Under consideration for publication in Formal Aspects of Computing

Frama-C
A Software Analysis Perspective

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto,
Julien Signoles and Boris Yakobowski [†]

CEA, LIST, Software Reliability Laboratory

Abstract. Frama-C is a source code analysis platform that aims at conducting verification of industrial-size C programs. It provides its users with a collection of plug-ins that perform static analysis, deductive verification, and testing, for safety- and security-critical software. Collaborative verification across cooperating plug-ins is enabled by their integration on top of a shared kernel and datastructures, and their compliance to a common specification language. This foundational article presents a consolidated view of the platform, its main and composite analyses, and some of its industrial achievements.

1. Introduction

The past forty years have seen much of the groundwork of formal software analysis being laid. Several angles and theoretical avenues have been explored, from deductive reasoning to abstract interpretation to program transformation to concolic testing. While much remains to be done from an academic standpoint, some of the major advances in these fields are already being successfully implemented [CCF$^+$05, Mat, DMS$^+$09, RSB$^+$99, WMMR05] – and met with growing industrial interest. The ensuing push for mainstream dissemination of software analysis techniques has raised several challenges.

Scaling is predictably important from the point of view of adoptability. Handling large problems is a prerequisite for the industrial diffusion of software analysis and verification techniques. Scaling also requires better understanding of how language idioms (e.g. pointers, unions, or dynamic memory allocation) influence the underlying architecture of large software developments. Overall, achieving scalability in the design of software analyzers for a wide range of software patterns remains a difficult question.

Interoperability enables the design of elaborate program analyses. Recent work on the interplay between program analyses and transformations [DJP10], the complementarity of forward and backward analyses [BHV11], or the precision gain afforded when combining static and dynamic approaches [BNR$^+$10] has demonstrated the value of interconnected approaches. Yet running multiple source code analyses and synthesizing their results in a coherent fashion requires carefully thought-out mechanisms.

[†]and Patrick Baudin, François Bobot, Richard Bonichon, Bernard Botella, Omar Chebaro, Loïc Correnson, Pascal Cuoq, Zaynah Dargaye, Philippe Herrmann, Matthieu Lemerre, Claude Marché, Benjamin Monate, Yannick Moy, Anne Pacalet, Armand Puccetti, Muriel Roger, Nicolas Stouls and Nicky Williams.

This work was partly supported by ANR projects U3CAT and Veridyc, FUI9 project Hi-Lite, and FP7 project STANCE.
An earlier version of this work was presented at the conference SEFM 2012.

Correspondence and offprint requests to: CEA Saclay Nano-Innov, Institut Carnot CEA LIST, DILS/LSL, Point courrier 174, F-91191 Gif-sur-Yvette Cedex, France. email: firstname.lastname@cea.fr

Plug-in	Short description	Section
Value	abstract interpretation based value analysis	4
WP	deductive verification	5
PathCrawler	concolic test generation	6
E-ACSL	runtime assertion checking	7
Aoraï	temporal specification	8
Mthread	analysis of concurrent code	9
Inout	function inputs and outputs	10.1
From	functional dependencies	10.1
PDG	program dependency graph	10.1
Scope	data dependencies	10.1
Slicing	dependency-based program slicing	10.1
Sante	combined static/dynamic analysis	10.2

Fig. 1. Main Frama-C analyzers

Soundness is a strong differentiator for formal approaches. By using tools that *over-approximate* all program behaviors, users are assured that none of the errors they are looking for remain undetected. This guarantee stands in stark contrast with the bug-finding capabilities of heuristic analyzers, and is paramount in the evaluation of critical software. But the design and implementation costs of such high-integrity solutions are hard to expend.

The Frama-C software analysis platform provides a collection of scalable, interoperable, and sound software analyses for the industrial analysis of ISO C99 source code. The platform is based on a common *kernel*, which hosts analyzers as collaborating *plug-ins* and uses the ACSL formal specification language as a *lingua franca*. Frama-C includes plug-ins based on abstract interpretation, deductive verification, and dynamic analysis; and a series of derived plug-ins which build elaborate analyses upon the former. In addition, the extensibility of the overall platform, and its open-source licensing, have fostered the development of an ecosystem of independent third-party plug-ins. The Frama-C plug-ins presented in this paper are listed in Fig. 1.

This article is intended as a foundational reference to the kernel (Section 3) and Frama-C's main analyses. Sections 4, 5, and 9 present static program verification plug-ins, based on abstract interpretation and deductive verification. Plug-ins using dynamic analysis are described in Sections 6 and 7. Frama-C's temporal specifications verification plug-in is depicted in Section 8. Finally, Section 10 outlines various plug-ins that combine the results of previously introduced analyzers.

2. Related Work

Combining analysis techniques (in particular static and dynamic ones) is a quite recent but not new idea [EMN12]. However, there are very few frameworks that aim at being easily extended and at tightly integrating various program analysis techniques around a common logic language such as ACSL. Notably, the Clang Static Analyzer [Cla] is built on top of the Clang/LLVM architecture to check various issues in C, C++ or Objective-C code. As with many LLVM-related projects, it is meant to be extensible so that users can add specific checks. However, its scope is much more limited than Frama-C. Indeed, its main aim is to issue meaningful warnings during compilation with Clang, not to provide an exhaustive verification of the absence of runtime error or of the functional properties of the program.

Considering other programming languages, Spark2014 [CKM12, DEL$^+$14, KCC$^+$14] is a subset of Ada which comes with a specification language and allows the user to mix deductive verification and testing: some functions may be tested while some others may be proved. However this framework does not provide any extensible mechanism to add new custom analyzers. Why and its successor Why3 [Fil03, FP13] propose another approach by offering a language dedicated to deductive verification using Hoare logic. This can then be used as an intermediate language for program verification by translating original code into Why, sharing the same back-end for several programming languages. This is in particular used by Spark2014, the Jessie plug-in [MM12] of Frama-C and the Krakatoa [FM07] platform to prove Ada, C and Java programs respectively. Why and Why3 specifically target deductive verification though, and are not aimed at providing support for other analysis techniques. This is also the case for the Boogie intermediate language [BCD+06] that is used as back-end for verifying programs written in Spec#, C, Dafny or Chalice.

When it comes to verification, Rushby's evidential tool bus [Rus05, CHOS13] offers an interesting approach to let different tools collaborate for a given verification task. It is based on a client/server architecture, where each server indicates which tasks it can handle, while clients make verification requests that have to be decomposed in elementary tasks using various strategies. Communication is based on Datalog predicates. The evidential tool bus is a very generic framework and can thus accommodate a very broad set of tools. On the other hand, the bus in itself does not know anything about the C language. Hence, it cannot provide directly the same level of coupling that is offered by Frama-C (although this could be achieved by defining an appropriate library of predicates). On the other hand, Frama-C could very well be integrated in an evidential tool bus that would be used to verify larger systems composed of other parts than simply C code.

Very few tools tackle the goal of formally verifying a program by combining different techniques in a consistent way. Heintze *at al.* [HJV00] proposes a framework by equational reasoning to combine an abstract interpreter with a deductive verification tool to enhance verification of user assertions. As Frama-C, it does not depend on specific analyzers and is correct modulo analyzer's correctness. However, instead of focusing on merging analyzer's results, it implements a new analyzer which operates on the results of the analyzers which it is based on. More recently, the Eve verification environment for Eiffel programs combines a deductive verification tool and a testing tool in order to make software verification practical and usable [TFNM11]. Eve reports the separated results obtained from this tool. Since tools which Eve is based upon are not supposed to be correct, Eve computes a so-called *correctness score* for each property. This score is a metrics indicating a level of confidence in its correctness. That is quite different from our approach where we suppose that analyzers are correct but can use other properties as hypotheses to build a proof.

In many cases, combined verification tools were developed starting from separate analyzers that were not specifically designed for collaboration with each other. Daikon [EPG+07] uses dynamic analysis to detect likely invariants. Check'n'Crash [SC07] applies first static analysis that reports alarms using intra-procedural weakest-precondition computation, then random test generation tries to confirm the bugs. DSD Crasher [SC07] applies Daikon [EPG+07] to infer likely invariants by dynamic analysis before the static analysis step of Check'n'Crash to reduce the rate of false alarms.

In the implementation of Check'n'Crash and DSD Crasher, the authors use independent open-source tools: ESC/Java [CK04] (developed by Compaq Systems Research Center) for static analysis, JCrasher [CS04] (developed by Georgia Institute of Technology) for dynamic analysis and Daikon [EPG+07] (developed by MIT Computer Science and Artificial Intelligence Lab) to detect likely invariants. They adapt each tool in order to communicate with the others. Moreover, enhancements of some components are necessary (like providing JML annotations to Daikon components) as described by the authors in [CS06].

Synergy [GHK+06] and BLAST [BHJM07] combine testing and partition refinement for property checking. The algorithm DASH [BNR+10], initially called Synergy, is implemented in the Yogi tool [GdHN+08]. Yogi uses independent tools SLAM and DART for static and dynamic analysis respectively, both developed by Microsoft.

Easy tool integration is an important advantage of the Frama-C plugin architecture, where analyses can communicate via well-defined interfaces. Developers do not even need to modify the combined tool if one of the components is modified or upgraded (unless the interface is modified).

Other individual software analysis tools related to separate Frama-C analyzers are also cited in the following sections.

3. The Platform Kernel

3.1. Architecture

The Frama-C platform is written in OCaml [LDF+13], a functional language whose features are very interesting for implementing program analyzers [CSB+09]. Fig. 2 shows a functional view of the Frama-C plug-in-oriented architecture (*à la* Eclipse) whose kernel is based on a modified version of CIL [NMRW02]. CIL is a front-end for C that parses ISO C99 programs into a normalized representation. For instance, loop constructs (`for`, `while`, `do ... while`) are given a single normalized form, normalized expressions have no side-effects, *etc.* Frama-C extends CIL to support dedicated source code annotations expressed in ACSL (see Section 3.2). This modified CIL front-end produces the C + ACSL abstract syntax tree (AST), an ab-

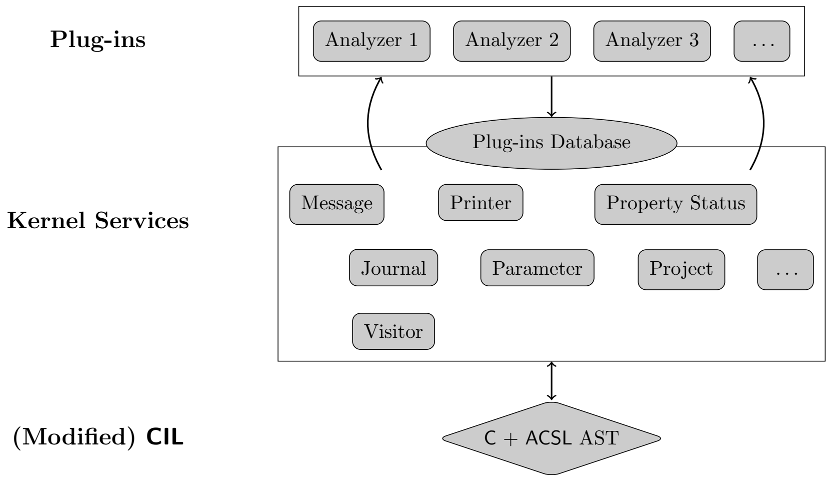


Fig. 2. Frama-C's functional view

stract view of the program shared among all analyzers. This AST takes into account machine-dependent parameters (size of integral types, endianness, *etc.*) which can easily be modified by the end-user.

In addition to the AST, the kernel provides several general services for helping plug-in development [SCP13] and providing convenient features to Frama-C's end-user.

- Messages, source code and annotations are uniformly displayed.
- Parameters and command line options are homogeneously handled.
- A journal of user actions can be synthesized and replayed afterwards, a useful feature in debugging and qualification contexts [Sig14].
- A safe serialization mechanism [CDS11] allows the user to save results of analyses and reload them later.
- Projects, presented in Section 3.3, isolate unrelated program representations, and guarantee the integrity of analyses.
- Consistency mechanisms control the collaboration between analyzers (Section 3.4).
- A visitor mechanism, partly inherited from CIL, facilitates crawling through the AST and writing code transformation plug-ins.

Analyzers are developed as separate plug-ins on top of the kernel. Plug-ins are dynamically linked against the kernel to offer new analyses, or to modify existing ones. Any plug-in can register new services in a *plug-in database* stored in the kernel, thereby making these services available to all plug-ins.

3.2. ACSL

Functional properties of C programs can be expressed within Frama-C as ACSL annotations [BFH+13]. ACSL, the ANSI/ISO C Specification Language, is a formal specification language inspired by Java's JML [BCC+05], both being based on the notion of function contract introduced by Eiffel [Mey97]. In its most basic form, the specification (or contract) of a function states the pre-conditions it **`requires`** from its caller and the post-conditions it **`ensures`** when returning. Among these post-conditions, one kind of clause plays a particular role by saying which memory locations the function **`assigns`**, i.e. which locations *might* have a different value between the pre- and the post-state. Conversely, any memory location not mentioned in this clause is supposed to be left unmodified by the function.

Annotations are written in first-order logic, and it is possible to define custom functions and predicates for use in annotations together with ACSL built-ins. Indeed, ACSL features its own functions and predicates to describe memory states. However, it does not introduce any notion beyond the C standard, leaving each plug-in free to perform its own abstractions over the concrete memory layout.

```
/*@ requires \valid(a) && \valid(b);
    requires \separated(a,b);
    assigns *a, *b;
    ensures *a == \at(*b,Pre) && *b == \at(*a,Pre);
 */
void swap(int* a, int* b);
```

Fig. 3. Example of ACSL specification

```
/*@ behavior not_null:
    assumes a!=\null && b!= \null;
    requires \valid(a) && \valid(b);
    requires \separated(a,b);
    assigns *a, *b;
    ensures *a == \at(*b,Pre) && *b == \at(*a,Pre);

    behavior null:
    assumes a == \null || b == \null;
    assigns \nothing;

    complete behaviors;
    disjoint behaviors;
 */
void swap_or_null(int* a, int* b);
```

Fig. 4. Example of ACSL behavior

For instance, Fig. 3 provides a formal ACSL specification for a `swap` function. Informally, `swap` is supposed to exchange the content of two (valid) pointer cells given as argument.

The first pre-condition states that the two arguments must be valid **`int`** pointers, i.e. that dereferencing `a` or `b` will not produce a run-time error. In addition, the second pre-condition asks that the two locations do not overlap. **`\valid`** and **`\separated`** are two built-in ACSL predicates.

The **`assigns`** clause states that only the locations pointed to by `a` and `b` might be modified by a call to `swap`; any other memory location is untouched. Finally, the post-condition says that at the end of the function, `*a` contains the value that was in `*b` in the pre-state, and *vice versa*.

Function contracts can also be structured into **`behavior`**s. In addition to the clauses described above, a **`behavior`** can be guarded by **`assumes`** clauses, which specify under which conditions the **`behavior`** is activated. This provides a convenient way to distinguish various cases under which the function can be called. Behaviors do not need to be either **`complete`** (at least one behavior is active for any given call) or **`disjoint`** (there is at most one behavior active for a given call), but this can be stipulated with the appropriate clauses in order to check that the specification is correct in this respect. As an example, Fig. 4 presents the specification of a function `swap_or_null` similar to `swap` except that it will attempt to swap contents only when both pointers are non-null. This corresponds to the **`behavior`** `not_null`, in which case the function expects to have valid (in C a non-null pointer is not necessarily valid) and separated pointers and will act as above. On the other hand, in the `null` case, the function does nothing: at least one pointer is null, so the values cannot be swapped. We are always in exactly one of these two cases, so `null` and `not_null` are clearly complete and disjoint.

In addition to function specifications, ACSL offers the possibility of writing annotations in the code, in the form of **`assert`**ions, properties that must be true at a given point, or **`loop invariant`**s, properties that are true for each loop step. More precisely, a loop invariant is associated to a **`for`**, **`while`**, or **`do ... while`** loop. It must hold when arriving at the loop entry for the first time, and must be preserved by a loop step. That is, if we enter the loop body in a state that verifies the invariant, then the invariant must hold at the end of the block – except if we exit from the block through **`goto`**, **`break`** or **`continue`**. With these two properties, we can then prove by induction that the invariant is true for any number of loop steps (including 0). As with **`assigns`** clauses in function contracts, **`loop assigns`** clauses are a particular case of invariant that state which memory locations might have been modified since the beginning of the loop (implicitly stating that all other locations are not modified by the loop). Loop invariants are required for deductive verification (see Section 5).

```
int a[10];
/*@ loop invariant 0 <= i <= 10;
    loop invariant \forall integer j; 0 <= j < i ==> a[j] == j;
    loop assigns i, a[0 .. i-1];
 */
for (int i = 0; i < 10; i++) a[i] = i;
```

Fig. 5. Example of loop annotated with ACSL invariants

As an example, Fig. 5 presents a simple **`for`** loop initializing the elements of an array, together with its associated invariants. The first invariant specifies the bounds of the index of the loop. Note that the invariant uses `i<=10`, while the test in the code is `i<10`. Indeed, the invariant must be preserved by any step of the loop, including the last one, in which `i` is equal to `10` at the end of the step.

The second invariant states that the `i-1` first cells of the array have been initialized. It is true before entering the loop, as there is no `j` such that `0<=j<0`, so the implication is trivially true. Then, if we suppose that the invariant holds at the beginning of a loop step, the step will initialize cell `i` and increment `i`, so that the invariant is still true af the end of the step.

Similarly, the **`loop assigns`** clause indicates that the loop has potentially modified `i` and the first `i-1` cells of the array. Namely, this is true when entering the loop (nothing has been modified), and if we suppose that only `i` and the beginning of the array has been assigned, the next loop step also assigns `i` and the `i`th cell, again making the invariant true at the end of the step.

Loop invariants are not concerned with termination. Instead, termination can be ensured through the special clause **`loop variant`**. It requires an integer expression that strictly decreases during a loop step, while remaining positive each time the loop enters a new iteration. When both these conditions are met, we know that the loop can only be taken a finite number of times. An example of variant for the loop in Fig. 5 would be the following:

```
loop variant 10 - i;
```

Each plug-in can provide a validity status to any ACSL property and/or generate ACSL annotations. This lets ACSL annotations play an important role in the communication between plug-ins, as explained in Section 3.4.

3.3. Projects

Frama-C allows a user to work on several programs in parallel thanks to the notion of *project*. A project consistently stores a program with all its required information, including results computed by analyzers and their parameters. Several projects may coexist in memory at the same time. A non-interference theorem guarantees project partitioning [Sig09]: any modification on a value of a project $\mathcal{P}$ does not impact a value of another project $\mathcal{P}'$.

Such a feature is of particular interest when a program transformer such as E-ACSL (Section 7), Aoraï (Section 8), or Slicing (Section 10.1) is used. The result of the transformation is a fresh AST that coexists with the original one, making backtracking and comparisons easy. This is illustrated in Section 10.2. Another use of projects is to process the same program in different ways – for instance with different analysis parameters.

3.4. Collaborations across analyzers

In Frama-C, analyzers can interoperate in two different ways: either *sequentially*, by chaining analysis results to perform complex operations; or *in parallel*, by combining partial analysis results into a full program verification. The former consists in using the results of an analyzer as input to another one thanks to the plug-in database stored by the Frama-C kernel. Refer to Section 10.1 for an illustration of a sequential analysis.

The parallel collaboration of analyzers consists in verifying a program by heterogeneous means. ACSL is used to this end as a collaborative language: plug-ins generate program annotations, which are then validated by other plug-ins. Partial results coming from various plug-ins are integrated by the kernel to provide a consolidated status of the validity of all ACSL properties. For instance, when the Value plug-in (Section 4) is unable to ensure the validity of a pointer p, it emits an unproved ACSL annotation **assert \valid**(p). ACSL's blocking semantics states that an execution trace leading to an invalid property stops its execution (see [GGJK08, HMM12, CS12]). Thus, Value can assume that p is valid from this program point onwards, since the execution can only continue if the **assert** is valid. The WP plug-in (Section 5) may later be used to prove this hypothesis, or it can be checked at runtime by leveraging the E-ACSL plug-in (Section 7). The kernel automatically computes the validity status of each program property from the information provided by all analyzers and ensures the correctness of the entire verification process. For that, plug-ins can set the *local* validity status of an ACSL property, together with its set of *dependencies*. Those dependencies can be:

- other ACSL properties that are assumed to be true;
- the reachability of a statement;
- some parameters of the plug-in that set a hypothesis over the memory state of the program.

With this information, the kernel can then compute a *consolidated* validity status for each ACSL property. [CS12] presents the algorithm that is used for that purpose. Its main correctness property can be stated as: "*if the consolidated status of a property is computed as valid [resp. invalid] by the kernel, then the property is valid [resp. invalid] with respect to ACSL semantics*". This algorithm is also complete: *"if a property is proven valid [resp. invalid] by at least one analyzer, then its computed consolidated status is valid [resp. invalid] as soon as one other analyzer does not prove the contrary, in which case the special status 'inconsistent' is computed."*[1]

4. Abstract Interpretation

The Value plug-in [CYP13], (short for Value Analysis) is a forward dataflow analysis based on the principles of *abstract interpretation* [CC77], which performs whole-program analyses. For each instruction of the program, the information inferred by the plug-in are twofold:

1. A flag indicating the possibility that the execution of the instruction may fail at runtime – or at least invoke an undefined behavior. In this case, an *alarm* (see Section 4.2) is emitted, to warn the user that the program may be incorrect.
2. For each memory location, an over-approximation of the values it may contain. All the executions of the instruction that are possible starting from the function chosen as the entry-point of the analysis are taken into account.

Abstract interpretation links a *concrete* semantics, typically the set of all possible executions of a program, in all possible execution environments, to a more coarse-grained, *abstract* one. Any transformation in the concrete semantics must have an abstract counterpart that captures all possible outcomes of the concrete operation. This ensures that the abstract semantics is a sound approximation of the runtime behavior of the program, and guarantees the correctness of the information inferred by Value.

Value, and abstract interpreters in general, proceed by symbolic execution of the program, translating all operations into the abstract semantics. When several execution paths are possible, *e.g.* when analyzing an **if** statement, the abstract interpreter explores all branches, and takes the union of the results at the point where those branches join together (*e.g.* after the **if** statement).

Special care must be taken for loops, as this process is not guaranteed to terminate – except for very simple abstract domains, that usually cannot give any useful result on real programs. In abstract interpretation, termination of looping constructs is ensured by *widening* operations. Here, instead of simply taking the union of the previous results and the abstract state resulting from the current loop step, the interpreter uses an over-approximation (a wider state) of this union. This operation is designed so that it can only be called a finite number of times before reaching the widest abstract value $\top$. (The value $\top$ usually indicates that we do not know anything about the possible concrete states of the program, but abstract interpreters typically widen only the parts of their state that change from one iteration to the next, resulting in a controlled loss of precision.) This property of the widening operation ensures that the analysis will only go through the loop

[1]If all the analyzers and axioms are correct, and all the implicit hypotheses which the analyzers are based on (about the memory model, etc.) are verified, inconsistency can not happen.

```c
int S=0;

int T[20];

int main(void) {
     int i;
     int *p = &T[0];
     for (i = 0; i < 20; i++) {
          S = S + i;
          *p = S;
          p++;
          Frama_C_show_each_loop(S,i,p);
     }
     return S;
}
```

Fig. 6. Example of program analyzed with Value

a finite number of times. The number of steps the operator can take before returning the most imprecise abstract value acts as a balance between the precision of the analysis and the time it takes to complete. In Value, the number of steps of analysis that may take place without any widening step is parameterized by the `-wlevel` option, and intermediate widening bounds can be added by the user. However, those two mechanisms are mostly superseded by the propagation of unjoined states (see Section 4.3).

Function calls are handled by Value essentially through a symbolic inlining of the body of the function. This ensures that the analysis is fully context-sensitive, and permits maximal precision. The downside of this approach is a potentially high analysis time. If needed, the user can abstract overly complex functions by an ACSL contract, verified by hand or discharged with another analysis. In this case, the body of the function is skipped by Value. This also allows the analysis of recursive function, that cannot be handled through body inlining without endangering the termination of the analysis. Namely, Value currently rejects programs with recursive calls (that are seldom encountered in embedded code anyway).

We illustrate an analysis with the program presented in Fig. 6. Value is launched using the following command-line:

```
frama-c -val value_example.c
```

The program stores the sum of the integers from `0` to `i` in the `i`th cell of `T`. In addition, we use the built-in `Frama_C_show_each_` to output the abstract values computed for its arguments each time the analysis reaches the call. The relevant parts of the output are shown in Fig. 7. As will be described in Section 4.1, Value abstracts integers with sets when there are few elements, and intervals otherwise. Pointers are modelled as pairs of a base address and an (integer) offset, computed in bytes. We use the default architecture proposed by Frama-C, namely `x86-32`, so that an **`int`** is 4 bytes long.

In the initial state, as mandated by the C standard, all global variables are initialized to `0`. It is also possible to use the `-lib-entry` when the starting point of the analysis is not the `main` function of the program. In that case a custom, generic analysis context is generated, in which globals are not assumed to be 0-initialized.

In the first step of the loop, both `S` and `i` are still at `0`, while `p` points at the first cell of `T`. The second step shows the union of the results seen so far at this program point. After some further steps, we start seeing the results of the widening operator: as `S` is always increasing, Value widen its contents, resulting in an interval with a quickly increasing upper bound. On the other hand, `i` is always known to be between `0` and `19`, thanks to the test for exiting the loop. Value is also able to compute an upper bound for `p`, using a mechanism that will be explained in Section 4.2. Note that an additional piece of information is inferred automatically: the offset of `p` is always equal to `0` modulo `4`, *i.e.* we are reading well-aligned **`int`**s.

The bound found for `p` is not sufficient to ensure that we can always read safely from `p`. Similarly, Value cannot show that we can add `S` and `i` without overflow (signed overflow is undefined by the C standard and thus treated as a potential error by Value). Conversely, the fact that the `S+i` computation cannot overflow is used to show that `S` and the elements of `T` are necessarily positive.

As it cannot guarantee that the corresponding operations are correct, the analyzer emits two *alarms* for the potential invalid read and signed overflow, according to the mechanism described in Section 4.2. In

```
[value] Values of globals at initialization
        S IN {0}
        T[0..19] IN {0}
[value] Called Frama_C_show_each_loop({0},{0},{{ &T + {4} }})
[value] Called Frama_C_show_each_loop({0; 1},{0; 1},{{ &T + {4; 8} }})
...
[value] Called Frama_C_show_each_loop([0..60],[0..19],
                                      {{ &T + {4; 8; 12; 16; 20; 24; 28; 32} }})
[value] Called Frama_C_show_each_loop([0..79], [0..19], {{ &T + [4..36],0%4 }})
[value] Called Frama_C_show_each_loop([0..146], [0..19], {{ &T + [4..80],0%4 }})
value_example.c:10:[kernel] warning: out of bounds write. assert \valid(p);
[value] Called Frama_C_show_each_loop([0..165], [0..19], {{ &T + [4..80],0%4 }})
...
[value] Called Frama_C_show_each_loop([0..32805], [0..19], {{ &T + [4..80],0%4 }})
value_example.c:9:[kernel] warning: signed overflow. assert S+i <= 2147483647;
[value] Called Frama_C_show_each_loop([0..2147483647],[0..19],
                                      {{ &T + [4..80],0%4 }})
[value] Recording results for main
[value] done for function main
[value] ====== VALUES COMPUTED ======
[value] Values at end of function main:
  S IN [0..2147483647]
  T[0..19] IN [0..2147483647]
  i IN {20}
  p IN {{ &T + [0..80],0%4 }}
```

Fig. 7. Output of Value on the example

fact, these alarms are an example of *false alarms*, that are only due the over-approximations made by the analyzer, but cannot occur on any concrete execution. Reducing the number of false alarms (without missing real issues) is of course the goal of any correct abstract interpreter. Section 4.3 will show how it is possible to increase the precision of Value, at the expense of the computation time, to make these alarms disappear.

4.1. Abstract domains

The domains currently used by Value to represent the abstract semantics of values and of the whole memory are described below.

Integer computations. Small sets of integers are represented as sets, whereas large sets are represented as intervals with congruence information [Gra91]. For instance, $x \in [3..255], 3\%4$ means that x is such that $3 \leq x \leq 255$ and $x \equiv 3 \bmod 4$. Another example is the offset for the pointer `p` of Fig. 6, which is guaranteed to be 32-bits aligned starting from `&T[0]`, and less than `40`.

Floating-point computations. The results of floating-point computations are represented as IEEE 754 [IEE08] double-precision finite intervals. Operations on single-precision floats are stored as doubles, but are rounded as necessary. Obtaining infinities or NaN as results of floating-point computations is treated as an undesirable error. By default, Value considers that floating-points computations are handled in the same way on the target architecture than on the machine where the analysis takes place, with the same rounding mode, which is not supposed to be changed by the program itself. It is however possible to compute results that take into account all possible rounding modes, as well as the most common hardware deviations from IEEE 754 standard, such as the use of 80-bits registers for storing intermediate results on IA32 processors, or the `fmadd` instruction on PowerPC.

Pointers. To verify that invalid (e.g. out-of-bounds) array/pointer accesses cannot occur in the target program, Value assumes (and verifies) that the program does not purposely use buffer overflows to access neighboring variables [ISO07, §6.5.6:8]. The abstract representation of pointers reflects this assumption: addresses are seen as offsets with respect to symbolic *base addresses*, and have no relation with actual locations in virtual memory space during execution. In particular, it is impossible to move from one base address to another using pointer arithmetic. Offsets are plain integers, and are represented by the corresponding abstract domain. Base addresses can be

- the address of a local or global variable,
- the address of a function formal parameter,
- the address of a string constant,
- the special `NULL` base, which is used to encode absolute addresses.

The fact that `NULL` is a base address implies that an absolute numerical address is considered by Value to be separated from any memory location belonging to another variable of the program. In addition, Value considers that, by default, absolute numerical addresses (that are pervasive in embedded code) are invalid. If needed, the user must explicitly indicate which range of absolute addresses can be accessed by the program, using option `-absolute-valid-range`.

The base addresses above are sufficient for programs that do not perform dynamic memory allocation. `malloc` requires a special treatment. The simplest possibility is to consider a special array on which all calls to `malloc` return distinct offsets. However, if some precision is lost on the indexes of the `malloc`'ed blocks, Value will start joining values over separated blocks. More precise implementations allow to consider separate arrays for each call site of `malloc`, or even for each single call, at the expense of the computation time of the analysis. In the latter, most precise case, termination is not even guaranteed anymore when a call to `malloc` occurs in a loop, as we keep adding new base addresses to the state. Various implementations of `malloc` are implemented in Value and can be chosen by the user depending on the precision needed and the time that one is willing to allow for the analysis to complete.

Addresses are represented as maps from base addresses to integers, the latter representing the possible offsets from the base. This is slightly more precise than the traditional abstraction in which an address is seen as the pair of (1) the possible bases, and (2) the possible offsets for all the bases. Hence, Value can represent precisely a pointer that could be either null, equal to the address of a variable `x`, or equal to the addresses of the cells 3 to 10 of an array `T` of 16 bits integers. Expressed using Value's concrete syntax, in which offsets mean a number of bytes, this pointer is represented as `{{ NULL; &x; &T+[4..18],0%2 }}`. During the analysis, the abstract value for a given pointer often contains at most two bases (`NULL` and another variable). However, the increase of precision compared to a representation such as `<{NULL; &x; &T}, [0..18],0%2}>` is important to the plug-ins that reuse Value results (Section 10.1).

Indeterminate values. The content of some memory locations is deemed *indeterminate* by the C standard. Examples include uninitialized local variables, struct padding, and dereferencing pointers to variables outside of their scope [ISO07, §6.2.4:2]. Having indeterminate contents in memory is not an error, but accessing an indeterminate memory location is. To detect those, the values used to represent the contents of memory locations are taken not directly from the abstract domain used for the values of an expression, but from the lattice product of this domain with two two-valued domains, one for initializedness and the other for danglingness.

Of course, the abstract transfer functions take into account the scopes of variables. A pointer containing the address of a local variable is automatically transformed into a dangling indeterminate value when the analyzer leaves the block in which the variable is declared. A naive implementation of this operation is linear in the side of the abstract state, which is unacceptably costly. Since exiting a block occurs very frequently, the interpreter keeps track of all the variables that may contain the address of a local variable or a formal parameter; only those variables need to be checked when exiting the corresponding scope.

Memory. The abstract memory state maps each base address to a representation of a chunk of linear memory, of a size that is defined by the type of the base address. Each chunk itself maps consecutive ranges of bits to values. The memory representation is untyped: a **`struct`** `s` with two scalar fields is *not* seen as two distinct variables. Instead, depending on the code that filled the contents of the **`struct`**, the corresponding memory chunk will contain one range (if `s` is uninitialized or is initialized to the same value everywhere), two (the standard case), or more than two ranges (if the **`struct`**'s fields are written byte after byte). This representation makes straightforward the handling of unions and heterogeneous pointer conversions during abstract interpretation.

The first example of Fig. 8 shows three functions filling the contents of the **`struct`** mentioned above. For

```
 1 struct s_t {
 2   int v1;
 3   int v2;
 4 };
 5
 6 struct s_t s;
 7
 8 void f1(int c) {
 9   s.v1 = 1;
10   s.v2 = c;
11 }
12
13 void f2() {
14   s.v1 = s.v2 = 5;
15 }
16
17 void f3() {
18   s.v1 = 12;
19   s.v2 = 0x01020304;
20   *((char*)&s+4) = 0x08;
21   //@ assert s.v2 == 0x01020308;
22 }
```

```
 1 union u_t {
 2   int* p;
 3   char v;
 4 };
 5
 6 union u_t u;
 7 int x[2], y;
 8
 9 void f(int c) {
10   if (c) {
11     u.v = c % 64;
12   } else {
13     u.p = &x[0];
14   }
15   y = 3;
16 [...]
17   if (...) {
18     *u.p = 1;
19     //@ assert y == 3;
20   }
21 }
```

Fig. 8. Example programs for memory abstraction

each case, we give below the abstract representation of the memory chunk corresponding to `s`. We assume an analysis in which **`sizeof(int)`**`=4`, on a little-endian processor.

$$\texttt{f1}: \begin{cases} [0..31] & \mapsto \{1\} \\ [32..63] & \mapsto [..-..] \end{cases}$$

$$\texttt{f2}: \begin{cases} [0..63] & \mapsto \{5\}\text{ repeated each 32 bits} \end{cases}$$

$$\texttt{f3}: \begin{cases} [0..31] & \mapsto \{12\} \\ [32..39] & \mapsto \{8\} \\ [40..63] & \mapsto \{0\text{x}01020304\}\text{ size 32 bits, bits 8 to 31} \end{cases}$$

The function `f1` yields the simplest result. Two ranges, 32 bits long each, are present. They correspond to each field of the type **`struct`** `s_t`. The first field is mapped to the abstract value corresponding to the singleton `{1}`. The second is mapped to an integer interval on which no precise information is known. Notice that we do not store the information `[-2147483648..2147483647]`, which would also have been correct (assuming the **`int`** type is signed). It is simpler to store the degenerate information `[..-..]`, which will be transparently converted to `[-2147483648..2147483647]` or `[0..4294967295]` when `s.v1` is read through an l-value of type **`int`** or **`unsigned int`** respectively. The representation `[..-..]` is more compact, and is easier to share when multiple successive fields have the same value, as explained just below.

Perhaps surprisingly, after the execution of function `f2`, the abstract memory state for `s` does not contain two bindings, but a single one. This binding encompasses the entire range of bits `0..63`, which are mapped to the 32 bits value `{5}`. Here, the size of the bound value 32 is kept by the memory abstract domain, not by the value one. As a result, since 32 bits are too short to cover the entire range `0..63`, the value is implicitly considered as repeating itself until all 64 bits are filled (here, twice). This representation is particularly economical for arrays, which often contain the same abstract value for all cells. Furthermore, the initial state of the analysis is also very compact, as all global variables that do not have an explicit initializer are represented by a single interval mapped to the singleton value `{0}`.

Function `f3` is the most involved example. The operation at line 20 overwrites a part of the value written at line 19, which is automatically handled by the memory abstraction domain. Since bits 32 to 39 are completely overwritten, the entire range is mapped to the value `{8}` (which has implicitly size 8 bits, as this is the width of the range). Things are more complex for the range 40-63. One solution would be to ask the value abstraction for a new value that corresponds to the bits 8-31 of the abstract value `{0x01020304}`. However, this results in an unrecoverable loss of precision if the abstract value contained pointer addresses, as explained in the section below. Instead, the memory layer itself handles the fact that this range of bits contains only parts of the value `{0x01020304}`, by indicating that only the bits 8 to 31 of the value are used. Notice that this choice does not result in a loss of precision when the value of `s.v2` is read later on. The memory and value abstract domain cooperate to "stitch" together the two ranges of bits into an abstract value. In this example, this results into `{0x01020308}`, and Value is able to prove the assertion at line 21.

It may seem strange to use the bit as the base unit to index ranges, as the byte is the smallest addressable unit in C. However, bit-based accesses are needed to handle bit-fields [ISO07, §6.2.6.1:4] with a memory layout compatible with the one used by standard compilers. The C standard itself does not specify bit-fields layout, but then again, no more than any kind of data. Aligning all bit-fields on a byte frontier would result in binary representations that violate too many assumptions made by low-level code. But Value users are in fact rarely exposed to bit-based ranges, as memory states are shown to the user through pretty-printers that use the type information available. At the end of functions `f1`, `f2` and `f3` respectively, the contents of `s` are shown as displayed below.

```
f1:  s.v1 IN {1}
      .v2 IN [--..--]

f2:  s{.v1; .v2} IN {5}

f3:  s.v1 IN {12}
      .v2[bits 0 to 7] IN {8}
      .v2[bits 8 to 31]# IN {16909060}%32, bits 8 to 31
```

Since there exists at least a memory abstraction for each reachable statement of the program, the data structure used to represent these states must be very efficient in order to scale to programs of several hundreds of thousands of line of codes, as in [CHK$^+$12]. In particular, it must share common parts of the states across statements as much as possible, and allows an efficient merge of two states as well as testing the inclusion of a state in another one. This sharing is obtained through *hash-consing*, a well-known technique in functional programming (see *e.g* [Fil00]). The representation of abstract memory states is described in [BC11]. Sharing common sub-states through hash-consing also required some improvement to OCaml's standard implementation of weak pointers and weak hash tables [CD08]. In addition, this sharing must be preserved across serialization and deserialization when saving Frama-C internal state on disk and then reading from such a state to perform further analyses. Finally sharing must be built on-the-fly when deserializing, and not after having deserialized the entire state. This also required some customization of OCaml's unmarshalling functions [CDS11].

Degenerate pointers. One of the key design choices of Value is to support low-level C code[2], that manipulates memory in a potentially type-unsafe way. Typical patterns include accessing the byte-level representation of the memory, potentially on pointer values. But even for code that respects the C standard strict aliasing rule [ISO07, §6.5/7], the imprecisions inherent to static analysis cause the apparition of abstract values that contain some bits of a pointer address. In the second example of Fig. 8, joining the memory states after both branches of the **`if`** results in a value quite complex to represent for `u.v`: "either the first 8 bits of `&x[0]` (that furthermore depend on the endianness), or the numeric interval [0..63]". The memory abstraction presented above is able to represent the result of joining `&x[0]` with an integer value of size **`sizeof`**`(`**`int`**`*)`, but not with something strictly smaller. As a consequence, there is no available value in the value and memory abstractions presented yet to represent the field `u.v` – or the three subsequent bytes – after the join.

This example also shows that it is not possible to abort the analysis as soon as a pointer value cannot be represented precisely. Otherwise, too many defined programs would be rejected. Making `u.v` equal to $\top$ would also be too coarse, as writing through `u.p` would cause the loss of any information about the memory state. Instead, we add to the abstraction a family of new values representing "some bits of the addresses of certain variables". In our example, inside `u.v`, only the address of `x` may be present, and Value represents it as `{{ garbled mix of &{x} }}`. Garbled mix embodies *controlled degeneration*: although quite a bit of precision has been lost, we know that `u.v` and `u.p` can only contain bits of the address of `x` – in particular, no bits of `&y`. Thus, if the program was defined in the first place,[3] `u.p` would have contained something

[2]With the restriction that the program does not manually manipulate the function stack.

[3]As explained in the paragraph "Pointers" above, the C standard forbids the use of pointer arithmetic to address one memory block starting from a pointer to another block, and a defined program must verify this property.

derived from &x, and the operation *u.p = 1 does not modify y. Hence, the assertion at line 19 does hold. On the other hand, after the operation at line 18, the entire block for x contains the abstract value [..-..] representing the entire numerical range, as we cannot be sure that we are writing at &x[0] or at another offset from &x[0]. Value also signals that u.p may be completely invalid, for example if it contained &x[4].

4.2. Alarms

Each time a statement is analyzed, any operation that can lead to an undefined behavior (e.g. division by zero, out-of-bounds access, etc.) is checked, typically by verifying the range of the involved expression – the denominator of the division, the index of the array access, etc.

When the abstract semantics guarantees that no undesirable value can occur, one obtains a static guarantee that the operation always executes safely. Otherwise, Value reports the possible error by an *alarm*, expressed as an ACSL assertion. As such, those alarms can afterwards be read and checked by other Frama-C plug-ins, such as WP (see Section 5). For instance, Fig. 7 contain two typical examples of alarms. First, we have a potential invalid dereference, that lead to the ACSL assertion **\valid**(p). Then, we find a potential arithmetic overflow, that leads to the constraint that S+i<=MAX_INT.

Each alarm may signal a real error if the operation fails at runtime on at least one execution, or a *false alarm*, caused by the difference in precision between the concrete and abstract semantics. More precise state propagation for the abstract semantics typically results in fewer false alarms, but lengthens analysis time.

Upon emitting an alarm, the analyzer reduces the propagated state accordingly, and proceeds onwards. Reducing the state means taking into account – when this is possible for the abstract domain – the fact that the predicate corresponding to the alarm does hold. Value is able to evaluate and to reduce its state based on a limited subset of ACSL properties. Of course, this subset includes all the alarm predicates that are emitted by Value itself. User-written annotations can also be used to propagate reduced states, thus improving precision and speed (see Section 4.3). The exact supported range of predicates increases at each new release of Frama-C.

In the example of Fig. 6, the alarm **\valid**(p) generated for the access to *p at line 10 is used to restrict the value of p to the range {{ &T + [0..76] }}, which contains all the valid offsets inside T. This is sufficient to show that the range {{ &T + [4..80],0%4 }} is indeed a fixpoint for the first instruction of the loop.[4]

4.3. Propagation of unjoined states

Value's domains are non-relational: although they store the possible values for all the variables of the program, no information is stored regarding *e.g.* the sum or the equality of two variables. Typically, for the example of Fig. 6, no information is kept about the difference between p and i. Each time we enter the loop, this difference is constant and equal to &T[0]. This information would be useful to prove that the memory access *p at line 10 remains within bounds, as the equality p == T+i combined with i<20 is sufficient to prove that p does not goes past the bounds of T.

To remedy the absence of such relational information, Value instead allows the independent propagation of multiple distinct states per statement. This alleviates for a large part the need for relational domains, by implicitly encoding relations in the disjunction of abstract states.

Typically, by choosing to propagate k distinct states, the user can ensure that simple loops with less than k iterations are entirely unrolled. The bound k can be adjusted on a per-function basis if required. As an example, we can go back at the example of Fig. 6 and use the following command, -slevel being the option that sets the value of k.

```
frama-c -val -slevel 20 value_example.c
```

Results are shown in Fig. 9. Since we are not mixing the results of each loop step in a single state, Frama_C_show_each always display precise values, and the values of the cells of T at the end of the computation are also singletons. Moreover, the (false) alarms emitted because of the over-approximations

[4]The upper bound &T + 80 is automatically tried by the widening heuristics, which explains why the analysis converges on this value.

```
...
[value] Called Frama_C_show_each_loop({0},{0},{{ &T + {4} }})
[value] Called Frama_C_show_each_loop({1},{1},{{ &T + {8} }})
[value] Called Frama_C_show_each_loop({3},{2},{{ &T + {12} }})
[value] Called Frama_C_show_each_loop({6},{3},{{ &T + {16} }})
[value] Called Frama_C_show_each_loop({10},{4},{{ &T + {20} }})
...
[value] Called Frama_C_show_each_loop({190}, {19}, {{ &T + {80} }})
[value] ====== VALUES COMPUTED ======
[value] Values at end of function main:
  S IN {190}
  T[0] IN {0}
   [1] IN {1}
   [2] IN {3}
   [3] IN {6}
[...]
   [18] IN {171}
   [19] IN {190}
  i IN {20}
  p IN {{ &T + {80} }}
```

Fig. 9. Output of Value with multiple states per statement

have disappeared: p is known to always point to a valid cell of T, and S+i is at most 190, so that there is no overflow.

Successive conditionals are also handled more precisely: the abstract states remain separate after the two branches of the conditional have been analyzed. In addition, it is also possible to force a separation of states by writing ACSL assertions in a disjunctive form. If k is large enough, Value will then attempt to reduce the current state according to each part of the disjunction, and will continue the analysis with the separate states. For instance, consider the following program.

```
int x,y;

void main (int c) {
     if (c) { x = 10; } else { x = 33; }
     if (!c) { x++; } else { x--; }

     if (c<=0) { y = 42; } else { y = 36; }
     if (c>0) { y++; } else { y--; }
}
```

Without using separate states, Value concludes that the possible values for x and y are {9; 11; 32; 34} and {35; 37; 41; 43} respectively. Indeed, there is no relation between the value of c and the values of x and y after the first and third **if**.

With -slevel 2, y is known to be either 37 or 41. Indeed after the third **if**, we have two states. one in which c<=0 and y==42, and one where c>0 and y==36. Each state can only take one branch of the fourth **if**, so that there is only two possible results for y. On the other hand, x still has three possible values, namely {9; 11; 34}. The issue is that in the then branch of the first **if**, c can be anything but 0, but this information is not representable as an interval (it is the union of two disjoint intervals). Thus, in the state where x is equal to 10, we do not know anything about c, and both branches of the second **if** have to be explored in that case. To overcome this issue, we can force Value to split its state beforehand, by adding the following ACSL assertion before the first **if**.

```
//@ assert c<=0 || c>0;
```

This assertion is of course marked as valid by Value, ensuring that we have not introduced any additional hypothesis on the program. This time, with -slevel 3, we have two states before the first **if**, one where c<=0, and one where c>0. Each state is propagated separately, and this time since 0 is a bound of the interval for c, it can be removed from the possible values in the then branch, leading to three possible states after the first **if**:

- c<0 and x==10

- `c>0` and `x==10`
- `c==0` and `x==33`

For each of these cases, only one branch of the second **`if`** can be activated, leading to the result that `x` can only be either `9` or `34`.

Such ACSL disjunctions can play an important role in improving the precision of Value. In particular, decomposing floating-points interval can have a major impact on the approximation of the computations, as smaller intervals allows for tighter bounds of the rounding that is made for each operation. [PL10] shows how it is possible to automatically generate such disjunctions to obtain an appropriate precision when analyzing programs that perform floating-point computations.

From a theoretical standpoint, Value's approach of propagating distinct states is reminiscent of the *trace-partitioning* abstraction [MR05], with two important differences. First, the trace information is not represented within Value's abstract domains: states are kept separate only when they are different, and two branches of *e.g.* an **`if`** instruction that would lead to the same abstract state would result in the propagation of a single state. Although this precludes using information from the trace domain (which is not available), this may also ensure a faster convergence when different traces actually lead to identical abstract states. Second, trace partitioning requires heuristics or user-provided annotations to specify where multiple states should be propagated: on which **`if`**, for how many loop iterations, *etc.* Value's approach does not offer the same granularity, but is conversely easier to implement.

Finally, efficient propagation of distinct states requires very efficient datastructures, to alleviate the cost of computing the transfer functions multiple times on the same statement. Within Value, the datastructures representing the abstract semantics have been heavily optimized for speed and reduced memory footprint [BC11, CD08].

5. Deductive Verification

The WP plug-in is named after the *Weakest Precondition* calculus, a technique used to prove program properties initiated by Hoare [Hoa69], Floyd [Flo67] and Dijkstra [Dij68]. Recent tools implement this technique efficiently, for instance Boogie [Lei08] and Why/Why3 [Fil03, FP13]. Jessie [MM12], a Frama-C plug-in developed at INRIA, also implements this technique for C by compiling programs into the Why language. Frama-C's WP plug-in is a novel implementation of a Weakest Precondition calculus for generating verification conditions (VC) for C programs with ACSL annotations. It differs from other implementations in two respects. First, WP focuses on parameterization with respect to the memory model. Second, it aims at being fully integrated into the Frama-C platform, and to ease collaboration with the other verification plug-ins (especially Value) as outlined in section 3.4.

The choice of a memory model is a key ingredient of Hoare logic-based VC generators that target C programs (or more generally a language with memory references). A weakest precondition calculus operates over a language that only manipulates plain variables. In order to account for pointers, memory accesses (both for reading and writing) must be represented in the underlying logic. The simplest representation uses a single functional array for the whole memory. However, this has a drawback: any update to the array (the representation of an assignment `*p=v`) has a potential impact on the whole memory – any variable might have been modified. In practice, proof obligations generated that way quickly become intractable. Thus, various refinements have been proposed, in particular by Bornat [Bor00], building upon earlier work by Burstall [Bur72]. The idea of such memory models is to use distinct arrays to represent parts of the memory known to be separated, e.g. distinct fields of the same structure in the "component-as-array trick" of Burstall and Bornat. In this setting, an update to one of the arrays will not affect the properties of the others, leading to more manageable VC.

However, abstract memory models sometimes restrict the functions that can be analyzed. Indeed, a given model can only be used to verify code that does not create aliases between pointers that are considered *a priori* separated by the model. In particular, Burstall-Bornat models that rely on static type information to partition the memory are not able to cope with programs that use pointer casts or some form of **`union`** types.

In order to generate simpler VC when possible while still being able to verify low-level programs, WP provides different memory models that the user can choose for each ACSL property. The current version offers two main models:

```
goal swap_post:
  let a_0 = shift(global(P_a_741), 0) in
  let a_1 = shift(global(P_b_742), 0) in
  let a_2 = shift(global(L_tmp_744), 0) in
  forall Malloc_0 : int farray.
  forall Mint_0 : (addr,int) farray.
  forall Mptr_0 : (addr,addr) farray.
  let a_3 = Mptr_0[a_0] : addr in
  let x_0 = Mint_0[a_3] : int in
  let a_4 = Mptr_0[a_1] : addr in
  let x_1 = Mint_0[a_4] : int in
  let m_0 =
    Mint_0[a_2 <- x_0] : (addr,int) farray
  in
  let m_1 =
    m_0[a_3 <- m_0[a_4]] : (addr,int) farray
  in
  let x_2 = m_1[a_2] : int in
  let x_3 = m_1[a_4 <- x_2][a_3] : int in
  framed(Mptr_0) ->
  linked(Malloc_0) ->
  is_sint32(x_0) ->
  is_sint32(x_1) ->
  valid_rw(Malloc_0, a_3, 1) ->
  valid_rw(Malloc_0, a_4, 1) ->
  separated(a_3, 1, a_4, 1) ->
  is_sint32(x_2) ->
  is_sint32(x_3) ->
  ((x_0 = x_2) and (x_1 = x_3))
```

Fig. 10. Proof obligation for `Typed+raw` model

- The most abstract model, `Hoare`, roughly corresponds to Caveat's model [RSB$^+$99]. This model only uses logical variables in its translation. Thus it can only be used over functions that do not explicitly assign pointers or take the address of a variable, but provides compact VC.
- The default model is `Typed`. In this model, the memory state is represented by three distinct arrays, for integers, floating-point and addresses. Each array is indexed by addresses. An address is a pair of a base address and an offset. This representation allows to handle pointers, and some limited form of pointer casts.

Previous versions of WP provided a `runtime` model, which was very close to the concrete memory model of the CompCert [LB08] compiler, but it led to too complex proof obligations and was thus not usable in practice. It has been deprecated in the current version of WP and will be replaced by a newer low-level model `Bytes` at some later point.

As a refinement, `Typed` can be configured to avoid converting assignments into array updates when the code falls in the subset supported by `Hoare`. By default, this is done for variables whose address is not taken anywhere in the program (which are really behaving as "plain" variables). Memory model `Typed+raw` implements the "pure" model where all accesses are translated as array accesses. Conversely, model `Typed+ref` extends the usage of logic variables to the value of pointers used as "references", that is pointers that are neither assigned nor used in pointer arithmetic operations. This way, the overhead of `Typed` model with respect to `Hoare` is kept to the places where it is really needed.

Once a VC has been generated, it must be discharged. First, WP use an internal tool, Qed, that performs some trivial simplifications over the obtained formula (inlining of short definition, propagation of equalities, normalization of terms, etc.), which in some cases are sufficient to prove the VC. Otherwise, the simplified formula is handed over to external provers. WP natively supports two theorem provers: the automated SMT solver Alt-ergo [C$^+$], and the Coq proof assistant [Coq11]. Other automated provers can also be used through the multi-prover backend of Why 3. The advantages of using a dedicated back-end rather than relying completely on Why 3 are twofold. First, it removes a dependency over an external tool, meaning that for verification of critical software, there is one component less that needs to be assessed. Second, WP can take advantage of specific features of Alt-ergo, most notably native support for arrays and records, that occur quite often in typical VC.

As an example of the differences between the various memory models, we can go back to our `swap` example of Fig. 3. Given the following implementation, we can try to use WP to check that it satisfies the specification.

```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Using `frama-c -wp-model Hoare -wp swap.c` leads to an error, as the code contains pointer dereferences. Of the three `Typed` model variants, `Typed+raw` produces the longest proof obligation for the **`ensures`** clause (see Fig. 10). Namely, the `tmp` variable is represented by a cell of the `Mint_0` array, as

```
goal swap_post:
  forall Malloc_0 : int farray.
  forall Mint_0 : (addr,int) farray.
  forall a_0,b_0 : addr.
  let x_0 = Mint_0[a_0] : int in
  let x_1 = Mint_0[b_0] : int in
  let x_2 =
    Mint_0[a_0 <- x_1][b_0 <- x_0][a_0] : int
  in
  linked(Malloc_0) ->
  is_sint32(x_0) ->
  is_sint32(x_1) ->
  valid_rw(Malloc_0, a_0, 1) ->
  valid_rw(Malloc_0, b_0, 1) ->
  separated(a_0, 1, b_0, 1) ->
  is_sint32(x_2) ->
  (x_1 = x_2)
```

Fig. 11. Proof obligation for `Typed` model

```
goal swap_post:
  forall ra_a_0,ra_b_0,ta_tmp_0 : bool.
  forall a_2,a_1,a_0,b_2,b_1,b_0,tmp_0 : int.
  (ra_a_0 = true) ->
  (ra_b_0 = true) ->
  (a_2 = a_0) ->
  (a_1 = b_0) ->
  (a_0 = tmp_0) ->
  (b_2 = tmp_0) ->
  (b_1 = b_0) ->
  ((ta_tmp_0 = false)) ->
  is_sint32(a_2) ->
  is_sint32(a_1) ->
  is_sint32(b_2) ->
  is_sint32(b_1) ->
  ((a_2 = b_2) and (a_1 = b_1))
```

Fig. 12. Unsimplified proof obligation for `Typed+ref` model

well as the content of `*a` and `*b`. Moreover, `a` and `b` themselves are not accessed directly, but through a cell of the `Mptr_0` array. All these accesses leads to a rather complicated formula. With the `Typed` model, whose result is shown in Fig. 11, some of the complexity has disappeared, as `a`, `b` and `tmp` are represented as logic variables. Furthermore, the definition of `tmp` has been inlined away by Qed, which is then able to prove the first part of the conjunction by propagating equalities. The resulting proof obligation passed to Alt-ergo is thus much simpler. This is even more the case with `Typed+ref`, where accesses to `*a` and `*b`, seen as references, are represented by variables. In this setting, Qed is able to prove the VC all alone, leaving no proof obligation to be given to Alt-ergo. Figure 12 presents the proof obligation when most simplifications performed by Qed are deactivated: we can clearly see that we only deal with integer variables and that all memory accesses have been eliminated, leading to a very easy formula.

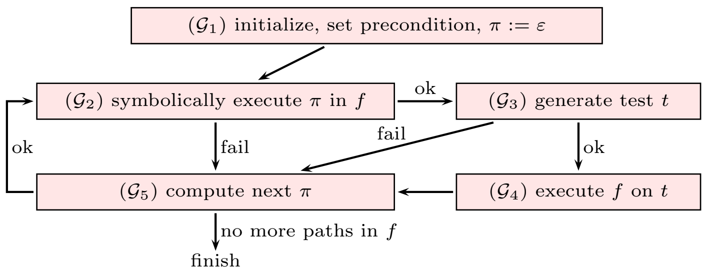
In contrast to Jessie, that relies on an external tool for VC generation, WP operates entirely within Frama-C. In particular, WP fills the property status table described in section 3.4 for each annotation on which it is run. The dependencies of such a status are the annotations taken as hypothesis during the weakest-precondition calculus, the memory model that has been used, and the theorem prover that ultimately discharged the VC. The memory model has a direct impact on the validity of the result: an annotation can very well be valid under model `Typed+ref` but not under `Typed`, as the former entails implicit separation hypotheses that are not present in the latter. In theory, the choice of a theorem prover is not relevant for the correctness of the status, but this information is important to fully determine a trusted toolchain.

Having WP properly embedded into Frama-C also allows for a fine-grained control over the annotations one wants to verify with the plug-in. WP provides the necessary interface at all levels (command-line option, programmatic API, and GUI) to verify targeted annotations (e.g. those yet unverified by other means in Frama-C, *cf.* section 3.4) as well as to generate all the VC related to a C function.

6. Concolic Testing

The PathCrawler plug-in operates on a C program p under test and a precondition restricting its inputs, and generates test cases respecting various test coverage criteria. The *all-path* criterion requires covering all feasible program paths of p. Since the exhaustive exploration of all paths is usually impossible for real-life programs, the k*-path* criterion restricts exploration to paths with at most k consecutive iterations of each loop. The PathCrawler [WMMR05, BDH$^+$09] method for test generation is similar to the so-called *concolic* (*conc*rete+symb*olic*) approach and to Dynamic Symbolic Execution (DSE), implemented by other tools (e.g. DART, CUTE, PEX, SAGE, KLEE).

PathCrawler starts by: *a.* constructing an instrumented version of p that will trace the program path exercised by the execution of a test case, and *b.* generating the constraints which represent the semantics


($\mathcal{G}_1$) initialize, set precondition, $\pi := \varepsilon$
($\mathcal{G}_2$) symbolically execute π in f
ok
($\mathcal{G}_3$) generate test t
fail
ok
fail
ok
($\mathcal{G}_5$) compute next π
($\mathcal{G}_4$) execute f on t
no more paths in f
finish
Fig. 13. The PathCrawler test generation method

of each instruction in p. The next step, illustrated by Fig. 13, is the generation and resolution of constraint systems to produce the test cases for a set of paths Π that satisfy the coverage criterion. This is done in the ECLiPSe Prolog environment [SS11] and uses Constraint Logic Programming. Given a *path prefix* π, i.e. a partial program path in p, the main idea [Kos10] is to solve the constraints corresponding to the symbolic execution of p along π. A constraint store is maintained during resolution, and aggregates the various constraints encountered during the symbolic execution of π. The test generation method follows the following steps:

($\mathcal{G}_1$) Create a logical variable for each input. Add constraints for the precondition into the constraint store. Let the initial path prefix π be empty (i.e. the first test case can be any test case satisfying the precondition). Continue to Step ($\mathcal{G}_2$).

($\mathcal{G}_2$) Symbolically execute the path π: add constraints and update the memory according to the instructions in π. If some constraint fails, continue to Step ($\mathcal{G}_5$). Otherwise, continue to Step ($\mathcal{G}_3$).

($\mathcal{G}_3$) Call the constraint solver to generate a test case t, that is, concrete values for the inputs, satisfying the current constraints. If it fails, go to Step ($\mathcal{G}_5$). Otherwise, continue to Step ($\mathcal{G}_4$) .

($\mathcal{G}_4$) Run a traced execution of the program on the test case t generated in the previous step to obtain the complete execution path. The complete path must start by π. Continue to Step ($\mathcal{G}_5$).

($\mathcal{G}_5$) Compute the next partial path, π, to cover. π is constructed by “taking another branch” in one of the complete paths already covered by a previous test case. This ensures that all feasible paths are covered (as long as the constraint solver can find a solution in a reasonable time) and that only the shortest infeasible prefix of each infeasible path is explored. If there are no more paths to cover, exit. Otherwise, continue to Step ($\mathcal{G}_2$).

PathCrawler uses Colibri, a specialized constraint solving library developed at CEA LIST and shared with other testing tools such as GATeL [MA00] and OSMOSE [BH11]. Colibri provides a variety of types and constraints (including non-linear constraints), primitives for labelling procedures, support for floating point numbers and efficient constraint resolution. Colibri also offers incremental constraint posting with on-the-fly filtering and automatic backtracking to a previous constraint state that are important benefits for search-based state exploration tools such as PathCrawler. PathCrawler is currrently limited to programs without function pointers, recursive functions and recursive data structures. PathCrawler is a proprietary plug-in, also available in the form of a freely accessible test-case generation web service [Kos].

7. Runtime Assertion Checking

The E-ACSL plug-in automatically translates an annotated C program into another program that reports a failure whenever an annotation is violated at runtime. If no annotation is violated, the behavior of the new program is exactly the same as that of the original one.

Such a translation brings several benefits. First, it allows the user to monitor the execution of a C code, in particular to perform what is usually called “runtime assertion checking” [CR06][5]. This is the primary goal of E-ACSL [KS13]. Second, it helps in combining Frama-C and its existing analyzers with other analyzers that do

[5]In our context, “runtime annotation checking” would be a better, more general expression.

a)

```
1  #include "limits.h"
2
3  /*@ requires x > INT_MIN;
4    @ ensures \result == -x; */
5  int opposite(int x);
6
7  int G = 10;
8
9  int main(void) {
10   int x = opposite(G);
11   int *p = &x;
12   /*@ assert *p == -10; */
13   return 0;
14 }
```

b)

```
1  extern int opposite(int x);
2
3  /*@ requires x > -2147483647-1;
4      ensures \result == -\old(x); */
5  int __e_acsl_opposite(int x)
6  {
7    int __e_acsl_at;
8    int __retres;
9    // check precondition
10   __store_block((void *)(& __retres),4U);
11   e_acsl_assert(x > -2147483647 - 1);
12   __e_acsl_at = x;
13   __retres = opposite(x);
14   // check postcondition
15   e_acsl_assert(__retres == -__e_acsl_at);
16   __delete_block((void *)(& __retres));
17   return __retres;
18 }
19
20 int G = 10;
21
22 int main(void)
23 {
24   int *p;
25   int x;
26   // memory monitoring
27   __store_block((void *)(& x),4U);
28   __store_block((void *)(& p),4U);
29   x = __e_acsl_opposite(G);
30   p = & x;
31   /*@ assert *p == -10; */
32   // check assertion
33   {
34     //check pointer dereferencing in assertion
35     int __e_acsl_valid_read;
36     __e_acsl_valid_read =
37       __valid_read((void *)p,sizeof(int));
38     e_acsl_assert(__e_acsl_valid_read);
39     e_acsl_assert(*p == -10);
40   }
41   // clear allocated memory
42   __delete_block((void *)(& x));
43   __delete_block((void *)(& p));
44   return 0;
45 }
```

Fig. 14. E-ACSL translation: **a)** initial code *vs* **b)** (simplified) instrumented code

not natively understand the ACSL specification language. Third, the possibility to detect invalid annotations during a concrete execution may be very helpful while writing a specification of a given program, *e.g.* for later program proving. Finally, an executable specification makes it possible to check at runtime assertions that cannot be verified statically, and thus to establish a link between monitoring tools and static analysis tools like Value or WP.

Annotations must be written in the E-ACSL specification language [Sig13, DKS13] which is a large subset of ACSL. Basically, it only excludes parts of ACSL which are not computable in finite time (unbounded quantifications and sets of values, axioms and lemmas, *etc*). Fig. 14 shows a simple example of translation (for a 32-bit architecture). Each E-ACSL annotation is checked by a call to `e_acsl_assert` function that reports a failure and exits the program if the annotation is not satisfied. The precondition and postcondition of function `opposite` are checked respectively at lines 11 and 15 of Fig. 14b, while the assertion at line 12 of Fig. 14a leads to the lines 33-40 of Fig. 14b that we explain below.

The translation scheme, shown in Fig. 15, translates each annotation into instrumented C code which ultimately performs a runtime check. Four specific features have to be pointed out. First, E-ACSL mathematical integers are represented in the translated code using GMP (GNU Multi-Precision library [GMP]) integers. This leads to a more verbose translation that can be avoided in many cases, for instance, when the result of an arithmetical operation can still be represented by a machine integer (of the same, or a longer C type). In the E-ACSL plug-in, for each integer, an interval-based type system infers on the fly a standard C integral type (if any) in which the integer fits. Thanks to this typing mechanism, there is in practice almost no GMP integer in the resulting code [DKS13] and in particular none in the example of Fig. 14.

Second, a custom C memory monitoring library has been implemented: it allows the user to monitor memory-related operations (calls to `malloc` and `free`, *etc*) in order to correctly handle memory-related E-ACSL constructs such as **`\valid`**. The instrumented code calls the monitoring library primitives to store validity and initialization information (whenever a memory location is allocated, deallocated and assigned), and to extract this information when evaluating memory-related E-ACSL constructs. To optimize the perfor-

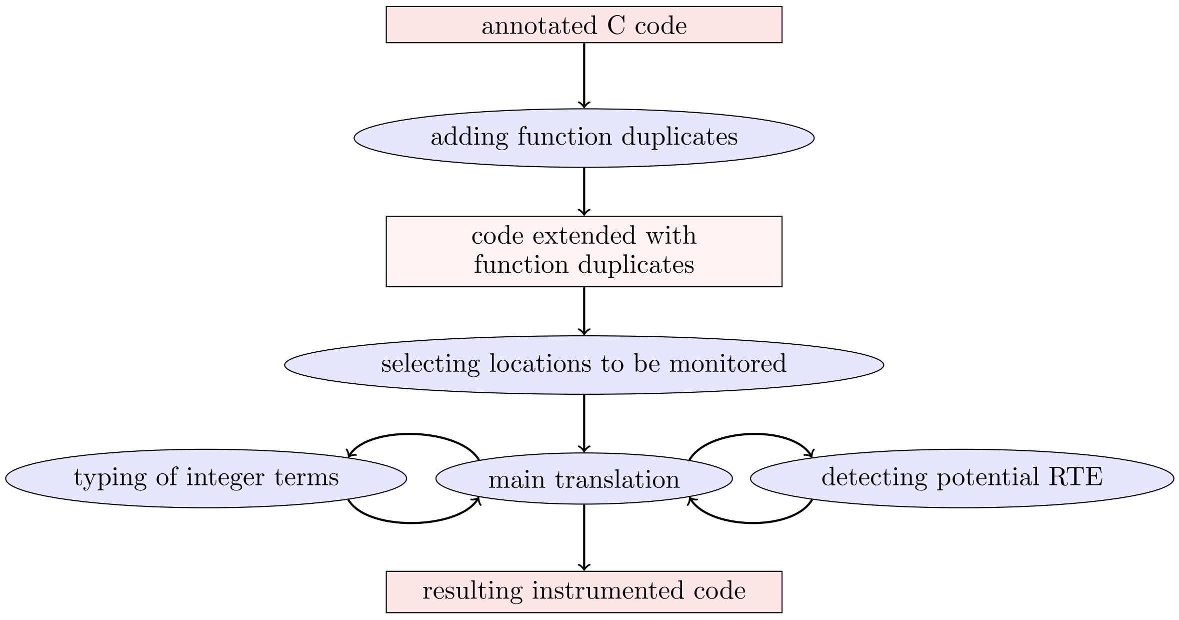

annotated C code
adding function duplicates
code extended with function duplicates
selecting locations to be monitored
typing of integer terms
main translation
detecting potential RTE
resulting instrumented code

Fig. 15. E-ACSL translation scheme

mance of the resulting code and avoid monitoring of irrelevant variables, a preliminary backward dataflow analysis has been implemented to determine a correct over-approximation of the set of memory locations that have to be monitored for a given annotated program [KPS13]. In the example of Fig. 14, calls to the library functions `store_block` and `delete_block` record allocation and deallocation of memory locations. The analysis detects that the monitoring of `G` is not required.

Third, the instrumented code added by the translation of E-ACSL annotations should not introduce runtime errors (RTE) itself. E-ACSL prevents runtime errors in the generated code by collaborating with another Frama-C plug-in called RTE [HS13]. E-ACSL calls RTE on each generated C expression. RTE generates E-ACSL annotations that prevent runtime errors in the corresponding expression. Then E-ACSL is used again to translate these additional annotations into C code [DKS13]. The process is guaranteed to terminate since the C expressions generated from these additional annotations produced by RTE are runtime-error-free. In our example, the access to `*p` at line 39 of Fig. 14b (resulting from the direct translation of the assertion by E-ACSL) could potentially provoke a runtime error. RTE detects this risk, and E-ACSL adds the pointer validity check using a call to the memory monitoring library `__valid_read` at line 37.

Fourth, a preliminary translation adds an additional function f' for each function f with an E-ACSL contract. f' contains the translation of the precondition, a call to f and the translation of the postcondition. The calls to f are replaced by calls to f' in the instrumented code to ensure that the contract of f is checked each time f is called. This way, it is possible to ensure runtime contract checking even for undefined (library) functions and to clearly separate the initial code from the code corresponding to the precondition and the one corresponding to the postcondition. In the example of Fig. 14, an additional function `__e_acsl_opposite` is used to check the contract of the declared-only function `opposite` each time it is called.

8. Temporal Specifications

The Aoraï plug-in [SP11, GS09] plays a particular role among the core Frama-C plug-ins. Indeed, it is one of the few whose primary aim is to generate ACSL annotations rather than attempting to verify them. Aoraï provides a way to specify that all possible executions of a program respect a given sequence of events, namely calling and returning from functions, possibly with constraints on the program's state at each event. The specification itself can be given either as a Linear Temporal Logic (LTL, [Pnu77]) formula or in the form of an automaton. In the former case, Aoraï uses ltl2ba [GO01] to obtain an equivalent Büchi automaton. In the

```
%init: S0;
%accept: OK;

S0: { f {{ N<=0}} () } -> OK
  | { f{{ N>0 }} ([g(){{\result==0}}]{,N-1}; g(){{\result!=0}}; h()) } -> OK
  | { f{{ N>0 }} ([g(){{\result==0}}]{N}) } -> OK;

OK: -> OK;
```

Fig. 16. Example of Aoraï automaton

latter case, Aoraï accepts as input an automaton in a custom language, where the transitions can be guarded by the event (call to, or return from of a given function) under consideration and a propositional formula over the globals of the program and the formals of the function (in the case of a call event) or the result (in the case of a return event).

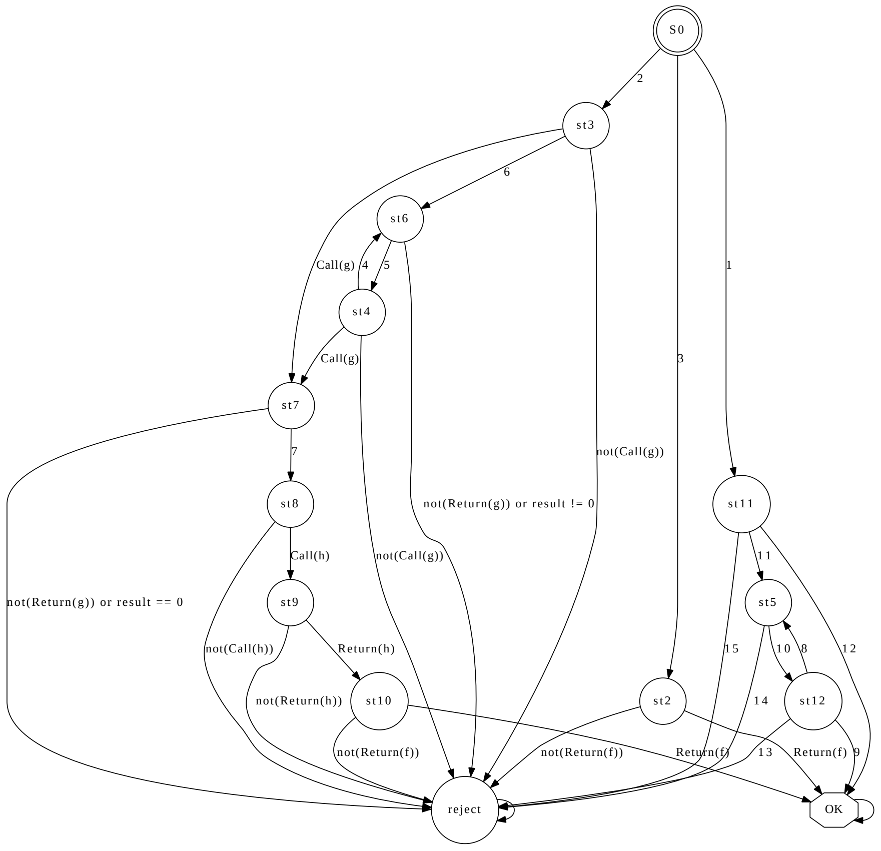
Newer versions of the plug-in feature an extended input language, where transitions can be guarded by a whole sequence of events, possibly including repetitions of sub-sequences. Such automata are translated by Aoraï into atomic ones, by introducing intermediate states and transitions so that each transition only deals with a single event. The translation can in addition introduce auxiliary variables, in particular counters that keep track of the number of times a repeated sub-sequence has been taken. These auxiliary variables are updated when the appropriate atomic transition is crossed. In addition, the normalized automaton contains a specific `reject` state, which is a sink. Transitions lead to this state from auxiliary states introduced by the normalization when the program does not follow the corresponding sub-sequence. Indeed, Aoraï's semantics mandates that at each event, for each currently active state in the automaton, there is at least one transition that can be taken. Thus, if we are exploring several sub-sequences and discover at some point that one of them is not matched by the current execution, we cross the transition to the `reject` state, leaving the other sub-sequences free to complete successfully.

As an example, Fig. 16 presents an automaton specifying three admissible scenarios for calling a function `f`. First, if the formal parameter `N` of `f` is non-positive, `f` must return immediately. Otherwise, `f` is supposed to call `g` at most `N` times, until `g` returns a non-zero result. Then, `f` calls `h` before exiting. Finally, if `g` returns `N` times `0`, `f` itself must return. The resulting automaton expressed with atomic transitions is shown on Fig. 17. The normalized automaton uses four auxiliary variables: `aorai_N` and `aorai_N_0` are used to recall the value of the formal parameter `N` of `f` throughout the execution, so that this value can be used in the guards of the transitions corresponding to the second and third cases of the original automaton respectively. Similarly, `aorai_counter` and `aorai_counter_0` count in both cases the number of calls to `g` made so far. We can also see the `reject` state and the transitions leading to it. For instance, `st5` has been introduced by the translation of the third case of the original automaton, where `g` is supposed to always return `0`. Then, if we return from `g` with a non-`0` value while `st5` is active, transition `14` will be taken, putting this branch in the `reject` state.

Given a normalized automaton, Aoraï generates ACSL specifications for each function `F` in the original C code. This instrumentation, summarized in Fig. 18, consists in two main parts: two prototypes whose specification represents the transitions for an atomic event (call or return from `F`), and the specification of `F` itself. As the automaton is not always deterministic, Aoraï represents active states by a set of boolean variables (`aorai_state_*`). These variables have either value `1` if the automaton is in the given state or `0` if this is not the case. As a refinement, if the user claims that the input automaton is deterministic, Aoraï will use a single **`int`** variable to represent the state of the automaton at a given program point. This usually leads to much more compact ACSL contracts. In addition, Aoraï also generates a set of lemmas to ensure that the automaton is truly deterministic. Namely, any two transitions from the same state of the automaton must then have disjoint guards.

Regardless of how active states are represented, functions `advance_automaton_*` provide for each state of the automaton a complete set of **`behavior`**s indicating at which conditions they are active or not after the corresponding event. The post-conditions of these behaviors also take care of setting the value of the auxiliary variables when a transition with auxiliary variables is crossed.

The specification of `F` comprises various items. First, at least one state among a given set must be active before the call. This set is determined by a coarse static analysis made by Aoraï beforehand. This static analysis phase does not handle indirect calls through function pointers, which is currently the main



Transitions:

1: `N > 0 and Call(f); aorai_N_0 <- N`

2: `N > 0 and Call(f); aorai_N <- N`

3: `N <= 0 and Call(f)`

4: `Call(g) and aorai_counter < aorai_N-1 aorai_counter <- aorai_counter + 1`

5: `\result == 0 and Return(g)`

6: `Call(g); aorai_counter <- 1`

7: `\result != 0 and Return(g)`

8: `Call(g) and aorai_counter_0 < aorai_N_0 aorai_counter_0 <- aorai_counter_0 + 1`

9: `aorai_N_0 <= aorai_counter_0 and Return(f)`

10: `result == 0 and Return(g)`

11: `Call(g); aorai_counter_0 <- 1`

12: `aorai_N_0 == 0 and Return(f)`

13:
```
(not(Return(f)) and not(Call(g))) or
(aorai_N_0 > aorai_counter_0 and not(Call(g))) or
(not(Return(f)) and aorai_counter_0 >= aorai_N_0)
```

14: `not(Return(g)) or result != 0`

15: `(not(Return(f)) and not(Call(g))) or (aorai_N_0 != 0 and not(Call(g)))`

Fig. 17. Example of normalized Aoraï automaton

```
/*@ behavior transition_1: assumes aorai_state_S_0 == 1 && condition;
    ensures aorai_state_S_next == 1; ... */
void advance_automaton_call_F(int x);

/*@ requires aorai_state_S_0 == 1 || aorai_state_S_1 == 1 || ...;
    requires aorai_state_S_0 == 1 ==> has_possible_transition_S0; ...
    ensures aorai_state_S_2 == 1 || aorai_state_S_3 == 1 || ...;
    ensures \old(aorai_state_S_0 == 1) ==> aorai_state_S_2 == 1 || ...; ...
    ensures aorai_state_S_2 == 1 ==> program_state_when_in_S_2; ... */
 int F(int x) {
   advance_automaton_call_F(x);
   // Body of F
   aorai_label:
   /*@ loop invariant aorai_state_I_0 == 1 || aorai_state_I_1 == 1 ...;
       loop invariant aorai_state_I_2 == 0 && aorai_state_I_3 == 0 ...;
       loop_invariant
         first_step && \at(aorai_state_S_0,Pre) == 1 ==>
         aorai_state_I_0 == 1;
       loop invariant
         !first_step && \at(aorai_state_S_0,Pre) == 1 ==>
         aorai_state_I_1 == 1;
       loop invariant
         \at(aorai_state_I_0,aorai_label) == 1 && aorai_state_I_1 == 1 ==>
         aux_variable_invariant_I_1;
   */
   while(c) ... ;
   advance_automaton_return_F(result);
   return result; }
```

Fig. 18. Aoraï's instrumentation

limitation of the plug-in. When a reject state exists as described above, the requirement becomes that at least another state beside it is active. Indeed, if all sub-sequences end up in the reject state, the implementation is not conforming to the automaton. In addition, as said above, for each active state, at least one transition must be activated by the call event. The main post-condition is that when the function returns, at least one state is active among those deemed possible by Aoraï's static analysis. It is refined by additional clauses relating active initial states with active final states, and the state of the program itself with the active final states. Aoraï's static analysis also keeps track of the value of auxiliary variables, and if they might be modified by F their possible value at the end of the function is given as post-conditions. Finally the main function has an additional post-condition stating that at least one of the acceptance states must be active at the end of the function (Aoraï does not consider infinite programs at the moment, *i.e.* it can only check for safety properties and cannot be used for liveness).

Annotation generation is geared towards the use of deductive verification plug-ins such as WP and Jessie for the verification of the specification. In particular, the refined post-conditions are mainly useful for propagating information to the callers of F in an Hoare-logic based setting. Aoraï also generates loop invariants for the same purpose. These loop invariants indicates which states of the automaton can be active at any given loop steps, and what are the possible values of the auxiliary variables during the execution of the loop. Again, they are obtained from Aoraï's static analysis. In addition, these invariants are refined according to the active states of the automaton when entering the function (marked by the Pre label) the active states when entering the loop for the first time (marked by the generated label aorai_label), and the fact that we are in the first step of the function or in a further step (indicated by the first_step ghost variable that is initialized to 1 and set to 0 as soon as the loop is entered). These distinctions allows to generate more precise loop invariants. However, it is still usually necessary to write additional invariants, in particular to glue together these auxiliary variables with the variables of the original program (for instance an auxiliary counter with the index of a corresponding **for** loop). This bias towards WP does not preclude the use of the Value plug-in to validate its specification, as Aoraï attempts to generate annotations that fit in the subset of ACSL that is understood by Value.

If we go back at the example of Fig. 16 over the very simple implementation given in Fig. 19, the resulting instrumented code is roughly 1000 lines long, mostly consisting in specifications. More precisely, Aoraï generates 200 post-conditions and 32 pre-conditions, while the loop in f is decorated with 39 invariants.

```
int random(void);

int g(int x)
{
  return random();
}

void h() { return; }

void f(int N)
{
  int i;
  int t;
  i = 0;
  t = 0;
  while (i<N && !t ) {
    t = g(i);
    i ++;
  }
  if (t) { h(); }
  return;
}
```

Fig. 19. Aoraï Implementation example

```
loop invariant
  1 == st12 ==> aorai_counter_0 == i;
loop invariant
  (1 == st8
  || (1 == st12 && aorai_counter_0 < aorai_N_0
      && 1 == st4 && aorai_counter < aorai_N)
  || 1 == st12 && aorai_counter_0 == aorai_N_0
  || (1 == st8) || (1 == st3 && 1 == st11));
```

Fig. 20. Additional invariants for Aoraï instrumentation

As explained before, some additional loop invariants must be provided by the user in order to bind the auxiliary variables generated from the automaton to the real variables of the program. Another class of invariants relates the auxiliary states stemming from the various branches of the automaton. All in all, 11 such invariants are needed. Two **`requires`** of `g` conveys the same information, as well as five **`behavior`**. Fig. 20 provides examples of such additional annotations. The first invariant states that `aorai_counter_0` is equal to the main loop counter `i` as long as the third branch (represented by `st12`) of the automaton is active. The second invariant indicates that `st12` and `st4` are active at the same time, except for the last step of the loop (since in that case the second branch can only succeed if the result of `g` is non-zero, that is when `st7` is active), and similarly for `st3` and `st11` (both active only when entering the loop). Once these additional annotations are in place, WP with Alt-ergo and CVC3 succeeds in proving the resulting 339 proof obligations (in about 30 seconds on a fairly standard laptop) generated from the specification, proving that the implementation is conforming to the initial automaton.

9. Mthread

Mthread is a plug-in that is able to handle concurrent code. It reuses Frama-C's existing abstract-interpretation based plug-in, Value (section 4), and extends it to obtain an over-approximation of the behavior of multi-threaded programs. Mthread analyses start by analyzing each thread independently with Value. The threads are discovered automatically and incrementally starting from the main thread, by monitoring Value's analysis for calls to thread creation primitives. By construction, this produces an over-approximation of the sequential behavior of the threads. However, this is not a correct over-approximation of the concurrent behavior of the program, as the interactions between the threads are not taken into account. Thus, Mthread confronts the various sequential approximations, to discover:

1. all the memory locations that are accessed by at least two threads performing read/write or write/write operations. Those locations are potentially subject to data races.
2. the messages that are sent by explicit message-passing, using primitives such as `msgsnd` in the POSIX API, and on which message queue those messages are sent;
3. the mutexes that are locked when the above multi-threaded events occur.

Using this information, Mthread re-analyzes all the threads that read shared memory, or received messages. This time, the plug-in takes into account the concurrent data that has been emitted during the first analysis of the threads. Messages sent on a queue q by a thread t are received each time a thread $t' \neq t$ reads on q. Shared zones are treated in two different ways:

- shared memory that is protected using mutexes in a consistent way is modeled precisely. That is, a value

```
#define N 5

int end2 = 0;
pthread_mutex_t  locks[N];
pthread_t        jobs[N];
msgqueue_t queue;

void aux (int l, int r, int mess) {
  pthread_mutex_lock(locks+l);
  pthread_mutex_lock(locks+r);
  if (random() && mess != 2) {
    char buf[2];
    buf[0]=mess;
    end2 = 1;
    msgsnd(queue, buf, 2);
  }
  pthread_mutex_unlock(locks+r);
  pthread_mutex_unlock(locks+l);
}

void * job( void * k ) {
  int p = (int) k ;
  int l = p>0 ? p-1 : N-1 ;
  int r = p<N-1 ? p+1 : 0 ;

  while(1)
    aux(l, r, p+1);
}

int main() {
  int i ;
  char end[2];
  end[0]=0;

  for(i=0;i<N;i++)
    pthread_mutex_init( &locks[i] , NULL);

  queuecreate(&queue, 5);

  //@ loop pragma UNROLL 5;
  for(i=0;i<N;i++)
    pthread_create( &jobs[i], NULL, &job, (void *) i );

  while(!(end[0] && __MTHREAD_SYNC(end2)))
    msgrcv(queue, 2, end);

  return 0;
}
```

Fig. 21. Example for Mthread plug-in

v written by a thread t is "seen" by all threads t' with $t' \neq t$ – provided that v is always protected by a certain mutex m.

- shared zones that are not protected by mutexes, or that are improperly protected (no mutex, or different mutexes on different accesses), are marked as volatile. This is the only correct approximation, although it is imprecise for *e.g.* lock-free algorithms.

If no new data is written or sent during this second analysis, Mthread has obtained an over-approximation of the concurrent behavior of the program. If this is not the case, the process above is repeated until a fixpoint is reached. Convergence is ensured because:

- there is only a finite set of possibly shared zones, as Mthread supposes either no dynamic allocation, or an implementation of `malloc` that creates finitely many base addresses (for example, by using summary variables);

```
philo.c:45:[mt] Receiving message on &queue, max size 2, stored in &end. Possible  values:
     From thread &jobs[0]: [0] IN {1}
                           [1] IN UNINITIALIZED
     From thread &jobs[2]: [0] IN {3}
                           [1] IN UNINITIALIZED
     From thread &jobs[3]: [0] IN {4}
                           [1] IN UNINITIALIZED
     From thread &jobs[4]: [0] IN {5}
                           [1] IN UNINITIALIZED
     msgrcv (philo.c:45) <- main (philo.c:30)
[...]
[mt] ***** Threads computed for iteration 4.
[mt] ***** Computing shared variables
[mt] _main_ reads  end2 IN {0; 1}  // philo.c:44
[mt] &jobs[0] writes  end2 IN {1}  // philo.c:14
[mt] &jobs[2] writes  end2 IN {1}  // philo.c:14
[mt] &jobs[3] writes  end2 IN {1}  // philo.c:14
[mt] &jobs[4] writes  end2 IN {1}  // philo.c:14
[mt] Possible read/write data races:
     end2:
       read by _main_ at philo.c:44, unprotected, // main (philo.c:30)
       write by &jobs[0] at philo.c:14, protected by &locks[1] &locks[4],
          // aux (philo.c:27) <- job (philo.c:21)
       write by &jobs[2] at philo.c:14, protected by &locks[1] &locks[3],
          // aux (philo.c:27) <- job (philo.c:21)
       write by &jobs[3] at philo.c:14, protected by &locks[2] &locks[4],
          // aux (philo.c:27) <- job (philo.c:21)
       write by &jobs[4] at philo.c:14, protected by &locks[0] &locks[3],
          // aux (philo.c:27) <- job (philo.c:21)
[mt] Write summary for &jobs[0]: end2 IN {1}
[mt] Write summary for &jobs[2]: end2 IN {1}
[mt] Write summary for &jobs[3]: end2 IN {1}
[mt] Write summary for &jobs[4]: end2 IN {1}
[mt] Mutexes for concurrent accesses:
     [end2] write protected by (?)&locks[0] (?)&locks[1] (?)&locks[2] (?)&locks[3]
       (?)&locks[4], read unprotected
[mt] ***** Shared variables computed
[mt] ******* Analysis performed, 4 iterations
```

Fig. 22. Output of Mthread on the example of Fig. 21

- if needed, the contents of emitted messages are over-approximated using the widening operation already present in the sequential analysis;
- the analysis assumes a finite number of distinct threads, that communicate through a finite number of concurrency constructs (mutexes, message queues, *etc.*).

At the end of the analysis, the plug-in emits three outputs:

1. a *concurrent control-graph*, which is similar to a very aggressive slicing of the program, the slicing criterion being the functions and instructions that are relevant to the concurrent part of the code;
2. all the memory zones that are shared amongst multiple threads, with detailed information on which functions access them and their contents;
3. the corresponding mutex information.

Example An example illustrating some features of Mthread is shown in Fig. 21. Roughly, the main function creates 5 different threads that act as the philosophers of the classical *dining philosophers* concurrency problem [Wik]. Each philosopher takes the mutex on its "left" and on its "right". Once the mutexes are acquired, 4 of the 5 threads write in the variable `end2`, and send a message through the first byte of the local buffer `buf` (the second byte being left uninitialized). The main thread loops until it receives something non-zero on the message queue `queue` and the variable `end2` is no longer equal to zero – which is possible only when information flows from the `job` threads to the main one.

Fig. 22 shows the output of Mthread once the analysis concludes, as well as the information inferred

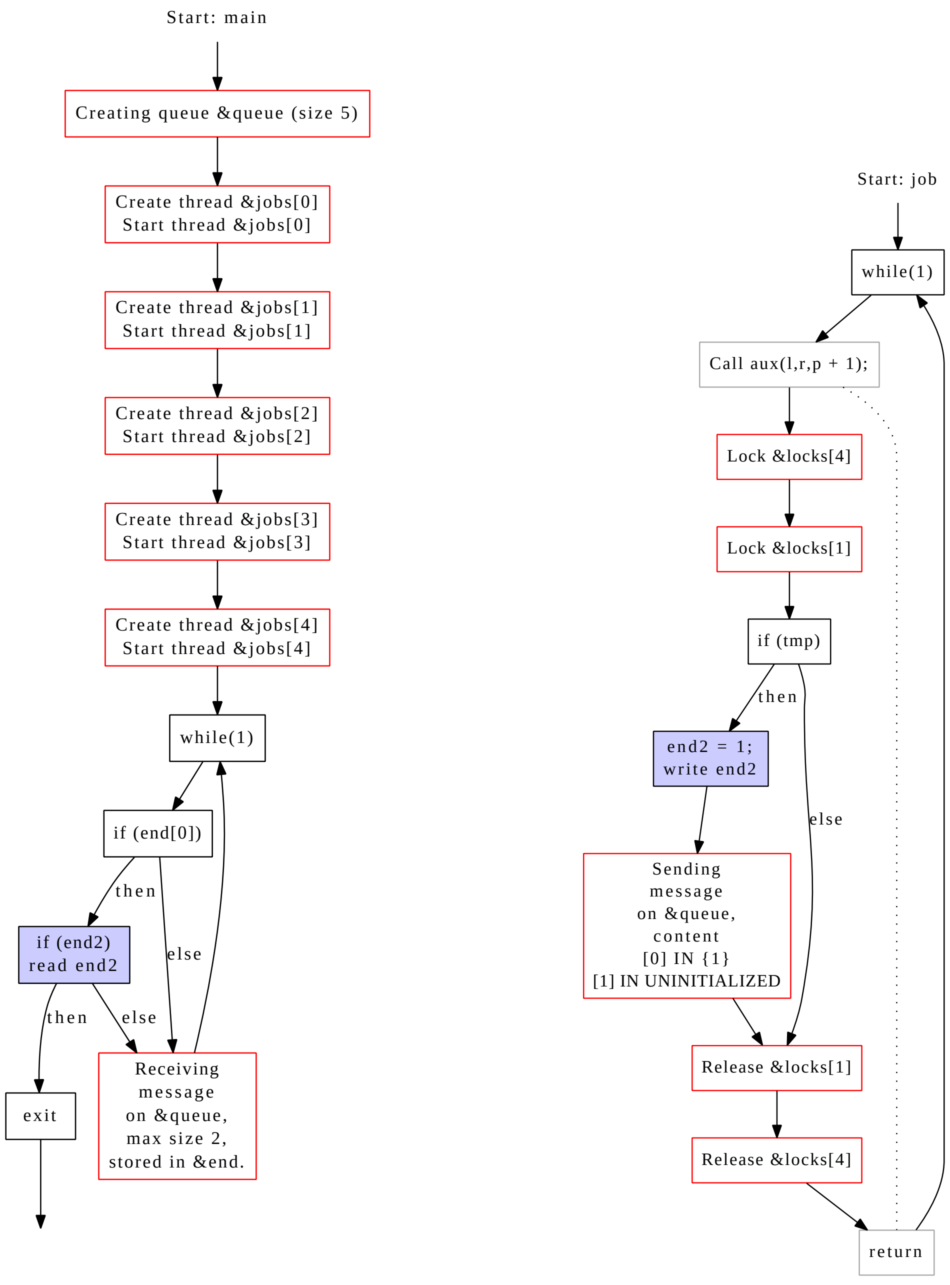


Fig. 23. Concurrent control-graphs for thread `main` and `&jobs[0]`

for the messages sent on `queue`. The contents sent by each thread are kept separate and precise (lines 1 to 9). We have also launched the analysis with an option that keeps the values of improperly protected variables precise. This is the case for `end2`, which is not always protected by the same mutex, and that would otherwise have been transformed into a volatile variable. Using this option requires the user to check that each shared variable is protected by at least one mutex or a call to the primitive `__MTHREAD_SYNC`, which is the case here. Since the plug-in lists all possibly racy accesses to the variable (lines 19 to 29), it is easy to check that this property is verified. In this example, all the threads write `1` inside `end2` (lines 15-18 and 30-34). The `main` thread can thus read `{0; 1}`, the `0` case corresponding to a scheduling in which the spawn threads have not run far enough before the main one reaches line 44. For each event, the analysis call-stack is displayed (*e.g.* line 23). Finally, for each access to a shared variables, the mutexes that protect the access are listed (lines *e.g.* 22 and 35-36 for a synthesis); a '?' indicates that the access may not have been protected by this mutex on at least one execution path and access.

Notice that the plug-in is able to handle the creation of multiple threads inside a loop. Here, the loop is syntactically unrolled using an appropriate directive, but this is not necessary. Unrolling the loop results in a more precise concurrent control-graph at the end of the analysis (Fig. 23); otherwise, all the threads would have been printed as created in a loop, without any ordering between them. Finally, all the concurrency constructs used in the programs – threads, mutexes, messages queues – are inferred by the plug-in automatically, including the names that are displayed. The user only needs to start the analysis on the main thread. This is done by recognizing, during the analysis, some primitives such as `pthread_create` or `pthread_mutex_init`. Although this example uses the POSIX API, other APIs are possible. Each one requires only writing some stubs functions that call Mthread basic primitives for thread or mutex creation, mutex locking, *etc.*.

Precision The approach followed by Mthread is sound by construction, as it produces a correct over-approximation of the behaviors of all the threads of the program. Yet it is also important to remain precise: this prompted the implementation of two important optimizations.

Context-sensitivity. It is crucial to have a context-sensitive analysis, meaning that a function f called by two functions g_1 and g_2 is analyzed separately in the two calling contexts. This is particularly important because embedded codes often encapsulates calls to concurrency primitives within their own functions, typically to simplify return codes.
Mthread is fully context-sensitive: as soon as a function is reached through two different paths (that is, two differing Value Analysis call-stacks), two different analyzes contexts are created by Mthread. This is consistent with what is done by Value, which proceeds by recursive inlining.

“Happened-before relation. Mthread tries to sequence events, so that if e_1 is guaranteed to always occur before e_2, then e_2 will not influence the analysis of e_1. For now, Mthread takes into account the fact that a thread has not yet been created, or has been canceled. In the following example, the zone v is detected as being *not* shared, as the possibly concurrent access in t_0 occurs before t_1 has been created.

t_0 :

```
v = 1;
a = v;      // Only possible value for a: v
create(t1);
```

t_1 :

```
v == 2;
```

In the future, we will try to enrich this “happened-before” relation with other sequencing operations. Condition variables, such as those used by the function `pthread_cond_signal`, would be very useful, as they are often used in concurrent C code to sequence events.

Similar tools The results of Mthread's mutex analysis is similar to the one of LockSmith [PFH11], and our approach is similar to theirs: identifying shared memory, propagating locked mutexes along the control-flow graph, *etc.* However, the methods differ significantly. LockSmith uses dedicated analyzes, that are specialized for this task. This ensures that their analysis is fast. On the contrary, Mthread reuses most of the machinery of Frama-C's Value Analysis, which usually gives more precise results, as well as a wealth of other information.

Mthread also share strong similarities with AstreeA [Min12]. Although both analysers have been developed independently, their approaches are the same. They are built on top of a sequential abstract interpreter (Value for Mthread, Astree [CCF$^+$05] for AstreeA), and automatically discover the interferences between the various threads. The sequential analyses are iterated until a fixpoint is reached. The main differences between the two analyzers stem from the more powerful domains available in Astree, as well as in some refinements. For example, AstreeA uses priorities in thread scheduling to deduce that one thread cannot be preempted by another, effectively protecting some shared memory. Conversely, the “happened-before” relation used by Mthread is not present in AstreeA.

10. Derived Analyses

10.1. Distilling Value

As explained in Section 4, Value automatically computes a per-statement over-approximation of the possible values for all memory locations. Other plug-ins can then take advantage of this result to provide more specific information. We present in this section such derived plug-ins that are included in the Frama-C distribution. In each case, the analysis is sound. Results from Value are used in particular to evaluate array indexes or resolve pointers, ensuring that e.g. pointer aliasing are always detected. These derived analyses also rely on the datastructures used by Value, that have been presented in Section 4.

Semantic constant folding. The simplest of the derived plug-ins performs code transformation based on the results of Value. It replaces each expression that is known to have a single value by this value, resulting in simpler code. Unlike basic syntactic constant folding, this analysis is semantic. This is particularly effective on code containing function pointers (that are often equal to a single function), or on functions that are called a single time. In particular, this plug-in is very useful as a front-end to other analyses. Furthermore, it inherits the precision of Value, in particular path- and context-sensitivity.

Inputs and outputs. Another plug-in computes for each function `f` approximations of its inputs and outputs. More precisely, the Inout plug-in can provide the following information.

- An over-approximation of the operational inputs of `f`, that is the memory locations that are read by `f` before having been written to.
- An under-approximation of the outputs of `f`, the locations where `f` is guaranteed to write to.
- An over-approximation of the outputs of `f`.
- An over-approximation of the imperative inputs of `f`, that is the memory locations that are read by `f`.

These sets of locations can be computed call-wise, for each distinct call to `f`, resulting in more precise results.

Functional dependencies. The From plug-in computes a relation between the outputs and the inputs of a function. For each possible output `x` of a function `f`, the plug-in will give an over-approximation of the sets of locations whose value at the beginning of the call to `f` might be used in computing the final value of `x`. If we go back to our example of Fig. 6, the result of launching `frama-c -deps value_example.c` is the following.

```
[from] Function main:
  S FROM S (and SELF)
  T[0..19] FROM S (and SELF)
  \result FROM S
```

This means that `S` and all the cells of `T` might be written during the execution, and that their value depends on (the initial value of) `S`. The `SELF` keyword indicates that the corresponding value might be left unchanged by the function, corresponding to the hypothetical situation in which the loop body would not have been executed.[6] Finally, we see that the value returned by `main` also depends solely on `S`.

As with Inout, From can compute separate information for each call to the function `f`, resulting in more precise results for the dependencies.

Program Dependency Graph (PDG). The PDG plug-in produces an intra-procedural graph that expresses the *data* and *control* dependencies between the instructions of a function [FOW87]. Namely, a statement s_2 has a data dependency over a statement s_1 when the computation performed in s_2 uses a value written in s_1. Similarly, if the execution of s_2 depends on the execution of s_1 (which is then typically an **`if`** statement, or more generally a statement with more than one successor), then there is a control dependency from s_1 upon s_2. In addition, PDG distinguishes two cases of data dependencies: plain data dependency, and *address* dependency, where the value written in s_1 is used at s_2 in the computation of an address that is dereferenced. Recalling again the example in Fig. 6, the statement `*p=S;` has dependencies on

[6]Taking this possibility into account is clearly sound, but also results in slightly imprecise results.

- `p=&T[0]` (address dependency)
- `p++` (address dependency)
- `S+=i` (data dependency)
- `while(i<20)` (control dependency)

While PDG is rarely requested directly by the user, it is the stepping stone for the various analyses described below. In addition, it makes itself heavy use of the From plug-in to compute the effects of a function call, illustrating sequential collaboration of plug-ins (see section 3.4).

Defs. The Scope plug-in provides various information about the dependencies of a memory location l at a given statement s. More precisely, it can display in the graphical user interface of Frama-C the following points:

- the statements that contribute to defining the value of l at s
- the statements where l has the same value than in s.
- For each statement s', the data d that is needed to compute the value of l at s.

This information is very useful when navigating through source code in order to understand where a particular value (in particular a value leading to an alarm) comes from.

Slicing & Impact. The Slicing plug-in returns a reduced program (also called a *slice* of the original program), equivalent to the original program according to a given *slicing criterion* [HRB88]. The Impact plug-in computes the values and statements impacted (directly or transitively) by the side effects of a given statement. In some sense, it is the dual of the slicing analysis.

Both Slicing and Impact accumulate a set of statements by walking through the PDG until the set is saturated for a given criterion. For the impact analysis, a statement s' is added when the outputs of an already selected statement s are required to evaluate s', or when s is a conditional that affects the execution of s'. The main difference between Slicing and Impact is the direction through which PDG edges are followed: forward in the case of Impact (to select the statements that are impacted by the statements already found), and backward for Slicing (to select the statements that influenced the statements already selected).

Possible slicing criteria include preserving a given statement, all calls to a function, a given alarm, the truth value of an ACSL assertion, the return value of a function, etc. The Sparecode plug-in is a special version of the Slicing plug-in, in which the criterion is to preserve the state of the program at the end of the main function under analysis. These plug-ins can in particular be used as front-ends to other, more specialized analyses, by removing parts of the code that are irrelevant to the property of interest.

10.2. Sante

The Sante plug-in (Static ANalysis and TEsting) [CKGJ12, CCK^{+}13] enhances static analysis results by testing. Given a C program p with a precondition, it detects possible runtime errors (currently divisions by zero and out-of-bound array accesses) in p and aims to classify them as real bugs or false alarms.

The Sante method contains three main steps illustrated in Fig. 24. Sante first calls Value to analyze p and to generate an alarm for each potentially unsafe statement. Next, Slicing is used to reduce the whole program with respect to one or several alarms. It produces one or several slices $p_1, p_2, \ldots, p_n$. Then, for each p_i, PathCrawler explores program paths and tries to generate test cases confirming the alarms present in p_i. If a test case activating an alarm is found, the alarm is confirmed and classified as a bug. If all feasible paths were explored for some slice p_i, all unconfirmed alarms in p_i are classified *safe,* i.e. they are in fact false alarms. If PathCrawler was used with a partial criterion (k-path), or stopped by a timeout before finishing the exploration of all paths of p_i, Sante cannot conclude and the statuses of unconfirmed alarms in p_i remain unknown.

The number of slices generated, hence the number of test generation sessions, is influenced by various Sante options. The *all* option generates a unique slice p_1 including all alarms of p, while the converse option *each* generates a slice for each alarm. Options *min* and *smart* take advantage of alarm dependencies (as computed by the dependency analysis) to slice related alarms together. The *smart* option improves *min* by iteratively refining the slices as long as one can hope to classify more alarms running PathCrawler on a smaller slice.

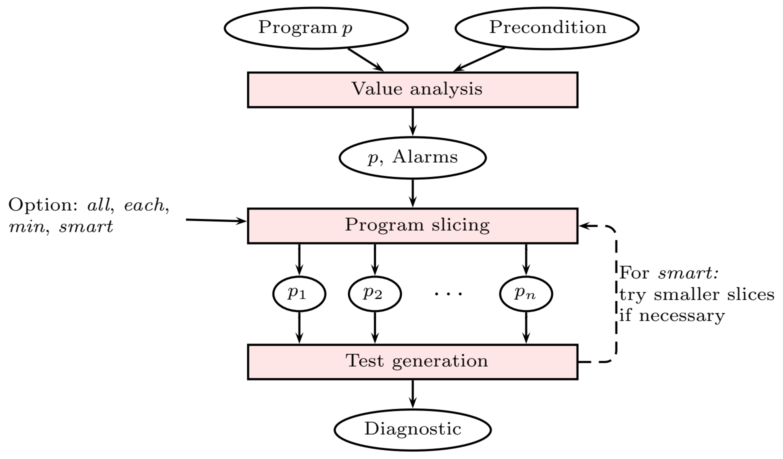


Fig. 24. Overview of the Sante method

Initial experiments [CKGJ12] on several real-life programs show that Sante can be in average 43% faster than test generation alone and leaves less unknown alarms. With the options *min* and *smart*, the number of remaining unclassified alarms decreases by 82% with respect to test generation alone, and by 86% with respect to Value alone.

The Sante plug-in is another good illustration of the benefits of the extensible plug-in oriented architecture adopted by Frama-C. Indeed, Sante relies on the Frama-C kernel and three main plug-ins Value (Section 4), Slicing (Section 10.1) and PathCrawler (Section 6) that may use in turn other plug-ins (e.g. Slicing uses PDG that uses Value). Thanks to the Frama-C design and services provided by the kernel (cf Sec. 3.1), the Sante plug-in does not need to generate and parse again a C program between Value and Slicing steps, or between Slicing and test generation steps. The program is parsed only once and its AST remains available for other Frama-C plug-ins. Similarly value analysis results are re-used by PDG and Slicing without being computed more than once. Moreover, the opportunity to work on several Frama-C projects in parallel allows an optimized implementation of slicing. Program slices can be constructed directly in AST form by Slicing and exploited in this form by PathCrawler.

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 Sante plug-in is only around 1500 lines of OCaml code.

10.3. Combining analyses

Sante is a quite big plug-in that builds upon existing analyses to provide more accurate results than each analysis could do alone. As Frama-C makes it easy for plug-ins to collaborate, it is also worthwhile to develop smaller plug-ins or even OCaml scripts that basically drive the main analysis plug-ins according to a specific usage.

An example is given by the yet-unreleased counter-examples plug-in. It is based on WP, and aims at providing test cases falsifying a given ACSL annotation when the related proof obligations are not discharged. For that, it takes advantage of the ability of some provers to give such a counter-example in the logic world. counter-examples then lifts this result back into the C world. An interesting use-case for the plug-in consists then in using it on the alarms emitted by Value, in an attempt to discriminate between false alarms and real bugs.

Furthermore, it is easy to combine analyses with an OCaml script for a given verification purpose. The

Frama-C wiki[7] gives an example of such a script that mixes constant propagation (Section 10.1), Aoraï (Section 8) and Value (Section 4) to find a sequence of calls to the API of a Linux driver that leads to an information leak.

11. Conclusion

This article attempts to distill a unified presentation of the Frama-C platform from a software analysis perspective. Frama-C answers the combined introductory challenges of scalability, interoperability, and soundness with a unique architecture and a robust set of analyzers. Its core set of tools and functionalities – about 150 OCaml kloc developed over the span of 7 years [CSB$^+$09] – has given rise to a flourishing ecosystem of software analyzers. In addition to industrial achievements and partnerships, including the birth of a startup[8], a community of users and developers has grown and strived.

Adoption in the academic world has stemmed from a variety of partnerships. The Jessie plug-in [MM12], developed at Inria, relies on a separation memory model but whose internal representation precludes its combination with other plug-ins. Verimag researchers have implemented a taint analysis [CMP10], producing explicit dependency chains pondered by risk quantifiers. Demay *et al.* generate security monitors based on fine-grained feedback from the Value plug-in [DTT09], while Assaf *et al.* generate security monitors for verifying non-interference properties by runtime assertion checking or by static analysis [ASTT13]. Berthomé *et al.* [BHKL10] propose a source-code model for verifying physical attacks on smart cards, and use Value to verify it. Bouajjani *et al.* [BDES11] automatically synthesize invariants of sequential programs with singly-linked lists. Ayache *et al.* [AARG12] automatically infer trustworthy ACSL assertions about the concrete worst-case execution cost of programs from so-called "cost annotations" generated by a custom C compiler. Jobredeaux *et al.* [JWF11] present a code generator that emits ACSL annotations so that its output can be verified by the Jessie plug-in of Frama-C. Finally, although the variety of objectives a static analyzer can have, and the variety of design choices for a given objective, make it difficult to benchmark static analyzers, Chatzieleftheriou and Katsaros [CK11] have produced one such comparison including Frama-C's Value plug-in.

On the industrial side of software analysis, many companies are evaluating and adopting Frama-C. Delmas *et al.* verify the compliance to domain-specific coding standards [DDLS10]; their plug-in is undergoing deployment and qualification. At the same company, Value is used to verify the control and data flows of a DAL C, 40-kloc ARINC 653 application [CDDM12]. Pariente and Ledinot [PL10] verify flight control system code using a combination of Frama-C plug-ins, including Value and Slicing. Their contribution includes a favorable evaluation of the cost-effectiveness of their adoption compared to traditional verification techniques. Bishop, Bloomfield and Cyra selected Frama-C *"after reviewing around twenty possible code analysis tools"* in order to implement a verification technique combining deductive verification and dynamic analysis [BBC13]. Adelard also developed a plug-in to analyze concurrent programs [Ade] in a complementary manner to Mthread, that trades off soundness for simplicity. Frama-C helped find bugs in various open-source components. Such issues have generally been acknowledged and promptly fixed by the authors, such as for the QuickLZ compression library[9] or the PolarSSL SSL implementation[10]. Frama-C was presented at the Static Analysis Tools Exposition (SATE V) organized by NIST, in the Ockham category dedicated to sound (i.e. correct) analyzers [Bla14]. Finally two ongoing projects further illustrate the intensity of current industrial analysis efforts with Frama-C:

- Yakobowski *et al* use Value in collaboration with WP to check the absence of runtime errors in a 50 kloc instrumentation and control (I&C) nuclear code [CHK$^+$12]. The conclusion of this case study is that *"the selectivity and absolute number of alarms obtained can be [...] compared advantageously to the state of the art in static analysis"*. Furthermore, the use of WP allowed to verify that 9 of the alarms emitted by Value were indeed false alarms, with only a very limited annotation effort, demonstrating the interest of being able to combine various analyses in a single tool.
- The TrustInSoft startup develops a full fledged industrial version of Frama-C pre-installed with proprietary

[7]https://bts.frama-c.com/dokuwiki/doku.php?id=mantis:frama-c:aorai-security
[8]See http://www.trust-in-soft.com.
[9]http://www.quicklz.com/changelog.html
[10]https://github.com/polarssl/polarssl/commit/e2f8ff67972a85bfc748b16b7128fee08c4dd4c0

extensions designed to facilitate the analysis of complex programs – including multi-threaded code, low-level code that uses volatile memory locations, or applications that make heavy use of the C standard library. TrustInSoft also provides formal verification "kits" for open-source software components, and in particular the PolarSSL library[11].

Through all of these successes and over the past few years, Frama-C has demonstrated its adoptability for a wide variety of purposes within a diverse community of researchers and engineers. The growth of this community, fostered by a number of active communication channels[12] should be interpreted as a testimony to the health of the software analysis community, and good omens for its future.

References

[AARG12] N. Ayache, R. Amadio, and Y. Régis-Gianas. Certifying and reasoning on cost annotations in C programs. In *17th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2012)*, 2012.

[Ade] Adelard LLP. Simple Concurrency Analysis Plugin for Frama-C. `https://bitbucket.org/adelard/simple-concurrency/`.

[ASTT13] M. Assaf, J. Signoles, E. Totel, and F. Tronel. Program transformation for non-interference verification on programs with pointers. In *the 28th IFIP TC-11 International Information Security and Privacy Conference (SEC 2013)*, pages 231–244. Springer, 2013.

[BBC13] Peter Bishop, Robin Bloomfield, and Lukasz Cyra. Combining Testing and Proof to Gain High Assurance in Software: a Case Study. In *Proceedings of IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2013.

[BC11] R. Bonichon and P. Cuoq. A Mergeable Interval Map. *Studia Informatica Universalis*, 9(1):5–37, 2011.

[BCC$^+$05] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[BCD$^+$06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proceedings of 4th International Symposium on Formal Methods Components and Objects (FMCO 2005)*, volume 4111 of *LNCS*. Springer, 2006.

[BDES11] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*, pages 578–589. ACM, 2011.

[BDH$^+$09] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams. Automating structural testing of C programs: Experience with PathCrawler. In *the 4th International Workshop on Automation of Software Test (AST 2009)*, pages 70–78. IEEE Computer Society, 2009.

[BFH$^+$13] P. Baudin, J.-C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, v1.6*, April 2013. `http://frama-c.com/acsl.html`.

[BH11] S. Bardin and P. Herrmann. OSMOSE: automatic structural testing of executables. *Software Testing, Verification and Reliability*, 21(1):29–54, 2011.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST: Applications to software engineering. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.

[BHKL10] P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky, and J.-F. Lalande. 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)*, pages 1–12. ACM, 2010.

[BHV11] S. Bardin, P. Herrmann, and F. Védrine. Refinement-based CFG reconstruction from unstructured programs. In *the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*, volume 6538 of *LNCS*, pages 54–69. Springer, 2011.

[Bla14] Paul E. Black. SATE V Ockham Sound Analysis Criteria. `http://samate.nist.gov/SATE5Workshop.html`, March 2014.

[BNR$^+$10] N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons, S. Tetali, and A. V. Thakur. Proofs from tests. *IEEE Trans. Software Eng.*, 36(4):495–508, 2010.

[Bor00] R. Bornat. Proving pointer programs in Hoare logic. In *the 5th International Conference on Mathematics of Program Construction (MPC 2000)*, volume 1837 of *LNCS*. Springer, 2000.

[Bur72] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[C$^+$] S. Conchon et al. The Alt-Ergo Automated Theorem Prover. `http://alt-ergo.lri.fr`.

[CC77] P. Cousot and R. Cousot. 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)*, pages 238–252, 1977.

[CCF$^+$05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *the 14th European Symposium on Programming (ESOP 2005), part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2005)*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.

[11]`http://trust-in-soft.com/polarssl-verification-kit/`

[12]See `http://frama-c.com/support.html`.

[CCK$^+$13] O. Chebaro, P. Cuoq, N. Kosmatov, B. Marre, A. Pacalet, N. Williams, and B. Yakobowski. Behind the scenes in SANTE: a combination of static and dynamic analyses. *Automated Software Engineering*, 2013. Published online.

[CD08] P. Cuoq and D. Doligez. Hashconsing in an incrementally garbage-collected system: a story of weak pointers and hashconsing in OCaml 3.10.2. In *Proceedings of the ACM SigPlan ML Workshop*, pages 13–22, September 2008.

[CDDM12] P. Cuoq, D. Delmas, S. Duprat, and V. Moya Lamiel. Fan-C, a Frama-C plug-in for data flow verification. In *the Embedded Real-Time Software and Systems Congress (ERTS^2 2012)*, 2012.

[CDS11] P. Cuoq, D. Doligez, and J. Signoles. Lightweight typed customizable unmarshaling. In *ACM SIGPLAN Workshop on ML*. ACM, 2011.

[CHK$^+$12] P. Cuoq, P. Hilsenkopf, F. Kirchner, S. Labbé, N. Thuy, and B. Yakobowski. 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)*, July 2012.

[CHOS13] Simon Cruanes, Grégoire Hamon, Sam Owre, and Natarajan Shankar. Tool Integration with the Evidential Tool Bus. In *Proceedings of Verification, Model-Checking and Abstract Interpretation (VMCAI)*, volume 7737 of *LNCS*, pages 275–294, 2013.

[CK04] David R. Cok and Joseph R. Kiniry. ESC/Java2: uniting ESC/Java and JML. In *the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *LNCS*, pages 108–128. Springer, 2004.

[CK11] G. Chatzieleftheriou and P. Katsaros. Test-driving static analysis tools in search of C code vulnerabilities. In *COMPSAC Workshops*, pages 96–103. IEEE Computer Society, 2011.

[CKGJ12] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *the ACM Symposium on Applied Computing (SAC 2012)*, pages 1284–1291. ACM, 2012.

[CKM12] Cyrille Comar, Johannes Kanig, and Yannick Moy. Integrating formal program verification with testing. In *Proc. ERTS*, 2012.

[Cla] Clang Static Analyzer. `http://clang-analyzer.llvm.org/`.

[CMP10] D. Ceara, L. Mounier, and M.-L. Potet. Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences. In *the 3rd International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2010)*, pages 371–380, 2010.

[Coq11] Coq Development Team. *The Coq Proof Assistant Reference Manual*, v8.3 edition, 2011. `http://coq.inria.fr/`.

[CR06] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006.

[CS04] Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software — Practice & Experience*, 34(11):1025–1050, September 2004.

[CS06] Christoph Csallner and Yannis Smaragdakis. Dynamically discovering likely interface invariants. In *the 28th ACM/IEEE International Conference on Software Engineering (ICSE 2006), Emerging Results Track*, pages 861–864. ACM, May 2006.

[CS12] L. Correnson and J. Signoles. Combining analyses for C program verification. In *the 17th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2012)*, August 2012.

[CSB$^+$09] P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti. Experience report: OCaml for an industrial-strength static analysis framework. In *the 14th ACM SIGPLAN International Conference on Functional programming (ICFP 2009)*, pages 281–286. ACM, 2009.

[CYP13] P. Cuoq, B. Yakobowski, and V. Prevosto. *Frama-C's value analysis plug-in*, fluorine-20130601 edition, June 2013. `http://frama-c.com/download/frama-c-value-analysis.pdf`.

[DDLS10] D. Delmas, S. Duprat, V. Moya Lamiel, and J. Signoles. Taster, a Frama-C plug-in to encode Coding Standards. In *the Embedded Real-Time Software and Systems Congress (ERTS^2 2010)*, 2010.

[DEL$^+$14] Claire Dross, Pavlos Efstathopoulos, David Lesens, David Mentré, and Yannick Moy. Rail, space, security: Three case studies for spark 2014. In *Proc. ERTS*, 2014.

[Dij68] E. W. Dijkstra. A constructive approach to program correctness. *BIT Numerical Mathematics*, Springer, 1968.

[DJP10] D. Demange, T. Jensen, and D. Pichardie. A provably correct stackless intermediate representation for java bytecode. In *the 8th Asian Symposium on Programming Languages and Systems (APLAS 2010)*, volume 6461 of *LNCS*, pages 97–113. Springer, 2010.

[DKS13] M. Delahaye, N. Kosmatov, and J. Signoles. Common specification language for static and dynamic analysis of C programs. In *the 28th Annual ACM Symposium on Applied Computing (SAC)*, pages 1230–1235. ACM, March 2013.

[DMS$^+$09] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE Companion*, pages 429–430. IEEE Computer Society, 2009.

[DTT09] J.-C. Demay, E. Totel, and F. Tronel. SIDAN: a tool dedicated to software instrumentation for detecting attacks on non-control-data. In *CRiSIS*, October 2009.

[EMN12] F. Elberzhager, J. Münch, and V. Tran Ngoc Nha. A systematic mapping study on the combination of static and dynamic quality assurance techniques. *Information & Software Technology*, 54(1):1–15, 2012.

[EPG$^+$07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, 2007.

[Fil00] Jean-Christophe Filliâtre. Hash consing in an ML framework. Research Report 1368, LRI, Université Paris Sud, September 2000.

[Fil03] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[Flo67] R. W. Floyd. Assigning meanings to programs. *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, 19, 1967.

[FM07] J.-C. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *CAV*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[FP13] J.-C. Filliâtre and A. Paskevich. Why3 – where programs meet provers. In *the 22nd European Symposium on Programming (ESOP 2013)*, volume 7792 of *LNCS*. Springer, 2013.

[GdHN$^+$08] Patrice Godefroid, Jonathan de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, Nikolai Tillmann, and Michael Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5):30–37, 2008.

[GGJK08] A. Giorgetti, J. Groslambert, J. Julliand, and O. Kouchnarenko. Verification of class liveness properties with Java Modeling Language. *IET Software*, 2(6), 2008.

[GHK$^+$06] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: a new algorithm for property checking. In *the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2006)*, pages 117–127. ACM, 2006.

[GMP] Gmp: Gnu multiple precision arithmetic library. `https://gmplib.org/`.

[GO01] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *the 13th International Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 53–65. Springer, 2001.

[Gra91] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT*, volume 493 of *LNCS*, pages 169–192. Springer, 1991.

[GS09] J. Groslambert and N. Stouls. Vérification de propriétés LTL sur des programmes C par génération d'annotations. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 2009)*, 2009. In French.

[HJV00] N. Heintze, J. Jaffar, and R. Voicu. A framework for combining analysis and verification. In *the 27th Symposium on Principles of Programming Languages (POPL 2000)*, 2000.

[HMM12] P. Herms, C. Marché, and B. Monate. A certified multi-prover verification condition generator. In *the 4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2012)*, volume 7152 of *LNCS*, pages 2–17. Springer, 2012.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1988)*, volume 23–7, pages 35–46, 1988.

[HS13] P. Herrmann and J. Signoles. *Annotation generation: Frama-C's RTE plug-in*, April 2013. `http://frama-c.com/download/frama-c-rte-manual.pdf`.

[IEE08] IEEE Std 754-2008. IEEE standard for floating-point arithmetic. Technical report, 2008. `http://dx.doi.org/10.1109/IEEESTD.2008.4610935`.

[ISO07] ISO/IEC JTC1/SC22/WG14. *9899:TC3: Programming Languages—C*, 2007. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.

[JWF11] Romain Jobredeaux, Timothy E. Wang, and Eric M. Feron. Autocoding Control Software with Proofs I: Annotation Translation. In *Proceedings of the IEEE/AIAA Digital Avionics Systems Conference (DASC)*, 2011.

[KCC$^+$14] Johannes Kanig, Rod Chapman, Cyrille Comar, Jerome Guitton, Yannick Moy, and Emyr Rees. Explicit assumptions - a prenup for marrying static and dynamic program verification. In *Proc. TAP*, 2014. To appear.

[Kos] N. Kosmatov. Online version of PathCrawler. `http://pathcrawler-online.com/`.

[Kos10] N. Kosmatov. *Artificial Intelligence Applications for Improved Software Engineering Development: New Prospects*, chapter XI: Constraint-Based Techniques for Software Testing. IGI Global, 2010.

[KPS13] N. Kosmatov, G. Petiot, and J. Signoles. An optimized memory monitoring for runtime assertion checking of C programs. In *the 4th International Conference on Runtime Verification (RV 2013)*, volume 8174 of *LNCS*, pages 167–182. Springer, 2013.

[KS13] N. Kosmatov and J. Signoles. A lesson on runtime assertion checking with Frama-C. In *the 4th International Conference on Runtime Verification (RV 2013)*, volume 8174 of *LNCS*, pages 386–399. Springer, 2013.

[LB08] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[LDF$^+$13] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.01.* INRIA, 2013. `http://caml.inria.fr/pub/docs/manual-ocaml-4.01/`.

[Lei08] K. R. M. Leino. *This is Boogie 2.* Microsoft Research, 2008.

[MA00] B. Marre and A. Arnould. Test sequences generation from Lustre descriptions: GATeL. In *the 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 229–237. IEEE Computer Society, 2000.

[Mat] MathWorks. Polyspace. `http://www.mathworks.com/products/polyspace`.

[Mey97] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1997.

[Min12] Antoine Miné. Static analysis of run-time errors in embedded real-time parallel c programs. *Logical Methods in Computer Science*, 8(1), 2012.

[MM12] C. Marché and Y. Moy. *The Jessie plug-in for Deduction Verification in Frama-C, version 2.30.* INRIA, 2012. `http://krakatoa.lri.fr/jessie.pdf`.

[MR05] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.

[NMRW02] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *the International Conference on Compiler Construction (CC 2002)*, volume 2304 of *LNCS*, pages 213–228. Springer, 2002.

[PFH11] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Practical static race detection for c. *ACM Trans. Program. Lang. Syst.*, 33(1):3, 2011.

[PL10] D. Pariente and E. Ledinot. Formal verification of industrial C code using Frama-C: a case study. In *FoVeOOS*, 2010.

[Pnu77] A. Pnueli. The temporal logic of programs. In *the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 46–57. IEEE Computer Society, 1977.

[RSB$^+$99] F. Randimbivololona, J. Souyris, P. Baudin, A. Pacalet, J. Raguideau, and D. Schoen. Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach. In *the Wold Congress on Formal Methods in the Development of Computing Systems (FM 1999)*, volume 1709 of *LNCS*, pages 1798–1815. Springer, 1999.

[Rus05] John Rushby. An Evidential Tool Bus. In *Formal Methods and Software Engineering, ICFEM*, volume 3785 of *LNCS*, 2005.

[SC07] Yannis Smaragdakis and Christoph Csallner. Combining static and dynamic reasoning for bug detection. In *the First International Conference on Tests and Proofs (TAP 2007)*, volume 4454 of *LNCS*, pages 1–16. Springer, 2007.

[SCP13] J. Signoles, L. Correnson, and V. Prevosto. *Frama-C Plug-in Development Guide*, April 2013. `http://frama-c.com/download/plug-in-developer.pdf`.

[Sig09] J. Signoles. Foncteurs impératifs et composés: la notion de projet dans Frama-C. In *JFLA*, volume 7.2 of *Studia Informatica Universalis*, 2009. In French.

[Sig13] J. Signoles. *E-ACSL: Executable ANSI/ISO C Specification Language. Version 1.7*, May 2013. `http://frama-c.com/download/e-acsl/e-acsl.pdf`.

[Sig14] J. Signoles. Comment un chameau peut-il écrire un journal? In *JFLA*, 2014. In French.

[SP11] N. Stouls and V. Prevosto. *Aoraï plug-in tutorial, version Nitrogen-20111001*, October 2011. `http://frama-c.com/download/frama-c-aorai-manual.pdf`.

[SS11] J. Schimpf and K. Shen. ECLiPSe - from LP to CLP. *Theory and Practice of Logic Programming*, 12(1-2):127–156, 2011.

[TFNM11] J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *the 9th International Conference on Software Engineering and Formal Methods (SEFM 2011)*, 2011.

[Wik] Wikipedia. Dining philosophers problem. `https://en.wikipedia.org/wiki/Dining_philosophers_problem`.

[WMMR05] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In *the 5th European Dependable Computing Conference on Dependable Computing (EDCC 2005)*, volume 3463 of *LNCS*, pages 281–292. Springer, 2005.