StaDy: Deep Integration of Static and Dynamic Analysis in Frama-C

Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, Jacques Julliand

StaDy: Deep Integration of Static and Dynamic Analysis in Frama-C

Guillaume Petiot[1,2], Nikolai Kosmatov[1],
Alain Giorgetti[2,3], and Jacques Julliand[2]

[1] CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[2] FEMTO-ST/DISC, University of Franche-Comté, 25030 Besançon Cedex France
firstname.lastname@femto-st.fr
[3] INRIA Nancy - Grand Est, CASSIS project, 54600 Villers-lès-Nancy France

Abstract. We present StaDy, a new integration of the concolic test generator PathCrawler within the software analysis platform Frama-C. When executing a dynamic analysis of a C code, the integrated test generator also exploits its formal specification, written in an executable fragment of the acsl specification language shared with other analyzers of Frama-C. The test generator provides the user with accurate verdicts, that other Frama-C plugins can reuse to improve their own analyses. This tool is designed to be the foundation stone of static and dynamic analysis combinations in the Frama-C platform. Our first experiments confirm the benefits of such a deep integration of static and dynamic analysis within the same platform.

Keywords: static analysis, dynamic analysis, acsl specification language, C program verification, Frama-C

1 Introduction

Correctness of critical systems can be established using various verification methods such as static analysis (SA) and dynamic analysis (DA). Static analysis is performed on the source code without executing the program, whereas dynamic analysis is performed during execution of the program. Combinations of static and dynamic analysis techniques have proven their benefits [14,26,15]. In this paper, we propose StaDy, a tool integrating PathCrawler [3], a concolic test generator, into Frama-C [11], a software analysis framework.

PathCrawler [3] is a structural test generator for C programs, combining concrete and symbolic (also known as *concolic*) execution. PathCrawler is based on a specific constraint solver, Colibri, that is shared with other testing tools and implements advanced features such as floating-point and modular integer arithmetics support. PathCrawler provides coverage strategies like *all-k-paths* (feasible paths with at most k consecutive loop iterations) and *all-paths* (all feasible paths without any limitation on loop iterations). PathCrawler is *sound*, meaning that each test case activates the test objective (path, branch,

statement, etc.) for which it was generated. This is verified by concrete execution. When all features of the program are supported by PathCrawler and when constraint solving terminates for all paths, PathCrawler is also *complete,* i.e. the absence of a test for some test objective means that this test objective is infeasible, since the tool does not approximate path constraints like some other concolic tools. It has however some limitations, for instance it does not support recursive structures, recursive functions and function pointers. The PathCrawler test generation method is similar to the one of other concolic test generators like Dart/Cute [27], Smart [17], Pex [29], Sage [18] and Exe/Klee [5].

Frama-C [11] is a platform dedicated to analysis of C programs that includes various static analyzers, for example, a value analysis plugin, Value [6], based on abstract interpretation, a deductive verification plugin, Wp, and a dependence-based program slicing [31], Slicing. Frama-C supports acsl (ANSI C Specification Language), a behavioral specification language allowing to express properties over C programs. For combinations with dynamic analysis, we consider its executable subset e-acsl [13]. e-acsl can express function contracts (pre-/postconditions, guarded behaviors, completeness and disjointness of behaviors), assertions and loop contracts (variants and invariants). It supports quantifications over finite intervals of integers, mathematical integers and memory-related constructs (e.g. on validity and initialization). Inductive predicates and logic definitions are not included in e-acsl. In addition to providing formal specifications for C programs, acsl annotations play a central role in communication between plugins: any plugin can add annotations to be verified by other plugins and notify other plugins about analysis results it performed by changing an annotation status [9]. The status can indicate that the annotation is valid, valid under conditions, invalid or unknown, and which analyzer established this result [11]. The support of acsl is thus necessary for each plugin combined in Frama-C.

The sante method [7] proposed an earlier combination of PathCrawler with Frama-C plugins in order to detect runtime errors in C programs. This method uses value analysis to report potential runtime errors, then slices the program code with respect to their locations in order to eliminate irrelevant program statements and reduce program paths, and finally generates tests on the simplified code to validate or invalidate those errors. The current implementation of sante uses the Frama-C plugins Value, Slicing and the PathCrawler test generator. However, sante only handles a small subset of acsl specifications (assertions preventing divisions by zero, out-of-bounds array accesses and arithmetic overflows), and does not share the results of the test generation with other plugins. The need to extend the support of acsl constructions was one of our main motivations that lead to a new plugin, StaDy, which generalizes the sante method and overcomes its limitations.

The objective of this work is to facilitate the design of combined static and dynamic analyzers in Frama-C. On the one hand, such combinations allow any SA plugin to reuse the results of test generation, allowing the DA to help the SA

and conversely [14,26,15]. On the other hand, during deductive verification of C programs, test generation can automatically provide the validation engineer with a fast and helpful feedback facilitating the verification task. For example, while specifying a C program, test generation may find a counter-example showing that the current specification does not hold for the current code. In case of a proof failure for a specified program property during program proof, when the validation engineer has no other alternative than manually analyzing the reasons of the failure, test generation can be particularly useful. The absence of counter-examples after a rigorous partial (or, when possible, complete) exploration of program paths provides additional confidence in (respectively, guarantee of) the correctness of the program with respect to its current specification. This feedback may encourage the engineer to think that the failure is due to a missing or insufficiently strong annotation (loop invariant, assertion, etc.) rather than to an error, and to write such additional annotations. On the contrary, a counter-example immediately shows that the program does not meet its current specification, and prevents the waste of time of writing additional annotations. Moreover, the concrete test inputs and corresponding program path reported by the testing tool precisely indicate the erroneous situation and help the validation engineer to find the problem.

The first step towards the design of such combined toolchains in Frama-C is to deeply integrate PathCrawler into the platform: PathCrawler has to handle acsl specifications that Frama-C plugins use to share information, and test results have to be made available for other plugins and the user. To the best of our knowledge, such a deep integration of static and dynamic analysis tools within a complete software analysis framework as mature as Frama-C has never been performed before.

The contributions of this paper include:

- a detailed description of how StaDy integrates dynamic and static analyzers together thanks to the three following functionalities: instrumentation for DA of C annotated programs for the largest part of e-acsl specification language, test generation and reporting of DA results for SA plugins;
- use cases of StaDy on concrete examples illustrating the benefits of combining test generation with other Frama-C analyzers, for example, completing the results of deductive verification using all-path testing by validating a property or by finding a counter-example, and applying value analysis to facilitate the debugging task;
- experimental results showing the efficiency and the bug detection power of StaDy.

The paper is organized as follows. Sec. 2 gives a global overview of the approach and the tool architecture. The use cases on some illustrative examples are presented in Sec. 3. Our experiments evaluating the error detection capacity of combined static-dynamic analysis based on StaDy are described in Sec. 4. Sec. 5 presents the related work, and finally, Sec. 6 concludes.

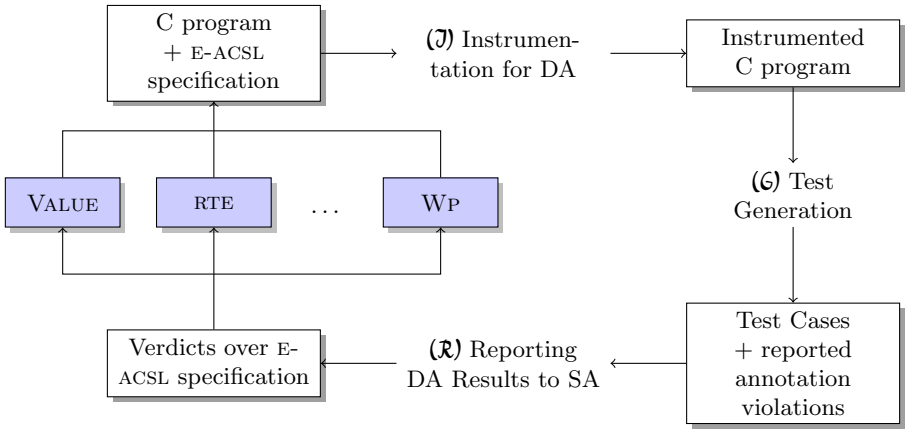

Fig. 1. Tool architecture

2 Tool Architecture

StaDy is designed as a Frama-C plugin. Fig. 1 shows how StaDy allows to use different static analyzers of Frama-C in combination with PathCrawler. StaDy sequentially performs three main steps that we detail below:

(J) Instrumentation that translates acsl annotations into executable C code,
(G) test generation with PathCrawler, and
(R) reporting and interpretation of the results.

(J) *Instrumentation for Dynamic Analysis.* The purpose of this step is to translate the e-acsl specification of a program, that is not yet validated, into a format that PathCrawler understands. Furthermore, the traceability of the properties' status has to be ensured by the translation.

PathCrawler requires a precondition of the function under test to generate input data that respect this precondition. PathCrawler accepts a precondition written in two different formalisms. The first one is a declarative constraint-based formalism (in Prolog), we call it the *native precondition*. The second one is defined as a C function that is called to check if the input data respect it, we call it the *C precondition* [12]. The C precondition can be much less efficient than the native precondition but has a larger expressive power. Indeed, the native precondition has some limitations forbidding the user to define constraints involving implications, disjunctions and some sorts of quantified predicates, those constraints should thus be defined in the C precondition. Simpler constraints like defining variable domains and unquantified constraints over them are defined in the native precondition. Using the native precondition for constraints that can be translated into this formalism rather than the C precondition in any case diminishes the number of treated paths and thus the test generation time.

Properties other than preconditions are translated into C code. This transformation is similar to the one described in [13] and implemented in the e-acsl2c

[21] plugin. Each property to check is translated as an `if` statement, asserting whether the property holds or not, so that two paths will split up at this point. The unique identifier of each program property is reused in the instrumented code. Thus the test generator associates its results to the properties it (in)validates, ensuring traceability. This step is parametrized by a set of properties: only the selected properties are translated so that we do not uselessly generate additional paths for concolic execution. This feature is important for combining StaDy with other analyzers, in particular, when some properties have already been validated and the user does not want to check them again.

(𝔊) *Test Generation and Verdict Traceability.* We use PathCrawler to generate inputs covering all feasible paths of the program. It performs a depth-first exploration of the program paths. Each path is symbolically executed and constraints corresponding to the current (partial) path are computed. The exploration goes deeper if the constraints are satisfiable or moves to the next partial path if the constraints are not satisfiable. If the path constraints are still satisfiable at the end of the path, test inputs satisfying those constraints are generated, so that running the program with those inputs will cover that particular path. Translating properties' checks into additional `if` statements (see step (𝔍)) generates additional branches in the program, so PathCrawler [3] tries to generate inputs satisfying the corresponding property and inputs satisfying its negation.

If the test generation terminates normally with no error after a complete *all-path* exploration, we have insurance that the properties hold. If only an incomplete path exploration is performed (with a partial coverage strategy like *all-k-paths*, or when test generation is stopped by a timeout, a segmentation fault, or because it encountered unsupported C constructs) test generation can increase the confidence the user can have in the correctness of the program with respect to its specification, but cannot guarantee it.

(𝔎) *Reporting DA Results to SA.* We retrieve the identifiers of violated properties with the inputs and outputs of the corresponding counter-examples. The unique identifier of the property received from PathCrawler allows to update the status of the corresponding property in the property database in Frama-C. At the end of a run of StaDy we get a mapping from invalid acsl properties to their counter-examples. Informations such as the number of generated test cases and counter-examples for each violated property are made available to other Frama-C plugins for reuse and are also displayed in the Frama-C Graphical User Interface (GUI).

StaDy is fully integrated in the GUI of Frama-C. When left-clicking on a violated annotation, the corresponding test inputs and outputs are textually displayed. When right-clicking on a violated annotation, we can open the corresponding test cases in Frama-C and then run other analyzers on it. The benefits of this feature will be illustrated in Sec. 3.3 by running the plugin Value on a generated counter-example.

```
1
2
3
4
5  /*@ requires 1 ≤ n;
6    @ requires \valid(t+(0..n-1));
7    @ typically n ≤ 5;
8    @ ensures ∀ x ∈ ℤ;
9        0 ≤ x < n-1 ⇒
10         t[x] ≤ t[x+1];
11   @*/
12 void insertion(int t[],int n) {
13  int i,j,mv;
14  for (i = 1; i < n; i++) {
15   mv = t[i];
16   for (j = i; j > 0; j--) {
17    if (t[j-1] ≤ mv) break;
18    t[j] = t[j-1];
19   }
20   t[j] = mv;
21  }
22
23
24
25 }
```

(a) Input function specified in ACSL

```
1  int forall_0 (int *t,int n) { int x;
2   for (x = 0; x < n-1; x++)
3    if (!(*(t+x) ≤ *(t+(x+1)))) return 0;
4   return 1; }
5
6  int insertion_precond(int *t,int n) {
7   if (!(1 ≤ n ∧ n ≤ 5)) return 0;
8   if (!(n-1 >= 0 ∧
9      pathcrawler_dim(t) > n-1)) return 0;
10  return 1; }
11
12 void insertion(int *t, int n) {
13  int i,j,mv;
14  for (i = 1; i < n; i++) {
15   mv = t[i];
16   for (j = i; j > 0; j--) {
17    if (t[j-1] ≤ mv) break;
18    t[j] = t[j-1];
19   }
20   t[j] = mv;
21  }
22  if (!((forall_0(t,n))))
23   pathcrawler_assert_exception
24    ("Post-condition!", 2);
25 }
```

(b) Its code instrumented for DA

Fig. 2. Insertion sort example

3 Applications

StaDy can be used to validate or invalidate ACSL properties, specified by the user or generated by another analyzer. Running after another analyzer, StaDy can complement or reinforce already computed results. For example, running StaDy after Value will check assertions (preventing divisions by zero, overflows, invalid memory accesses, ...) added by Value. Running StaDy after Wp will give a useful feedback to continue the proof, indicating that either a counter-example was found (and thus explaining a proof failure) or validating some unproven properties (and thus giving users more confidence that their programs respect their specification). Dually, it is also possible to run other analyzers after StaDy to check properties StaDy was not able to (in)validate, and use the results computed by StaDy, such as the counter-examples for the violated properties. This section details some of these use cases.

3.1 Validating a property by all-path testing

We now detail how StaDy validates an E-ACSL property by all-path testing and discuss each of the three steps (see Sec. 2) on an illustrative example. Let us consider the example of an insertion sort (in Fig. 2(a)), with a precondition (lines 5 and 6 in Fig. 2(a)) which specifies that the array length is at least one and that its elements are valid, and with a postcondition (lines 8-10 in Fig. 2(a)) which specifies that the resulting array `t` is sorted in ascending order.

We propose the new clause **`typically`** `c` that extends E-ACSL and defines a precondition `c` only to limit test generation. It restricts the number of paths to be explored. It is ignored by the static analyzers of Frama-C.

(𝒥) *Instrumentation.* Fig. 2(b) displays the source code after the instrumentation. The postcondition (**ensures** clause, lines 8–10 in Fig. 2(a)) is translated using an auxiliary function `forall_0` (lines 1–4 in Fig. 2(b)) and a call to a built-in function `pathcrawler_assert_exception` that reports a failure whenever the postcondition is not satisfied. The function at lines 6–10 in Fig. 2(b) translates the precondition (lines 5–7 in Fig. 2(a)) into C. When the function is called, `insertion_precond` is used by PathCrawler to check whether the inputs respect the precondition.

(𝒢) *Test generation.* StaDy runs PathCrawler to explore all feasible paths of the instrumented program. The clause **typically** states that n takes values between 1 and 5 and the precondition states that t is an array of length n (lines 6–7 in Fig. 2(a)). So, feasible paths may contain 0, 1, 2, 3 or 4 iterations of the outer loop. For each of those paths, test generation for the instrumented program tries to enforce that `!forall_0(t,n)` holds and to generate test inputs covering this branch. It does the same for the negation of this condition. For this example, the constraint solver used by PathCrawler establishes that for each value of n, the constraints of the paths so that the `!forall_0(t,n)` condition is *true* are unsatisfiable, thus the paths where the postcondition does not hold are infeasible. It means that the postcondition always holds when the specified precondition is respected.

(ℛ) *Reporting.* The messages received from PathCrawler for this example are the number of generated test cases (1863) and the information that there were no run-time errors such as stack overflow, segmentation fault or assertion violation on these test cases. This feedback indicates that all the instrumented properties hold for the cases where n is in the interval [1..5]. The only instrumented property in Fig. 2(a) being the postcondition, we can validate its status in Frama-C for $n \in [1..5]$.

3.2 Completing the results of deductive verification

Having validated the postcondition for $n \in [1..5]$ we have a good level of confidence in the contract. Now, we can pursue the specification work to prove the program. Let us now reconsider the insertion sort implementation with a more complete ACSL specification (see Fig. 3). Applying the deductive verification plugin Wp will assign the status "Unknown" to the invariant `INV` of line 10, and the status "Valid under hypotheses" to the properties of lines 3, 4, 9, 11, 12 and 17 to 20. After running Wp the dependence graph of properties [9] – available from the GUI of Frama-C – states that if the invariant `INV` holds, then the other properties would also hold. The failure of Wp is fully explainable: The program is not sufficiently specified to discharge the verification condition associated to that invariant by applying the weakest precondition calculus. If we run StaDy on the `INV` invariant, we can validate it according to the technique from Sec. 3.1. StaDy generates 1863 test cases and changes `INV`'s status to "valid under hypotheses": `INV` holds if `n` $\leq$ `5` holds. This example demonstrates how StaDy can help the user to trust her program and specification even if some properties have

```
1  /*@ requires 1 ≤ n ≤ 5;
2    @ requires \valid(t+(0..n-1));
3    @ assigns t[0..n-1];
4    @ ensures ∀ x ∈ ℤ; 0 ≤ x < n-1 ⇒ t[x] ≤ t[x+1];
5    @*/
6  void insertion_sort(int t[], int n) {
7    int i = 1,j,mv;
8
9    /*@ loop invariant 1 ≤ i ≤ n;
10     @ loop invariant INV: ∀ x ∈ ℤ; 0 ≤ x < i-1 ⇒ t[x] ≤ t[x+1];
11     @ loop assigns i, j, mv, t[0..n-1];
12     @ loop variant n-i;
13     @*/
14   for (; i<n; i++) {
15     mv = t[i];
16     j = i;
17     /*@ loop invariant 0 ≤ j ≤ i;
18       @ loop invariant ∀ k ∈ ℤ; j ≤ k < i ⇒ t[k] > mv;
19       @ loop assigns j, t[0..n-1];
20       @ loop variant j;
21       @*/
22     for (; j > 0; j--) {
23       if (t[j-1] ≤ mv) break;
24       t[j] = t[j-1];
25     }
26     t[j] = mv;
27   }
28 }
```

Fig. 3. Insertion sort annotated

not been automatically discharged by Wp. The process helps the user to refine specifications to the point of being able to formally verify them automatically with tools such as Wp.

3.3 Generating a counter-example

To illustrate the counter-example generation we consider the Traffic Alert and Collision Avoidance System (TCAS) whose implementation has been detailed and verified in [20]. More complete results on this program are in Sec. 4. Figure 4 displays a short part of this program considered in this section, composed of global variables, their initialization, the precondition of the entry point and the behavior `P2b`.

When checking this property, StaDy finds some counter-examples, such as: `alt_layer_value = 0`, `down_separation = 0`, `other_rac = 0`, `other_tracked_alt = 441`, `own_tracked_alt = 438`, `two_of_three_reports_valid = 1`, `up_separation = -15`. We are able to open the C program corresponding to this counter-example in Frama-C and run Value on it with those inputs. The abstract interpretation analysis of Value establishes that the return value of the tested function will take values in the set $\{1\}$, thus violating the postcondition of behavior `P2b`. So counter-examples generated by StaDy can be confirmed by Value. We provide a way to do this in the graphical user interface of Frama-C, where the user can click on each variable of the program and see the over-approximated set of values computed by Value at each point of the program. This feature is very useful when debugging a program, however it suffers from the limitations of Value: Quan-

```
1  int Cur_Vertical_Sep = 16684, High_Confidence = 32767,
2    Two_of_Three_Reports_Valid, Own_Tracked_Alt, Own_Tracked_Alt_Rate = 450,
3    Other_Tracked_Alt, Alt_Layer_Value, Up_Separation, Down_Separation,
4    Other_RAC, Other_Capability = 0, Climb_Inhibit = 1,
5    Positive_RA_Alt_Thresh[4] = {16434,0,0,0};
6
7  /*@ requires 300 ≤ other_tracked_alt ≤ 1000;
8    @ requires 300 ≤ own_tracked_alt ≤ 1000;
9    @ requires 0 ≤ alt_layer_value < 4;
10   @ requires 0 ≤ other_rac ≤ 2;
11   @ requires 0 ≤ two_of_three_reports_valid ≤ 1;
12   @
13   @ behavior P2b :
14   @   assumes up_separation < Positive_RA_Alt_Thresh[alt_layer_value];
15   @   assumes down_separation < Positive_RA_Alt_Thresh[alt_layer_value];
16   @   assumes up_separation < down_separation;
17   @   ensures \result ≠ 1;
18   @*/
```

Fig. 4. ACSL specification of the TCAS

tified predicates are not supported, and higher precision comes with a higher computation cost.

4 Experimental Results

We apply StaDy on safe and unsafe programs from the TACAS 2014 Software Verification Competition [1] to evaluate its efficiency. First, we track down bugs in faulty programs. Secondly, we use mutation testing to evaluate the ability of StaDy in finding bugs on unsafe programs automatically generated from safe programs. Finally, we also run StaDy on a real application (TCAS) verified in [20] by a constraint solving test generator.

Bug detection on faulty programs. We executed StaDy on 20 faulty programs of the TACAS 2014 Software Verification Competition [1] (selected from the subdirectory *loops*). These programs handle arrays with loops. The properties to invalidate are expressed as C assertions. They have been manually rewritten in ACSL. Adequate ACSL preconditions have also been added. The programs containing infinite loops and reachability properties to invalidate are not handled by StaDy due to the necessity to execute the program in PathCrawler. We detected the failure of properties in each considered program. The results can be seen in Fig. 5. The time taken to invalidate the properties (see the corresponding column of Fig. 5) includes all the steps of StaDy: instrumentation from the E-ACSL specifications, test generation in PathCrawler and reporting results to Frama-C.

Validation of safe programs and mutation testing. We execute StaDy on 20 safe programs of the same benchmark, and 6 additional safe programs from our own benchmarks. All of them are annotated in ACSL. They contain preconditions, postconditions, assertions, memory-related properties, loop variants

example	time (sec.)
array-unsafe	1.169
count-up-down-unsafe	1.187
eureka-01-unsafe	1.816
for-bounded-loop1-unsafe	1.246
insertion-sort-unsafe	1.267
invert-string-unsafe	1.401
linear-search-unsafe	1.309
matrix-unsafe	1.375
nec20-unsafe	1.363
string-unsafe	1.436
sum01-bug02-basecase-unsafe	1.167
sum01-bug02-unsafe	1.134
sum01-unsafe	1.148
sum03-unsafe	1.193
sum04-unsafe	1.129
sum-array-unsafe	1.298
trex01-unsafe	1.561
trex03-unsafe	1.359
verisec-sendmail-tTflag-arr-one-loop-unsafe	1.319
vogal-unsafe	1.977

Fig. 5. Bug detection of TACAS 2014 Software Verification Competition unsafe programs

and invariants. We use *mutation testing* on these safe programs to generate unsafe programs – mutants – and see if StaDy is able to invalidate them. The mutations performed on the source code mimic usual programming errors. They include modifications of numerical and/or pointer arithmetic operators, comparison operators, condition negation and logical operators (*and* and *or*). Figure 6 summarizes the results of StaDy on the mutants. StaDy killed erroneous mutants with a success rate varying from 71.43 to 100 % (average rate is 95.09 %). The missing percents are mostly due to the incompleteness of the specifications in some programs, or explained by the limitations of the tool: some complex acsl constructs are not yet supported.

Validating TCAS properties. We also apply StaDy on the TCAS introduced in Sec. 3.3. StaDy was able to detect as many failures as the constraint solving based test generator presented in [20], i.e. to generate a counter-example for five properties, in particular `P2b`.

5 Related Work

Executing specifications. Our work continues previous attempts to execute formal specifications. A method for executing Boogie [23] specifications based on deductive synthesis and constraint solving at run-time (using SMT solvers) is described in [22]. It infers partial implementation from Boogie contracts. Similarly, another method uses symbolic execution and SMT constraint solving to

example	mutants	non equiv.	killed	success rate
merge-sort	96	92	88	95.65%
merge-arrays	68	63	59	93.65%
quick-sort	130	130	130	100%
binary-search	40	40	39	97.5%
bubble-sort	52	49	42	85.71%
insertion-sort	39	37	36	97.3%
TACAS 2014 Software Verification Competition Benchmarks				
array-safe	18	16	15	93.75%
bubble-sort-safe	64	58	55	94.83%
count-up-down-safe	14	13	13	100%
eureka-01-safe	60	60	60	100%
eureka-05-safe	36	36	36	100%
insertion-sort-safe	43	41	40	97.56%
invert-string-safe	47	47	47	100%
linear-search-safe	19	17	16	94.12%
matrix-safe	30	27	25	92.59%
nc40-safe	20	20	20	100%
nec40-safe	20	20	20	100%
string-safe	65	65	65	100%
sum01-safe	14	14	13	92.86%
sum02-safe	14	14	11	78.57%
sum03-safe	10	10	10	100%
sum04-safe	14	14	10	71.43%
sum-array-safe	17	17	15	88.24%
trex03-safe	56	56	56	100%
verisec-sendmail-tTflag-arr-one-loop-safe	31	31	31	100%
vogal-safe	71	68	67	98.53%

Fig. 6. Error detection for mutants

execute Boogie specifications. This method is implemented in Boogaloo [25] and its purpose is to generate counter-examples. StaDy offers the possibility to execute specifications and to test program conformance with respect to its specification for C programs. In addition, it allows to integrate the results into a complete analysis framework and to reuse them automatically in other analyzers or manually, e.g. during proof failure analysis.

Invariant inference, static analysis and test generation. DSD-Crasher [10] infers invariants from concrete executions, uses the static analyzer ESC-Java to check the inferred invariants and report alarms, and finally confirms the alarms dynamically. However, ESC-Java is not sound [16], and the overall analysis depends on the quality of the inferred invariants. In comparison, StaDy does not infer invariants from concrete executions, but specification can be automatically added by applying static analysis plugins such as rte (adding assertions preventing potential runtime errors) or Value (adding assertions preventing potential unspecified behaviors that were not proved safe by value analysis). Contrary to

ESC-Java, the deductive verification plugin Wp and the Value plugin, used to check the specification, are sound.

Combination of value analysis, slicing and test generation. Abstract interpretation, slicing and concolic test generation are combined within the sante method [7]. The current implementation of the method only handles divisions by zero, out-of-bounds array accesses and arithmetic overflow assertions generated by the Value analyzer. StaDy supports a larger subset of the e-acsl specification language, including quantified predicates, pre-/postconditions, behaviors loop invariants and variants. sante does not provide a feedback of the test cases generated by PathCrawler, so its results cannot be reused in another analysis. However, sante uses a *slicing* step to reduce the program size and thus quicken the test generation. This missing feature is part of our future work on StaDy.

Synergy of instrumentation, slicing and symbolic execution. Symbiotic [28] implements a technique combining instrumentation, slicing and symbolic execution. This method aims to detect bugs in C programs described by finite state machines, and is applied to reachability problems. Instrumentation adds `assert(0)` assertions at error labels, that are supposed to be unreachable. Slicing produces a reduced state machine with the same behaviors as the original program. Symbolic execution is used to find bugs in the sliced state machine. This method uses similar techniques as sante [7], but it is only applicable to reachability problems, and thus cannot handle general properties over C programs like StaDy does. In counterpart StaDy cannot handle reachability problems.

Integration of static and dynamic analysis in a common framework. Among previous combinations of static and dynamic analyses, [2,19] developed combinations of predicate abstraction and software testing. [4] described HOL-TestGen, a formally verified test-system extending the interactive theorem prover Isabelle/HOL. The design of JML accommodates both deductive and runtime verification [24]. Combinations of deductive verification and testing for imperative languages were recently studied and implemented for C# programs specified with Boogie in [25], and combining Dafny and Pex in [8]. Static and dynamic analysis have been combined within the Eve [30] verification environment, built on top of an Eiffel IDE. Static provers and dynamic testing tools can run on the same programs and complement their results. The modularity and extensibility of Eve are similar to Frama-C ones. AutoTest, the test generator used by Eve, uses the contracts of the program to generate test cases, whereas PathCrawler produces a structural coverage of the program. AutoProof, the static checker used by Eve and relying on the Boogie verifier, does not support floating-point arithmetic, whereas the Value analyzer of Frama-C supports this feature.

6 Conclusion and Future Work

We have presented an original integration of a concolic test generator within the framework Frama-C, facilitating the design of combined analyzers of C code where SA plugins work and exchange results with the test generator PathCrawler. We described some applications of StaDy and demonstrated its benefits by several use cases. We stated that StaDy can validate a property by generating test cases with an all-path coverage. It can also complete the results of the deductive verification plugin Wp, generate counter-examples for violated properties of a program and takes benefits from a full integration within the graphical user interface of Frama-C. We showed its applicability with experiments on various programs. Finally, we discussed the advantages of StaDy compared to other approaches and tools. The overall advantage of StaDy is the possibility to combine it with other Frama-C plugins to benefit from their strengths to design powerful SA and DA analysis, and the ability to express complex properties thanks to the acsl specification language.

Future work on StaDy includes better support of e-acsl constructs, such as (non-inductive) logic predicates. The better we support acsl, the better we can take advantage of its richness and expressivity to handle more complex programs with complex specifications. Future work also includes an integration of a slicing step to reduce the size of the programs we run PathCrawler on, to improve the scalability of our tool, as it is done in the sante method [7].

Acknowledgment. The authors thank the Frama-C and PathCrawler teams for providing the tools and support. Special thanks to Patrick Baudin, Mohammed Bekkouche, François Bobot, Bernard Botella, Loic Correnson, Pascal Cuoq, Zaynah Dargaye, Mathieu Lemerre, Virgile Prevosto, Muriel Roger, Julien Signoles, Nicky Williams and Boris Yakobowski for many fruitful discussions, suggestions and advice.

References

1. Tacas 2014 software verification competition (2014), `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/c/`
2. Beyer, D., Henzinger, T., Theoduloz, G.: Program analysis with dynamic precision adjustment. In: ASE (2008)
3. Botella, B., Delahaye, M., Hong Tuan Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: AST (2009)
4. Brucker, A.D., Wolff, B.: On theorem prover-based testing. FAC (2012)
5. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI (2008)
6. Canet, G., Cuoq, P., Monate, B.: A value analysis for C programs. In: SCAM (2009)
7. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC (2012)

8. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: FM (2012)
9. Correnson, L., Signoles, J.: Combining analyses for C program verification. In: FMICS (2012)
10. Csallner, C., Xie, T.: DSD-Crasher: A hybrid analysis tool for bug finding. In: ISSTA (2006)
11. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C - a software analysis perspective. In: SEFM (2012)
12. Delahaye, M., Kosmatov, N.: A late treatment of c precondition in dynamic symbolic execution testing tools. In: RV (2013)
13. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: SAC (2013)
14. Ernst, M.D.: Static and dynamic analysis: Synergy and duality. In: WODA (2003)
15. Ernst, M.D.: How tests and proofs impede one another: the need for always-on static and dynamic feedback. In: TAP (2010)
16. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI (2002)
17. Godefroid, P.: Compositional dynamic test generation. In: POPL (2007)
18. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: NDSS (2008)
19. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.D.: Compositional may-must program analysis: unleashing the power of alternation. In: POPL (2010)
20. Gotlieb, A.: TCAS software verification using constraint programming. Knowledge Eng. Review (2012)
21. Kosmatov, N., Petiot, G., Signoles, J.: An optimized memory monitoring for runtime assertion checking of C programs. In: RV (2013)
22. Kuncak, V., Kneuss, E., Suter, P.: Executing specifications using synthesis and constraint solving. In: RV (2013)
23. Le Goues, C., Leino, K.R.M., Moskal, M.: The boogie verification debugger. In: SEFM (2011)
24. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. In: FMCO (2003)
25. Polikarpova, N., Furia, C.A., West, S.: To run what no one has run before: Executing an intermediate verification language. In: RV (2013)
26. Rajamani, S.K.: Verification, testing and statistics. In: FM (2009)
27. Sen, K., Agha, G.: CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: CAV (2006)
28. Slaby, J., Strejček, J., Trtík, M.: Symbiotic: synergy of instrumentation, slicing, and symbolic execution. In: TACAS (2013)
29. Tillmann, N., de Halleux, J.: Pex-white box test generation for .net. In: TAP (2008)
30. Tschannen, J., Furia, C.A., Nordio, M., Meyer, B.: Usable verification of object-oriented programs by combining static and dynamic techniques. In: SEFM (2011)
31. Weiser, M.: Program slicing. In: ICSE (1981)