No Smoke without Fire: Detecting Specification Inconsistencies with Frama-C/WP

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

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

Abstract. Deductive verification provides a proof that, under the provided pre-conditions, each terminating execution of a given function satisfies the stated post-conditions. In general, pre- and post-conditions are expressed in a logical specification language and typically rely on theories including abstract definitions, axioms and lemmas. As they are written by humans, errors may be introduced into specifications. Some errors can be detected when the proof fails, but sometimes, they remain unnoticed due to misleading proofs: most of the program may become dead code under the provided pre-conditions, or the proof may succeed because of inconsistencies in hypotheses and axioms. In this tool paper, we explore how to detect such unwanted situations by using deductive verification techniques and describe the smoke test mechanism in Frama-C/WP, a popular deductive verifier for C code. We show that, while the intuitive idea is simple, making it practical requires optimizations to scale up, and report on experiments with critical industrial code. Although our method is based on proof techniques, it is not complete and is similar to testing. In the end, can we ever be sure that our programs are proved well enough?

1 Introduction

Deductive verification [14] is aimed at proving that a given program respects its formal specification. The specification is an essential ingredient of the process: if the specification is wrong, the whole verification is of no interest. Similarly, too strong specification requirements might silently eliminate large portions of the code, dramatically reducing the effective scope of the verification claim.

The purpose of this paper is to describe a mechanism for tracking inconsistent specifications and unwanted dead code that we implemented in Frama-C/WP [6], a mature and popular deductive verification tool for C programs annotated in ACSL specification language [7]. It is based on so-called *smoke tests*, which are specific annotations generated by WP such that their proof would demonstrate the presence of such unwanted situations.[4] Such annotations typically assert `false` at specific program points. Then, an unwanted situation can be revealed if the corresponding assertion of `false` is proved. Such an assertion of `false` is provable only if the corresponding program point is not reachable by a *valid execution path*[5], i.e. an execution path leading to this program point such that all annotations met on this path are satisfied. In this way, detecting inconsistencies can be reduced to the detection of program points that are unreachable (or dead) *from the point of view* of a deductive verification tool. Notice that this unreachability property is larger than the usual notion of dead code — when a program point is not reachable by *any* execution path. In the following sections, when talking from the point of view of WP, unreachable (dead) code will be understood in this broader sense.

This mechanism is potentially capable to detect various kinds of inconsistencies and dead code — but only those explicitly tracked, and only if the prover manages to prove the corresponding annotation. By essence, this approach amounts to testing whether the tracked unwanted situations occur or not, without any guarantee to detect them, nor any guarantee that there is no *other* sources of inconsistency in user specifications.

This tool paper describes the design and implementation of the smoke tests feature of Frama-C/WP, optimizations to make this mechanism more scalable and more practical, and presents some evaluation results on industrial code.

2 Tracked Inconsistencies

Specifications can contain various kinds of errors. They are often related to inconsistencies among the hypotheses that form the context of the proof. The context typically includes pre-conditions of the entry point function, global variables and global properties such as axioms. Sometimes, the user configuration of the deductive verification engine can also introduce additional hypotheses in the proof context, for instance, global assumptions regarding the memory model or the absence of specific kinds of runtime errors. These hypotheses might also interfere with the specification or with the code in a misleading way.

In this section, we explore different situations where inconsistencies in specifications might remain unnoticed during modular deductive verification. Most examples presented in the article are simple enough to be fully and trivially verified by Frama-C/WP [6,17] with a simple command `frama-c myfile.c -wp`. However, all the presented examples are actually *doomed* by specification errors. We also show that all these situations are trivially revealed by WP thanks to smoke tests by simply adding the option `-wp-smoke-tests` to the command.[6]

[4] some of which can be justified in some specific cases (e.g. by a partial verification scope, architecture assumptions), but should not remain unnoticed (cf. Sect. 2.7).

[5] See Sect. 3.3 for a formal definition of a valid execution path.

[6] All examples were run with Frama-C 29 (Copper) and Alt-Ergo 2.5.3 public releases. Terminal outputs are shortened or omitted, and only focus on relevant parts.

2.1 Pre-condition Errors

Inconsistent pre-conditions can be revealed as they can never be proved at their call sites. Indeed, WP checks that pre-conditions are valid at every call site. But for root functions in the call graph, there is no way to check this property. For instance, consider the following program (say, in file `requires.c`):

```
/*@ requires x < 0 ;
    requires x > 0 ;
    ensures \result == 42 ; */
int f(int x){ return x ; }
```

The post-condition is obviously wrong: there is no reason for the result of function `f` to be equal to 42. However, this program is fully verified by WP:

```
> frama-c requires.c -wp
[wp] Proved goals:    3 / 3
```

Indeed, the two pre-conditions of `f` are mutually exclusive: it is not possible to call it with valid pre-conditions. Hence, the program is valid because there is *no* valid call context for it. Fortunately, WP with smoke tests trivially detects this misleading specification:

```
> frama-c requires.c -wp -wp-smoke-tests
[wp] [Failed] (Doomed) typed_f_wp_smoke_default_requires (Qed)
[wp] requires.c:4: Warning: Failed smoke-test
[wp] Proved goals:    3 / 4
[wp] Smoke Tests:     0 / 1
```

When a function's contract is split into different ACSL behaviors, inconsistencies may also occur among a behavior's assumptions, or between the default behavior pre-conditions and the assumptions of a specific behavior. For instance, consider the following program (file `assumes.c`):

```
/*@ requires x < 0 ;
    behavior Positive:
      assumes x > 0 ;
      ensures \result == 42 ; */
int f(int x){ return x ; }
```

Here also, the program is fully verified by WP, although its `Positive` behavior is doomed since its assumption is inconsistent with the function pre-condition, a situation trivially detected by WP smoke tests:

```
> frama-c assumes.c -wp
[wp] Proved goals:    3 / 3

> frama-c assumes.c -wp -wp-smoke-tests
[wp] [Failed] (Doomed) typed_f_wp_smoke_Positive_assumes (Qed)
[wp] assumes.c:5: Warning: Failed smoke-test
[wp] Proved goals:    4 / 5
[wp] Smoke Tests:     1 / 2
```

Detecting errors at the level of pre-conditions naturally extends to inconsistencies with respect to the global context. Consider the example (file `axioms.c`):

```
/*@ axiomatic Ax {
      logic integer A reads \nothing ;
      axiom Positive: A >= 0 ;
    } */
```

```
/*@ requires A < 0 ;
    ensures \result == 42 ; */
int f(void){ return 0 ; }
```

The ACSL axiomatic block introduces logical variable `A` assumed to be positive. This function is also proven valid by WP, while, contrary to its post-condition, it never returns 42. Indeed, its pre-condition is doomed: it is inconsistent with the global assumption on logic parameter `A`, as revealed by WP smoke tests:

```
> frama-c axioms.c -wp -wp-smoke-tests
[wp] [Failed] (Doomed) typed_f_wp_smoke_default_requires (Alt-Ergo 14ms)
[wp] axioms.c:8: Warning: Failed smoke-test
[wp] Proved goals:    3 / 4
[wp] Smoke Tests:     0 / 1
```

2.2 Post-condition Errors

Post-conditions of a function are verified by WP with respect to the function body and its pre-conditions. Now consider external functions that only have a specification but no code. Their post-conditions cannot be verified, hence, they might silently introduce inconsistencies with respect to their calling context.

Consider for instance the following program (file `ensures.c`), where function `main` is indeed wrong, since it always returns 1 according to the post-condition of `incr` function, and hence cannot return 42 as claimed by its specification:

```
//@ ensures \result == 42;
int main(void){
  int x = 0 ;
  incr(&x) ;
  return x ;
}

/*@ assigns \nothing ;
    ensures *p == \old(*p) + 1 ; */
void incr(int* p);
```

However, this program is fully proved by WP. Here, the problem comes from external function `incr` whose post-condition is doomed by an inconsistency with its assigns clause: `*p` and `\old(*p)` shall be indeed equal since `incr` is assumed to have no side-effects. Its correct assigns clause would be `assigns *p`. Such an inconsistency is trivially detected by WP smoke tests:

```
> frama-c ensures.c -wp -wp-smoke-tests
[wp] [Failed] (Doomed) typed_incr_wp_smoke_dead_call_s2 (Qed)
[wp] ensures.c:4: Warning: Failed smoke-test
[wp] [Failed] (Doomed) typed_main_wp_smoke_dead_code_s4 (Qed)
[wp] ensures.c:5: Warning: Failed smoke-test
[wp] Proved goals:    3 / 5
[wp] Smoke Tests:     0 / 2
```

In this case, the doomed post-condition leads to detecting two issues. First, because the post-condition of function `incr` is inconsistent with its assigns clause. Second, because the code in `main` *after* calling `incr(&x)` becomes unreachable (or dead) code (from the point of view of WP, cf. Sect. 1). Recall that for a function call, its pre-conditions are verified while its post-conditions are assumed. Hence, inconsistent post-conditions introduce a `false` hypothesis into the proof context for any property after the call.

2.3 Dead Code Errors

Dead code generally stays outside the eyes of deductive verification: any properties for infeasible execution paths are silently proved valid because their associated branching conditions are always false. However, as illustrated with doomed post-conditions in the previous section, some execution paths might be silently discarded by doomed specifications. Consider the following code (file `dead.c`):

```
/*@ requires \valid(x);
    ensures \result == *x; */
int read(int *x) {
  if (!x) return 42;
  return *x;
}
```

The program is proven valid by WP. The if statement makes the function return 42 when it receives a `NULL` pointer in argument. Although it might be the expected function behavior, this is *not* reflected by the specification, which assumes a valid (hence, non-null) pointer, that makes unreachable the statement `return 42`. Such an unwanted doomed code is detected by WP smoke tests:

```
> frama-c dead.c -wp -wp-smoke-tests
[wp] [Failed] (Doomed) typed_read_wp_smoke_dead_code_s2 (Qed)
[wp] dead.c:4: Warning: Failed smoke-test
[wp] Proved goals:    5 / 6
[wp] Smoke Tests:     2 / 3
```

Dead code is generally unwanted for software quality, so even if the function above is correct according to ACSL, it is still important to track dead code.

2.4 Non-Terminating Loops and Other Loop Errors

Non-terminating loops can be left undetected by deductive verification, unless explicit termination proofs are required by the user through `loop variant` and `terminates` clauses. By lack of such specifications, which are optional in ACSL and Frama-C, buggy non-terminating code might be silently verified.[7]

Consider for instance the following program (file `loop.c`):

```
/*@ requires x >= 0;
    ensures \result == 0;
*/
int f(int x) {
  /*@
    loop invariant x >= 0;
    loop assigns x;
  */
  while (x > 0) ;
  return 42;
}
```

This program seems to incorrectly return 42, contrary to its specification. However, the program is fully verified by WP. Actually, there are two bugs in the code: the loop is non-terminating (making the code after it unreachable), and the

[7] Since the Frama-C 29 (Copper) release, terminating annotations are generated by default. In this example, it has been turned off by `-generated-spec-custom terminates:skip` command line option.

return statement is incorrect. Both problems are silently ignored by deductive verification. Fortunately, WP smoke tests detect them:

```
> frama-c loop.c -wp -wp-smoke-tests
[wp] [Failed] (Doomed) typed_f_wp_smoke_dead_code_s6 (Qed)
[wp] loop.c:10: Warning: Failed smoke-test
[wp] Proved goals:    7 / 8
[wp] Smoke Tests:     2 / 3
```

A possible corrected version of the code, where the loop terminates and the returned value is correct, is the following:

```
/*@ requires x >= 0;
    ensures \result == 0;
*/
int f(int x) {
  /*@
    loop invariant x >= 0;
    loop assigns x;
  */
  while (x > 0)
    { x--; }
  return x;
}
```

Another common issue with loops occurs when there is an inconsistency in loop invariants, or between them and the current proof context: this makes the entire loop provably dead code. Moreover, it is often the case where a buggy, non-proved loop specification introduces such an inconsistency. Hence, detecting inconsistent loop invariants early might help to fix buggy loop specifications. WP smoke tests are also able to detect inconsistencies in loop invariants.

2.5 Exiting Errors

Functions that might exit (by calling `exit` function) typically have explicit `false` post-conditions, that are indeed good candidates for introducing unwanted inconsistencies in proof contexts.

Consider for instance the following program (file `exit.c`):

```
/*@ ensures \false ;
    exits \true ; */
void exit(void);

/*@ ensures \result == 0; */
int main()
{
  exit();
  return 42;
}
```

This program is verified by WP[8], although it contains a buggy statement returning 42, which is silently ignored after the `exit()` call. As expected, WP smoke tests detect such a dead code:

```
> frama-c exit.c -wp -wp-smoke-tests
[wp] [Failed] (Doomed) typed_main_wp_smoke_dead_code_s2 (Qed)
[wp] exit.c:8: Warning: Failed smoke-test
[wp] Proved goals:    2 / 3
```

[8] To reproduce this example with Frama-C 29 (Copper), clause generation shall be customized with `-generated-spec-custom terminates:skip,exits:skip`.

The main function and its specification are actually completely wrong: it never returns a value, always exits and its `return` statement is never executed.

2.6 Avoiding Redundant Alarms

As illustrated above, inconsistencies are closely related to unreachable code. However, unreachability of a program point extends to a sequence of the following instructions, provided there is no goto leading into the middle of the sequence.

From a user perspective, we should avoid reporting as unreachable all instructions along an entire *sequence* of instructions. Similarly, for performance reasons, when WP has put some effort in deciding whether a given program point is reachable or not, it is not useful to spend the effort again on subsequent instructions, unless they might become reachable through another execution path. Hence, it is important to focus on *necessary* program points for generating smoke tests.

2.7 Avoiding Spurious Alarms

In the case of dead code, alarms might be raised by WP smoke tests whereas the developer knows that the code is dead for *good* reasons (such as verification scope or architecture restrictions, countermeasures against fault injection attacks, etc.). In this case, the developer may provide an `assert \false` annotation [9] to tell WP that a given branch is intentionally dead and that it should not warn about it.

Consider for instance the following program (file `div.c`):

```
/*@ requires b > 0;
    ensures \result == a/b;
    assigns \nothing; */
int div(int a, int b) {
  if (b==0) {
    /*@ assert \false; */
    return 0;
  }
  return a/b;
}
```

The defensive code to avoid the division by zero is actually not needed thanks to the pre-condition. Normally, WP smoke tests would detect dead code. However, in this example, the user provides `assert \false` annotation indicating that this dead code is expected.[10] The program is fully verified by WP, including the assert annotation indicating the dead code, annotations generated to prove the absence of runtime errors and remaining smoke tests:

```
> frama-c div.c -wp -wp-rte -wp-smoke-tests
[wp] Proved goals:   10 / 10
[wp] Smoke Tests:     2 / 2
```

[9] ACSL has two assertion clauses, `assert p;` and `check p;`, whose only difference is that with the former, property `p` is verified and preserved in the proof context, while with the latter, `p` is just verified.

[10] We do not debate here whether this is a good practice or not.

3 Design and Implementation

As illustrated in Sect. 2, misleading or inconsistent specifications can be revealed by the fact that specific program points are unreachable. In the context of deductive verification, this can be implemented by checking that `false` is provable at the considered program point. Hence, tracking inconsistencies can be implemented by inserting `false` annotations at the right places. Those annotations will play the role of *Smoke* detectors that would reveal misleading specifications, if any. We discuss later the incompleteness of the method, but for now, it suffices to state that a smoke test expresses the unreachability of some program point.

Let us also recall that, from the ACSL point of view, any code annotation attached to unreachable program points is *de facto* valid because there will be no execution path to invalidate it. However, it is important for the end users to have a feedback regarding the *source* of a program point's unreachability. For instance, as pointed out in Sect. 2.7, we shall take into account user-defined intended dead code. We shall also take into account that the property of being dead code may propagate through the program control flow (cf. Sect. 2.6).

Hence, reachability, explicit dead code, unreachable annotations and smoke tests are closely related with each other and need specific feedback to users.

In this section, we provide a method to determine an optimized collection of program points that shall be equipped with a smoke test. First, we introduce a reachability analysis dedicated to smoke tests (Sect. 3.1). For an efficient generation of smoke tests, we also introduce the concept of nodes protected by smoke tests and how this concept propagates through the control flow graph (Sect. 3.2). Then, we illustrate how we can derive the generation of smoke tests from standard Weakest-Precondition Calculus of the WP plug-in (Sect. 3.3). We finally discuss the completeness of the method (Sect. 3.4) and how to provide users with accurate feedback (Sect. 3.5) for smoke test results.

3.1 Reachability Analysis

Although reachability is a standard problem in static analysis and compilation techniques, we need some adaptations to take redundancy and spurious alarms into account.

We use a Control Flow Graph (CFG) of functions. As illustrated in Fig. 1, the CFG is made of program points (statements, Stmt) and edges labeled with instructions (Instr). We assume here that the CFG is *normalized*, that is, the large variety of C instructions and statements, such as loops, switches, and conditional operators are decomposed into branching statements and elementary instructions. Moreover, expressions with side effects are assumed to be decomposed into assignments $l = e$ of a side-effect-free expression $e \in \mathsf{Expr}$ to an l-value $l \in \mathsf{Lval}$ and function calls $l = e(\overline{e})$. Expressions and l-values are not detailed here since this is mostly irrelevant for generating smoke tests. Jumping C instructions such as `goto`, `break` and `continue` are represented by transitions labeled with the `skip` instruction, which has no effect at all.

Program points:	$a, b \in \mathsf{Stmt}$
Instructions:	$k \in \mathsf{Instr} ::= l = e \mid l = e(\overline{e}) \mid \mathtt{if}(e) \mid \mathtt{skip} \mid \mathtt{return}(e)$
Control Flow Graph:	$\{a \xrightarrow{k} b, \ldots\} \in \mathsf{Cfg}$
Entry point:	$\mathsf{entry} \in \mathsf{Stmt}$
Predecessors:	$\mathsf{pred}(b) \triangleq \{\, a \mid a \xrightarrow{k} b \in \mathsf{Cfg} \,\}$

Fig. 1. Normalized Control Flow Graph (CFG) definition

We consider sequential C programs (without threads), hence the CFG is deterministic, which means that a branching node a has a complete and disjoint collection of transitions of the form $a \xrightarrow{\mathtt{if}(e)} b$. A non-branching node a has exactly *one* outcoming transition $a \xrightarrow{k} b$, where k is an assignment $l = e$, a function call $l = e(\overline{e})$, a $\mathtt{skip}$ or a $\mathtt{return}(e)$ instruction. Moreover, since we need to distinguish the program point right after each call, we further decompose a call $l = e(\overline{e})$ into the sequence $\{\mathtt{r} = e(\overline{e}); l = \mathtt{r}\}$ with an intermediate temporary variable $\mathtt{r}$ to store the result. We can thus define the following classification of node categories — with the first category for a branching node and the others for non-branching ones — that will be very handy when defining smoke tests:

- $\mathsf{branch}(a)$ when a has multiple outcoming instructions $\mathtt{if}(e)$;
- $\mathsf{skip}(a)$ when a has a unique outcoming instruction $\mathtt{skip}$;
- $\mathsf{call}(a)$ when a has a unique outcoming instruction $\mathtt{r} = e(\overline{e})$;
- $\mathsf{write}(a)$ when a has a unique outcoming instruction $l = e$;
- $\mathsf{return}(a)$ when a has a unique outcoming instruction $\mathtt{return}(e)$.

Nodes after a $\mathtt{return}(e)$ transition have no outcoming transition, and for a given function, there is a unique entry point denoted by $\mathsf{entry} \in \mathsf{Stmt}$.

Standard reachability analysis can be formally defined as the smallest set $R \subset \mathsf{Stmt}$ of program points satisfying the following equation:

$$\mathsf{entry} \in R \quad \wedge \quad \forall (a \xrightarrow{k} b) \in \mathsf{Cfg},\ a \in R \implies b \in R.$$

In practice, a well-known efficient algorithm to compute reachable program points uses a memoization table and the following lazy definition[11]:

$$R(n) \triangleq \quad n = \mathsf{entry} \ \vee\ \exists p \in \mathsf{pred}(n),\ R(p),$$

where $\mathsf{pred}(n)$ is the set of predecessors p of n in the CFG defined in Fig. 1. For WP smoke tests, we also need to account for user-defined dead code annotations, as stated in Sect. 2.7. To this end, let us introduce predicate $\mathsf{dead}(a)$ that

[11] The memoization table is initialized to $\mathtt{false}$ before recursive calls to cut off cycles in the CFG. All recursive definitions presented use this technique. This is completely standard and not represented here for simplicity.

checks if a program point a has been annotated with an explicit `assert \false` annotation[12]. Hence, we introduce the following lazy definition for reachability:

$$\mathsf{reachable}(n) \triangleq \neg\,\mathsf{dead}(n) \ \wedge\ (\ n = \mathsf{entry}\ \vee\ \exists p \in \mathsf{pred}(n),\ \mathsf{reachable}(p)\ ).$$

Indeed, $\mathsf{dead}(a)$ corresponds to a program point that is explicitly marked to be unreachable by the user. The associated annotation `assert \false` is expected to be eventually proved as part of the user specification. In this case, such program points will be provably unreachable.

3.2 Protected and Protecting Nodes

In theory, it suffices to equip all reachable nodes (according to the above definition) with a smoke test to detect any specification inconsistency. However, doing so will generate far too many smoke tests. As a simple example, consider a block consisting of a sequence of write instructions, without any call or jump from/into the block: then all instructions in the block have the same reachability status, and a smoke test is necessary only for the first instruction. All the following instructions of the sequence will be *protected*: the smoke test for the first instruction will be sufficient to detect a potential unreachability (from the point of view of WP, cf. Sect. 1) for any of the following instructions of the sequence, hence they do not need to be equipped with smoke tests. Of course, the situation becomes more complex in the general case.

We give here an overview of different heuristics that we have implemented in order to efficiently remove most redundant smoke tests. This section is relatively technical and might be skipped by readers who are not interested in the underlying implementation details.

Strictly speaking, computing the optimal set of protected nodes is hard and would require to compute the *dominators* of nodes throughout the CFG, which is complex and costly. Instead, we consider a lightweight heuristic based on local transitions and a weak definition of protecting nodes. Although non-optimal, our approach appears to be sufficient in practice and avoids most redundancies.

Let us start with the actual implementation, which is based on three mutually inductive predicates, formally defined in Fig. 3.2, that we can briefly summarize as follows:

- $\mathsf{smoking}(a)$ indicates that program point a shall be finally equipped with a smoke test;
- $\mathsf{protected}(a)$ indicates that program point a *does not require* additional protection (by a smoke test) because its predecessor nodes in the CFG are already sufficiently (directly or indirectly) protected by smoke tests, or because a is provably unreachable according to the user-provided specification.
- $\mathsf{protecting}(a)$ is the dual notion and indicates that program point a is already protected by smoke tests so that its immediate successors are *also* protected by a.

$$\begin{aligned}
\mathsf{smoking}(a) &\triangleq \mathsf{entry}(a) \lor (\ \mathsf{reachable}(a) \land \neg\, \mathsf{protected}(a)\ ),\\
\mathsf{protected}(a) &\triangleq \mathsf{entry}(a) \lor \mathsf{dead}(a) \lor (\ \forall p \in \mathsf{pred}(a),\ \mathsf{protecting}(p)\ ),\\
\mathsf{protecting}(a) &\triangleq \mathsf{entry}(a) \lor \mathsf{dead}(a) \lor \mathsf{return}(a)\\
&\quad \lor (\ \mathsf{write}(a) \land \mathsf{protected}(a)\ )\\
&\quad \lor (\ \mathsf{skip}(a) \land \mathsf{protected}(a)\ )\\
&\quad \lor (\ \mathsf{branch}(a) \land \neg\, \mathsf{reachable}(a)\ ).
\end{aligned}$$

Fig. 2. Characterization of smoking nodes

Typically, consider a single write instruction $l = e$ from node a to node b. Assume that node a is already equipped with a smoke test and that b has no other predecessor than a. Then a is protecting b and, dually, b is protected by a. Indeed, they have the same reachability status, hence it is not necessary to equip b with a smoke test.

Not all nodes are *protecting*. Typically, consider now that a is a branching node, already equipped with a smoke test, having two conditional outcomes: $\mathsf{if}(e)$ to node b and $\mathsf{if}(\neg e)$ to node c. In this case, nodes b and c are *not* protected by a, since condition e or condition $\neg e$ might be inconsistent with previous conditions or user specifications from the above context. Node b does not necessarily have the same reachability status as node a, and neither does node c. Dually, we shall consider that node a is *not* protecting its successors. Both nodes b and c will then need to be equipped with a smoke test, even if a is already protected.

Let us now comment on our definitions of predicates smoking, protected and protecting given in Fig. 3.2.

Smoking Nodes. Predicate $\mathsf{smoking}(a)$ characterizes all nodes a that require a smoke test. The function entry point and loop entry points, characterized by predicate $\mathsf{entry}(a)$, must always be equipped with smoke tests in order to detect inconsistencies in function pre-conditions and loop contracts. Then, only *reachable* nodes a that are *not* already protected by other means shall be equipped with a new smoke test. This explains the definition.

Protected Nodes. Predicate $\mathsf{protected}(a)$ notably includes nodes that are *de facto* equipped by a smoke test or that are provably unreachable by user specifications. Moreover, if all predecessors of node a are protecting nodes, then a will be sufficiently protected and no additional smoke test will be needed. This explains the definition.

Protecting Nodes. Predicate $\mathsf{protecing}(a)$ designates nodes a that propagate their protection to their successors. All nodes that are *de facto* equipped by smoke tests or user-indicated as unreachable will be protecting. Non-branching nodes

[12] Actually, any code annotation or loop invariant syntactically equivalent to `false` is accepted by WP as well.

(write and skip instructions) will be protecting as soon as they are already protected. Branching nodes are protecting only when they are unreachable.

This heuristic appears to be pretty efficient in practice: we have observed quite few redundant smoke tests, and only for very complex control flow graphs with weird inter-block gotos.

3.3 Weakest-Precondition Calculus for Smoke Tests

We are now ready to (semi-formally) define the generation of WP smoke tests to address all potential unwanted situations described in Sect. 2. This section is more technical: it assumes that the reader is familiar with the weakest precondition calculus and can be skipped by the readers who are not interested in implementation details.

As introduced above, a smoke test at program point $a \in \mathsf{Stmt}$ shall detect an inconsistency in the proof context that makes a to be unreachable through valid execution paths. Hence, such a smoke test amounts to checking the satisfiability or unsatisfiability of the so-called *path predicate* over a path towards the corresponding node.

Indeed, the Frama-C/WP deductive verification engine is based on an efficient *Weakest Precondition Calculus* that relies on unsatisfiability of formulas that also involve path predicates. Hence, we want to adapt our WP calculus in order to efficiently generate smoke tests on-the-fly.

First, recall the main objective of WP. Given an ACSL assertion P attached to program point $a \in \mathsf{Stmt}$, we want to prove that predicate P holds for every memory state m reached by a valid execution path going through node a.

More generally, every ACSL annotation [5] can be decomposed into a collection of predicates attached to the relevant program points in the CFG. Hence, we denote by $\mathsf{asserts}(a)$ the set of ACSL predicates P to be proved at node a, and $\mathsf{assumes}(a)$ the set of ACSL predicates P assumed at node a.

More formally, let us consider an execution path $\sigma = (a_i, m_i)_{i<n}$ where $a_i \in \mathsf{Stmt}$ are program points and $m_i \in \mathsf{Mem}$ are memory states $(0 \leq i < n)$. Execution paths shall be consistent with the semantics of C programs and the CFG of executed functions, as defined in Fig. 3. Moreover, *valid* execution paths shall also satisfy ACSL annotations of the program, that is, $[\![P]\!](m_i)$ holds for every $P \in \mathsf{asserts}(a_i)$ at every step i (with $0 \leq i < n$), provided that $[\![H]\!](m_j)$ holds for every $H \in \mathsf{assumes}(a_j)$ at every preceding step j (with $0 \leq j < i$). We write $\mathsf{valid}(\sigma)$ for an execution path σ that satisfies both the program semantics and ACSL annotations. Notice that we consider below execution paths from arbitrary nodes, not only those originating from the function entry point.

The Frama-C/WP engine actually computes a collection of proof obligations for each ACSL annotation. The intuitive meaning is that, provided all the proof obligations are valid, then all execution paths are valid. We now expose the classical definition of WP and extend it to generating proof obligations.

A proof obligation $A : \phi \in \Omega$ is a logical formula ϕ labelled with its originating ACSL annotation A. For the sake of simplicity, we consider collections of

$$\begin{array}{rl}
\text{Memory states:} & m \in \mathsf{Mem} \\
\text{Logical formula:} & \varphi,\ \phi,\ (\varphi \Rightarrow \phi), \ldots \in \mathsf{Prop} \\
\mathsf{C}\text{ instruction semantics:} & [\![\, k \,]\!](m, m') \in \mathsf{Prop} \\
\mathsf{ACSL}\text{ predicate semantics:} & [\![\, P \,]\!](m) \in \mathsf{Prop} \\
\mathsf{ACSL}\text{ predicate collection:} & [\![\, P_1, \ldots, P_n \,]\!](m) \ \equiv\ \forall i,\ [\![\, P_i \,]\!](m) \\
\text{Execution paths} & \sigma = (a_i, m_i)_{i<n} \\
\text{Program semantics:} & \forall i,\ \exists k,\ a_i \xrightarrow{k} a_{i+1} \in \mathsf{Cfg}\ \wedge\ [\![\, k \,]\!](m_i, m_{i+1}) \\
\text{Valid execution paths :} & \forall i,\ (\forall j < i, [\![\, \mathsf{assumes}(a_j) \,]\!](m_j)) \implies [\![\, \mathsf{asserts}(a_i) \,]\!](m_i)
\end{array}$$

Fig. 3. Semantics of C programs annotated in ACSL

$$\begin{array}{rcl}
\mathsf{wp}(a) & \stackrel{\triangle}{=} & \mathsf{wpGoals}(a)\ \cup\ \mathsf{wpCfg}(a) \\
\mathsf{wpGoals}(a) & \stackrel{\triangle}{=} & \bigcup \{\, A : [\![\, P \,]\!](m_a) \mid A : P \in \mathsf{asserts}(a) \,\} \\
\mathsf{wpCfg}(a) & \stackrel{\triangle}{=} & \bigcup \{\, A : \mathsf{wpPath}(a, k, b, \varphi) \mid a \xrightarrow{k} b \in \mathsf{Cfg},\ A : \varphi \in \mathsf{wp}(b) \,\} \\
\mathsf{wpPath}(a, k, b, \varphi) & \stackrel{\triangle}{=} & [\![\, \mathsf{assumes}(a) \,]\!](m_a) \Rightarrow [\![\, k \,]\!](m_a, m_b) \Rightarrow \varphi
\end{array}$$

Fig. 4. Weakest pre-condition calculus

proof obligations represented by sets, although their efficient implementation in WP is completely different.

We introduce a collection of free memory state variables $(m_a)_{a \in \mathsf{Stmt}}$ indexed by program points. WP generates for each node a a collection of partial proof obligations formally defined in Fig. 4. Informally, from the bottom-up:

– $\mathsf{wpGoals}(a)$ ensures that all asserted ACSL predicates at node a are valid in memory state m_a;
– $\mathsf{wpPath}(a, k, b, \varphi)$ ensures that, if property φ holds for execution paths originating from b, then $\mathsf{wpPath}(a, k, b, \varphi)$ holds for execution paths going through transition $a \xrightarrow{k} b$. To achieve that, the assumed assertions at point a on memory state m_a are added to the proof context, together with the instruction semantics $[\![\, k \,]\!](m_a, m_b)$. Then, $\mathsf{wpCfg}(a)$ aggregates $\mathsf{wpPath}(a, k, b, \varphi)$ for every transition $a \xrightarrow{k} b$ of the CFG and partial proof obligations $\varphi \in \mathsf{wp}(b)$ computed for node b;
– Finally, $\mathsf{wp}(a)$ collects both $\mathsf{wpGoals}(a)$ and $\mathsf{wpCfg}(a)$.

Inductively, one can then easily prove that, provided all proof obligations computed by $\mathsf{wp}(\mathsf{entry})$ are valid formulas, all execution paths in the CFG are actually *valid.*

Now to generate a WP smoke test for node a, it suffices to augment the definition of $\mathsf{wp}(a)$ with a new proof obligation for detecting that valid execution paths cannot reach node a. For this purpose, we simply generate a proof obligation $\varphi = \mathsf{false}$ for smoking nodes. A smoke test proof obligation at node a

is labelled with a special ACSL annotations denoted by Smoke_a, and we finally update the definition of proof obligations at node a as follows:

$$\mathsf{wp}(a) \stackrel{\triangle}{=} \quad \ldots \cup \{\ \mathsf{Smoke}_a : \mathsf{false} \mid \mathsf{smoking}(a)\ \}$$

Moreover, since we now have the $\mathsf{unreachable}(a)$ analysis at hand, we can further optimize the generation of proof obligations for explicit dead code. We omit here those details for lack of place, but this is actually implemented in WP.

The interpretation of $\mathsf{wp}(a)$ shall now be interpreted in a slightly different way: actually, each proof obligation $A : \varphi$ labelled with non-smoking ACSL annotation A still means that the annotation is valid on all execution paths. But proof obligation $\mathsf{Smoke}_a : \varphi$ labelled with a smoke test annotation at node a actually entails that node a is provably unreachable for valid execution paths. This is indeed equivalent to verifying the ACSL annotation `check \false` at node a. Notice that this would *not* be same to verify the ACSL annotation `assert \false`, since such an annotation would be also inserted as an assumption into the proof context of other proof obligations, which would be completely incorrect.

Hence, the global objective of WP for smoke tests has now changed: we still want to succeed in proving all proof obligations $A : \varphi$ labeled by non-smoking ACSL annotations A, but we want to *invalidate* the proof obligations $\mathsf{Smoke}_a : \varphi$ labeled by smoke tests. We debate further on how to fulfill those proof objectives when using SMT solvers in Sect. 3.4. We will then discuss how to provide users with informative feedback on smoke test results in Sect. 3.5.

3.4 Proof Method

In this section, we discuss the smoke test verification method itself. As explained previously, smoke tests are similar to standard proof obligations for a `check \false` annotation, although we expect this annotation to be *unprovable*.

Since we use SMT solvers to discharge proof obligations, let us reformulate smoke test verification in terms of satisfiability. A logical formula is submitted to an SMT solver, and it outputs one of the following possible answers:

- UNSAT: the formula is *not* satisfiable.
- SAT: the formula is satisfiable, and sometimes, the SMT solver can output a model validating the formula.
- UNKNOWN: the solver has failed to decide the satisfiability of the formula, or it has exhausted the allocated resources (time or memory).

The proof obligation $A : \phi$ shall be understood as universally quantified over its free variables. WP submits its negation, $\exists\neg\phi$, to SMT solvers. Hence, an UNSAT result means that $\forall\phi$ is true, which is actually the expected result. SAT results are currently not exploited by WP, although they can be used to generate counter-examples for debugging[13].

[13] Counter-examples have been integrated into WP in the latest release Frama-C 29 (Copper).

For a smoke test, WP submits the negated formula as usual, but the expected result is now SAT, in order to prove that the associated program point is *actually* reachable. If the SMT solver replies with UNSAT — which is not the expected result — we actually have detected some dead code or a *proven* inconsistency in the program specifications, and we can report the doomed program point to the user. However, when the SMT solver replies UNKNOWN, we have *no* guarantee that the formula is satisfiable or not.

Hence, smoke test verification is an incomplete process. This is the fundamental reason to refer to such a verification artifact as a *test* rather than a proof.

3.5 Providing Feedback to Users

The various results obtained from the generated proof obligations, especially smoke tests, deserve various kinds of feedback to users. Let us first consider the (set of possibly several, e.g. for loop invariants) proof obligations $A : \phi$ associated to some non-smoking ACSL annotation A. When *all* proof obligations labelled with A are discharged, we obviously report that the ACSL annotation A is valid[14].

For smoke test results, there are several situations to consider. Consider first a successful smoke test, that is, proof obligation $\mathsf{Smoke}_a : \phi$ is unproven. Indeed, no inconsistency at node a has been detected, and no further feedback is provided to the user: sanity checks are all right in this situation.

Consider now a failed smoke test, that is, proof obligation $\mathsf{Smoke}_a : \phi$ is proved. Here, the program point a is proved to be unreachable although *not* explicitly specified by the user to be intentional dead code. As illustrated by examples of Sect. 2, we report a failed smoke test. Further, in order to communicate this result to other plug-ins, WP also generates a new ACSL annotation `assert \false` at program point a, with a *valid* status, hence promoting the implicit `check \false` — actually proved by the smoke test — to a newly proven dead-code program point.

Moreover, all other ACSL annotations attached to the same program point become also *de facto* valid because of unreachability. More generally, all annotations that are associated to node a such that $\neg\mathsf{reachable}(a)$ are also valid because of unreachability. To inform the user, WP assigns a special validity status "Valid but Unreachable" to all those properties.

4 Industrial Experiments

The smoke test generation has been experimented on large industrial case studies, in particular, on real code bases under certification processes. Generating smoke tests during normal WP processing does not produce any visible overhead. However, as mentioned earlier, *verifying* smoke tests implies an additional

[14] More precisely, other proof obligations used as hypotheses for A are also taken into account by WP. We omit those details here for simplicity.

	dead-requires	dead-assumes	dead-call	dead-code	dead-loop	total
Total generated	363	150	732	1089	16	2350
Proved by CFG	0	0	0	7	0	7
Proved by Qed	0	0	0	24	0	24
Proved by SMT	0	0	0	6	0	6

Fig. 5. Generated and failing smoke tests per category and proving analyzer (with a default timeout of 2s per smoke test) for the JCVM project

cost, which can become significant for large projects, since we need to wait for SMT solvers to *fail* at proving satisfiability. Most of the time, on complex industrial studies, we must wait for external provers to timeout for each generated smoke test. This delay is alleviated by using prover caches, however, users generally turn smoke test verification off during specification and proof debugging, and only turn smoke tests on for nightly builds and final proof replay.

Despite its operational cost, smoke tests generation proves to be very useful for early detection of specification bugs, that can sometimes be difficult and time-prone to find. We also found internal soundness bugs of WP thanks to smoke tests, especially inside its standard logic library which contains many axioms. To avoid those kinds of inconsistencies, we have decided to turn most of our axioms into proper lemmas or Coq realizations.

Industrial Application on the JavaCard Virtual Machine (JCVM). WP has been applied on a large industrial code (a JavaCard Virtual Machine with over 7,000 lines of C code) to prove functional and global security properties [12] at Thales. This project was conducted according to a stringent common criteria certification process [19] and succeeded to reach the highest assurance level (EAL7, certificate available at [1]). Besides the effort to specify target properties with more than 30,000 lines of ACSL, the most important challenge was to ensure the full consistency of the formal specification.

Smoke tests have been systematically applied throughout the project to monitor the consistency of formal specification. They effectively guided the initial steps of verification especially when introducing hypotheses to define the perimeter of analyzed code. Given the already high cost of proof computation (over 4 hours on a PC with 3.10GHz 12-core CPU and 64 GB RAM for more than 80,000 verification conditions generated by WP), smoke tests provided a convenient way for sanity checks with a reasonable overhead. The generation of smoke tests is very efficient and fully automatic. The verification of smoke tests has a reasonable cost thanks to proof parallelization and prover cache. Overall, only a 20min (i.e., ≈8%) overhead is observed for the full proof time of the JCVM due to smoke tests.

Figure 5 shows the number of various kinds of generated smoke tests: precondition errors (dead-requires, dead-assumes), non-terminating calls (dead-call), dead-code and post-condition errors (dead-code), and inconsistent loop invariants (dead-loop). In total, 2,350 smoke tests were generated and 37 of them were proved to be failing. The last three lines of the table show the number of smoke tests proved as failing by different analyzers with the default timeout of 2s per smoke test. The lines for the three analyzers are shown in the order in which they are applied, so that an SMT solver is used only for goals not proved with CFG and Qed. In this project, we used only one SMT solver, Alt-Ergo [18]. All failing smoke tests belong to the dead-code category and are justified by the assumptions for the considered version. They are spurious alarms and may be avoided with explicit `assert \false` annotations at the corresponding code locations. Most smoke tests were simple enough to be either detected as unreachable by the CFG (control flow graph) analysis of Frama-C (7), or discharged by Qed (24). The default timeout of 2s per smoke test seems to be sufficient in practice to detect failing smoke tests for the JCVM. Proof sessions with a bigger timeout (3s, 5s and even 10s) did not detect more failing smoke tests. However, with a smaller timeout of 1s, 5 of them were not detected: the SMT solver identified only one failing smoke test instead of 6 (so for this timeout, in the last line of Fig. 5, 6 should be replaced by 1). As expected (for theoretical reasons of proof incompleteness), this illustrates that the absence of proof of the proof obligation for a smoke test *increases confidence* but *cannot guarantee* that the smoke test does not fail. Thus, a suitable timeout duration should be chosen. Our experiments and observations give confidence that the default timeout of 2s is suitable for the JCVM project. Indeed, rigorous code and specification review did not allow us to suspect other failing smoke tests in the project.

5 Related Work

Hoenicke et al. [15] define the notion of doomed statement we use in this work and relies on weakest pre-condition calculus with Boogie [4]. Bertoloni et al. [8] follow the same path. The main difference is that our work was originally meant to find inconsistencies in *specifications*, yet we have extended the analysis to error detection, in particular, the presence of dead code and, in a limited way, memory access errors. Other efforts, instead of deductive verification, use pattern-based and dataflow-based approaches [2] or constrain solving through Horn clauses [16].

The authors of [15,8,22,11] focus mainly on undefined behavior detection. For this, they make some assumptions about pointers that we do not currently make with WP. While useful to detect true errors, these assumptions might create false alarms (for example when a pointer is supposed not to be in the beginning of a memory block in the input of a function). We plan to extend WP with such kinds of assumptions but in a more controlled way, in order to optimize detection of runtime errors.

Most of these efforts were meant to tackle the problem of having too many false alarms, like [13], or for example WP without smoke tests, where we have proof failures on verification of absence of runtime errors but no way to know whether the alarm is a false alarm. Another approach for this is using model checking to find traces that bring to problems like in [10], or at least to find an explanation about the failure [21], or to produce a counter-example to the correct behavior through a specialized mechanism [20]. While we plan to add this last capability to WP, we prefer to rely on the ability of SMT solvers like CVC5 [3] or Z3 [9] to produce counter-examples.

6 Conclusion

This tool paper presented the design and implementation of smoke tests in Frama-C/WP. Introduced to address potential inconsistencies in specifications, they provide to verification engineers a practical and pragmatic solution for detecting inconsistencies and dead code, even if it is incomplete and cannot guarantee their absence. Along with the extensions we mentioned in Sect. 5, future work directions include a larger evaluation, as well as proof of correct generation of smoke tests, as part of a long-term effort on formalization of WP. Suggesting a suitable timeout for a smoke test in a given project (e.g. based on statistics or machine-learning) is another work perspective.

Acknowledgment. Part of this work was supported by ANR grants ANR-22-CE39-0014 and ANR-22-CE25-0018.

References

1. ANSSI: The EAL7 certificate ANSSI-CC-2023/45, https://cyber.gouv.fr/sites/default/files/document_type/Certificat-CC-2023_45fr_0.pdf
2. Ayewah, N., Pugh, W.W., Morgenthaler, J.D., Penix, J., Zhou, Y.: Evaluating static analysis defect warnings on production software. In: Das, M., Grossman, D. (eds.) Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'07, San Diego, California, USA, June 13-14, 2007. pp. 1–8. ACM (2007). https://doi.org/10.1145/1251535.1251536
3. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
4. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17
5. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. SIGSOFT Softw. Eng. Notes **31**(1), 82–87 (Sep 2005)

6. Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., Williams, N.: The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. Commun. ACM **64**(8), 56–68 (2021). https://doi.org/10.1145/3470569
7. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html
8. Bertolini, C., Schäf, M., Schweitzer, P.: Infeasible code detection. In: Joshi, R., Müller, P., Podelski, A. (eds.) Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7152, pp. 310–325. Springer (2012). https://doi.org/10.1007/978-3-642-27705-4_24
9. Bjørner, N.S., Eisenhofer, C., Kovács, L.: Satisfiability modulo custom theories in Z3. In: Dragoi, C., Emmi, M., Wang, J. (eds.) Verification, Model Checking, and Abstract Interpretation - 24th International Conference, VMCAI 2023, Boston, MA, USA, January 16-17, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13881, pp. 91–105. Springer (2023). https://doi.org/10.1007/978-3-031-24950-1_5
10. David, C., Kesseli, P., Kroening, D., Lewis, M.: Danger invariants. In: Fitzgerald, J.S., Heitmeyer, C.L., Gnesi, S., Philippou, A. (eds.) FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9995, pp. 182–198 (2016). https://doi.org/10.1007/978-3-319-48989-6_12
11. Dillig, I., Dillig, T., Aiken, A.: Static error detection using semantic inconsistency inference. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. pp. 435–445. ACM (2007). https://doi.org/10.1145/1250734.1250784
12. 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). https://doi.org/10.1007/978-3-030-90870-6_23
13. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: PLDI 2002: Extended static checking for java. ACM SIGPLAN Notices **48**(4S), 22–33 (2013). https://doi.org/10.1145/2502508.2502520
14. 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). https://doi.org/10.1007/978-3-319-91908-9_18
15. Hoenicke, J., Leino, K.R.M., Podelski, A., Schäf, M., Wies, T.: It's doomed; we can prove it. In: Cavalcanti, A., Dams, D. (eds.) FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5850, pp. 338–353. Springer (2009). https://doi.org/10.1007/978-3-642-05089-3_22
16. Kahsai, T., Navas, J.A., Jovanovic, D., Schäf, M.: Finding inconsistencies in programs with loops. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9450, pp. 499–514. Springer (2015). https://doi.org/10.1007/978-3-662-48899-7_35

17. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. **27**(3), 573–609 (2015). `https://doi.org/10.1007/s00165-014-0326-7`
18. OCamlPro: The Alt-Ergo solver, `https://alt-ergo.ocamlpro.com/`
19. Portal, T.C.C.: Common criteria for information technology security evaluation, `https://www.commoncriteriaportal.org/`
20. Rümmer, P., Shah, M.A.: Proving programs incorrect using a sequent calculus for java dynamic logic. In: Gurevich, Y., Meyer, B. (eds.) Tests and Proofs - 1st International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers. Lecture Notes in Computer Science, vol. 4454, pp. 41–60. Springer (2007). `https://doi.org/10.1007/978-3-540-73770-4_3`
21. Schäf, M., Schwartz-Narbonne, D., Wies, T.: Explaining inconsistent code. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013. pp. 521–531. ACM (2013). `https://doi.org/10.1145/2491411.2491448`
22. Wang, X., Zeldovich, N., Kaashoek, M.F., Solar-Lezama, A.: A differential approach to undefined behavior detection. Commun. ACM **59**(3), 99–106 (2016). `https://doi.org/10.1145/2885256`