Automated Software Engineering manuscript No.
(will be inserted by the editor)

Behind the Scenes in SANTE: A Combination of Static and Dynamic Analyses

Omar Chebaro · Pascal Cuoq · Nikolai Kosmatov · Bruno Marre · Anne Pacalet · Nicky Williams · Boris Yakobowski

the date of receipt and acceptance should be inserted later

Abstract While the development of one software verification tool is often seen as a difficult task, the realization of a tool combining various verification techniques is even more complex. This paper presents an innovative tool for verification of C programs called Sante (Static ANalysis and TEsting). We show how several tools based on heterogeneous techniques such as abstract interpretation, dependency analysis, program slicing, constraint solving and test generation can be combined within one tool. We describe the integration of these tools and discuss particular aspects of each underlying tool that are beneficial for the whole combination.

O. Chebaro
ASCOLA (EMN-INRIA, LINA), École des Mines de Nantes, 44307 Nantes France
E-mail: firstname.lastname@mines-nantes.fr

P. Cuoq
CEA, LIST, Software Safety Laboratory, PC 174, 91191 Gif-sur-Yvette France
E-mail: firstname.lastname@cea.fr

N. Kosmatov
CEA, LIST, Software Safety Laboratory, PC 174, 91191 Gif-sur-Yvette France
Tel.: +331 69 08 71 83, Fax: +331 69 08 83 95
E-mail: firstname.lastname@cea.fr

B. Marre
CEA, LIST, Software Safety Laboratory, PC 174, 91191 Gif-sur-Yvette France
E-mail: firstname.lastname@cea.fr

A. Pacalet
work done at: INRIA-Sophia-Antipolis, BP 93, 06902 Sophia Antipolis, France
now at: SafeRiver, 55 rue Boissonade, 75014 Paris, France
E-mail: firstname.lastname@safe-river.com

N. Williams
CEA, LIST, Software Safety Laboratory, PC 174, 91191 Gif-sur-Yvette France
E-mail: firstname.lastname@cea.fr

B. Yakobowski
CEA, LIST, Software Safety Laboratory, PC 174, 91191 Gif-sur-Yvette France
E-mail: firstname.lastname@cea.fr

Keywords C program verification · SANTE tool · Frama-C toolset · static analysis · program slicing · test generation · constraint solving

1 Introduction

Modern software engineering essentially relies on a wide range of automated tools, going from requirements engineering and constructing the initial model up to evaluation and deployment of the final product. Among the most sophisticated tools are those for software verification, reconciling complex mathematical methods for difficult, often undecidable, verification problems, solutions for their optimal implementation and appropriate design facilitating their usability and integration into the complete software engineering process. Such tools become even more complex when they combine different techniques, or several independent tools, with specific features and limitations for each of them. The architectural design of these new integrated tools plays a crucial role in the whole combined product.

Nobody could imagine in the early 1970's, when the first version of the C programming language was developed by Dennis Ritchie and Kenneth Thompson, that some forty years later one of the numerous verification tools for C programs would include such heterogeneous techniques and paradigms as the following:

- program parsing and abstract syntax tree (AST) construction,
- abstract interpretation,
- value, pointer and alias analysis,
- intra-procedural data and control dependency analysis,
- inter-procedural dependency analysis,
- dependence-based program slicing,
- constraint solving,
- constraint logic programming,
- program instrumentation,
- test generation combining symbolic and concrete program execution.

Indeed, most of these techniques are much younger than C itself, and originally were not developed in order to cooperate with the others.

This paper presents an innovative verification tool, called SANTE, where all these techniques are combined within one tool for verification of C programs. The SANTE[1] (Static ANalysis and TEsting) tool is implemented inside FRAMA-C [25,17], an open-source platform dedicated to analysis of C programs and developed at CEA LIST. FRAMA-C integrates various analyzers sharing a common specification language called ACSL (ANSI/ISO C Specification Language) [4].

In this paper we only give a brief presentation of the SANTE method and focus on the tool aspects of its implementation. The SANTE tool strongly relies on powerful underlying tools for each of its steps, that sometimes solicit

[1] The French word *santé* means *health,* and sometimes also *Cheers!*

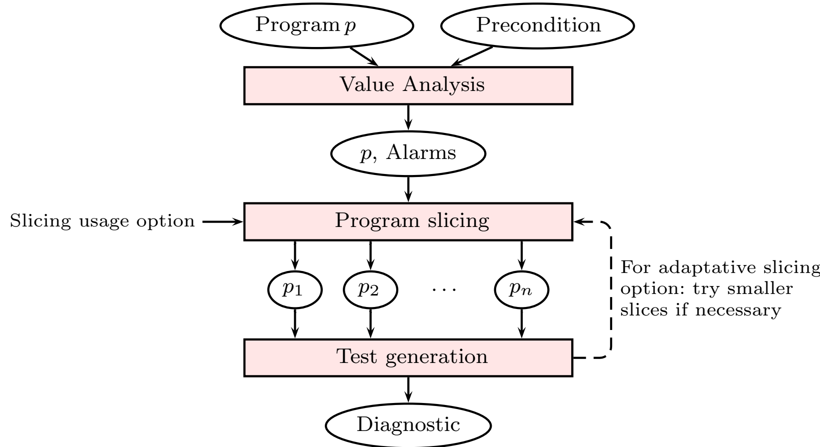

Fig. 1 Overview of the SANTE method

in turn other tools for specific analyses. Our first objective is to emphasize the particular technical and implementation aspects of these tools that made them particularly interesting for using in SANTE. Secondly, we discuss the architecture and tool integration issues that appeared particularly beneficial for efficient, maintainable and well-structured implementation. Finally, we evaluate the SANTE tool from both technical and architectural points of view, illustrating the interest of the combined verification method and the advantages of the adopted tool integration solutions.

The paper is organized as follows. Section 2 briefly introduces the SANTE method. Section 3 presents each of the tools used by SANTE. Next, in Section 4 we discuss integration issues of all these tools within one tool inside the global framework of FRAMA-C. Section 5 evaluates the combined tool by experiments and on the basis of our implementation feedback. Section 6 presents related work. Section 7 concludes and outlines several future work directions.

2 Overview of the SANTE method

This section provides an overview of the SANTE verification method for single-threaded C programs proposed by Chebaro et al. [12,13,14]. For simplicity, we do not consider here the case of non-terminating programs representing some theoretical difficulties (briefly discussed in Sections 3.2.2, 3.3.2) which are not in the scope of this paper. The method is illustrated by Fig. 1. Its inputs are a C program p and its precondition which defines value ranges for acceptable inputs of p and relationships between them.

At the first step, a value analysis produces a set of *alarms* A reporting threatening statements in p for which it detects a risk of runtime error. The

current version of Sante treats the risks of division by zero, out-of-bounds array access and some cases of invalid pointers. The value analysis step uses the Value plugin of Frama-C.

The objective of the second step, based on program slicing, is to simplify the initial program before the last step. According to the user-defined slicing option and the structure of dependencies in A, this step determines which and how many simplified programs (*slices*) should be generated and sent to dynamic analysis. Each simplified program contains a subset of alarms that can be triggered. The second step uses the Pdg and Slicing plugins of Frama-C.

Finally, for each simplified program p_i, the dynamic analysis step tries to activate the potential threat for each alarm present in p_i. It may generate a *counter-example* for an alarm, i.e. a test case showing that the instruction reported by the alarm is not safe and provokes a runtime error. Test generation allows Sante to produce for each alarm a *diagnostic* that can be *safe* for a false alarm, *bug* for an effective bug confirmed by some input state, or *unknown* if it does not know whether this alarm is an effective error or not. We say that an alarm is *classified* if its diagnostic is bug or safe. This step uses the PathCrawler plugin of Frama-C.

Several slicing options are available in Sante, each of them reflecting a different usage of program slicing. A trivial one skips program slicing, and sends the initial program $p_1 = p$ to the dynamic analysis step. Another option applies program slicing only once to generate a unique simplified program p_1 containing all alarms of A. The disadvantage of this usage is that test generation may lack time or space, and alarms that could be easier to classify in a smaller slice are penalized by the analysis of a bigger slice containing more complex alarms. Another option performs program slicing with respect to each alarm separately. It produces n slices $p_1, \ldots, p_n$ where $n = \text{card}(A)$. Since test generation is executed separately for each slice, there is a risk of redundancy and waste of time with this option if some alarms are included in several slices.

To obtain a better compromise between a bigger number of smaller slices to be analyzed by the costly dynamic analysis step and a smaller number of bigger slices where some alarms can remain unclassified, the Sante method proposes to optimize the number of slices (and test generation sessions). It exploits the dependencies between alarms in advanced usages of program slicing. In these usages, the slicing is performed with respect to subsets of alarms selected in a specific way in order to obtain a reasonable trade-off between the number and the size of the slices. Moreover, one of the advanced options proposes an iterative verification process, where smaller slices are generated and sent to dynamic analysis as long as there is a chance to classify more alarms on these smaller slices. The slicing options of the Sante method are described in detail and discussed in [14].

3 Underlying tools

In this section, we present the underlying tools and techniques used by SANTE: value analysis (VALUE), dependency analysis (PDG), program slicing (SLICING), constraint solving library (COLIBRI) and test generation tool (PATHCRAWLER). We discuss specific features and design solutions appearing to be particularly valuable for SANTE or enlarging their applications in other contexts.

3.1 Abstract interpretation based value analysis

The FRAMA-C plugin for value analysis [11,22], called VALUE, is loosely based on the principles of Abstract Interpretation [19]. Abstract interpretation offers a sound approximation of the behavior of a program. To do so, it links a *concrete semantics* (the set of possible executions of the program in all possible execution environments), to an *abstract semantics.* Since the concrete semantics is in general undecidable, the abstract semantics is chosen more coarse-grained. Both semantics are related by *abstraction* and *concretization functions.* Those functions must be chosen *sound:* any transformation in the concrete semantics must be such that its counterpart in the abstract semantics captures all possible outcomes of the concrete operation.

Static analyzers proceed by symbolic execution of the program, translating all operations into the abstract semantics. If the abstract semantics uses infinite domains, *widening* operations are introduced to ensure the convergence of the analysis. Together with the soundness of the abstraction and concretization functions, this ensures that the analysis terminates, and that it returns an over-approximation of all concrete behaviors of the program.

3.1.1 Alarms

Value analysis computes an (over-approximated) set of possible values of each variable at each point of the program. In particular, it makes it possible to check whether an operation that can lead to an error at runtime (like a division by zero, or an out-of-bounds access) is safe, by verifying the range of the involved expression (respectively, the denominator of the division, the offset of the pointer access) at the relevant program point. If the abstract semantics guarantees that the undesirable values cannot occur, we have statically proved that the execution of the operation will always succeed at runtime.

If the undesirable values cannot be excluded, value analysis reports a possible error by an *alarm*, expressed as an assertion that must be verified to avoid the error. In this case, there are two possibilities. First, this alarm may signal a real error: the operation fails at runtime on at least one execution. Alternatively, when the reported error can never occur at runtime, we have a *false alarm,* which stems from the difference in precision between the concrete

and abstract semantics. Using more precise domains for the abstract semantics typically results in less false alarms, at the expense of longer analysis time.

In Frama-C, alarms take the form of ACSL assertions. As an example, an access `t[i]` to index `i` of an array `t` of size 15 gives rise to the assertion assert $0 \leq i < 15$. Those alarms can afterwards be read and checked by other Frama-C plugins.

Upon emitting an alarm, the analyzer also reduces the propagated state accordingly. Typically, in the program below, at most one alarm is emitted for the access to `t[i]`: if the first instruction evaluates correctly, the variable `i` is necessarily within the proper range for the second instruction.

```
x = t[i] + 1;
y = t[i] - 3;
```

3.1.2 Abstract domains

This section describes the (non-relational) abstract domains propagated by Value to represent the program state. A more complete description of this hierarchy is given by Cuoq et al. [22].

Integer computations. Small sets of integers are represented as sets, whereas large sets are represented as intervals with congruence information [37]. Congruences are typically useful to represent offsets inside arrays of structures. For example, in the program below, the array takes 160 bytes, and the offsets x (expressed in bytes) in the array `t` of the fields `i2` have the form $x = 4j + 1$ for some $j \in \{0, 1, 2, \ldots, 39\}$, that can be equivalently written as $1 \leq x \leq 157$ and $x \equiv 1 \pmod 4$.

```
1 struct s {
2   char i1; char i2; char i3; char i4;
3 };
4 struct s t[40];
5
6 void main(unsigned short k) {
7   char *p = &t[k+2].i2;
8   *p = 1;
9 }
```

Since `k+2` ≥ 2, the first two elements `t[0]` and `t[1]` cannot be addressed by `p`, and there is an out-of-bounds access when `p` is dereferenced at line 8 for `k+2` $\geq$ 40. So Value generates an assertion `assert \valid(p)` before the assignment at line 8, and assumes afterwards that the address `p` of the byte modified at line 8 is valid. Its possible offset x in `t` is computed as `[9..157],1%4`, which means $9 \leq x \leq 157$ and $x \equiv 1 \pmod 4$.

Floating-point computations. The results of floating-point computations are represented as IEEE 754 [41] finite intervals. All abstract operations take into account IEEE 754-specified rounding for the program's precision (single or double). The analyzer by default assumes round-to-nearest-even mode, with

an option to capture other IEEE 754 rounding modes and C implementations that compute intermediate results at a precision other than indicated by the expression's type[42, §5.2.4.2.2:8].

In the following program, since the constant 1.6 cannot be represented exactly as a C `float` and because of possible rounding errors, the possible interval for f after line 3, taking into account all possible rounding modes, is $f \in [3.59999990463..5.60000038147]$.

```
1 void main(float f) {
2   //@ assert 2.0 <= f <= 4.;
3   f = f + 16.0 / 10;
4 }
```

Obtaining infinities or NaN as results of floating-point computations is treated as undesirable errors. This is a conservative (sound) behavior, albeit a potentially restrictive one, as some valid programs may be marked as potentially invalid.

Pointers. The analyzer reasonably assumes that the program does not purposely use buffer overflows to access neighboring variables. The memory model employed to represent the state of a C program reflects this assumption: addresses are represented as offsets with respect to *base addresses*, that are always manipulated symbolically, without consideration for the actual location of the base addresses in the concrete virtual memory space during execution. The assumption does not compromise soundness: an alarm is emitted for any invalid (e.g. out-of-bounds) array or pointer accesses. If all such alarms are verified, then the assumption is effectively guaranteed to hold.

Addresses are represented as maps using base addresses as keys. In the map representing the set of values a particular pointer may have in the C program, the sets of integers associated to each base address represent the possible offsets with respect to this base. That is, `{{ &t+[2..159] ; &u+{24} }}` means all addresses at an offset between 2 and 159 bytes starting from the base address of the variable `t`, or the address "base address of `u` plus 24 bytes".

The memory representation is untyped. This makes it straightforward to handle unions and heterogeneous pointer conversions during abstract interpretation. In an abstract address, the set of integers associated to a base is always interpreted as an offset in bytes, regardless of the type of the pointer and of the type of the base pointed into. A pointer to the second element of an array `t` of integers, represented as `t + 2` in the C program, is abstracted as `{{ &t+{8} }}` when the analyzer is configured to target a 32-bit architecture, where `sizeof(int)=4`.

Memory. Following the (verified) assumption that base addresses are separated, an abstract memory state maps each base address to a representation of a chunk of linear memory. For each such chunk, a memory state maps ranges of bits in the chunk to values [8]. While byte-level reasoning is sufficient for pointers (since a byte is the smallest unit of memory addressable by a C pointer), bit-level reasoning is needed to handle bit-fields [42, §6.2.6.1:4],

which compilers usually do not align on byte frontiers when a more compact layout exists.[2] Thus, using bit-expressed offsets is needed to remain compatible with the layout strategy of standard compilers. Byte-level offsets are transparently scaled into bit-level ones when needed by multiplying the former by 8 (i.e. `sizeof(char)`).

The choice of intervals as keys, as opposed to an abstract representation for "i^{th} array cell" or "field `f` of struct `s`", is intended to make practical the handling of *type-punning*, i.e. the action of reading or writing values in a non type-safe way.[3] As an example, consider the code below.

```
unsigned int t[10];

void main(unsigned int x) {
  t[0] = 0x000000F0;
  t[1] = 0x000007F0;
  unsigned char c = *((char *)t+5)+1;
}
```

After the first two instructions, assuming a 32-bit representation for integers, the memory is abstracted as

$$\mathtt{t} \mapsto \begin{cases} [0..31] \mapsto \mathtt{0x000000F0} \\ [32..63] \mapsto \mathtt{0x000007F0} \end{cases}$$

To read from `*(((char*)t) + 5)`, the analyzer determines that the relevant value is to be found between the bits 40 to 47 of `t`; thus the binding at key [32..63] is relevant in computing the result. The analyzer then extracts bits 8 to 15 from the value `0x000007F0` bound to this interval, resulting in the singleton value `0x07` (assuming a little-endian representation). At the end of the analysis, `c` is exactly equal to `0x08`.

The C standard uses the adjective *indeterminate* to refer to the contents of memory locations. This should not be confused with *undefined behaviors* caused by illegal computations. In the example below, the contents of pointer `p` after a call to function `f` are indeterminate, but the program is defined as long as it does not read these indeterminate contents.

```
int *p;

void f(void) {
  int l;
  p = &l;
}
```

Having indeterminate contents in memory is not an undefined behavior, but the C99 standard's intention is that accessing an indeterminate memory location is. Examples include reading the contents of uninitialized local variables, padding, and dangling pointers such as `p` after a call to `f` above.

[2] The C standard itself does not specify the layout of bit-fields, but then again, it does not specify the layout of any kind of data.

[3] Type-punning is allowed when the lvalue used to access is of type char [42, §6.5.7] or a union type (this has been made explicit in Technical Corrigendum 3 footnote 82 [42, §6.5.2.3]).

VALUE detects such programming errors. In order to do so, the values used to represent a memory state are not directly from the abstract domain used for the values of an expression, but from the lattice product of this domain with two-valued domains, one for initializedness and padding, and the other for danglingness.

Propagation of unjoined states. In order to improve precision, a user-settable option allows to postpone the join of abstract states during the analysis. When option `-slevel k` is set, up to `k` distinct abstract states are propagated unjoined through each statement. By setting `k` to a high enough value, finite loops can be unrolled entirely. Successive conditionals are also handled more precisely. For a large part, this alleviates the need for relational domains, many relations ending up encoded implicitly in the disjunction of abstract states.

3.1.3 Applications

VALUE has already been used with great success in both academic and industrial contexts. Berthome et al. [6] propose a source-code model for verifying physical attacks on smart cards. Pariente and Ledinot [59] verify flight control system code using a combination of FRAMA-C plugins, including VALUE and SLICING (§3.3). Yakobowski et al. use VALUE to check the absence of runtime errors in a 50 kloc instrumentation and control (I&C) nuclear software [24]. Delmas et al. use VALUE to verify the control and data flows in a DO-178B-certified aeronautics development [22].

3.2 Program dependency graph

In order to provide an (intra-procedural) view of the dependencies between the different constructs of a function, FRAMA-C uses a plugin called PDG. While PDGs (Program Dependency Graph) [40] are typically a first step towards program slicing, the precise dependency information they offer can also be used for program optimization [31], or semantic code navigation [58].

The nodes of the PDG for a function `f` are essentially the instructions of `f`, plus some special nodes for the arguments of the function, the declaration of variables, etc. The four possible kinds of dependencies computed by PDG are the following:

Value dependencies are the most common. In an assignment `x = a + b`, there is a value dependency from `x` to `a` and `b`, as their values are needed to compute the new value of `x`.

Address dependencies are specific to languages with pointers, which include C. In an assignment `x = *p`, there is a dependency from the leftmost part of the instruction to the value of `p`.

Control dependencies stem from jumps in the control flow. Typically, for the program `if (c) x = a; L:` the value of `x` at point `L` depends on the value of `c`.

Declaration dependencies link a variable used in an instruction to its declaration in the source code, and indirectly to the declaration of its type.

In the PDG plugin, edges are computed differently depending on the kind of dependencies, as explained hereafter.

3.2.1 Data (value and address) dependencies

Although technically different, value and address dependencies are related and computed simultaneously by the PDG plugin, using a forward dataflow analysis on the control-flow graph of each function. Each data node of the PDG represents a value computation in the program. The propagated state is a map similar to the one used by VALUE for the representation of pointers (cf Section 3.1.2), but instead of mapping memory locations to values, it maps them to sets of PDG nodes. So at each program point, it is possible to find the PDG nodes that are used to compute the value of a location.

Alias information is needed, in order to compute value or address dependencies. For instance, after the sequence: `p = &x; x = a; y = *p;` `y` has an address dependency on `p`, and a value dependency on `a`, to which `*p` evaluates. VALUE's results are used to expand pointers into the range of their possible values, using the over-approximation of the memory states it computes.

3.2.2 Control dependencies

In structured langages, the general definition of control dependencies is given considering paths in the control flow graph: a node N depends on a test node T if some paths starting from T go to N, but not all. This means that whether N is executed or not depends on the branch chosen at T. However, in unstructured programs with unconditional jumps, such as the C `goto`, `break` and `continue` statements, control dependencies become harder to compute. Even though there is only one branch starting from an unconditional jump statement I, there is a form of control dependency from I to the following statement in the code S. Indeed, it the jump I was not present, the control would flow directly to S.[4] This issue has been identified in the literature, and we use the solution proposed in [15]. We consider I as an `if` with one branch going to the normal successor in the control-flow, and the other one going to S. Although potentially imprecise, this is always correct [15].

Another difficulty lies in statement termination: every statement S that follows a statement E that might not terminate should have a control dependency on it since its execution depends on the termination of E. However, those dependencies would lead to very imprecise results. As a design choice, the PDG plugin does not add them, and special care must be taken by plugins that exploit the PDG and want to deal with non-terminating programs.

[4] In particular, for program slicing, without a dependency S→I, I can be sliced out, while S is still present in the slice. It may introduce new control flow not present in the original program, and lead to incorrect slices [15, 68].

```
1 int abs(int x) {
2   int ret;
3   ret = x;
4   if (x < 0)
5     ret = -x;
6   return ret;
7 }
```

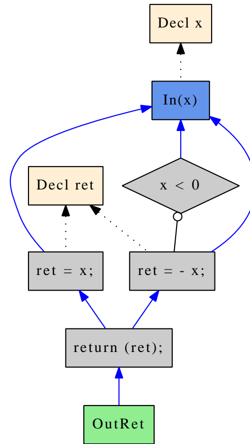

Fig. 2 Example of a PDG

3.2.3 Example of a program dependency graph

A function `abs` and its PDG are shown in Fig. 2. Data (resp. declaration, control) dependencies are represented as plain (resp. dotted, circle-ended) edges. There is a single control dependency, between lines 4 and 5, as the latter gets executed only if the condition `x < 0` holds. There are three data dependencies (lines $3, 4, 5$) on the initial value of the formal variable `x`; this initial value is materialized by the node `In(x)`. The assignments to `ret` at lines 3 and 5 give rise to two declaration dependencies on this variable. However, the dependencies on the declaration of `x` are not needed at lines $3, 4, 5$, as they are implied by the data dependencies on `In(x)`, which itself depends on the declaration of `x`. Finally, the `OutRet` node represents the output of the function, hence the data dependency on the value being returned.

3.2.4 Function calls

The PDG plugin builds a unique PDG for each function, regardless of the number of contexts the function is called in. This is a precision trade-off, as building fully contextual PDGs is very costly in terms of computation and memory space.

Within the PDG for a function `g`, a call to a function `f` is represented in an abstracted form. Only the inputs and outputs of `f` exist as PDG nodes, with data dependencies edges linking the outputs to the inputs that are used to compute them. Typically, a call to the function `f` below

```
void f() {
  z = y;
  w = x;
  z = z + w;
```

```
}
```

will be summarized by four nodes and three edges:

- two inputs nodes `In(x)` and `In(y)`, for `x` and `y`;
- two output nodes `Out(z)` and `Out(w)`, for `z` and `w`;
- three dependency edges from `Out(z)` to `In(x)`, from `Out(z)` to `In(y)`, and from `Out(w)` to `In(x)`.

Those data dependencies are called *functional dependencies* [26], and computed by the plugin From of Frama-C (using the results of Value). Notice there is a link between those nodes and the PDG of the called function: they consist in the transitive closure of the data dependency edges of the PDG, retaining only the input and output nodes.

While this representation is concise and compositional, it can lead to some precision loss, typically for functions receiving pointers as arguments. Consider the code below.

```
int a, b;

void f(int *p) {
  (*p)++;
}
```

```
void main() {
  f(&a);
  f(&b);
  return b;
}
```

The PDG nodes for a call to `f` in the PDG of `main` would include two input nodes `In(a)`, `In(b)` and two output nodes `Out(a)`, `Out(b)`. There would be four data dependencies, from `Out(a)` to `In(a)` and `In(b)`, and from `Out(b)` to `In(a)` and `In(b)`. This is an unfortunate byproduct of evaluating pointers to all their possible values — and things would get more intermingled with a third call, say `f(&c)`.

Still, it is possible to recover the aforementioned form of context-sensitivity, leading to simpler, more precise PDG. The plugin From can optionally produce *callwise* results, i.e. one result per function call. In our example, it correctly infers that the call `f(&a)` does not involve `b` at all. Thus, in the PDG of `main` this call is represented by only two nodes `In(a)` and `Out(a)` with a single data dependency from `Out(a)` to `In(a)`. Likewise, the call `f(&b)` only requires two `b`-labeled nodes. The resulting PDG for `main` is much more precise, as it shows that the two calls are independent (they operate on different memory zones), and could in fact be reordered. Such context-sensitive PDGs are useful to gain precision for programs calling functions that take pointers as arguments, with different values for different calls, in particular, `memcpy` or `strlen`.

3.3 Dependence-based program slicing

Program slicing [69, 70] consists in the computation of the set of program instructions (the *program slice*), that may influence the program state at some point of interest (referred to as a *slicing criterion*). As an example, a slice of

a program at a given point `p` and a given variable `x` corresponds to the set of statements that might affect the value of said variable `x` at point `p`.

From a programming standpoint, program slicing is a very useful tool, as it can considerably simplify the portion of code that must be considered. It can be used e.g. for improving program comprehension, debugging or refactoring code, etc. For example, the SLICING plugin of FRAMA-C automatically discards function arguments that are not related to the current slicing criterion (see e.g. [61] for such a transformation on functional programs). This is reminiscent of *amorphous slicing* [51]. Also, the SLICING plugin optionally performs *specialization*: the body of a function can be duplicated, and sliced differently for different call sites. To the best of our knowledge, no other full-scale slicing tool offers this facility.

Finally, in the context of FRAMA-C, the program slice should still be a compilable, executable program on its own, and well-defined; indeed FRAMA-C's various analyses must be usable on the resulting program. FRAMA-C's slicing is static, as it is built on top of the results of PDG and VALUE— that are themselves static analyses.

3.3.1 Slicing criteria

Within the SLICING plugin of FRAMA-C, a wide range of slicing criteria can be given, either regarding code observation or logical properties of the program. In the case of code observation, several elements can be marked as elements to be preserved in the resulting slice. Slicing can be made on:

- program statements;
- function calls and returns;
- read and write accesses to selected left-values of the code;
- values of a global variable after the execution of the `main` function;
- logical annotations.

Slicing on function returns (e.g., for functions `f1`, . . . , `fn`) ensures for example that each time these functions do indeed return in the original code, their sliced counterparts also terminate with the same return value.

The SLICING plugin of FRAMA-C also has the ability to be used on logical properties specified in ACSL. In this case, properties verified by the sliced code are ensured to be verified by the initial code. This can be used to slice function assertions, loop variants and invariants or threats emitted by the value analysis plugin. This last possibility is the one used by the SANTE tool.

Finally, SLICING is able to handle a conjunction of atomic slicing criteria. The resulting slice is in general different from a superposition of all the individual slices, as more statements may need to be kept. By construction, the slice will verify all the slicing criteria simultaneously.

3.3.2 Slicing in case of non-termination

SLICING is sound in presence of potentially infinite loops, according to the following semantics: all execution traces that reach the criterion in the original

```
int X, Y;

void g (int x, int y) {
  X = x;
  Y = y;
}

int fX (int x, int y) {
  g (x, y);
  return X;
}
```

```
int fY (int x, int y) {
  g (x, y);
  return Y;
}

int main (int i1, int i2,
          int i3, int i4)
{
  return
     fX (i1, i2) + fY (i3, i4);
}
```

Fig. 3 Original program

```
int X, Y;

void g__2(int x) { X = x; }
void g__1(int y) { Y = y; }

int fX(int x) {
  g__2(x);
  return X;
}
```

```
int fY(int y) {
  g__1(y);
  return Y;
}

int main(int i1, int i4) {
  return fX(i1) + fY(i4);
}
```

Fig. 4 Program slice with specialization

program will exist in the sliced one, and will validate the criterion identically. This however means that a criterion that involves an instruction `S` located after a potentially infinite loop will get interpreted for all executions that reach `S`, i.e. when the loop terminates. In those cases, special care may be required to keep the reason for non-termination. Typically, this can be done e.g. by adding to the slicing criterion the `while (1)` or `while (c)` statement of the loop. In this paper, we focus on the version of the SANTE tool that treats terminating programs, so the issue for non-terminating programs is not discussed here further.

3.3.3 Computing the slice

Once a PDG for the given program has been computed, the problem of slicing is partially reduced to the problem of the reachability of a given node in this graph [40]. However, the implementation of the slicing part itself is highly non-trivial, for at least two reasons:

1. the PDG is intra-procedural, thus the slicing must handle all the inter-procedural part of the computations;
2. in presence of multiple calls to the same function, the slicing may decide to keep statements that are not relevant for one particular call.

The slicing plugin associates a mark to every PDG node. Initially, the mark *bottom* is given to all nodes, except those that are directly relevant to the slicing criterion. Then, the non-bottom marks are propagated backwards,

following the PDG. For example, if an instruction `x = a + b` in a function `f` is marked as needed, the PDG is queried to get the nodes of all the instructions that directly influence the values of `a` and `b`. Those nodes are then marked as needed, and the process is iterated. If no such node exists within `f`, we consult the PDG of all its callers, above the instructions where `f` is called.

During this process, we may decide to keep some instructions or sub-instructions only because we want a compilable program. Consider for example the program in Fig. 3, where the slicing criterion is the value returned by `main`. We need to keep both arguments of `g`, because `x` is needed for the call to `fX`, and `y` for the one to `fY`. However, the argument `y` in the call to `g` in `fX` is not useful, and conversely for `x` in `fY`. As a result, the slicing marks `i2`, `i3`, `y` in `fX` and `x` in `fY` as *spare* – meaning "not relevant for the criterion, only for compilation". Although this is not visible in the program slice, the difference can be observed on the original code, in Frama-C's graphical user interface.

As mentioned above, one can also ask for the specialization of some functions. In the code of Fig. 3, this is especially useful for `g`, which is called twice. The resulting slice is shown in Fig. 4, with some minor reformatting; no spare marks remain. Together with the fact that the returned code is executable, this functionality makes of the Slicing plugin of Frama-C a very precious tool to analyze various aspects of given code.

3.4 The Colibri constraint solving library

Constraint solving techniques are widely recognized as a powerful tool for Validation and Verification activities such as test data generation or counter-example generation from a formal model [52,53,47], program source code [35, 34,56] or binary code [2]. A constraint solver maintains a list of posted constraints (*constraint store*) over a set of variables and a set of allowed values (*domain*) for each variable, and provides facilities for constraint propagation (*filtering*) and for instantiation of variables (*labeling*) in order to find a solution.

In this section we present the Colibri library (COnstraint LIBrary for veRIfication) developed at CEA LIST and used inside the PathCrawler tool for test data generation purposes. The variety of types and constraints provided by Colibri makes it possible to use it in other testing tools at CEA LIST like GATeL [53], for model based testing from Lustre/SCADE, and Osmose [2], for structural testing from binary code.

3.4.1 General presentation

Colibri provides basic constraints for arithmetic operations and comparisons of various numeric types (integers, reals and floats). Cast constraints are available for cast operations between these types. Colibri also provides basic procedures to instantiate variables in their domains making it possible to design

different instantiation strategies (or *labeling procedures*). These implement specific heuristics to determine the way the variables should be instantiated during constraint resolution (e.g. a particular order of instantiation) and the choice of values inside their domain (e.g. trying boundary or middle values first). Thus the three aforementioned testing tools have designed their own labeling procedures on the basis of COLIBRI primitives.

The domains of numerical variables are represented by unions of disjoint intervals with finite bounds (no infinities or NaNs): integer bounds for integers, double float bounds for reals, and double/simple float bounds for double/simple floating point formats. These unions of intervals make it possible to accurately handle domain differences. For each numeric type and each basic unary/binary operation or comparison, COLIBRI provides the corresponding constraint.

Moreover, for each arithmetic operation, additional filtering rules apply algebraic simplifications, which are very similar for integer and real arithmetics, whereas floating arithmetics uses specific rules. Here are a few examples of basic algebraic simplifications.

- *Factorization of constraints.* If the constraint store contains $A + B = C$ and a new contraint $A + B = X$ is added (or derived) in the store, then only one of these two constraints remains in the store and the variables C and X are *unified*, that is, they are identified and their domain is set to the intersection of the domains of C and X.
- *Special values (neutral elements, absorbing elements).* The constraint $A + 0 = X$ leads to the unification of A and X.
- *Identities between arguments.* The constraint $A + A = C$ is transformed into $2 * A = C$ which is more precisely handled by interval arithmetics. The constraint $A + X = A$ in integer and real arithmetics leads to the unification of X with 0 (which is not valid in floating point arithmetics, cf Section 3.4.4).

3.4.2 Bounded and modular integer arithmetics

COLIBRI provides two kinds of arithmetics for integers: bounded arithmetics for ideal finite integers and modular arithmetics for signed/unsigned computer integers.

Bounded arithmetics is implemented with classical filtering rules for integer interval arithmetics. These rules are managed in the projection functions of each arithmetic constraint. Moreover, a congruence domain is associated to each integer variable. Filtering rules handle these congruences in order to compute new ones and maintain the consistency of interval bounds with congruences (as in [49]). The congruences are introduced by multiplications by a constant and propagated in the projection functions of each arithmetic constraint.

Modular arithmetics constraints are implemented by a combination of bounded arithmetics constraints with modulus constraints as detailed in [36].

Thus they benefit from the mechanisms provided for bounded integer arithmetics. Notice that using the unions of disjoint intervals for the domain representation makes it possible to precisely represent the domain of signed/unsigned integers. For example, consider the constraint $A +_{2^3} B = C$ over 3-bit unsigned integers where $A \in 2..4$, $B \in 4..7$ and $C \in 0..7$. This constraint is handled by the constraints corresponding to the bounded arithmetic expression $C = A + B - K * 8$ where $K \in 0..1$ represents the overflow status. The filtering of these constraints converges to the interval $C \in [0..3, 6..7]$ where the sub-interval $0..3$ of C is reached when there is an overflow (i.e. $K = 1$). The domain representation by compact intervals in this case would be less precise and result in a complete interval $C \in [0..7]$ without any reduction.

3.4.3 Example of test generation in bounded vs. modular arithmetics

Suppose that the constraints for the program path π executing the lines 3,4,5,6 in the program below are incrementally posted to COLIBRI in order to generate a test for this path.

```
1 int f(int x) {
2   int y;
3   if(x >= 0)
4     y = x + 1;
5   if(y < x)
6     y = 0;
7   return y;
8 }
```

Assume that logical variables X and Y represent the values of `x` and `y`, and the initial domain of the input is $X \in [MinInt..MaxInt]$. Posting the constraint $X \geq 0$ reduces the domain of X to $[0..MaxInt]$ in any kind of arithmetics. If bounded arithmetics (without overflows) is used, then posting the second constraint $Y = X+1$ activates filtering rules for the *plus* constraint and results in the following constraints and domains:

$$Y = X + 1, \quad X \in [0 \ .. \ MaxInt - 1], \quad Y \in [1 \ .. \ MaxInt].$$

Posting the third constraint $Y < X$ activates filtering rules again, and COLIBRI reports that the constraints have no solution. The desired path π cannot be activated without overflows.

If modular arithmetics (with authorized overflows) is chosen, then $MaxInt + 1 = MinInt$, thus posting the second constraint $Y = X + 1$ results in

$$Y = X + 1, \quad X \in [0 \ .. \ MaxInt], \quad Y \in [MinInt] \cup [1 \ .. \ MaxInt].$$

When the third constraint $Y < X$ is posted, COLIBRI finds the unique solution and generates a test $X = MaxInt$ activating the chosen path π.

3.4.4 Real and floating point arithmetics

Real arithmetics is implemented with classical filtering rules for real interval arithmetics where interval bounds are double floats. In real interval arithmetics each projection function is computed using different rounding modes for the lower and the upper bounds of the resulting intervals. The lower bound is computed by rounding downward, towards $-1.0\mathit{Inf}$ (i.e. $-\infty$), while the upper bound is computed by rounding upward, towards $+1.0\mathit{Inf}$ (i.e. $+\infty$). This enlarging ensures that the resulting interval is the smallest interval of doubles including all real solutions.

Floating point arithmetics is implemented with a specific interval arithmetics as introduced by Claude Michel in [55]. Notice that properties like associativity or distributivity do not hold in floating point calculus. The projection functions in this arithmetics have to take into account *absorption* and *cancellation* phenomena specific to floating point computations. These phenomena are handled by specific filtering rules allowing to further reduce the domains of floating point variables. For example, the constraint $A +_F X = A$ over floating point numbers means that X is absorbed by A. The minimal absolute value in the domain of X can be used to eliminate all the values in the domain of A that do not absorb this minimum. Thus, in double precision with the default rounding mode (called *nearest to even*), for $X = 1.0$ the domain of A is strongly reduced to the union of two interval of values that can absorb X:

$$[\mathit{MinDouble} \,..\, -9007199254740996.0,\ 9007199254740992.0 \,..\, \mathit{MaxDouble}].$$

Colibri uses very general and powerful filtering rules for addition and subtraction operations as described in [54]. For example, for the constraint $A + B = 1.0$ in double precision with the *nearest to even* rounding mode, such filtering rules converge to the same interval for A and B

$$[-9007199254740991.0 \,..\, 9007199254740992.0].$$

3.4.5 Implementation details

Colibri is implemented in ECLiPSe Prolog [62]. Its suspensions, generic unification and meta-term mechanisms make it possible to easily design new abstract domains and associated constraints. Incremental constraint posting with on-the-fly filtering and automatic backtracking to a previous constraint state provided by Colibri are important benefits for search-based state exploration tools, and in particular, for test generation tools.

To conclude this short presentation of Colibri, let us remark that the accuracy of its implementation relies a lot on the use of union of intervals and the combination of abstract domain filtering rules with algebraic simplifications. Experiments in [3] using SMT-LIB benchmarks show that Colibri can be competitive with powerful SMT solvers.

3.5 Concolic test generation with PathCrawler

PathCrawler [46] is a unit structural testing tool based on the Frama-C abstract syntax tree and the Colibri constraint solver. PathCrawler's principal functionality is to automatically generate test inputs to guarantee all-path coverage of the C function under test. It can also be used to satisfy other coverage criteria (like k-path coverage restricting the all-path criterion to paths with at most k consecutive loop iterations, branch coverage,...), to signal anomalies such as uninitialized variables or integer overflow, or to discover the symbolic calculation effected by each execution path. Below, we will mainly consider the use of PathCrawler to test all feasible execution paths.

PathCrawler [72, 73] is based on a method which was subsequently baptized *concolic*, i.e. it uses a combination of concrete test inputs and symbolic reasoning. Like other concolic tools (CUTE [63], PEX [67], SAGE [33], EXE [10], KLEE [9]) PathCrawler runs the program under test on each test case in order to recover a trace of the execution path. However, in PathCrawler's case actual execution is chosen over symbolic execution merely for reasons of efficiency and to demonstrate that the test does indeed activate the intended execution path. Unlike other concolic tools, PathCrawler does not use actual execution to recover the concrete results of calculations that it cannot treat. This is because these results can only provide an incomplete model of the program's semantics and PathCrawler aims for complete coverage of a certain class of programs rather than for incomplete coverage of any program. Indeed, incomplete coverage often enables many bugs to be detected but PathCrawler was designed for use in more formal verification processes where coverage must be quantified and justified. If a branch or path is not covered by a test, then unreachableness of the branch or infeasibility of the path must be demonstrated.

The use of PathCrawler for classification of alarms for potential runtime errors in the Sante method also demands completeness. If all feasible program paths of the program (or a program slice) are covered and the paths leading to the error state for an alarm present in the program (resp., in the slice) are all infeasible, then this alarm is a false alarm. Other concolic tools, that use approximated constraints and incomplete path exploration, cannot ensure that a path is infeasible. Simplifying the original program by program slicing increases the chances to fulfil a complete path exploration on a smaller slice and to confirm or infirm more alarms. Partial path exploration (stopped by a timeout or with a partial k-path criterion) cannot classify an alarm as a false alarm, but can still confirm it if a counter-example was found.

Another example of where completeness is necessary is the measurement of execution time. PathCrawler can be used to generate tests to activate all feasible execution paths, or a set of the *slowest* execution paths [74], in order to measure the effective execution time on the target architecture (or on a simulator) of each path. If certain hypotheses hold, then the longest execution time found by this method can be considered to be the maximum execution time of the program under test [71].

```
1  #define N 5
2  typedef int perm[N];
3  int getOrder(perm p){
4    perm q, t;
5    int i, k=1, isId;
6    for(i=0; i<N; i++)
7      q[i]=p[i]; //Now q = p
8    while(1){ //Here q = p^k
9      // Check if q = p^k = Id
10     // if so, we are done
11     for(i=0,isId=1;i<N && isId;i++)
12       isId = isId && (q[i]==i);
13     if( isId ) return k; // Done
14     for(i=0; i<N; i++)
15       t[i]=q[i]; // Now t = p^k
16     for(i=0; i<N; i++)
17       q[i]=t[p[i]]; // Now q = tp = p^(k+1)
18     k++;
19   }
20 }
```

Fig. 5 Function `getOrder` takes a permutation p of $\{0, 1, \ldots, N-1\}$ and returns its order

A final example is the detection of unreachable code. Some unreachable code can be detected by static analysis (e.g. value analysis, cf Section 3.1) but the only way to ensure detection of all instances of unreachable code is to demonstrate the reachability of each block of code, for example by generating a test to activate it. This requires complete statement coverage.

Consider the program of Fig. 5 studied in [44]. Given a *permutation* p of $0, 1, \ldots, N-1$, that is, a bijection $p : \{0, 1, \ldots, N-1\} \to \{0, 1, \ldots, N-1\}$, the function `getOrder` computes the *order* of p, that is, the smallest integer $k \geq 1$ such that p^k is identity Id. The algorithm is straightforward: the consecutive powers p^k, $k = 1, 2, 3, \ldots$, are computed and stored in q (lines 6–7, 14–17). As soon as $q = \text{Id}$, the order k is returned (lines 11–13). The statement at line 17 gives rise to constraints with symbolic array indices depending on the inputs, and all-path testing has to deal with these constraints. Kosmatov [44] shows that complete path coverage for programs with this kind of constraints is particularly difficult to achieve if path constraints are approximated, or concrete results try to replace exact constraint resolution. For example, CUTE fails to meet all-path coverage on this program already for $N = 5$ (program with 16 feasible paths). After being stopped by a 12-hour timeout, CUTE achieves 10% path coverage for $N = 7$ (with 62 paths) and 1% coverage for $N = 8$ (with 110 paths), while PathCrawler covers 100% paths within a few minutes [44]. Together with symbolic array indices, symbolic pointer offsets, double (multiple) pointers and complex arithmetic computations are other examples of features leading to complex constraints that are sometimes approximated by other concolic tools.

3.5.1 The limits to completeness

The theoretical limit to completeness of test coverage is the complexity of the underlying constraint resolution. All-path test generation is NP-hard (see e.g. [45, Sec. 4]) and indeed some programs will always give constraint systems which the solver cannot resolve (or demonstrate as inconsistent) in a reasonable time. When this occurs, the test generation tool can only interrupt constraint resolution after a certain time and report that the corresponding path is probably infeasible but that this cannot be demonstrated. This problem

is usually only posed by functions under test which implement certain types of numerical algorithm, in which the branch conditions involve the results of complex calculations.

Another type of limit to complete path coverage concerns the sheer number of feasible execution paths. This suffers from a combinatorial explosion in the presence of loops with input-dependent limits, cascades of conditional instructions, function calls, etc. If complete coverage is really necessary then the user must ensure that the code can be decomposed into reasonably small modules (in terms of control flow rather than lines of code).

3.5.2 Implementing completeness

Complete coverage, within the limits mentioned above, implies that PathCrawler must translate each and every C instruction (and call to a library function) into constraints over variable values (and library functions must be given complete stubs). Colibri can treat non-linear arithmetic and provides specialized constraints for modular integer arithmetic and floating-point arithmetic. Within PathCrawler, further specialized constraints have been developed to treat bit operations, casts, dynamic allocation, arrays with (any number of) variable dimensions and array accesses using variable index values. Indeed, array accesses using variable indices pose two different problems. In the case of the value on input of an element of an input array, treatment of a variable index is based on an *element* constraint that represents the fact that the element could be any one of the known array elements. In the case of arrays with variable dimensions, this constraint is dynamically updated to take further elements into account as necessary. The other problem concerns the *internal aliases* arising from successive operations on the same array (or memory zone) when one or more operations use variable indices. These aliases are treated by constructing the constraints to model each combination of equalities and disequalities of the elements involved in the successive operations. The different possibilities are explored one by one by assimilating these internal alias constraints to extra path predicate constraints.

The attempt to correctly treat all C instructions is ongoing but PathCrawler can already treat a large class of C programs.

3.5.3 Test generation with Constraint Logic Programming

Development of specialized constraints is facilitated by the fact that, unlike other concolic tools based on black-box SAT, linear or SMT solvers, PathCrawler and Colibri are both implemented in Constraint Logic Programming (CLP). As explained in Section 3.4.5, CLP, and in particular the Eclipse language [62] enables low-level control of constraint resolution. Path predicate constraints can be incrementally added to the solver, just activating the filtering mechanism which can detect some inconsistencies relatively cheaply (cf Sec. 3.4.3). Full constraint resolution is only activated to create a new test

case (or demonstrate infeasibility if a test case cannot be found) when a new path predicate is formed.

Thanks to CLP, the resolution of new path predicates has been optimized in PathCrawler. Indeed, when concolic test generation forms a new path predicate by negating the final branch condition of a feasible path predicate [73], the resulting constraint system has a specific property. The path prefix up to the negated condition is known to be feasible (because it formed part of the path covered by a previous test case) so any infeasibility must be due to the final, negated, condition. PathCrawler optimizes constraint resolution by interrogating the constraint store to find which variables are either directly constrained by the final branch condition, or indirectly constrained by being linked by another constraint to a directly constrained variable. The closure of the constraints linking the variables directly constrained by the negated condition is the minimal constraint system which must be solved. The other constraints in the path predicates form one or more other, completely separate, constraint systems. These other systems already have a solution in the previous test cases which covered the path prefix up to the negated condition. PathCrawler therefore uses the values from a previous test case for these variables and just tries to solve the minimal system. Similar optimizations based on constraint independence have been used by Cadar et al. [9]. CLP has also enabled the development of specialized labeling procedures (cf Section 3.4.1) based on the properties of the arithmetic and other, specialized, constraints.

To illustrate using CLP in PathCrawler, consider the following program `cmpVectors` that, given two vectors $V_1 = (X_1, Y_1)$ and $V_2 = (X_2, Y_2)$, checks if both coordinates of V_1 are equal to (or greater, or less than) the respective coordinates of V_2. Let us denote a program path by line numbers indicating for the sake of clarity by a "+" and "−" indices respectively true and false branches of conditional statements. Suppose PathCrawler has generated first the test case $(X_1, Y_1, X_2, Y_2) = (16, 18, 16, 15)$ executing the path $6^+, 7^-, 9^+, 10^+, 11$. PathCrawler explores program paths in a depth-first search. The next partial path to be covered, obtained by negating the final branch condition 10^+ (i.e. $Y_1 \geq Y_2$), is $6^+, 7^-, 9^+, 10^-$. Since the negated constraint is disjoint from X_1 and X_2, PathCrawler reuses the values for X_1 and X_2 from the previous test case and has to solve a smaller set of constraints $Y_1 < Y_2$. Suppose it generates a second test case $(16, 0, 16, 18)$ activating the path $6^+, 7^-, 9^+, 10^-, 12^+, 13^+, 14$. Next it tries and fails to find a test case for infeasible partial path $6^+, 7^-, 9^+, 10^-, 12^+, 13^-$, and so on.

CLP offers an efficient backtracking mechanism which is also used to optimise PathCrawler's performance. Indeed, concolic test generation is based on the opportunistic enumeration of successive paths in order to treat the relatively simple constraint system representing the predicate of a single feasible path, rather than attempting to treat the more complex system representing all paths at once. PathCrawler explores the binary tree of all feasible execution paths in source code which has been simplified and expanded so as to decompose multiple conditions, inline function calls, unroll loops, etc,

```
1 typedef unsigned int uint;
2
3 char cmpVectors(
4     uint x1, uint y1,
5     uint x2, uint y2){
6   if(x1==x2)
7     if(y1==y2)
8       return '='; // V1=V2
9   if(x1>=x2)
10    if(y1>=y2)
11      return '>'; // V1>V2
12  if(x1<=x2)
13    if(y1<=y2)
14      return '<'; // V1<V2
15  return '!'; // otherwise
16 }
```

Fig. 6 Function `cmpVectors` compares two given vectors $V_1 = (X_1, Y_1)$ and $V_2 = (X_2, Y_2)$

and even enumerate possible internal alias combinations, as described above. The atomic constraint attached to each node in this tree of feasible execution paths is first treated with the truth value it has in at least one feasible path and then one single attempt is made to negate it. As a result, the smallest possible number of calls to full constraint resolution is made. Moreover, the built-in backtracking mechanism of CLP automatically stores, on the stack, the constraint system associated with the path prefix leading to each node so that it does not have to be reconstructed to explore negation of the constraint attached to the node. In the previous example, the backtracking mechanism ensures that, once posted, the constraint $X_1 = X_2$ (line 6^+) will not be withdrawn until all paths starting with 6^+ have been explored, so constraint solving for the following test cases does not have to treat it again. The shortest infeasible path prefixes are detected before any time has been wasted developing and exploring their ramifications. So, the solver will never be called for (structurally possible) partial paths $6^+, 7^-, 9^-, 12^+, 13^+$ or $6^+, 7^-, 9^-, 12^-$ in `cmpVectors` since a shorter partial path $6^+, 7^-, 9^-$ will be detected as infeasible. PathCrawler's treatment of many constructions is based on the same philosophy of treatment of what is known to be feasible, whilst creating CLP backtracking *choice points* on the stack, so that the other possibilities can be subsequently explored.

3.5.4 Adaptation of the all-path test generation

This section illustrates a simple program instrumentation performed by Sante thanks to the services provided by Frama-C. In Sante, the role of test generation is to activate potential threats, i.e. to cover execution paths in which the associated alarms are triggered. Technically, in order to force test generation to activate potential errors on each feasible program path in a program p, we add special *error branches* into the source code of p in the following way. For each alarm, the corresponding statement, say

```
  threatStatement;
```

is automatically replaced by the following branching statement:

```
if( errorCondition )
  error();
else
```

```
threatStatement;
```

where the condition determines if the error reported by the alarm occurs. Test generation is then executed for the C program with error branches denoted p'. We call this technique *alarm-guided test generation*. If the error condition is verified in p', a run-time error can occur in p, so the function `error()` reports the error and stops the execution of the current test case. If there is no risk of run-time error, the execution continues normally and p' behaves exactly as p. The transformation of p into p' adds new branches for error and error-free states so that PathCrawler algorithm will automatically try to cover error states. For an alarm a, PathCrawler may confirm it as a bug when it finds an input state and an error path leading to the bug. PathCrawler may also prove that the alarm is safe when all-path test generation on p' terminates without activating the corresponding threat. When all-path test generation on p' does not terminate, or when incomplete test coverage criterion was used (e.g. k-path), no alarm is classified safe. Finally, all alarms that are not classified as bug or safe remain unclassified or unknown.

4 SANTE tool architecture

This section describes tool integration solutions used in Sante. We present first the plugin-oriented architecture of Frama-C and the services provided by the platform to plugins. Then we describe how the analyzers presented in the previous section were integrated in Sante.

4.1 Frama-C's plugin architecture

Frama-C has an open plugin-oriented architecture structured around a kernel that provides services to all plugins [17]. Those services include:

– an abstract syntax tree (AST) for ISO C99 code, based on a modified version of the CIL framework [57];
– a logic layer on top of the C AST, that implements most of the ACSL specification language [4];
– a notion of *statuses* for logical assertions, with an inter-plugin consolidation. Different properties of a function can be proven by different plugins [18];
– a powerful notion of *projects*, so that plugins can perform their analyses on different views of the source code without involuntarily interfering [64];
– generic and unified mechanisms to parse (per-plugin) command-line options, or to emit messages;
– a *dynamic API* in which plugins register the functions they want to export. Those functions can be called by other plugins, in a fully type-safe way [65]. (Frama-C is written in OCaml, which is a strongly typed language [28].)

Frama-C's notion of projects is at the heart of the framework. Plugins that need to modify the AST never do so in-place: some other plugins may have computed results on the current program, that may become inconsistent after a modifcation. Instead, the kernel can duplicate the AST into a new project, which shares no mutable data-structures with the previous ones. This is typically the approach used by the SLICING plugin, which creates a new project (hence C program) per user slicing request. The resulting new program can then be analyzed as an isolated, separate entity.

Inside the platform, VALUE and some of the plugins based on it (PDG, SLICING) use highly-optimized datastructures. Typically, *hash-consing* [23] is employed to increase performance and reduce memory consumption. Those structures include generic lattices for products or sets, maps from bits to arbitrary values, etc. They can be reused as-is, for example to query the results of those plugins. Or, they can be instantiated to better suit the needs of the plugin developers.

The plugin API of FRAMA-C has already been used with great success to develop new plugins, either by industrial users [22], or academics [29, 43].

Community. FRAMA-C is characterized by a very active community behind it. First, the main FRAMA-C developers are available to answer questions on a public mailing list. Worldwide academic, industrial users and external plugin developers are also present on this list to ask and answer questions.

Besides, a bug tracking system, a wiki and a blog are available to interact with this community and to inform on what is going on within FRAMA-C. Those various media are available through the unique support webpage[5].

4.2 Tool integration in SANTE

The integration of the underlying tools used in SANTE was realized within the FRAMA-C platform and strongly relies on its functionalities. It is illustrated by Fig. 7 where rectangular forms indicate FRAMA-C plugins, while ovals show some of the basic techniques they use. An arrow indicates that a plugin calls another one.

We developed a new plugin, called SANTECONTROLLER, that calls different analyzers and implements the SANTE method described in Section 2. SANTECONTROLLER executes the following tasks:

1. It asks the VALUE plugin to perform the value analysis on the analyzed program. The results of this analysis are recorded with the AST and can be reused by other plugins.
2. It gathers the emitted alarms directly from the AST.
3. It constructs the subsets of alarms that will be used as slicing criteria. Dependency analysis is called if an advanced slicing option (taking into account alarm dependencies) was provided. Dependency analysis uses directly the results of value analysis without calling it again.

[5] http://frama-c.com/support.html

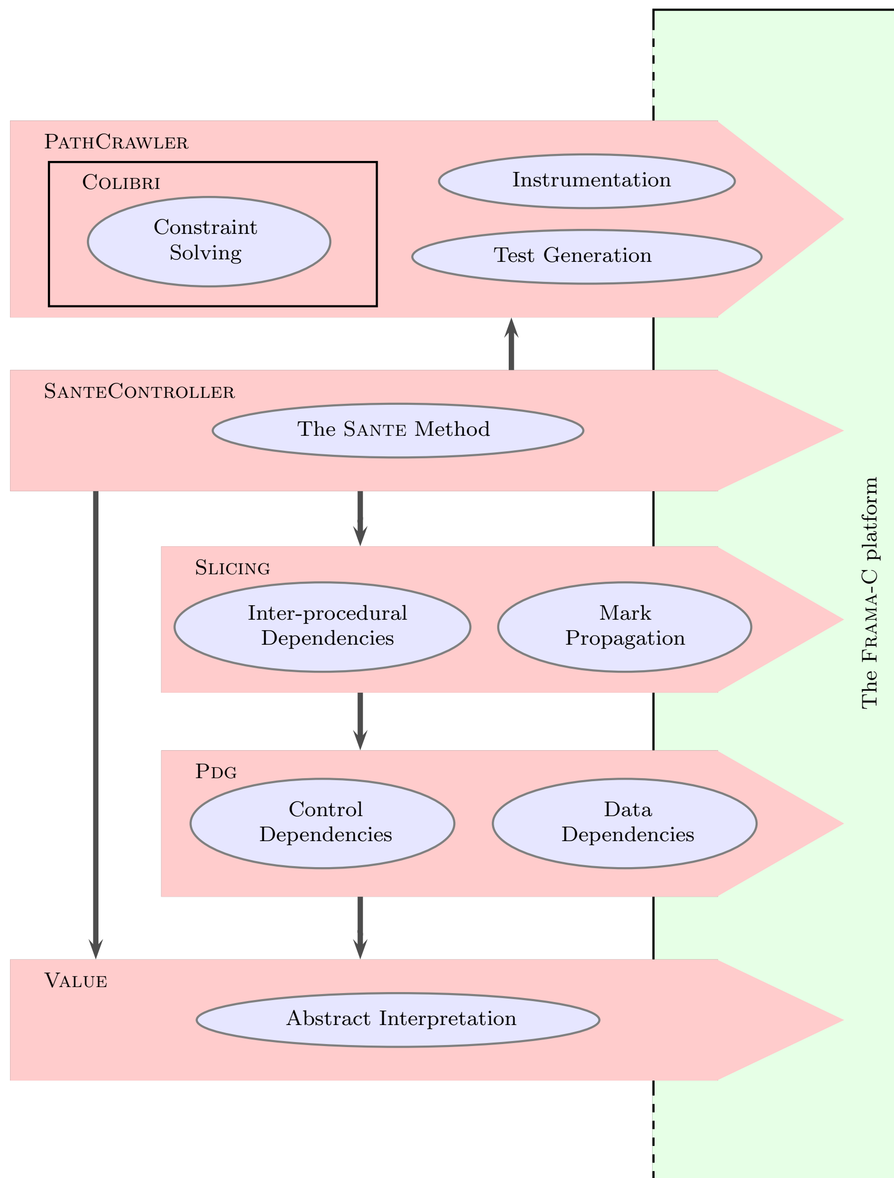

Fig. 7 The Sante tool architecture

4. The Slicing plugin is then asked to simplify the program with respect to each subset of alarms. The results of dependency analysis, if already computed, can be used without being computed again.

5. Each simplified program is then adapted (as explained in Section 3.5.4) and analyzed with PathCrawler. It can read the C program directly from the AST constructed by Slicing without parsing the program again.
6. The diagnostic is constructed.

SanteController is not distributed with the open source version of Frama-C (which includes Value, Pdg and Slicing) because it requires an installation of PathCrawler, proprietary tool of CEA LIST.

As mentioned above, Value, Pdg and Slicing plugins strongly rely on the Frama-C kernel. The first step of PathCrawler, called PC Analyzer, also uses the services provided by Frama-C. From an AST inside Frama-C, PC Analyzer instruments the C program (to prepare concrete execution that will trace the program path executed by a test case) and translates its instructions into constraints (to prepare symbolic execution). It uses the primitives provided by the kernel of Frama-C to create a new analysis project and to manage it. It also uses Frama-C primitives to copy the AST of the analyzed program in a newly created project and to manage ASTs. It traverses the AST to collect the necessary information for the intermediate files using the visitor `PathCrawlerVisitor` developed for this purpose, that inherits from the visitor provided by the kernel of Frama-C. PC Analyzer is also well situated to exploit other analyzers distributed with Frama-C. It allows PathCrawler to accept information provided by other plugins.

5 Evaluation

In this section, we evaluate the Sante tool from both theoretical and architectural points of view. We present first a summary of our experiments on several real-life programs and detail the results for a representative example. Then we report on the tool design and implementation issues.

5.1 Experiments on real-life programs

In this section, we provide experiments for Sante with and without slicing and compare them with one another, with value analysis alone (denoted *VA*) and with dynamic analysis alone (denoted *DA*). The *VA* method uses the value analysis plugin Value of Frama-C. The *DA* method uses PathCrawler alone, in alarm-guided mode for the exhaustive list of all potential threats (without filtering by value analysis and slicing) and considers each threat as an alarm. The Sante−*slicing* method uses Sante with the option that skips the program slicing step and goes directly to the dynamic analysis step after the value analysis step. The Sante+*slicing* method uses Sante with program slicing to simplify the program before the dynamic analysis step. The results are shown in Fig. 8.

We use nine examples extracted from real-life software where bugs were previously detected. The examples have been chosen randomly, taking into account error types and C language features already supported by Sante. The

Nº	*module* *function*	*threats*	*DA*			*VA*		Sante−*slicing*			Sante+*slicing*		
			✓	?	ϟ	✓	?	✓	?	ϟ	✓	?	ϟ
1	Apache escape_absol-ute_uri(simp.)	8	7	0	1	4	4	7	0	1	7	0	1
			22s			1s		14s			11s		
			86 nodes			-		39 nodes			37 nodes		
2	Apache escape_absol-ute_uri(full)	16	15	0	1	11	5	15	0	1	15	0	1
			20s			1s		10s			7s		
			63 nodes			-		52 nodes			32 nodes		
3	Spam Assassin message_write	17	0	17	0	2	15	15	0	2	15	0	2
			TO			1s		8min 43s			8min 43s		
			67* nodes			-		66 nodes			48 nodes		
4	Apache get_tag	12	0	9	3	0	12	0	9	3	4	5	3
			TO			1s		TO			54s + 1 TO		
			778* nodes			-		774* nodes			703* nodes		
5	QuickSort partition	8	7	0	1	4	4	7	0	1	7	0	1
			1min 56s			1s		1min 12s			1min 12s		
			67 nodes			-		50 nodes			50 nodes		
6	libgd gdImageString-FTEx	15	0	14	1	3	12	3	11	1	14	0	1
			TO			1s		TO			32m 16s		
			83* nodes			-		58* nodes			53 nodes		
7	polygon main	29	27	0	2	19	10	27	0	2	27	0	2
			5m 33s			<1s		1m 31s			7s		
			1092 nodes			-		560 nodes			106 nodes		
8	rawcaudio adpcm_decoder	10	0	10	0	8	2	8	2	0	9	1	0
			TO			<1s		TO			5s + 1 TO		
			1502* nodes			-		1252* nodes			569* nodes		
9	eurocheck main	19	18	0	1	14	5	18	0	1	18	0	1
			25s			<1s		18s			6s		
			129 nodes			-		77 nodes			58 nodes		

Fig. 8 Experimental results for dynamic analysis alone (DA), static value analysis alone (VA), Sante without slicing (Sante−*slicing*) and Sante with slicing (Sante+*slicing*). Columns '✓', '?' and 'ϟ' provide respectively the number of alarms proven safe, the number of remaining unclassified alarms and the number of detected bugs. "+ n TO" indicates that test generation was stopped by a timeout for n programs. The average length of program paths explored by dynamic analysis (when applicable) is given below the process duration.

size of these examples is between 33 and 97 lines of C code for Examples 1, 2, 3, 5, and between 154 and 705 lines for Examples 4, 6, 7, 8, 9. All bugs are out-of-bounds accesses or invalid pointers. Examples 1, 2, 3, 4 and 6 come from Verisec C analysis benchmark [48]. Example 5 comes from [1]. Example 7 is an open-source program[6] that computes the area of a convex polygon from the coordinates of its vertices. Example 8 comes from Mediabench [50]. Example 9 is an open-source program[7] containing a single function validating serial numbers on European bank notes. Experiments were conducted on an Intel Duo 1.66 GHz notebook with 1 GB of RAM. The timeout was set to ten minutes. For some examples, test generation could be stopped by the timeout several times on several different slices.

The columns of Fig. 8 show the example number and the results for each technique. The column *threats* gives the total number of potential threats before any analysis. The column '✓' provides the number of alarms proven safe

[6] http://c.happycodings.com/Mathematics/code4.html
[7] http://freshmeat.net/projects/eurocheck

by the method. The columns '?' and '↯' provide respectively the number of remaining unclassified alarms and the number of detected bugs. The full process duration and the number of timeouts (TO) are given below the numbers. The average length of program paths explored by dynamic analysis (when applicable) is given below the process duration. This length is expressed in terms of branching nodes (e.g. if, for, while) traversed through the path. For a test session interrupted by a timeout before exploring all program paths, we indicate the average length of paths explored so far and mark it with a '$*$'. In these experiments, all known bugs are detected with SANTE.

The column SANTE+*slicing* gives the results of SANTE with the advanced options (taking into account alarm dependencies) since they give the best results. The only exception is Example 3 where the indicated time corresponds to simplifying the program once with respect to all alarms, that was the most efficient.

In presence of timeouts, for a fair comparison between the methods with and without slicing, we increase the timeout of the analyses without slicing to the total time taken by all steps of SANTE+*slicing* (including timeouts), but it does not change the results. In particular, for Example 6, the timeout for *DA* and SANTE−*slicing* was set to 32m16s.

5.1.1 Detailed results for `gdImageStringFTEx`

In this section, we illustrate each step of the SANTE tool on the example 6 of function `gdImageStringFTEx`. It is part of a module in the GD open source library for dynamically creating images. This module contains 705 lines of C code and comes from Verisec C analysis benchmark [48]. The function `gdImageStringFTEx` contains 181 lines and takes a string `str` as input. We define the precondition for the function as:

`str` is a zero-terminated string.

SANTE starts by applying the value analysis step, which reports 12 alarms. Technically, the value analyzer marks each alarm by an annotation using the `assert` keyword (cf Section 3.1.1).

The second step automatically simplifies the program by program slicing. Without any simplification by program slicing, dynamic analysis applied to this program detects a bug and times out before classifying any other alarm. 11 alarms remain unclassified (unknown).

When program slicing is applied once with respect to the set containing the 12 alarms, it produces one slice p_{all}, in which the sliced function `gdImageStringFTEx` contains 151 lines of code instead of 181 initially. Then dynamic analysis applied to p_{all} detects the bug and times out before classifying any other alarm. 11 alarms remain unclassified.

When program slicing is performed 12 times, once with respect to each alarm, it produces 12 slices whose size vary from 131 to 140 lines of code. Then dynamic analysis is called 12 times to analyze the 12 resulting programs.

Analysis time of each slice varies from 7 minutes 25 seconds to 8 minutes 34 seconds. The complete time needed for the 12 slices is around 1 hour 32 minutes. Dynamic analysis detects the bug and classifies all the remaining alarms as safe. No alarm remains unclassified.

With advanced options taking into account alarm dependencies, program slicing is performed 4 times producing 4 slices whose size vary from 132 to 140 lines of code. All 12 alarms are preserved in these 4 slices. Then dynamic analysis is called 4 times. The complete time needed for the 4 slices is around 32 minutes 16 seconds. Dynamic analysis detects the bug and classifies all the remaining alarms as safe. No alarm remains unclassified.

5.1.2 Summary of the experimental results

SANTE without slicing vs. DA. Alarm-guided test generation in SANTE only treats the alarms raised by value analysis while *DA* alone has to exhaustively consider all potential threats. In Example 9, *DA* alone analyzes 19 alarms, and it takes 25 seconds to find a bug and to prove that the error states are unreachable for the remaining 18 threats, while *DA* in SANTE analyzes only 5 alarms because 14 threats have been already proven safe by value analysis. Our experiments show that test generation in SANTE can detect bugs faster and leave less unknown alarms (cf Examples 6, 8). Of course, when value analysis can't filter any threat (cf Example 4), SANTE can take as much time as *DA* alone.

SANTE without slicing vs. VA. *VA* used alone reduces the number of potential threats and proves that some of them are safe (cf Examples 1, 2, 3, 5, 6, 7, 8 and 9), but it generates a large number of alarms. In our experiments, SANTE leaves less unclassified alarms thanks to the dynamic analysis step. It confirms some alarms as real bugs and provides a counter-example illustrating each bug. It classifies some alarms as safe by proving that the error states for these alarms are unreachable.

SANTE with slicing vs. SANTE without slicing. After simplification by program slicing, SANTE runs dynamic analysis on a number of simplified programs. For the presented examples SANTE with slicing detects the bugs in less time (cf Examples 1, 2, 7, 9). It terminates in some cases where SANTE without slicing times out (cf Examples 4, 6, 8), and then it can classifies more alarms.

Program reduction. The number of paths can be over-exponential in the program size. Hence even a slight reduction of the program by the slicing step before test generation is beneficial and can give better results for larger programs. The average rate of program reduction in SANTE for our examples is about 32%, and it goes up to 89% for some alarms in some examples. The average length of program paths explored by the dynamic analysis step decreases after the use of slicing with an average of 19%. This rate can reach

82% (cf Example 7). On the worst case, where the program is not reduced by program slicing, the average length remains the same (cf Example 5).

Simpler counter-examples. Program slicing removes irrelevant code for the analyzed alarms. The counter-examples execute significantly shorter paths on the simplified programs. In our experiments, the execution path length in counter-examples diminishes on average by 12%, this rate going up to 71% on some programs. Simpler counter-examples reported for reduced programs may help the validation engineer more easily understand the reason of the error and fix it (cf [70]). This simplification could be particularly valuable for automatically generated code when the engineer does not have a deep knowledge of the resulting C source code.

The number of unclassified alarms with SANTE in our experiments becomes smaller than for *DA* (resp. for *VA*), decreasing on average by 88% (resp. 91%). For some examples this rate reaches 100% when all alarms are classified.

Speedup. SANTE is less time-consuming than *DA*, and it may allow to avoid timeouts during test generation. The average speedup rate in our experiments is around 43%, going up to 98% on some examples.

A more detailed evaluation of various options of SANTE is available in [14].

5.2 Tool design

To provide a successful tool combination, each one of the combined tools has to achieve a high maturity level and benefit from a regular support of its developers. Developing so different and mature tools as those described in Section 3 is clearly not within the reach of a single person. Indeed, their development requires quite different theoretical background and programming skills, so a certain independence in the development of these tools seems necessary and logic.

On the other hand, combining quite independent tools within a new one poses well-known technical difficulties of tool integration, for example:

- How to efficiently communicate the results from one tool to another?
- How to avoid duplicated actions (typically, parsing or alias analysis) with their inherent waste of time?
- How to provide a global view of current and final results?

Our experience shows that an open plugin-oriented architecture for software analysis tools like that of the FRAMA-C platform is an excellent solution for the dilemma between efficient parallel development of each tool and the ease of their integration. FRAMA-C is extensible by nature, thanks to its plugin architecture. Each plugin can exploit the results of analysis made by other plugins and the services provided by the kernel of FRAMA-C. These services

(cf Section 4.1) easily and naturally solve all aforementioned communication, optimization and result visibility issues. For instance, in Sante (cf Fig. 7) instead of writing and re-reading a C program between value analysis and slicing steps, or between slicing and test generation steps, the program can be parsed only once and its AST remains available for all Frama-C plugins. Similarly, being computed only once, value analysis results may be used by dependency analysis and program slicing.

Indeed, analysis results of a plugin A are saved in the abstract syntax tree of the analyzed program. These results can be seen through the graphical user interface (or in batch mode) and, most importantly, they can be accessed by another plugin B through the API registered by A in the Frama-C kernel. Moreover, the opportunity to work on several abstract syntax trees (Frama-C projects) in parallel allows an optimized implementation of slicing. Program slices can be constructed directly in the AST form by Slicing and exploited in this form by PathCrawler.

Furthermore, in a plugin-oriented architecture, one plugin can be modified or upgraded without having to update other plugins or causing the tool to stop functioning. As long as the defined interfaces are respected, the plugin developers can independently work in parallel and no communication between them is needed.

The architecture and collaborative approach of Frama-C make it possible to create powerful combined verification tools with relatively little effort. While the cumulative size of the tools combined by Sante is several hundreds of thousands of lines of code, the size of the SanteController plugin is only around 1500 lines in OCaml. It exploits the services provided by the kernel of Frama-C and the existing analyzers (Value, Slicing, PathCrawler). Several of the Frama-C services used by Sante (front-end, Value, Pdg, Slicing) have been intensively tested on randomly generated programs [27], increasing the chances they behave as intended and work well even on corner cases.

The development of the SanteController plugin has taken only a few months and was mainly realized by only one person.

6 Related Work

Recently, several papers presented combinations of static and dynamic analyses for program verification. Daikon [30] uses dynamic analysis to detect likely invariants. Check'n'Crash [66] applies static analysis that reports alarms but uses intra-procedural weakest-precondition computation rather than value analysis, so it necessitates code annotations and can have a high rate of false alarms. Next, random test generation tries to confirm the bugs. Sante uses an inter-procedural value analysis that necessitates only a precondition, and all-path test generation, that may in addition show that some alarms are unreachable.

DSD Crasher [66] applies Daikon [30] to infer likely invariants before the static analysis step of Check'n'Crash to reduce the rate of false alarms. This method admits generated invariants that may be wrong and can result in proving some real bugs as safe, unlike SANTE which never reports a bug as safe.

In the implementation of Check'n'Crash and DSD Crasher, authors use some open-source tools. They use ESC/Java [16] (developed by Compaq Systems Research Center) for static analysis, JCrasher [20] (developped by Georgia Institute of Technology) for dynamic analysis and Daikon [30] (developped by MIT Computer Science and Artificial Intelligence Lab) to detect likely invariants. They adapt each tool in order to communicate with the others. Then enhancements of some components are needed like providing JML annotations to Daikon components and others described by the authors in [21]. In SANTE, easy tool integration is an important advantage of FRAMA-C plugin architecture, where analyses can communicate via well-defined interfaces, and we don't need to modify SANTE if a component is modified or upgraded unless the interface is modified.

Synergy [39], BLAST [7] and [60] combine testing and partition refinement for property checking. SANTE is relative to the Yogi tool that implements the algorithm DASH [5], initially called Synergy. In SANTE, we use value analysis whereas Yogi uses weakest precondition with template-based refinement. The objective of both tools is to detect error states. They are specified as an input property in Yogi whereas in SANTE they are automatically computed by value analysis and error-branch introduction. Yogi does not use program slicing. It iteratively refines an over-approximation using information on unsatisfiable constraints from test generation. Its approach is more adapted for one error statement at a time, while SANTE can be used on several alarms simultaneously. [60] combines predicate abstraction and test generation in a refinement process, guided by the exactness of the abstraction with respect to operations of the system rather than by test generation. In the same spirit, [75] uses test generation to compute an abstraction from concrete states. If the abstraction contains abstract counter-examples, new concrete states are fabricated along the abstract counter-examples as a heuristic to increase the coverage of testing. The method is also able to detect when the current program abstraction provides a proof. DyTa [32] is another implementation of the main ideas of SANTE [12]. However in DyTa some irrelevant code is excluded before dynamic analysis only for CFG connectivity reasons, that is weaker than program slicing.

Little detail about the implementation of these tools is available. Yogi uses SLAM and DART for static and dynamic analysis respectively, both developed by Microsoft. BLAST is developed at UC Berkeley, and [75] is implemented on the top of the XRT framework [38].

To the best of our knowledge, SANTE is the first verificaion tool combining abstract interpretation based static analysis, dependence-based program slicing and all-path test generation.

7 Discussion and future work

Originated from different communities, various static and dynamic analysis techniques evolved along parallel but separate tracks. Traditionally, they were viewed as separate domains. However, static and dynamic analysis have complementary strengths and weaknesses, and combining them can provide new efficient methods for software verification.

In this paper we presented the SANTE tool combining several static and dynamic analysis tools. We emphasized the particular features of these individual tools exploited by SANTE, as well as tool integration solutions adopted in the tool and more generally in the FRAMA-C platform. FRAMA-C's plugin architecture favors both intensive development of several tools in parallel and their integration for creating combined analysis tools.

In the context of FRAMA-C in general, and SANTE in particular, the VALUE plugin has two uses. The first one is to detect potentially unsafe instructions, and flag them as such. The second one consists in providing comprehensive aliasing information, which can then be exploited by all other plugins (notably PDG and SLICING). Crucially, improving the precision of the value analysis in general – through better abstract domains, or exploration strategies – can potentially lead to better results for both functionalities.

In FRAMA-C, the PDG plugin is separate from the SLICING one. This was a conscious design choice, as an independent PDG computation tool is useful on its own right. For example, another plugin (SCOPE) uses the results of PDG to remove alarms emitted by the value analysis at two different points, when one is logically equivalent to another. This automatically lowers the number of alarms that must be verified by SANTE. In the future, more aggressive forms of removal could be envisioned. Typically, we could define a notion of implication between assertions, and remove those *implied* by another.

Regarding the SLICING plugin, the possibility to require multiple slicing criteria is very important in the context of SANTE, when selecting many alarms simultaneously. More subtle but equally useful is the removal of useless function arguments by specialization. It can result in more important code reduction than is typically possible with standard slicing tools. This in turn means that PATHCRAWLER needs to explore less code. An even tighter integration could be envisioned: supplying to PATHCRAWLER which instructions are spare (cf §3.3.3) would be interesting. Indeed, full coverage of those instructions is not useful, as by definition they do not participate to the slicing criterion.

The dynamic analysis step of SANTE advantageously uses the PATHCRAWLER tool whose completeness is essential for SANTE. It allows alarms remaining non-confirmed after a complete all-path exploration of a program (slice) to be classified as false alarms. On the other hand, for programs having a too big number of paths even after program slicing, partial k-path coverage remains available to confirm alarms. PATHCRAWLER's efficiency relies on its fast concolic test generation method, ingeniously combining symbolic and concrete execution, with incremental path predicate construction and early infeasibility detection. Using the constraint logic programming paradigm plays a

crucial role in PathCrawler's current implementation. It also benefits from the Colibri constraint solving library providing a variety of types and constraints, primitives for labeling procedures, support for floating point numbers and efficient constraint resolution. Future work on PathCrawler includes optimisation for partial coverage criteria such as all-branches, and the treatment of the whole of ANSI-C.

In our future work on Sante, we would like to extend the tool to other classes of threats such as reading an uninitialized variable, arithmetic overflow and untreated cases of invalid pointers. We expect that the current tool architecture will be perfectly appropriate for these extensions. Sante offers a promising opportunity to run test generation sessions for different slices independently, and perspectives of parallelization for Sante should be further explored.

Future work also includes studying further combinations of analysis techniques, such as:

– other configurations of value analysis in order to find a reasonable compromise between the analysis precision and time,
– other usages of program slicing,
– other test generation techniques like using a generational search rather than the depth-first search, and other types of coverage like the all-branch test generation.

Another perspective is to combine a proof technique with dynamic analysis within Frama-C, where we intend to use the Wp or Jessie plugin for proof of programs.

Finally, the results of Sante could be integrated into the Frama-C platform where they can be exploited by the kernel or any other plugin. Typically, SanteController would flag assertions that PathCrawler has proven correct as valid in the original code. More ambitious would be to mark as *dead code* the code shown to be unreachable by PathCrawler in the sliced code – but only if it is also unreachable in the original one. The facilities for collaboration, communication and result visibility between different analyzers within the Frama-C platform promise to be very helpful for realization of these perspectives.

Acknowledgements The authors thank Patrick Baudin, Bernard Botella, Loïc Correnson, Benjamin Monate, Virgile Prevosto and Julien Signoles for their support and advice, as well as the editors and anonymous referees for profound analysis of the paper and lots of valuable comments. Special thanks to Alain Giorgetti and Jacques Julliand for their contribution on the theoretical aspects of the Sante method.

References

1. Ball, T.: A theory of predicate-complete test coverage and generation. In: the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004), *LNCS*, vol. 3657, pp. 1–22. Springer (2004)

2. Bardin, S., Herrmann, P.: Structural testing of executables. In: the First International Conference on Software Testing, Verification, and Validation (ICST 2008), pp. 22–31. IEEE Computer Society (2008)
3. Bardin, S., Herrmann, P., Perroud, F.: An alternative to SAT-based approaches for bit-vectors. In: the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010), *LNCS*, vol. 6015, pp. 84–98. Springer (2010)
4. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, v1.6 (2012). URL: http://frama-c.com/acsl.html
5. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), pp. 3–14. ACM (2008)
6. Berthomé, P., Heydemann, K., Kauffmann-Tourkestansky, X., Lalande, J.F.: Attack model for verification of interval security properties for smart card c codes. In: the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2010). ACM (2010)
7. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast: Applications to software engineering. Int. J. Softw. Tools Technol. Transfer **9**(5-6), 505–525 (2007)
8. Bonichon, R., Cuoq, P.: A mergeable interval map. Studia Informatica Universalis, JFLA 2010 **9**(1), 5–37 (2011)
9. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), pp. 209–224. USENIX Association (2008)
10. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: the 13th ACM Conference on Computer and Communications Security (CCS 2006), pp. 322–335. ACM Press (2006)
11. Canet, G., Cuoq, P., Monate, B.: A value analysis for C programs. In: the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009), pp. 123–124. IEEE Computer Society (2009)
12. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Combining static analysis and test generation for C program debugging. In: the 4th International Conference on Tests and Proofs (TAP 2010), LNCS, pp. 652–666. Springer (2010)
13. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: The SANTE tool: Value analysis, program slicing and test generation for C program debugging. In: the 5th International Conference on Tests and Proofs (TAP 2011), LNCS, pp. 78–83. Springer (2011)
14. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: the ACM Symposium on Applied Computing (SAC 2012), pp. 1284–1291. ACM (2012)
15. Choi, J.D., Ferrante, J.: Static slicing in the presence of goto statements. ACM Trans. Program. Lang. Syst. **16**(4), 1097–1113 (1994)
16. Cok, D.R., Kiniry, J.R.: ESC/Java2: uniting ESC/Java and JML. In: the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2004), *LNCS*, vol. 3362, pp. 108–128. Springer (2004)
17. Correnson, L., Cuoq, P., Kirchner, F., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C User Manual (2012). URL: http://frama-c.com
18. Correnson, L., Signoles, J.: Combining Analyses for C Program Verification. In: the 17th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2012), *LNCS*, vol. 7437, pp. 108–130. Springer (2012)
19. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: the 4th Symposium on Principles of Programming Languages (POPL 1977), pp. 238–252 (1977)
20. Csallner, C., Smaragdakis, Y.: JCrasher: An automatic robustness tester for Java. Software — Practice & Experience **34**(11), 1025–1050 (2004)
21. Csallner, C., Smaragdakis, Y.: Dynamically discovering likely interface invariants. In: the 28th ACM/IEEE International Conference on Software Engineering (ICSE 2006), Emerging Results Track, pp. 861–864. ACM (2006)

22. Cuoq, P., Delmas, D., Duprat, S., Moya Lamiel, V.: Fan-C, a Frama-C plug-in for data flow verification. In: the Embedded Real-Time Software and Systems Congress (ERTS^2 2012) (2012)
23. Cuoq, P., Doligez, D.: Hashconsing in an incrementally garbage-collected system: a story of weak pointers and hashconsing in OCaml 3.10.2. In: the ACM Workshop on ML (ML 2008), pp. 13–22. ACM (2008)
24. Cuoq, P., Hilsenkopf, P., Kirchner, F., Labbé, S., Thuy, N., Yakobowski, B.: Formal verification of software important to safety using the Frama-C tool suite. In: the 8th International Conference on Nuclear Plant Instrumentation and Control (NPIC 2012) (2012)
25. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C, a program analysis perspective. In: the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012), *LNCS*, vol. 7504, pp. 233–247. Springer (2012)
26. Cuoq, P., Monate, B., Pacalet, A., Prevosto, V.: Functional dependencies of C functions via weakest pre-conditions. Int. J. Softw. Tools Technol. Transfer **13**(5), 405–417 (2011)
27. Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B., Yang, X.: Testing static analyzers with randomly generated programs. In: the 4th International NASA Formal Methods Symposium (NFM 2012), *LNCS*, vol. 7226, pp. 120–125. Springer (2012)
28. Cuoq, P., Signoles, J., Baudin, P., Bonichon, R., Canet, G., Correnson, L., Monate, B., Prevosto, V., Puccetti, A.: Experience report: OCaml for an industrial-strength static analysis framework. In: the 14th ACM SIGPLAN international conference on Functional programming (ICFP 2009), pp. 281–286. ACM (2009)
29. Dragoi, C., Sighireanu, M.: CELIA User manual (2011). `http://www.liafa.jussieu.fr/celia/`
30. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
31. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987)
32. Ge, X., Taneja, K., Xie, T., Tillmann, N.: DyTa: dynamic symbolic execution guided with static verification results. In: the 33rd International Conference on Software Engineering (ICSE 2011), pp. 992–994. ACM (2011)
33. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: the Network and Distributed System Security Symposium (NDSS 2008). The Internet Society (2008)
34. Gotlieb, A.: Euclide: A constraint-based testing platform for critical C programs. In: the Second International Conference on Software Testing Verification and Validation (ICST 2009), pp. 151–160. IEEE Computer Society (2009)
35. Gotlieb, A., Botella, B., Watel, M.: INKA : Ten years after the first ideas. In: the International Conference on Software and Systems Engineering and their Applications (ICSSEA 2006) (2006)
36. Gotlieb, A., Leconte, M., Marre, B.: Constraint solving on modular integers. In: The CP 2010 Workshop on Constraint Modelling and Reformulation (ModRef 2010) (2010)
37. Granger, P.: Static analysis of linear congruence equalities among variables of a program. In: the International Joint Conference on Theory and Practice of Software Development (TAPSOFT 1991), Volume 1: Colloquium on Trees in Algebra and Programming (CAAP 1991), LNCS, pp. 169–192. Springer (1991)
38. Grieskamp, W., Tillmann, N., Schulte, W.: Xrt– exploring runtime for .net architecture and applications. Electron. Notes Theor. Comput. Sci. **144**(3), 3–26 (2006)
39. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2006), pp. 117–127. ACM (2006)
40. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1988), vol. 23-7, pp. 35–46 (1988)

41. IEEE Std 754-2008: IEEE standard for floating-point arithmetic. Tech. rep. (2008). http://dx.doi.org/10.1109/IEEESTD.2008.4610935
42. International Organization for Standardization: ISO/IEC 9899:TC3: Programming Languages — C (2007). http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf
43. Iosif, R., Garnier, F.: Flata-C. http://www-verimag.imag.fr/FLATA-C.html
44. Kosmatov, N.: All-paths test generation for programs with internal aliases. In: the 19th International Symposium on Software Reliability Engineering (ISSRE 2008), pp. 147–156. IEEE Computer Society (2008)
45. Kosmatov, N.: On complexity of all-paths test generation. From practice to theory. In: Proceedings of the Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART 2009), pp. 144–153. IEEE Computer Society Press (2009)
46. Kosmatov, N.: Online version of PathCrawler. (2010–2012). http://pathcrawler-online.com/
47. Kosmatov, N., Legeard, B., Peureux, F., Utting, M.: Boundary coverage criteria for test generation from formal models. In: the 15th International Symposium on Software Reliability Engineering (ISSRE 2004), pp. 139–150. IEEE Computer Society (2004)
48. Ku, K., Hart, T.E., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), pp. 389–392. ACM (2007)
49. Leconte, M., Berstel, B.: Extending a cp solver with congruences as domains for software verification. In: the CP 2006 Workshop on Constraints in Software Testing, Verification and Analysis, CSTVA 2006 (2006)
50. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In: the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 1997), pp. 330–335. IEEE Computer Society (1997)
51. Mark Harman, D.B., Danicic, S.: Amorphous program slicing. Journal of Systems and Software **68**(1), 45–64 (2003)
52. Marre, B., Arnould, A.: Test sequences generation from Lustre descriptions: GATeL. In: the 15th IEEE International Conference on Automated Software Engineering (ASE 2000), pp. 229–237. IEEE Computer Society (2000)
53. Marre, B., Blanc, B.: Test selection strategies for Lustre descriptions in GATeL. Electronic Notes in Theoretical Computer Science **111**, 93–111 (2005)
54. Marre, B., Michel, C.: Improving the floating point addition and subtraction constraints. In: the 16th International Conference on Principles and Practice of Constraint Programming (CP 2010), *LNCS*, vol. 6308, pp. 360–367. Springer (2010)
55. Michel, C.: Exact projection functions for floating point number constraints. In: the 7th International Symposium on Artificial Intelligence and Mathematics (AIMA 2002) (2002)
56. Mouy, P., Marre, B., Willams, N., Le Gall, P.: Generation of all-paths unit test with function calls. In: the First International Conference on Software Testing, Verification, and Validation (ICST 2008), pp. 32–41. IEEE Computer Society (2008)
57. Necula, G.C., Mcpeak, S., Rahul, S.P., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of c programs. In: the International Conference on Compiler Construction (CC 2002), *LNCS*, vol. 2304, pp. 213–228. Springer (2002)
58. Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. In: the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1984), pp. 177–184. ACM Press (1984)
59. Pariente, D., Ledinot, E.: Formal verification of industrial c code using Frama-C: a case study. In: the International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010), pp. 205–218 (2010)
60. Pasareanu, C., Pelanek, R., Visser, W.: Concrete model checking with abstract matching and refinement. In: the 17th International Conference on Computer Aided Verification (CAV 2005), *LNCS*, vol. 3576, pp. 52–66. Springer (2005)
61. Reps, T., Turnidge, T.: Program specialization via program slicing. In: the Dagstuhl Seminar on Partial Evaluation, *LNCS*, vol. 1110, pp. 409–429. Springer (1996)
62. Schimpf, J., Shen, K.: ECLiPSe - from LP to CLP. Theory and Practice of Logic Programming **12**(1-2), 127–156 (2011)

63. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005), pp. 263–272. ACM (2005)
64. Signoles, J.: Foncteurs impératifs et composés: la notion de projets dans Frama-C. Studia Informatica Universalis, JFLA 2009 **7**(2), 20–51 (2009)
65. Signoles, J.: Une bibliothèque de typage dynamique en OCaml. In: Journées Francophones des Langages Applicatifs (JFLA 2011), pp. 209–242. Hermann, Studia Informatica Universalis (2011)
66. Smaragdakis, Y., Csallner, C.: Combining static and dynamic reasoning for bug detection. In: the First International Conference on Tests and Proofs (TAP 2007), *LNCS*, vol. 4454, pp. 1–16. Springer (2007)
67. Tillmann, N., de Halleux, J.: White box test generation for .NET. In: the Second International Conference on Tests and Proofs (TAP 2008), *LNCS*, vol. 4966, pp. 133–153. Springer (2008)
68. Tip, F.: A survey of program slicing techniques. J. Prog. Lang. **3**(3) (1995)
69. Weiser, M.: Program slicing. In: the 5th International Conference on Software Engineering (ICSE 1981), pp. 439–449. IEEE Computer Society (1981)
70. Weiser, M.: Programmers use slices when debugging. Commun. ACM **25**(7), 446–452 (1982)
71. Williams, N.: WCET measurement using modified path testing. In: the 5th International Workshop on Worst-Case Execution Time Analysis (WCET 2005) (2005)
72. Williams, N., Marre, B., Mouy, P.: On-the-fly generation of k-paths tests for C functions : towards the automation of grey-box testing. In: the 19th IEEE International Conference on Automated Software Engineering (ASE 2004), pp. 290–293. IEEE Computer Society (2004)
73. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: the 5th European Dependable Computing Conference on Dependable Computing (EDCC 2005), *LNCS*, vol. 3463, pp. 281–292. Springer (2005)
74. Williams, N., Roger, M.: Test generation strategies to measure worst-case execution time. In: the 4th International Workshop on Automation of Software Test (AST 2009), pp. 88–96. IEEE Computer Society (2009)
75. Yorsh, G., Ball, T., Sagiv, M.: Testing, abstraction, theorem proving: better together! In: the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), pp. 145–156. ACM (2006)