How Test Generation Helps Software Specification and Deductive Verification 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. This paper describes an incremental methodology of deductive verification assisted by test generation and illustrates its benefits by a set of frequent verification scenarios. We present STADY, a new integration of the concolic test generator PATHCRAWLER within the software analysis platform FRAMA-C. This new plugin treats a complete formal specification of a C program during test generation and provides the validation engineer with a helpful feedback at all stages of the specification and verification tasks.

Keywords: static analysis, test generation, specification, Frama-C, deductive verification

1 Introduction

Validation of critical systems can be realized using various verification methods based on static analysis, dynamic analysis or their combinations. Static analysis is performed on the source code without executing the program, whereas dynamic analysis is based on the program execution. Both are complementary and can be advantageously combined [10, 3, 14, 6, 7, 5, 18].

Among static techniques, formal deductive verification allows to establish a rigorous, mathematical proof that a given annotated program respects its specification. The modular verification approach requires a formal specification (contract) for each function describing its admissible inputs and expected results. Modern theorem proving tools can automatically establish many proofs of correctness, but achieving a fully successful proof in practice needs a lot of tedious work and manual analysis of proof failures by the validation engineers. Klein [15] estimates that the cost of one line of formally verified code is about $700. This high cost is explained by the great difficulty of understanding why a proof fails, and of writing correct and sufficiently complete specifications suitable for automatic proof of contracts for which loop variants and invariants can be required.

The main motivation of this methodology and tool paper is to study how automatic test generation can help to write a correct formal specification and to achieve its deductive verification. The contributions of this paper include:

* The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement Nº 269335 and from the French government.

– a brief presentation (in Sec. 2) of a combined STAtic/DYnamic tool named STADY. Within the software analysis framework FRAMA-C [11], this tool fills the gap between deductive verification and test generation and allows to treat a complete formal specification (including pre-/postconditions, assertions, loop invariants and variants) during test generation with PATHCRAWLER [4];
– a methodology of iterative deductive verification taking advantage of feedbacks provided by test generation (in Sec. 3). Its benefits are illustrated on a set of frequent verification scenarios;
– a summary of experiments showing STADY's bug detection power (in Sec. 3.4).

2 STADY Tool Overview

The STADY tool integrates the concolic test generator PATHCRAWLER [4] into the software analysis framework FRAMA-C [11], and in particular allows the user to combine it with the deductive verification plugin WP [11].

FRAMA-C [11] is a platform dedicated to analysis of C programs that includes various source code analyzers as separate plugins such as WP performing weakest-precondition calculus for deductive verification, VALUE performing value analysis by abstract interpretation, etc. FRAMA-C supports ACSL (ANSI C Specification Language) [2, 11], a behavioral specification language allowing to express properties over C programs. Moreover, ACSL annotations play a central role in communication between plugins: any analyzer can add annotations to be verified by other ones and notify other plugins about results of the analysis it performed by changing an annotation status. The status can indicate that the annotation is valid, valid under conditions, invalid or undetermined, and which analyzer established this result [9].

For combinations with dynamic analysis, we consider the executable subset of ACSL named E-ACSL [12, 19]. 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 bounded intervals of integers, mathematical integers and memory-related constructs (e.g. on validity and initialization).

PATHCRAWLER [4] is a structural (also known as *concolic*) test generator for C programs, combining *conc*rete and symb*olic* execution. PATHCRAWLER is based on a specific constraint solver, COLIBRI, that implements advanced features such as floating-point and modular integer arithmetics support. PATHCRAWLER provides coverage strategies like *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 for which it was generated. This is verified by concrete execution. PATHCRAWLER is also *complete* in the following sense: when the tool manages to explore all feasible paths of the program, all features of the program are supported by the tool and constraint solving terminates for all paths, the absence of a test for some test objective means that this test objective is infeasible, since the tool does not approximate path constraints [4, Sec. 3.1].

Given a C program annotated in the executable specification language E-ACSL [11], STADY first translates its specification into executable C code, instruments the program

for error detection, runs PATHCRAWLER to generate tests for the instrumented code, and finally returns the results to FRAMA-C. To detect errors, the translation generates additional branches, enforcing test generation to trigger erroneous cases, and thus to generate inputs activating the error if such inputs exist. In this way, STADY treats and triggers errors in assertions, postconditions, loop invariants and variants, and also in pre- and postconditions of called functions (also called *callees*). PATHCRAWLER being complete, whenever test generation terminates without finding any error after an exhaustive "all-path" coverage, we are sure that the translated E-ACSL properties hold. If the coverage is only partial but no error occurred, the test generation increases the confidence that the program respects its specification but cannot guarantee it. However, errors can be found and used to invalidate the annotations in FRAMA-C even when the coverage is incomplete.

STADY currently supports most ACSL clauses. Quantified predicates `\exists` and `\forall` and builtin terms as `\sum` or `\numof` are translated as loops. Logic functions and named predicates are handled, however recursivity is currently not supported. `\old` constructs are treated by saving the value of the formal parameters of a function. Validity checks of pointers are partially supported due to the current limitation of the underlying test generator: we can only check the validity when a base address is an input pointer. `assert`, `assumes`, `behavior`, `ensures`, `loop invariant`, `loop variant` and `requires` clauses are supported as well. `assigns` clauses and complex constructs like inductive predicates are not handled yet and are part of our future work.

3 Verification Scenarios Combining Proof and Testing

During specification and deductive verification, test generation can automatically provide the validation engineer with a fast and helpful feedback facilitating the verification task. While specifying a program, test generation may find a counter-example showing that the current specification does not hold for the current code. It can be used at early stages of specification, even when formal verification has no chances to succeed yet (e.g. when loop annotations, assertions or callees' contracts are not yet written). 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 (resp., guarantee of) 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, called function contract 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 activated program path reported by the testing tool precisely indicate the erroneous situation. Notice that the objective is certainly not *to fit the specification to (potentially erroneous) code*, but *to help the validation engineer to identify the problem (in the specification or in the code)* with a counter-example. Let us illustrate these points on concrete verification scenarios.

```
1  int delete_substr(char *str, int strlen, char *substr, int sublen, char *dest) {
2   int start = find_substr(str, strlen, substr, sublen), j, k;
3   if (start == -1) {
4    for (k = 0; k < strlen; k++) dest[k] = str[k];
5    return 0;
6   }
7   for (j = 0; j < start; j++) dest[j] = str[j];
8   for (j = start; j < strlen-sublen; j++) dest[j] = str[j+sublen];
9   return 1;
10 }
```

Fig. 1. Unspecified function `delete_substr` calling the function of Fig. 2

```
1  /*@ requires 0 < sublen ≤ strlen;
2    @ requires \valid(str+(0..strlen-1)) ∧ \valid(substr+(0..sublen-1));
3    @ assigns \nothing;
4    @ behavior found:
5    @   assumes ∃ i ∈ ℤ; 0 ≤ i < strlen-sublen ∧
6    @     (∀ j ∈ ℤ; 0 ≤ j < sublen ⇒ str[i+j] == substr[j]);
7    @   ensures 0 ≤ \result < strlen-sublen;
8    @   ensures ∀ j ∈ ℤ; 0 ≤ j < sublen ⇒ str[\result+j] == substr[j];
9    @ behavior not_found:
10   @   assumes ∀ i ∈ ℤ; 0 ≤ i < strlen-sublen ⇒
11   @     (∃ j ∈ ℤ; 0 ≤ j < sublen ∧ str[i+j] ≠ substr[j]);
12   @   ensures \result == -1; */
13 int find_substr(char *str, int strlen, char *substr, int sublen);
```

Fig. 2. Verified function `find_substr` with a "pretty-printed" E-ACSL contract

Suppose Alice is a skilled validation engineer in charge of specification and deductive verification of the function `delete_substr` (Fig. 1). We follow Alice throughout her validation process. The `delete_substr` function is supposed to delete one occurrence of a substring `substr` of length `sublen` from another string `str` of length `strlen` and to put the result into `dest` (pre-allocated for `strlen` characters), while `str` and `substr` should not be modified. For simplicity, we use arrays rather than usual zero-terminated strings. The `delete_substr` function returns 1 if an occurrence of the substring was found and deleted, and 0 otherwise. We assume Alice has already successfully proved the correctness of `find_substr` (Fig. 2) supposed to return the index of an occurrence of `substr` in `str` if this substring is present, and -1 otherwise.

Alice first writes the following precondition (added before line 1 of Fig. 1):

```
requires 0 < sublen ≤ strlen;
requires \valid(str+(0..strlen-1));
requires \valid(dest+(0..strlen-1));
requires \valid(substr+(0..sublen-1));
requires \separated(dest+(0..strlen-1), substr+(0..sublen-1));
requires \separated(dest+(0..strlen-1), str+(0..strlen-1));
typically strlen ≤ 5;
```

We propose here the new clause **`typically`** `C;` that extends E-ACSL and defines the precondition `C` only for test generation. It allows Alice to strengthen the precondition if she desires to restrict the (potentially too big) number of paths to be explored by test generation to user-controlled partial coverage. Here the clause **`typically`** `strlen` $\leq$`5` asks to cover all feasible paths where `str` is of length `5` or less. Ignored by deductive verification, this clause does not impact the proof. The extension of ACSL with the **`typically`** keyword is an experimental feature, not available in the distributed version of FRAMA-C.

3.1 Early Validation

Now Alice specifies that the function can assign only the array `dest`, and defines the postcondition for the case when the substring does not occur in the string. She adds the following (erroneous) clauses into the contract after the precondition defined above:

```
assigns dest[0..strlen-1];
behavior not_present:
 assumes !(∃ i ∈ ℤ; 0 ≤ i < strlen-sublen ∧
  (∀ j ∈ ℤ; 0 ≤ j < sublen ⇒ str[i+j] ≠ substr[j]));
 ensures ∀ k ∈ ℤ; 0 ≤ k < strlen ⇒ \old(str[k]) == dest[k];
 ensures \result == 0;
```

To validate it before going further, Alice applies STADY. It runs test generation and reports that both **`ensures`** clauses are invalidated by the counter-example `strlen = 2`, `sublen = 1`, `str[0] = 'A'`, `str[1] = 'B'`, `substr[0] = 'A'`, `dest[0] = 'B'` and **`\result`** `= 0`. Alice sees that in this case the string `substr` has to be found in the string `str` and the behavior `not_present` should not apply, so its **`assumes`** clause must be erroneous. This helps Alice to correct the assumption by replacing $\neq$ with $==$, to get:

```
  assumes !(∃ i ∈ ℤ; 0 ≤ i < strlen-sublen ∧
   (∀ j ∈ ℤ; 0 ≤ j < sublen ⇒ str[i+j] == substr[j]));
```

Running STADY again reports that all feasible paths with `strlen` ≤ 5 have been covered (within 3.4 sec.) and 9442 test cases have been successfully generated and executed. Alice is now pretty confident that this behavior is correctly defined.

For the complementary case Alice copy-pastes the `not_present` behavior and (wrongly) modifies it into the following behavior:

```
behavior present:
 assumes ∃ i ∈ ℤ; 0 ≤ i < strlen-sublen ∧
  (∀ j ∈ ℤ; 0 ≤ j < sublen ⇒ str[i+j] == substr[j]);
 ensures ∃ i ∈ ℤ; 0 ≤ i < strlen-sublen ∧
  (∀ j ∈ ℤ; 0 ≤ j < sublen ⇒ \old(str[i+j]) == \old(substr[j])) ∧
  (∀ k ∈ ℤ; 0 ≤ k < i ⇒ \old(str[k]) == dest[k]) ∧
  (∀ l ∈ ℤ; i ≤ l < strlen ⇒ \old(str[l+sublen]) == dest[l]);
 ensures \result == 1;
```

Again, Alice runs STADY. The tool reports an out-of-bounds error in accessing the element of `str` at index `l+sublen` in the last **`ensures`**. This helps Alice to understand that the upper bound of index `l` should be `strlen-sublen` instead of `strlen`. She fixes this error and re-runs STADY. Test generation reports that 13448 test cases cover without errors the feasible paths for `strlen` ≤ 5. Alice is now satisfied with the defined behaviors. Notice that these cases exhibit errors in the specification. In other cases errors could be in the program (cf Sec. 3.4).

3.2 Incremental Loop Validation

Alice now specifies as follows the first for-loop at line 4 in Fig. 1:

```
loop invariant ∀ m ∈ ℤ; 0 ≤ m < k ⇒ dest[m] == \at(str[m],Pre);
loop assigns k, dest[0..strlen-1];
loop variant strlen-k;
```

Then Alice runs WP. The deductive verification tool cannot validate the postcondition of `delete_substr`, in particular because the other two loops are not yet specified. However, WP could validate the annotations of the first loop. Here it fails, and Alice does not know whether it is because the loop specification is already incorrect, or because it is not complete enough to be verified. She runs STADY, which does not find any error in the loop specification and the postcondition, after 15635 test cases. Alice now believes that loop specification is valid but incomplete. This confidence helps her to add an additional invariant

```
loop invariant 0 ≤ k < strlen;
```

defining the bounds for k. Alice tries again to prove the loop, and WP fails again. She runs STADY and this time the new loop invariant is invalidated. After analyzing the failure on a simple counter-example, Alice understands that the loop invariant `k <strlen` is not correct. Indeed, `k` is equal to `strlen` after the last iteration, so the loop invariant should say `k ≤strlen`. After fixing this error, WP succeeds to prove the loop annotations. Similarly, Alice iteratively specifies and verifies the other two loops.

The now completely specified function `delete_substr` can be fully proved by WP. However its default timeout (10 seconds per property) has to be significantly extended (e.g. to 50 seconds per property). The fact that test generation achieves (within only 4 sec.!) a significant partial coverage (restricted by the **`typically`** clause for testing) and finds no error convinces Alice to increase the timeout, that could be a waste of time when a counter-example can show why the program does not respect the specification.

3.3 Adaptation of Callees' Contracts for Modular Verification

It often happens that the contract of a called function is fully proved, but is too weak to prove the caller. For instance assume that the clause at line 7 of Fig. 2 is missing. Running WP on the whole program, Alice sees that `find_substr` is totally proved, but the postcondition and loop annotations of `delete_substr` are not proved. Since test generation does not find any counter-example, Alice believes that some necessary clause is too weak or missing. Moreover, all properties depending on the behavior `not_found` being fully proved, Alice reasonably suspects that the `found` behavior of `find_substr` is not strong enough.

3.4 Detecting Errors in Source Code

Counter-examples generated by STADY can also help to detect potential errors in the code. To evaluate its bug detection ability, we specified in E-ACSL 26 programs mostly taken from the TACAS 2014 Competition, generated 1088 mutants (that mimic frequent programming errors) and applied STADY to detect errors in them. The E-ACSL contract in mutants was not changed. 96.68% of non equivalent mutants have been successfully reported as buggy.

4 Conclusion and Future Work

We showed by a number of selected verification scenarios how automatic test generation provides a useful feedback that helps the validation engineer to test the conformance of

a program to its (even partial) specification, identify errors, understand them thanks to generated counter-examples, and finally find missing, insufficient or wrong annotations, or detect bugs in the code. These scenarios sketch an iterative methodology assisted by test generation that makes deductive verification easier, less costly in time, more interactive and less error-prone. We presented the STADY tool, integrating a concolic test generator into FRAMA-C. A more detailed description of the STADY tool, some verification scenarios and initial experiments are available in [17].

Among previous combinations of static and dynamic analyses, [3, 14] developed combinations of predicate abstraction and software testing. [5] 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 [16]. Combinations of deductive verification and testing for imperative languages were recently studied and implemented for C# programs specified with Boogie in [18], and combining Dafny and Pex in [7]. In [8], the specification-based random testing tool Quickcheck is used to find counter-examples to invariants that have not been formally verified by automated theorem provers. [13] described an approach to show the correctness of a Java program and in case of a verification failure to show a counter-example or to guide the user. A counter-example is found based on information contained in proof trees of failed verification attempts, so the process has to start with a proof attempt. In our approach it is not necessary to start with a proof, the user may start by testing if she thinks the program is more likely to contain bugs. [1] addressed the verification of first-order logic axioms, that are provided by the user to theorem provers and supposed to hold. In this work, model-based random testing is used to find counter-examples to axiomatizations, but no coverage is ensured.

Our work continues these efforts for C programs in the FRAMA-C framework and proposes a methodology of incremental specification and deductive verification assisted by test generation. The SANTE method [6] proposed a combination of value analysis, slicing and test generation in order to detect runtime errors. Our present work combines deductive verification with testing, treats complete E-ACSL specifications (while SANTE treated only simple assertions) and thus handles in addition a large class of functional properties that were not supported in SANTE.

Future work includes further evaluation of the proposed methodology, experiments on real-size programs and a better support of E-ACSL constructs in our implementation (inductive predicates, assigns clauses, validity checks for non-input pointers).

Acknowledgment. The authors thank the FRAMA-C and PATHCRAWLER teams for providing the tools and support. Special thanks to François Bobot, Bernard Botella, Loïc Correnson, Pascal Cuoq, Bruno Marre, Julien Signoles and Nicky Williams for many fruitful discussions, suggestions and advice.

References

1. Ahn, K.Y., Denney, E.: Testing first-order logic axioms in program verification. In: TAP (2010)
2. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, URL: http://frama-c.com/acsl.html

3. Beyer, D., Henzinger, T., Theoduloz, G.: Program analysis with dynamic precision adjustment. In: ASE (2008)
4. 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)
5. Brucker, A.D., Wolff, B.: On theorem prover-based testing. FAC (2012)
6. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC (2012)
7. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: FM (2012)
8. Claessen, K., Svensson, H.: Finding counter examples in induction proofs. In: TAP (2008)
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., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: SAC (2013)
13. Gladisch, C.: Could we have chosen a better loop invariant or method contract? In: TAP (2009)
14. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.D.: Compositional may-must program analysis: unleashing the power of alternation. In: POPL (2010)
15. Klein, G.: From a verified kernel towards verified systems. In: APLAS (2010)
16. 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)
17. Petiot, G., Kosmatov, N., Giorgetti, A., Julliand, J.: StaDy: Deep Integration of Static and Dynamic Analysis in Frama-C. Tech. rep. (2014), http://hal.archives-ouvertes.fr/hal-00992159
18. Polikarpova, N., Furia, C.A., West, S.: To run what no one has run before: Executing an intermediate verification language. In: RV (2013)
19. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language, URL: http://frama-c.com/download/e-acsl/e-acsl.pdf