Goal-oriented test data generation for programs with pointer variables

Arnaud Gotlieb, Tristan Denmat, Bernard Botella

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Goal-oriented test data generation for programs with pointer variables

Arnaud Gotlieb and Tristan Denmat and Bernard Botella

N°5528

Mars 2005

Systèmes symboliques

ISSN 0249-6399 ISRN INRIA/RR--5528--FR+ENG

Goal-oriented test data generation for programs with pointer variables

Arnaud Gotlieb * and Tristan Denmat † and Bernard Botella ‡

Systèmes symboliques
Projet Lande

Rapport de recherche n°5528 — Mars 2005 — 21 pages

Abstract: Automatic test data generation leads to identify 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 extends an existing goal-oriented test data generation technique to deal with multi-level pointer variables. These pointers are responsible for the existence of difficult conditional aliasing problems that usually provoke the failure of the test data generation process. The key point of our approach is the definition of a new static single assignment form based on the results of an intraprocedural flow-sensitive points-to analysis. This form allows us to propose an overall approach for generating goal-oriented test data in the presence of pointer variables based on the Constraint Logic Programming framework. The approach has been implemented and tested over a few examples extracted from the literature.

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

(Résumé : tsvp)

* Arnaud.Gotlieb@irisa.fr
† Tristan.Denmat@irisa.fr
‡ Bernard.Botella@fr.thalesgroup.com

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Génération automatique de données de test en présence de pointeurs

Résumé : La génération automatique de données de test vise à trouver des valeurs d'entrée d'un programme qui exécutent une branche (méhode orientée but) ou un chemin (méthode orientée chemin) particulier. Dans les deux cas, des techniques à base de résolution de contraintes ont été étudiées mais jusqu'à maintenant seules des méthodes orientées chemin ont traîtées les programmes manipulant des pointeurs. Ce papier étend une technique orientée but existante, basée sur la programmation logique par contraintes, aux pointeurs de plusieurs niveaux. Ce type de pointeurs entraîne des problèmes de synonymie conditionnelle qui provoquent l'échec de la génération des données de test. La méthode proposée repose sur la définition d'une forme à assignation unique utilisant une analyse de pointeurs intraprocédurale. Cette approche a été implémentée puis testée sur quelques programmes issus de la littérature.

Mots-clé : Génération de données de test orientée but, programmation logique par contraintes, forme statique à assignation unique, pointeurs

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.

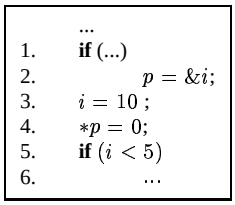


Figure 1: A conditional aliasing problem

What is exactly the problem ? In imperative programs, a dereferenced pointer and a variable may refer to the same memory location at some program point (this is known as the pointer aliasing problem). This can be due either to a statement into the code where a pointer 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. In the following, we will call this, a conditional aliasing problem. In the context of automatic goal-oriented test data generation, consider the task of generating a test datum that reaches branch 5-6 into the C code of Fig.1. If the assignment of statement 4 is considered to have no effect on variable i, then the branch 5-6 will be declared as unreachable by an automatic test data generator as $i = 10$ and $i < 5$ are contradictory. However, this is incorrect if the flow passes through statement 2 as, in this case, p points to i and then i is assigned to 0 at statement 4, which satisfies the decision of branch 5-6. On the opposite, if statement 4 is considered to be able to modify any pointed variable in the program, then the test data generation process just suspend as it cannot decide whether $i < 5$ is satisfied or not. Note that when a path is selected first, the pointing relations are all known and conditional aliasing problems are trivially handled, but note also that if the selected path is unfeasible then this must be demonstrated before switching to another choice and carry on the process.

To the best of our knowledge, prior work on automatic test data generation in the presence of pointers did not address the conditional aliasing problem. Korel proposes in [17] to exploit several executions of the program to find a test datum on which a selected path is executed. In [8], the approach is adapted to generate goal-oriented test data by making use of data flow analysis but it does not suffer from the conditional aliasing problem as it is based solely on program executions. More recently, Visvanathan and Gupta in [23], Zhang in [24] and Williams et al. in [20] address the problem of generating test data for C functions with pointer 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 within input data structures. All these three approaches have in common to require a path to be selected first and fall so into the path-

oriented methods category. Unlike path-oriented and among other advantages [8], goal-oriented methods exploit the early detection of non-feasible paths to prune the search space made of all the paths that reach a given branch [13]. Considering all paths that reach a given branch is usually unreasonable as the number of control flow paths is usually exponential on the number of decisions of the program or even infinite when loops are unbounded. In [13, 14], we proposed a novel framework that generates automatically goal-oriented test data for the coverage of structural criteria. The underlying method consists in generating a Constraint Logic Program over Finite Domains (a CLP(FD) program) associated with a C function and solving a CLP goal, obtained by the selection of a given branch. The approach relies on the use of Static Single Assignment (SSA) form [4] and Constraint Logic Programming techniques [15, 21] to efficiently solve the CLP goal and then to generate a test datum that reaches the selected branch. Although our method has proved to be useful to address non-trivial academic and industrial test data generation problems (including loops, arrays, structures, bitwise operations and so on) its incapacity to deal correctly with the conditional aliasing problem was considered by us as a major drawback.

This paper describes the extension of our test data generation method on a restricted class of pointer variables : multi-level pointers toward statically named variables. This class of pointers is mainly used in real-time control systems where dynamic allocation and unconstrained use of pointers is prohibited [18]. The main contributions of this paper are two-fold. First, we propose the definition of the *Pointer–SSA form* which respects the properties initially required for SSA even in the presence of pointers. This Pointer–SSA form integrates the results of an intraprocedural flow–sensitive pointer analysis into SSA in order to reveal the hidden definitions made by dereferenced pointers. The second contribution is the proposition of an overall goal-oriented test data generation method even in the presence of conditional pointer aliasing problems.

Outline of the paper. In Section 2, the background on our CLP-based test data generation technique is recalled. Section 3 gives an overview of our approach on a motivating example. Section 4 details the Pointer–SSA form while section 5 presents both specific CLP(FD) combinators used to model pointer use and definition. Section 6 reports on the preliminary results obtained with our prototype implementation and Section 7 indicates several perspectives to this work.

2 Background

Our approach is based on the use of Static Single Assignment form [4] and Constraint Logic Programming over finite domains [15].

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, v_2 otherwise. SSA has been used in several applications area such as optimizing compilers, automatic parallelization, static analysis, etc. 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 $\begin{bmatrix} x_i \\ ... \\ z_i \end{bmatrix}$.

2.2 The CLP(FD) framework

Following the definitions of [21], 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 goal[1]. A *goal* is a sequence of either primitive constraints or combinators. *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 primitive in the CLP(FD) library of Sicstus Prolog [2] : it holds if V is the I^{th} element in the list L of FD_variables.

When considered for solving, a CLP(FD) goal leads to build dynamically a *constraint system*, which is made of variables, domains and constraints. 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) on a constraint entailment mechanism which tries to infer new constraints from existing ones, 3) on a labelling procedure which makes hypothesis to find a solution to the constraint system.

Constraint propagation. During this process, primitive constraints and combinators are incrementally introduced into a propagation queue. An iterative algorithm manages 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 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 so 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

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

Original C code	Pointer–SSA form	CLP(FD) program (*where* $\&j = 21$ and $\&k = 22$)
int foo(**int** i) **int** j,k,r,$*p$;	**int** foo(**int** i) **int** j,k,r,$*p$;	**foo**(I, J_4) :– $I \in -2^{31}..2^{31}-1$
1. $j = 0$;	$j_1 = 0$;	$J_1 = 0$,
2. $k = 0$;	$k_1 = 0$;	$K_1 = 0$,
3. $p = \&j$;	$p_1 = \&j$;	$P_1 = 21$,
4. **if** ($i < 6$)	**if** ($i < 6$)	$\texttt{ite}(I < 6, \begin{bmatrix} J_2 \\ P_1 \end{bmatrix}, \begin{bmatrix} J_1 \\ P_2 \end{bmatrix}, \begin{bmatrix} J_3 \\ P_3 \end{bmatrix}, J_2 = 2, P_2 = 22)$
5. $j = 2$; **else**	$j_2 = 2$; **else**	
6. $p = \&k$; **fi**	$p_2 = \&k$; $\begin{bmatrix} j_3 \\ p_3 \end{bmatrix} = \phi(\begin{bmatrix} j_2 \\ p_1 \end{bmatrix}, \begin{bmatrix} j_1 \\ p_2 \end{bmatrix})$; **fi**	
7. $r = *p$;	$r_1 = \phi_u(p_3, \begin{bmatrix} \&j \\ \&k \end{bmatrix}, \begin{bmatrix} j_3 \\ k_1 \end{bmatrix})$	$R_1 = \Phi_u(P_3, \begin{bmatrix} 21 \\ 22 \end{bmatrix}, \begin{bmatrix} J_3 \\ K_1 \end{bmatrix})$,
8. $*p = r * i$;	$\begin{bmatrix} j_4 \\ k_2 \end{bmatrix} = \phi_d(p_3, \begin{bmatrix} \&j \\ \&k \end{bmatrix}, r_1 * i, \begin{bmatrix} j_3 \\ k_1 \end{bmatrix})$;	$R_2 = R_1 * I$, $\begin{bmatrix} J_4 \\ K_2 \end{bmatrix} = \Phi_d(P_3, \begin{bmatrix} 21 \\ 22 \end{bmatrix}, R_2, \begin{bmatrix} J_3 \\ K_1 \end{bmatrix})$,
9. **if** ($j > 8$)	**if** ($j_4 > 8$)	$\texttt{ite}(J_4 > 8, ...)$,
10. ...	...	

Figure 2: Foo example and its PSSA form and the generated CLP(FD) program

guarded–constraint $C_1 \longrightarrow C_2$ is just removed from the constraint–store. Otherwise, the guarded–constraint is suspended until awaked by the constraint propagation mechanism.

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 when a fixpoint is reached. One must resort to enumerate to get particular solutions. This labelling procedure tries to give a value to a 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 $\texttt{labelling}([X_1, .., X_n])$ 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 Translating into 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 [13, 14, 12].

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 (primitive–constraint or CLP combinator). Type declarations are translated into domain constraints. For example, the declaration of a signed integer x is translated into : $X \in -2^{31}..2^{31}-1$ where X is a logical FD_variable. Assignments and decisions are translated into arithmetical constraints. For example, assignment $x = x + 1$ is converted into the SSA form $x_2 = x_1 + 1$ and it is translated into $X_2 = X_1 + 1$ where X_1, X_2 are logical FD_variables.

Conditional statement. The conditional statement is treated with a user–defined combinator `ite/6` introduced in [13]. Arguments of `ite/6` 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 into 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 `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). The user-defined combinator `ite/6` is defined as :

Definition 1 `ite/6`
`ite`$(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Then}, C_{Else}) : -$
$c \longrightarrow C_{Then} \wedge \vec{v_2} = \vec{v_0}$
$\neg c \longrightarrow C_{Else} \wedge \vec{v_2} = \vec{v_1}$
$\neg(c \wedge C_{Then} \wedge \vec{v_2} = \vec{v_0}) \longrightarrow \neg c \wedge C_{Else} \wedge \vec{v_2} = \vec{v_1}$
$\neg(\neg c \wedge C_{Else} \wedge \vec{v_2} = \vec{v_1}) \longrightarrow c \wedge C_{Then} \wedge \vec{v_2} = \vec{v_0}$
$(c \wedge C_{Then} \wedge \vec{v_2} = \vec{v_0}) \veebar (\neg c \wedge C_{Else} \wedge \vec{v_2} = \vec{v_1})$

The two former guarded–constraints result from the operational semantic of the **if_else** statement whereas the three latter allow more effective deductions. Particularly, the last constraint contains the constructive disjunction operator $\veebar$. The interest is that, if ,for example, the constraint
`ite`$(..., [X_0], [X_1], [X_2], X_0 = 1, X_1 = 3)$ stands, the constructive disjunction allows us to know that $X_2 \in \{1, 3\}$. Note that The `ite/6` combinator is awaked by the solver when at least one of the domain of its variable has changed. For example, in the previous example, learning that $X_2 \geq 3$ will awake the combinator and will trigger the third guarded–constraint, as $X_2 \neq X_0$ would be entailed.

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
`w`$(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Body})$[13]. 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[2] :

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

Definition 2 `w/5`
$\mathtt{w}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Body}) : -$
$c \longrightarrow (C_{Body} \wedge \mathtt{w}(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, C_{Body}))$
$\neg c \longrightarrow \vec{v_2} = \vec{v_0}$
$\neg(c \wedge C_{Body}) \longrightarrow (\neg c \wedge \vec{v_2} = \vec{v_0})$
$\neg(\neg c \wedge \vec{v_0} = \vec{v_2}) \longrightarrow (c \wedge C_{Body} \wedge \mathtt{w}(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, C_{Body}))$

Note that the vector $\vec{v_3}$ is a vector of fresh variables. Following the treatment of the conditional, the first two guarded–constraints are deduced from the operational semantic of the **while** statement and model it faithfully [14]. The last two constraints 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.

Goal-oriented Test data generation. The selection of a branch into the C function leads to set up a CLP goal built with the control-dependencies [9]. 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 even if they must be determined dynamically for the loop statements [13]. In the example of Fig.1, the control-dependency associated to branch 5-6 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 semantic of the program is modeled faithfully, any solution of the CLP request is interpreted as a test datum that reaches the selected branch. In case where the solving process shows that there is no solution, then the selected branch is declared unreachable. This approach has been implemented into the INKA tool [1] and evaluated on a set of academic and reasonably-sized industrial problems [14]. In [12], we also proposed to use this framework to generate test data that violate high-level properties called metamorphic–relations [3].

3 An overview of the approach

We start by giving an overview of our approach for conditional pointer aliasing problems on a motivating example. Consider the task of generating a test datum on branch 9-10 is executed in the C program of Fig.2. Our goal-oriented test data generation process is composed of three main steps. The first step aims at generating the Pointer SSA form (PSSA) of the C code, which is given in the second column of Fig.2. The definition of PSSA is mainly based on two ideas :

1. First, to exploit 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 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 exist during execution.

2. Second, to introduce two new forms of ϕ–functions to model the dereferencing process. ϕ_u–functions model uses of dereferenced pointers. At statement 7, $\phi_u(p_3, \begin{bmatrix}\&j\\ \&k\end{bmatrix}, \begin{bmatrix}j_3\\ k_1\end{bmatrix})$ 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 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 (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 PSSA form into a CLP(FD) clause as shown in the third column of Fig.2. The clause's 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's address is associated to a unique integer[3] ($\&j = 21$, $\&k = 22$ where 21 and 22 correspond to internal symbol table's 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.

Note that the ϕ_d combinator is related to the *IsAlias* function that was formerly introduced by Cytron and Gershbein [5] 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 goal-oriented test data generation 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 this example, the result of the

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

Figure 3: A test data generation request

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 $\begin{bmatrix}J_4\\ K_2\end{bmatrix} = \Phi_d(P_3, \begin{bmatrix}21\\ 22\end{bmatrix}, R_2, \begin{bmatrix}J_3\\ K_1\end{bmatrix})$ lead to $P_3 = 21$ and $J_4 = R_2$. 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 ends up 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.

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

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

4 The Pointer–SSA form (PSSA)

4.1 A simple language over pointer variables

In the above example, only pointers to integer variables were considered. In practice, this situation is infrequent because real programs deal with multi-level pointers, pointer arithmetic, pointers to structures, etc. However, note that although the C language offers extended possibilities for manipulating pointers, several coding rules that forbid an unconstrained usage of pointers and unconstrained flow of control (goto) are usually enforced when developing real-time control systems [18]. So, in this context, we will confine ourselves to a simple language over the pointers based on multi-level pointers toward statically named variables. In programs that use this class of pointers, the only 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.

4.2 Normalization

Normalizing a function consists in breaking complex statements into a set of elementary statements by introducing temporary variables. In [7, 19, 11], 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 semantic. 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. For the presentation, only four assignment statements are considered : $p = \&q$, $p = q$, $p = *q$, $*p = q$.

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 choosed a points-to analysis formerly introduced by Emami et al. [7]. 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 points to a only on some control flow paths. In fact, the analysis does not even say whether there

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

exists a feasible control flow path that contains the pointing relation. Although it can be very imprecise, a points-to 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 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 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 flow-sensitive

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,defi nite) pto(q,a,defi nite)	pto(q,a,possible) pto(q,b,possible)

Figure 5: *Points-to analysis*

analysis for the two following main reasons :

1. 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 number of hidden definitions ;

2. efficient algorithms exist for the C functions that we considered, e.g. that respect the coding rules given in subsection 4.1.

In [7], an intraprocedural syntax-based algorithm that computes the results of a flow-sensitive *points–to analysis* for structured C programs is given. Every statement can modify the pointing relations set with the help of sets : the set of "killed" relations ($kill\_set$) and the set of relations generated by the statement (gen_set). In the algorithm, the notation for these two sets makes use of existentially quantified variables, denoted by "$\_x$". For example, $\{pto(p, \_x, \_rel) | pto(p, \_x, \_rel) \in Input\}$ denotes the set of all pointing relations associated with p in $Input$. Fig.6 introduces the algorithm for the elementary statements that manipulate pointers.

/* Given a statement S, and $Input$ a set of pointing relations */
/* process_basic_stmt(S, $Input$) returns the set of relations after S */

Points–to process_basic_stmt(**Statement** S, **Points–to** $Input$)

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

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

S is of the form $p = *q$ **then**
kill_set := $\{pto(p, \_x, \_rel) | pto(p, \_x, \_rel) \in Input\}$;
gen_set := $\{pto(p, \_b, \_rel) | pto(q, \_a, \_r_1)$ and $pto(\_a, \_b, \_r_2) \in Input\}$;
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 $pto(\_x, \_y, \_rel) \in Input\}$;
gen_set := $\{pto(\_x, \_z, \_rel) | pto(p, \_x, \_r_1)$ and $pto(q, \_z, \_r_2) \in Input\}$;
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

In the presence of control flow structures, the results of the analysis of every branch are merged into a single set. Into this merge process, a definite *points–to* relation can be transformed into 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. Fig.7 shows

a recursive algorithm that handle the two basic control flow structures, namely if– and while– statement. 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.

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

Points–to process_stmt( Statement S, Points–to Input)

    If S is void then returns Input

    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 */
        Output := process_basic_stmt(Head, Input) ;
        returns process_stmt(Tail, Output) ;

    If S is of the form [if(C) then S1 else S2 fi ]
    then
        Output_then := process_stmt(S1,Input) ;
        Output_else := process_stmt(S2,Input) ;
        returns merge(Output_then, Output_else) ;

    If S is of the form [while(C) do S od]
    then
        do
            LastIn := Input ;
            Output := process_stmt(S, Input) ;
            Input := merge(Input, Output) ;
        while LastIn ≠ Input
        returns Input
```

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

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 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 iff $p = \&a_i$.

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 $\phi_d(p, \begin{bmatrix} \&a_1 \\ .. \\ \&a_n \end{bmatrix}, expr, \begin{bmatrix} v_1 \\ .. \\ v_n \end{bmatrix})$ where $expr$ denotes any expression, returns a vector of variables $\begin{bmatrix} x_1 \\ .. \\ x_n \end{bmatrix}$ where $x_i = expr$ if $p = \&a_i$ and $x_j = v_j$ for all $j \neq i$. Although very similar in appearance, both functions distinguish themselves by their syntactical role : ϕ_u–function is a right hand side form whereas ϕ_d is a left hand side form.

4.5 An algorithm for constructing the PSSA form

```
/* Given a 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(q, PTO_S)) ;
            /* addresses of aliased variables of (*p) */
      v⃗ := vector_of_dereferenced(from(q, 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 on other statements */
  return St ;
```

Figure 8: Algorithm for constructing PSSA

Few notations are required to describe the algorithm. When analyzing a statement, $p\downarrow$ denotes the last numbered variable associated to p whereas $p\uparrow$ denotes the new fresh numbered variable associated to p. So, a definition of p (resp. $*p$) will be noted as $p\uparrow$ (resp. $(*p)\uparrow$) and a use of p (resp. $*p$) will be viewed as $p\downarrow$ (resp. $(*p)\downarrow$). If PTO_S denotes the set of pointing relations available at statement S, the set of variables pointed by p is noted $from(p, PTO_S)$. Formally speaking, $from(p, PTO_S) = \{\_a \mid pto(p, \_a, \_) \in PTO_S\}$.

The algorithm for constructing 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 only for the main basic statements obtained after normalization, bearing in mind that other statements can easily be deduced from the treatment of these ones.

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 semantic. However, two new functions have been introduced : ϕ_u- and ϕ_d- functions. In the CLP(FD) program, these functions are extended by the means of two relational combinators, namely Φ_u and Φ_d. The definition of these CLP(FD) combinators is based on guarded–constraints as done for both `ite/3` and `w/5` combinators. 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 points to a single variable.

Definition 3 $\Phi\_u/4$

Declarative view : *Let $X, P, V_1, .., V_n$ be $\mathcal{FD}$_variables and $P_1, .., P_n$ be n be 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 semantic : $X = \Phi_u(P, \begin{bmatrix} P_1 \\ .. \\ P_n \end{bmatrix}, \begin{bmatrix} V_1 \\ .. \\ V_n \end{bmatrix})$ *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 view : *Let $X, P, V_1, .., V_n$ be $\mathcal{FD}$_variables and $P_1, .., P_n$ be n be distinct numeric constants, then* $\begin{bmatrix} X_1 \\ .. \\ X_n \end{bmatrix} = \Phi_d(P, \begin{bmatrix} P_1 \\ .. \\ P_n \end{bmatrix}, DP, \begin{bmatrix} V_1 \\ .. \\ V_n \end{bmatrix})$ *is true iff* $\exists i, P = P_i \wedge X_i = DP \wedge \{X_j = V_j\}_{\forall j \neq i}$.

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

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

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

Note that in case where p is not assigned to a valid address, then both Φ_d and Φ_u combinators just **fail** during the solving process. As usual in CLP, failure is interpreted as unsatisfiability of the constraint store. In fact, dereferencing a null or invalid pointer is usually considered as a fault but recall that our approach is not targeted to find bugs. Our goal is to generate test data on which on a selected branch in the code is executed.

6 Preliminary results

We implemented our approach into the goal-oriented test data generator InKa [14, 12]. The tool automatically generates test data for the coverage of several structural criteria such as all_statements, all_branches, MC/DC. It handles a non-trivial subset of the C and C++ languages [1]. The tool has several other functionalities, such as test coverage measurements, control flow monitoring, test cases management, etc. It is mainly developed in Prolog, Java and C and makes use of the clp(fd) library of Sicstus Prolog [2] to solve the test data generation requests. Our implementation to deal with pointer programs includes a pointer analyzer, a PSSA form generator and the design of both combinators Φ_u and Φ_d.

To correct our implementation and evaluate the approach, we generated test data for C functions that present conditional pointer aliasing problems. Most of them were extracted from the literature. In this paper, we report the results of only two of them that are the most representatives. The first program, extracted from [19], is shown in Fig.9 along with its PSSA form. It presents a conditional aliasing problem with two-level indirection pointers when one wants to reach branch 10-11. The

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. $p = \&q$;	$p_2 = \&q$;
	$\begin{bmatrix} g_3 \\ p_3 \end{bmatrix} = \Phi(\begin{bmatrix} g_2 \\ p_2 \end{bmatrix}, \begin{bmatrix} g_1 \\ p_1 \end{bmatrix})$;
fi	**fi**
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 foo3

points-to relations computed for program lh98 at statement 9 are given by the following diagram:

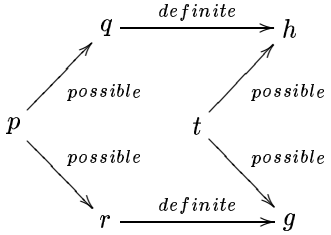


Fig.10 contains the CLP(FD) program generated for function lh98 and the requests asking for test data able to reach branch 10-11.

The results show that 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.

The second program is inspired from a part of the josephus program that belongs to the folklore of pointer analysis problems [10]. In this part, pointing relationships are modified within a loop. The program, called jos97, is given in Fig.11.

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 relations com-

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).

$H_0 \in 8..9$

?- $H_1 > 100$, lh98(H_0, H_1), $labelling([H_0])$.

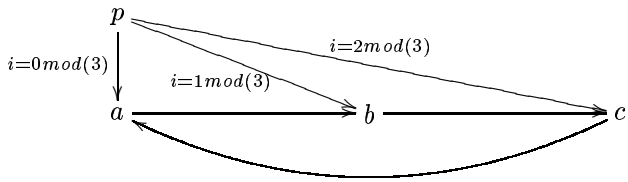
$H_0 = 8$;

$H_0 = 9$;

no

Figure 10: CLP(FD) program for lh98

puted by the flow-sensitive points-to analysis at statement 8 are illustrated by the following diagram:



In this diagram, the edges are labelled 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 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 by our method. The first request corresponds to the problem of reaching branch 8-9 by using only constraint propagation, whereas the second contains a labelling request in addition. In these examples, the domain of the input parameter is restricted to be a signed integer. For the first request, the solver shrinks the domain of I_0 to $2..2^{31} - 1$,

Normalized C code	PSSA form
int jos97(**int** i)	**int** jos97(**int** i_0)
int ****p,***a,**b,*c;	**int** ****p,***a,**b,*c ;
1. $p = \&a$;	$p_1 = \&a$;
2. $a = \&b$;	$a_1 = \&b$;
3. $b = \&c$;	$b_1 = \&c$;
4. $c = \&a$;	$c_1 = \&a$;
	$\begin{bmatrix} i_2 \\ p_3 \end{bmatrix} = \phi(\begin{bmatrix} i_0 \\ p_1 \end{bmatrix}, \begin{bmatrix} i_1 \\ p_2 \end{bmatrix})$;
5. **while**($i > 0$)	**while**($i_2 > 0$)
do	**do**
6. $i = i - 1$;	$i_1 = i_2 - 1$;
7. $p = *p$;	$p_2 = \phi_u(p_3, \begin{bmatrix} \&a \\ \&b \\ \&c \end{bmatrix}, \begin{bmatrix} a_1 \\ b_1 \\ c_1 \end{bmatrix})$;
od	**od**
8. **if**($p == \&c$)	**if**($p_3 == \&c$)
9. **return** 1 ;	**return** 1 ;
else	**else**
10. **return** 0 ;	**return** 0 ;
fi	**fi**

Figure 11: PSSA form of function jos97

which is, as expected, a correct over-estimation of the variation domain of I_0. The inferior 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 requests exhibits the exact solutions through the backtracking process of Prolog. The values found by the solver correspond to the values that satisfy the constraint $i = 2 \ mod \ 3$. These two requests show that the pointing relations were correctly propagated by the solving process.

7 Conclusion

In this paper, we have presented a new method for generating automatically goal-oriented 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 works has been done into this area [7, 22], technical problems remain to handle properly function pointers (second-order programming) and recursive calls. Second, we will study how to extend our approach for pointers that address the heap. In the presence

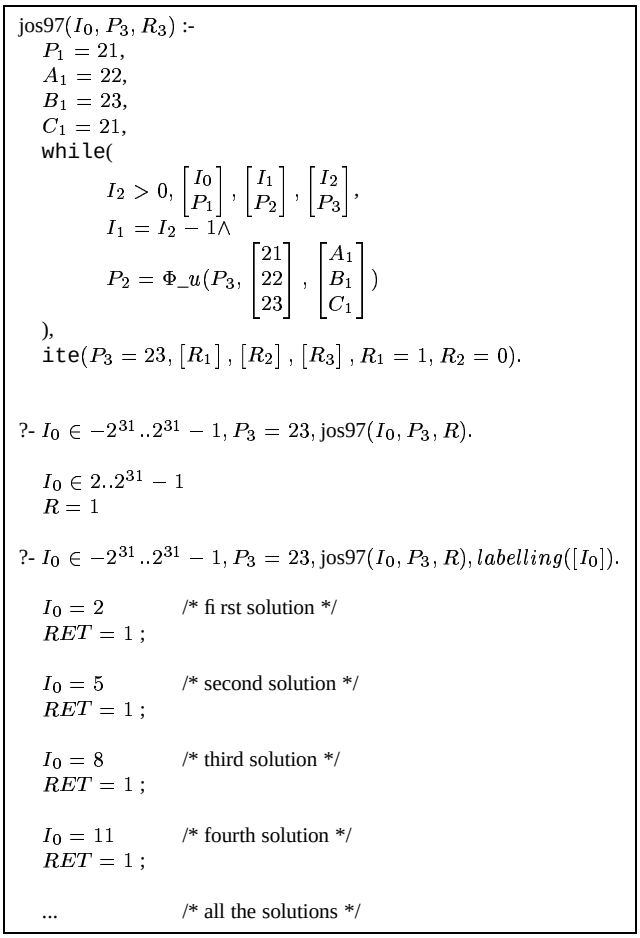


Figure 12: CLP(FD) program for foo4

of dynamic allocation, the points-to analysis we used don't converge anymore. Hence, it should be replaced by another pointer analysis such as a shape analysis [6, 16, 10, 11]. In the future, we plan to model the relation between the pointers and the heap by the means of CLP combinators, as we did for pointers toward statically named variables. This approach could be of particular interest to deal with programs that require particular shapes of dynamic structures to execute properly.

References

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

[2] M. Carlsson, G. Ottosson, and B. Carlson. An Open–Ended Finite Domain Constraint Solver. In *Prog. Lang., Impl., Logics, and Programs (PLILP)*, 1997.
[3] T. Chen, T. Tse, and Z. Zhou. Fault-based testing in the absence of an oracle. In *IEEE Int. Comp. Soft. and App. Conf. (COMPSAC)*, pages 172–178, 2001.
[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Effi ciently Computing Static Single Assignment Form and the Control Dependence Graph. *Trans. on Programming Languages and Systems*, 13(4):451–490, 1991.
[5] R. Cytron and R. Gershbein. Effi cient Accommodation of May–Alias Information in SSA Form. In *Prog. Language Design and Implementation*, Albuquerque, NM, Jun. 1993. ACM.
[6] A. Deutsch. Interprocedural May–Alias Analysis for Pointers: Beyond k–limiting. In *Prog. Language Design and Implementation*, Orlando, FL, Jun. 1994. ACM.
[7] M. Emami, R. Ghiya, and L. J. Hendren. Context–Sensitive Interprocedural Points–to Analysis in the Presence of Function Pointers. In *Prog. Language Design and Implementation*, Orlando, FL, Jun. 1994. ACM.
[8] R. Ferguson and B. Korel. The Chaining Approach for Software Test Data Generation". *ACM Trans. on Soft. Eng. and Meth.*, 5(1):63–86, Jan. 1996.
[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its use in optimization. *Trans. on Programming Languages and Systems*, 9-3:319–349, July 1987.
[10] R. Gaugne. *Techniques d'analyse statique pour l'aide à la mise au point de programmes avec manipulation explicite de pointeurs*. Phd thesis, Université de Rennes 1, Oct. 1997.
[11] R. Ghiya and L. Hendren. Putting Pointer Analysis to Work. In *Principles of Programming Languages (POPL)*, San Diego, CA, Jan. 1998. ACM.
[12] A. Gotlieb and B. Botella. Automated metamorphic testing. In *27th IEEE Annual International Computer Software and Applications Conference (COMPSAC)*, Dallas, TX, USA, Nov. 2003.
[13] A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *ACM Int. Symp. on Soft. Testing and Analysis (ISSTA). Soft. Eng. Notes*,23(2):53-62, 1998.
[14] A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Computational Logic (CL)*, pages 399–413, LNAI 1891, 2000.
[15] 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.
[16] J. Hummel, L. J. Hendren, and A. Nicolau. A General Data Dependence Test for Dynamic, Pointer–Based Data Structures. In *Prog. Language Design and Implementation*, Orlando, FL, Jun. 1994. ACM.
[17] B. Korel. Automated Software Test Data Generation. *Trans. on Software Engineering*, 16(8):870–879, Jul. 1990.
[18] S. Kowshik, D. Dhurjati, and 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, Oct. 2002.
[19] 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).
[20] 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, Sep. 2004.
[21] K. Marriott and P. J. Stuckey. *Programming with Constraints : an Introduction*. The MIT Press, 1998.
[22] M. Shapiro and S. Horwitz. Fast and Accurate Flow–Insensitive Points–to Analysis. In *Principles of Programming Languages (POPL)*, Paris, France, Jan. 1997. ACM.
[23] 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, Sep. 2002.
[24] 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, Sep. 2004.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399