Goal-oriented test data generation for pointer programs

Arnaud Gotlieb, Tristan Denmat, Bernard Botella

Goal-oriented test data generation for pointer programs

Arnaud Gotlieb [a,*], Tristan Denmat [a], Bernard Botella [b].

[a] *IRISA / INRIA Campus Beaulieu 35042 Rennes Cedex, France*

[b] *THALES AEROSPACE 78851 Elancourt Cedex, France*

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. Pointers are responsible for the existence of conditional aliasing problems that usually provoke the failure of the goal-oriented test data generation process. In this paper, we propose an overall constraint-based method that exploits the results of an intraprocedural points-to analysis and provides two specific constraint combinators for automatically generating goal-oriented test data. This approach correctly handles multi-levels stack-directed pointers that are mainly used in C programs. The method has been fully implemented in the test data generation tool INKA and first experiences in applying it to a variety of existing programs are presented.

Key words: Goal-oriented test data generation, Constraint Logic Programming, Static Single Assignment form, pointer variables

1 Introduction

Goal–oriented test data generation leads to identify 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.

* Corresponding author
Email address: Arnaud.Gotlieb@irisa.fr (Arnaud Gotlieb).

```
        int f(int i, int j, int c) {
1.      int* p = &j;
2.      if (c == 1)
3.              p = &i;
4.      i = 0 ;
5.      *p = 1;
6.      if (i != 0)
7.              ...
```

Figure 1. A conditional aliasing problem

What is exactly the problem? In imperative programs, a dereferenced pointer and a variable may refer to the same (stack-based) memory location at some program point (a.k.a. an aliasing problem). This can be due either to a statement in the code where a pointer variable is assigned the address of another variable or to a relation over the pointer input values of a function. In the former case, the dependence may be conditioned by the control flow: a dereferenced pointer may be aliased with a variable only if some conditions that depend on the flow are satisfied. We call this situation a conditional aliasing problem. For example in the C code of Fig.1, `*p` may be aliased to `i` or `j` at statement 5. Consider the problem of generating a test datum that reaches branch 6-7. If the assignment of statement 5 is considered to have no effect on variable i, then the branch 6-7 will be declared as unreachable by an automated test data generator as $i = 0$ and $i \neq 0$ are contradictory. However, this is incorrect if the flow passes through statement 3 as, in this case, p points to i and then i is assigned to 1 at statement 5, which satisfies the decision of branch 6-7. On the contrary, if statement 5 is considered to be able to modify any pointed variable in the program, then the test data generation process suspends as it cannot decide whether $i \neq 0$ is satisfied or not. In this example, it is worth noticing that reaching branch 6-7 requires $c = 1$ to be satisfied, which is a necessary (and sufficent) condition to solve this problem. Note that when a path is selected first, the pointing relations are all known and such conditional aliasing problems are trivially handled, but if the selected path is infeasible then this must be demonstrated before switching to another choice and carrying on the process.

Contributions. In this paper, we propose to extend an existing constraint-based goal-oriented method [1,2] to take into account conditional pointer aliasing problems. Firstly, we propose the definition of a *Static Single Assignment form* (SSA)[3] in the presence of pointer variables. This SSA form integrates the results of an intraprocedural flow–sensitive pointer analysis in order to reveal the hidden definitions realized by dereferenced pointers. In the example of Fig.1, such an analysis says that `p` points either to `i` or `j` at statement 5. Secondly, we proposed the definition of two specific constraint combinators that model the existing relations between dereferenced pointers and variables.

In order to reach branch 6-7 in the example, these combinators correctly entail that $c = 1$. In this paper, we formally present the operational semantics of these combinators under the form of guarded constraints, and we detail our implementation while providing preliminary experimental results. As a consequence, this paper introduces an overall goal-oriented method able to correctly deal with programs that contain conditional pointer aliasing problems and (multi-level) pointers toward stack memory locations. Note however that our approach suffers from the following restriction: it cannot handle accurately dynamic allocated structures when dynamic allocation is placed within the body of an unbounded loop. This restriction should be minimized by the fact that in most critical systems (military and civil avionic, railway and automotive industries, etc.) dynamic allocation is prohibited [4].

Outline of the paper. In Section 2, the background on our constraint-based technique is recalled. Section 3 gives an overview of the approach on a motivating example. Section 4 details the Pointer–SSA form while section 5 presents two specific combinators used to model pointer use and definition. Section 6 reports on the experimental results we obtained with our implementation in the test data generator INKA and Section 7 discusses related work. Finally, Section 8 recalls the contributions of the paper and indicates several perspectives.

2 Background

Constraint-based test data generation. Originally introduced by DeMillo and Offut in the context of mutation testing [5], Constraint-Based Test data generation (CBT) aims at exploiting constraint satisfaction techniques to generate test data able to reach a selected branch in a program under test. The method builds a constraint system associated to a given branch and then tries to solve the system by using domain reduction techniques. Several tools support CBT: the Godzilla system [6,7] exploits a dynamic domain reduction technique to reach mutation operators in Fortran programs, INKA [1,2] uses Static Single Assignment form and Constraint Logic Programming (CLP) techniques over finite domains to generate test data for the structural coverage of C programs, and ATGen [8] exploits symbolic execution for structural coverage of Spark ADA programs. Recently, INKA has been improved to handle floating-point computations [9], function calls and structures [10]. Although these CBT's implementations have proved to be useful to address non-trivial academic and industrial test data generation problems (including loops, arrays, structures, bitwise operations and so on) it is worth noticing that none of them was able to deal correctly with pointer aliasing problems.

In the CBT approach of [1,2], the selection of a branch in the C function leads

to set up a Constraint Logic Programming request built with the control-dependencies [11]. Control-dependencies are decisions that must be evaluated to "true" to reach a selected branch. In well–structured programs (without goto statement), they can easily be computed [12], even if they must be determined dynamically for the loop statements. In the example of Fig.1, the control-dependency associated to branch 6-7 is just $i \neq 0$. In addition, type declarations are translated into domain constraints. For example, using a signed 32-bits integer x as an input variable leads to set up the following domain constraint: $X \in -2^{31}..2^{31}-1$. The last phase of the test data generation process consists in solving the resulting CLP request by using the techniques described in section 2.2. As the semantics of the program is modeled faithfully, a solution of the CLP request is correctly interpreted as a test datum that reaches the selected branch. In cases where the solving process shows that there is no solution, then the selected branch is declared unreachable. This approach has been implemented in the INKA tool [13] and evaluated on a set of academic and reasonably-sized industrial problems [2]. In [14], we also proposed to use this framework to generate test data that violate high-level properties called metamorphic–relations [15].

Our approach is based on the use of Static Single Assignment form [3] and Constraint Logic Programming over finite domains [16]. It is worth noticing that generating test data for reaching a given decision within a program requires to solve the problem of destructive assignment in imperative programming, e.g. to convert $i = i + 1$ into a relation of the form $i_2 = i_1 + 1$ where i_2, i_1 corresponds to distinct variables. In our approach, we proposed to use SSA for such a task several years ago [1].

2.1 SSA form

The SSA form is a semantically equivalent version of a program that respects the following principle : each variable has a unique definition and every use of this variable is reached by the definition. Every program can be transformed into SSA by renaming the uses and definitions of the variables. For example $i = i + 1; j = j * i$ is transformed into $i_2 = i_1 + 1; j_2 = j_1 * i_2$. At the junction nodes of the control structures, SSA introduces special assignments called ϕ-functions, to merge several definitions of the same variable : $v_3 = \phi(v_1, v_2)$ assigns the value of v_1 in v_3 if the flow comes from the first branch of the decision, the value of v_2 otherwise. SSA has been used in several applications area such as optimizing compilers, automatic parallelization, static analysis and automatic test data generation [1,2,17]. For convenience throughout the paper, we will write a list of successive ϕ-functions with a single statement over vectors of variables : $x_2 = \phi(x_1, x_0), .., z_2 = \phi(z_1, z_0) \Longleftrightarrow \vec{v_2} = \phi(\vec{v_1}, \vec{v_0})$ where

$\vec{v_i}$ denotes a vector $\begin{bmatrix} x_i \\ ... \\ z_i \end{bmatrix}$. SSA provides special expressions to handle arrays : $access(a, k)$ which evaluates to the k^{th} element of a, and $update(a_0, j, v)$ which evaluates to an array a_1 which has the same size and the same elements as a_0, except for j where value is v.

2.2 The CLP(FD) framework

Following the definitions of [16], a *CLP(FD) program* is a set of clauses of the form $A :- B$ where A is a user-defined constraint and B is a a sequence of either primitive constraints or combinators calls [1] . Such a sequence is called a *query*. *Primitive constraints* are built with variables, domains, arithmetical operators in $\{ +, -, \times, \setminus \}$ and relations $\{ >, \geq, =, \neq, \leq, < \}$. In general, variables of the CLP(FD) program (called *FD_variables*) take their values into a non-empty finite set of integers.

Combinators are language constructs expressing a high-level relation between other constraints. They can be either built-in or user-defined constraint depending on the CLP(FD) interpreter that is used. For example, the combinator `element`(I, L, V) is built-in in the CLP(FD) library of Sicstus Prolog [18] : it holds if V is the I^{th} element in the list L of FD_variables.

When considered for solving, a CLP(FD) query leads to build dynamically a *constraint system*, which is made of variables, domains and constraints. Note that constraint solving over finite domains is NP-hard hence some approximations are usually performed before going into a brute force approach. Informally speaking, the solving process of a constraint system is based on 1) a constraint propagation mechanism which makes use of the constraints to prune the search space, 2) a constraint entailment mechanism which tries to infer new constraints from existing ones, 3) a labeling process which explores the search tree by making assumptions in order to find solutions to the constraint system.

Constraint propagation. During this process, primitive constraints and combinators are incrementally introduced into a propagation queue. An iterative algorithm considers each constraint one by one into this queue by filtering the domains of FD_variables of their inconsistent values. Filtering algorithms consider usually only the bounds of the domains. When the domain

[1] Throughout the paper, we will use the Prolog syntax for CLP(FD) programs

of a FD_variable is pruned then the algorithm reintroduces in the queue all the constraints where this FD_variable appears (awaked constraints) to propagate this information. The algorithm iterates until the queue becomes empty, which corresponds to a state where no more pruning can be performed (a fixpoint). When selected in the propagation queue, each constraint is added into a *constraint–store* which memorizes all the considered constraints. The constraint–store is contradictory if the domain of at least one FD_variable becomes empty during the propagation.

Constraint entailment. Some constraints are designed to include conditional information. These constraints are defined with the help of *guarded–constraints*, noted $C_1 \longrightarrow C_2$. During constraint propagation, if C_1 is entailed then C_2 is introduced into the propagation queue, allowing so to dynamically enrich the constraint system. When the constraint C_1 is disentailed then the guarded–constraint $C_1 \longrightarrow C_2$ is just removed from the constraint–store. Otherwise, the guarded–constraint is suspended until being awaked by the constraint propagation mechanism.

Variable labeling. As it is usually the case with finite domain constraint solvers, constraint propagation does not ensure that the set of constraints is satisfiable when a fixpoint is reached. One must resort to enumerate to get particular solutions. This labeling procedure tries to give a value to every FD_variable one by one and propagates throughout the constraint system. This is done recursively until all the FD_variables are instantiated. It is noted `labeling`$([X_1, .., X_n])$ where $X_1, .., X_n$ is a n–tuple of FD_variables to instantiate. If this valuation leads to a contradiction then the procedure backtracks to other possible values. The valuation is done according to some strategies of choice of FD_variables and values. A simple one consists in selecting the minimum value of the domain of the first unbounded FD_variable. Of course other more sophisticated strategies can be used.

2.3 Combinators for program analysis

Based on the CLP(FD) framework, our approach consists in translating every statement into a constraint or a specific combinator. In [1], we introduced such combinators for the conditional and the iterative statements. Let us recall in this subsection the semantics of these combinators.

Conditional statement. The conditional statement is treated with the (user–defined) combinator `ite/6` [2] . Arguments of `ite/6` are made of the vari-

[2] where /6 denotes the arity of the combinator

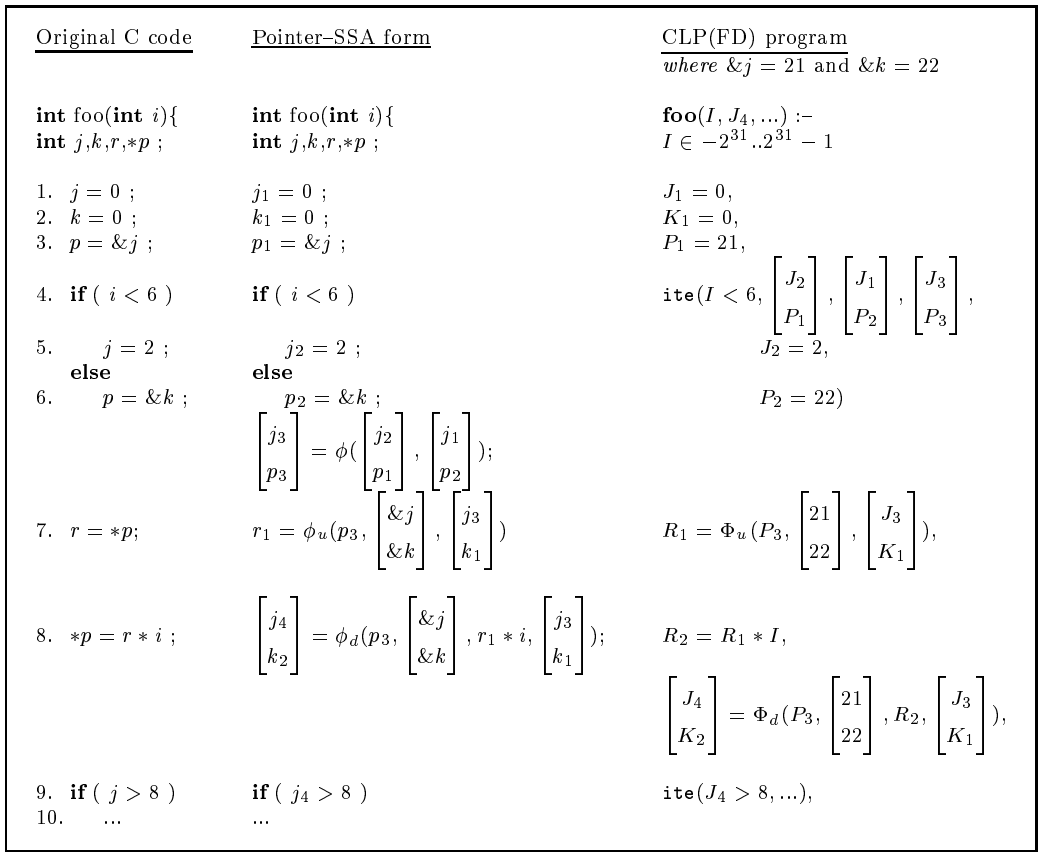


Figure 2. An example, its PSSA form and the generated CLP(FD) program

ables that appear in the ϕ-functions and the constraints generated from the then– and the else– parts of the statement. Other combinators may be nested in the arguments of `ite/6`. An SSA **if_else** 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 primitive–constraint generated by the analysis of *exp*, and C_{Then} (resp. C_{Else}) is a set of constraints generated by the analysis of the then–part (resp. else–part).

Definition 1 `ite/6`

Declarative semantics: $\texttt{ite}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Then}, C_{Else})$ *is true iff*
$(c \wedge C_{Then} \wedge \vec{v_2} = \vec{v_0}) \vee (\neg c \wedge C_{Else} \wedge \vec{v_2} = \vec{v_1})$

Operational semantics: $\texttt{ite}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Then}, C_{Else})$
rewrites to the following guarded-constraints:
$c \longrightarrow C_{Then} \wedge \vec{v_2} = \vec{v_0}$
$\neg c \longrightarrow C_{Else} \wedge \vec{v_2} = \vec{v_1}$
$\neg(c \wedge C_{Then} \wedge \vec{v_2} = \vec{v_0}) \longrightarrow \neg c \wedge C_{Else} \wedge \vec{v_2} = \vec{v_1}$
$\neg(\neg c \wedge C_{Else} \wedge \vec{v_2} = \vec{v_1}) \longrightarrow c \wedge C_{Then} \wedge \vec{v_2} = \vec{v_0}$
$(c \wedge C_{Then} \wedge \vec{v_2} = \vec{v_0}) \uplus (\neg c \wedge C_{Else} \wedge \vec{v_2} = \vec{v_1})$

The former two guarded–constraints result from the operational semantics of the **if_else** statement whereas the three latter allow more effective deductions. Particularly, the last constraint contains the constructive disjunction operator $\uplus$. This operator joins the results computed in both branches of the conditional. For example, if $\texttt{ite}(C, \left[X_0\right], \left[X_1\right], \left[X_2\right], X_0 = 1, X_1 = 3)$ holds then the constructive disjunction operator leads to deduce that $X_2 \in \{1, 3\}$. Note that the `ite/6` combinator is awaked by the solver when the domain of at least one of its variable has changed. For example, learning that $X_2 \geq 2$ prunes the domain of X_2 to $\{3\}$, awakes the combinator and triggers its third guarded–constraint, as the constraint $X_2 \neq X_0$ is entailed by $dom(X_0) \cap dom(X_2) = \emptyset$. Hence, the constraint $\neg C$ is added to the constraint store and the `ite` combinator is removed from it.

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 $\texttt{w}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Body})$. When evaluating `w/5`, it is necessary to allow the generation of new constraints and new variables with the help of a substitution mechanism. `w/5` is defined as [3] :

Definition 2 `w/5`

Declarative semantics: $\texttt{w}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Body})$ *is true iff*
$(c \wedge C_{Body} \wedge \texttt{w}(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, C_{Body})) \vee (\neg c \wedge \vec{v_2} = \vec{v_0})$

Operational semantics: $\texttt{w}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Body})$ *rewrites to*
$c \longrightarrow (C_{Body} \wedge \texttt{w}(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, C_{Body}))$
$\neg c \longrightarrow \vec{v_2} = \vec{v_0}$
$\neg(c \wedge C_{Body}) \longrightarrow (\neg c \wedge \vec{v_2} = \vec{v_0})$
$\neg(\neg c \wedge \vec{v_0} = \vec{v_2}) \longrightarrow (c \wedge C_{Body} \wedge \texttt{w}(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, C_{Body}))$
$(c \wedge C_{Body} \wedge \texttt{w}(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, C_{Body})) \uplus (\neg c \wedge \vec{v_2} = \vec{v_0})$

Note that the vector $\vec{v_3}$ is a vector of fresh variables. The first two guarded–constraints come from the operational semantics of a while in an imperative language. The last two come from the following observations: first, if the constraints extracted from the body are proved to be contradictory with the current constraint system then the loop cannot be entered; second, if any variable possesses distinct values before and after the execution of the **while** statement, then the loop must be entered at least once.

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

3 An overview of the approach

Consider the task of generating a test datum on which branch 9-10 is executed in the C program of Fig.2. The process is composed of three main steps. The first step aims at generating the Pointer SSA form (PSSA), which is given in the second column of Fig.2. The definition of PSSA is mainly based on two ideas:

(1) to exploit the results of a specific pointer analysis, namely a points-to analysis, in order 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 7 of function foo, a *points-to analysis* says that p can (only) points to j or k. Note that the analysis usually overestimates the set of pointing relations that could exist during execution.

(2) to introduce two new forms of ϕ–functions to model the dereferencing process. A ϕ_u–function models the use of a dereferenced pointer: it returns one of its arguments depending on the points-to relations. For example, at statement 7,
$\phi_u(p_3, \begin{bmatrix} \&j \\ \&k \end{bmatrix}, \begin{bmatrix} j_3 \\ k_1 \end{bmatrix})$ returns j_3 if p points to j while it returns k_1 if p points to k.
ϕ_d–functions are used to reveal the hidden definitions realized through dereferenced pointers. At statement 8,
$\begin{bmatrix} j_4 \\ k_2 \end{bmatrix} = \phi_d(p_3, \begin{bmatrix} \&j \\ \&k \end{bmatrix}, r_1 * i, \begin{bmatrix} j_3 \\ k_1 \end{bmatrix})$, assigns $r_1 * i$ to j_4 if p points to j and j_3 otherwise, it assigns $r_1 * i$ to k_2 if p points to k and k_1 otherwise.

The second step of our approach translates the PSSA form into a CLP(FD) clause as shown in the third column of Fig.2. The clause head takes I and J_4 as arguments. I refers to the FD_variable generated for the input variable i, whereas J_4 refers to the variable that determines whether the branch 9-10 is executed or not. In this translation, each variable address is associated to a unique integer, noted $\&j$ for a SSA-variable j_i. ($\&j = 21$, $\&k = 22$ where 21 and 22 correspond to internal symbol table keys [4]) 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 shrink the domain of the others.

[4] Keys from 0 to 20 are reserved to special symbols. For example, 0 represents the NULL pointer

Note that the ϕ_d combinator is related to the *IsAlias* function that was formerly introduced by Cytron and Gershbein [19] to realize hidden definitions in SSA form. Our approach distinguishes by providing relations and not only functions to model the use and definition of dereferenced pointers.

Finally, the last step consists in generating a request by making use of the control-dependencies of the program. Reaching branch 9-10 in the C code of the example requires $J_4 > 8$ hence the request shown in Fig.3 is generated. In

```
?- J4 > 8, foo(I, J4).

 I = 5  ;          /* first solution and forces backtracking */

 no                /* no other solution */
```

Figure 3. A test data generation request

this example, the result of the request 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 $J_3 \in \{0, 2\}$ (deduced by the ite operator), $J_4 > 8$ and $\begin{bmatrix} J_4 \\ K_2 \end{bmatrix} = \Phi_d(P_3, \begin{bmatrix} 21 \\ 22 \end{bmatrix}, R_2, \begin{bmatrix} J_3 \\ K_1 \end{bmatrix})$ entails $P_3 = 21$ and $J_4 = R_2$ as $dom(J_3) \cap dom(J_4) = \emptyset$. As a consequence, $P_3 = P_2$ is refuted and the then–part of the conditional must be executed leading to $I < 6$. Finally, the constraints $R_1 = J_3$ and $R_2 = R_1 * I$ implies $I > 4$ which leads to $I = 5$ as the only solution.

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 The Pointer–SSA form (PSSA)

In this section, we introduce the PSSA form as an extension of SSA where pointer uses and definitions are treated with special functions that are defined with the help of a points-to analysis. Note that our proposition differs from other work where the goal is to exploit SSA in order to improve the accuracy of a points-to analysis [20], as our will is only to preserve the properties of SSA in the presence of conditional aliasing problems.

4.1 A simple language over pointer variables

In this paper, we will confine ourselves to a simple language over pointers: a structured language over multi-level pointers toward statically named variables (stack-directed pointers). In programs that use this class of pointers, the only operations that are allowed on pointers are (multiple) dereferencing (e.g. $**p$), addressing ($\&q$), pointer assignment ($p = q$), and pointer comparison ($p == q$, $p! = q$). Further, we suppose that programs are structured and do not contain unconstrained pointer arithmetic, type casting through pointers, pointers to functions or pointers to dynamically allocated structures. 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.

4.2 Normalization

Normalizing a function consists in breaking complex statements into a set of elementary statements by introducing temporary variables. It is well-known that most C programs can be automatically translated in a program implemented with only a set of fifteen elementary statements [21,22,23]. 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).

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

4.3 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 previously introduced by *Emami et al.* [21]. In this analysis, 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 state-

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

ment where the pointing relation has been computed. In the latter case, p may points to a only on some control flow paths. However, the analysis does not say whether there exists a feasible control flow path that contains the pointing relation. Points-to relations can be captured by labeled directed graph (V, E), called the points-to graph. V denotes the set of vertices that corresponds to the variables of the program, while E denotes the set of labeled edges associated to the points-to relations. There exists an edge between p and a two variables iff $pto(p, a, definite)$ or $pto(p, a, possible)$ is true. Labels can be either symbolic (such as $possible$ or $definite$) or expressions that denote the conditions under which the relation holds. This graph is computed on every statement of the program. Examples of points-to graphs are given in Section 6. Although it can be very inacurate, a points-to analysis is always conservative, meaning that if p points to a during an execution of the program then the results of the *points–to analysis* contains at least $pto(p, a, possible)$.

Points–to analysis can be either flow-sensitive and flow-insensitive, which is just a way to control the cost/accuracy tradeoff of the analysis. In the former case, the order on 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 accurate 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 flow-sensitive analysis for the two following main reasons:

(1) when a statement contains a definition of a dereferenced pointer, every

C Code	Flow-sensitive on 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*

pointing relation hides a possible definition, hence the accuracy of the analysis directly plays on the number of hidden definitions;

(2) efficient algorithms exist for structured C functions.

Our method makes use of the syntax-based algorithm given in [21] to compute a flow-sensitive *points–to analysis*. In this algorithm, every statement can modify the pointing relations by maintaining the set of "killed" relations ($kill\_set$) and the set of relations generated by the statement (gen_set). The notation for both sets makes use of existentially quantified variables, denoted by "$\_x$" in this paper. For example, $\{pto(p, \_x, \_rel) | pto(p, \_x, \_rel) \in In\}$ denotes the set of all pointing relations associated with p in the set In. Fig.6 presents the algorithm for elementary statements. For control flow structures, the results of the analysis on every branch are merged in a single set. In this merge process, a definite *points–to* relation can be transformed into a possible one. For loop statements, a fixpoint is computed by iterating on the body of the loop until no more modification can be exercised. Fig.7 shows the algorithm that handle the basic control flow structures. As the total number of possible points-to relations is bounded in the program (there is no dynamic allocation) and the merge process can only increase the set of points-to relation, existence and unicity of the fixpoint can easily be shown.

4.4 ϕ_u– and ϕ_d– functions in PSSA

In PSSA, ϕ_u–function models the use of a dereferenced pointer. Let $\begin{bmatrix} a_1 \\ .. \\ a_n \end{bmatrix}$ and $\begin{bmatrix} v_1 \\ .. \\ v_n \end{bmatrix}$ denote two vectors of n program's variables, let $\begin{bmatrix} \&a_1 \\ .. \\ \&a_n \end{bmatrix}$ denotes the vector of distinct addresses of the first vector and p be a pointer variable, then the

// Given statement S and In a set of pointing relations
// process_basic(S, In) returns the set of
// pointing relations after S

Points–to process_basic(**Statement** S, **Points–to** In)

Case of
S is of the form $p = \&q$ **then**
$kill\_set := \{pto(p, \_x, \_rel) | pto(p, \_x, \_rel) \in In\}$;
$gen\_set := \{pto(p, q, definite)\}$;

S is of the form $p = q$ **then**
$kill\_set := \{pto(p, \_x, \_rel) | pto(p, \_x, \_rel) \in In\}$;
$gen\_set := \{pto(p, \_a, \_rel) | pto(q, \_a, \_rel) \in In\}$;

S is of the form $p = *q$ **then**
kill_set := $\{pto(p, \_x, \_rel) | pto(p, \_x, \_rel) \in In\}$;
gen_set := $\{pto(p, \_b, \_rel) | pto(q, \_a, \_r_1)$ and
$pto(\_a, \_b, \_r_2) \in In\}$;
If $\_r_1$ and $\_r_2$ are $definite$ **then**
$\_rel = definite$ **else** $\_rel = possible$

S is of the form $*p = q$ **then**
kill_set := $\{pto(\_x, \_y, \_rel) | pto(p, \_x, definite)$ and
$pto(\_x, \_y, \_rel) \in In\}$;
gen_set := $\{pto(\_x, \_z, \_rel) | pto(p, \_x, \_r_1)$ and
$pto(q, \_z, \_r_2) \in In\}$;
If $\_r_1$ and $\_r_2$ are $definite$ **then**
$\_rel = definite$ **else** $\_rel = possible$

returns $(Input \setminus kill\_set) \cup gen\_set$;

Figure 6. Flow–sensitive points–to analysis of basic statements

ϕ_u–function $\phi_u(p, \begin{bmatrix} \&a_1 \\ .. \\ \&a_n \end{bmatrix}, \begin{bmatrix} v_1 \\ .. \\ v_n \end{bmatrix})$ returns v_i if $p = \&a_i$. Note that each $\&a_i$ is a distinct constant. In PSSA, ϕ_d–function models the definition of a dereferenced pointer. A ϕ_d–function $\phi_d(p, \begin{bmatrix} \&a_1 \\ .. \\ \&a_n \end{bmatrix}, expr, \begin{bmatrix} v_1 \\ .. \\ v_n \end{bmatrix})$, returns a vector of variables

```
/* Given statement S and In a set of pointing relations */
/* process(S, In) returns the set of relations after S*/

Points–to process( Statement S, Points–to In)

    If S is void then returns In

    If S is a list of basic statements then
        Head := pop(S) /* returns the head of S */
        Tail := tail(S)  /* returns the tail of S */
        Out := process_basic(Head, In) ;
        returns process(Tail, Out) ;

    If S is of the form [if(C) then S1 else S2]
    then
        Out_then := process(S1,In) ;
        Out_else := process(S2,In) ;
        returns merge(Out_then, Out_else) ;

    If S is of the form [while(C) do S]
    then
        do
            LastIn := In ;
            Out := process(S, In) ;
            In := merge(In, Out) ;
        while LastIn ≠ In
        returns In
```

Figure 7. Flow-sensitive points–to analysis of control structures

$\begin{bmatrix} x_1 \\ .. \\ x_n \end{bmatrix}$ where $x_i = expr$ if $p = \&a_i$ and $x_j = v_j$ for all $j \neq i$.

4.5 Building the PSSA form

A few notations are required to describe the algorithm used to build the PSSA form. When analyzing a statement, $p\downarrow$ denotes the last numbered variable associated to p whereas $p\uparrow$ denotes the new fresh numbered variable for p. So, a definition of p is noted $p\uparrow$ and a use of p is noted $p\downarrow$. If PTO_S denotes the set of pointing relations available at statement S, then $from(p, PTO_S)$ is the set of variables pointed by p. Formally speaking, $from(p, PTO_S) = \{\_a \mid pto(p, \_a, \_) \in PTO_S\}$.

The algorithm that builds the PSSA form works on each statement by generating ϕ_u or ϕ_d functions each time a dereferenced pointer is encountered. Fig.8 contains the algorithm for basic statements obtained after normalization, bearing in mind that other statements can easily be deduced from the treatment of these ones.

```
/* Given statement S, PTO_S a set of pointing relations*/
/* pointer_ssa(S, Input) returns a statement*/
/* to add to the PSSA form */

Statements pointer_ssa(Statement S, Points–to PTO_S)

  Case of
    S is of the form p = *q then
      q⃗ := vector_of_addresses(from(q, PTO_S)) ;
              /* addresses of aliased variables of (*q) */
      v⃗ := vector_of_dereferenced(from(q, PTO_S));
              /* dereferenced aliased variables of (*q) */
      St := (p ↑= φ_u(q ↓, q⃗, v⃗ ↓));

    S is of the form *p = q then
      p⃗ := vector_of_addresses(from(p, PTO_S)) ;
              /* addresses of aliased variables of (*p) */
      v⃗ := vector_of_dereferenced(from(p, PTO_S));
              /* dereferenced aliased variables of (*p) */
              St := (v⃗ ↑= φ_d(p ↓, p⃗, q ↓, v⃗ ↓))

    S is of any other form then
      St := SSA(S) ; /* Standard SSA */
  return St ;
```

Figure 8. Algorithm for constructing PSSA

5 Combinators Φ_u and Φ_d in CLP(FD)

As a result of the PSSA translation, the operators '&' and '*' of the C language have been removed without any loss of semantics. In PSSA, two new functions have been introduced: ϕ_u– and ϕ_d– functions. These functions are translated into two new relational combinators in the CLP(FD) program. The definition of these CLP(FD) combinators is based on guarded–constraints as done for both `ite/6` and `w/5`. The Φ_u combinator maintains a relation between a pointer, the set of possibly pointed variables and a variable to be assigned.

It exploits the fact that, during an execution, a pointer can only point to a single variable.

Definition 3 $\Phi\_u/4$

Declarative semantics : *Let* $X, P, V_1, .., V_n$ *be FD_variables and let* $P_1, .., P_n$ *be* n *distinct numeric 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* $\exists i | P = P_i \wedge X = V_i$.

Operational semantics : $X = \Phi_u(P, \begin{bmatrix} P_1 \\ .. \\ P_n \end{bmatrix}, \begin{bmatrix} V_1 \\ .. \\ V_n \end{bmatrix})$

rewrites to:
$dom(X) := dom(X) \cap (\cup_{i=1}^{n} dom(V_i))$,

if $n = 0$ *then* **fail** *else forall* i *in* $1..n$ *do*

$(P = P_i) \longrightarrow X = V_i$,
$\neg(X = V_i \wedge P = P_i) \longrightarrow P \neq P_i \wedge$

$$X = \Phi_u(P, \begin{bmatrix} P_1 \\ .. \\ P_{i-1} \\ P_{i+1} \\ .. \\ P_n \end{bmatrix}, \begin{bmatrix} V_1 \\ .. \\ V_{i-1} \\ V_{i+1} \\ .. \\ V_n \end{bmatrix}),$$

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

Definition 4 $\Phi\_d/4$

Declarative semantics : *Let* $X, P, V_1, .., V_n$ *be*

FD_variables and $P_1, .., P_n$ *be* n *be distinct numeric constants, then* $\begin{bmatrix} X_1 \\ .. \\ X_n \end{bmatrix} =$

$$\Phi_d(P, \begin{bmatrix} P_1 \\ .. \\ P_n \end{bmatrix}, DP, \begin{bmatrix} V_1 \\ .. \\ V_n \end{bmatrix}) \textit{ is true iff } \exists i | P = P_i \wedge X_i = DP \wedge \{X_j = V_j\}_{\forall j \neq i}.$$

Operational semantics: $\begin{bmatrix} X_1 \\ .. \\ X_n \end{bmatrix} = \Phi_d(P, \begin{bmatrix} P_1 \\ .. \\ P_n \end{bmatrix}, DP, \begin{bmatrix} V_1 \\ .. \\ V_n \end{bmatrix})$ *rewrites to:*

$dom(DP) := dom(DP) \cap (\cup_{i=1}^{n} dom(X_i))$,

if $n = 0$ *then* **fail** *else forall* i *in* $1..n$ *do*

$dom(X_i) := dom(X_i) \cap (dom(DP) \cup dom(V_i))$
$\neg(X_i = V_i \wedge P \neq P_i) \quad \longrightarrow (P = P_i \wedge X_i = DP$
$\wedge \{X_j = V_j\}_{j \neq i})$,

$\neg(X_i = DP \wedge P = P_i) \longrightarrow P \neq P_i \wedge X_i = V_i$

$$\wedge \begin{bmatrix} X_1 \\ .. \\ X_{i-1} \\ X_{i+1} \\ .. \\ X_n \end{bmatrix} = \Phi_d(P, \begin{bmatrix} P_1 \\ .. \\ P_{i-1} \\ P_{i+1} \\ .. \\ P_n \end{bmatrix}, DP, \begin{bmatrix} V_1 \\ .. \\ V_{i-1} \\ V_{i+1} \\ .. \\ V_n \end{bmatrix}),$$

When p is assigned to an invalid address, then both Φ_d and Φ_u combinators **fail** during the solving process. As a failure in CLP corresponds to the unsatisfiability of the constraint store, this correctly entails that the current subpath under investigation is non-feasible.

6 Preliminary results

6.1 The InKa tool

We implemented our approach within the test data generator INKA [2,14]. The tool automatically generates test data for the coverage of structural criteria such as all_statements and all_decisions. It handles a non-trivial subset of the C and C++ languages [13]. The tool has several other functionalities, such as test

coverage measurements, control flow monitoring and test cases management. It is mainly developed in Prolog, Java and C and makes use of the clp(fd) library of Sicstus Prolog [18] to solve the test data generation requests. Our current implementation includes a pointer analyzer, a PSSA form generator and the design of both combinators Φ_u and Φ_d.

	Normalized C Code	PSSA form
	int lh98(**int** h)	**int** lh98(**int** h_0)
	int $g, **p, *q, *r$;	**int** $g, **p, *q, *r$;
1.	$g = 3$;	$g_1 = 3$;
2.	$q = \&h$;	$q_1 = \&h$;
3.	$r = \&g$;	$r_1 = \&g$;
4.	$p = \&r$;	$p_1 = \&r$;
5.	**if**($h < 10$)	**if**($h_0 < 10$)
6.	$g = (h + 2) * 5$;	$g_2 = (h_0 + 2) * 5$
7.	$p = \&q$;	$p_2 = \&q$;
		$\begin{bmatrix} g_3 \\ p_3 \end{bmatrix} = \phi(\begin{bmatrix} g_2 \\ p_2 \end{bmatrix}, \begin{bmatrix} g_1 \\ p_1 \end{bmatrix})$;
8.	$t = *p$;	$t_1 = \phi_u(p_3, \begin{bmatrix} \&q \\ \&r \end{bmatrix}, \begin{bmatrix} q_1 \\ r_1 \end{bmatrix})$;
		$tmp = \phi_u(t_1, \begin{bmatrix} \&h \\ \&g \end{bmatrix}, \begin{bmatrix} h_0 \\ g_3 \end{bmatrix})$;
9.	$h = 2 * g + *t$;	$h_1 = 2 * g_3 + tmp$;
10.	if($h > 100$) ;	if($h_1 > 100$) ;
11.	...	...

Figure 9. Pointer–SSA form of lh98

6.2 Experimental evaluation

To evaluate the approach, we generated test data for C functions that contain conditional pointer aliasing problems. In this paper, we report the experimental results on several programs extracted from the literature. Two of them are detailed as they introduce particular technical difficulties for goal-oriented test data generation. The first program, extracted from [22], is shown in Fig.9 along with its PSSA form. It contains an aliasing problem with two-level indirection pointers when one wants to reach branch 10-11. The points-to graph computed for program lh98 at statement 9 by the flow-sensitive points-to analysis is given by the following diagram:

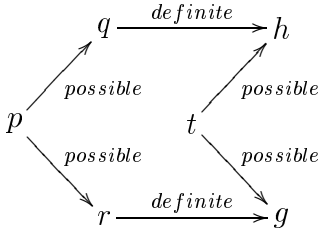


Fig.10 contains the CLP(FD) program generated for function lh98 and the requests asking for test data able to reach branch 10-11. The results show that

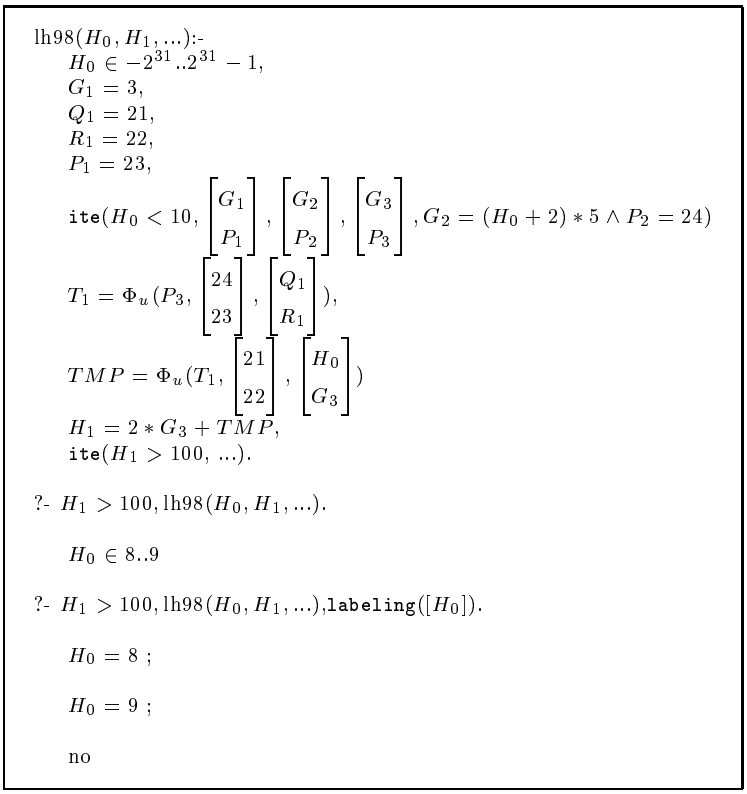
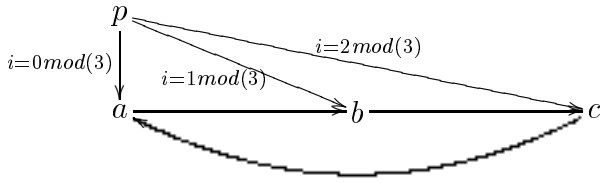


Figure 10. CLP(FD) program for lh98

there are only two values for H_0 able to reach branch 10-11. In this example, the propagation step is so efficient that all the inconsistent values are removed from the domain of H_0, as shown by both requests. The first one gives the results of the propagation step: if there exists a value for H_0 that satisfies the request, then it belongs to 8..9. The second one makes choices and shows that both values (8 and 9) are actual solutions.

In the second program given in Fig.11, pointing relationships are modified within a loop which makes it difficult to analyze. At each iteration, p is assigned the value of $*p$ that has been computed by the previous iteration. As the points-to relations are cyclic in this example (a points-to b, b points-to c and c points-to a), the loop iterates over the possible points-to relations. The points-to graph computed by the flow-sensitive points-to analysis at statement 8 is as follows:



In this diagram, the edges are labeled by the conditions that hold over i in order to satisfy a given points-to relation. For example, p points-to c iff $i = 2mod(3)$. Hence, statement 9 ($p == \&c$ is satisfied) is executed all three loop

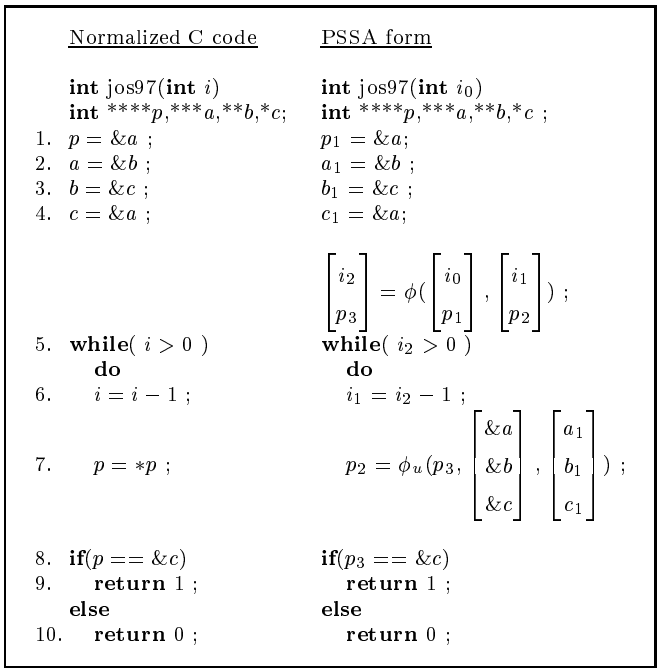


Figure 11. PSSA form of function jos97

iterations, starting from the second one. These conditions were determined manually in order to check our implementation.

Fig.12 contains the CLP(FD) program generated for jos97. The first request asks for a test datum to be generated to reach branch 8-9. It uses only constraint propagation. The second request includes a labeling call. With the first request, the solver prunes the domain of I_0 to $2..2^{31}-1$, which is, as expected, a correct over-estimation of the variation domain of I_0. Note however that not all the values of this domain are solutions of the problem. The low bound of the interval is 2, showing that two non-feasible paths where automatically detected (paths 1-2-3-4-5-8-9 and 1-2-3-4-5-6-7-5-8-9) by the constraint propagation process. The second request gives the correct solutions through the backtracking process of Prolog. The values found by the solver correspond to the values that satisfy the constraint $i = 2mod(3)$. These two requests show that the pointing relations were correctly propagated by the solving process.

In addition to our results for foo (Fig.2), lh98 (Fig.9) and jos97 (Fig.11), test data were generated for several other programs: ow91 extracted from [24], lm03 from [25], bl98 from [26], a version of the famous trityp program [5] that contains conditional pointer aliasing problems and a version of the jos97 program in which the number of points-to relation appears as a parameter and thus, can be easily increased. For each program, we generated a test set that guarantees the complete coverage of the *all_decisions* criterion with InKa. This criterion requires that every decision is covered at least once during the testing phase. We compared our approach with a random test data generation technique which blindly generates test data until a given criterion is covered.

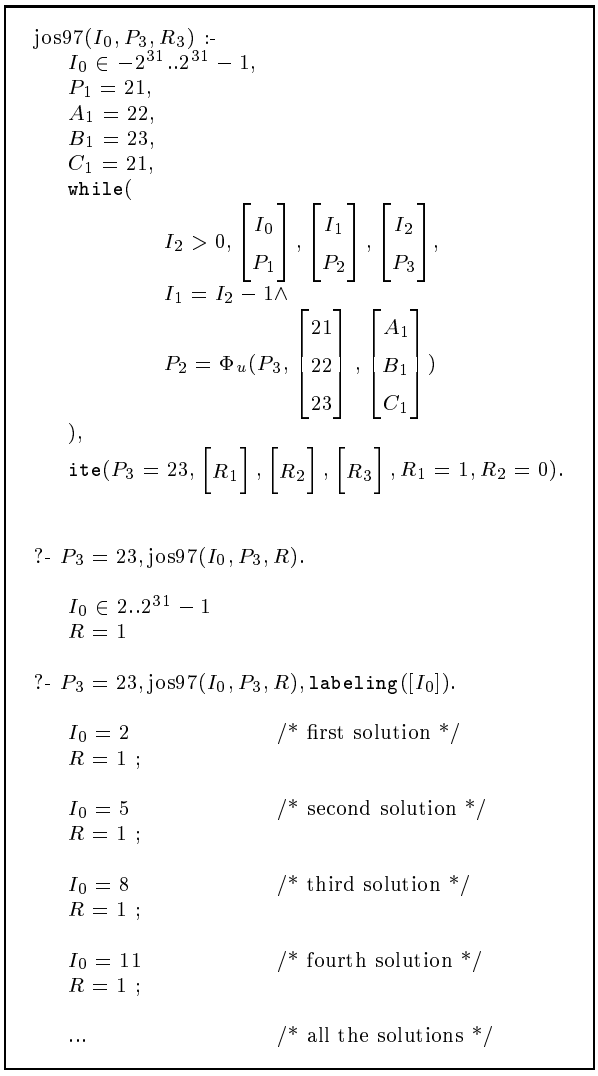


Figure 12. CLP(FD) program for jos97

As random testing can be developed very easily, we argue that every new automated test data generation technique should be shown to outperform random testing. Note however that random testing is unable to show the unfeasibility of a decision, unlike our approach. For random testing, in order to provide a fair comparison with a randomized technique, we also evaluated the *almost_all_decisions* criterion which only requires all but one decisions being covered.

The results given in Tab.1 were computed on a 1.8Ghz Pentium 4 personal computer with 512Mb of RAM. In the left side of the table, we report the number of control flow paths (#p), the number of unfeasible paths (#u) deduced by a manual analysis, and the maximum number of edges in the points-to graphs of the program (#a). These parameters give an insight into the difficulty of conditional pointer aliasing problems in the tested programs. Typically, a path-oriented method would probably fail on a program that contains

a lot of non-feasible paths. We also report two statistics automatically computed by the tool: the number of constraints that are dynamically added to the constraint store (#c) and the number of times a domain is reduced (#pr). These data characterize respectively the size of the constrained problem and the pruning capacity of constraints. We report the results computed by InKa and by the random testing approach. To minimize the development effort and to provide a fair comparison, we used the constrained program model itself to implement random testing. Random numbers were generated inside our tool and tested again the constraint system generated for each program by following a generate–and–test approach. In the random approach, the variation domains were shrinked to -1000 .. 1000 and the time allocated to the evaluation of a single test data was restricted to 4sec of CPU time to avoid long-term computations. In Tab.1, #td corresponds to the number of test data generated for the complete coverage of a criterion, and rt corresponds to the CPU time required to do so. Note that our test data generation strategy is not optimal and so does not lead necessarily to the minimum number of test data required for the full coverage of the criterion. For random testing, we repeated ten times the experience to avoid the factor of bad "luck" introduced by the starting point sequence of random numbers. The values that are shown in the latter columns of Tab. 1 correspond to the mean values of test data generated and runtime measured for the coverage of the *all_decisions* and the *almost_all_decisions* criteria. When testing objectives were not fulfilled within 1 hour of user time, we stopped the random generation and reported failure (shown with "-").

Table 1
Criteria coverage in the presence of conditional aliasing pointers

	program features			Our approach				Random approach			
				all_decisions				all_decisions		almost_all_decisions	
Programs	#p	#u	#a	#c	#pr	#td	rt(sec)	#td	rt(sec)	#td	rt(sec)
ow91	3	0	3	34	259	3	0.5	-	-	1417	2.5
lm03	4	2	1	5	27	2	0.1	3278	3.07	2215	1.9
lh98	4	0	6	36	174	2	0.1	1303	1.93	3	0.1
bl98	4	0	7	47	1042	3	0.1	3199	4.45	1007	1.4
foo	8	6	2	100	680	2	0.1	-	-	-	-
trityp	57	43	4	30734	80822	8	5.3	-	-	2870	12.1
jos97	∞	∞	6	106796	117802	2	18.0	5	1.27	2	0.4

6.3 Analysis

In all the cases, our approach achieves the full coverage of the *all_decisions*, even when the maximum number of points-to relations is high (bl98) or the number of infeasible paths is high (trityp) or infinite (jos97). By looking at the number of prunings, we confirm our intuition that the constraints play

an active role in the generation. Let us recall that this number corresponds to the number of times a domain is pruned by a constraint. The program jos97 is probably the more demanding as shown by the number of constraints generated and the number of prunings performed and requires so the greatest CPU runtime value (18sec). In all other cases, our approach outperforms random testing in terms of CPU time required to get the coverage of a criterion. The number of test data generated is not significative as our approach is deterministic while the random testing approach is probabilistic. In some cases (ow91, foo and trityp), the random testing approach fails to get a test set that covers the selected criterion. This is due to the fact that several decisions have a very low probability to be satisfied, especially in the presence of conditional aliasing problems. In our approach, the search of solutions is "guided" by the constraint solving process, even in the presence of pointers. We tried to change several parameters in random testing (such as the size of domains, or the CPU time allocated to the evaluation of a single test data) to get better results . However, this does not change the quality of results as examplified by the following evidence. In the trityp program, there is a decision that corresponds to the event $i = j = k$ where i, j, k represents the lengths of a triangle. It is trivial to see that the event which consists to generate a triple of equal numbers has a very low probability to happen whatever be the modified parameters. The program `jos97` distinguishes by the fact that all its decisions can be covered very quickly by the random testing approach. Only 5 test data in average are required to do so and the coverage can be obtained in 1.27sec in average. On the contrary, our approach requires 18sec to set up and solve the constraint system generated for this program. This is due to the fact that every time the loop of the program is unrolled, the solving process examines all the 6 possible points-to relations.

7 Related work

To the best of our knowledge, prior work on automatic goal-oriented test data generation did not address specifically pointer aliasing problems. However, several test data generation approaches deal with pointer variables. Korel [27] proposed exploiting several executions of the program to find a test datum on which a selected path is executed. Its approach does not suffer from the pointer aliasing problem as it is based solely on program executions. As a drawback, this approach cannot detect unsatisfiability of selected paths. More recently, Visvanathan and Gupta in [28], Marre et al. in [29] and Sai-ngern et al. [30] 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 with constraints on input values and aliasing problems occur only within input data structures.

PathCrawler [29] and CUTE [31] are two similar test data generators that try to cover all the feasible paths of C programs. Both systems work by combining concrete and symbolic execution. They solve path conditions in order to find the next test data that will follow a path that improve the current coverage of the program. 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 prune the search space made up of all the paths that reach a given branch [32]. However, unlike path-oriented ones, goal-oriented methods suffer from the conditional pointer aliasing problem. Considering all paths that reach a given branch is usually unreasonable as the number of control flow paths can be exponential on the number of decisions of the program or even infinite when loops are unbounded. As a consequence, we argue that goal-oriented methods scale up more easily than path-oriented methods for programs that contain only pointer variables. The algorithmic complexity of our approach is strongly related to the maximum number of possible aliases computed at each point of the program and there are studies showing this number remain small in general. For example, the experimental results section of [21] shows this number is almost always less than five. For us, the main drawback of our approach concerns its weakness to deal with dynamic allocation. Indeed, allowing dynamic allocation into a while-loop leads to the creation of a potentially unbounded number of aliases. Hence, the points-to analysis we used is no more suitable in this case. Note however that dealing with dynamic allocated structures in a goal-oriented method remains an open problem and none of the work previously cited addressed it.

8 Conclusion

In this paper, we have presented a new goal-oriented method for generating automatically test data for programs with multi-level pointer variables. The method is based 1) on the Pointer SSA form that extends traditional SSA by integrating the results of an intraprocedural flow-sensitive pointer analysis and 2) on the design of two CLP combinators that model the relation between pointers and pointed variables. The next steps of this work will be to study extensions in several directions. First, our approach could address the problem of function calls by exploiting the results of an interprocedural pointer analysis. Although a lot of work has been done in this area, technical problems remain to handle properly function pointers (second-order programming) and recursive calls. Second, we would like to extend our approach for pointers that address the heap. We could use other pointer analysis such as a shape analysis or dynamic points-to analysis to deal with dynamic allocation. These two extensions could open the road to the development of a symbolic goal-oriented

test data generation method able to deal with real-sized programs.

Acknowledgments

Thanks to Nicky Williams for her helpful comments on an earlier draft of this paper.

References

[1] A. Gotlieb, B. Botella, M. Rueher, Automatic test data generation using constraint solving techniques, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'98), Clearwater Beach, FL, USA, 1998, pp. 53–62.

[2] A. Gotlieb, B. Botella, M. Rueher, A clp framework for computing structural test data, in: Proceedings of Computational Logic (CL'2000), LNAI 1891, London, UK, 2000, pp. 399–413.

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, F. Zadeck, Efficently computing static single assignment form and the control dependence graph, ACM Transactions on Programming Language and Systems 13 (4) (1991) 451–490.

[4] S. Kowshik, D. Dhurjati, V. Adve, Ensuring code safety without runtime checks for real-time control systems, in: Proc. of the Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES'02), Grenoble, FR, 2002.

[5] R. DeMillo, J. Offut, Constraint-based automatic test data generation, IEEE Transactions on Software Engineering 17 (9) (1991) 900–910.

[6] R. DeMillo, J. Offut, Experimental results from an automatic test case generator, ACM Transactions on Software Engineering Methodology 2 (2) (1993) 109–127.

[7] J. Offut, Z. Jin, P. J., The dynamic domain reduction procedure for test data generation, Software–Practice and Experience 29 (2) (1999) 167–193.

[8] C. Meudec, ATGen: automatic test data generation using constraint logic programming and symbolic execution, Software Testing, Verification and Reliability 11 (2) (2001) 81–96.

[9] B. Botella, A. Gotlieb, C. Michel, Symbolic execution of floating-point computations, The Software Testing, Verification and Reliability journal 16 (2) (2006) pp 97–121.

[10] A. Gotlieb, B. Botella, M. Watel, Inka: Ten years after the first ideas, in: 19th International Conference on Software, Systems Engineering and their Applications (ICSSEA'06), Paris, France, 2006.

[11] J. Ferrante, K. Ottenstein, J. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Language and Systems 9 (1987) 319–349.

[12] M. Brandis, H. Mőssenbőck, Single-Pass Generation of Static Single-Assignment Form for Structured Languages, ACM Transactions on Programming Language and Systems 16 (6) (1994) 1684–1698.

[13] Axlog Ingenierie and Thales Airborne Systems, INKA–V1 User's Manual (december 2002).

[14] A. Gotlieb, B. Botella, Automated metamorphic testing, in: 27th IEEE Annual International Computer Software and Applications Conference (COMPSAC'03), Dallas, TX, USA, 2003.

[15] T. Chen, T. Tse, Z. Zhou, Fault-based testing in the absence of an oracle, in: IEEE Int. Comp. Soft. and App. Conf. (COMPSAC), 2001, pp. 172–178.

[16] K. Marriott, P. Stuckey, Programming with Constraints : An Introduction, The MIT Press, 1998.

[17] N. T. Sy, Y. Deville, Consistency techniques for interprocedural test data generation, in: ESEC/FSE-11: Proc. of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press, 2003, pp. 108–117.

[18] M. Carlsson, G. Ottosson, B. Carlson, An open–ended finite domain constraint solver, in: Proc. of Programming Languages: Implementations, Logics, and Programs, 1997.

[19] R. Cytron, R. Gershbein, Efficient accommodation of may–alias information in SSA form, in: Proceedings of Programming Languages Design and Implementation, ACM, Albuquerque, NM, 1993.

[20] R. Hasti, S. Horwitz, Using static single assignment form to improve flow-insensitive pointer analysis, in: PLDI '98: Proc. of conference on Programming language design and implementation, ACM Press, 1998, pp. 97–105.

[21] E. Emami, R. Ghiya, L. Hendren, Context–sensitive interprocedural points–to analysis in the presence of function pointers, in: Proceedings of Programming Languages Design and Implementation, ACM, Orlando, FL, 1994.

[22] C. Lapkowski, 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), LNCS 1383 Kai Koshimies (Ed), Lisbon, Portugal, 1998, pp. 128–143.

[23] R. Ghiya, L. Hendren, Putting pointer analysis to work, in: Proceedings of Symp. on Principles of Programming Languages, ACM, San Diego, CA, 1998.

[24] T. Ostrand, E. Weyuker, Data flow-based test adequacy analysis for languages with pointers, in: In Proceedings of the Symposium on Testing, Analysis, and Verification (TAV'91), 1991, pp. 74–86.

[25] V. Livshits, M. Lam, Tracking pointers with path and context sensitivity for bug detection in C programs, in: ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03), 2003, pp. 317–326.

[26] D. Binkley, J. Lyle, Application of the pointer state subgraph to static program slicing, Journal of Systems and Software 40 (1) (1998) 17–27.

[27] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering 16 (8) (1990) 870–879.

[28] S. Visvanathan, N. Gupta, Generating test data for functions with pointer inputs, in: Proceedings of the 17th IEEE Int. Conf. on Automated Software Engineering (ASE'02), Edinburgh, UK, 2002.

[29] B. Marre, P. Mouy, N. Williams, On-the-fly generation of k-path tests for c functions, in: Proceedings of the 19th IEEE Int. Conf. on Automated Software Engineering (ASE'04), Linz, Austria, 2004.

[30] S. Sai-ngern, C. Lursinap, P. Sophatsathit, An address mapping approach for test data generation of dynamic linked structures, Information and Software Technology 47 (2005) 199–214.

[31] K. Sen, D. Marinov, G. Agha, CUTE: A concolic unit testing engine for C, in: 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05), ACM, 2005, pp. 263–272.

[32] R. Ferguson, B. Korel, The chaining approach for software test data generation, ACM Transactions on Software Engineering Methodology 5 (1) (1996) 63–86.