A Semantic Recognizer Infrastructure for Computing Loop Behavior

Ali Mili[1,2] Tim Daly[2], Mark Pleszkoch[2], and Stacy Prowell[2]

[1]: College of Computer Science
New Jersey Institute of Technology
Newark NJ 07102-1982

[2]: Software Engineering Institute
Carnegie Mellon University
Pittsburgh PA 15213-3890

mili@cis.njit.edu, {daly,mpleszko,sprowell}@cert.org

September 7, 2006

Abstract

Next-generation software engineering is envisioned as a computational discipline that complements human capability through automated computation of software behavior and properties to the maximum extent possible. To help realize this vision, we are exploring the technology of function extraction. Behavior computation for sequences and alternations is relatively straightforward, but no general theory for loop computation can exist, and engineering solutions must be sought. This paper proposes an infrastructure for loop computation based on hierarchies of semantic recognizers operating within a refinement calculus, and outlines an extraction algorithm for computing loop behavior based on application of the recognizers.

1 Introduction: Computing Program Behavior

1.1 Position of the Problem

It is increasingly evident that software engineering is reaching cost and complexity limits of development technologies evolved in the first fifty years of computing. We envision next-generation software engineering (NGSE) as a computational discipline, capable of rigorous automated analysis of its subject matter, just as, for example, present-day electrical engineering benefits from computational methods in developing and analyzing its engineering artifacts [12].

A principal objective of NGSE is dramatic reduction in the cost and complexity associated with software engineering. We believe that a key capability for achieving this objective is automated computation of the behavior of programs and other software engineering representations to the maximum extent possible. This capability will fill a gap in the ability of software engineers to quickly and reliably understand the functionality of programs written by themselves and others, both during and after development. Also, this capability would have substantial impact on many engineering activities, from design and verification, to implementation and testing, to maintenance and evolution. Behavior computation is an extremely difficult problem, but the substantial value of a solution motivates a closer look.

The function-theoretic view of software illuminates a strategy for computing program behavior [15, 16, 18]. Function-theoretic methods treat programs as rules for mathematical functions or relations, that is, mappings from domains to ranges, regardless of their subject matter or programming language. A function theorem defines behaviorally equivalent but non-procedural functional representations for sequence, alternation, and iteration control structures (and their variants), which are themselves sufficient to represent any sequential logic. These functional forms are the starting point for behavior computation. In structured form, programs present a hierarchy in their constituent control structures that can be traversed bottom up to compute intermediate and eventually overall behavior in a stepwise manner. The theorem of sequence structures shows that the behavior of sequence structures can be computed by means of ordinary function composition to arrive at net effects that are easily expressed as concurrent assignments of initial values to final values. The behavior of alternation structures is captured in a case analysis of true and false branches that is easily expressed as conditional concurrent assignments.

The behavior of iteration structures is defined by the function theorem as a recursive equation that, while correct, does little to abstract loop behavior in terms meaningful to programmers. In addition, mathematical results show that no comprehensive theory for loop behavior computation can exist, so engineering solutions must be sought. This does not mean that rigorous formulations for loop computation cannot be developed. In particular, this paper provides the infrastructure for such a formulation in terms of semantic recognizers operating within a refinement algebra.

CERT STAR*Lab at the Software Engineering Institute is developing the emerging technology of function extraction (FX) for automated computation of program behavior to the maximum extent possible. An FX prototype was used in a rigorous experiment to compare traditional manual and automated analysis of program behavior. Results showed automated function extraction to be a significant improvement over manual methods [4]. The Function Extraction for Malicious Code (FX/MC) system is being developed with the aim of providing fast and reliable analysis of malicious code expressed in Intel assembly language [17]. Function extraction has substantial implications across the software engineering life cycle, as discussed in [8].

1.2 Related Work

Computing the behavior of loops is akin to deriving loop invariants, in that they are both aimed at shedding light on the inductive argument that underlies the loop behavior. While the extraction of loop functions has not attracted much attention in the past, the analysis and derivation of loop invariants has gained renewed attention recently. In [5], Colon et al. consider loop invariants of numeric programs as linear expressions and derive the coefficients of the linear expressions by solving a set of linear equations; they extend this work to non linear expressions in [19]. In [10, 11] Kovacs and Jebelean derive loop invariants by solving recurrence relations; they pose the loop invariants as solutions to recurrence relations, and derive closed forms of the solution using a theorem prover (Theorema) to support the process. In [2] Rodriguez Carbonnell et al. derive loop invariants by forward propagation and fixed point computation, with robust theorem proving support; they represent loop bodies as conditional concurrent assignments, whence their insights are of interest to us as we envision to integrate conditionals into our concurrent assignments. Less recent work on loop invariants includes work by Cheatham and Townley [3], Karr [9], Cousot and Halwachs [6], and Mili et al [13]. Work on loop analysis and loop transformations in the context of compiler construction is also related to functional extraction, although to a lesser degree than work on loop invariants [1, 7].

1.3 Research Plan

In section 2 we introduce the main premises of our approach by articulating our separation of concerns discipline and highlighting the role of semantic recognizers therein. In section 3 we present some sample recognizers and discuss the properties of the recognizer infrastructure, and its impact on the performance of the function extraction machinery. In section 4 we present the main algorithms and data structures of the loop extraction process, using the Z notation. We illustrate the operation of this algorithm in section 5 using a simple illustrative example, then we conclude in section 6 by summarizing our findings and sketching future directions of research.

2 Research Background

In Mili et. al. [14] we propose a divide-and-conquer framework that allows us to derive the function of a loop in a stepwise manner. This framework proceeds by successive approximations of the loop function, in the form of refinement statements such as:

$$[w] \sqsupseteq T_i,$$

where w is the while loop, $[w]$ is the function of the loop, and T_i is a relation that expresses some functional properties between initial states and final states; this refinement statement basically says that the loop has all the functional properties expressed in relation T_i. We then say that T_i is a lower bound for $[w]$. In Mili et. al. [14] we have shown that from refinement statements above we can infer an aggregate statement such as

$$[w] \sqsupseteq T_1 \cap T_2 \cap T_3 .. \cap T_n.$$

We have also shown that if the right hand side produces a total deterministic relation, then we can infer

$$[w] = T$$

from $[w] \sqsupseteq T$, where $T = T_1 \cap T_2 \cap T_3 .. \cap T_n$. This simple mathematical result forms the basis of our approach to the automated derivation of loop functions, provided we learn how to derive individual refinement claims.

We have found three broad sources for deriving lower bounds T_i for the loop function:

- *Using Invariant Functions*. The invariant function of a while loop of the form `while t do B` is a total function F such that:

  $$t(s) \Rightarrow F(s) = F([B](s)).$$

  We have found for any invariant function F of w, we can find a lower bound T of $[w]$ in the form

  $$T = F \circ \widehat{F} \circ I(\neg t).$$

- *Using Non Surjective Loop Bodies*. We have found a lower bound of the loop function in the form

  $$T = (L \circ [B] \cup I) \circ I(\neg t).$$

  If $[B]$ is surjective, then this lower bound becomes $L \circ I(\neg t)$, which is hardly informative. But if $[B]$ is non-surjective, we obtain a non-trivial lower bound of $[w]$.

- *Using Boundary Conditions*. If the combination of the lower bounds obtained from the previous analyses fails to produce a function, we have recourse to the observation that the final state of the loop not only satisfies $\neg t$, it is also the first state that does so in the iteration process (i.e. its predecessor by $[B]$ satisfies t). This observation leads to the following lower bound:

  $$T = I(t) \circ L \circ I(t) \circ [B] \circ I(\neg t) \cup I(\neg t).$$

  We refer to this as the *Boundary Condition Relation*; in the Z specification (below) we refer to it as `BCRelation` (BC: boundary condition).

One way to define a function from S to S is to proceed in two steps:

- Define the level sets of F. A *level set* of a function is the set of elements in the function's domain that have the same image.

- Determine which image gets assigned to each level set.

Interestingly, it seems the invariant functions method (above) aims to partition the domain of $[w]$ into the level sets of $[w]$, and the other two methods cited thereafter appear to be geared towards assigning a unique image to each level set. This seems to suggest some degree of completeness to the proposed approach. Be that as it may, the results we have found so far [14] all fall into one of these three categories.

3 A Structure for Semantic Recognizers

We view the semantic recognizers not as an unstructured monolith of patterns, but rather as a hierarchical structure ordered by generality. Also we envision that the algorithm that attempts to recognize patterns in the source code to derive lower bounds of the loop function attempts first to match lower level patterns, and climbs up the tree only if lower level patterns do not produce a match with the source code. Figure 1 illustrates this structure, and highlights the tension between generality/ usefulness and genericity/ usability that arises in defining these patterns. To illustrate the contrast between generality (usefulness: likelihood of use) and genericity (usability: ease of use), we consider the following example: We know from [14] that if the loop body contains two statements such as

SR: `i:= i-1, x:= x+i`

then the loop refines the following relation (where t is the loop condition)

$$T = \{(s,s')|x + \frac{i(i+1)}{2} = x' + \frac{i'(i'+1)}{2} \wedge \neg t(s')\}.$$

Because these two statements are too specific, we may want to generalize them into the following form:

SR': `i:=i-c, x:=x⊕i.`

Now the step by which we decrement i is arbitrary, and the operation by which we compose x and i is also arbitrary; whence we have obtained a more general pattern, that is more widely applicable. But this additional generality comes at a cost in terms of usability, since now the lower bound of the loop function has the following form (provided $\oplus$ is associative):

$$T' = \{(s,s')|x \oplus \bigoplus_{k=1}^{i \div c} k \times c = x' \oplus \bigoplus_{k=1}^{i' \div c} k \times c\}.$$

SR' is more general than SR, hence more widely applicable. But this generality comes at a cost: The abstract operators in this relation must be instantiated with concrete operators at each application; also the rewrite rules that are specific to the concrete operators must be brought to bear in any subsequent manipulation and/ or simplification of the relation. By contrast, relation T is readily usable as it is. This is why our policy is to always match any statement or set of statements with the lowest possible node in the tree of semantic recognizers.

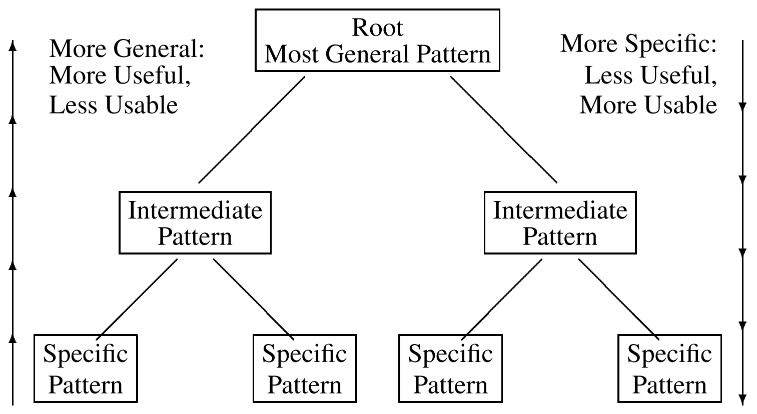

Figure 1: Pattern Tree

State Space	Semantic Pattern	Refined Specification

i: int
x: sometype
a: sometype
	i:=i-1,
	x:=f(x)
	a:=a+x
		$T =$
		$\{(s,s')\mid f^i(x) = f^{i'}(x') \wedge$
		$a + \Sigma_{k=1}^{i} f^k(x) = a' + \Sigma_{k=1}^{i'} f^k(x') \wedge \neg t(s')\}$

Figure 2: Cross Product Recognizer

In this section, we present some semantic recognizers, to build on the structure started in [14]. Semantic Recognizers are defined by their state space, the pattern of statements they recognize, and the the lower bound that they claim for the loop function ($[w]$) whenever a match is successful. The first pattern, which we call the *cross product pattern*, captures the common situation where a variable is updated in one assignment while being accumulated in another; it is described in Figure 2. This pattern can be generalized in many ways (generalizing the '+', the increment of i, the direction of the increment, etc) but we keep it simple here. The basic idea of this pattern is to combine the computation of a variable (x) with the use of that variable (in the assignment of a); this is clearly a recurring situation in programs. We briefly illustrate this pattern using the following loop:

```
w =
while (i<>0) do
   [i:= i-1,
   x:= x-1,
   y:= y+x]
```

We find the following lower bound for $[w]$:

$$T$$

$$\begin{aligned}
= & \quad \{ \text{According to the Table} \} \\
& \{(s,s')\mid f^i(x) = f^{i'}(x') \\
& \wedge y + \Sigma_{k=1}^{i} f^k(x) = y' + \Sigma_{k=1}^{i'} f^k(x') \wedge i' = 0\} \\
= & \quad \{ \text{since } f^0(x') = x' \} \\
& \{(s,s')\mid x' = f^i(x) \wedge \\
& y + \Sigma_{k=1}^{i} f^k(x) = y' + \Sigma_{k=1}^{i'} f^k(x') \wedge i' = 0\} \\
= & \quad \{ \text{since } f^k(x) = x - k \} \\
& \{(s,s')\mid x' = x - i \wedge \\
& y + \Sigma_{k=1}^{i} f^k(x) = y' + \Sigma_{k=1}^{i'} f^k(x') \wedge i' = 0\} \\
= & \quad \{ \text{since } \Sigma_{k=1}^{0} A = 0, \text{ whatever } A \text{ is} \} \\
& \{(s,s')\mid x' = x - i \wedge \\
& y' = y + \Sigma_{k=1}^{i} f^k(x) \wedge i' = 0\} \\
= & \quad \{ \text{since } f^k(x) = x - k \} \\
& \{(s,s')\mid x' = x - i \wedge \\
& y' = y + \Sigma_{k=1}^{i} x - k \wedge i' = 0\} \\
= & \quad \{ \text{case analysis} \} \\
& \{(s,s')\mid x \geq i \wedge x' = x - i \wedge \\
& y' = y + \tfrac{x(x+1)}{2} - \tfrac{i(i+1)}{2} \wedge i' = 0\} \\
& \cup \{(s,s')\mid x < i \wedge x' = x - i \wedge \\
& y' = y + \tfrac{x(x+1)}{2} - \tfrac{(i-x)(i-x+1)}{2} \wedge i' = 0\}.
\end{aligned}$$

This function is clearly total, since the domains of the two terms are complementary. It is also deterministic, since the domains of the two terms are disjoint and each term is deterministic. Whence we infer that $[w]$ not only refines this function; it actually equals it.

We introduce, in Figure 3, three more semantic recognizers, which will be used in our subsequent discussion of a loop extraction algorithm. One of these recognizers matches a single statement, and two match pairs of statements.

4 Outline of an Extraction Algorithm

We attempt to specify the extraction algorithm using the Z notation [20]. To this effect, we provide, in turn, data types, then state variables, then initializations, and finally some operations.

4.1 Data Types

On the basis of the foregoing analysis, we have determined that it is worthwhile to pursue a pattern recognition approach, and we speculate that we can extract loop behaviors in a stepwise manner, using predefined semantic recognizers. In the sequel, we present a Z specification of the data structures that are needed, in our view, to support the extraction of loop functions according to the framework discussed in [14]. First, we introduce some basic Z types for the purpose of this discussion, which include:

NAMETYPE (for variable names),
TYPETYPE (for variable and expression types),
EXPRESSIONTYPE (for expressions),
RELATIONTYPE (for relations).

The type `RELATIONTYPE` can be represented as a logical expression whence it is a sub-type of `EXPRESSIONTYPE`, but we do not necessarily want to make that determination now. Part of the matching process is a type matching, hence we briefly introduce the concept of typed variable declaration.

VARDECLARATION

```
Name: NAMETYPE
Type: TYPETYPE
```

Using the type `VARDECLARATION`, we introduce the type `PROGRAMSPACE`, as follows:

PROGRAMSPACE

```
Vars: P(VARDECLARATION)
-----------------------------
// No two vars have the same name
```

We need a variable of type `PROGRAMSPACE` to keep track of the program variables that are of interest for the function extraction. Because loop bodies are represented as conditional concurrent assignments (CCA) prior to the extraction step, we introduce the type `CCATYPE`, as follows:

CCATYPE

```
Var: NAMETYPE;
Expr: EXPRESSIONTYPE;
-----------------------------
type(Var) = type(Expr)
```

The condition of equality between the type of the variable (left hand side) and the type of the expression (right hand side) can be replaced by an equality, to account for the fact that it is sufficient for the expression type to be convertible into the variables type. Conditional Concurrent Assignments are actually more complex than that (since they include conditions), but we keep this for now, as the semantic recognizers we have so far do not handle conditions yet. Using `CCATYPE`, we derive a characterization of the loop as follows:

LOOPTYPE

```
Space: PROGRAMSPACE
Cond: EXPRESSIONTYPE
Body: P(CCATYPE)
-----------------------------
// No Undeclared Variables
```

Semantic Recognizers recognize CCA's or combinations of CCA's and infer from them lower bounds of the loop function in the form of a relation. We classify Semantic Recognizers by the number of CCA's they recognize at once. For the sake of this initial prototype, we can consider recognizers that match up to three CCA's.

SR1TYPE

```
Cca: CCATYPE
Relation: RELATIONTYPE
```

This type of recognizer produces a lower bound of the loop function using a single CCA. We briefly introduce similar structures for pairs and triplets of CCA's.

SR2TYPE

```
Cca1: CCATYPE
Cca2: CCATYPE
Relation: RELATIONTYPE
```

State Space	Syntactic Pattern	Refined Specification

x: int;
const c: int;
$c \neq 0$
	x:= x+c	$T = \{(s,s') \mid x \ \mathbf{mod}\ c = x' \ \mathbf{mod}\ c \wedge \neg t(s')\}$

x, y: int
canst a, b: int
	x:=x+a,
	y:=y+b
		$\{(s,s') \mid x \ \mathbf{mod}\ \lvert a\rvert = x' \ \mathbf{mod}\ \lvert a\rvert \wedge$
		$Au - bx = Au' - bx' \wedge \neg t(s')\}$

x, y: int
canst a, b: int
	x:=x+a
	y:= y+b*x
		$\{(s,s') \mid x \ \mathbf{mod}\ \lvert a\rvert = x' \ \mathbf{mod}\ \lvert a\rvert$
		$\wedge y - b \times \frac{x(x-a)}{2\times a} = y' - b \times \frac{x'(x'-a)}{2\times a}$
		$\wedge \neg t(s')\}$

Figure 3: Arithmetic Recognizers

SR3TYPE

Cca1: CCATYPE
Cca2: CCATYPE
Cca3: CCATYPE
Relation: RELATIONTYPE

Using individual recognizer types, we define the type recognizer library; we produce one such library for each number of CCA's.

SR1LIBRARYTYPE = P(SR1TYPE)
SR2LIBRARYTYPE = P(SR2TYPE)
SR3LIBRARYTYPE = P(SR3TYPE)

Strictly speaking, these should not be flat sets but rather tree structures, where the most general patterns fit higher in the tree. The tree structure allows us to zoom in faster on the most specific recognizer that matches a given statement or combination of statements. However, for an initial prototype, the libraries are small (include a small set of recognizers), hence the search algorithm makes little difference as far as performance is concerned.

4.2 State Space

The variables that we need to represent the state of the extraction process include: a representation of the loop, a representation of the recognizer libraries, and a representation of the relation that is progressively taking shape as the loop function. We write:

LoopFunctionExtractor

Loop: LOOPTYPE
SR1Library: SR1LIBRARYTYPE
SR2Library: SR2LIBRARYTYPE
SR3Library: SR3LIBRARYTYPE
LoopFunction: RELATIONTYPE

4.3 Initialization

ExtractorInit

Δ LoopFunctionExtractor
ObjectLoop?: LOOPTYPE
SR1Lib?: SR1LIBRARYTYPE
SR2Lib?: SR2LIBRARYTYPE
SR3Lib?: SR3LIBRARYTYPE

Loop' = ObjectLoop?
SR1Library' = SR1Lib? // 1-recognizers
SR2Library' = SR2Lib? // 2-recognizers
SR3Library' = SR3Lib? // 3-recognizers
LoopFunction' = MinimalTotalRelation

The loop takes on the object of the extraction; the libraries are initialized to the known recognizers of the given length. As for the relation that is used to approximate the loop function, we initialize it to the least defined total relation. By virtue of our refinement ordering, the minimal total relation is the universal relation, represented by predicate $\mathbf{true}$.

4.4 Operations

As we have discussed in section 2, we have identified three sources of information that can be used to derive lower bounds (in the refinement ordering) of the loop function: Using termination conditions; using non-surjective loop bodies; and using invariant functions. Invariant functions provide typically most of the functional details, and pose the greatest challenge to deploy. We discuss three steps in the derivation of lower bounds by means of invariant functions.

- Functions `match1`, `match2` and `match3` match combinations of statements from the loop body against existing patterns of length 1, 2 or 3, with-out worrying about permutations (this will be done by the calling procedure).

- Function `match` will coordinate calls to `match1`, `match2` and `match3` to combine their collective capability.
- After function `match` completes, the variable `Relation` contains the loop function, or failing that (if the relation is non-deterministic), a relation that is known to be refined by the loop. We can then simplify the relation and/or, if it is deterministic, turn it into a CCA. .

We depart here from strict Z notation, because we really want to articulate algorithms rather than simply dictate outcomes. We use a straightforward C-like algorithmic notation.

5 Illustrative Example

To illustrate the extraction algorithm, we consider the following loop, where variables x, y and z are of type integer:

```
while x>5 do
   {x:= x-3,
    y:= y+5,
    z:= z+4*x
   }
```

In terms of the Z specification presented above, this loop is defined by its three components:

Space	`x, y, z:  int`
Cond	`x>5`
Body	`x:=x-3, y:=y+5, z:=z+4*x`

For simplicity, we consider that `SR1Library` contains one Semantic Recognizer, represented in Figure 5: Also, we consider that `SR2Library` contains two Semantic Recognizers, represented in Figure 6: We now follow the extraction routine as it proceeds through the execution of the `extract` function; the result of executing this function is recorded in `Relation`. First, we observe that the loop body defines a surjective function (this can be automated using theorem proving technology, though there are ample shortcuts). Whence we initialize variable `Relation` (represented by its characteristic predicate) to **true** . We find

$$Relation = \{(s, s')|\mathbf{true}\ \}.$$

Because the condition of the loop is an inequality ($x > 5$), we perform the combination

```
Relation := Relation ∩ BCRelation,
```

where

$$BCRelation = I(t) \circ L \circ I(t) \circ [B] \circ I(\neg t) \cup I(\neg t),$$

where t is the loop condition, which merely says that the final state of the loop satisfies $x \leq 5$ and that the state immediately before it (in the iteration process) satisfies $x > 5$. Because B decrements x by 3, we find

$$BCRelation = \{(s, s')|x > 5 \wedge x' \leq 5 \wedge x' + 3 > 5\}$$
$$\cup I(x \leq 5)\}.$$

This can be simplified into

$$BCRelation = \{(s, s')|x > 5 \wedge 2 < x' \leq 5\} \cup I(x \leq 5)\}.$$

Taking the join (in this case, the intersection since the two relations are total) with `Relation` and placing the result in `Relation` yields:

$$Relation = \{(s, s')|x > 5 \wedge 2 < x' \leq 5\} \cup I(x \leq 5)\}.$$

This value of `Relation` gets passed to function `IF-match`, along with the loop. This function calls `match1`, which matches the first line of loop body against the (sole) semantic recognizer of `SR1Library`, yielding after instanciation:

$$T_1 = \{(s, s')|x \ \mathbf{mod}\ 3 = x' \ \mathbf{mod}\ 3 \wedge x' \leq 5\}.$$

Taking the join (intersection) with the current value of `Relation`, we find

$$Relation = \{(s, s')|x > 5 \wedge x \ \mathbf{mod}\ 3 = x' \ \mathbf{mod}\ 3 \wedge x' \leq 5\}$$
$$\cup I(x \leq 5).$$

Function `match1` also matches the second line of the loop body against the sole semantic recognizer of `SR1Library`, yielding, after taking the join:

$$Relation = \{(s, s')|x > 5 \wedge x \ \mathbf{mod}\ 3 = x' \ \mathbf{mod}\ 3$$
$$\wedge y \ \mathbf{mod}\ 5 = y' \ \mathbf{mod}\ 5 \wedge x' \leq 5\} \cup I(x \leq 5).$$

Because these are the only two matches, this completes the for loop that calls `match1`. We now consider the for loop that calls `match2`. This loop finds two matches: A match between the first and second line of the loop body and the first semantic recognizer of `SR2Library`, which produces (after instantiation and simplification) the following relation:

$$T_2 = \{(s, s')|3y + 5x = 3y' + 5x' \wedge x' \leq 5\}.$$

```
 function match1 (in statement: CCATYPE; in recognizer:  SR1TYPE;
  in LoopCond:  EXPRESSIONTYPE;
                 in, out Relation:  RELATIONTYPE);
 function match2 (in st1, st2: CCATYPE; in recognizer:  SR2TYPE;
  in LoopCond:  EXPRESSIONTYPE;
                 in, out Relation:  RELATIONTYPE);
  // attempt to match st1, st2 in this order
 function match3 (in st1, st2, st3: CCATYPE; in recognizer:  SR3TYPE;
  in LoopCond:  EXPRESSIONTYPE;
  in, out Relation:  RELATIONTYPE);
  // attempt to match st1, st2, st3 in this order
 function IFmatch  (in Loop:  LOOPTYPE;
                   in, out:  Relation:  RELATIONTYPE)
    //  uses invariant functions to derive lower bounds
    //  for the loop function
    {for all (st: CCATYPE in Loop.Body)
{for all (Sr1: SR1TYPE in SR1Library)
    {match1(st,Sr1,Loop.Cond,Relation);
     // if there is a match, corresponding instantiated
     // relation is added to Relation
        }   }
     for all (st1, st2:  CCATYPE in Loop.Body)
{for all (Sr2:  SR2TYPE in SR2Library)
    {match2(st1,st2,Sr2,Loop.Cond,Relation);
     match2(st2,st1,Sr2,Loop.Cond,Relation);
}   }
     for all (st1, st2, st3:  CCATYPE in Loop.Body)
{for all (Sr3:  SR3TYPE in SR3Library)
    {match3(st1,st2,st3,Sr3,Loop.Cond,Relation);
     match3(st2,st1,st3,Sr3,Loop.Cond,Relation);
     match3(st1,st3,st2,Sr3,Loop.Cond,Relation);
     match3(st2,st3,st1,Sr3,Loop.Cond,Relation);
     match3(st3,st1,st2,Sr3,Loop.Cond,Relation);
     match3(st3,st2,st1,Sr3,Loop.Cond,Relation);
     // all six permutations
     }   }  }
 function extract (in Loop:  LOOPTYPE;
   out Relation:  RELATIONTYPE)
    //  uses IFmatch and other means to derive
    //  the loop function by successive approximations.
    {if surjective(Loop.Body)
         {Relation := MinimalTotalRelation;}
     else
 {Relation := NSRelation;}
     if inequality(Loop.Cond)
 {Relation := Relation join BCRelation;}
     IFmatch(Loop, Relation);
     simplify(Relation); // perhaps turn it into CCA
    }
```

Figure 4: Outline of an Extraction Algorithm

Cca	Relation
x:=x+a	$\{(s,s')\|x \textbf{ mod } \|a\| = x' \textbf{ mod } \|a\| \wedge \neg Cond'\}$

Figure 5: A 1-Semantic Recognizer

Cca1	Cca2	Relation
x:=x+a	y:=y+b	$\{(s,s')\|ax - by = ax' - by' \wedge \neg Cond'\}$
x:=x+a	y:=y+bx	$\{(s,s')\|y - b \times \frac{x(x-a)}{2\times a} = y' - b \times \frac{x'(x'-a)}{2\times a} \wedge \neg Cond'\}$

Figure 6: Two 2-Recognizers

We also find a match between the first and third line of the loop body and the second semantic recognizer of SR2Library, which produces (after instantiation) the following relation:

$$T_3 = \{(s,s')|z+2\times\frac{x(x+3)}{3} = z'+2\times\frac{x'(x'+3)}{3}\wedge x' \leq 5\}.$$

Taking the join of T_2 and T_3 with the current value of Relation yields

$$Relation = \{(s,s')|x > 5 \wedge x \textbf{ mod } 3 = x' \textbf{ mod } 3\wedge$$
$$y \textbf{ mod } 5 = y' \textbf{ mod } 5 \wedge 3y + 5x = 3y' + 5x'\wedge$$
$$z + 2 \times \frac{x(x+3)}{3} = z' + 2 \times \frac{x'(x'+3)}{3}\wedge$$
$$2 < x' \leq 5\} \cup I(x \leq 5).$$

These are the only two matches that will succeed with SR2Library, hence this terminates the second for loop in IFmatch. The third for loop does not run since we do not have any 3-line patterns. Hence IFmatch returns Relation as written above. Function simplify(Relation) transforms it into:

$$Relation = \{(s,s')|x > 5 \wedge x' = 3 + x \textbf{ mod } 3\wedge$$
$$y' = y + 5(x \textbf{ mod } 3 - 1)\wedge$$
$$z' = z+2x-12(1+x \textbf{ mod } 3)+2(x\div 3)(x+x \textbf{ mod } 3)\}.$$

6 Conclusion

Computing the function of a while loop in closed form is a non trivial problem, not only because of the difficulty of building the necessary inductive argument, but also because of the need to cast the result in terms the user can relate to. Much of the functional attributes of the loop are derived through the use of invariant functions, which are obtained by matching loop body statements against precatalogued patterns that we call *semantic recognizers*. The most critical success factor in this effort is the construction and use of the semantic recognizer infrastructure; we envision that this infrastructure will evolve in the future to cover more and more loops, and more and more functional aspects of loops. In this paper we have presented the broad outlines of an algorithm that derives loop functions by successive approximations, by accumulating independent items of information about the loop function, and composing them into an aggregate relation. If the resulting relation is total and deterministic, then it is the function of the loop. If not, then this relation represents the most information we can collect about the loop at hand, on the basis of the available recognizer structure.

Perspectives of further research include, in addition to evolving the recognizer infrastructure: means to integrate ADT axiomatizations into the (algorithms-oriented) semantic recognizers, thereby enabling us to handle programs that manipulate advanced data types; means to integrate domain knowledge into the loop extraction machinery, to enable us to present the results in a form that is meaningful to the user; means to support post-extraction user interactions, to enable us to answer queries about the loop function.

References

[1] Utpal Banerjee. *Loop Transformations for Restructuring Compilers*. Kluwer Academic Publishers, Boston, MA, 1993.

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

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

[4] Rosann W Collins, Gwendolyn H. Walton, Alan R Hevner, and Richard C Linger. The CERT function extraction experiment: Quantifying FX impact on software comprehension and verification. Technical Report CMU/SEI-2005-TN-047, Software Engineering Institute, Carnegie Mellon University, December 2005.

[5] M. A. Colon, S. Sankaranarayana, and H. B. Sipna. Linear invariant generation using non linear constraint solving. In *Proceedings, Computer Aided Verification, CAV 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer Verlag, 2003.

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

[7] Thomas Fahringer and Bernhard Scholz. *Advanced Symbolic Analysis for Compilers*. Springer Verlag, Berlin, Garmany, 2003.

[8] Alan R Hevner, Richard C Linger, Rosann W Collins, Mark G Pleszkoch, Stacy J. Prowell, and Gwendolyn H Walton. The impact of function extraction technology on next generation software engineering. Technical Report CMU/SEI-2005-TR-015, Software Engineering Institute, July 2005.

[9] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

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

[11] T. Jebelean L. Kovacs. An algorithm for automated generation of invariants for loops with conditionals. In D. Petcu et. al., editor, *Proceedings of the Computer-Aided Verification on Information Systems Workshop (CAVIS05), 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC05)*, pages 16–19, Department of Computer Science, West University of Timisoara, Romania, 2005.

[12] Richard C. Linger, Gwendolyn Walton, Alan Hevner, and Luanne Burns. Next-generation software engineering: Function extraction for computation of software behavior. In *Proceedings, Hawaii International Conference on System Sciences, HICSS-40*, Kona, Hawaii, 2007. IEEE Computer Society Press, Los Alamitos, CA.

[13] Ali Mili, Jules Desharnais, and Jean Raymon Gagne. Strongest invariant functions: Their use in the systematic analysis of while statements. *Acta Informatica*, April 1985.

[14] Ali Mili, Mark Pleszkoch, and Richard C. Linger. Towards the automated derivation of loop functions. Technical report, New Jersey Institute of Technology, http://web.njit.edu/ mili/loopx.pdf, 2006.

[15] H.D. Mills. The new math of computer programming. *Communications of the ACM*, 18(1), January 1975.

[16] Mark Pleszkoch, Phillip Hausler, Alan Hevner, and Richard C Linger. Function-theoretic principles of program understanding. In *Proceedings of the 23rd Annual Hawaii International Conference on System Science (HICSS35)*, Hawaii, 1990. IEEE Computer Society Press, Los Alamitos, CA.

[17] Mark Pleszkoch and Richard C. Linger. Improving network system security with function extraction technology for automated calculation of program behavior. In *Proceedings, Hawaii International Conference on System Sciences*, Los Alamitos, CA, January 2004. IEEE Computer Society Press.

[18] S.J. Prowell, C.J. Trammell, R.C. Linger, and J.H. Poore. *Cleanroom Software Engineering: Technology and Process*. SEI Series in Software Engineering. Addison Wesley, 1999.

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

[20] J.M. Spivey. *The Z Notation —A Reference Manual*. Prentice Hall, London, UK, 1998.