Frama-C

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prévosto, Julien Signoles, Boris Yakobowski

▶ **To cite this version:**

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prévosto, Julien Signoles, et al.. Frama-C. Lecture Notes in Computer Science, 2012, 7504, pp.233-247. 10.1007/978-3-642-33826-7_16 . cea-04469477

HAL Id: cea-04469477
https://cea.hal.science/cea-04469477v1

Submitted on 20 Feb 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Frama-C

A Software Analysis Perspective*

Pascal Cuoq, Florent Kirchner, Nikolaï Kosmatov, Virgile Prevosto,
Julien Signoles, and Boris Yakobowski

with Patrick Baudin, Richard Bonichon, Bernard Botella, Loïc Correnson,
Zaynah Dargaye, Philippe Herrmann, Benjamin Monate, Yannick Moy,
Anne Pacalet, Armand Puccetti, Muriel Roger, and Nicky Williams

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

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 synthetic 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 [18,41,21,47,52] – and met with growing industrial interest. The ensuing push for mainstream diffusion of software analysis techniques has raised several challenges. Chief among them are: *a.* their scalability, *b.* their interoperability, and *c.* the soundness of their results.

Point *a* is predictably important from the point of vue of adoptability. Scaling to large problems is a prerequisite for the industrial diffusion of software analysis and verification techniques. It also represents a mean to better understand 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.

*This work was partly supported by ANR U3CAT and Veridyc, and FUI9 Hi-Lite projects

Point b – interoperability – enables the design of elaborate program analyses. Consider indeed the interplay between program analyses and transformations [23], the complementarity of forward and backward analyses [2], or the precision gain afforded when combining static and dynamic approaches [4]. Yet running multiple source code analyses and synthesizing their results in a coherent fashion requires carefully thought-out mechanisms.

Point c – soundness – is a strong differentiator for formal approaches. By using tools that *over-approximate* all program behaviors, industrial 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 three fundamental plug-ins based on abstract interpretation, deductive verification, and concolic testing; and a series of derived plug-ins which build elaborate analyses upon the former. The extensibility of the overall platform, and its open-source licensing, have fostered the development of an ecosystem of independent third-party plug-ins [12,19,22,24,43,45]. This article is intended as a foundational reference to the platform, its three main analyses, and its most salient derived plug-ins.

2 The Platform Kernel

2.1 Architecture

Figure 1 shows a functional view of the Frama-C architecture. Frama-C is based on CIL [44], a front-end for C that parses ISO C99 programs into their normalized representation: loop constructs are given a single form, expressions have no side-effects, etc. Frama-C extends CIL to support dedicated source code annotations expressed in ACSL (see § 2.2). This modified CIL front-end produces the C + ACSL AST, an abstract view of the program shared among all analyzers.

The Frama-C kernel provides several services, helping plug-in development [50] and providing convenient features to the 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 be replayed afterwards, a feature of interest in debugging and qualification contexts.
- A project system, presented in § 2.3, isolates unrelated program representations, and guarantees the integrity of their analyses.
- Consistency mechanisms control the collaboration between analyzers (§ 2.4).

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

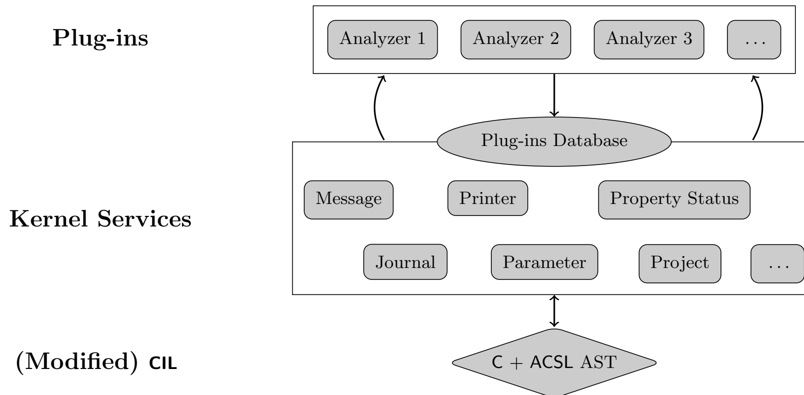

Fig. 1. Frama-C's Functional View

existing ones. Any plug-in can register new services in a *plug-ins database* stored in the kernel, making these services available to all plug-ins.

2.2 ACSL

Functional properties of C programs can be expressed within Frama-C as ACSL annotations [3]. ACSL is a formal specification language inspired by Java's JML [10], both being based on the notion of function contract introduced by Eiffel [42]. In effect, the specification 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.

For instance, Fig. 2 provides a specification for a `swap` function. 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 predicates: ACSL features various 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. The **assigns** clause states that only the locations pointed

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

Fig. 2. Example of ACSL specification

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*.

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 must be preserved across any number of loop steps). 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. Plug-ins can provide a validity status to any ACSL property and generate ACSL annotations. This allows ACSL annotations to play an important role in the communication between plug-ins, as explicited in § 2.4.

2.3 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 [49]: 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 like Slicing (§ 6.1) or Aoraï (§ 6.2) is used. The result of the transformation is a fresh AST that coexists with the original, making backtracking and comparisons easy. Another use of projects is to process the same program in different ways – for instance with different analysis parameters.

2.4 Analyzers Collaboration

In Frama-C, analyzers can collaborate 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-ins database stored by the Frama-C kernel. Refer to § 6.1 for a comprehensive 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 (§ 3) is unable to ensure the validity of a pointer p, it emits an unproved ACSL annotation `assert \valid(p)`. In accordance with the underlying blocking semantics, it assumes that `p` is valid from this program point onwards. The WP plug-in (§ 4) may later be used to lift this hypothesis. The kernel automatically computes the validity status of each program property from the information provided by all analyzers and ensures the consistency of the entire verification process [16]: "*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's semantics".

3 Fundamental Analysis: Abstract Interpretation

The Value plugin (short for Value Analysis) is a forward dataflow analysis based on the principles of *abstract interpretation* [17]. Abstract interpretation links a *concrete* semantics, typically the set of all possible executions of a program, 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.

Value, and abstract interpreters in general, proceed by symbolic execution of the program, translating all operations into the abstract semantics. Termination of looping constructs is ensured by *widening* operations. For function calls, Value proceeds essentially by recursive inlining of the function (recursive functions are currently not handled). This ensures that the analysis is fully context-sensitive. If needed, the user can abstract overly complex functions by an ACSL contract, verified by hand or discharged with another analysis.

Abstract domains The domains currently used by Value to represent the abstract semantics are described below.

Integer computations. Small sets of integers are represented as sets, whereas large sets are represented as intervals with congruence information [30]. For instance, $x \in [3..255], 3\%4$ means that x is such that $3 \leq x \leq 255$ and $x \equiv 3 \bmod 4$.

Floating-point computations. The results of floating-point computations are represented as IEEE 754 [34] double-precision finite intervals. Operations on single-precision floats are stored as doubles, but are rounded as necessary. Obtaining infinities or NaN is treated as undesirable errors.

Pointers and memory. To verify that invalid (e.g. out-of-bounds) array/pointer accesses cannot occur in the target program, Value assumes that the program does not purposely use buffer overflows to access neighboring variables [35, §6.5.6:8]. Abstract representation of memory states in a C program 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.

Memory representation is untyped. It is thus straightforward to handle unions and heterogeneous pointer conversions during abstract interpretation. The abstract memory state maps each base address to a representation of a chunk of linear memory. Each such object itself maps ranges of bits to values. Given an array of 32-bit integers `t`, and reading from `*(((char*)t) + 5)`, the analyzer determines that the relevant abstract value is to be found between bits `[40..47]` of the value bound to `&t`. Using bit as unit, instead of byte, allows to handle bit-fields [35, §6.2.6.1:4] by fixing a layout strategy (the C standard itself does not specify bit-fields layout, but then again, no more than any kind of data).

Finally, 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 their scope [35, §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.

Propagation of unjoined states Value's domains are non-relational. Instead, the datastructures representing the abstract semantics have been heavily optimized for speed and reduced memory footprint, to allow the independent propagation of k multiple states. Typically, the user can set a high enough k, so that finite loops are entirely unrolled. Successive conditionals are also handled more precisely. This parameter can be adjusted on a per-function basis, alleviating for a large part the need for relational domains, by implicitly encoding relations in the disjunction of abstract states.

Alarms Each time a statement is analyzed, any operation that can lead to an 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.

If 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. This 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 typically results in fewer false alarms, but lenghten analysis time. Upon emitting an alarm, the analyzer reduces the propagated state accordingly, and proceeds onwards.

4 Fundamental Analysis: Deductive Verification

The WP plug-in is named after the *Weakest Precondition* calculus, a technique used to prove program properties initiated by Hoare [32], Floyd [28] and Dijkstra [25]. Recent tools implement this technique efficiently, for instance Boogie [38] and Why [27]. Jessie [39], 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 parametrization 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 § 2.4.

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 quickly become intractable. Thus, various refinements have been proposed, in particular by Bornat [7], building upon earlier work by Burstall [11]. 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 three main models:

- The most abstract model, `hoare`, roughly corresponds to Caveat's model [47]. 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 `store`. It is a classical Burstall-Bornat model, that supports pointer aliasing, but neither cast nor union types.
- The `runtime` model is designed for code that perform low-level memory operations. In this model, the memory is seen as a single array of **`char`**, so that most C operations can be taken into account, at the expense of the complexity of the generated VC.

As a refinement, `store` and `runtime` can avoid converting assignments into array updates when the code falls in the subset supported by `hoare`. In particular, variables whose addresses are not taken and plain references – pointers that are neither assigned nor used in a pointer arithmetic operation – are translated as standard Hoare logic variables. This way, the overhead of other models with respect to `hoare` is kept to the places where it is really needed.

Once a VC has been generated, it must be discharged. WP natively supports two theorem provers: the automated SMT solver Alt-ergo [6], and the Coq proof assistant [15]. Other automated provers can also be used through the multi-prover backend of Why. Advantages of using a dedicated back-end rather than relying completely on Why are twofold. First, it removes a dependency over an external tool, meaning that for verification of critical software, there is one

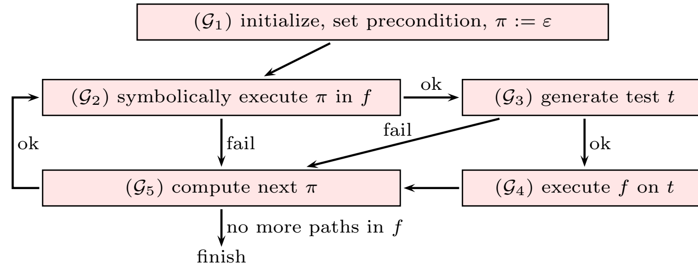

Fig. 3. The PathCrawler test generation method

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), that are not supported by Why yet.

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 § 2.4 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 `store` but not under `runtime`, 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 targetted annotations (e.g. those yet unverified by other means in Frama-C, *cf* § 2.4) as well as to generate all the VC related to a C function.

5 Fundamental Analysis: Concolic Testing

Given a C program p under test and a precondition restricting its inputs, the PathCrawler plug-in 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 [52,8] 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 of each instruction in p. The next step, illustrated by Fig. 3, 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 [48] and uses Constraint Logic Programming. Given a *path prefix* π, i.e. a partial program path in p, the main idea [37] 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.

PathCrawler uses Colibri, a specialized constraint solving library developed at CEA LIST and shared with other testing tools GATeL [40] and OSMOSE [1]. 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. PathCrawler is a proprietary plug-in, also available in the form of a freely accessible test-case generation web service [36].

6 Derived Analyses

6.1 Distilling values

The outputs of the Value plugin are twofold. In addition to emitting alarms for statements it cannot guarantee are safe (§3), Value automatically computes a per-statement over-approximation of the possible values for all memory locations. The derived analyses below reuse those synthetic results. In each case, the

analysis is sound. Value's results are used to evaluate array indexes or resolve pointers, ensuring that e.g. pointer aliasing are always detected.

Outputs: over-approximates the locations a function may write to.

Operational inputs: over-approximates the locations whose initial values are used by the function.

Functional dependencies: computes a relation between outputs and inputs of a function; `x FROM y, t[1] (and SELF)` means that output `x` is either unchanged (`SELF`), or that its new value can be computed exclusively from inputs `y` and `t[1]`.

Program Dependency Graph (PDG): produces an intra-procedural graph that expresses the *data* and *control* dependencies between the instructions of a function, used as a stepping stone for various analyzes [26].

Defs: over-approximates which statements define a given memory location.

Impact: computes the values and statements impacted (directly or transitively) by the side effects of a chosen statement.

Slicing: returns a reduced program (a *slice*), equivalent to the original program according to a given *slicing criterion* [33]. Possible criteria include preserving a given statement, all calls to a function, a given alarm, etc.

Analyses such as Defs or Impact make compelling code understanding tools, as they express in a very concise way the relationships between various parts of a program. Slicing goes one step further: while it is essentially dual to the impact analysis, it also builds reduced, self-contained programs, that can be re-analyzed independently. Those three analyses are fully inter-procedural.

The analyses above illustrate sequential collaboration (§ 2.4). PDG makes heavy use of Functional dependencies, while Defs, Impact and Slicing leverage the information given by PDG. Some of those analyses can optionally compute *callwise* versions of their results, yielding one result per syntactic call, instead of one result per function. The improved precision automatically benefits the derived analyses. All the results are stored by the Frama-C kernel, and can be reused without being recomputed.

6.2 Annotation Generator

The Aoraï plug-in [51,31] 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 the call and return of 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,[46]) formula or in the form of an automaton. In the former case, Aoraï uses ltl2ba [29] to obtain an equivalent Büchi automaton.

Given such an automaton, Aoraï provides ACSL specifications for each function `f` in the original C code. This instrumentation is summarized in Fig. 4. It consists of two main parts: prototypes whose specification takes care of performing

```
1   /*@ behavior transition_1: assumes aorai_state_S_0 == 1 && condition;
2       ensures aorai_state_S_next == 1; ... */
3   void advance_automaton_call_f(int x);
4
5   /*@ requires aorai_state_S_0 == 1 || aorai_state_S_1 == 1 || ...;
6       requires aorai_state == S_0 ==> has_possible_transition_S0; ...
7       ensures aorai_state_S_2 == 1 || aorai_state_S_3 == 1 || ...;
8       ensures \old(aorai_state_S_0 == 1) ==> aorai_state_S_2 == 1 || ...; ...
9       ensures aorai_state_S_2 == 1 ==> program_state_when_in_S_2; ... */
10   int f(int x) {
11     advance_automaton_call_f(x);
12     // Body of f
13     advance_automaton_return_f(result);
14     return result; }
```

Fig. 4. Aoraï's instrumentation

transitions for the corresponding atomic event (call or return from `f`), and the specification of `f` itself. As the automaton is not deterministic in general, Aoraï uses a set of boolean variables to represent possible active states (`aorai_state_S_*`). Functions advancing the states of the automaton provide, for each of those variables, a complete set of **`behavior`**s indicating when they are set to `1` or `0`.

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 rough static analysis made by Aoraï beforehand. In addition, 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. 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).

Annotation generation is geared towards the use of deductive verification plugins 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. Likewise Aoraï also generates loop invariants for the same purpose. However it does not preclude the use of the Value plug-in to validate its specification, and therefore attempts to generate annotations that fit in the subset of ACSL that is understood by Value.

6.3 SANTE

The Sante Frama-C plug-in (Static ANalysis and TEsting) [14] 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. 5. 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

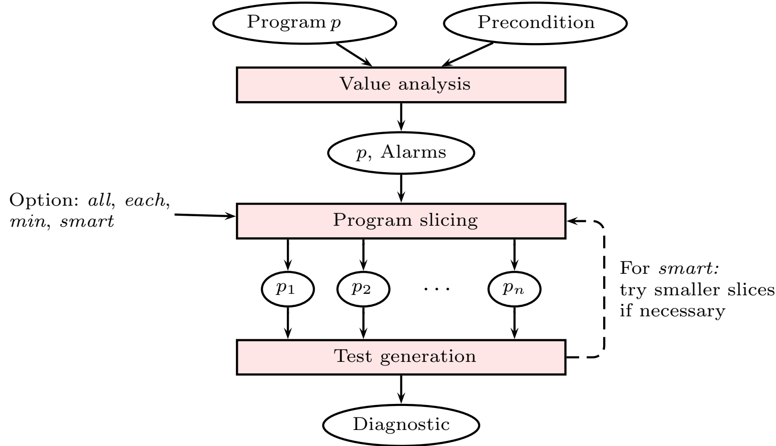

Fig. 5. Overview of the Sante method

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.

7 Adoption

Adoption in the academic world has stemmed from a variety of partnerships. Foremost is the Jessie plug-in [39] developed at Inria, which relies on a separation memory model but whose internal representation precludes its combination with other plug-ins. Verimag researchers have implemented a taint analysis [12], producing explicit dependency chains pondered by risk quantifiers. Demay *et al* generate security monitors based on fine-grained feedback from the Value plug-in [24]. Berthome *et al* [5] propose a source-code model for verifying physical attacks on smart cards, and use Value to verify it. Bouajjani *et al* [9] automatically synthesize invariants of sequential programs with singly-linked lists.

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 [13] have valiantly produced one such comparison, including the Value plug-in. Industrial reception has been extremely positive. Delmas *et al* verify the compliance to domain-specific coding standards [22]; their plug-in is undergoing deployment and qualification. At the same company, the value analysis is used to verify the control and data flows of a DAL C, 40-kloc ARINC 653 application [19]. Pariente and Ledinot [45] verify flight control system code using a combination of Frama-C plugins, including Value and Slicing. Their contribution includes a favorable evaluation of the cost-effectiveness of their adoption compared to traditional verification techniques. 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 [20]. Through these successes and over the past few years, Frama-C has demonstrated its adoptability within numerous industrial environments.

8 Conclusion

This article attempts to distill a synthetic 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 kloc. developed over the span of 7 years – has given rise to a flourishing ecosystem of software analyzers. In addition to industrial achievements and partnerships, a community of users and developers has grown and strived, contributing to the dissemination of the tools. This growth, fostered by a number of active communication channels[1], should be interpreted as a testimony to the health and stability of the platform, and good omens for its future success.

References

1. S. Bardin and P. Herrmann. OSMOSE: automatic structural testing of executables. *Software Testing, Verification and Reliability*, 21(1):29–54, 2011.
2. S. Bardin, P. Herrmann, and F. Védrine. Refinement-based CFG reconstruction from unstructured programs. In *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 54–69. Springer, 2011.
3. P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language Preliminary design, version 1.5*, 2010. `http://frama-c.com/downloads/acsl_1.5.pdf`.
4. N. Beckman, A. Nori, S. Rajamani, R. Simmons, S. Tetali, and A. Thakur. Proofs from tests. *IEEE Trans. Software Eng*, 36(4):495–508, 2010.
5. 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 *PLAS*, pages 2:1–2:12. ACM, 2010.

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

6. F. Bobot, S. Conchon, É. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The Alt-Ergo Automated Theorem Prover. http://alt-ergo.lri.fr.
7. R. Bornat. Proving Pointer Programs in Hoare Logic. In *Mathematics of Program Construction*, volume 1837 of *LNCS*, 2000.
8. 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 *AST*, pages 70–78, 2009.
9. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI*, pages 578–589, 2011.
10. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. 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.
11. R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
12. D. Ceara, L. Mounier, and M.-L. Potet. Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences. In *ICSTW*, pages 371–380, Washington, DC, USA, 2010.
13. G. Chatzieleftheriou and P. Katsaros. Test-driving static analysis tools in search of C code vulnerabilities. In *COMPSAC Workshops*, pages 96–103. IEEE, 2011.
14. O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *SAC*, 2012.
15. Coq Development Team. *The Coq Proof Assistant Reference Manual*, v8.3 edition, 2011. http://coq.inria.fr/.
16. L. Correnson and J. Signoles. Combining Analyses for C Program Verification. Submitted.
17. P. & R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
18. P. & R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *ESOP*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.
19. P. Cuoq, D. Delmas, S. Duprat, and V. Moya Lamiel. Fan-C, a Frama-C plug-in for data flow verification. In *ERTSS*, 2012.
20. 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 *NPIC*, July 2012.
21. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE Companion*. IEEE, 2009.
22. D. Delmas, S. Duprat, V. Moya Lamiel, and J. Signoles. Taster, a Frama-C plug-in to encode Coding Standards. In *ERTSS*, May 2010.
23. D. Demange, T. Jensen, and D. Pichardie. A provably correct stackless intermediate representation for java bytecode. In *APLAS*, volume 6461 of *LNCS*, pages 97–113. Springer, 2010.
24. 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.
25. E. W. Dijkstra. A constructive approach to program correctness. *BIT Numerical Mathematics*, Springer, 1968.
26. 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.
27. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.
28. R. W. Floyd. Assigning meanings to programs. *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, 19, 1967.

29. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV*, volume 2102 of *LNCS*, pages 53–65. Springer, July 2001.
30. P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT*, volume 493 of *LNCS*, pages 169–192. Springer, 1991.
31. J. Groslambert and N. Stouls. Vérification de propriétés LTL sur des programmes C par génération d'annotations. In *AFADL*, 2009. In French.
32. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.
33. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI, SIGPLAN Notices*, volume 23-7, pages 35–46, June 1988.
34. IEEE Std 754-2008. IEEE standard for floating-point arithmetic. Technical report, 2008. `http://dx.doi.org/10.1109/IEEESTD.2008.4610935`.
35. ISO/IEC JTC1/SC22/WG14. *9899:TC3: Programming Languages—C*, 2007. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.
36. N. Kosmatov. Online version of PathCrawler. `http://pathcrawler-online.com/`.
37. N. Kosmatov. *Artificial Intelligence Applications for Improved Software Engineering Development: New Prospects*, chapter XI: Constraint-Based Techniques for Software Testing. IGI Global, 2010.
38. K. R. M. Leino. *This is Boogie 2*. Microsoft Research, 2008.
39. C. Marché and Y. Moy. *The Jessie plugin for Deduction Verification in Frama-C, version 2.30*. INRIA, 2012. `http://krakatoa.lri.fr/jessie.pdf`.
40. B. Marre and A. Arnould. Test sequences generation from Lustre descriptions: GATeL. In *ASE*, pages 229–237, Grenoble, France, September 2000.
41. MathWorks. Polyspace. `http://www.mathworks.com/products/polyspace`.
42. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1997.
43. Y. Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, University Paris 11, January 2009.
44. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC*, 2002.
45. D. Pariente and E. Ledinot. Formal Verification of Industrial C Code using Frama-C: a Case Study. In *FoVeOOS*, 2010.
46. A. Pnueli. The temporal logic of programs. In *FOCS*. IEEE, 1977.
47. F. Randimbivololona, J. Souyris, P. Baudin, A. Pacalet, J. Raguideau, and D. Schoen. Applying formal proof techniques to avionics software: a pragmatic approach. In *FM*, volume 1709 of *LNCS*, pages 1798–1815. Springer, 1999.
48. J. Schimpf and K. Shen. ECLiPSe - from LP to CLP. *Theory and Practice of Logic Programming*, 12(1-2):127–156, 2011.
49. 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.
50. J. Signoles, L. Correnson, and V. Prevosto. *Frama-C Plug-in Development Guide*, October 2011. `http://frama-c.com/download/plugin-developer.pdf`.
51. N. Stouls and V. Prevosto. *Aoraï plug-in tutorial, version Nitrogen-20111001*, October 2011. `http://frama-c.com/download/frama-c-aorai-manual.pdf`.
52. N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In *EDCC*, volume 3463 of *LNCS*, pages 281–292. Springer, 2005.