Goal-oriented test data generation for programs with pointer variables

Arnaud Gotlieb Tristan Denmat
IRISA / INRIA
35042 Rennes Cedex, France
{*Arnaud.Gotlieb,Tristan.Denmat*}*@irisa.fr*

Bernard Botella
THALES AEROSPACE
78851 Elancourt Cedex, France
Bernard.Botella@fr.thalesgroup.com

Abstract

Automatic test data generation leads to the identification of input values on which a selected path or a selected branch is executed within a program (path-oriented vs goal-oriented methods). In both cases, several approaches based on constraint solving exist, but in the presence of pointer variables only path-oriented methods have been proposed. This paper proposes to extend an existing goal-oriented test data generation technique to deal with multi-level pointer variables. The approach exploits the results of an intraprocedural flow-sensitive points-to analysis to automatically generate goal-oriented test data at the unit testing level. Implementation is in progress and a few examples are presented.

1. Introduction

Goal-oriented test data generation leads to the identification of input values on which a selected branch in a program is executed. The presence of pointer variables introduces technical difficulties making the extension of current goal-oriented test data generation methods a challenging task.

What is exactly the problem ? In imperative programs, a dereferenced pointer and a variable may refer to the same memory location at some program point. This is known as the aliasing problem. Sometimes, a dereferenced pointer may be aliased with a variable only if some conditions that depend on the control flow are satisfied. As an example, consider the problem of generating a test datum that activates the then-part of statement 4 in the C code of Fig.1. If the assignment of statement 3 is considered to have no effect on variable i, then the then-part of statement 4 will be declared as unreachable by an automatic test data generator ($i = 10$ and $i < 5$ are contradictory). However, if the flow passes through the then-part of statement 1, then p points to i and then i is assigned to 0 at statement 3. On the contrary, if one suppose that statement 3 can modify any pointed variable in the program, then the test data generation process just suspends as it cannot decide whether $i < 5$ is satisfied or not. Hence, activating the then-part of statement 4 requires $*p$ to be aliased with i and so requires the decision of statement 1 to be satisfied. We call this a *conditional aliasing problem*. Note however that when a path is selected first, the pointing relations are all known and conditional aliasing problems are trivially handled. Hence, only goal-oriented test data generation methods are concerned with conditional aliasing problems. According to our knowledge, prior work

```
1.  if (...) p = &i;
2.  i = 10 ;
3.  *p = 0;
4.  if (i < 5) ...
```

Figure 1. A conditional aliasing problem

on automatic test data generation in the presence of pointers did not address the conditional aliasing problem. Korel [8] proposed exploiting several executions of the program to find a test datum on which a selected path is executed. In [4], the approach was adapted to generate goal-oriented test data by making use of dynamic data flow analysis but it did not suffer from the conditional aliasing problem as it was solely based on program executions. More recently, Visvanathan and Gupta [11], Zhang [12] and Williams et al. [10] addressed the problem of generating test data for C functions with pointers as input parameters by using symbolic execution and constraint solving techniques. In their approaches, pointer relationships are handled by constraints on input values and aliasing problems occur only on input data structures. All these approaches have in common the need for a path to be selected first and so fall in the path-oriented methods category. Unlike path-oriented and among other advantages, goal-oriented methods exploit the early detection of non-feasible paths to narrow the search space made up of all the paths that reach a given branch [4].

Considering all paths that reach a given branch is usually unreasonable as the number of control flow paths is exponential on the number of program decisions or even infinite when loops are unbounded. In [5, 6], we proposed a novel framework that automatically generates goal-oriented test data for the coverage of structural criteria. The underlying method consists in generating a Constraint Logic Program over Finite Domains associated with a C function and solving a CLP goal, obtained by the selection of a given branch. The approach relies on Static Single Assignment (SSA) forms [2] and Constraint Logic Programming (CLP) techniques [7] to generate test data that reach selected branches. Although our method has addressed non-trivial academic and industrial test data generation problems (including loops, arrays, bitwise operations and so on), its incapacity to deal correctly with conditional aliasing problems was considered by us as a major drawback.

This paper gives an overview of the extension of our test data generation method to a restricted class of pointer variables : multi-level pointers toward statically named variables. This class of pointers is the one most often used in real-time control systems where dynamic allocation and unconstrained use of pointers is prohibited.

Outline of the paper. In Section 2, background on our CLP-based test data generation technique is recalled. Section 3 gives an overview of our approach on an example. Section 4 details the SSA form in the presence of pointers while section 5 presents declarative semantics for two specific CLP combinators used to model pointer use and definition. Section 6 reports on the preliminary results.

2. Background

2.1. SSA form

The SSA form is a semantically equivalent version of a program where each variable has a unique definition. Every program can be transformed to SSA by numbering the references and definitions of variables. For example $i = i+1; j = j*i$ is transformed to $i_2 = i_1+1; j_2 = j_1*i_2$. At the junction nodes of the control structures, SSA introduces special assignments called ϕ-functions, to merge the definitions of a given variable : $v_3 = \phi(v_1, v_2)$ assigns the value of v_1 to v_3 if the flow comes from the first branch of the control structure, v_2 otherwise. For convenience, a list of ϕ-functions is written as a single statement over vectors of variables :
$x_2 = \phi(x_1, x_0), .., z_2 = \phi(z_1, z_0) \iff \vec{v_2} = \phi(\vec{v_1}, \vec{v_0})$
where $\vec{v_i}$ denotes a vector $\begin{bmatrix} x_i \\ \dots \\ z_i \end{bmatrix}$.

2.2. The CLP(FD) framework

A *CLP(FD) program* is a set of clauses of the form A:–B where[1] A is a user-defined constraint and B is a goal. A *goal* is a sequence of constraint or combinator calls. *Combinators* are language constructs expressing high-level relations between other constraints and variables in CLP(FD) (called *FD_variables*) take their values in a non-empty finite set of integers.

Informally speaking, the solving process of a CLP(FD) goal is based on constraint propagation, which exploits the constraints to narrow the search space, and a labelling process that enumerates all the remaining values to eventually find a solution to the set of constraints.

Constraint propagation. During this process, constraints and combinators are incrementally introduced into a propagation queue. An iterative algorithm manages each constraint one by one in this queue by filtering the domains of FD_variables of their inconsistent values. When the domain of a FD_variable is narrowed then the algorithm reintroduces into the queue all the constraints where this FD_variable appears (awaken constraints). The algorithm iterates until the queue becomes empty, which corresponds to a state where no more narrowings can be performed. The set of constraints is contradictory if the domain of at least one FD_variable becomes empty during the propagation.

A labelling procedure. As is usually the case with finite domain constraint solvers, constraint propagation does not ensure that the set of constraints is satisfiable. Enumeration must be used to get particular solutions. This labelling procedure tries to give values to FD_variables one by one and propagates them throughout the constraint system. This is done recursively until all the FD_variables are instantiated. If this valuation leads to a contradiction then the procedure backtracks to other possible values.

2.3. Translating into a CLP(FD) programs

The idea behind our test data generation technique consists in translating the imperative program into a CLP(FD) program via the SSA form [5].

First, for each C function, a single clause is generated. The clause takes as arguments several logical variables that correspond to the input variables of the C function and the variables which are used in the decisions of the program. Second, each statement under SSA form is translated into a constraint or a combinator. Assignments and decisions are translated into arithmetical constraints. For example, $x = x + 1$ is converted into $X_2 = X_1 + 1$. For control structures, we introduced in [5, 6] two specific combinators for which only declarative semantics is given below.

[1] In the paper, the Prolog syntax is used for CLP(FD) programs

Original C code	SSA form	CLP(FD) program (*where* $\&j = 21$ and $\&k = 22$)
int foo(**int** i)	**int** foo(**int** i)	**foo**(I, J_4) :–
1. $j = 0, k = 0, p = \&j$;	$j_1 = 0, k_1 = 0, p_1 = \&j$;	$J_1 = 0, K_1 = 0, P_1 = 21,$
2. **if** ($i < 6$) $j = 2$ **else** $p = \&k$;	**if** ($i < 6$) $j_2 = 2$ **else** $p_2 = \&k$;	$\texttt{ite}(I < 6, \begin{bmatrix} J_2 \\ P_1 \end{bmatrix}, \begin{bmatrix} J_1 \\ P_2 \end{bmatrix}, \begin{bmatrix} J_3 \\ P_3 \end{bmatrix}, J_2 = 2, P_2 = 22)$
	$\begin{bmatrix} j_3 \\ p_3 \end{bmatrix} = \phi(\begin{bmatrix} j_2 \\ p_1 \end{bmatrix}, \begin{bmatrix} j_1 \\ p_2 \end{bmatrix})$;	
fi	**fi**	
3. $r = *p$;	$r_1 = \phi_u(p_3, \begin{bmatrix} \&j \\ \&k \end{bmatrix}, \begin{bmatrix} j_3 \\ k_1 \end{bmatrix})$	$R_1 = \Phi_u(P_3, \begin{bmatrix} 21 \\ 22 \end{bmatrix}, \begin{bmatrix} J_3 \\ K_1 \end{bmatrix}),$
4. $*p = r * i$;	$\begin{bmatrix} j_4 \\ k_2 \end{bmatrix} = \phi_d(p_3, \begin{bmatrix} \&j \\ \&k \end{bmatrix}, r_1 * i, \begin{bmatrix} j_3 \\ k_1 \end{bmatrix})$;	$R_2 = R_1 * I,$
		$\begin{bmatrix} J_4 \\ K_2 \end{bmatrix} = \Phi_d(P_3, \begin{bmatrix} 21 \\ 22 \end{bmatrix}, R_2, \begin{bmatrix} J_3 \\ K_1 \end{bmatrix}),$
5. **if** ($j > 8$) ...	**if** ($j_4 > 8$)...	$\texttt{ite}(J_4 > 8, ...),$

Figure 2. The foo example, its SSA form and the generated CLP(FD) program

Conditional statement. The conditional statement is treated with a user–defined combinator `ite` adapted from [6]. Arguments of `ite` are the variables that appear in the ϕ-functions and the constraints generated from the then– and the else– parts of the statement. Note that other combinators may be nested in the arguments of `ite`. An SSA **if** statement : **if** (*exp*) { *stmt* } **else** { *stmt* } $\vec{v_2} = \phi(\vec{v_0}, \vec{v_1})$ is converted into $\texttt{ite}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Then}, C_{Else})$ where c is a constraint generated by the analysis of *exp* and C_{Then} (resp. C_{Else}) is a set of constraints generated for the then–part (resp. else–part). The user-defined combinator `ite` is :

Definition 1 `ite` *(declarative semantics)*
$\texttt{ite}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Then}, C_{Else}) : -$
$(c \wedge C_{Then} \wedge \vec{v_2} = \vec{v_0}) \vee (\neg c \wedge C_{Else} \wedge \vec{v_2} = \vec{v_1})$

Iterative statement. The SSA **while** statement $\vec{v_2} = \phi(\vec{v_0}, \vec{v_1})$ **while** (*exp*) { *stmt* } is treated with the recursive user-defined combinator $\texttt{w}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Body})$, adapted from [6]. When evaluating `w`, it is necessary to allow the generation of new constraints and new variables with the help of a substitution mechanism. `w` is defined as[2] :

Definition 2 `w` *(declarative semantics)*
$\texttt{w}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Body}) : -$
$(c \wedge C_{Body} \wedge \texttt{w}(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, C_{Body})) \vee (\neg c \wedge \vec{v_2} = \vec{v_0})$

Note that the vector $\vec{v_3}$ is a vector of fresh variables.

Goal-oriented Test data generation. The selection of a branch in the C function defines a CLP goal. Control-dependencies, which are decisions that must be evaluated to "true" to reach a selected branch, are used to build the CLP goal. In well–structured programs (without goto statement), they can easily be computed even if they must be determined dynamically for the loop statements [6]. In the example of Fig.1, the control-dependency associated with the then-part of statement 4 is just $i < 5$. The last phase of the test data generation process consists in solving the resulting CLP goal by using the techniques described in section 2.2. As the semantics is modeled faithfully, any solution of the CLP request is interpreted as a test datum that reaches the selected branch. When the solving process shows that there is no solution, the selected branch is declared unreachable. This approach has been implemented in the INKA tool [1] and evaluated on a set of academic and reasonably-sized industrial problems.

3. An overview of the approach

Consider the problem of generating a test datum that executes the then-part of statement 5 in the foo program of Fig.2. Our goal-oriented test data generation process is composed of three main steps. The first step aims at generating the SSA form of the C code, which is given in the second column of Fig.2. The definition of SSA in the presence of pointer variables is mainly based on two ideas :

1. First, our approach exploits the results of a specific pointer analysis, namely a points-to analysis, to perform all the hidden definitions. A points-to analysis is a static analysis that determines the set of memory locations that can be accessed through pointer dereferences. For every variable p of pointer type, a points-to analysis computes a set of variables that may be pointed by p during the execution. For example, at statement 4 of program foo, a *points-to analysis* says that p can (only) point to j or k. Note that the analysis usually overestimates the set of pointing relations that exist during execution.

[2] For the sake of clarity, the constraint c generated through the substitution mechanism is not distinguished from c itself

2. Second, we introduce two new forms of ϕ–functions to model the dereferencing process. ϕ_u–functions model uses of dereferenced pointers. At statement 3, $\phi_u(p_3, \begin{bmatrix}\&j\\ \&k\end{bmatrix}, \begin{bmatrix}j_3\\ k_1\end{bmatrix})$ returns j_3 (resp. k_1) if p points to j (resp. k). ϕ_d–functions are used to reveal the hidden definitions realized through dereferenced pointers. At statement 4, $\begin{bmatrix}j_4\\ k_2\end{bmatrix} = \phi_d(p_3, \begin{bmatrix}\&j\\ \&k\end{bmatrix}, r_1 * i, \begin{bmatrix}j_3\\ k_1\end{bmatrix})$, assigns $r_1 * i$ to j_4 (resp. k_2) if p points to j (resp. k) and assigns j_3 (resp. k_1) otherwise.

The second step of our approach translates the SSA form into a CLP(FD) clause as shown in the third column of Fig.2. In this translation, each variable's address is associated with a unique key[3] and specific CLP(FD) combinators extend ϕ_u and ϕ_d functions. These combinators maintain a **relation between their arguments**. So, partial information such as the variation domain of an argument, can be exploited to narrow the domain of the others.

Finally, the last step consists in generating a test-data-generation request by making use of the control-dependencies of the program. Reaching the then-part of statement 5 requires $J_4 > 8$ hence the request shown in Fig.3 is generated. 1 In this example, the result of the re-

```
?- J4 > 8, foo(I, J4).
  I = 5 ;              /* first solution and backtracks */
  no                   /* no other solution */
```

Figure 3. A test data generation request

quest says that there exists only a single test datum ($i = 5$) satisfying the request. If we examine the resolution process, we see that the three constraints $J_3 \in \{0, 2\}$, $J_4 > 8$ and $\begin{bmatrix}J_4\\ K_2\end{bmatrix} = \Phi_d(P_3, \begin{bmatrix}21\\ 22\end{bmatrix}, R_2, \begin{bmatrix}J_3\\ K_1\end{bmatrix})$ lead to $P_3 = 21$ and $J_4 = R_2$. As a consequence, $P_3 = P_2$ is refuted and the then-part of statement 2 must be executed, leading to $I < 6$. Finally, the constraints $R_1 = J_3$ and $R_2 = R_1 * I$ implies $I > 4$ which ends the process.

The interesting point is that the combinator Φ_d provokes the assignment of the pointer variable P_3. In this example, numeric information over integer variables is used to refine pointer relationships.

4. SSA in the presence of multi-level pointers

In this paper, we confine ourselves to a simple language over the pointers based on multi-level pointers toward statically named variables. The operations that are allowed on pointers are (multiple) dereferencing ($**p$), addressing ($\&q$), pointer assignment ($p = q$), and pointer comparison ($p == q$, $p != q$). We suppose that programs do not contain unconstrained pointer arithmetic, type casting through pointers, pointers to functions and pointers to dynamically allocated structures. Furthermore, this paper is devoted to the treatment of pointers in the context of automated testing of C programs at the unit level, meaning that function calls are supposed to be stubbed or inlined. Extension to the handling of function calls in the presence of conditional aliasing problems is not trivial and is discussed in Sec.7.

4.1. Normalization

Normalizing a function consists in breaking complex statements into a set of elementary statements by introducing temporary variables. In [3], it is shown that C programs that respect the previous hypothesis, can be translated into a set of fifteen elementary statements. In particular, a multi-level dereferenced pointer can be translated into a set of single dereferenced pointer by introducing temporary variables without modifying the program semantics. Fig. 4 contains a few examples of normalization that can easily be generalized to other statements. Note however that normalization is not required when a statement holds over non-pointer types (for example, $*p = *q$ does not need to be normalized if p and q are of pointer–to-integer type).

Original code	Normalized code
$p = ***q$;	$tmp_1 = *q$; $tmp_2 = *tmp_1$; $p = *tmp_2$;
$**p = q$;	$tmp_1 = *p$; $*tmp_1 = q$;
$*p = \&q$;	$tmp_1 = \&q$; $*p = tmp_1$;
$*p = *q$;	$tmp_1 = *q$; $*p = tmp_1$;

Figure 4. Examples of normalization

This normalization process allows to reason on a small number of statements without any loss of generality. For the presentation, only four assignment statements are considered : $p = \&q$, $p = q$, $p = *q$, $*p = q$.

4.2. A *Points–to* analysis

As previously said, a *points–to analysis* statically collects a set of variables that may be pointed by the pointers of the program and determines the set of memory locations that can be accessed through a dereferenced pointer. In our work, we have chosen a points-to analysis formerly introduced by *Emami et al.* [3]. A points-to relation is a triple : $pto(p, a, definite)$ or $pto(p, a, possible)$ where a denotes a variable pointed by p. In the former case, p points definitely to a on any control flow path that reaches the statement where the pointing relation has been computed. In the latter case, p may point to a only on some control flow paths. In fact, the analysis does not even say whether there exists a feasible control flow path that contains the pointing relation. Although it can be very imprecise, a points-to

[3] A variable's address is noted $\&j$ even when j is decomposed in several SSA names $j_0, j_1, \ldots$ as $\&j_0, \&j_1, \ldots$ represent the same constant.

analysis is always conservative, meaning that if p points-to a during any execution of the program then the results of the *points–to analysis* contains at least $pto(p, a, rel)$ where rel is either $definite$ or $possible$.

There are two kinds of *points–to analysis* : flow-sensitive and flow-insensitive. In the former case, the order in which the statements are executed is taken into account and the analysis is computed on each statement of the program. In the second case, the order is just ignored and the results of the points-to analysis are the same for all the statements. A flow-sensitive analysis is usually more precise than a flow-insensitive but it is also more costly to compute. Fig.5 shows the difference between these two analyses on a very small piece of C code. In our approach, we use a

C Code	Flow-sensitive at statement 3	Flow-insensitive
1. $p = \&a$;		pto(p,a,possible)
2. $q = p$;		pto(p,b,possible)
3. $p = \&b$;	pto(p,b,definite)	pto(q,a,possible)
	pto(q,a,definite)	pto(q,b,possible)

Figure 5. *Points-to analysis*

flow-sensitive analysis. In fact, when a statement contains a definition of a dereferenced pointer, every pointing relation hides a possible definition, hence the precision of the analysis directly plays on the efficency of the overall approach.

In [3], an intraprocedural syntax-based algorithm that computes the results of a flow-sensitive *points–to analysis* for structured C programs is given. In the presence of control flow structures, the results of the analysis of each branch are merged into a single set. In this process, a definite *points–to* relation can become a possible one. For the loop statements, the set is computed by a fixpoint computation. The analysis is done by iterating on the body of the loop until no more modification can be exercised. Existence and unicity of the fixpoint is trivial as the merge process cannot remove pointing relations and the number of such relations is bounded, as dynamic allocation of pointer variables is forbidden.

4.3. ϕ_u– and ϕ_d– functions in SSA

In our SSA form, the ϕ_u–function models the use of a dereferenced pointer. Let $a_1, ..., a_n$ and $v_1, ...v_n$ be variables, let $\&a_1, ..., \&a_n$ denote the distinct addresses of the first set of variables and let p be a pointer variable, then $\phi_u(p, \begin{bmatrix} \&a_1 \\ .. \\ \&a_n \end{bmatrix}, \begin{bmatrix} v_1 \\ .. \\ v_n \end{bmatrix})$ returns v_i iff $p = \&a_i$.
The ϕ_d–function models the definition of a dereferenced pointer. Let $expr$ denotes an expression, then $\phi_d(p, \begin{bmatrix} \&a_1 \\ .. \\ \&a_n \end{bmatrix}, expr, \begin{bmatrix} v_1 \\ .. \\ v_n \end{bmatrix})$ returns a vector $\begin{bmatrix} x_1 \\ .. \\ x_n \end{bmatrix}$ where $x_i = expr$ if $p = \&a_i$ and $x_j = v_j$ for all $j \neq i$. Although very similar in appearance, the two functions differ by their syntactical role : ϕ_u–functions model right hand side usages whereas ϕ_d model left hand side usages.

5. Combinators Φ_u and Φ_d in CLP(FD)

As a result of the SSA translation, the operators '&' and '*' of the C language have been removed and two new functions have been introduced : ϕ_u– and ϕ_d– functions. In the CLP(FD) program, these functions are modeled by the means of two relational combinators, namely Φ_u and Φ_d. The Φ_u combinator maintains a relation between a pointer, the set of possibly pointed variables and a variable to be assigned. It just exploits the fact that, during execution, a pointer can only point to a single variable.

Definition 3 *Φ_u (declarative semantics)*
Let $X, P, V_1, .., V_n$ be $\mathcal{FD}$ variables and $P_1, .., P_n$ be n be distinct non null constants, then
$X = \Phi_u(P, \begin{bmatrix} P_1 \\ .. \\ P_n \end{bmatrix}, \begin{bmatrix} V_1 \\ .. \\ V_n \end{bmatrix})$ *is true iff* $\exists i | P = P_i \wedge X = V_i$.

The Φ_d combinator maintains a relation between a pointer, a variable associated with the dereferenced pointer, the set of possibly pointed variables, and possibly assigned variables.

Definition 4 *Φ_d (declarative semantics)*
Let $X, P, V_1, .., V_n$ be $\mathcal{FD}$ variables and $P_1, .., P_n$ be n be distinct non null constants, then
$\begin{bmatrix} X_1 \\ .. \\ X_n \end{bmatrix} = \Phi_d(P, \begin{bmatrix} P_1 \\ .. \\ P_n \end{bmatrix}, DP, \begin{bmatrix} V_1 \\ .. \\ V_n \end{bmatrix})$ *is true iff* $\exists i$ *such as* $P = P_i \wedge X_i = DP \wedge \{X_j = V_j\}_{\forall j \neq i}$.

Note that when P is assigned to an invalid address ($P = 0$), then both Φ_d and Φ_u combinators **fail** during the solving process. As usual in CLP, failure is interpreted as unsatisfiability of the set of constraints.

6. Preliminary results

We implemented our approach in the goal-oriented test data generator InKa [1]. The tool automatically generates test data for the coverage of several structural criteria such as all_statements, all_branches, MC/DC. The tool has several other functionalities, such as test coverage measurements, control flow monitoring, test case management, etc. Our implementation includes a pointer analyzer, a SSA form generator and both combinators Φ_u and Φ_d.

To evaluate the approach, we generated test data for C functions extracted from the literature that present conditional pointer aliasing problems. In this paper, we only report the results on the program shown in Fig.6. It is extracted from [9] and presents a conditional aliasing problem with two-level indirection pointers. The points-to relations

Normalized C Code	SSA form
int lh98(**int** h)	**int** lh98(**int** h_0)
int $g, **p, *q, *r$;	**int** $g, **p, *q, *r$;
1. $g = 3, q = \&h$;	$g_1 = 3, q_1 = \&h$;
2. $r = \&g, p = \&r$;	$r_1 = \&g, p_1 = \&r$;
3. **if**($h < 10$)	**if**($h_0 < 10$)
4. $g = (h + 2) * 5$;	$g_2 = (h_0 + 2) * 5$
5. $p = \&q$;	$p_2 = \&q$;
	$\begin{bmatrix} g_3 \\ p_3 \end{bmatrix} = \phi(\begin{bmatrix} g_2 \\ p_2 \end{bmatrix}, \begin{bmatrix} g_1 \\ p_1 \end{bmatrix})$;
fi	**fi**
6. $t = *p$;	$t_1 = \phi_u(p_3, \begin{bmatrix} \&q \\ \&r \end{bmatrix}, \begin{bmatrix} q_1 \\ r_1 \end{bmatrix})$;
	$tmp = \phi_u(t_1, \begin{bmatrix} \&h \\ \&g \end{bmatrix}, \begin{bmatrix} h_0 \\ g_3 \end{bmatrix})$;
7. $h = 2 * g + *t$;	$h_1 = 2 * g_3 + tmp$;
8. **if**($h > 100$) ... ;	**if**($h_1 > 100$) ... ;

Figure 6. SSA form of lh98

computed for program lh98 at statement 8 are given by the following diagram:

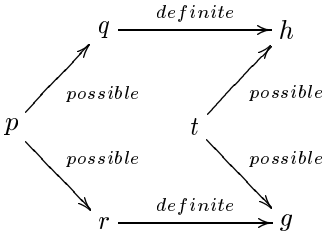

Fig.7 shows the CLP(FD) program generated for lh98 and shows a request asking for a test data generation.

lh98(H_0, H_1):-
$G_1 = 3, Q_1 = 21, R_1 = 22, P_1 = 23,$
ite($H_0 < 10, \begin{bmatrix} G_1 \\ P_1 \end{bmatrix}, \begin{bmatrix} G_2 \\ P_2 \end{bmatrix}, \begin{bmatrix} G_3 \\ P_3 \end{bmatrix}, G_2 = (H_0 + 2) * 5 \wedge P_2 = 24$)
$T_1 = \Phi_u(P_3, \begin{bmatrix} 24 \\ 23 \end{bmatrix}, \begin{bmatrix} Q_1 \\ R_1 \end{bmatrix})$,
$TMP = \Phi_u(T_1, \begin{bmatrix} 21 \\ 22 \end{bmatrix}, \begin{bmatrix} H_0 \\ G_3 \end{bmatrix})$
$H_1 = 2 * G_3 + TMP$,
ite($H_1 > 100$, ...).

?- $H_1 > 100$, lh98(H_0, H_1), $labelling([H_0])$.
$H_0 = 8$;
$H_0 = 9$;
no

Figure 7. CLP(FD) program for lh98

The results show that there are only two values for H_0 able to reach the then-part of statement 8.

7. Conclusion

In this paper, we have presented a new method for automatically generating goal-oriented test data for programs with multi-level pointer variables. Several extensions are in progress to adress larger C programs. Firstly, our approach could be extended to function calls by exploiting the results of an interprocedural pointer analysis. Although a lot of work has been carried out in this area, technical problems still need to be solved in order to properly handle multiple function calls, function pointers and recursive calls. Second, our approach could be extended to pointers that address the heap. In the presence of dynamic allocation, the points-to analysis we used do not converge anymore. Hence, it should be replaced by other pointer analysis adapted to dynamic structures. However, dealing with these constructs is yet a challenging problem.

References

[1] Axlog Ingenierie and Thales Airborne Systems. *INKA–V1 User's Manual*, december 2002.

[2] R. Cytron, J. Ferrante, B. Rosen, M. N. Wegman, and F. K. Zadeck. Efficently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. on Prog. Language and Systems*, 13(4):451–490, Oct. 1991.

[3] M. Emami, R. Ghiya, and L. J. Hendren. Context–Sensitive Interprocedural Points–to Analysis in the Presence of Function Pointers. In *Proc. of Programming Languages Design and Implementation*, Orlando, FL, Jun. 1994. ACM.

[4] R. Ferguson and B. Korel. The Chaining Approach for Software Test Data Generation". *ACM Trans. on Software Engineering and Methodology*, 5(1):63–86, Jan. 1996.

[5] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proc. of the Int. Symp. on Software Testing and Analysis (ISSTA'98)*, pages 53–62, Clearwater Beach, FL, USA, March 1998.

[6] A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Proc. of Computational Logic (CL'2000)*, LNAI 1891, pages 399–413, London, UK, July 2000.

[7] P. V. Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). In *LNCS 910*, pages 293–316. Springer Verlag, 1995.

[8] B. Korel. Automated Software Test Data Generation. *IEEE Trans. on Software Engineering*, 16(8):870–879, Aug. 1990.

[9] C. Lapkowski and L. Hendren. Extended SSA Numbering: Introducing SSA Properties to Languages with Multi–level Pointers. In *7th Proc. of the Conference on Compilers Construction (CC'98)*, pages 128–143, Lisbon, Portugal, Mar. 1998. LNCS 1383 Kai Koshimies (Ed).

[10] B. Marre, P. Mouy, and N. Williams. On-the-fly generation of k-path tests for c functions. In *Proc. of the 19th IEEE Int. Conf. on Automated Software Engineering (ASE'04)*, Linz, Austria, September 2004.

[11] S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In *Proc. of the 17th IEEE Int. Conf. on Automated Software Engineering (ASE'02)*, Edinburgh, UK, September 2002.

[12] J. Zhang. Symbolic execution of program paths involving pointer and structure variables. In *Proc. of the 4th Int. Conf. on Quality Software (QSIC'04)*, Braunschweig, Ge, September 2004.