_0, s)\}. \]
**Proof.** By hypothesis, we have

\[ T \cap [B] \subseteq R. \]

Left multiplying by \( R \) on both sides, we obtain

\[ R \circ (T \cap [B]) \subseteq R \circ R. \]

Because \( R \) is transitive, we get:

\[ R \circ (T \cap [B]) \subseteq R. \]

By set theory, we get:

\[ (s_0, s) \in (R \circ (T \cap [B])) \Rightarrow (s_0, s) \in R. \]

By definition of the relational product, we get:

\[ (s_0, s') \in R \land t(s') \land (s', s) \in [B] \Rightarrow (s_0, s) \in R. \]

Because \([B]\) is a function, we can write this as:

\[ (s_0, s') \in R \land t(s') \Rightarrow (s_0, [B](s')) \in R. \]

Using the definition of predicate \( \chi(s_0, s) \), we find

\[ \chi(s_0, s') \land t(s') \Rightarrow \chi(s_0, [B](s')). \]

Interpreting this in the Hoare notation, and replacing the mute variable \( s' \) by the equally mute \( s \), we find

\[ \{ \chi(s_0, s) \land t(s) \} [B] \{ \chi(s_0, s) \}. \quad \Box \]

In other words, the invariance condition (in the sense of **Definition 3**) of invariant relations yields the invariance condition (in the sense of Hoare’s method) of invariant assertions. As for the convergence condition of **Definition 3**, it can be interpreted as follows: for any state \( s \) in space \( S \), there exists a state \( s' \) such that \((s, s')\) is an element of \( R \) and \( s' \) satisfies \( \neg t \). In other words, \( R \) links any state \( s \) into a state \( s' \) that satisfies the termination condition \( (\neg t) \).

### 4. Relational characterizations

Whereas in the previous section we presented definitions of invariant assertions and invariant relations, in this section we attempt to give general characterizations of these invariants. Of course, a given loop can have many invariant relations and the same loop may have many invariant assertions, even for the same pre-specification/post-specification pair. In order to make the contrast between invariant assertions and invariant relations palatable, we aim to compare the strongest (in a sense to be defined) invariant assertions with the strongest (in a sense to be defined) invariant relations.

#### 4.1. Invariant assertions

We consider a while loop on space \( S \), of the form

\[
\text{w = while } t \text{ do } B
\]

and we are interested in a strongest loop invariant for this loop. As we have discussed above, a loop invariant does not depend exclusively on the loop, but on the loop’s context. Hence we embed this loop in a larger program structure, which we use to derive an invariant assertion. Specifically, we consider the following program structure, which we annotate by intermediate assertions and invariant assertions:
\begin{verbatim}
A.Milietal./JournalofSymbolicComputation45(2010)1114–1143
1123
end.

f =
begin
{s=s0}
init;
{s=[init](s0)}
while t do
\{F(s)=F(s0) \&\& s in dom(W inter F)\}
{B;}
{s=F(s0)}
end.

A theorem by Mili et al. (1987), which is based on the earlier findings by Morris and Wegbreit (1977), Mills (1975) and Basu and Misra (1975) provides that program \( f \) computes some total function \( F \) on \( S \) if:

(1) The specification \( Y \) defined by

\[ Y = \widehat{FF} \cap \widehat{L(F \cap W)} \]

(where \( W \) is the function of the while loop) is total.

(2) Segment init is correct with respect to specification

\[ Y = \widehat{FF} \cap \widehat{L(F \cap W)}. \]

(3) The following predicate is an invariant assertion:

\[ \chi(s_0, s) \equiv (s_0, s) \in \widehat{FF} \cap \widehat{L(F \cap W)}. \]

To illustrate this result, we consider again the program of array sum that we used in Section 1.2. The space of our program is defined by the following variable declarations:

\begin{verbatim}
  x: xtype;
i: 1..N+1;
a: array [1..N] of xtype;
\end{verbatim}

As for the program structure, it is defined as follows:

\begin{verbatim}
f =
begin
x:= 0; i:= 1;
while (i <> N+1) do
begin
  x:= x+a[i];
i:= i+1
end
end
\end{verbatim}

The function of program \( f \), which we denote by \( F \), is given by the following formula:

\[ F(x_i) = \left( \sum_{k=1}^{N} a[k] \right) \div \left( N + 1 \right) \div a. \]

As for the function of the loop, it is defined by

\[ W(x_i) = \left( x + \sum_{k=i}^{N} a[k] \right) \div \left( N + 1 \right) \div a. \]
From these definitions, we derive the specification $Y$ according to the formula provided above:

$$Y = \{ \text{proposed formula} \}$$

$$F \hat{\cap} \cap L((F \cap W))$$

$$= \{ \text{substituting the first term} \}$$

$$\{(s, s')|F(s) = F(s') \cap L((F \cap W)) \}$$

$$= \{ \text{substitution, using the formula of } F \}$$

$$\left\{ (s, s')\left| \sum_{k=1}^{N} a[k] = \sum_{k=1}^{N} a'[k] \land \right. \right.$$

$$N + 1 = N + 1 \land a = a' \}$$

$$\cap L((F \cap W))$$

$$= \{ \text{logical simplifications} \}$$

$$\{(s, s')|a = a' \cap L((F \cap W)) \}.$$  

To continue this construction, we need to compute the intersection of $F$ and $W$. We write,

$$F \cap W$$

$$= \{ \text{substitutions} \}$$

$$\left\{ (s, s')|x' = \sum_{k=1}^{N} a[k] \land i' = N + 1 \land a' = a \land \right.$$  

$$x' = x + \sum_{k=1}^{N} a[k] \land i' = N + 1 \land a' = a \}$$

$$= \{ \text{logical simplifications} \}$$

$$\left\{ (s, s')\left| \sum_{k=1}^{N} a[k] = x + \sum_{k=1}^{N} a[k] \land \right. \right.$$  

$$x' = \sum_{k=1}^{N} a[k] \land i' = N + 1 \land a' = a \}$$

$$= \{ \text{arithmetic simplification} \}$$

$$\left\{ (s, s')|x = \sum_{k=1}^{i-1} a[k] \land \right.$$  

$$x' = \sum_{k=1}^{N} a[k] \land i' = N + 1 \land a' = a \}.$$  

From this we infer

$$(F \cap W)L$$

$$= \{ \text{domain of } (F \cap W) \}$$

$$\left\{ (s, s')|x = \sum_{k=1}^{i-1} a[k] \right\}.$$  

Whence the term $L(F \cap W)$, which is the inverse of $(F \cap W)L$, can be written as

$$\left\{ (s, s')|x' = \sum_{k=1}^{i'-1} a'[k] \right\}.$$
Returning to specification $Y$, we find:

$$Y = \left\{ (s, s') | a' = a \land x' = \sum_{k=1}^{i-1} a'[k] \right\}.$$  

This specification is total; also, the initialization segment of the program of interest is correct with respect to $Y$, since by setting $x$ to zero and setting $i$ to 1 it makes the following equation hold:

$$x = \sum_{k=1}^{i-1} a[k].$$

A (strongest) invariant assertion (strong enough to capture all the functional details of $F$) for this while loop is then

$$(s_0, s) \in Y,$$

which we interpret as

$$a = a_0 \land x = \sum_{k=1}^{i-1} a[k],$$

which is the same invariant assertion that we have proposed in Section 1.2, only there we proposed it intuitively, whereas here we compute it from our formula.

### 4.2. Invariant relations

Whereas invariant assertions depend on the loop as well as its context, invariant relations depend solely on the while loop. The following proposition provides a strongest invariant relation for a while loop.

**Proposition 4.** Given a while loop $w = \text{while } t \text{ do } B$ on some space $S$, we let $W$ be the function of $w$. Then, the relation $R = W\hat{W}$ is a reflexive transitive invariant relation for $w$.

**Proof.** We verify in turn all four conditions for invariant relations.

- **Reflexivity.** Generally, $W\hat{W}$ is a superset of $I(\text{dom}(W))$. Because $W$ is total, $I(\text{dom}(W)) = I$.
- **Transitivity.** We must prove that $RR \subseteq R$. We write

  \[
  RR = (W\hat{W})(W\hat{W}) = W\hat{W}(W\hat{W}) = W(I)\hat{W} = W\hat{W} = R.\]

- **Invariance Condition.** A theorem by Mills (1975) provides that if $W$ is the function of the loop, then the following condition holds:

  $$t(s) \Rightarrow W(s) = W([B](s)).$$
We analyze this premise as follows:

\[ t(s) \Rightarrow W(s) = W([B](s)) \]

\[ \Rightarrow \quad \text{(rewriting)} \]

\[ t(s) \land s' = [B](s) \Rightarrow W(s) = W(s') \]

\[ \Leftrightarrow \quad \text{(Relational rewriting)} \]

\[ (s, s') \in T \cap [B] \Rightarrow (s, s') \in W \hat{\circ} W \]

\[ \Rightarrow \quad \text{(since s and s' are arbitrary)} \]

\[ T \cap [B] \subseteq W \hat{\circ} W \]

- **Convergence condition.** For all \( s \) such that \( \neg t(s) \), we know from Mills (1975) that \( W(s) = s \). We infer from this that \( I(\neg t) \subseteq W \). We use this corollary in the following proof. We write,

\[
W \hat{\circ} W \circ \overline{T} \\
\supseteq \quad \text{(corollary above)} \\
W \circ I(\neg t) \circ \overline{T} \\
= \quad \text{(rewriting the vector \( \overline{T} \))} \\
W \circ I(\neg t) \circ I(\neg t) \circ L \\
= \quad \text{(idempotence)} \\
W \circ I(\neg t) \circ L \\
= \quad \text{(post-restricting a relation to its range)} \\
W \circ L \\
= \quad \text{(W is total)} \\
L. \quad \square
\]

Proposition 4 shows that \( W \hat{\circ} W \) is an invariant relation, but does not show that it is a strongest invariant relation, not to mention that we did not even define what it means to be strongest. The next section will elucidate both questions.

5. Invariant relations and loop functions

5.1. Theorem of invariant relations

The interest of invariant relations is reflected in the following theorem, due to Mili et al. (2008) (where the interested reader is referred for a proof).

**Theorem 4.** We consider a while loop on space \( S \) of the form \( w = \text{while } t \text{ do } B \). If \( R \) is an invariant relation for \( w \), then the loop function \( W \) refines the following expression:

\[ R \cap \hat{T} \]

where \( T \) is the vector defined by predicate \( t \).

In other words, if \( R \) is an invariant relation for \( w \), then we can infer

\[ W \supseteq R \cap \hat{T} \].

This theorem maps each invariant relation of the loop into a lower bound (in the refinement ordering) of the loop function.

Interpretation of this theorem: The function of the loop is actually given by the following expression:

\[ W = (T \cap [B])^* \cap \hat{T}, \]
where \((T \cap [B])^*\) is the reflexive transitive closure of \((T \cap [B])\), i.e. the smallest reflexive transitive relation that is a superset of \([B]\). Of course, in practice, it is very difficult in general to derive the transitive closure of an arbitrary function. What this theorem does is to strike a bargain with us:

- Rather than ask us to derive the smallest superset of \((T \cap [B])\) that is reflexive and transitive, it asks us for an invariant relation of \(w\), which is an arbitrarily large superset of \((T \cap [B])\) that is reflexive and transitive.
- On the other hand, rather than provide us with the exact function of the loop, it provides us with a lower bound (in the refinement ordering), i.e. an approximation, of the loop function.

This theorem provides a basis for a divide-and-conquer approach to the derivation of loop functions. We can use this theorem to derive a set of lower bounds, which we then combine to derive or approximate the loop function. Questions that this approach raises include:

- How do we divide a complex loop body into simpler components?
- How do we derive lower bounds from individual components?
- How do we combine lower bounds?
- How do we know we have enough lower bounds?

We discuss these questions in turn, below.

5.2. A divide-and-conquer discipline

Theorem 4 provides a way to derive a lower bound (in the refinement ordering) of the function of a loop for each invariant relation we can derive for the loop. As we recall, an invariant relation of a loop is a reflexive transitive relation that is a superset of the loop body’s function. The loop body is written in some programming language (C, C++, Java) whose statements include sequence (;), conditional statements (if-then), alternation statements (if-then-else, case, switch, etc), iteration statements (while, for-do, repeat until, etc). The relational semantics of these statements maps sequence statements onto relational product, conditional and alternation statements onto unions, and iteration statements onto reflexive transitive closures. If we apply our loop analysis algorithm inside out, by analyzing inner loops before outer loops, then we can assume that whenever we consider a loop, all the loops within its body have been replaced by their function. Hence the only two constructs we need to worry about are relational product and union; we consider them in turn, below.

5.2.1. Products

It turns out that product and union are both inadequate structures if we are interested in deriving supersets: indeed, in order to find a superset of a product of relations, we must analyze each factor of the product; likewise, in order to find a superset of a union of relations, we must consider each term of the union. A much more interesting structure for the loop body function is an intersection. Indeed, if the loop body function is structured as an intersection, say

\[ [B] = B_0 \cap B_1 \cap B_2 \cap \cdots \cap B_n, \]

then any superset of \(B_0\) is a superset of \([B]\), any superset of \(B_0 \cap B_1\) is a superset of \([B]\), any superset of \(B_0 \cap B_1 \cap B_2\) is a superset of \([B]\), etc. Once the loop body function is structured as an intersection, we can find supersets of it (hence invariant relations of the loop) by considering one term at a time, or two terms at a time, or three terms at a time, etc. This is a valuable separation of concerns’ opportunity.

To transform a sequential relational expression, of the form

\[ [B] = B_0 \circ B_1 \circ B_2 \circ \cdots \circ B_n, \]

into an intersection, we use the notation known as (conditional) concurrent assignments, which substitutes concurrent assignments for sequential assignments by eliminating all the sequential dependencies (Collins et al., 2005; Linger et al., 2007). For example, the sequence \(\{x=x+1; \ y=2*x;\}\) (notice the semi-colon separators) is transformed into \(\{x=x+1, \ y=2*x+2,\}\) (notice the comma separators).
5.2.2. Unions

As for handling union operators (which stem from alternative statements or sometimes from iterative statements), we have the following theorem (due to Miliet al. (2008)), which we present without proof.

**Theorem 5.** We consider a while statement on space $S$ that terminates for all $s$ in $S$ and has the form

$$\text{while } t \text{ do } B$$

where $[B] = P \cup Q$, and we let $W$ be the function of this while statement. If $R$ and $R'$ are reflexive transitive relations such that

$P \subseteq R,
QR \subseteq R'$

and

$R \circ R' \circ I(\neg t) \circ L = L$

then

$W \sqsupseteq R \circ R' \circ I(\neg t)$

i.e. $T = R \circ R' \circ I(\neg t)$ is a lower bound for $W$.

The interest of this theorem is that we can find a lower bound for the function of the loop by analyzing $P$ and $Q$ separately rather than simultaneously. This theorem is counter intuitive, in the sense that it maps the function of the loop body (which is a union of two terms, $P$ and $Q$) onto the compositions of relations ($R$ and $R'$) in the lower bound. It stems in fact from a relational identity which formulates the transitive closure of a union in terms of composition of transitive closures:

$$(p \cup q)^* = p^*(q \circ p^*)^*.$$  

To gain some intuition about this identity, consider that the left-hand side is the union of terms formed by interleaving instances of $p$ and $q$ in an arbitrary order. An example of such a term could be, for example:

$\text{ppqpqpppqppqppqpqqqqqq}$.

The right-hand side merely parses this sequence into: first a set of instances of $p$; then a set of sequences which start with $q$ and continue with an arbitrary number of $p$’s. We parse the sequence above as follows:

$$(\text{ppp})(q(p))(q(pp))(q())(q())(q(pp))(q(p))(q(pp))(q())(q())(q()).$$

The analysis of arbitrary structures of the loop body is currently under investigation, and is beyond the scope of this paper. The prototype that we are currently working with (subject of Section 6) handles loop bodies whose function is structured as an intersection, without union operators in the mix.

5.3. Generating lower bounds

Given a loop body whose function is written as an intersection, say

$$[B] = B_0 \cap B_1 \cap B_2 \cdots \cap B_n$$

the foregoing divide-and-conquer discipline provides that we can analyze it by considering any number of (including very few, e.g. one, two or three) terms at a time. But we still need to figure out how to derive a reflexive transitive superset of any term or combination of terms. The concept of invariant function, which we introduce below, helps in this critical task.

**Definition 6** (Miliet al., 1985). Let $w$ be the while statement $\text{while } t \text{ do } B$ on some space $S$, and let $\Phi$ be a total function on $S$. We say that $\Phi$ is an invariant function for $w$ if and only if

$I(t) \circ [B] \circ \Phi = I(t) \circ \Phi.$
Invariant functions are not created equal. In the same way as we order invariant assertions by logical implication and invariant relations by refinement, we also order invariant functions, by injectivity. For example, the following functions
\[
    \begin{align*}
        \Phi_0(x) &= x \mod 6, \\
        \Phi_1(x) &= x \mod 3, \\
        \Phi_2(x) &= x \mod 2, \\
        \Phi_3(x) &= x \mod 1 (= 0)
    \end{align*}
\]
are all invariant functions of the following while loop:
\[
    \text{while } x < N \text{ do } \{x = x + 6;\}.
\]
But \( \Phi_0 \) is more injective than \( \Phi_1, \Phi_2, \) and \( \Phi_3; \) \( \Phi_1 \) and \( \Phi_2 \) are not comparable; and all are more-injective than \( \Phi_3 \), which is a constant function. We define (relative) injectivity as follows.

**Definition 7 (Relative Injectivity).** Given two total functions \( F \) and \( F' \) on space \( S \), we say that \( F \) is more-injective than \( F' \) if and only if
\[
    F \hat{\circ} F \subseteq F' \hat{\circ} F'.
\]
A function is injective if it is more-injective than the identity relation. The following Proposition shows why invariant functions are useful/interesting for our purposes.

**Proposition 5.** If \( \Phi \) is an invariant function for the while loop \( w = \text{while } t \text{ do } B \), and further satisfies the condition
\[
    \Phi \circ \hat{\circ} \Phi \circ I(\neg t) \circ L = L
\]
then \( R = \Phi \hat{\circ} \Phi \) is an invariant relation for \( w \).

This proposition suggests that whenever we have an invariant function \( \Phi \), we can use it to derive an invariant relation \( R = \Phi \hat{\circ} \Phi \). Of course, this is helpful only if is easier to derive an invariant function than to derive an invariant relation. In Mil et al. (1985) have presented a number of formulas that map specific code patterns onto invariant functions.

This proposition provides that given an invariant function \( \Phi \), we can derive an invariant relation \( R \) as the nucleus of \( \Phi \); this begs the question of whether any invariant relation is the nucleus of an invariant function. We can readily answer no to this question, in light of the following counter-example:
\[
    \text{while } i < N \{i = i + 1;\}
\]
Relation \( R = \{(s, s')| i \leq i'\} \) is an invariant relation of this while loop, because it is reflexive and transitive, and is a superset of the function of the loop body (the condition \( R \circ I(t) \circ L = L \) is also satisfied, as the reader may check); yet, it is not the nucleus of any function, because a nucleus is necessarily symmetric. More generally, the invariant relation derived as a nucleus of an invariant function is not only reflexive and transitive, it is also symmetric. Hence not all invariant relations are nuclei of invariant functions, but symmetric invariant relations might be. The proposition below provides that they are.

**Proposition 6.** Let \( R \) be an invariant relation of a while statement. If \( R \) is symmetric, then there exists an invariant function \( \Phi \) of the while statement such that \( R \) is the nucleus of \( \Phi \).

**Proof.** Because it is symmetric, in addition to being reflexive and transitive, the invariant relation \( R \) is an equivalence relation on \( S \). We let \( \Phi \) be the function that to each element of \( S \) associates its equivalence class modulo \( R \) (strictly speaking this is not a function from \( S \) to \( S \), but we can assume that each equivalence class is represented by an element of \( S \)). As defined, function \( \Phi \) satisfies the condition \( \Phi \hat{\circ} \Phi = R \). Function \( \Phi \) is clearly total, since \( R \) is an equivalence relation. We must now verify the condition
\[
    I(t) \circ [B] \circ \Phi = I(t) \circ \Phi.
\]
In general, to prove that two relations $R$ and $R'$ (or two sets) are equal, we must prove that each is included in the other. But to prove that two functions $F$ and $F'$ are equal, it suffices to prove $F \subseteq F'$ and $F'L \subseteq FL$. We consider the invariance condition of $R$,

$$T \cap [B] \subseteq R$$

and the identity $T \cap [B] \subseteq T$. Taking their intersection on both sides, we obtain

$$T \cap [B] \subseteq T \cap R,$$

which we rewrite as

$$I(t) \circ [B] \subseteq I(t) \circ R.$$

Multiplying on the right by $\Phi$, we find

$$I(t) \circ [B] \circ \Phi \subseteq I(t) \circ R \circ \Phi.$$

Substituting $R$ by $\Phi \circ \hat{\Phi}$, we find

$$I(t) \circ [B] \circ \Phi \subseteq I(t) \circ \Phi \circ \hat{\Phi} \circ \Phi.$$

Because $\Phi$ is a function, we can write $\hat{\Phi} \circ \Phi \subseteq I$, from which we infer:

$$I(t) \circ [B] \circ \Phi \subseteq I(t) \circ \Phi \circ \hat{\Phi} \circ \Phi \subseteq I(t) \circ \Phi.$$

By transitivity, we find

$$I(t) \circ [B] \circ \Phi \subseteq I(t) \circ \Phi.$$

Now we consider the reverse inequality, in terms of domains. To this effect, we submit a lemma, to the effect that $I(t) \circ [B] \circ L = I(t) \circ L$. The left-hand side is clearly a subset of the right-hand side. If the right-hand side were not a subset of the left-hand side, there would exist a state $s$ such that $t(s)$ and $s \not\in \dom([B])$. Execution of the loop on such a state would not terminate normally, hence $s$ would not be in the domain of $w$, which contradicts the hypothesis $S = \dom(w)$. We write

$$I(t) \circ \Phi \circ L = I(t) \circ L \quad \text{[By definition, $\Phi$ is total]}$$

$$I(t) \circ [B] \circ L = I(t) \circ [B] \circ L \quad \text{[By Lemma above]}$$

This proposition provides that all symmetric invariant relations are nuclei of invariant functions. In addition to their practical importance (as means to derive invariant relations), invariant functions also have a conceptual interest, which is highlighted by the following proposition.

**Proposition 7.** The function of the loop is an invariant function of the loop.

The proof of this theorem is straightforward (Mili et al., 1985); it stems readily from Mills’ theorem (Mills, 1975; Basu and Misra, 1975). Being an invariant function, the function of the loop can itself be used to generate an invariant relation, from which we can generate a lower bound for $W$; not surprisingly, the lower bound generated by the loop function is special, as the proposition below provides.

**Proposition 8.** We consider a while loop $w = \text{while } t \text{ do } B$ and we denote its function by $W$. We let $R$ be the invariant relation generated as the nucleus of the invariant function $W$ and we let $T$ be the lower bound generated by the invariant relation $R$. Then, we find $W = T$.

**Proof.** Given the invariant function $W$, we construct the corresponding invariant relation by the formula $R = WW$, from which we derive the lower bound by the formula $T = R \circ I(\neg t)$. To prove
that $T = W$, we proceed as follow:

\[
T = W \circ \hat{W} \circ I(\neg t)
\]

\[
= \{ \text{substitution} \}
W \circ \hat{W} \circ I(\neg t)
\]

\[
= \{ \text{inverse of an identity} \}
W \circ \hat{W} \circ (I(\neg t))
\]

\[
= \{ \text{inverse of a product} \}
W \circ (I(\neg t) \circ W)
\]

\[
= \{ \text{Theorem of Mills} (1975), I(\neg t) \circ W = I(\neg t) \}
W \circ (I(\neg t))
\]

\[
= \{ \text{inverse of an identity} \}
W \circ I(\neg t)
\]

\[
= \{ \text{post-restriction of a function to its range} \}
W. \quad \square
\]

5.4. Combining lower bounds

Given a set of lower bounds, say $T_0, T_1, T_2, \ldots, T_k$, we know by hypothesis that $W$ refines each one of these lower bounds, hence we know that $W$ refines their join, provided their join exists. As we recall from Section 2.2, the join of two relations does not always exist. However, a theorem by Boudriga et al. (1992) provides that a set of relations have a join if and only if they have an upper bound; and $T_0, T_1, T_2, \ldots, T_k$ clearly have an upper bound, namely $W$. Hence their join exists. As we identify more and more lower bounds of the loop function, the resulting join climbs in the lattice of refinement; see Fig. 2.

To illustrate what it means to resolve an equation in a lattice by means of inequalities, we briefly present a numeric analogy, using a lattice whose structure is very similar to the lattice structure of the refinement ordering. Specifically, we consider the set of positive natural numbers included between 1 and 2000, and we consider the divisible-by relation between such numbers, which is a partial ordering (as is refinement): Any two numbers in $S$ have a greatest lower bound, the GCD (the meet, in the lattice of refinement); not all pairs have a least upper bound (the smallest common multiple of 300 and 301, for example, is not in $S$—likewise, not all pairs of specifications have a join); the set has a universal lower bound which is 1, that divides every element of $S$ (likewise, the lattice of refinement has a universal lower bound, the empty relation); the set has no universal upper bound (as is the case for refinement); all numbers between 1001 and 2000 are maximal (for refinement, all total deterministic relations are maximal).

Imagine having to derive a number $X$ in $S$ given that we know the following properties about it:

- $X$ is divisible by 15,
- $X$ is divisible by 21,
- $X$ is divisible by 33,
- $X$ is divisible by 35,
- $X$ is divisible by 55.

From all these claims, we infer that $X$ is divisible by the least common multiple of 15, 21, 33, 35 and 55, which is 1155. Because 1155 is maximal, the only number that is divisible by 1155 is 1155 itself. Hence $X = 1155$.

If all we knew about $X$ were that $X$ is divisible by 3, 5, 11, 15, 33, and 55, then all we could infer would be that $X$ is divisible by 165, i.e. that $X$ is in the set

\[
\{165, 330, 495, 660, 825, 990, 1155, 1320, 1485, 1650, 1815, 1980\}.
\]

As this example shows, we can compute or approximate an element in a lattice if all we know about it are some lower bounds.
5.5. Concluding the search for lower bounds

The question we ponder in this section is: how do we know we have enough lower bounds? To answer this question, we refer to the analogy with the numeric example of Section 5.4: If the join of all the lower bounds is maximal, then that join is indeed the element we are looking for; if not, then the join gives us an approximation of the element we are looking for, but does not determine it fully. Likewise, for the derivation of the loop function, if the join of all the lower bounds we find is maximal, i.e. is a total deterministic relation, then it is the function of the loop; else, it is an approximation (a comprehensive lower bound) of the function of the loop. In practice, it means we can keep deriving lower bounds until their join is total and deterministic, or (realistically) until we have derived all the lower bounds we know how to derive, whichever comes first.

6. Deployment: A prototype for deriving loop functions

To illustrate proposed applications of invariant relations, we discuss in what follows a prototype of a tool that analyzes C++ loops to derive their function.

6.1. Broad system structure

To derive the function of a loop written in a given programming language, we proceed in three steps.

1. Map the loop from its source programming language notation to a predefined language-independent internal notation. The internal notation is defined in such a way as to support the divide-and-conquer approach that we advocate. We make it language independent so as to support a wide range of programming languages with minimal overhead.

2. We analyze the loop written in the internal notation to derive equations between the initial (unprimed) variables and the final (primed) variables. This step is the core of our algorithm. We analyze small parts of the loop at a time with a view to answering the question: What equations hold between the initial values and the final values of the loop.

3. We submit the equations derived in the previous step to a system for solving symbolic equations. We obtain the function of the loop by solving the equations in the primed variables, using the unprimed variables as parameters. For now we are using Mathematica (©Wolfram Research), but we are also exploring other systems or combinations of systems.

The first step is currently being automated using compiler generation technology. The third step is fairly trivial, since the equations generated by the second step are written directly in Mathematica.
notation. Nevertheless, this step is currently the bottleneck of our capability, in the sense that it determines what aspects of a program we can or cannot handle. In fairness, we make provisions for the possibility that what is limited is not the capability of Mathematica, but rather our (limited, but growing) understanding of it.

The second step is the focus of our subsequent discussion.

6.2. The internal representation

Since Theorem 4 requires that we find a superset of the loop body, we must represent the loop body in a way that makes supersets visible. As we discussed in Section 5.2, we can make supersets visible by structuring the loop body’s function as an intersection of relations. In its current form, our algorithm does not handle loop bodies structured as if-then-else statements, even though we have the theory to do so (re: Theorem 5). So that the focus of the internal notation is to rewrite the loop body as a set of (conditional) concurrent assignments (CCA’s). Accessorily, we also record some data type information that is relevant to the algorithm. Note that unlike a programming language, this notation does not require that constants be assigned a value, only that they be declared as constants; they are subsequently used as parameters in deriving the loop function. A Sample program written in this notation is given in Section 7.2.

6.3. Deriving lower bounds

Once the loop body is structured in CCA form, we can derive lower bounds by looking at one statement at a time, or two statements at a time, or three statements at a time, etc. For the sake of controlling combinatorics, we resolve not to look at more than three statements at a time. The device we use to this effect is called a recognizer, and is made up of the following elements:

- A pattern of typed constants.
- A pattern of typed variables.
- A pattern of statements.
- A corresponding relation expressed in terms of the formal constants and variables, that represents a lower bound.

When a formal statement pattern of a recognizer matches an actual set of statements, we generate an instantiation that maps formal variable names to actual variable names; this instantiation is then applied to the formal lower bound of the recognizer to generate an actual lower bound. We distinguish between one-recognizers that match one statement at a time, two-recognizers that match two statements at a time, three-recognizers that match three statements at a time.

The current status of development of the extraction algorithm can be characterized by the following statements:

- All the machinery for recognizing code patterns and generating instantiated lower bounds is currently in place.
- We have a total of 29 recognizers, including 10 one-recognizers, 16 two-recognizers, and 3 three-recognizers.

We can augment the scope of applicability of the algorithm by adding more recognizers, to handle new control structures and new data structures. Figs. 3–5 show some sample recognizers that are currently implemented. Even though it may sound redundant, we show both the invariant relation and the corresponding lower bound, to highlight the role that invariant relations play in our approach.

6.4. Combining lower bounds

Each recognizer produces (when it is successfully matched) a logic formula, which represents the relevant lower bound. In principle, we must now compute the join of all the lower bounds. However, all the lower bounds are total relations; by virtue of Proposition 1, their join equals their intersection.
In logical terms, this means that we take the conjunction of the formulas generated by the recognizers. The join of all the lower bounds is itself a lower bound of the loop function. If this join is a total deterministic relation then it is the function of the loop; else it is a lower bound of the function of
the loop (i.e. it specifies some, but not all, of the functional properties of the loop). In practice, if Mathematica returns an expression for each primed state variable, and no restriction on the unprimed state variables, then we have found the function of the loop (because then the equations that represent the join of the lower bounds define a total deterministic relation).

6.5. Generating recognizers

The current prototype includes a total of 29 recognizers, including 10 one-recognizers, 16 two-recognizers, and 3 three-recognizers. All the machinery for storing, retrieving and using recognizers is in place; hence we can add recognizers at will to expand the capability of our tool. When the size of the database grows significantly, performance may become an issue, and we may have to alter the algorithm that matches recognizers against .cca code to control combinatorics, but we are not there yet.

Generating recognizers is not difficult, and is not the bottleneck in the development of our prototype; we usually generate them either by referring to the definition of invariant relations (finding a reflexive transitive superset), or by means of invariant functions (finding an invariant function then taking its nucleus). What has been our greatest difficulty is to produce recognizers whose lower bounds can subsequently be solved by the computer algebra system. While Mathematica is proving to be very effective with numeric computations and with some simple data structures (e.g. arrays), it is proving unable to deal with other aspects that are important to us (axiomatized data structures, relevant hypotheses, symbolic operations, etc). We are investigating a hybrid solution, which combines numeric computations and some theorem proving capabilities.

7. Illustration

7.1. A C++ loop

We consider the C++ program given below, and we are interested deriving the function of its loop. This program handles integer variables, and also includes arrays, lists and (symbolic) function calls. Note that we have a number of constants in this program; even though we must define a value for these constants to run the program, we are interested in deriving the loop function using these constants as parameters. For the sake of simplicity, we assume that constants \( a, b \) are different from 0, and that constant \( d \) is different from 0 and 1 (without these hypotheses, the expression of the function would be very complex). Also, we assume that variable \( i \) is non-negative, and that it causes no failure of this loop (indices \( i \) and \( j \) remain within range of their arrays, and the length of list \( l \) is greater than or equal to \( i \)).

```cpp
#include <iostream>       // 1.
#include <cmath>          // 2.
#include <math.h>         // 3.
#include <list>            // 4.
using namespace std;     // 5.

canst int a= , b= , c= , d= , e= ; // 8.
canst int N= ;           // 10.
typedef list <int>      // 12.
listtype;               // 13.
```

\footnote{The current algorithm can automatically generate some of these conditions; we are currently exploring means to automatically generate all of them.}
listtype l; // 15.
listtype m; // 16.
int q, qc; // 18.
int x, y, z, t, i, j, v, w, SA, Sn; // 20.
void loop (); // 23.
int f (int x); // 25.
int main () // 27.
{loop();} // 28.
void loop () // 30.
{ // 31.
    while (i != 0) // 32.
    { // 33.
        y = y+b; // 34.
        v = v+a*t; // 35.
        w = w+e*y-b*e; // 36.
        x = x+a; // 37.
        t = t*d; // 38.
        sA = sA + A[i]; // 39.
        sB = sB + B[j]; // 40.
        i = i-1; // 41.
        z = z+c*x-a*c; // 42.
        j = j+1; // 43.
        m.push_back(l.front()); // 44.
        l.pop_front(); // 45.
        q = f(q); // 46.
        qc = qc + q; // 47.
    } // 48.
} // 49.
int f (int x) // 51.
{return (//some function of // 52.
    x);} // 53.

7.2. The loop in internal form

loop.cca:
{
const int a; const int b; const int c;
const int d; const int e; const int N;
const function f;
array int A; array int B;
list l; list m;
int q; int qc;
int x; int y; int z; int t; int i;
int j; int v; int w; int sA; int sB;
while !(i == 0)
The algorithm produces 56 equations, of which we present the following excerpts:

```
loop.mat

1. Reduce[ Reduce[ {
2. Mod[x,Abs[a]]==Mod[xP,Abs[a]],
3. Mod[y,Abs[b]]==Mod[yP,Abs[b]],
4. Mod[t,Abs[Log[d,10]]]==
   Mod[tP,Abs[Log[d,10]]],
5. Mod[i,Abs[1]]==Mod[iP,Abs[1]],
6. i>=iP,
7. Mod[j,Abs[1]]==Mod[jP,Abs[1]],
8. j<=jP,
9. A==AP,
10. B==BP,
11. v+a*t/(1-d)==vP+a*tP/(1-d),
12. z-c*x*(x-a)/(2*a)==
    zP-c*xP*(xP-a)/(2*a),
13. w-e*y*(y-b)/(2*b)==
    wP-e*yP*(yP-b)/(2*b),
14. a*y-b*x==a*yP-b*xP,
15. b*x-a*y==b*xP-a*yP,
16. t/d^(x/a)==tP/d^(xP/a),
17. a*i+1*x==a*iP+1*xP,
18. a*j-1*x==a*jP-1*xP,
19. 1*x-a*j==1*xP-a*jP,
20. t/d^(y/b)==tP/d^(yP/b),
21. b*i+1*y==b*iP+1*yP,
22. b*j-1*y==b*jP-1*yP,
23. 1*y-b*j==1*yP-b*jP,
24. t/d^(j/1)==tP/d^(jP/1),
25. 1*i+1*j==1*iP+1*jP,
26. 1P==Nest[Rest,1,i-iP],
27. i-Length[l]==iP-Length[lP],
28. Nest[f,q,i]==Nest[f,qP,iP],
29. 1P==Nest[Rest,1,jP-j],
30. j+Length[l]==jP+Length[lP],
31. Join[m,l]==Join[mP,lP],
32. sA+Sum[A[k], {k,1,i}] ==
    sAP+Sum[AP[k], {k,1,iP}],
33. sB+Sum[B[k], {k,j,N}] ==
    sBP+Sum[BP[k], {k,jP,N}],
34. qc+Sum[Nest[f,q,k],{k,1,i}]==
    qcP+Sum[Nest[f,qP,k],{k,1,iP}],
35. (iP==0),
```
36. Exists [ {APP,BPP,iPP,lPP, mPP,qPP,qcPP,sAPP,sBPP, tPP,vPP,wPP,xPP,yPP,zPP},
37. !(iPP==0) && ... ... ...
38. zP==zPP+c*xPP &&
39. vP==vPP+a*tPP]
40. ]},
42. Backsubstitution->True]

Lines 1 and 43 are Mathematica instructions/options. Lines 41 and 42 specify that we want the given equations resolved in these variables, which are the final values of the program variables. Lines 2–8 represent the application of one-recognizers. Lines 9–31 represent the application of two-recognizers. And lines 32–34 represent the application of the three-recognizers. The reader can trace back some of the equations to a recognizer among those we presented in Figs. 3–5. Line 35 represents the clause \( \neg t(s') \) that we have factored out from all the lower bounds. As the reader may have noted, these equations are highly redundant; we have made no effort to minimize redundancy, as that would complicate the algorithm, and performance is not yet an issue, given the small number of recognizers that we have now.

Lines 36–40 represent the application of a theorem (which we have not presented here, Mili et al. (2008)) which provides a trivial lower bound that says in effect that not only does the final state satisfy condition \( \neg t \), but it has a predecessor by \([B]\) that satisfies \( t \). This theorem specifies that the state immediately preceding the final state (represented by \( PP \)) satisfies the loop condition \( t \); in other words, the final state (specified by \( P \)) is the first state that fails to satisfy the loop condition. This clause is useful in cases where the loop condition is an inequality.

### 7.4. The function of the loop

The function of this loop is given in Fig. 6 (where list concatenation is represented by a dot). It includes two terms: the trivial term where \( i = 0 \) and all variables are preserved; the non-trivial term where \( i \neq 0 \) and program variables are altered. This figure gives the final values (primed) of the program variables as a function of the initial values (unprimed). Notice that even though we do not have an explicit expression for \( m' \) in the first term of the loop function (Fig. 6), the equation \( m'.Rest^i(l) = m.l \) uniquely characterizes \( m' \). Most, though not all, clauses of this function can be inferred by analyzing the text of the program. To gain confidence in the validity of the proposed function, we ran tests on this program using the proposed function as an oracle. We obtain different test cases by varying the values of the variables and the constants (within the prescribed ranges, e.g. avoiding 0 for \( a, b, d \) and 1 for \( d \)). Given the number of variables and constants, the volume of the test data sample grows very quickly, to the tens of millions. The program ran successfully on all of them.

For the sake of comparison, we submitted the same program to Daikon (Ernst et al., 2006), which generate invariant assertions by applying machine learning techniques to the execution trace. Because it operates on execution traces (rather than on source code), Daikon requires that we fix all the constants (a significant loss of generality, since then it makes a statement not about a broad family of programs, but rather about a single program). Daikon did find some of the clauses of the function given in Fig. 6, duly specialized to the constant values. In fairness, we must recognize that, because Daikon operates on execution traces, it can handle any program structure, whereas we can only handle program structures for which we have made prior provisions (as discussed in what follows).

### 8. Related work

Our work is related to three lines of research: research on deriving loop functions, with which it shares a common goal; research on deriving invariant assertions, with which it shares common
analytical methods; and research on program slicing, with which it shares common divide-and-conquer approaches. We discuss these in turn, below.

The closest work we have found to our effort, in terms of goal (generating loop functions) and means (using Mills-like functional/relational logic) is the work by Dunlop and Basili (1984). In this work, Dunlop and Basili discuss a syntactic method that derives the function of a loop by attempting to generalize from known formulas that capture the behaviors of the loop under special conditions. Dunlop and Basili’s approach is very syntactic, and uses a very small set of rules, that has limited scope of application. Carette and Janicki (2008) infer functional properties of loops from an analysis of their source code: they derive recurrence equations of numeric loops and submit them to Maple to be solved. In some cases, the invariant relations that we are interested in are identical to the recurrence equations derived by Carette and Janicki, from which the recurrence variable has been removed.

Generally, the derivation of invariant assertions is closely related to the derivation of loop functions since they both aim to discover the inductive argument that underlies the behavior of the loop. Furthermore, a theorem by Mills (1975) shows how loop functions can be used to produce loop invariants. Also, the generation of lower bounds that we carry out to approximate the function of a loop is reminiscent of the extensive work that has been done and is being done on generating invariant assertions (Jebelean and Giese, 2007). Many researchers in the theorem proving and the program verification communities have lent much attention to the goal of extracting invariant assertions (Carbonnell and Kapur, 2004; Scholz and Fahringer, 2003; Colon et al., 2003; Sankaranarayana et al., 2004; Kovacs and Jebelean, 2005; Cheatham and Townley, 1976; Cousot and Halbwachs, 1978; Karr, 1976; Marlowe and Ryder, 1990; Sharir and Pnueli, 1981). Ernst et al. (2006) discuss a system for dynamic detection of likely invariants; this system, called 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. Denney and Fischer (2006), analyze generated code against safety properties, for the purpose of certifying the code. Colon et al. (2003), consider invariant assertions of numeric programs as linear expressions and derive the coefficients of the expressions by solving a set of linear equations; they extend this work to nonlinear expressions.
in Sankaranarayana et al. (2004). Kovacs and Jebelean (2005) derive invariant assertions by solving recurrence relations; they pose the invariant assertions as solutions to recurrence relations, and derive closed forms of the solution using a theorem prover (Theorema) to support the process. Carbonell and Kapur (2004) derive invariant assertions by forward propagation and fixed point computation, with robust theorem proving support; they represent loop bodies as conditional concurrent assignments, whereas their insights are of interest to us as we envision to integrate conditionals into our concurrent assignments. In Mili (2007), we discuss the difference between traditional invariant assertions (in the sense of Hoare’s logic (Hoare, 1969; Gries, 1981)) and the invariant relations that we derive in this paper from invariant functions, which we call reflexive transitive invariant relations.

Hu et al. (2004) present a technique for slicing while loops while attempting to minimize slice sizes. The technique is based on identifying the induction variable of the loop, and applying semantics-preserving transformations that represent the effect of the loop by an if-then-else statement. Our work differs from that of Hu et al. in many ways, including: first, we do not need to identify an inductive variable (we can think of cases where no such variables can be defined, let alone identified); by finding reflexive transitive supersets of the loop body, we in fact do away with the inductive argument altogether; second, our lower bounds can be arbitrarily partial, as they are not driven by the syntactic structure of the loop (while slicing techniques slice the program, our divide-and-conquer techniques slice the program’s function); third, the relation of our lower bound to the function of the loop is well defined (refinement), as is the rule for composing lower bounds (join).

9. Conclusion

9.1. Invariant assertions, relations, functions

In this paper, we have discussed three concepts pertaining to a while loop of the form while \( t \) do B:

- **Invariant assertions**, which are predicates that are preserved by execution of the loop body; they are preserved in the sense that if they are true before execution of the loop body, they remain true after.
- **Invariant relations**, which are reflexive transitive relations that are supersets of the function of the loop body (as such, they characterize the relation that holds between two states of the program separated by an arbitrary number of iterations).
- **Invariant Functions**, which are total functions that take the same value before and after execution of the loop body.

We note that invariant assertions are not invariant in the sense of invariant functions: whereas an invariant function takes the same value before and after execution of the loop body, an invariant assertion may be false before execution of the loop body and true after. One may argue that invariant assertions ought to be called increasing rather than invariant. We have found the following generality relations to hold between these concepts:

- From an invariant function, we can derive an invariant relation (Proposition 5).
- From an invariant relation, we can derive an invariant assertion (Proposition 3).
- From an invariant relation that is symmetric (in addition to being reflexive and transitive) we can derive an invariant function (Proposition 6).

All these invariants are ordered: invariant assertions are ordered by logical implication; invariant relations are ordered by refinement; and invariant functions are ordered by injectivity (more injective functions define a finer partition of their domain).

9.2. Invariant assertions v. invariant relations

Invariant relations of while loops are reflexive transitive binary relations that link pairs of program states separated by an arbitrary number of iterations. They bear some resemblance with invariant assertions (with which they are often confused), but they are distinct from them in many ways:
Whereas invariant assertions are unary predicates, invariant relations are binary relations.

Whereas invariant assertions depend not only on the loop, but also on its specification (in the form of a precondition/postcondition pair), invariant relations depend solely on the loop.

Whereas invariant assertions express an inductive argument on the trace of execution of the loop starting from a fixed initial state, invariant relations express an inductive argument on the number of iterations separating two states.

Whereas the strongest invariant assertion is (a subset of) the nucleus of the function of the initialized loop, the strongest invariant relation is the nucleus of the function of the uninitialized loop.

Whereas invariant assertions are useful to prove the correctness of loops with respect to specifications (in the form of precondition/postcondition pairs), invariant relations are useful to compute or approximate loop functions.

Not surprisingly, in light of the above observations, invariant assertions and invariant relations have but a tenuous relation. Our experience shows two crucial facts about the contrast between invariant assertions and invariant relations:

- From a given invariant assertion, we do not know how to generate an invariant relation (even though we do know how to derive an invariant assertion from an invariant relation: see Proposition 3).
- From a given invariant assertion generation technique, we do not know how to generate an invariant relation generation technique.

9.3. Prospects

In this paper we have explored the differences and similarities between invariant assertions and invariant relations. In the process, we have encountered an auxiliary concept, that of invariant function, whose differences and similarities with invariant assertions and invariant relations we have also touched upon. Among venues of further research, we cite the following:

- Given that invariant functions can be used to generate invariant relations and that invariant relations can be used to generate invariant assertions, it appears that invariant functions play a central role in analyzing loops. We envision to further investigate invariant functions so as to refine techniques for generating them, as well as explore their properties and applications.
- Another direction we are exploring is the parallel between invariant assertions (ordered by logical implication), invariant relations (ordered by refinement) and invariant functions (ordered by injectivity). We have alluded to this parallel in Section 5.3, but there is a lot more to say on this subject than we have.
- One of the byproducts of the parallel alluded to above is an analysis of the structure of invariant assertions. As we said in Section 5.3, an invariant assertion is not an invariant function that happens to take its values in the set of booleans. Rather, while an invariant function keeps the same value before and after each iteration, an invariant assertion does not, since it may be false before execution of the loop body and become true afterwards, leading us to suggest that invariant assertions ought to be called monotonic assertions instead. There is more to this issue than how we call invariant assertions: we conjecture that invariant assertions are typically made up of a genuinely invariant term and a monotonic term; techniques for generating these terms may differ (with the former resembling techniques we use to generate invariant functions). Whereas monotonic terms are characterized by the inductive condition

\[ \{ \chi(s_0, s) \land t \} B \{ \chi(s_0, s) \}, \]

invariant terms can be characterized by the condition

\[ \{ \chi(s_0, s) \land t \} B \{ \chi(s_0, s) \}, \]

\[ \land \]

\[ \{ \neg \chi(s_0, s) \land t \} B \{ \neg \chi(s_0, s) \}. \]
Another venue of research is the continued development of the prototype. We envision to expand the database of recognizers to cover various application domains, and to explore additions or alternatives to the use of Mathematica to solve the equations generated by the recognizers. In practice, solving the equations generated by recognizers has been a much bigger challenge for us than generating recognizers, and is proving to be a much tighter bottleneck.

Acknowledgements

We are very grateful to the anonymous reviewers for their valuable feedback, which has greatly enhanced the focus of the paper. We are also very grateful to Guest Editors Professor Martin Giese and Professor Tudor Jebelean, for their patience and their support through the reviewing process.

References


