INKA: TEN YEARS AFTER THE FIRST IDEAS

Arnaud Gotlieb
IRISA – INRIA
Campus Beaulieu
35042 Rennes, France

Bernard Botella
THALES AEROSPACE
2, avenue Gay-Lussac
78851 Elancourt, France

Mathieu Watel
AXLOG INGENIERIE
19-21 rue du 8 mai 1945
94110 Arcueil, France

Phone : +33 (0)1 99 84 75 76 – Fax : +33 (0)1 99 84 71 71 – Email : Arnaud.Gotlieb@irisa.fr

Abstract: Ten years ago, we wrote an ICSSEA paper[1] on the use of constraint (logic) programming to automatically generate white-box test data for imperative programs. This work opened the road to several research and development projects on this topic and led us to the design of INKA a software test data generator for C/C++. Ten years later, where are we? This paper proposes to review the advances we made on constraint-based test data generation and details the architectural design of INKA V2 which is a state-of-the-art implementation of our research and development works. The paper draws some perspectives on constraint-based white-box testing which we consider to be a roadmap for the next (ten) years.

Keywords: Software testing, test data generation, constraint-based white-box testing

[1] In 1996, ICCSEA was called «Le Génie Logiciel & ses Applications : 9èmes journées internationales»

1. INTRODUCTION

It is well known that black-box testing and white-box testing are complementary techniques. While black-box testing checks whether an implementation conforms its specifications, white-box testing focuses on non-functional requirements that are typically formulated in external documents such as standards and norms. In the software quality plans, the testing process of a system usually refers to standards and norms that belong to the contractual basis of the project and express requirements on the structural code coverage of the software. For example, the RTCA DO178B standard used in civil and military airborne systems requires any critical software to be tested with statement coverage, decision coverage or MC/DC[2]. Notice that after the black-box testing of an implementation, the requirements on code coverage are rarely satisfied. In fact, executing certain paths or branches of the software highly depends on implementation choices such as algorithmic optimisations, tabulations of particular cases, program specialization, etc. that cannot be found in the software specifications. For these reasons, white-box testing remains indispensable.

Although they are widely used, existing marketed white-box testing tools are restricted to coverage monitoring and coverage measurements. In most industrial projects, the generation of white-box test data (test data that execute a given path or branch in the source code) is currently performed manually and automatic white-box test data generation remains a holly grail for most testers. In fact, constructing test data for which a part in the code is executed is a non-obvious intellectual process: it requires to understand the conditions under which that part can be executed and to solve a kind of soduku to find the data respecting these conditions.

Ten years ago, we proposed an original idea which consisted to benefit from well-proved constraint (logic) programming techniques to automate the generation of white-box test data [1]. Constraint programming replaces classical imperative code statements by constraints which are logical relations between different variables of a problem to be solved. We developed a first prototype tool called INKA which exploited this property by translating C source code into a constraint program. The selection of an element of the source code (a test objective) led to set up a constraint system on the input variables for which INKA automatically computed a solution [2]. By combining the test objectives, INKA allowed the tester to find a test suite ensuring the coverage of all executable statements or decisions leading to a 100% coverage of the structural criteria. For the tester, the only remaining task consisted in defining an oracle able to provide a verdict on the generated test data (or just validate the computed results provided by the tool). Furthermore, as INKA exploited the results of a constraint solving procedure, it was able to detect some parts of the program that cannot be executed (dead code). In 2000, we started the development of an industrial version of INKA for C/C++ programs in cooperation with three French academic laboratories. Grants from the French Ministry of Research helped us to develop the tool and INKA V1 was released in 2002 [3] although it still suffers from a lot of restrictions on the C/C++ language used to write the tested programs. Nevertheless, one year later, the research projects initiated by the academic partners led to original solutions that decided us to build a totally new version of INKA able to deal with C/C++ constructs that was considered at that time as almost impossible to deal with on real-word programs. These included floating-point data types [4], pointers and dynamically allocated structures [5] and highly modular programs. After three years of intensive works, INKA V2 currently implements the best state-of-the-art solutions of constraint-based white-box test data generation in a framework that includes test case management, test report generation, coverage monitoring, automatic stubs and makefile generation, property-oriented testing [6] and automatic test data generation [7]. However, we must confess that lots of work need yet to be undertaken to reach a state where any C/C++ program can be analyzed, converted into a constraint system and treated for test case generation.

This paper presents the design of INKA V2 and introduces its main functionalities. Next section details the architecture of the tool while section 3 explains how programs are translated into constraint systems. Section 4 details some of strategies exploited for finding test data able to satisfy a given test objective. Finally, section 5 draws some perspectives on the necessary works required for improving the scope and the efficiency of the tool.

2. ARCHITECTURE OF INKA V2

INKA V2 is based on an original architecture shown in Fig.1. As explain below, this architecture is targeted to tackle difficult C/C++ language constructs in constraint-based white-box testing.

[2] Modified Condition/Decision coverage (MC/DC) is required only for the highest level of critical software

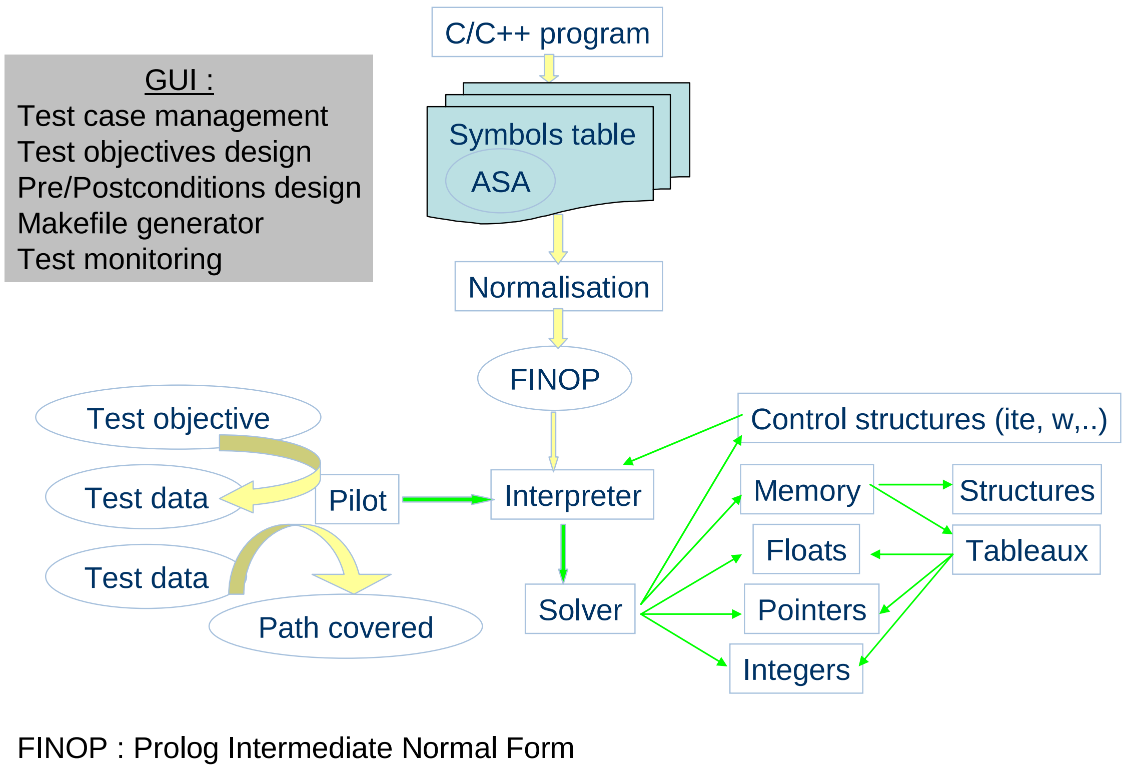


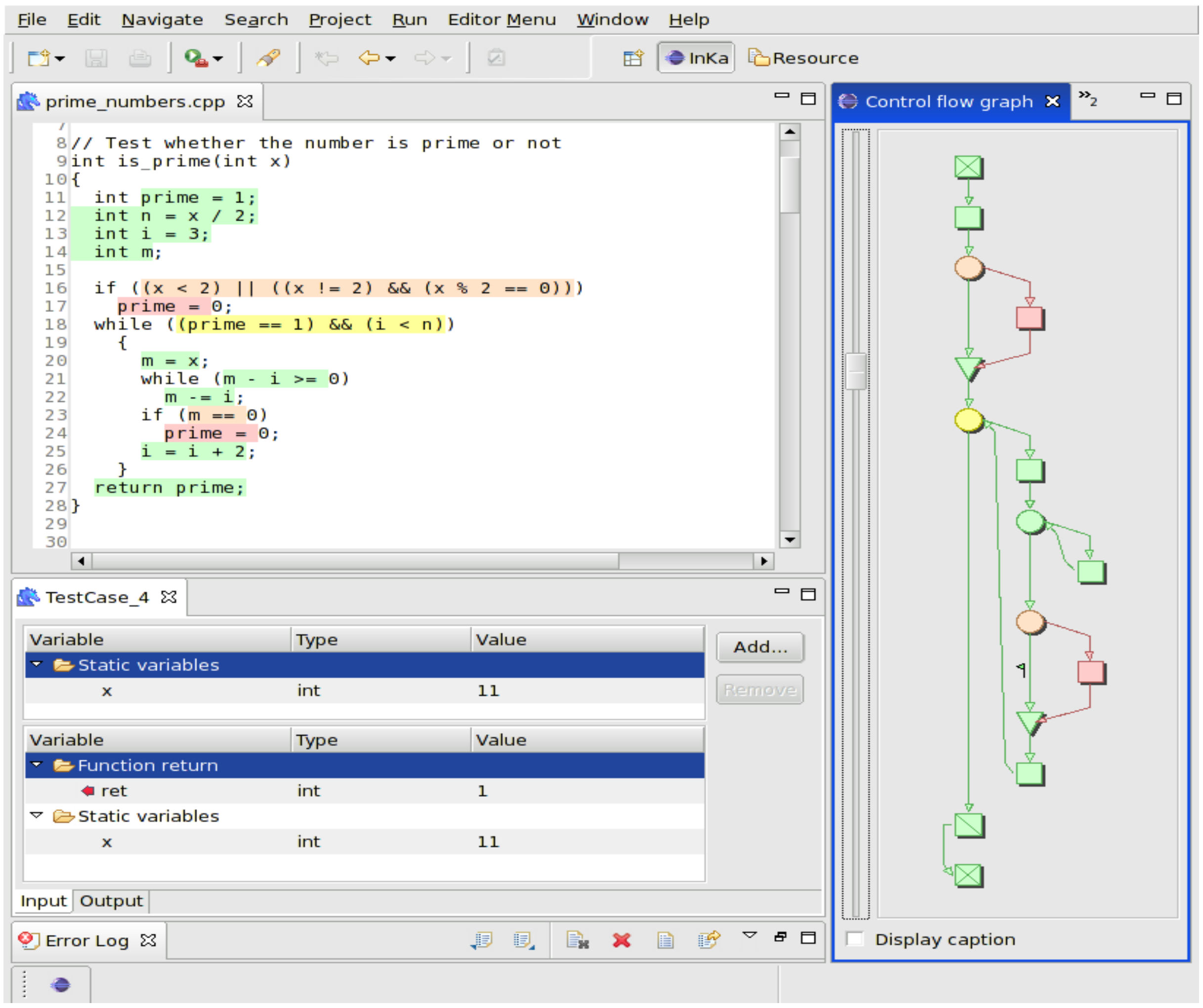
Fig. 1: INKA V2: architecture

Firstly, a ANSI/ISO C/C++ file is parsed to build a symbol table that contain all the symbols defined in the program as well as the abstract syntax tree (ASA) associated to each function and each method. A single parser is used to parse both languages as C++ integrates almost all the syntactical constructions of the C language: the divergence is mainly due to distinct semantics. C++ parsing is usually considered as a non-trivial task as it requires dealing with several ambiguities. A key point of our approach has been to develop a backtrackable parser that can explore a possible derivation and undo all the bindings it did whenever a contradiction is found. Note that the development of such a backtrackable parser was facilitated by the use of Prolog as our main development language.

Secondly, the ASA of each function/method is normalized and translated into a program written in an intermediate pivot language called FINOP (Prolog Intermediate Normal Form). Normalization is a process that breaks complex statements into a sequence of simpler statements by introducing new temporary variables. Normalization also makes explicit numerous C++ treatments that remain hidden to the developer (e.g. implicit type casting and conversions, overloading resolution, namespaces elimination, etc.). FINOP is a language in which variables are strongly typed, statements are simple (they handle usually less than four references) and control structures are guarded by Boolean variables. The language is expressive enough to allow the management of objects (memory allocation and deallocation), the combination of multiple data types (pointer, structures, integer and floating-point data types), and the use of almost all C++ operators including bit-to-bit, logical, arithmetical and dereferencing operators.

Thirdly, a pilot processes the test data generation/monitoring requests coming from the GUI. Apart from these requests, the GUI allows the user to manage a set of XML-formatted test cases (import/export existing test cases), to define test objectives such as coverage criteria (statements, decisions or MC/DC) or pre/post properties[3], and to manage the testing scripts of the application under test (makefile generation, oracle definition and test scripts generation). However, the main functionalities of the tool remains automatic test data generation and monitoring. Fig. 2 shows a screen shot of the tool that exemplifies these two functionalities. The GUI is implemented as a plug-in Eclipse.

[3] Pre/post properties are formulas over the input/output program variables for which the tester wishes to find counter-examples [6].



```
// Test whether the number is prime or not
int is_prime(int x)
{
  int prime = 1;
  int n = x / 2;
  int i = 3;
  int m;

  if ((x < 2) || ((x != 2) && (x % 2 == 0)))
    prime = 0;
  while ((prime == 1) && (i < n))
    {
      m = x;
      while (m - i >= 0)
        m -= i;
      if (m == 0)
        prime = 0;
      i = i + 2;
    }
  return prime;
}
```

Variable	Type	Value
Static variables		
x	int	11

Variable	Type	Value
Function return		
ret	int	1
Static variables		
x	int	11

Fig. 2: INKA V2 (legend: green=covered, red=not covered, orange=only some decisions covered, yellow=all the decisions covered but not MC/DC covered)

Fourthly, the interpreter analyses each FINOP statement and generates new constraints online. This is the keystone of our approach as this process allows new constraints to be added to the constraint system while deductions are obtained from other constraints. Technically, we say that the constraint system is dynamic. Let us explain this process on a very basic example. Consider the following C code that manipulates a list data type (struct cell) and suppose we would like reaching statement 10

```
…
1.    struct cell *t = (struct cell *) malloc(sizeof(struct cell));
2.    int i = 1;
3.    t->key = 1;
4.    while(…) {
5.        t->next = (struct cell *) malloc(sizeof(struct cell));
6.        t = t->next ;
7.         t->key = i;
8.        i++; }
9.    if( i > 2) {
10.       …
```

As we do not know whether le loop condition is satisfied or not (in this example, suppose the loop condition depends on input variables), classical constraint-based test data generation approaches usually fail to deduce anything on the shape of the list pointed by `t`. However, by using an interpreter that can be awaked when it is shown that the loop must be unrolled (`i > 2` and `i = 1` are contradictory), some deductions can be automatically obtained. In our example, the interpreter will be launched at least two times on the body of the while statement and a list of at least two elements will be created. Hence, we get a correct approximation of the shape of the data structure pointed by t that should be created in order to reach statement 10. In fact, this is even the only deduction we can have with respect to the given test objective. Note that in this example each iteration of the loop modifies the physical heap by allocating a new `struct cell` object. A similar process is routinely performed in C++ when class instances are created on the heap.

Finally, the interpreter interacts with a constraint solver that manages constraints from five specific constraint libraries (control structures, memories, integers, floats and pointers). These libraries define constraints for dealing with most C/C++ operators. The control structures library includes a relational view of if_then_else (called ite), while_do (w), do_while (dow), switch_case (switch) statements, whereas the memory library defines constraints that access/update the memories (load/store). The memories are based on an abstract model that includes tableaux that represent basic elements (pointers, integers and floats) and a list of structures that is exploited to access/updates the structures and the classes of programs. This abstract model of memory is detailed in the next section.

3. CONSTRAINT REASONNING IN INKA V2

Automatic constraint reasoning is a key issue in INKA V2. The architectural design of the tool allows dealing with C/C++ operators on pointers, integers, floats and structures but reasoning *efficiently* on these operations requires to pay a special attention to the way the constraints are defined. Let us first recall that constraint reasoning usually includes two interleaved processes: constraint propagation and tree search. Constraint propagation exploits each constraint in turn to prune the search space of solutions whereas tree search exhaustively explores the search space by making assumptions. In INKA V2, almost all C/C++ operations are treated as relations over two (abstract) memory states. The first state represents the memory before executing the statement whereas the second state represents the memory after executing the statement. A memory state is an abstract model of the physical memory on a given point of the resolution that is organized as shown in Fig.3. Any basic type variable i possesses an address noted $@_i$ and a value V_i that can be both unknown in a given memory at a given time of the resolution process. Dynamic addresses are associated to anonymous program locations, such as the results of dynamic allocations. An abstract memory contains three tableaux[4] and a list of available structures. A status is associated to each component that says whether the component is closed or not. A closed tableau denotes a tableau in which all the created objects are determined: It cannot accept any new object to be created. Such status is of particular interest when one looks for the possible shapes of a dynamic structure during the tree search as it strongly constrains the set of identifiers a pointer can point to. Each tableau contains the pairs $@_i$-V_i associated to each declared or newly allocated variable of a given abstract type (pointers, integers or floats). Additional information is available on the integer and floating-point variables such as their concrete type (size and sign) and the domain of their possible values in the current state of the resolution. For pointers, the tool collects some non-numeric information: the domain of their only possible values (resulting from C expressions such as `p==&a || p==&b || p==&c` for example) and the domain of some impossible values (resulting from C expressions such as `p!=&a`).

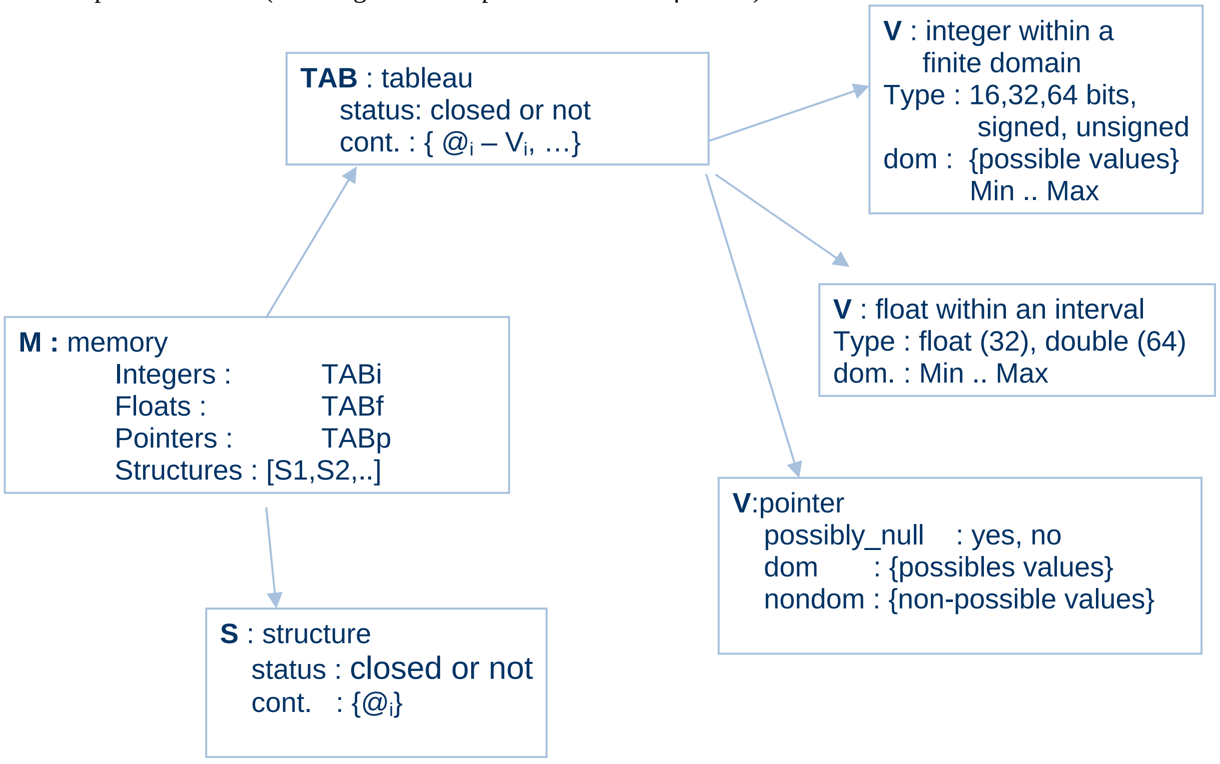
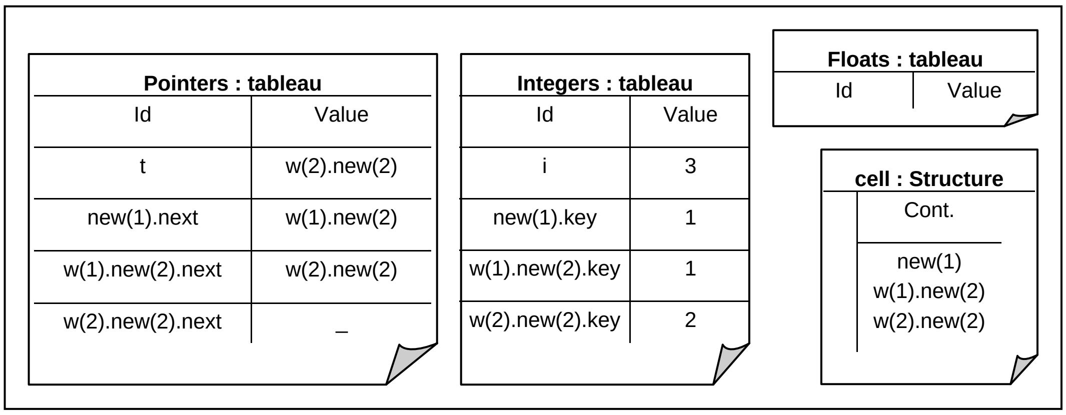


Fig.3: An abstract model of the physical memory

For pointers, possible values are either symbolic names or anonymous program locations. Fig.4 shows the abstract memory obtained on statement 10 of the C program given above after constraint propagation. In this memory, w(2).new(2) denotes the anonymous program location obtained in the second dynamic allocation statement of the program (in statement 5) of the second iteration of the while loop. This location represents a structure defined in the list of structures. Notice that the value of pointer w(2).new(2).next is unknown and can points to any cell structure. In fact, only the tree search step will give a value to this location by closing the memory.

[4] The term *Tableau* denotes only an internal data structure. Recall that arrays are only simulated in C/C++ via the use of pointers. In the INKA framework, arrays are converted into pointers in the normalization step.

Each statement of the FINOP program is translated into a constraint over abstract memories. We are not going to enter all the details of this translation, nor explaining all the deduction capacities of these constraints. Instead, we just explain the results of the constraint propagation step on a single constraint example. Consider the following constraint: **`store_elt(P,V,M1,M2)`** which is set up whenever the program stores any value **`V`** in memory at address **`P`**. This constraint is posted every time an assignment statement is called in FINOP programs and states the following relation: **`M2`** is the same memory as **`M1`** except in location **`P`** where **`V`** is stored.



Pointers : tableau

Id	Value
t	w(2).new(2)
new(1).next	w(1).new(2)
w(1).new(2).next	w(2).new(2)
w(2).new(2).next	_

Integers : tableau

Id	Value
i	3
new(1).key	1
w(1).new(2).key	1
w(2).new(2).key	2

Floats : tableau

Id	Value

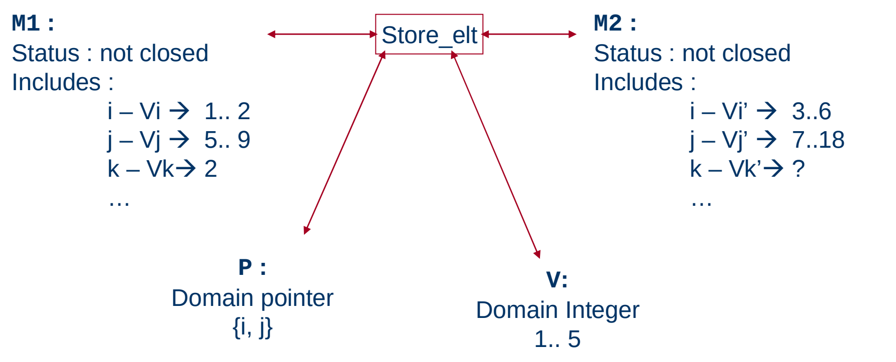
cell : Structure

Cont.
new(1)
w(1).new(2)
w(2).new(2)

Fig.4: An abstract memory on statement 10

Fig.5 shows an (artificial) invocation example of this constraint obtained after the translation of the following C statement **`*P = V`**. In the constraint model of INKA V2, **`P`** is stored in the pointer tableau and we suppose that the tool has deduced that its domain of only possible values is **`{i,j}`**. **`V`** is stored in the integer tableau and its possible values are 1,2,3,4,5 (also noted 1..5). Memories **`M1, M2`** are not closed as the exact number of allocated objects in each memory is unknown. Moreover, the value of **`k`** in **`M2`** is not constrained (noted with an "?"). In spite of this, a lot of deductions can be obtained during constraint propagation from this constraint on these two memories. For example, from the fact that the domains of possible values for **`Vi`** and **`Vi'`** do not intersect, the constraint deduces that **`i`** is the stored variable and then **`P`** points to **`i`** and **`V = Vi'`**. All other deductions are summarized in Fig.5.

`store_elt(P,V,M1,M2)`



Automatic deductions after the constraint propagation step :

P = i, V = Vi' in 3..5, Vj = Vj' in 7..9, Vk = Vk' =2

Fig.5: Constraint propagation in INKA V2

This example shows the potential interest of constraint reasoning in INKA V2 to deduce test data. The model is complex enough to deal with most C/C++ operations, including object-oriented ones such as inheritance, object allocation/deallocation, method invocation and overloading, namespaces and so on. However, some operations remain challenging to deal with in this constraint model. In particular, virtual method calls and templates that are the keystone of parametric polymorphism in C++ cannot be handled without additional treatments on object types. Other perspective work needs to be undertaken in order to address the challenging problem of generating test data for complex real-world C++ programs.

4. PERSPECTIVES IN CONSTRAINT-BASED WHITE-BOX TESTING

Constraint-based white-box testing is a very promising approach. It allows the tester to focus on functional test cases generation by automating the most-demanding task of white-box testing. As exemplified by the work we performed on INKA V2, constraint-based approaches have demonstrated a great potential for this application. In fact, constraints are suitable to handle some of the most complex C/C++ operations such as dynamic allocation and object creation, method invocations or floating-point computations. However, in our view, the main problem that remains in constraint-based white-box testing is scalability. The constraints model we designed in INKA V2 is accurate enough to provide exact results on most C/C++ operations but it is not currently scalable to programs that contain more than thousands lines of code. We believe that a key issue to address this problem will be to perform static analysis and combine its results within the constraints model. Static analysis has been shown to scale very well in many real-world situations by compromising the accuracy of the results. We believe that combining approximated results within an exact constraint model will be the next step in constraint-based white-box testing. Several research works have already been launched into this direction such as the INRIA Lande project initiative in the RNTL CAT in the combination of static and dynamic analysis within a constraint-based approach for verifying C programs. The extension of INKA to deal with JAVA programs is in progress in the RNTL DANOCOPS project. In this work, INKA core is used for a different purpose, to detect non-conformities between a program and its specification expressed with an assertion language.

[1] A. Gotlieb, F. Calvet, M. Rueher : Génération Automatique de Cas de Test : une approche Programmation Logique par Contraintes ; Le Génie Logiciel & ses Applications : 9èmes journées internationales, Paris La défense, also published in Génie Logiciel, Nov. 1996, EC2.
[2] Arnaud Gotlieb, Bernard Botella, Michel Rueher : Automatic Test Data Generation using Constraint Solving Techniques ; Software Engineering Notes, 23 (2) :53-62, Mar. 1998, ACM.
[3] INKA --V1 User's Manual; Axlog Ingenierie and Thales Airborne Systems, Déc. 2002
[4] B.Botella, A. Gotlieb, C. Michel : Symbolic execution of floating-point computations
Software Testing ; Verification and Reliability Journal (STVR), Wiley & Sons, Jun. 2006
[5] A. Gotlieb, T. Denmat, B. Botella : Constraint-based test data generation in the presence of stack-directed pointers ; 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'2005), Long Beach, USA, Nov. 2005
[6] A. Gotlieb, B. Botella : Automated Metamorphic Testing ; 27th IEEE Annual International Computer Software and Applications Conference (COMPSAC'2003), Dallas, USA Nov. 2003
[7] A. Gotlieb : InKa: An Automatic Software Test Data Generator ; DAta Systems In Aerospace (DASIA 2001), Eurospace, The Association of European Space Industry, Nice, FRANCE May 2001