Automate where Automation Fails: Proof Strategies for Frama-C/WP

Loïc Correnson[1][0000−0001−6554−404X], Allan Blanchard[1][0000−0001−7922−4880], Adel Djoudi[2][0000−0002−8238−6490], and Nikolai Kosmatov[3][0000−0003−1557−2813]

[1] Université Paris-Saclay, CEA, List, Palaiseau, France
{loic.correnson,allan.blanchard}@cea.fr
[2] Thales Digital Identity & Security, Meudon, France
[3] Thales Research & Technology, Palaiseau, France
{adel.djoudi,nikolai.kosmatov}@thalesgroup.com

Abstract. Modern deductive verification tools succeed in automatically proving the great majority of program annotations thanks in particular to constantly evolving SMT solvers they rely on. The remaining proof goals still require interactively created proof scripts. This tool demo paper presents a new solution for an automatic creation of proof scripts in Frama-C/WP, a popular deductive verifier for C programs. The verification engineer defines a proof strategy describing several initial proof steps, from which proof scripts are automatically generated and applied. Our experiments on a large real-life industrial project confirm that the new proof strategy engine strongly facilitates the verification process by automating the creation of proof scripts, thus increasing the potential of industrial applications of deductive verification on large code bases.

Keywords: deductive verification, proof automation, interactive proof scripts, proof strategies, Frama-C.

1 Introduction

Recent years have seen many successful applications of deductive verification [7, 8]. Modern deductive verifiers manage to automatically prove the greatest number of *proof goals*, also called *proof obligations*, or *verification conditions* (VCs). This is in particular due to powerful and constantly evolving SMT solvers they rely on. The remaining unproven goals typically require some form of interactive proof: either with a proof script indicating a few initial proof steps to make the goal more suitable for an automatic prover, or a fully interactive proof in a proof assistant like Coq. The need for an interactive proof remains an important obstacle to a wider application of deductive verification on large projects.

It can be illustrated by a recent proof [6] of real-life smart card code—a JavaCard Virtual Machine (JCVM)—that was performed by Thales for the highest EAL6–EAL7 levels of Common Criteria certification[4] using Frama-C/WP [9],

[4] The EAL7 certificate delivered by the French certification body ANSSI is available at https://cyber.gouv.fr/sites/default/files/document_type/Certificat-CC-2023_45fr_0.pdf.

a popular deductive verifier for C programs. Even if a very high level of automation is achieved in that project and less than 2% of proof goals require manually created proof scripts, a significant effort is still required for the remaining goals because hundreds of properties are concerned.

Moreover, proof scripts are sensitive to the versions of the deductive verifier, of the code and the specification. Thus, proof scripts not only need to be created once for a given version of the target code, its specification and the verifier, but often have to be recreated when the code or the specification are updated, or the verifier evolves (and hence the way to generate VCs is modified). Thus, the need for manually created proof scripts for the unproven goals is seen as an important obstacle to a better maintenance of the proved code in the industrial setting.

This tool demo paper presents a new mechanism[5] for an automatic creation of proof scripts in Frama-C/WP. The verification engineer defines a proof strategy describing the alternative proof steps to be tried, from which proof scripts are automatically generated and applied. Our experiments on the JCVM verification project confirm that the new mechanism strongly facilitates the verification process, thus increasing the potential of industrial applications of deductive verification on large code bases.

The contributions of this work include a demonstration of the new mechanism for automating the creation of proof scripts in Frama-C/WP based on user-defined proof strategies, its illustration on simple examples and its evaluation on a real-life industrial project.

2 Deductive Verification with Frama-C/WP

Frama-C is an open-source, industrially mature, extensible framework for verifying C programs annotated with ACSL [2] specifications. The WP plug-in of Frama-C allows the user to prove that the C code respects the ACSL specifications using *deductive verification* [7, 8]. More precisely, WP implements an efficient variant of *weakest precondition calculus* [10], hence the name of the plug-in.

ACSL specifications, written inside special comments "`/*@...*/`", basically consist of function contracts and code annotations. Function contracts include pre-conditions (`requires` clauses) and post-conditions (`ensures` clauses), containing pure logical formulas that shall be verified respectively before and after any call to a function. The `assigns` clause specifies the possible side effects of the function on global variables and pointers received in parameters. Code annotations (e.g. `assert` clauses) contain pure logical formulas attached to a particular program point that shall be verified at each execution path going through this program point. These clauses are illustrated by the program below:

```
1 /*@
2   requires 0 ≤ x ≤ y ;
3   ensures \result == (x + y) / 2;
4   assigns \nothing;
5  */
6 int middle(int x,int y)
```

[5] publicly available on `https://git.frama-c.com/pub/frama-c/` as part of the current development version (and in the upcoming release planned for November 2023).

```
7  {
8    /*@ assert 0 ≤ y - x < MAX_INT; */
9    /*@ assert 0 ≤ x + (y-x) / 2 < MAX_INT; */
10   return x + (y - x) / 2;
11 }
```

Frama-C contains plug-ins that can generate assertions, and plug-ins that can prove assertions, or both. For instance, the RTE plug-in can generate code annotations that are sufficient for the program to never go into unspecified or undefined behaviors. The assertions in the previous example (Lines 8–9) show two of the five assertions generated by RTE on this code.

WP is able to *prove* code annotations written by the user or generated by other plug-ins. It works by using *deductive verification*: ACSL logic formula *and* C-code instructions are translated to some equivalent pure logic formulæ in a first-order logic language. Each generated formula is first simplified by a built-in solver named Qed [4] and then submitted to external provers, generally automated SMT solvers such as Alt-Ergo, Z3, CVC4 or CVC5. On the above program, WP can prove all ACSL annotations written by the user and generated by RTE:

```
1 $ frama-c -wp -wp-rte middle.c
2 [rte:annot] annotating function f
3 [wp] 8 goals scheduled
4 [wp] Proved goals:    8 / 8
5   Qed:              2
6   Alt-Ergo 2.4.2:   6 (4ms-14ms)
```

In this example, RTE generated 5 annotations and WP generated 8 formulas for proving all resulting ACSL annotations, 2 of which being proved by Qed simplification, and the remaining 6 being proved by Alt-Ergo in few milliseconds.

3 Automated vs. Interactive Proofs

In most cases, ACSL annotations are automatically proven by Qed and SMT solvers. However, sometimes an automated proof might fail for a correct formula because deductive verification is not complete in general, and WP in particular.

In such a situation, WP offers different features to complete the proofs. First, the user might help SMT solvers by introducing intermediate code annotations, hence providing proof hints and intermediate proof results. Second, the user might enter the interactive proof mode with the Frama-C graphical interface, in which the user can apply so-called *tactics* to transform a proof goal into a conjunction of several, typically simpler ones, that WP can try to prove in turn. This process can be iterated, and all the applied tactics can be saved on disk in a *proof script* file that can be replayed later from the Frama-C command line.

After some efforts, the user can thus manage to achieve full automation in proof replay for a proof campaign: all proof goals are discharged automatically by SMT solvers, possibly thanks to proof hints provided as code annotations, and possibly after applying tactics from saved proof scripts.

WP offers a large variety of tactics. Common ones include splitting over a boolean expression; brute-forcing an integer expression within a given range (detailed below); unfolding predicate or function definitions; removing hypotheses; etc. Applying tactics is simple in spirit, although it raises complex issues in

practice. Consider for instance the *Range* tactic, which can be defined as follows, where φ is the current goal, e some expression and $a \leq b$ two integer constants:

$$\mathsf{range}(\varphi, e, a, b) \equiv \bigwedge_{k \in a \ldots b} (e = k \implies \varphi) \wedge (e < a \implies \varphi) \wedge (e > b \implies \varphi)$$

Applying it on goal φ consists in replacing[6] φ by $\mathsf{range}(\varphi, e, a, b)$. It requires to have at hand the expression e and the two constants a and b. Under the graphical user interface (GUI), those arguments are selected by the user from the goal. However, bookkeeping them in a proof script is not that simple, especially if we want the proof script to resist to minor changes in the code or the specifications. WP has dedicated features to achieve this choice but up to a certain extent.

In practice, managing proof scripts during the lifetime of large projects *is* an industrial issue. On the contrary, proof hints in the form of intermediate code annotations are quite robust. However, writing code annotations by hand is tedious. On the other hand, applying tactics to decompose goals is quite efficient, and it appears that, on a given application, many pending goals are solved by applying few tactics with very similar patterns. Those observations lead us to the design of *proof strategies*.

4 Definition of Proof Strategies

This section introduces the main principles and selected features of proof strategies through illustrative examples, which can be tested using the companion artifact [5]. We refer the reader to the WP manual [1] for a detailed description.

Proof strategies are user-defined specifications for combining automated solvers with pattern-driven tactics. A proof strategy consists of a list of alternatives to be tried *in sequence* on a proof goal until success. Elementary alternatives consist in trying one or several SMT solvers with a specified timeout, or applying a tactic on a goal. Lists of alternatives can be grouped and given a (strategy) name, that can be used as an elementary alternative as well. Then, specific proof strategies can be associated to specific proof goals, functions or lemmas. For instance, the user may associate proof strategy A to every code annotation *with* name P and proof strategy B to every code annotation *without* name Q, and finally proof strategy C to other code annotations.

Proof strategies and their association to proof goals are user-written as specific ACSL extensions defined and managed by the Frama-C/WP plug-in. An overview of these annotations is provided below:

```
strategy strategyname : alternative , ... , alternative ;
proof    strategyname : target , ... , target ;
```

The `strategy` clause introduces a new proof strategy *strategyname*, whereas the `proof` clause associates it to some property *targets*, i.e. individual goals

[6] We have $\mathsf{range}(\varphi, e, a, b) \implies \varphi$, which is sufficient for the tactic to be safely applied.

or sets of goals, using the same syntax as for `frama-c` command line, which simplifies users' learning curve. As introduced above, each *alternative* might consist of:

- `\provers(p,...,p,time)` which tries the specified provers in sequence with a specified timeout.
- `\tactic(id,param...)` tries to apply the specified tactic with the associated parameter(s).
- *strategyname* or `\default` tries the specified named strategy.

Parameters for applying tactics are the most expressive but also the most complex components of proof strategies. As briefly introduced in previous section, a tactic transforms a proof goal into one or several sub-goals that are sufficient to entail the initial goal. The difficult point with tactics is that they need *parameters* to be applied. For instance, the tactic range illustrated in previous section must be applied to an *expression* and a range of two integer constants. From the Frama-C GUI, proof engineers often pick those parameters from the goal itself, according to some patterns of interest and their experience. Our proof strategy language allows proof engineers to specify those patterns, and to build tactic parameters with required values accordingly.

A trade-off between robustness and precise definition of tactic applications is an important design objective. The proposed strategy language allows a significant flexibility in choosing precise (and less robust) or more general (and more robust) patterns. The latter include '`_`' for any expression, '`..`' for any number of arguments, '`A:_`' to introduce a variable to name a subexpression and to use it in a tactic parameter or a pattern to select, etc.

Consider lemma `dn3` in Fig. 1, not proved by Alt-Ergo. It can now be proved by associating to it the following strategy (we omit surrounding `/*@...*/`):

```
1 strategy RangeThenProver:
2   \tactic ("Wp.range",
3     \pattern(is_uint8(e)),
4     \select(e),
5     \param("inf",0),\param("sup",255),
6     \children(RangeThenProver) ),
7   \prover("alt-ergo",2);
8 proof RangeThenProver: dn3;
```

The `"Wp.range"` name identifies the range tactic introduced above. This strategy looks for a variable e of type `unsigned char` (pattern `is_uint8(e)`, cf. Line 3) in the goal. If such a pattern is found in goal φ, the tactic range$(\varphi, e, 0, 255)$ is applied on φ (cf. Lines 2–5). Otherwise, the Alt-Ergo prover is applied for 2 s (Line 7). The tactic specification language also offers directives to specify which strategies shall be applied on the resulting sub-goals. Line 6 above indicates that the strategy should be applied recursively. In this way, it enumerates first the values of c, then those of d. Indeed, the recursive application to all subgoals in this case is equivalent to selecting a first variable of type `unsigned char` and enumerating its values, then for each fixed value, doing so for a second variable of type `unsigned char` (and in this case, there are no more such variables). WP takes only $\sim$1 s to automatically create the script and prove the lemma, while its manual creation would take several minutes.

Moreover, each sub-goal generated by applying a tactic has predefined names. For instance, tactic range(φ, e, a, b) generates a sub-goal named `"Lower a"` for

```
1 lemma dn3:
2   ∀ unsigned char c d;
3  (c & 0x8E) == 2 ∧
4  (c & 0x01) == 1 ∧
5  (d & 0x8F) == 0
6   ⇒ ((c+d) & 0x03) == 0x03;
```

```
7 lemma vhm_preserved{L1,L2}:
8   valid_heap_model{L1} ∧
9   mem_model_footprint_intact{L1,L2} ∧
10  \at(gNumObjs,L1) == \at(gNumObjs,L2) ∧
11  object_headers_intact{L1,L2}
12  ⇒ valid_heap_model{L2};
```

Fig. 1. Two ACSL lemmas not proved by automatic prover Alt-Ergo (with a 5 min. timeout).

```
1   strategy FastAltErgo:  \prover("alt-ergo", 1); // run Alt-Ergo for 1s
2   strategy EagerAltErgo: \prover("alt-ergo",10); // run Alt-Ergo for 10s
3   strategy UnfoldVhmThenProver:                  // Strategy with three steps:
4     FastAltErgo,                                 // 1) fast prover attempt
5     \tactic("Wp.unfold",                         // 2) if unproved, unfold
6       \pattern(P_valid_heap_model((..))),        // predicate valid_heap_model
7       \children(UnfoldVhmThenProver) ),          // and apply itself recursively
8     EagerAltErgo;                                // 3) longer prover attempt
9   proof UnfoldVhmThenProver: vhm_preserved;      // Associate strategy to goal
```

Fig. 2. Strategies to automatically create a proof script for lemma `vhm_preserved` of Fig. 1.

case $e < a$, `"Upper b"` for case $e > b$ and `"Value k"` for each case $e = k$ with $k \in a..b$. The user can then specify which strategy shall be used for each generated sub-goal. More detailed documentation can be found in the WP manual [1].

The second lemma in Fig. 1 comes from the example in [6] on the proof of the JCVM. It was not proved by the Alt-Ergo prover [3] (used in that work) and required a proof script. Basically, lemma `vhm_preserved` deduces predicate `valid_heap_model` at label (i.e. program point) `L2` from the same predicate at label `L1` (Lines 8, 12 in Fig. 1) if additional conditions are satisfied: the variables defining the memory state and the number of allocated objects do not change between labels `L1` and `L2` (Lines 9–10), and the headers of the allocated objects (indicating object owner, object size, etc.) do not change between labels[7] `L1` and `L2` either (Line 11). Such lemmas are useful in large verification projects with lots of variables: by showing the preservation of values only for a few variables between two program points, this lemma allows the tool to deduce the predicate of interest at a new program point. The exact definition of predicates is not necessary to follow the present paper (and can be found in [6]).

With the presented extension of WP, the verification engineer can define a strategy `UnfoldVhmThenProver` (see Fig. 2) indicating which proof steps should be applied in order to achieve the proof. First, it calls the Alt-Ergo prover to check whether the goal can be proved with a short timeout (cf. Lines 4 and 1). If not, Lines 5–7 provide another alternative: to apply the *Unfold* tactic to unfold the definition of predicate `valid_heap_model` (in any part of the goal and with any number of arguments). Line 7 indicates that after a successful unfolding, the same strategy should be applied iteratively on the resulting sub-

[7] Labels `L1` and `L2` can be C labels or predefined ACSL labels [2]. While labels are not directly preserved in the resulting VCs, the variables at those labels typically have different names, so it is still possible to match the corresponding values.

goals (children). Finally, Line 8 indicates that if the unfolding alternative cannot be applied anymore, a longer prover attempt is tried (cf. Line 2). This strategy allows WP to prove the target lemma in ∼2 s.

5 Industrial Evaluation and Conclusion

We have applied the presented extension of Frama-C/WP to the proof of the real-life JCVM code[8] (with 8,000+ lines of C and 30,000+ lines of ACSL) at Thales. The complete proof for 85,000 goals using Alt-Ergo with a 250s timeout requires 800+ proof scripts. The new tool saves a very significant effort: after a manual creation of strategies (∼2 days), *WP automatically produces more than 50% of the required scripts, whose manual creation would take ∼1 person-month.* This effort is estimated by the authors based on the experience of manual proof script creation in the industrial context over four years. In this experiment, the strategies are created by the same verification engineers who have previously created proof scripts. The same strategy is often able to successfully prove several dozens of proof goals, which confirms the reusability of strategies for multiple goals.

We summarize our experiment as a two-step workflow. First, the verification engineer creates proof strategies. Frequently used tactics (*Unfold*, *Split*, etc.) may be used as an initial guess with a large timeout in order to maximize proof automation. If some goals are still not proved, the engineer uses their experience to propose new ones, tuned to failed goals. The generated scripts are then saved for a proof replay session. Second, the engineer optimizes the strategies, e.g. by optimizing the script generation or replay time. The creation of strategies requires similar skills as for the creation of proof scripts.

We believe that an even greater number of proof scripts can be generated from strategies, which will strongly facilitate industrial verification. Future steps include identification and implementation of further strategy features, and their rigorous evaluation on various industrial projects. A detailed analysis of the reasons why some goals remained unproven in our experiment on the JCVM code will provide a better understanding of the nature of those goals and the required additional strategies. Finally, an evaluation of the usability of strategies by various categories of users (e.g. verification engineers who are not familiar with the target project or with proof scripts in Frama-C) is another future work perspective.

Acknowledgment. Part of this work was supported by ANR (grants ANR-22-CE39-0014, ANR-22-CE25-0018). We thank Nathan Koskas de Diego and Virgile Prevosto for many fruitful discussions and preliminary investigations that encouraged this work, as well as the anonymous referees for helpful comments.

[8] Being highly security-critical, this code cannot be shared or included in an artifact.

References

1. Baudin, P., Bobot, F., Correnson, L., Dargaye, Z., Blanchard, A.: WP Plug-in Manual (2023), https://frama-c.com/download/frama-c-wp-manual.pdf
2. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language (2021), https://www.frama-c.com/download/acsl.pdf
3. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: International Workshop on Satisfiability Modulo Theories(SMT 2018). https://hal.inria.fr/hal-01960203
4. Correnson, L.: Qed. Computing what remains to be proved. In: NASA Formal Methods Symp. (NFM 2014). LNCS, vol. 8430, pp. 215–229. Springer (2014)
5. Correnson, L., Blanchard, A., Djoudi, A., Kosmatov, N.: Automate where automation fails: Proof strategies for Frama-C/WP. Companion artifact for the paper submitted to TACAS 2024. (Nov 2023), https://doi.org/10.5281/zenodo.10047833
6. Djoudi, A., Hána, M., Kosmatov, N.: Formal verification of a JavaCard virtual machine with Frama-C. In: the 24th Int. Symp. on Formal Methods (FM 2021). vol. 13047, pp. 427–444. Springer (2021)
7. Filliâtre, J.: Deductive Software Verification. International Journal on Software Tools for Technology Transfer 13(5), 397–403 (2011)
8. Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: Computing and Software Science – State of the Art and Perspectives, LNCS, vol. 10000, pp. 345–373. Springer (2019)
9. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. pp. 1–37 (2015)
10. Leino, K.R.M.: Efficient Weakest Preconditions. Information Processing Letters 93(6), 281–288 (2005)