ORSAY
N° d'ordre : 2119

UNIVERSITÉ DE PARIS-SUD 11

CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

L'HABILITATION À DIRIGER DES RECHERCHES
DE L'UNIVERSITÉ PARIS-SUD 11

PAR

Nikolai KOSMATOV

SUJET :

Combinations of Analysis Techniques for Sound and Efficient Software Verification

soutenue le 20 novembre 2018 devant la commission d'examen

Wolfgang	AHRENDT	Professeur	Chalmers Univ.	rapporteurs
Roland	GROZ	Professeur	Grenoble INP	
Marieke	HUISMAN	Professeur	Univ. of Twente	
Catherine	DUBOIS	Professeur	ENSIIE	examinateurs
Claude	MARCHÉ	Directeur de recherche	Inria	
Stephan	MERZ	Directeur de recherche	Inria	
Burkhart	WOLFF	Professeur	Univ. Paris-Sud	

ORSAY
N^o d'ordre : 2119

UNIVERSITÉ DE PARIS-SUD 11

CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

L'HABILITATION À DIRIGER DES RECHERCHES
DE L'UNIVERSITÉ PARIS-SUD 11

PAR

Nikolai Kosmatov

SUJET :

Combinations of Analysis Techniques for Sound and Efficient Software Verification

Acknowledgments

First of all, I would like to thank the Defense Committee members: Catherine, Marieke, Burkhart, Claude, Roland, Stephan and Wolfgang, for accepting to take part in the Committee and many interesting questions during the Defense. I know that coming to Paris at that very busy period was not easy for some of you, and I greatly appreciate your effort! Many thanks to Marieke, Roland and Wolfgang for accepting to review the thesis and their remarks on the document. I also thank Burkhart for chairing the Defense.

I am particularly grateful to Claude Marché who accepted to guide me through the whole habilitation process, helped me to organize the document, gave useful feedback and always found encouraging words when things were not moving forward fast enough. *Merci beaucoup, Claude !*

Also many thanks to Bernhard Aichernig, Marie-Laure Potet and Burkhart Wolff for accepting to act as preliminary reviewers on my research during the authorization step.

I am very much grateful to Catherine Dubois and Burkhart Wolff who, by their enthusiasm of researchers and their interest to combined analysis techniques, highly influenced my work in that direction. They also introduced me to the Test and Proof community and gave me the opportunity to organize TAP 2015, which was a great experience for me.

Many thanks to the PathCrawler team, Nicky, Bernard, Muriel and Patricia, for introducing me into the tool and letting me contribute to this project, without which many of my later projects would never become possible.

I am also very grateful to the (current and past) members of the Frama-C team, and in particular, Patrick, Virgile, Loïc, François, André, Pascal, Boris, Anne, Julien, Benjamin, Richard, David, Valentin, Matthieu, and all those I did not mention, without whom the whole story would not be possible. *Bravo à tous !*

I warmly thank all students and colleagues I collaborated (and continue to collaborate) with. I tried to carefully mention their names for each project described in the document. I am very grateful in particular to Jacques Julliand, Pascale Le Gall and Frédéric Loulergue with whom I co-supervised several students and who shared with me their experience of researchers and scientific advisors.

I would like to thank Fabrice Derepas and Benjamin Monate who previously headed our lab and gave me the opportunity to propose PhD projects on first analysis collaborations in Frama-C, which contributed to its current worldwide image of a rich collaborative platform for software verification. I thank Florent Kirchner, Head of Software Reliability Lab, and François Terrier, Head of our Department, who encouraged me to prepare this thesis. I warmly thank our secretary, Frédérique, for her daily support in everyday procedures and interactions with the administration of a very big and complex institution.

I cannot forget my PhD advisor, Alexander Generalov, for his excellent guidance and for having taught me how to become a true researcher. I would particularly like to thank Bruno Legeard who gave me the opportunity to follow the MS (DEA) in Computer Science and who accompanied my very first steps in this research field in 2002–2004 and taught me so much. I am also very grateful to the excellent faculty members at Saint-Petersburg State University and University of Besançon whose courses I followed.

Finally, I am extremely grateful to my wife Peggy and my children who saw me writing this thesis by sacrificing multiple weeks of vacation, as well as so many week-ends and nights spent in developing tools, writing papers, interacting with students, etc. This work would never have been finished without their support. And of course, I thank my parents and my sister for their constant encouragements over years.

Contents

List of Figures

Chapter 1

Introduction

This thesis presents selected research results of the author mainly related to source code based analysis of C programs and combinations of various static and dynamic analysis techniques.

1.1 Software Analysis: from Foundations to Combinations

Software quality, including in particular software reliability and security, has become one of the major concerns today. Modern systems and infrastructures in various areas of our life are increasingly dependent on software. Software components have become critical in various domains such as energy, avionics, rail, automobile, manufacturing, communication and medical devices. The complexity of software has increased very rapidly during the past decades, and is likely to increase even more in the near future.

Various Verification and Validation (V&V) techniques have been proposed to ensure software quality. They include dynamic and static analysis techniques. *Dynamic analysis* involves running the target program to evaluate its behavior, usually in order to compare it with the expected one. Examples of such techniques include software testing, runtime assertion checking and runtime verificaton. *Static analysis* techniques reason about the properties of the program without executing it and rely on some form of approximation of the set of possible states. Examples of static techniques[1] include abstract interpretation, deductive verification and model-checking.

Theoretical foundations for many of these techniques were established in the 20-th century. Floyd-Hoare logic for reasoning about program properties was proposed in the 1960's by Tony Hoare [266] and Robert W. Floyd [268], and extended by weakest precondition calculus — a basis for deductive verification — later in the 1970's by Edsger W. Dijkstra [265]. Abstract

[1] The term *static analysis,* however, is often used in a more restrictive sense.

interpretation was proposed by Patrick and Radhia Cousot in the 1970's [262]. Model-checking was introduced in the 1980's by E. Allen Emerson and Edmund M. Clarke [259] and Jean-Pierre Queille and Joseph Sifakis [255]. The theoretical result on undecidability of non-trivial semantic properties of programs was proven by Henry G. Rice [269] even earlier, in the 1950's. While the basic idea of testing is presumably as old as the first computer program, modern testing techniques also considerably evolved through the past decades. For instance, symbolic execution and its application for automatic test generation were proposed by James C. King [264] in the 1970's.

However, a successful practical application of these techniques was not achieved immediately. It often took several years or even decades to develop efficient and applicable tools, and to conduct convincing industrial case studies using them. Examples of first successful tools and case studies based on abstract interpretation in the late 20th – early 21st century include POLYSPACE [92, 218], born after a successful detection of errors by Alain Deutsch in Ariane 5 software after its crash in 1996, and ASTRÉE [202] used by Airbus for runtime error detection. Another tool, FLUCTUAT [170], developed in Software Reliability Lab of CEA List, was used by Airbus for numerical analysis of programs with floating-point numbers. An example of a successful industrial application of a deductive verification tool is the CAVEAT tool [227], also developed at CEA List and used by Airbus for certification of the aircraft A380. The development of the tool had started earlier, in the 1990's, and took several years. Probably the most spectacular example of application of automatic test generation was realized at Microsoft with SAGE [182] used in parallel on more than 100 servers to detect bugs in Windows. The success of deductive verification and test generation tools is partially due to the evolution of constraint solvers, having made a remarkable progress in the recent years.

While these recent achievements are certainly promising and encouraging, each of the techniques keeps its own limitations. Static analysis reasons over all program behaviors and usually relies on approximations. It is conservative and sound: the results may be imprecise, but guaranteed to generalize to all executions. On the other hand, dynamic analysis operates by executing a program and observing one or several executions. It is in general incomplete because of a big (or even infinite) number of possible executions, but efficient and precise because no approximation or abstraction are needed.

Combinations of static and dynamic analyses can be beneficial from the point of view of (the users of) both techniques. For example, a dynamic analysis tool looking for defects can take advantage of an additional static analysis step to ensure that some parts of the program are defect-free, and that potential defects can only be found in other parts, on which it should therefore focus. Conversely, a static analysis technique is only able to detect potential defects, some of which can be real and some others not. It can therefore take advantage of a dynamic analysis to confirm some of the potential defects as real ones, or rigorously examine other potential defects to get confidence that they are not real, or in some cases — when an exhaustive dynamic analysis is possible — even to ensure the absence of defects.

The exact notion of *defects* depends on the considered techniques and can apply to various situations. In the contributions described in this thesis, such situations include the combination of value analysis and structural path-oriented testing to detect runtime errors (in the SANTE method described in Section 2.3), or the combination of deductive verification and testing to debug proof failures (in the STADY method described in Section 3.2). Of course, these combinations are not magic and cannot make decidable an undecidable verification problem, but they turn out to have the potential of making the global analysis more efficient and to reduce the need for manual analysis of the *unknowns* by the verification engineer. Similarly, the detection of infeasible test objectives (realized in the LTEST toolset described in Chapter 4) before the test generation step makes test generation more efficient and reduces the need for manual analysis of uncovered test objectives.

In many cases, such combinations also bring other context-specific benefits. In the case of test generation, for example, the knowledge of infeasible test objectives allows for a more precise computation of coverage of a given test suite. In the case of proof debugging, a (likely) reason of the proof failure can be identified to provide the verification engineer with additional guidance.

Conceptually different are some other combinations where static analysis is used to modify the analyzed program itself, or more generally, to impact the problem to be considered by the following analysis. It is the case when program slicing is used to simplify the program itself before test generation (like in the STADY method described in Section 3.2). Another, even more complex combination was developed for runtime assertion checking (with E-ACSL2C described in Section 3.1), where a static dataflow analysis is used to optimize memory monitoring, by detecting irrelevant variables to be ignored during the monitoring.

Exploring combinations of different static and dynamic analyses in order to take benefit from their complementarity and limit their shortcomings has become one of my main research topics since 2008.

1.2 Challenges for Combinations of Analyses

Despite the variety of combinations of analyses presented in this document, several general challenges appeared to be important through various research projects. I state them below as four distinct concerns. Some of them are not fully independent from one another, and are not necessarily specific to combined analyses (e.g. see a detailed list of challenges for deductive verification [116]), but I try state them here in the most general terms, applicable to various techniques. These concerns guided to a great extent my research activities.

Efficiency. The first concern is to ensure that the considered combination of analyses brings the benefit of solving the target problem more efficiently than each of the combined analysis techniques. Criteria of evaluation for *efficiency*, used here in a very general sense, can include analysis time, number of detected defects and precision of the results.

Soundness. The second concern is *soundness* of the analysis combination. Indeed, when different techniques are combined, particular care should be taken to ensure that the assumptions and conclusions of the combined techniques are properly taken into account and interpreted. For example, some simplification or restrictive hypotheses made by one technique should not alter the soundness of the global results. Providing a careful semi-formal or formal justification of soundness of the whole combination, or ideally a machine-checked proof of it (e.g. in the COQ proof assistant [211]), is generally desirable. Such a formalization brings the benefit of explicit definitions of the language, assumptions and guarantees of each technique, and allows for formally establishing the soundness of the combined technique.

Specification Mechanisms. The third concern is to rely on sufficiently expressive, generic and well-defined *specification mechanisms*. Even when there already exists a well-defined specification language (like the ACSL specification language in FRAMA-C, presented in the next section), it can become necessary to deal with new kinds of properties and/or to handle them by different analysis techniques.

Practical Applicability. Finally, another key challenge is to ensure that the proposed analysis combinations are applied by practitioners and industrial users. This requires to carefully take into account the needs of the users, to communicate the obtained research results and to accompany the users in the evaluation and application of the developed tools.

1.3 Tool Context: the FRAMA-C Program Analysis Platform

The challenges stated in the previous section are very seriously taken into account in Software Reliability Lab of CEA List, where I work since 2006. CEA List is an applied research institute whose budget is only partially ensured by the French state, while the majority of fundings should come from industrial contracts, technology transfer and collaborative projects. This constrains the choice of topics investigated by CEA researchers, but provides an inexhaustible source of inspiration thanks to a strong and up-to-date connection with ongoing projects and industrial partners. This section presents the tool context of my research activities, the majority of which were related to FRAMA-C. Its presentation is partly based on our previous papers [70, 72, 11].

FRAMA-C [11] is a source code analysis platform developed at Software Reliability Lab of CEA List in collaboration with Inria. It is aimed at conducting analysis and verification of industrial-size programs written in ISO C 99 language. The platform is based on a common kernel, which hosts analyzers as collaborating plugins. If offers the ACSL formal specification language [109] as a *lingua franca*. FRAMA-C includes plugins based on abstract interpretation, deductive verification, dynamic symbolic execution, etc. In addition, the extensibility of the overall platform, and its open-source licensing, have fostered the development of an ecosystem of independent third-party plugins.

The FRAMA-C platform is mostly written in OCaml. Its development was initiated in 2004. The first public FRAMA-C release, called FRAMA-C Hydrogen (v.1), was published in 2008. At the time of writing, the most recent version is FRAMA-C Chlorine (v.17), released in May 2018. FRAMA-C is used worldwide both by researchers and practitioners, in particular, in such critical domains as avionics, energy, transport, defense. In 2012, a start-up, called TrustInSoft, was created to commercialize two products based on FRAMA-C, TIS-Analyzer and TIS-Interpreter.

FRAMA-C fully supports combinations of different analysis techniques. Collaborative verification across cooperating plugins is enabled by their integration on top of a shared kernel, and their compliance to a common specification language: ACSL [109]. Indeed, FRAMA-C plugins collaborate thanks to ACSL annotations whose status is maintained by the kernel and shared by different analyzers [149].

For example, some plugin A can prove and mark as “valid” an existing annotation a_1, then adds some other annotations a_2, a_3 to be proved and sets their status to “unknown”. Then some other plugin B can show that a_3 is valid whenever a_2 is valid, and sets the status of a_3 to “valid under condition a_2”. Now, if another plugin C manages to prove a_2, FRAMA-C will be able to deduce that a_3 is valid as well.

This section briefly presents a few FRAMA-C plugins intensively used in my work, and the underlying analysis techniques. A more complete description of the platform and its analyzers can be found on the FRAMA-C website [104] and in the foundational papers [11, 50]. A tentative detailed list of existing FRAMA-C plugins is given in the Habilitation Thesis of Julien Signoles [113].

1.3.1 Dynamic Symbolic Execution and Test Generation Tool PATHCRAWLER

Proposed in the 2000’s, Dynamic Symbolic Execution (DSE) [200, 207], is often seen today as one of the most advanced test generation techniques. If offers a structural path-oriented testing technique capable to produce test inputs for different program paths. It is sometimes also called *concolic* (*conc*rete + symb*olic*). It combines symbolic execution proposed in the 1970’s by King [264] with concrete execution.

For a given (execution) path π of a given program, the main idea of symbolic execution is to compute a *path predicate* ϕ_π, that is, a condition on program inputs that ensures that the program execution activates π. The path predicate ϕ_π can then be sent to a constraint solver or an SMT solver to produce a solution — a test input (or *test datum*) t — that will activate the path π. The path predicate ϕ_π is unsatisfiable if and only if the path π is infeasible. In this way, by exploring different paths of the program, an automatic tool can try to generate a test datum for each considered path. Concrete execution of the program on the produced input t can be used to check that the path activated by t is indeed π. It can also be used to accelerate test generation, or even find candidate solutions.

DSE has become very popular in recent years and has been implemented in several tools such as PATHCRAWLER [56, 207], DART/CUTE [200], KLEE [179], PEX [186], SAGE [182] (where it is combined with white-box fuzzing), etc. My work contributed to the PATHCRAWLER tool and essentially relied on it for combined analysis techniques.

The development of PATHCRAWLER was started in 2004–2006 as an independent tool by its main author Nicky Williams in collaboration with Patricia Mouy and Muriel Roger at Software Reliability Lab at CEA List. Later, I contributed to its development and its integration as a FRAMA-C plugin to facilite its combinations with other analyzers. PATHCRAWLER is based on a specific constraint solver, called COLIBRI, developed at CEA List mainly by Bruno Marre, and shared with other testing tools such as GATeL [225]. COLIBRI implements advanced features such as floating-point and modular integer arithmetic.

Given a C program p under test, a precondition restricting its inputs and an optional oracle, PATHCRAWLER generates test cases respecting a given test coverage criterion. PATHCRAWLER provides coverage strategies like *all-path* (all feasible paths) and *k-path* (feasible paths with at most k consecutive loop iterations with $k \geqslant 0$). It is *sound*, meaning that each test case activates the test objective for which it was generated. This is verified by concrete execution. On the class of programs it supports, PATHCRAWLER is also *relatively complete* in the following sense: given a program with a finite number of paths and a sufficient time, the tool will exhaustively explore all feasible paths of the program. In this case, the absence of a test for some test objective means that the test objective is infeasible (i.e. impossible to activate). This is due to the fact that the tool does not approximate path constraints [56, Sec. 3.1]. Of course, given only a (bounded) finite time, the tool can time out without generating a test for a given test objective.

Example. Let us illustrate how PATHCRAWLER works on the toy example shown in Figure 1.1. Assume this function has a precondition $x \leqslant 40$ and test generation is run for f with the *all-path* criterion. The tool identifies program inputs (here, parameter x) and takes into account the precondition, so that all generated test cases will respect it. Then the tool explores (partial) program paths, also called *path prefixes*, in a depth-first search, constructs a path predicate for each one and tries to solve it, before moving to the next path. If a solution for the path predicate

```
1 int f(int x){
2   if(x < 0)
3     x = x + 1;
4   if(x ≠ 1)
5     x = 2*x;
6   return x;
7 }
```

Test case N^0	Inputs	Path (prefix)
TC 1	$x = -5$	$1, +2, 3, +4, 5, 6$
—	(infeasible)	$1, +2, 3, -4$
TC 2	$x = 25$	$1, -2, \quad +4, 5, 6$
TC 3	$x = 1$	$1, -2, \quad -4, \quad 6$

Figure 1.1 – A C function f with three feasible execution paths, and the generated test cases

is found, a new test case with the generated inputs and the complete path is recorded. Otherwise the tool simply moves to the next path.

An example of a test generation session is shown on the right in Figure 1.1, where $+n$ (resp., $-n$) indicate that the condition on line n is true (resp., false) in this path. Notice that the second line corresponds to an infeasible partial path $1, +2, 3, -4$ since we cannot have $x = 1$ at line 4 after the assignment at line 3 if we had $x < 0$ at line 2. The tool deduces that fact since the corresponding path predicate

$$x_0 \leqslant 40 \wedge x_0 < 0 \wedge x_0 + 1 = 1$$

expressed in terms of the input value (denoted by x_0) is shown to be unsatisfiable by the solver. Recall that the first part of the predicate comes from the precondition. For the three other paths, the solver manages to solve the path predicate, and produces a solution — a test input. A more detailed presentation of PATHCRAWLER can be found in [56].

1.3.2 Abstract Interpretation and Value Analysis Plugin EVA

Value analysis is a program analysis technique that computes a set of possible values for every program variable at each program point. It is based on *abstract interpretation* proposed by Cousot and Cousot in the 1970's [262]. Its main idea is to compute an abstract view of values of variables in the form of *abstract domains*. For example, a possible abstract view for a numeric value is an interval.

Value analysis can be very useful to detect potential runtime errors or prove their absence. Typical examples include invalid pointers, invalid array indices, arithmetic overflows or division by zero. It can also help to prove other properties for which domain-based reasoning can be efficient.

The original value analysis plugin (called VALUE) [11] for FRAMA-C was mainly developed and maintained by Pascal Cuoq and Boris Yakobowski. Since the FRAMA-C Aluminium (v.13) release in May 2016, FRAMA-C offers a new value analysis plugin EVA (Evolved Value Analysis) [114], initiated by David Bühler. Today, the EVA team also includes André Maroneze and

```
1  int f ( int a ) {
2    int x, y;
3    int sum, result;
4    if(a == 0){
5      x = 0; y = 0;
6    }else{
7      x = 5; y = 5;
8    }
9    sum = x + y;      // sum can be 0
10   result = 10/sum; // risk of division by 0
11   return result;
12 }
```

Figure 1.2 – A C program with a possible division by 0 at line 10

Valentin Perrelle. EVA implements value analysis as a generic extendable analysis parameterized by cooperating abstract domains. Different, highly optimized domains are used to represent integers, floating-point numbers and pointers.

Example. Figure 1.2 shows an example of a program where EVA detects an alarm of division by zero. Indeed, after the then branch, both x and y are zero, so their sum is zero as well. To run FRAMA-C/EVA on this program (supposed to be contained in file div.c), the user can type the following command

```
frama-c-gui -val div.c -main f
```

in order to explore the results in the Graphic User Interface (GUI) of FRAMA-C. The option -val activates EVA, while the option -main f indicates that the entry point is function f (rather than the usual function main, not given in this toy example).

EVA computes the sets of possible values of variables at each program point. After the conditional statement, the domains of x (and similarly for those of y) computed inside the then and else branches are joined into a more general domain containing both cases. That is why the domain of x and that of y at line 9 are $\{0, 5\}$. Thus, after the assignment on line 9, the domain of sum is computed as $\{0, 5, 10\}$. Since 0 is identified as a possible value of sum before the division on line 10, EVA detects a potential risk of division by 0 and reports an *alarm*. It is reported in the FRAMA-C GUI by adding an ACSL assertion

```
assert sum ≠0;
```

before line 10 in the GUI. (For convenience of the reader, in the examples in this document, some ACSL symbols (like \forall, \exists, integer, <=, !=, &&, etc.) are pretty-printed in the corresponding mathematical notation (resp., as $\forall$, $\exists$, $\mathbb{Z}$, $\leqslant$, $\neq$, $\wedge$, etc.).)

This assertion indicates that the risk of division by zero at line 10 cannot be excluded by EVA, and the given property should be verified (manually or by other means) to exclude this risk. In this case, this property cannot be proved, and the risk is actually real since a division by zero occurs if the input value `a` is zero and the then branch is taken. Notice that the computed domains are over-approximated: in fact, the exact domain of `sum` after line 9 is $\{0, 10\}$, but it is computed as $\{0, 5, 10\}$ even if value 5 cannot be taken.

Over-approximation can lead to detecting *false alarms*, that is, situations where a potential error is reported while the error can never occur in practice. Several options are available in EVA to control the precision of the analysis and reduce the number of false alarms. An increase in precision comes at the cost of making the analysis potentially slower. Moreover, as it follows from the theorem proved by Rice [269], static analysis cannot precisely determine all errors, and false alarms remain a major issue of its application in practice.

EVA is strongly integrated into the FRAMA-C ecosystem and offers a basis for many other derived plugins that reuse its results (see [11]). Among them, the SLICING plugin for program slicing. Program slicing reduces a given program and produces a smaller program (a slice) that is equivalent to the original one with respect to a given point of interest.

1.3.3 Function Contracts and ACSL Specification Language

ACSL (ANSI/ISO C Specification Language) [109] is a formal behavioral specification language offered by FRAMA-C and shared by different FRAMA-C analyzers. It allows users to specify functional properties of C programs similarly to EIFFEL [232] and JML [124, 224]. It is based on the notion of function contract.

The *(function) contract* of a function f specifies the *preconditions* that are supposed to be true before a call of f (i.e. ensured by the caller), and the *postconditions* that should be satisfied after the call of f (and should be thus established during the verification of f). In ACSL, the preconditions are specified in **`requires`** clauses, while the postconditions are stated in **`ensures`** clauses. An additional type of postconditions, specified in an **`assigns`** clause, states a list of global variables (or, more precisely, locations of the global memory state) that may have a different value before and after the call. When the contract of f contains such a clause, all locations that are not mentioned in it must have the same value before and after the call of f. Function contracts can be also represented in the form of different cases (behaviors).

ACSL annotations are written directly in the C source file in special comments of the form `//@...` or `/*@ ... */`, the latter being, as usual in C, possibly multi-line. Thus, annotations are ignored by the compilation and execution steps. Predicates used in ACSL annotations are written in typed first-order logic. Variables have either a C type or a logical type (e.g. $\mathbb{Z}$ or

```
1 /*@ requires n ⩾ 0 ∧ \valid(t+(0..n-1));
2   assigns \nothing;
3   ensures \result ≠ 0 ⇔ (∀ ℤ j; 0 ⩽ j < n ⇒ t[j] == 0);
4 */
5 int all_zeros(int *t, int n) {
6   int k=0;
7   /*@ loop invariant 0 ⩽ k ⩽ n;
8     loop invariant ∀ ℤ j; 0 ⩽ j < k ⇒ t[j] == 0;
9     loop assigns k;
10    loop variant n-k;
11  */
12  while(k < n){
13    if (t[k] ≠ 0)
14      return 0;
15    k++;
16  }
17  return 1;
18 }
```

Figure 1.3 – Function `all_zeros` specified in ACSL

ℝ for mathematical integer or real numbers). The user can define custom logical functions and predicates and use them in annotations together with ACSL built-ins. Indeed, ACSL features its own functions and predicates. In particular, regarding memory-related properties, **`\valid`**`(p)` expresses validity of a pointer `p` (i.e. being a non-null pointer which can be safely read or written by the program); **`\base_addr`**`(p)`, **`\block_length`**`(p)`, and **`\offset`**`(p)` express respectively the base address, the size of the memory block containing `p` and the offset of `p` inside it (in bytes), while **`\initialized`**`(p)` is true whenever the pointed location `*p` has been initialized. The simplest form of an ACSL annotation is an assertion of the form

`assert` `<condition>;`

It requires that the given condition be always satisfied at the given program point. We refer the reader to [109] for detailed documentation of all ACSL features.

Example. Figure 1.3 illustrates a C function `all_zeros` specified in ACSL. This function receives as arguments an array `t` and its size `n`, and checks whether all elements of the array are zeros. If yes, it returns a nonzero value, and 0 otherwise. The function contract contains a precondition (line 1) and postconditions (lines 2–3). The precondition states that the array size `n` is non negative and that the input array contains `n` valid memory locations at indices `0..(n-1)` that can be safely read or written. If this is not true, the function can have undefined behavior (and provoke runtime errors) when accessing the array contents on line 13. Thus, this property must be ensured by the caller and should be specified in the precondition.

The **assigns** clause at line 2 states that the function is not allowed to modify any non-local variable. Without this clause, an erroneous implementation writing zeros in all elements of the array and returning 1 would be considered correct with respect to the contract. Finally, the clause on line 3 states that the result is nonzero if and only if all elements of the array are equal to zero. The loop contract at lines 7–11 will be discussed in the next section.

1.3.4 Weakest Precondition Calculus and Deductive Verification Plugin WP

Deductive program verification consists in establishing a rigorous mathematical proof that a given program respects its specification. When no confusion is possible, one also says for short that deductive verification consists in “proving a program”. It is usually based on weakest precondition calculus.

Weakest precondition calculus was proposed by Dijkstra [267] in the 1960’s. It automates reasoning using Floyd-Hoare logic [266, 268]. It computes, for a given program S and a given predicate R, the weakest (that is, the most general) condition $P_{\text{w}} = \text{WP}(S, R)$ which, being assumed as a precondition before the execution of S, will ensure the validity of R at the end of S. Thanks to this computation, it can be easily checked if a given precondition P ensures the validity of R at the end of S: it is sufficient to verify whether P is stronger than P_w, that is, to prove the condition $P \Rightarrow P_w$. If it is the case, one says that $\{P\}S\{R\}$ is a *valid Hoare triple*. Thus, weakest precondition reduces any deductive verification problem to establishing the validity of first-order formulas called *verification conditions*. However, in the presence of loops, this calculus is not able to reason automatically and requires loop contracts illustrated in the example below.

The WP plugin [100, 11] of FRAMA-C, mainly developed by Loïc Correnson with Patrick Baudin, François Bobot and Zaynah Dargaye, performs weakest precondition calculus for deductive verification of C programs. Various automatic SMT solvers, such as Alt-Ergo, CVC4 and Z3, can be used to prove the verification conditions generated by WP.

Example. Let us illustrate deductive verification with WP on the example of Figure 1.3 (supposed to be contained in file `all_zeros.c`). The command

```
frama-c-gui -wp all_zeros.c
```

runs the proof with WP on this example and shows the results in the FRAMA-C GUI.

Suppose first that the user has specified the contract at lines 1–3 without writing the loop contract at lines 7–11. In this case, the proof of the postcondition will not be successful. Since the number of loop iterations can be arbitrary, loops cannot be automatically traversed by the weakest precondition calculus.

In the presence of loops, the deductive verification tool requires a *loop invariant*, i.e. an additional property on the program state that is true before the loop and after each complete loop iteration. It can be specified in a loop contract using **`loop invariant`** and **`loop assigns`** clauses. The loop invariant at line 7 specifies the interval of values of the loop variable `k`. The clause at line 8 specifies that all elements at indices `0..(k-1)` are equal to 0 (that is indeed true after any complete loop iteration that was not interrupted by the return statement at line 14). Similarly to **`assigns`**, the **`loop assigns`** clause specifies the variables (both global and local in this case) that may change their value during the loop. The loop contract can also indicate a **`loop variant`**, which defines a decreasing natural expression that is positive whenever a new iteration starts. A loop variant can be seen as an upper bound of the number of remaining loop iterations and is used to prove that the loop terminates. In this example, `n-k` provides such a bound (cf. line 10).

On the complete program of Figure 1.3 with the loop contract, WP successfully proves that this function respects its specification, and that the loop terminates. In addition, it is possible to make WP check the absence of runtime errors using the option `-wp-rte`. For this program, thanks to the array validity assumed at line 1 and the interval of values specified at line 7, WP successfully proves that runtime errors cannot occur: the array access at line 13 is valid and the arithmetic operation at line 15 does not overflow.

For more detail on deductive verification with WP, the reader may refer to articles [11, 136], dedicated tutorials [102, 74] and the WP manual [100].

1.4 Structure of this Document

The document is organized in five chapters. Chapter 1 contains an introduction.

The next three chapters offer the reader three journeys through various scientific contributions that I realized in collaboration with several colleagues and students. Each chapter presents the collaborations and students involved, motivations, main achievements, and the resulting publications and patents. The degree of detail is not constant: some topics are presented in more detail, while others are described more briefly. It was done on purpose and motivated by space limitations, a large variety of different topics and the intention to make the document easy to follow for the reader.

Chapter 2 presents the evolution of my early research interests from model-based testing and constraint solving to structural testing, and then to a combination of value analysis with testing and program slicing. It also presents the usages of this combination in collaborative projects, and finally explores the foundations of program slicing that were necessary to ensure the soundness of this combination. This chapter demonstrates three major concerns of my research activities: combining various analysis techniques to get better results, doing it in a sound way, and applying the results in practice.

Chapter 3 proposes another journey starting by our work on executable specifications and their efficient tool support for runtime assertion checking in FRAMA-C. This research topic — that was very active, exciting and fruitful in itself — also provided the necessary basis for a combined verification approach for analysis of proof failures using dynamic analysis. This chapter illustrates my concern to design sound and efficient combined verification tools, where dynamic analysis is capable to assist the verification engineer in the difficult task of program proving, and to provide suitable specification mechanisms for such combinations. It finishes by a recent extension to support relational properties that was motivated by several industrial collaborations and is already used in some collaborative projects. It shows the importance of practical applications in the research of our lab and in my own work.

Chapter 4 — as a circular ending would require — brings the reader back to the beginning and focuses on testing. More precisely, it presents our work on specification and tool support for advanced test coverage criteria. It emphasizes the concern to deal with rich, well-defined, amenable to efficient support specification mechanisms — here for test objectives — and presents their analysis using structural testing and combined analysis techniques. An ongoing adoption of this technology by a major industrial partner, Mitsubishi Electric, illustrates the interest of this work in practice.

Chapter 5 finishes the thesis. It first summarizes some other projects and results that were not presented in this thesis. Then it provides some concluding remarks and future work directions.

Chapter 2

From Testing to Static Analysis

This chapter first briefly presents my initial research activities in Computer Science on model-based testing and constraint solving conducted during postdocs and temporal assistant positions in 2003-2006. Next, it describes my contributions to the PATHCRAWLER test generation tool and its online version after joining CEA List in December 2006. Then a few later projects are presented in more detail.

A combination of value analysis, slicing and test generation within FRAMA-C (in the SANTE method and tool) was proposed during the PhD work of Omar Chebaro[1], performed from November 2008 to December 2011 and co-supervised in collaboration with Alain Giorgetti and Jacques Julliand (University of Franche-Comté, Besançon). Omar Chebaro defended his thesis [156] on December 13, 2011. We presented these results in several publications. The original idea to combine value analysis with testing appeared in TAP'10 [55], and was enriched by a combination with slicing in our TAP'11 paper [53]. The most complete description of the method with various slicing options was given in our SAC'12 paper [49]. The design of the combined tool SANTE in FRAMA-C and its underlying analyzers were detailed in a journal paper in *Autom. Softw. Eng.* [12]. Support for input pointers in SANTE was presented in ICSSEA'12 [48]. Later, the SANTE method was adapted for detection of security vulnerabilities in the European FP7 project STANCE as described in our HVC'15 paper [37].

The ambition to formally establish the soundness of the SANTE method gave rise to a research project on formalization of program slicing, conducted during the PhD work of Jean-Christophe Léchenet[2], performed from January 2015 to July 2018 and co-supervised in collaboration with Pascale Le Gall (CentraleSupélec). Jean-Christophe Léchenet defended his thesis [112] on July 19, 2018. The main results of this work were published in FASE'16 [31], its extended journal version in *Formal Asp. Comput.* [8], and FASE'18 [17]. They were also presented in French conferences JFLA'16 [65] and JFLA'18 [64].

[1] currently, teaching position at the University of Beirut, Lebanon

[2] currently, postdoc at Inria, Rennes

2.1 First Activities: Model-based Test Generation and Constraint Solving

My first research projects in Computer Science were conducted during my MS thesis, postdocs and teaching assistant positions at the University of Franche-Comté (Besançon), RWTH-Aachen and Inria (Nancy) in 2003–2006. They were related to model-based test generation and constraint solving.

In collaboration with Bruno Legeard, Fabien Peureux (University of Franche-Comté, Besançon) and Mark Utting (University of Waikato, New Zeland [3]) we have proposed a new family of test coverage criteria formalizing boundary-value testing heuristics. It was published in our ISSRE'04 paper [63] based on my MS thesis. The new criteria form a hierarchy of data-oriented coverage criteria, and can be used either to measure the coverage of an existing test set, or to generate tests from a formal model. We define techniques to generate tests that satisfy these criteria. These criteria and techniques have been incorporated into the BZ-TT toolset (technology later transferred to Smartesting start-up) for automated test generation from B, Z and UML/OCL specifications.

In another research project, I proposed an original constraint solver for sequences [80]. It was presented in the BeyondFD'05, SAC'06 and INAP'05 papers [59, 60, 62] and implemented in Constraint Handling Rules (CHR) in SICStus Prolog. It is based on Thom Frühwirth's solver for lists. Sequences are one of the data types used in notations such as B or Z and representing finite lists of elements such as stacks, queues, communication channels, sequences of transitions or any other data with consecutive access to elements. The problem of constraint solving for sequences is very close to that of words or lists. While satisfiability of word equations (where concatenation is the unique operation) was proven decidable by Makanin in 1977, more general theories of words can become undecidable. The general constraint solving problem for sequences is even more complex than that for words, because sequences generalize words and are usually considered with more operations. I addressed the problem from the practical point of view and proposed a constraint solving technique that supports 11 basic operations on sequences.

I also contributed to two other research projects. One of them was dedicated to verification of parameterized protocols, in collaboration with Alain Giorgetti et Jean-François Couchot (University of Franche-Comté, Besnçon). It was presented in our ASE'05 paper [61] and mainly based on the PhD work of Jean-François Couchot, where I contributed to the proof of decidability results. The second one, on verification of cryptographical protocols in collaboration with Ralf Treinen (ENS de Cachan), Yohan Boichut (University of Franche-Comté, Besançon) and Laurent Vigneron (LORIA, Nancy), was described in our TFIT'06 paper [68] and was based on my postdoc work at Inria. Its main purpose was protocol translation into the intermediate

[3] Here and below, the affiliation is given at the time of the corresponding project.

language of the AVISPA protocol verification tool [201].

2.2 Structural Unit Testing: PATHCRAWLER and PATHCRAWLER-online

2.2.1 PATHCRAWLER Test Generation Tool

After joining Software Reliability Lab of CEA List in 2006, my research topics extended to white-box test generation (in particular, for the *all-path* criterion requiring to cover all feasible program paths), based on *concolic* testing or *dynamic symbolic execution* (cf. Section 1.3.1).

I contributed to the development of the PATHCRAWLER testing tool [207], that had been started in 2004 by Nicky Williams and other colleagues at CEA List and was not a FRAMA-C plugin at that time. In particular, after joining the PATHCRAWLER team, I worked on test generation strategies, interprocess communication, code instrumentation and output features. I also contributed to its integration as a FRAMA-C plugin, that was a decisive step having allowed its collaborations with other FRAMA-C analyzers. The design of PATHCRAWLER was detailed in our AST'09 paper [56]. An overview of constraint-based test generation approaches and related challenges was given in a book chapter [14].

One of my specific contributions regarding test generation strategies was related to the treatment of aliases. The presence of aliases may force a test generator to enumerate all possible inputs, generate several test cases for the same path and/or fail to generate a test case for some feasible path. In the ISSRE'08 paper [58], I classify aliases into two groups: external aliases, existing already at the entry point of the function under test (e.g. due to pointer inputs), and internal ones, created during its symbolic execution. I propose an original extension of the classical depth-first test generation method for C programs with internal aliases where the application of alias relations is delayed until the end of the path evaluation. It limits the enumeration of inputs and the generation of superfluous test cases.

One theoretical issue of *all-path* test generation concerns its complexity. To better identify practical and theoretical limits of this technique, I studied in the TAIC-PART'09 and JFPC'10 papers [67, 57] two particular classes of programs. For the first class containing the simplest programs with strong restrictions, *all-path* test generation is possible in polynomial time. For a wider class of programs in which inputs may be used as array indices (or pointer offsets), *all-path* test generation is shown to be NP-hard.

Another contribution was motivated by the PATHCRAWLER users' requests to express the precondition of the function under test in a separate C function rather than in an internal (less user-friendly) format of PATHCRAWLER. We proposed an original technique to treat the precondition expressed in a separate C function. This technique delays the resolution of the precondition constraints until the end of evaluation of the constraints of the function under test. This technique makes the expression of the precondition for test generation not only more convenient, but also efficient. This effort has been presented in our CSTVA'13 [43] and RV'13 [44] papers. This feature also appeared to be helpful for the users of the PATHCRAWLER-online testing service, presented in the next section. It also became a key feature of PATHCRAWLER that made it possible to generate a precondition for testing from an executable formal specification expressed in E-ACSL and allowed some combinations of static and dynamic analysis techniques (this work will be described below in Chapter 3).

2.2.2 PATHCRAWLER-online Testing Service

In 2009, CEA List decided to create an online testing service based on PATHCRAWLER. I was in charge of this project and became the main author of the PATHCRAWLER-online test generation service [79], whose first version was released online in 2010. Two interns, Amine Kouider in 2009 and Nicolas Dugué in 2011, also contributed to this project under my supervision. Later, in 2011–2012, Richard Bonichon (CEA List) also greatly contributed to the development of its new interface.

PATHCRAWLER-online allows the user to generate test cases for one of the predefined examples, as well as for a C code submitted by the user. The last feature is particularly challenging: since PATHCRAWLER compiles and executes the code, the online testing service basically allows the user to submit and execute any code on the server, which makes it particularly vulnerable. The architecture of the service relies on a carefully designed virtualization solution that isolates the test session of each user from all other users and system processes. Following this project, the design, experience and challenges of online test generation have been reported in our SOSE'13 paper [47] and a book chapter [13]. In 2011, the demo of PATHCRAWLER-online received a best tool demo award at CSTVA'11 [54].

Today, PATHCRAWLER-online is widely used for research, teaching and evaluation of the underlying test generation technique. Compared to a similar service, Microsoft's Pex4Fun [107], it offers richer outputs helping the user to understand the test generation process in detail, and appears to be particularly suitable for teaching. The users of PATHCRAWLER-online can upload a C code, generate a test suite, evaluate its coverage, provide an oracle, check for errors, explore symbolic and concrete outputs, path predicates, etc. It is used by teachers and students in many French universities, for example, in Paris, Orléans, Orsay, Evry, Strasbourg, Bourges, Toulouse..., but also in China, Germany, USA, India, Iran, Austria, Canada, etc. Over 1000 test sessions are run on PATHCRAWLER-online every month, and the server's current capacity to support up to 20–30 users simultaneously has become a serious limitation in front of its growing success.

2.3 Value Analysis, Slicing and Testing: the SANTE Method

2.3.1 Motivation

Value analysis is one of the existing techniques to detect (potential) runtime errors (cf. Section 1.3.2). Such errors include, for instance, division by 0, out-of-bound array accesses or invalid pointers. Value analysis is based on *abstract interpretation* [262] that offers a sound approximation of the behavior of a program. It computes an (over-approximated) set of possible values of each variable at each point of the program. In particular, it makes it possible to check whether an operation that can lead to an error at runtime is safe, by verifying the range of the involved expression at the relevant program point. If the abstraction guarantees that the undesirable values cannot occur, we have statically proved that the execution of the operation will always succeed at runtime. If the undesirable values cannot be excluded, value analysis reports a possible error by an alarm. Essentially, an alarm can be seen as a pair containing the threatening statement and the potential error condition. In FRAMA-C, the value analysis plugin VALUE, or its more recent version EVA, report an alarm by generating an ACSL assertion that should be proved to avoid the potential error.

Some of the detected potential errors may actually never appear at runtime; they are reported because of the overapproximation used by the analysis. An alarm that cannot occur at runtime is called a *false alarm*. An alarm in a program p is an *error*, or a *bug*, if there exist some inputs for p that activate the corresponding threatening statement with the undesirable behavior, and thus confirm the error. Notice that a runtime error does not necessarily result in a program crash, when for instance an out-of-bound array access leads by chance to another accessible user memory location, but we still consider such cases as bugs since they provoke an undefined program behavior.

Detection of runtime errors using value analysis is complete but imprecise. In practice, false alarms represent an important drawback of value analysis. On the other hand, testing operates by executing a program and observing one or several executions. It is in general incomplete because of a big (or even infinite) number of possible executions, but efficient and precise because no approximation or abstraction are needed: it can examine the actual, exact runtime behavior of the program for the corresponding test input.

The motivation of the PhD work of Omar Chebaro, supervised in collaboration with Alain Giorgetti and Jacques Julliand (University of Franche-Comté, Besançon) was to take benefit of the complementarity of value analysis and testing for detection of runtime errors. Another important motivation of this work was to automatically provide the validation engineer with as precise information as possible on each detected error. For example, while the original program can be rather big, an error can often be illustrated on a simpler program, with a shorter program path, a smaller constraint set at the erroneous statement, giving values for relevant variables only,

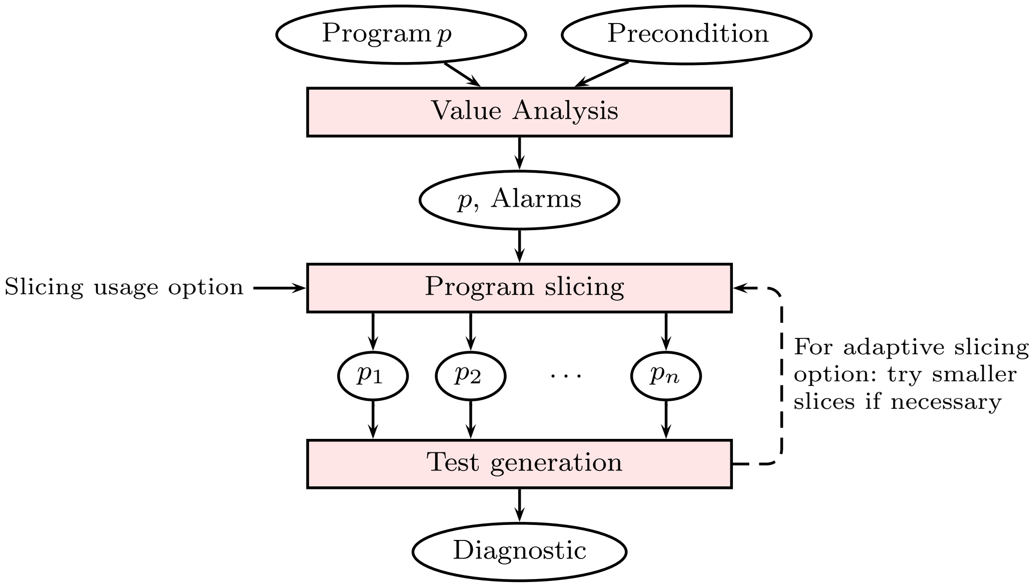


Figure 2.1 – Overview of the SANTE method

etc. Such information can considerably reduce the time of manual analysis and correction of the error by a software developer.

2.3.2 Approach and Main Results

We proposed a method called SANTE (Static ANalysis and TEsting) combining abstract interpretation based value analysis, program slicing and structural testing [48, 49, 53, 55]. The method uses value analysis to report alarms for possible runtime errors (some of which may be false alarms), and test generation to confirm or to reject them.

However, test generation may time out on real-sized programs before confirming some alarms as real bugs or rejecting some others as unreachable. To tackle this problem, we propose to reduce the source code by program slicing before test generation.

The complete method is illustrated by Figure 2.1. Its inputs are a C program p and its precondition. At the first step, value analysis produces a set of *alarms* A reporting threatening statements in p for which it detects a risk of runtime error.

The objective of the second step, based on program slicing, is to simplify the initial program before the last step. According to the user-defined slicing option and the structure of dependencies in A, this step determines which and how many simplified programs (*slices*) should be generated and sent to dynamic analysis. Each simplified program p_i contains a subset of alarms that can be triggered.

Finally, for each simplified program p_i, the dynamic analysis step tries to trigger the potential threat (i.e. provoke a runtime error) for each alarm present in p_i. It can generate a *counterexample* for the assertion expressing an alarm, i.e. a test datum showing that the instruction reported by the alarm is not safe and provokes a runtime error. Test generation allows SANTE to produce for each alarm a *diagnostic* that can be *safe* for a false alarm, *bug* for an effective bug confirmed by some input state, or *unknown* if it does not know whether this alarm is an effective error or not. We say that an alarm is *classified* if its diagnostic is bug or safe.

Several slicing options are available in SANTE, each of them proposing a different usage of program slicing.

— A trivial option *none* skips program slicing, and sends the initial program $p_1 = p$ to the dynamic analysis step. In this case, test generation is run on the complete program p and does not benefit of any reduction by slicing.

— Option *all* applies program slicing only once to generate a unique simplified program p_1 containing all alarms of A. It removes some irrelevant parts of the initial program. The disadvantage of this usage is that test generation may time out. Thus, some alarms that can be easier to classify in a smaller slice are penalized by the analysis of a bigger slice containing other alarms.

— Option *each* performs program slicing with respect to each alarm separately. It produces n slices $p_1, \dots, p_n$, where $n = \text{card}(A)$. Since test generation is executed separately for each slice, there is a risk of redundancy and waste of time with this option if some alarms are included in several slices. Indeed, if the threatening statement of alarm a is dependent on that of alarm a', any slice containing a will also contain a'.

— Options *min* and *smart*. To obtain a trade-off between a better precision (with a bigger number of slices leading to a bigger number of costly dynamic analysis steps) and better performances (with a smaller number of slices, where more alarms can remain unclassified), the SANTE method also proposes to exploit the dependencies between the alarms in advanced usages of program slicing called *min* and *smart*. The *min* usage produces a slice with respect to every alarm that is not a dependency of any other alarm[4], thus avoiding any redundancy. But here again, some alarms can be penalized by the analysis of a bigger slice and remain unclassified when test generation times out. To tackle this drawback, option *smart* first considers the same slices as *min* and then proposes an iterative verification process, where smaller slices are generated and sent to dynamic analysis as long as there is still a chance to classify more alarms on these smaller slices.

The most complete presentation of the SANTE method and a formalization of alarm dependencies are given in our SAC'12 paper [49].

[4] In the presence of mutually dependent alarms, it is a bit more technical: we consider in fact equivalence classes of mutually dependent alarms.

Summary of Main Results. The main results of our work on this topic include:

— a combined method for detection of runtime errors using value analysis, slicing and test generation,

— several slicing options, including optimized and adaptive usages of program slicing,

— definition of the underlying notions of dependencies between alarms and proof of their properties,

— an implementation of the proposed solution in a tool called SANTE and its experimental evaluation,

— a later extension of the SANTE method to security vulnerabilities.

2.3.3 Implementation and Evaluation

The proposed method was implemented in the SANTE prototype plugin in FRAMA-C. It treats the risks of division by zero, out-of-bounds array access and some cases of invalid pointers. It relies on tools developed at CEA List (described in Section 1.3). The first step uses the value analysis plugin of FRAMA-C. The second step uses the SLICING plugin. The last step uses the PATHCRAWLER test generator relying on the COLIBRI constraint solver.

We performed experiments for different options of SANTE and compared them with each other, and with a dynamic analysis technique that applies test generation for all potential threats (without filtering by value analysis and slicing) and considers each threat as an alarm. We use five examples (up to several hundreds of lines of C code) extracted from real-life software where bugs were previously detected. All bugs are out-of-bound accesses or invalid pointers.

The results confirmed the interest of the proposed combination. In particular, thanks to filtering threats by value analysis, the diagnosis of alarms by testing in SANTE becomes faster (in average, by 43%) compared to dynamic analysis without filtering. Program slicing significantly reduces the program (with an average reduction rate of 32%), that accelerates test generation. Moreover, the average length of program paths decreases after the use of slicing by 19%. Simpler counterexamples reported for reduced programs may help the validation engineer to more easily understand the reason of the error and fix it. This simplification can be particularly valuable for automatically generated code when the engineer does not have a deep knowledge of the resulting C source code.

In our journal paper in *Autom. Softw. Eng.* [12], we describe design and implementation aspects of SANTE and underlying tools, discuss the tool architecture and present some experiments. This publication, that I initiated and coordinated, was the first systematic description of a significant subset of plugins and the architecture of FRAMA-C. It was later extended to a foundational paper describing the kernel and the main modules of FRAMA-C at SEFM'12 [50] and its later journal version in *Formal Asp. Comput.* [11].

We also worked on a formal proof of soundness of the SANTE method. This effort rapidly met an obstacle at that time: the lack of theoretical results on program slicing in the context of programs with potentially non-terminating or erroneous statements. Such results would be necessary to make a formal link between the presence or the absence of errors in the slice and in the initial program. We had to assume without proof a suitable soundness property [156, Proposition 4.4.1] of program slicing in order to perform the proof of soundness of the method. A sketch of proof relying on this conjecture was included in the PhD thesis of Omar Chebaro [156]. My concern to perform a complete proof of soundness of the method motivated a later research project (described below in Section 2.4) that allowed to formally establish the assumed property of slicing.

2.3.4 Related Work

Many static and dynamic analysis tools have been proposed and widely used in practice separately. Static analysis tools (e.g. value analysis of FRAMA-C [11], POLYSPACE [92, 229], ASTRÉE [202]) are often based on abstract interpretation [244, 262] or predicate abstraction [230]. Dynamic analysis tools, such as PATHCRAWLER [207], DART/CUTE [200, 205], SAGE [181], EXE [195], KLEE [179], PEX [186], automatically generate program inputs satisfying symbolic constraints collected by symbolic execution.

Some combinations of static and dynamic analyses for program verification were previously studied. DAIKON [191] uses dynamic analysis to detect likely invariants. CHECK'N'CRASH [193, 203] applies static analysis that reports alarms. It uses intraprocedural weakest-precondition computation (the ESC/JAVA tool [213]) rather than value analysis, so it necessitates code annotations. Next, random test generation (with JCRASHER) tries to confirm the bugs. SANTE uses an interprocedural value analysis that necessitates only a precondition, and *all-path* test generation, that may in addition prove some alarms unreachable.

DSD CRASHER [193] applies DAIKON [191] to infer likely invariants before the static analysis step of CHECK'N'CRASH to reduce the false alarms rate. This method admits generated invariants that may be wrong and can result in proving some real bugs as safe, unlike SANTE which is sound and never reports a bug as safe.

SYNERGY [198], BLAST [188] and the approach of Pasareanu et al. [204] combine testing and partition refinement for property checking. SANTE is relative to the YOGI tool that implements the algorithm DASH [178], initially called SYNERGY. In SANTE, we use value analysis whereas YOGI uses weakest precondition with template-based refinement. Both tools track down error states. They are specified as an input property in YOGI whereas in SANTE they are automatically computed by value analysis. YOGI does not use program slicing. It iteratively refines an over-approximation using information on unsatisfiable constraints from test generation. Its approach is more adapted for one error statement at a time, while SANTE can be used on several alarms simultaneously. Pasareanu et al. [204] combine predicate abstraction and test generation in a refinement process, guided by the exactness of the abstraction with respect to operations of the system rather than by test generation. The approach of Ge et al. [159] is similar to the first version of SANTE [55] where some irrelevant code is excluded before dynamic analysis for CFG connectivity reasons, that is weaker than program slicing.

Christakis et al. [148] propose a technique ralated to SANTE but assume that static analysis can be unsound, while SANTE considers sound static analysis. Chimento et al. [128] propose a combination of static analysis with runtime verification rather than test generation.

Aïssat et al. [119, 120] propose a provably correct technique for pruning infeasible paths using symbolic execution.

Advanced strategies for the integration of program slicing into a combination of value analysis and test generation for C program debugging proposed in this work were not previously studied by other authors.

Other combinations of analyses, related to deductive verification and our work on the STADY method, are presented below in Section 3.2.7.

2.3.5 Application to Detection of Security Vulnerabilities

More recently, in the context of the European FP7 project STANCE in collaboration with industrial partners (Dassault Aviation and Search Lab, Budapest), the SANTE method was extended to detection of security flaws and successfully applied to the Heartbleed vulnerability detected in OpenSSL in 2014. This work was presented in our HVC'15 paper [37].

In addition to value analysis, the extended method uses taint analysis to identify a subset of alarms that are most likely to lead to attacks (by tainting variables potentially controlled by an attacker and therefore representing a high risk to introduce malicious behavior). Again, slicing is used to reduce the source code by removing statements that are irrelevant with respect to the identified subset of alarms. These two steps help to focus on security-relevant alarms in the last step. Finally, a fuzz testing step is applied on the reduced code in order to try to confirm the selected alarms. An important benefit of the method for industrial applications is its capacity to detect bugs with reasonable efforts, e.g. without providing a detailed specification of the input state or additional annotations in the code.

Another application of this methodology is currently in progress in the ongoing European H2020 project VESSEDIA targeting in particular formal verification of IoT (Internet of Things) software.

2.4 Formalization of Program Slicing

During our work on the proof of soundness of the SANTE method, we met an unexpected issue related to the lack of theoretical foundations of slicing for programs with errors and non-termination. The application of program slicing in the context of SANTE and, more generally, the need to establish the soundness of combined verification methods using slicing motivated our study of program slicing and the PhD work of Jean-Christophe Léchenet in 2015–2018 co-supervised in collaboration with Pascale Le Gall (CentraleSupélec). The main results of this work were published in our JFLA'16 [65] and FASE'16 [31] papers, and an extended journal version in *Formal Asp. Comput.* [8].

2.4.1 Motivation

Program slicing was initially introduced by Weiser [256, 258] as a technique allowing to transform a given program p into a simpler one, called a program slice, by analyzing its control and data flow. In the classic definition, a *(program) slice* is an executable program subset of the initial program whose behavior must be identical to a specified subset of the initial program's behavior. This specified behavior that should be preserved in the slice is called the *slicing criterion*. A common slicing criterion is a program point l. Informally speaking, program slicing with respect to the criterion l should guarantee that any variable v at program point l takes the same value in the slice and in the original program. Examples of slices for a toy program are given below in Figure 2.2.

Most previous applications of slicing to debugging used slices in order to *better understand an already detected error,* by analyzing a simpler program rather than a more complex one [152, 194, 238]. Our goal is quite different: to perform verification and validation (V&V) on slices in order to discover yet unknown errors, or show their absence. But can the results obtained on the slices be transposed to the initial program directly? In other words, our goal was to address the following research question:

(RQ) How can we soundly conduct V&V activities on slices instead of the initial program? In particular, if there are no errors in a program slice, what can be said about the initial program? And if an error is found in a program slice, does it necessarily occur in the initial program?

The interpretation of the absence or presence of errors in a slice in terms of the initial program requires solid theoretical foundations. The link between the original program and its slice is made by a soundness property relating their semantics. The classic soundness property of slicing (cf. [164, Def. 2.5] or [251, Slicing Th.]) can be informally stated as follows.

Proposition 2.1 (Classic soundness property of slicing) *Let p be a program, q a slice of p, and let σ be an input state of p. Suppose that p halts on σ. Then q halts on σ and the executions of p and q on σ agree after each statement preserved in the slice on the variables that appear in this statement.* [5]

[5] Formally, this "agreement" of traces is expressed as an equality of projections of the traces in p and q on a

This property actually states that the program and its slice have the same behavior not only with respect to the slicing criterion, but also with respect to all statements preserved in the slice. For this reason, it seems desirable to apply slicing in order to detect certain behaviors of the original program by studying the slices rather than the original program itself.

Unfortunately, the classic soundness property cannot be used to answer the research question **(RQ)**. Indeed, it was originally established for classic dependence-based slicing for programs without runtime errors and only for executions with terminating loops: nothing is guaranteed if p does not terminate normally on σ.

Dealing with potential runtime errors and non-terminating loops is very important for realistic programs since their presence cannot be a priori excluded, especially during V&V activities. Although quite different at first glance, both situations have a common point: they can in some sense interrupt normal execution of the program preventing the following statements from being executed. Therefore, slicing away (that is, removing) potentially erroneous or non-terminating sub-programs from the slice can have an impact on the link between the original program and a slice.

While some aspects of **(RQ)** were discussed in previous papers (cf. Section 2.4.4), none of them provided a complete formal answer in the considered general setting, for programs with errors and non-termination.

2.4.2 Illustrative Examples

Figure 2.2a presents a simple (buggy) C-like program that takes as inputs an array `a` of length `N` and an integer `k` (with $0 \leqslant$ `k` $\leqslant 100$, $0 \leqslant$ `N` $\leqslant 100$), and computes in two different ways the average of the elements of `a`. We suppose that all variables and array elements are unsigned integers (so an out-of-bound array access `a[i]` can only be due to a too big index `i`). We also assume that all elements of `a` whose index is not a multiple of `k` are zeros, so it suffices to sum array elements over the indices multiples of `k` and to divide the sum by `N`. The sum is computed twice (in `s1` at lines 3–8 and in `s2` at lines 9–16), and the averages `avg1` and `avg2` are computed (lines 17–20) and compared (lines 21–22). We assume that necessary assertions with explicit guards (at lines 5, 10, 13, 17, 19) are inserted to prevent potential runtime errors.

Figure 2.2b shows a (classic dependence-based) slice of this program with respect to the statement at line 18. Intuitively, it contains only statements (at lines $1, 3, 4, 6, 7, 18$) that can influence the slicing criterion, i.e. the values of variables that appear at line 18 after its execution. In addition, we keep the assertions to prevent potential errors in preserved statements. Similarly, Figure 2.2c shows a slice with respect to line 20, again with protecting assertions.

Figure 2.3 summarizes the behavior of the three programs of Figure 2.2 on some test data, denoted $\sigma_1, \ldots, \sigma_5$. The elements of `a` do not matter here and can be chosen arbitrarily (re-

suitable sets of instructions and variables [164].

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k ≠ 0);
11 last = N/k; //must be (N-1)/k
12 while (j ⩽ last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N ≠ 0);
18 avg1 = s1 / N;
19 assert (N ≠ 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

(a)

```
1  s1 = 0;
2
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9
10
11
12
13
14
15
16
17 assert (N ≠ 0);
18 avg1 = s1 / N;
19
20
21
22
```

(b)

```
1
2  s2 = 0;
3
4
5
6
7
8
9  j = 0;
10 assert (k ≠ 0);
11 last = N/k;
12 while (j ⩽ last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17
18
19 assert (N ≠ 0);
20 avg2 = s2 / N;
21
22
```

(c)

Figure 2.2 – A program and two slices illustrating the relations between errors in the slices and in the initial program. **(a)** A program computing in two ways the average of elements of a given array a of size N whose only nonzero elements can be at indices $\{0, k, 2k, \ldots\}$ divisible by k; **(b)** its slice w.r.t. line 18; and **(c)** its slice w.r.t. line 20. Explicit assertions are inserted to prevent any potential runtime errors.

specting the assumption that a contains zeros at the indices that are not multiples of k). For convenience of the reader, the behavior of the three programs is schematically illustrated by the last line of Figure 2.3. Let us discuss what these examples illustrate for our purpose: verification and validation on slices.

Test datum σ_1 shows the case where neither of the three programs produces an error on this input. Suppose now we execute slice **(b)** on test datum σ_2 without detecting any error. This does not mean that program **(a)** has none: we see that **(a)** fails on line 13, not preserved in slice **(b)**.

Next, suppose we found an error at line 17 in slice **(b)** provoked by test datum σ_4. Program **(a)** does not contain the same error: it fails earlier, at line 13. We say that the error at line 17 in slice **(b)** is *hidden by the error* at line 13 of the initial program. Similarly, test datum σ_5 provokes an error at line 17 in slice **(b)** while this error is hidden by an error at line 10 in **(a)**. In fact, the error at line 17 cannot be reproduced on the initial program **(a)** (since either line 10 or line 13 will necessarily fail before). We say that it is *totally hidden* by other errors.

State		σ_1	σ_2	σ_3	σ_4	σ_5
Inputs		$k=2$ $N=5$	$k=2$ $N=4$	$k=0$ $N=4$	$k=2$ $N=0$	$k=0$ $N=0$
Anomaly	**(a)**	—	↯ line 13	↻ line 4	↯ line 13	↯ line 10
	(b)	—	—	↻ line 4	↯ line 17	↯ line 17
	(c)	—	↯ line 13	↯ line 10	↯ line 13	↯ line 10

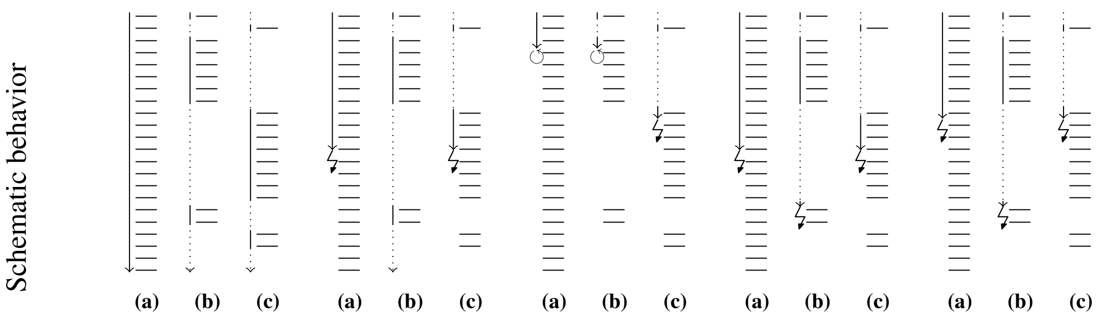


Figure 2.3 – Errors (↯), non-termination (↻) and normal termination (—) of programs of Figure 2.2 for some inputs

For slice **(c)**, detecting an error at line 10 on test datum σ_5 would allow us to observe the same error in **(a)**. However, while this error in slice **(c)** is also provoked by test datum σ_3, this test datum does not provoke any error in **(a)** because the loop at line 4 does not terminate. We say that this error in slice **(c)** is *(partially) hidden by a non-termination* of the loop at line 4.

These examples clearly show that Property 2.1 is not true in the presence of errors or non-terminating loops for classic slices. Indeed, the executions of program p and slice q may disagree at least for two reasons:

(i) a previously executed non-terminating loop not preserved in the slice, or

(ii) a previously executed failing statement not preserved in the slice.

Finally, let us mention another example related to error-free programs. If we suppose that $0 < \texttt{k} \leqslant 100$, $0 < \texttt{N} \leqslant 100$, and replace `N/k` by the correct expression `(N-1)/k` at line 11 in programs **(a)** and **(c)** of Figure 2.2, neither slice contains any error. Suppose we manage to verify the absence of errors on both slices. Can we be sure that the initial program is error-free as well?

2.4.3 Approach and Main Results

We consider a realistic setting of programs with possible errors and with potentially non-terminating loops, even if this non-termination is unintended. We assume neither that all loops terminate, nor that all loops do not terminate, nor that we have a preliminary knowledge of which loops terminate and which loops do not. We consider errors determined by the current program state such as runtime errors (that can either interrupt the program or lead to an undefined behavior). We assume that all threatening statements are annotated with explicit assertions **assert**(C) placed before them, that interrupt the execution [6] whenever the condition C is false. This assumption is convenient for the formalization: possible runtime errors can only occur in assertions. Such assertions can be generated syntactically (for example, using the RTE plugin of FRAMA-C [11]). For instance, line 10 in Figure 2.2a prevents division by zero at line 11, while line 13 makes explicit a potential runtime error at line 14 if the array a is known to be of size N. In addition, the **assert**(C) keyword can also be used to express additional user-defined properties on the current state.

One possible way to satisfy the traditional soundness property (Proposition 2.1) is to consider additional dependencies of each statement on previous loops and error-prone statements. This solution was previously adopted by several authors [192, 235, 246]. It would however lead to larger slices, where we would *systematically preserve all potentially erroneous or non-terminating statements* executed before the slicing criterion. Such slices would have a very limited benefit for our purpose of performing V&V on slices instead of the initial program. That is why this solution is not satisfactory for our goal. For instance, to ensure that the executions of program **(a)** and slice **(b)** agree on all statements of slice **(b)** (and to avoid the issue illustrated by test datum σ_4), line 13 should be preserved in slice **(b)**. By transitivity of dependencies, that would result in keeping e.g. the loop at line 12 and lines 9–11 in slice **(b)** as well. Similarly, the loop at line 4 should be kept in slice **(c)** to avoid disagreeing executions due to non-termination of the loop (e.g. for test datum σ_3).

This work follows another approach and proposes *relaxed slicing,* a slicing technique where additional dependencies on previously executed (potentially) erroneous or non-terminating statements are not required. Regarding assertions, an assertion is preserved in a relaxed slice when the threatening statement it protects is preserved in the slice (so all threatening statements in the slice are still protected by explicit assertions, like in the slices in Figure 2.2). This approach leads to smaller slices, but needs a different soundness property. We prove a new soundness

[6] To ensure that the execution is interrupted before an error occurs, in this formalization we use assert statements rather than ACSL-like assert annotations that would be ignored by the execution. This is a minor technical difference since the former can be generated from the latter.

property that can be informally stated as follows.

Theorem 2.2 (Soundness of a relaxed slice) *Let q be a relaxed slice of p, and let σ be an initial state. Then there exists a prefix T' of the execution of q on σ that agrees with the (whole) execution T of p on σ after each statement preserved in the slice on the variables that appear in this statement.*

This new property does not assume normal termination of program p, and only relates the execution of the original program with some prefix of the execution of the slice. Thus, considered on the statements preserved in slice q[7], the execution of q can be longer than that of the complete program p. It is illustrated by the examples of Figure 2.3.

Theorem 2.2 basically establishes the same soundness property that was assumed earlier in the PhD thesis of Omar Chebaro [156, Proposition 4.4.1] as a conjecture for the proof of soundness of SANTE. This soundness result allows us to justify V&V on slices by characterizing possible verification results on slices in terms of the initial program. Let us state first the results on the absence of errors.

Theorem 2.3 *Let q be a relaxed slice of p. If all assertions contained in q never fail, then the corresponding assertions in p never fail either.*

Corollary 2.4 *Let $q_1, \ \ldots, \ q_n$ be relaxed slices of p such that each assertion in p is preserved in at least one of the q_i. If no assertion in any q_i fails, then no assertion fails in p.*

In other words, Theorem 2.3 allows us to deduce the absence of errors in the initial program from the absence of errors in its slices, but only for the preserved instructions. If all instructions are preserved in at least one slice, then Corollary 2.4 deduces the absence of errors in the whole initial program from the absence of errors in the slices.

In particular, consider again the corrected version of the example of Figure 2.2 (where we suppose that $0 < \texttt{k} \leqslant 100$, $0 < \texttt{N} \leqslant 100$, and fix line 11 with the correct expression `(N-1)/k` in programs **(a)** and **(c)**). In this case, if we show that both slices do not contain any errors, Corollary 2.4 allows us to deduce that the original program is error-free as well.

The last result justifies the detection of errors in a relaxed slice.

Theorem 2.5 *Let q be a relaxed slice of p and σ an initial state of p. We assume that the execution of q on σ ends with an error. Then one of the following cases holds for p:*

($\dagger$) *the execution of p on σ ends with an error at the same statement,*

($\dagger\dagger$) *the execution of p on σ ends with an error at a statement not preserved in q,*

($\dagger\dagger\dagger$) *the execution of p on σ contains a non-terminating loop not preserved in q.*

[7] formally speaking, projected to the statements preserved in slice q.

This result shows that the cases illustrated above by Figure 2.3 are the only possible cases. In other words, if an error occurs in a slice executed on some initial state but does not occur in the original program executed on the same initial state, this error is hidden either by an earlier error or non-termination in a non-preserved statement.

When an error is detected in a slice for some test datum, it is very easy to check the behavior of the initial program for the same test datum. In some sense, for programs supposed to terminate within a given time T, this property confirms the interest of verification on relaxed slices: even if an error detected in a slice is not confirmed as an error in the same statement in the initial program, the detection of this error was not a waste of time as it will necessarily lead to detecting another anomaly in the initial program: either another error or a non-termination within time T.

Relaxed slicing, its soundness property, and the justification of its use in V&V have been formalized in the COQ proof assistant [211] for a simple language representative for our purpose: the language WHILE with errors and possibly non-terminating loops. A certified implementation of relaxed slicing is automatically extracted from the COQ formalization.

Summary of Main Results. The main contributions of this work include:

— a comprehensive analysis of issues arising for V&V on classic slices;

— the notion of relaxed slicing for structured programs with possible errors and non-termination, which keeps fewer statements than it would be necessary to satisfy the classic soundness property of slicing;

— a new soundness property for relaxed slicing (Theorem 2.2);

— a characterization of verification results, such as absence or presence of errors, obtained for a relaxed slice, in terms of the initial program, that constitutes a theoretical foundation for conducting V&V on slices (Theorems 2.3, 2.5);

— formalization and proof of our results in COQ, providing a certified implementation of relaxed slicing extracted from the COQ formalization.

These contributions are detailed in our FASE'16 paper [31] and its extended journal version in *Formal Asp. Comput.* [8].

2.4.4 Related Work

Weiser [256, 258] introduced the basics of intraprocedural and interprocedural static slicing. A thorough survey provided by Tip [238] explores both static and dynamic slicing, compares different approaches and lists the application areas of program slicing. More recent surveys can be found [152, 208, 212]. Foundations of program slicing have been studied by several authors e.g. [158, 164, 177, 192, 194, 242, 247, 248, 250, 251]. This section presents a selection of efforts that are most closely related to the goals of this work.

Debugging and dynamic slicing. Program debugging and testing are traditional application domains of slicing (e.g. [226, 241, 256]) where it can be used to better understand an already detected error, to prioritize test cases (e.g. in regression testing), simplify a program before testing, etc. In particular, dynamic slicing [194] is used to simplify the program for a given (e.g. erroneous) execution. However, theoretical foundations of applying V&V on slices instead of the initial program in the presence of errors and non-termination have been only partially studied.

Slicing and non-terminating programs. A few works tried to propose a semantics preserved by classic slicing even in the presence of non-termination. Among them, we can cite the lazy semantics of Cartwright and Felleisen [247], and the transfinite one of Giacobazzi and Mastroeni [220], improved by Nestra [174]. Another semantics proposed by Barraclough et al. [164] has several improvements compared to the previous ones. Despite the elegance of these proposals, they turn out to be unsuitable for our purpose because they consider non-existing trajectories, that are not adapted to V&V techniques, for example, based on path-oriented testing like in the SANTE method or the work of Ge et al. [159]. Dealing with real, non-modified executions, that can be effectively found by testing, appears to be necessary for our goal.

Ranganath et al. [192] provide foundations for the slicing of modern programs, i.e. programs with exceptions and potentially infinite loops, represented by control flow graphs (CFG) and program dependence graphs (PDG). Their work gives two definitions of control dependence, non-termination sensitive and non-termination insensitive, corresponding respectively to the weak and strong control dependences of Podgurski and Clark [246] and further generalized for any finite directed graph by Danicic et al. [158]. Ranganath et al. [192] also establish the soundness of classic slicing with non-termination sensitive control dependence in terms of weak bisimulation, more adapted to deal with infinite executions. Their approach requires to preserve all loops, that results in much bigger slices than in relaxed slicing.

Amtoft [177] establishes a soundness property for non-termination insensitive control dependence in terms of simulation. Ball and Horwitz [242] describe program slicing for arbitrary control flow. Both works state that an execution in the initial program can be a prefix of that in a slice, without carefully formalizing runtime errors. Our work establishes a similar property, and in addition performs a complete formalization of slicing in the presence of errors *and* non-termination, explicitly formalizes errors by assertions and deduces several results on performing V&V on slices.

Slicing in the presence of errors. Harman et al. [235] note that classic algorithms only preserve a lazy semantics. To obtain correct slices with respect to a strict semantics, it proposes to preserve all potentially erroneous statements through adding pseudo-variables in the $\mathrm{def}(l)$ and $\mathrm{ref}(l)$ sets of all potentially erroneous statements l. Our approach is more fine-grained in the sense that we can independently select assertions to be preserved in the slice and to be considered by V&V on this slice. This benefit comes from our dedicated formalization of errors with assertions and a rigorous proof of soundness using a trajectory-based semantics. In addition, we make a formal link about the presence or the absence of errors in the program and its slices. Harman and Danicic [237] use program slicing as well as meaning-preserving transformations to analyze a property of a program not captured by its own variables. For that, their method adds variables and assignments in the same idea as our assertions. Allen and Horwitz [215] extend data and control dependences for Java program with exceptions. In both papers, no formal justification is given. Another seemingly related work by Barros et al. [146] focuses on assertion-based slicing. But in their work, the meaning of "assertion" is quite different, since it refers to contracts of functions, and assertion-based slicing is another term for specification-based slicing.

Certified slicing. The ideas developed by Amtoft and Ranagath et al. [177, 192] were applied by Blazy et al. [127] and Wasserrab [162]. Wasserrab [162] builds a framework in Isabelle/HOL to formally prove a slicing defined in terms of graphs, therefore language-independent. Blazy et al. [127] propose an unproven but efficient slice calculator for an intermediate language of the CompCert C compiler [173], as well as a certified slice validator and a slice builder written in COQ [211]. The modeling of errors and the soundness of V&V on slices were not specifically addressed in these works.

To the best of our knowledge, the present work is the first complete formalization of program slicing for structured programs in the presence of errors and non-termination. Moreover, it has been formalized in the COQ proof assistant on a representative structured language, that provides a certified program slicer and justifies conducting V&V on slices instead of the initial program.

2.5 Computation of Arbitrary Control Dependencies

After the formalization of relaxed slicing and justification of the usage of slicing during V&V for a simple WHILE language, a natural research direction is a generalization of these results to a large real-life language with unstructured control flow. It would require to formalize control and data dependencies for the target language, and can strongly depend on this specific language.

An even more ambitious long-term research goal would be to provide a generic program slicing, applicable to various input languages, possibly with unstructured control flow. An advance in this direction was done by the second part of the PhD work of Jean-Christophe Léchenet by providing a new efficient algorithm for computation of control dependencies in a very general setting of arbitrary directed graphs, and its mechanized proof. The main results of this effort were published in FASE'18 [17] and presented in French conference JFLA'18 [64].

2.5.1 Motivation

Control dependence is a fundamental notion in software engineering and analysis (e.g. [234, 246, 252, 253, 258, 263]). It reflects structural relationships between different program statements and is intensively used in many software analysis techniques and tools, such as compilers, verification tools, test generators, program transformation tools, simulators, debuggers, etc. Along with data dependence, it is one of the key notions used in program slicing.

In 2011, Danicic et al. [158] proposed an elegant generalization of the notions of closure under non-termination insensitive (*weak*) and non-termination sensitive (*strong*) control dependence. They introduced the notions of weak and strong control-closures, that can be defined on any directed graph, and no longer only on control flow graphs. They proved that weak and strong control-closures subsume the closures under all forms of control dependence previously known in the literature. In the present work, we are interested in the non-termination insensitive form, i.e. *weak control-closure*, as it corresponds to the approach of relaxed slicing.

Besides the definition of weak control-closure for any directed graph, Danicic et al. also provided an algorithm computing the closure for a given set of vertices. This algorithm was proved by paper-and-pencil. Under the assumption that the given graph is a CFG (or more generally, that the maximal out-degree of the graph vertices is bounded), the complexity of the algorithm can be expressed in terms of the number of vertices n of the graph, and was shown to be $O(n^3)$. This may explain why this algorithm was not used until now. Danicic et al. themselves suggested that it should be possible to improve its complexity.

The main motivation of this work was the design of a more efficient algorithm for control closure computation, and its mechanized proof avoiding any risk of error.

2.5.2 Approach and Main Results

Danicic et al. introduced basic notions used to define weak control-closure and to justify the algorithm, and proved a few lemmas about them. While formalizing these concepts in the COQ proof assistant [110, 211], we have discovered that, strictly speaking, the paper-and-pencil proof of one of them [158, Lemma 53] is inaccurate (a previously proven case is applied while its hypotheses are not satisfied), whereas the lemma itself is correct. We have fixed this issue in our proof.

Danicic's algorithm does not take advantage of its iterative nature and does not reuse the results of previous iterations in order to speed up the following ones. Our main objective was to design a more efficient algorithm sharing information between iterations to speed up the execution. We have proposed a new algorithm that is carefully optimized and more complex. Therefore, its correctness proof relies on more subtle arguments than for Danicic's algorithm. To deal with them and to avoid any risk of error, we used again a mechanized verification tool — this

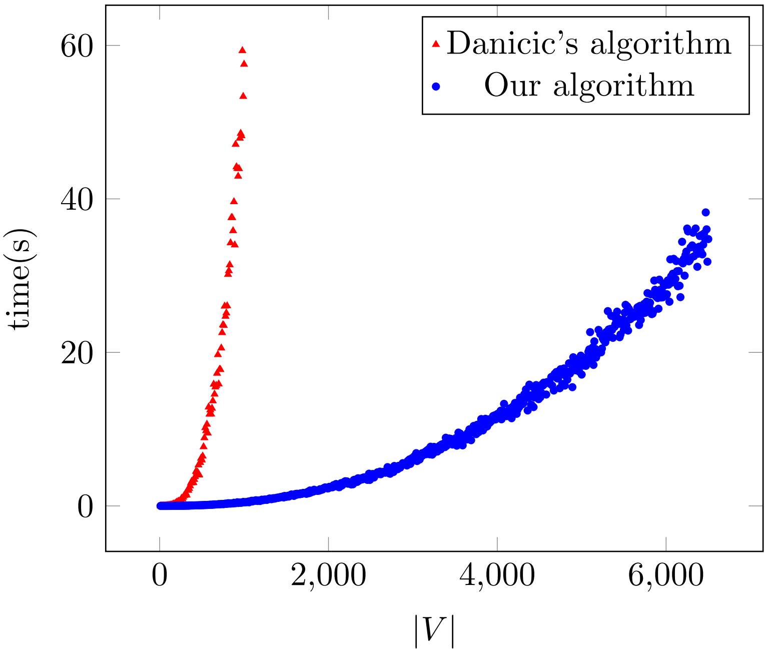


Figure 2.4 – Danicic's vs. our algorithm: experiments

time, the WHY3 proof system [108, 142] — to guarantee correctness of the optimized version.

Finally, in order to evaluate the new algorithm with respect to Danicic's initial technique, we have implemented both algorithms in OCaml (using OCamlgraph library [189]) and intensively tested them on a large set of randomly generated graphs with up to thousands of vertices. Figure 2.4 shows the computation time (in seconds) recorded for different sizes of the set of nodes V of the generated graphs. Experiments demonstrate that the proposed optimized algorithm is applicable to large graphs (and thus to CFGs of real-life programs) and significantly outperforms Danicic's original technique.

Summary of Main Results. The contributions of this work include:

- — A formalization of Danicic's algorithm and proof of its correctness in COQ;
- — A new algorithm computing weak control-closure and taking benefit from preserving some intermediary results between iterations;
- — A mechanized correctness proof of this new algorithm in the WHY3 tool including a formalization of the basic concepts and results of Danicic et al.;
- — An implementation of Danicic's and our algorithms in OCaml, their evaluation on random graphs illustrating that the new algorithm significantly outperforms Danicic's original method.

These contributions are detailed in our FASE'18 paper [17].

2.6 Summary

This chapter has illustrated the evolution of my research interests from first projects on software testing and constraint solving to combined static-dynamic verification methods. They were guided by the intention to combine different techniques in order to take benefit from the capacities of each of them. We have proposed a combination of value analysis, slicing and test generation in the SANTE method.

Another motivation of my research was the design of sound and efficient methods with well-understood properties. This motivation lead to considering various options of the SANTE method, investigating the properties of alarm dependencies and working on the proof of the method. This proof further lead to studying the properties of program slicing and to their mechanized proof.

Finally, the ambition to generalize slicing to a larger programming language motivated the study of control dependence and the design of a new, more efficient and provably correct method to compute control closure — a significant step towards a generic, provably correct slicer.

Chapter 3

From Executable Specifications to Counterexamples

Combinations of deductive verification and dynamic analysis require a common specification language to express program properties. ACSL [109], the specification language of FRAMA-C, was originally intended for static analysis, and some of its features cannot be evaluated by dynamic analysis techniques.

First, this chapter briefly presents our work on an executable specification language, called E-ACSL, and a FRAMA-C plugin for runtime verification, called E-ACSL2C in this document [1]. This work was done in collaboration with Julien Signoles (CEA List), the main author of the E-ACSL language and the E-ACSL2C plugin. This effort was mainly conducted in the context of two postdocs, Mickaël Delahaye [2] and Kostyantyn Vorobyov [3], and two internships, Guillaume Petiot [4] and Arvid Jakobsson [5], that we co-supervised with Julien. I strongly contributed to the design of combined techniques using E-ACSL, solutions for efficient memory monitoring and optimizations of E-ACSL2C using static analysis. E-ACSL2C and the underlying techniques evolved very rapidly during the last five years. The main principles of a common specification language appeared in our SAC'13 paper [45], and were later discussed in a broader context in our ISOLA'16 paper [71]. The design, optimizations and evaluation of the runtime verification tool E-ACSL2C were presented in our papers in RV'13 [46], French conference JFLA'15 [66], SAC'15 [36] and its journal version in *Sci. Comput. Prog.* [10], RV'17 [27], RVCuBES'17 [26] and TAP'18 [21]. The most recent design of the memory model, based on shadow memory, was presented in ISMM'17 [28], and gave rise to the submission of two patents [5, 6] in Europe and USA.

[1] to avoid any risk of confusion (this plugin is also called E-ACSL in other documents)

[2] currently, engineer in software verification

[3] currently, engineer at Coverity

[4] later, PhD student at CEA List

[5] later, PhD student at Huawei France and LIFO, Univ. of Orléans

The next part of this chapter presents in more detail a combined verification methodology using E-ACSL language, where test generation helps to identify the reason of a proof failure obtained during deductive verification, and to exhibit a counterexample. It is based on the results obtained during (and following) the PhD work of Guillaume Petiot[6], conducted from October 2012 to November 2015 and co-supervised in collaboration with Alain Giorgetti and Jacques Julliand (University of Franche-Comté, Besançon). Bernard Botella and Julien Signoles (CEA List) also contributed to some parts of this work. Guillaume Petiot defended his thesis [132] on November 4, 2015. It lead to several publications. The original idea of the combination was published in TAP'14 [42]. The underlying code transformation was formalized in our SCAM'14 paper [41] and implemented as a FRAMA-C plugin STADY [105]. Finally, the most complete classification of proof failures, the corresponding detection techniques and experiments were presented in TAP'16 [33] and its extended journal version at *Formal Asp. Comput.* [9].

Finally, the last part of this chapter briefly presents our work on the support of relational properties in FRAMA-C, where we proposed a solution for specification and verification of relational properties based on self-composition. It is suitable for both static and dynamic verification, in particular, using WP, E-ACSL2C and STADY. This activity was conducted in the context of the PhD work of Lionel Blatter[7], started in October 2015 and co-supervised in collaboration with Pascale Le Gall (CentraleSupélec) and Virgile Prevosto (CEA List). The main results of this work (so far) were published in our papers in TACAS'17 [23] and TAP'18 [16].

3.1 Executable Specification and Runtime Verification: the E-ACSL2C tool

3.1.1 Motivation

While some static and dynamic analysis techniques (like value analysis and test generation in the SANTE method described in Section 2.3) can be combined without sharing an expressive specification language, that is not possible for other analysis techniques. Deductive verification takes as input a program with a formal specification and thus, its combination with dynamic analysis requires a common specification language supported by both techniques. The specification language of FRAMA-C, ACSL [109], was not originally designed for dynamic analysis and contains features that cannot be evaluated at runtime. They include unbounded quantifications, lemmas and axiomatics. One motivation of this work was to provide a common *executable* specification language, suitable for both static and dynamic analysis techniques.

Another motivation was to provide a runtime verification tool, capable to check the annotations at runtime and report failures. Such a tool should be able to support a large class of

[6] currently, engineer at Systerel, Paris area

[7] currently, temporal teaching assitant at Univ. Paris-Sud, Orsay

properties. In particular, for the C language, it should deal with potential runtime errors (or undefined behavior) in C programs. The monitoring of properties of pointers and memory locations (such as being valid, initialized, in a particular block, with a particular offset, etc., cf. Section 1.3.3) is not straightforward and requires to systematically record and maintain metadata for allocated locations.

Next, this runtime verification tool should be efficient. Systematic monitoring of memory operations can be inefficient for several reasons. Let us give two examples. First, the performance strongly depends on the design of the database where the metadata for allocated memory locations are recorded. The speed of update and lookup operations can have a dramatic impact on the global performances of the tool. Second, an exhaustive monitoring of all locations can be costly and useless if only some memory locations are actually involved in the properties to evaluate. Identifying irrelevant locations that should be excluded from the monitoring is a good way to optimize the performances of the tool. This step can be done by static analysis.

Finally, the runtime verification tool should be sound: the provided output for a given execution (a failure or absence of failures) should be correct. But to keep the tool sound, the optimizations should rely on sound analysis techniques.

3.1.2 Approach and Main Results

We proposed E-ACSL [45], an executable subset of ACSL, as a common specification language for static and dynamic analysis. It comes with an automatic translator of annotations into C implemented in the E-ACSL2C plugin of FRAMA-C. The tool transforms a given program P with E-ACSL annotations into a new program P' — an instrumented version of P — such that if some annotation fails during an execution of P, this failure is detected and reported during the execution of P' on the same input. If P does not fail, the execution of P' is equivalent to that of P. Thus, P' performs *runtime assertion checking* for annotations in P.

As mentioned above, a systematically instrumented code can be quite verbose and inefficient. We proposed several optimizations of the translation tool to avoid these issues. One of them reduces the amount of instrumented code using a static dataflow analysis step that over-approximates the set of variables that must be monitored to evaluate the given annotations. We defined the corresponding data-flow analysis and evaluated its benefits [10, 66].

Other optimizations aimed at improving the performances of the underlying memory monitoring algorithms and datastructures. We proposed several original solutions for an efficient storage of metadata of allocated memory blocks, first based on Patricia tries [46], then on a combination of Patricia tries with shadow memory [10, 36]. Finally we managed to create a powerful and efficient technique based on shadow memory only [28, 5, 6].

One interesting problem with memory errors is related to the detection of temporal errors. State-of-the-art memory debuggers [151, 154, 206, 245] and the first versions of E-ACSL2C have become efficient in detecting *spatial* memory errors (accesses to unallocated memory). They track memory allocated by a program at runtime and report an error if a program attempts to access a memory location that is currently not tracked as valid. Memory debuggers, however, cannot detect uses of invalid pointers in all cases. Consider, for instance, the following code snippet.

```
1 int *p = malloc(sizeof(int));
2 free(p);
3 int *q = malloc(sizeof(int));
4 *p = 1;
```

Execution of the assignment at line 4 leads to an error. This is because the block initially pointed to by p has been deallocated (at line 2) making p a stale pointer to unallocated memory. In a typical execution, however, the second call to malloc at line 3 can reuse the freed space and place the newly allocated block (at address q) at the same location as the first one, implicitly making p point to the allocated memory. A memory debugger analysing the code observes an access to allocated memory at line 4 and does not raise an alarm. Such behavior, however, is not enforced by ISO C semantics: another execution can put the second allocation in a different area and the access at line 4 will become invalid. Dereference of a pointer to an allocated memory block that has not been made point to that block leads to a *temporal* memory error.

We proposed a solution to this problem based on the storage of

- a unique identifier, called *origin number*, for each allocated block, and
- an additional metadata, called *referent number*, for each pointer p, defined as the origin number of the block that pointer p points to at the time of creating the reference.

The proposed technique to detect a temporal error during a dereference of a pointer p basically relies on comparing the referent number of p and the origin number of the block pointed by p. If the referent number of the pointer is different from the origin number of the pointed block, a temporal error is reported: the pointer does not refer any more to the same block as at the time of creating the reference.

In the example above, assume the blocks allocated at lines 1 and 3 receive origin numbers n_1 and n_2. Then pointer p receives referent number n_1 at line 2 (as it refers to a block with origin number n_1 at that time). After the deallocation of the first block at line 2, this block is not recorded any more as a valid block with origin number n_1, so a mismatch is necessarily detected at line 4: the referent number n_1 of pointer p is not equal any more to the origin number of the pointed block at that time. Indeed, even if by chance the same block was allocated at line 3 for q, it would have a different origin number n_2. This technique, whose first prototype was proposed during the internship of Arvid Jakobsson, was presented in detail and evaluated in our RV'17 paper [27] during the postdoc of Kostyantyn Vorobyov.

Summary of Main Results. The main results of our work on this topic include:

— an executable specification language E-ACSL suitable for both static and dynamic analysis,

— the E-ACSL2C tool producing an instrumented version of the given program for runtime verificaton of annotations,

— several generations of optimized (runtime) libraries for memory monitoring with E-ACSL2C allowing to accelerate the operations required to monitor memory related properties,

— several optimizations based on static analysis that allow to make the produced code faster at runtime,

— an evaluation of the proposed solutions.

During International Competitions on Runtime Verification, in the C program track, the E-ACSL2C tool finished at the second place out of four registered tools in 2014, and at the first place out of six registered tools in 2015.

The proposal of the executable specification language E-ACSL made it possible to develop new combinations of static and dynamic analyis techniques, one example of which will be described in the next section.

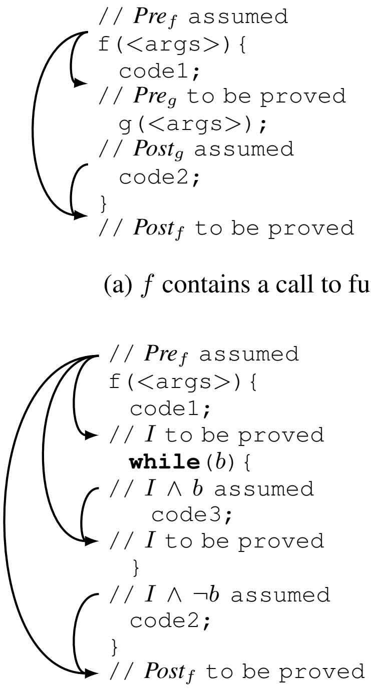
3.2 Deductive Verification Assisted by Test Generation: the STADY Method

3.2.1 Motivation

In modular deductive verification of a function f calling another function g, the roles of the pre- and postconditions of f and of the callee g are dual. The precondition of f is assumed and its postcondition must be proved, while at any call to g in f, the precondition of g must be proved before the call and its postcondition is assumed after the call. The situation for a function f with one call to g is presented in Figure 3.1a. An arrow in this figure informally indicates that its initial point provides a hypothesis for a proof of its final point. For instance, the precondition Pre_f of f and the postcondition $Post_g$ of g provide hypotheses for a proof of the postcondition $Post_f$ of f. The called function g is proved separately.

The verification of the loop invariant I of a loop in f is illustrated by Figure 3.1b: I must be proved to hold initially before the first loop iteration, and $I \wedge \neg b$ is assumed after exiting the loop. In addition, the preservation of the loop invariant I by each unique iteration of the loop must be established during the proof of f. (Loop termination, not illustrated in Figure 3.1b, can be proved as well thanks to a loop variant.)

To reflect the fact that some contracts become hypotheses during deductive verification of f



```
// Pre_f assumed
f(<args>){
  code1;
// Pre_g to be proved
  g(<args>);
// Post_g assumed
  code2;
}
// Post_f to be proved
```

(a) f contains a call to function g

```
// Pre_f assumed
f(<args>){
  code1;
// I to be proved
  while(b){
// I ∧ b assumed
    code3;
// I to be proved
  }
// I ∧ ¬b assumed
  code2;
}
// Post_f to be proved
```

(b) f contains a loop

Figure 3.1 – Verification of a function f with a function call or a loop

we use the term *subcontracts for* f to designate contracts of loops and called functions in f.

One of the most important difficulties in deductive verification is the manual processing of proof failures by the verification engineer since proof failures may have several causes. Indeed, a failure to prove Pre_g in Figure 3.1a may be due to a *non-compliance* of the code to the specification: either an error in the code `code1`, or a wrong formalization of the requirements in the specification Pre_f or Pre_g itself. The verification can also remain inconclusive because of a *prover incapacity* to finish a particular proof within allocated time. In many cases, it is extremely difficult for the verification engineer to decide how to proceed:

— either suspect a non-compliance and look for an error in the code or in the specification,

— or suspect a prover incapacity and write additional lemmas and assertions, or even give up automatic proof and try to achieve an interactive proof with a proof assistant (like COQ [110, 211]).

A failure to prove the postcondition $Post_f$ (cf. Figure 3.1a) is even more complex to analyze: along with a prover incapacity or a non-compliance due to errors in the pieces of code `code1` and `code2` or to an incorrect specification Pre_f or $Post_f$, the failure can also result from a *too weak* postcondition $Post_g$ of g that does not fully express the intended behavior of g. Notice that in this last case, the proof of g can still be successful. However, as many current automated tools for program proving, the WP plugin of FRAMA-C does not provide a sufficiently precise indication on the reason of the proof failure. Some advanced tools produce a counterexample extracted from the underlying solver but such a counterexample cannot precisely indicate if the verification engineer should

— look for a non-compliance, or

— strengthen subcontracts (and which one of them), or

— consider adding additional assertions or lemmas, or using interactive proof.

So the verification engineer must basically consider all possible reasons one after another, and maybe initiate a very costly interactive proof. For a loop, the situation is similar (cf. Figure 3.1b), and offers an additional challenge: to prove the invariant preservation, whose failure can be due to several reasons as well.

The motivation of this work is twofold. First, we want to provide the verification engineer with a more precise feedback indicating the reason of each proof failure. To do that, we provide a classification of the reasons of proof failures in several categories. Second, we look for a counterexample that either confirms the non-compliance and demonstrates that the unproven predicate can indeed fail on a test datum, or confirms a subcontract weakness showing on a test datum that some subcontracts are insufficient. Absence of counterexamples after an exhaustive exploration of all program paths (at least for a reduced input domain) gives an indication of a (likely) prover incapacity to finish the unsuccessful proof for the initial program.

3.2.2 Approach and Main Results

Modular and Non-modular Vision of a Program. Suppose a function under verification f contains one loop or one function call (see Figure 3.1). In deductive verification, during the proof of the postcondition of f, the code of the loop or called function is replaced by the corresponding contract. We say that the deductive verification tool has a *modular vision* of the function under verification: the code of the loop or called function is ignored by the tool, while their contracts are taken into account instead. The contracts of loops and called functions in f are referred to as subcontracts for f. The proof that the loops and called functions respect the corresponding subcontracts leads to separate verification conditions and is conducted separately.

On the other hand, for a given test datum, runtime assertion checking (RAC) checks every annotation reached by the program execution. RAC has a *non-modular vision* of the program, where the code of loops and called functions is executed without replacing them by the corresponding subcontracts.

Consider the C program of Figure 3.2a where f is the function under verification. It calls another function g. The postconditions of g and f on lines 2 and 5 state that the variable x is increased at least by 1 in g and at least by 2 in f. Lines 3 and 6 specify that x is the only variable that can change its value after each call. For simplicity, we ignore arithmetic overflows in this example.

```
1 int x;
2 /*@ ensures x ⩾ \old(x)+1; // Proved
3     assigns x; */
4 void g() { x=x+1; }
5 /*@ ensures x ⩾ \old(x)+2; // Proof failure
6     assigns x; */
7 void f() {        // If x = 0 here, then after the call to g:
8   g();            // x ⩾ 1 in modular vision of g,
9 }                 // x = 1 in non-modular vision of g
```

(a) Proof failure for line 5 caused by a non-compliance

```
1 int x;
2 /*@ ensures x ⩾ \old(x)+1; // Proved
3     assigns x; */
4 void g() { x=x+2; }
5 /*@ ensures x ⩾ \old(x)+2; // Proof failure
6     assigns x; */
7 void f() {        // If x = 0 here, then after the call to g:
8   g();            // x ⩾ 1 in modular vision of g,
9 }                 // x = 2 in non-modular vision of g
```

(b) Proof failure for line 5 caused by a subcontract weakness of g

Figure 3.2 – Toy examples of non-compliance and subconstract weakness (where, for simplicity, overflows are ignored)

Let us compare the execution for the input value $x = 0$ in modular and non-modular vision. In non-modular vision of the call to g, the value of x after this call is equal to 1. In modular vision of the call to g, the new value satisfies $x \geqslant 1$. Notice that several concrete values satisfy the postcondition of g: $x = 1, 2, 3, \ldots$.

Similarly, for the program of Figure 3.2b where the only modified statement is the assignment on line 4, the resulting value of x is equal to 2 in non-modular vision of the call to g. In modular vision, the resulting value of x satisfies $x \geqslant 1$ again. Notice that the property $x \geqslant 1$ is the only information on x the program prover has during the proof of f after the call to g. In both examples, the proof of f fails for the postcondition on line 5, while g is successfully proved.

Testing-Based Approach. The diagnosis of proof failures based on a counterexample generated by a prover can be imprecise since from the prover's point of view, in modular vision of the program, the code of callees and loops in f is replaced by the corresponding subcontracts.[8] To make this diagnosis more precise, one should take into account their code as well as their contracts, to consider both modular and non-modular vision of the program. A study [145] proposed to use function inlining and loop unrolling for this purpose. We propose an alternative approach: to use test case generation techniques based on Dynamic Symbolic Execution (cf. Section 1.3.1) in order to diagnose proof failures and produce counterexamples. Their usage requires code transformation translating the annotated C program into an executable C code suitable for testing. The application of test generation to the translated program in order to produce counterexamples is also referred to as *(testing-based) counterexample synthesis*. By nature, this method is in general incomplete since the set of program paths can be infinite or too large to explore, or too complex (for example, for the underlying solver to solve the path constraints within a reasonable time).

Summary of Main Results. The overall goal of the present part of the work is to provide a methodology for a more precise diagnosis of proof failures in all cases, to implement it and to evaluate it in practice. The proposed method is composed of two steps. The first step looks for a non-compliance. If none is found, the second step looks for a subcontract weakness. We propose a new classification of subcontract weaknesses into *single* (due to a single too weak subcontract) and *global* (possibly related to several subcontracts), and investigate their relative properties. Another goal is to make this method automatic and suitable for a non-expert verification engineer.

The main results of our work on this topic include:

- a classification of proof failures into three categories: *non-compliance* (NC), *subcontract weakness* (SW) and *prover incapacity*, and two subcategories: *global* and *single* subcontract weaknesses,
- program transformation techniques for diagnosis of non-compliances and subcontract weaknesses,
- a testing-based methodology for diagnosis of proof failures and generation of counterexamples, suggesting possible actions for each category,
- an implementation of the proposed solution in a tool called STADY[9], and its evaluation showing its capability to diagnose proof failures.

[8] Some program provers like KEY [117, 187] can replace callees either by code or by contract. For loops, however, it can only be possible to unroll a loop a finite number of times.

[9] See also `https://github.com/gpetiot/Frama-C-StaDy`.

3.2.3 Categories of Proof Failures and their Detection

In this section we informally describe various kinds of proof failures that can occur during the proof of an annotated program, and the corresponding diagnosis techniques. A more formal definition can be found in [9, 33, 41].

We distinguish three categories of proof failures:

— a *non-compliance* (NC) between program code and specification,

— a *subcontract weakness* (SW),

— a *prover incapacity.*

Non-Compliance

A *non-compliance* occurs when (concrete) program execution of some test datum V respecting the precondition of the function under verification leads to a failure of some annotation. This failure can be detected by runtime assertion checking, which has a non-modular vision of all callees and loops. Such a test datum V is called a *non-compliance counterexample* (NCCE).

For example, for the program of Figure 3.2a, the input $x = 0$ is a non-compliance counterexample: its execution in non-modular vision leads to a resulting value $x = 1$ that does not satisfy the postcondition of f (cf. line 5).

We formally define a technique for non-compliance detection in [41]. We denote it $\mathfrak{D}^{\mathrm{NC}}$. This technique first translates an annotated program P into another C program, denoted P^{NC}, and then applies test generation to produce test data violating some annotation(s) at runtime.

The $\mathfrak{D}^{\mathrm{NC}}$ step can have three outcomes:

— (nc, V, a) if an NCCE V has been found for a failing annotation a; the program path π_V activated by V on P^{NC} is recorded as well,

— no if $\mathfrak{D}^{\mathrm{NC}}$ has managed to perform a complete exploration of all program paths without finding any NCCE (cf. the discussion of relative completeness of PathCrawler in Section 1.3.1),

— ? (unknown) if only a partial exploration of program paths has been performed (due to a timeout, partial coverage criterion or any other limitation).

Subcontract Weakness

A *subcontract weakness* occurs when the contracts of some loop(s) or called function(s) are too weak to deduce an annotation, though this proof failure cannot be explained by a non-compliance. In other words, there is a counterexample in modular vision of the corresponding function calls or loops that is not a counterexample in non-modular vision. This needs to be explained in more detail. In modular vision, several concrete executions (possibly with different concrete values of outputs) can be considered as valid executions of the same test datum. Indeed, in general a test datum cannot be executed concretely in a deterministic way in modular vision since some subcontracts can be satisfied for several output values of variables they are allowed to modify. But it can be executed symbolically, and one value can be chosen for each variable potentially modified by callees or loops considered in modular vision. We call such values *subcontract outputs.* For a called function, the returned value is a subcontract output as well. Their choice makes it possible to consider concrete execution of other parts of code that are not replaced by subcontracts.

For example, for the input $x = 0$, any value satisfying $x \geqslant 1$ after the call to g can be part of a valid execution in modular vision of f for Figure 3.2b. Variable x is the only variable that function g is allowed to modify according to the assigns clause, thus it is the only output of the subcontract of g. We denote by nondet_x^i the subcontract output for x after the i-th subcontract ($i \geqslant 1$) traversed by program execution (in our example, after the call to g). If there is only one traversed subcontract, we omit the upper index and simply write nondet_x.

To illustrate the proof failure of the postcondition of f, the subcontract output nondet_x of x after the call to g should be 1. Taking a greater value, say $\mathrm{nondet}_x = 2$, does not provoke a failure of the postcondition of f. Thus the values

x=0 (input value),
$\mathrm{nondet}_x = 1$ (after the call to g)

illustrate the subcontract weakness of g for the postcondition of f. Notice that this is not a non-compliance counterexample: in non-modular vision, the test input $x = 0$ leads to an output $x = 2$ that respects the postcondition of f.

A *(global) subcontract weakness counterexample* (SWCE) includes a test datum V respecting the precondition of the function under verification, and subcontract outputs [10] each time a subcontract is traversed by the execution, such that

(i) the chosen execution of V in modular vision leads to an annotation failure, and
(ii) the execution of V in non-modular vision does not fail.

Remark 1 *Notice that we do not consider the same counterexample as an* NCCE *and an* SWCE*, as it follows from condition (ii) above. Indeed, even if it is arguable that some counterexamples may illustrate both a subcontract weakness and a non-compliance, we consider that non-compliances usually come from a direct conflict between the code and the specification and should be addressed first, while subcontract weaknesses are often more subtle and will be easier to address when non-compliances are eliminated. For instance, the input value* $x = 0$ *with the subcontract output* $\mathrm{nondet}_x^1 = 1$ *after the call to g is not considered as a subcontract weakness counterexample for function f for Figure 3.2a since* $x = 0$ *is a non-compliance counterexample.*

[10] For simplicity of definition of an SWCE, we sometimes give only the test datum V and omit subcontract outputs. Of course, they are always reported by the STADY tool and are very useful for a detailed analysis of the proof failure.

Remark 2 *To describe executions in non-modular vision and detect subcontract weakness counterexamples, it is necessary to know subcontract outputs (i.e. which variables can be modified). For subcontract weakness detection, we assume that every subcontract for f contains a (`loop`) `assigns` clause. Such a clause defines the list of variables (surviving at the end of the subcontract) that can change their values after the corresponding function call or loop. Requiring such clauses is not a strong limitation since such clauses are anyway necessary to prove any nontrivial code.*

We define a technique for subcontract weakness detection in [9, 33]. We denote it by $\mathfrak{D}^{\mathrm{SW}}$. This technique first translates an annotated program P into another C program, denoted P^{SW}, that simulates a modular vision of the program. In this version, all loops and function calls are replaced by the most general code respecting the corresponding subcontracts. Then, test generation is applied to produce test data violating some annotations at runtime. We further discuss subcategories of subcontract weaknesses below in Section 3.2.4

In the complete method described below in Section 3.2.5, $\mathfrak{D}^{\mathrm{SW}}$ will be applied after $\mathfrak{D}^{\mathrm{NC}}$. Notice that if $\mathfrak{D}^{\mathrm{NC}}$ has timed out without finding an NCCE, an NCCE can still exist. Thus, $\mathfrak{D}^{\mathrm{SW}}$ can still find a NCCE. It can be easily detected by executing each counterexample candidate V found by $\mathfrak{D}^{\mathrm{SW}}$ on the instrumented program P^{NC} to check if V appears to be, in fact, an NCCE.

The $\mathfrak{D}^{\mathrm{SW}}$ step may have four outcomes:

— (nc, V, a) if $\mathfrak{D}^{\mathrm{SW}}$ has found a test datum V that is an NCCE for the failing annotation a,

— (sw, V, a) if $\mathfrak{D}^{\mathrm{SW}}$ has found a test datum V that was classified as an SWCE, where a is the failing annotation.

— no if $\mathfrak{D}^{\mathrm{SW}}$ has managed to perform a complete exploration of all program paths without finding an SWCE,

— ? (unknown) if only a partial exploration of program paths has been performed (due to a timeout).

The program path π_V activated by V and leading to the failure (on P^{NC} or P^{SW}) as well as subcontract outputs for an SWCE, are recorded as well.

Prover Incapacity

Finally, in some cases, the prover can be unable to deduce an annotation while it does follow mathematically from the assumptions. In this case, there exist neither non-compliance counterexamples nor subcontract weakness counterexamples. We call this case a *prover incapacity*. It can happen for properties with non-linear arithmetics, properties requiring reasoning by induction or additional lemmas, etc. Such cases were very frequent a few years ago and become less common for many simple programs today thanks to a very significant progress made by the modern SMT solvers. Unfortunately, they cannot be fully eliminated because of prover incompleteness. An example of a prover incapacity can be found in [9, 33].

3.2.4 Global vs. Single Subcontract Weaknesses

It is interesting to investigate in more detail subcontract weaknesses. In modular vision, as we said before, all function calls and loops are replaced by the corresponding subcontracts. A (global) subcontract weakness counterexample found by the $\mathfrak{D}^{\mathrm{SW}}$ technique for such a modular vision illustrates the weakness of the whole set of all subcontracts involved in the proof of a failing annotation.

However, this diagnosis can be made more precise in some cases. Indeed, if several subcontracts are used in the proof of an annotation a, each of these subcontracts is not necessarily too weak. Sometimes, only one subcontract can be too weak. It is then desirable to indicate to the verification engineer which one.

This is the purpose of defining two kinds of subcontract weaknesses. As defined in Section 3.2.3, we have a *global subcontract weakness counterexample* (global SWCE) if the subcontracts used for the proof of an annotation a are shown to be too weak together to prove a. We have a *single subcontract weakness counterexample* (single SWCE) if it demonstrates that a unique subcontract is too weak to prove an annotation a. Basically, a partially modular vision of the program, where only one subcontract replaces the corresponding code (of a called function or loop), will allow us to detect a single subcontract weakness of this subcontract. A more detailed definition of these subcategories and the technique for detection of single subcontract weaknesses can be found in [9, 33]. Here, we illustrate and compare these notions on two examples.

Remark 3 *A proof failure can be due to the weakness of several subcontracts, while no single one of them is too weak. In other words, the absence of single* SWCE*s does not imply the absence of global* SWCE*s. On the other hand, when a single* SWCE *exists, it can indicate a single too weak subcontract more precisely than a global* SWCE.

Indeed, consider the example in Figure 3.3a, where the proof of the postcondition of f fails. If we consider a modular vision where the three subcontracts are used instead of the code of the called functions, we only get `x>=\old(x)+3` after executing the three subcontracts (since the value of x increases at least by 1 according to each of the three subcontracts). We can exhibit a global subcontract weakness counterexample:

`x=0` (input value),
$\mathrm{nondet}_x^1 = 1$ (after the call to g_1),

```
int x;
/*@ ensures x ⩾ \old(x)+1; assigns x;*/
void g1() { x=x+2; }
/*@ ensures x ⩾ \old(x)+1; assigns x;*/
void g2() { x=x+2; }
/*@ ensures x ⩾ \old(x)+1; assigns x;*/
void g3() { x=x+2; }
/*@ ensures x ⩾ \old(x)+4; assigns x;*/
void f() { g1(); g2(); g3(); }
```

(a) Absence of single SWCEs for any subcontract does not imply absence of global SWCEs

```
int x;
/*@ ensures x ⩾ \old(x)+1; assigns x;*/
void g1() { x=x+1; }
/*@ ensures x ⩾ \old(x)+1; assigns x;*/
void g2() { x=x+1; }
/*@ ensures x ⩾ \old(x)+1; assigns x;*/
void g3() { x=x+2; }
/*@ ensures x ⩾ \old(x)+4; assigns x;*/
void f() { g1(); g2(); g3(); }
```

(b) Global SWCEs do not help to find precisely a too weak subcontract

Figure 3.3 – Two examples where the proof of f fails, illustrating global and single subcontract weaknesses (where, for simplicity, overflows are ignored)

$\mathrm{nondet}_x^2 = 2$ (after the call to g_2),
$\mathrm{nondet}_x^3 = 3$ (after the call to g_3).

The output value $x = 3$ at the end of f does not respect the postcondition `x ⩾ \old(x)+4`.

If we consider a modular vision for only one of the function calls (and replace its code by its contract) and a non-modular vision for the other two calls, we always have `x ⩾ \old(x)+5` at the end of f (since we add at least 1 to x by executing the translated subcontract, and add 2 twice by executing the other two functions' code). So the postcondition of `f` holds and no single subcontract weakness counterexample can be detected. Therefore, looking for single subcontract weaknesses is not sufficient to diagnose proof failures in general.

Consider now the program in Figure 3.3b. If the three callees are replaced by their subcontracts (for global subcontract weakness detection), we can obtain the same counterexample as above and cannot find out which subcontract is too weak. Counterexamples generated by a prover suffer from the same precision issue: taking into account all subcontracts instead of the corresponding code prevents from a precise identification of a single too weak subcontract.

In this example the technique we propose to detect single subcontract weaknesses can be more precise. The replacement of `g1` by its subcontract does not lead to generating a single SWCE: the value of x is increased at least by 1 in the subcontract of `g1`, then by 1 and by 2 by the code of `g2` and `g3`, so that the value increases overall at least by 4 as stated in the postcondition. The subcontract of `g1` is not too weak alone. Similarly, the replacement of `g2` by its subcontract does not lead to a single SWCE either. However, the replacement of `g3` by its subcontract leads to a single SWCE: we have `x ⩾\old(x)+3` by executing `g1`, `g2` and the subcontract of `g3`, exhibiting the subcontract weakness of `g3`. Thus, a global SWCE does not indicate which of the subcontracts is too weak, while a single SWCE can be more precise.

3.2.5 Diagnosis of Proof Failures using Structural Testing

Let us now present an overview of the method we propose for diagnosis of proof failures, and suggestions of actions for each category of proof failures. We start by presenting an additional ACSL clause **`typically`** that can be used in the method.

The typically Clause

Test generation during counterexample synthesis can time out due to a big number of program paths. It can also be stuck in some part of the execution tree, and never reach some other part within a given time, that can make counterexample synthesis less efficient if the error can be found only in that other part. However, in many cases, a similar program with a smaller, but still representative input domain can be interesting to study, since

- an error in the program with the initial domain can already occur for the program with a reduced domain, and
- a counterexample detected for the program with a reduced domain remains a counterexample for the program with the initial domain.

In order to explore such a program with a reduced input domain automatically, we introduce the possibility to reduce the input domain for test generation by using a new ACSL clause **`typically`** supported by STADY. This clause is ignored by the proof but is taken into account during counterexample synthesis as an additional precondition. For instance, if the program has an array as an input, instead of considering an array of any possible length `0 <= n < INT_MAX` the verification engineer can add the clause **`typically`** `0 <= n < 5` and consider only array lengths `0 <= n < 5.`

A **`typically`** clause can allow test generation to visit more easily various parts of the execution tree and find a counterexample. It can also help to perform a complete exploration of all program paths both for $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$, showing that there are no counterexamples for the program with the reduced input domain and the proof failure comes from a prover incapacity.

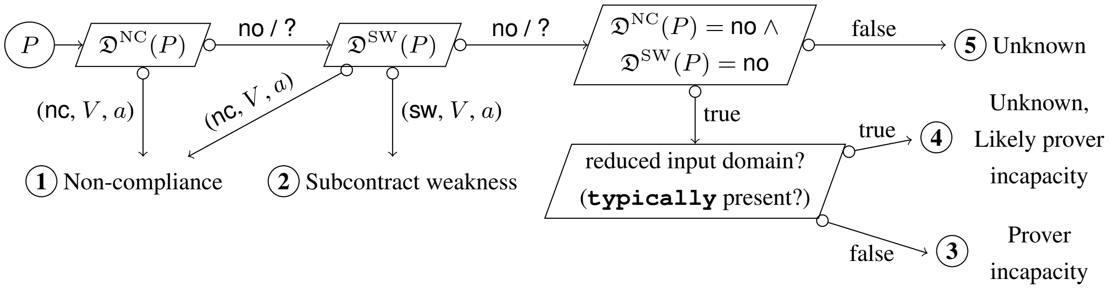


Figure 3.4 – Combined verification methodology in case of a proof failure on P

In the latter case, that gives the verification engineer an indication that the proof failure has the same reason on the program with the initial input domain.

The Diagnosis Method

The proposed method is illustrated by Figure 3.4. Suppose that the proof of the annotated program P fails for some annotation a. The first step tries to find a non-compliance using $\mathfrak{D}^{\mathrm{NC}}$. If such a non-compliance is found, it generates a non-compliance counterexample (case ① in Figure 3.4) and classifies the proof failure as a non-compliance.

If the first step cannot generate a counterexample, the $\mathfrak{D}^{\mathrm{SW}}$ step tries to generate a subcontract weakness counterexample. For the most precise dignosis, we apply both global and single subcontract weakness detection. A generated counterexample can be classified either as a non-compliance ① (that is possible if path testing in $\mathfrak{D}^{\mathrm{NC}}$ was not exhaustive, cf. Remark 1) or a subcontract weakness (case ②). In the latter case, a set S of too weak subcontracts (all subcontracts or one too weak subcontract) is reported as well.

If no counterexample has been found, the third step checks the outcomes. If both $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$ have returned no, that is, both $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$ have performed a complete path exploration without finding a counterexample, we have two situations. If both $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$ steps were performed on the complete input domain (without using a **typically** clause, which reduces the precondition for testing as detailed below), the proof failure is classified as a prover incapacity (case ③). If the user reduced the input domain for testing to a smaller domain (using a **typically** clause), the result is still unknown, but it is likely that there is a prover incapacity for the initial program (case ④). This choice of diagnosis is based on the fact that for many programs a counterexample can be found already on a reduced input domain if the reduced domain is representative of the complete input domain.

Otherwise, if at least one of $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$ was inconclusive and returned ?, the proof failure remains unclassified (case ⑤).

Suggestions of Actions

Based on the possible outcomes of the method illustrated in Figure 3.4, we are able to suggest the most suitable actions for the verification task. A reported *non-compliance* (nc, V, a) means that there is an inconsistency between the precondition, the annotation a and the code of the program path π_V leading to a. Thanks to the counterexample, the user will understand the issue by *tracing the values of variables along* π_V, or exploring them in a debugger [160]. In FRAMA-C, the execution on V can be conveniently explored using VALUE or PATHCRAWLER [11]. If an NCCE is generated, there is *no need to try an automatic or interactive proof, or look for a subcontract weakness* — it will not help.

A reported *subcontract weakness* (sw, V, a) for a set of subcontracts S means that at least one of them has to be strengthened. By definition, a non-compliance is excluded here. Thus the suggested action is to *strengthen the subcontract(s) of* S. In the case of a single subcontract weakness, S is a singleton so the suggestion is very precise and helpful to the user. Again, *trying interactive proof or writing additional assertions or lemmas will be useless* here since the property can obviously not be proved.

For a *prover incapacity,* the verification engineer may *add lemmas, assertions or hypotheses* that can help the theorem prover to succeed, or *try another theorem prover*, or *use a proof assistant* like COQ, even if it can be more complex and time-consuming.

For an *unknown, likely prover incapacity* result, $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$ steps were used on a reduced domain and detected no counterexamples on it. If the reduced domain is representative of the initial program domain, the verification engineer can prioritize the prover incapacity reason of the failure and conduct the actions described above. She should however use this result with care: the method only suggests to seriously consider the prover incapacity reason, but cannot guarantee it. Counterexamples can still exist for a bigger domain if the reduced domain is too small or not representative and the program can have a different behavior on a bigger domain.

Finally, when the verdict is *unknown,* i.e. test generation for $\mathfrak{D}^{\mathrm{NC}}$ and/or $\mathfrak{D}^{\mathrm{SW}}$ times out, the verification engineer may *strengthen the precondition* for test generation to reduce the input domain, or *extend the timeout* to give STADY more time to conclude.

3.2.6 Implementation and Evaluation

Implementation. The proposed method for diagnosis of proof failures has been implemented as a FRAMA-C plugin, named STADY [11]. It relies on two other plugins: WP [11] for deductive verification and PATHCRAWLER [207] for structural test generation (cf. Section 1.3). STADY currently supports a significant subset of the E-ACSL specification language, including **`requires`**, **`ensures`**, **`behavior`**, **`assumes`**, **`loop invariant`**, **`loop variant`** and

[11] Available at `https://github.com/gpetiot/Frama-C-StaDy`

`assert` clauses. The `assigns` and `loop assigns` clauses are considered only during the $\mathfrak{D}^{\text{SW}}$ phase: we do not try to diagnose a non-compliant (`loop`) `assigns` clause by $\mathfrak{D}^{\text{NC}}$ because provers give a clear feedback about such a non-compliance, but we do try to identify a too weak (i.e. too permissive) (`loop`) `assigns` clause since provers would report a failure elsewhere in this case. A more detailed description of the implementation can be found in [9, 33].

Experimental protocol. The evaluation used 26 correct annotated programs whose size varies from 35 to 100 lines of annotated C code. Among them, 20 originate from an independent benchmark [102], developed and maintained by the Formal Methods group of Fraunhofer FOKUS (Berlin) independently from the developers of FRAMA-C. These programs manipulate arrays, they are fully specified in ACSL and their specification expresses non-trivial properties of C arrays. To evaluate the method presented in Section 3.2.5 and its implementation, we apply STADY on systematically generated altered versions (or *mutants*) obtained from the correct program examples. Each mutant is obtained by performing a single modification (or *mutation*) on the initial program. The mutations include: a binary operator modification in the code or in the specification, a condition negation in the code, a relation modification in the specification, a predicate negation in the specification, a partial loop invariant or postcondition deletion in the specification. Such mutations model frequent errors in the code and specification (e.g. confusions between $+$ and $-$, $\leqslant$ and $<$, $\leqslant$ and $\geqslant$, a missing loop invariant, a missing pre- or postcondition, etc.) that can lead to proof failures.

The first step tries to prove each mutant using WP. In our experiments, each prover tries to prove each verification condition during at most 40 seconds. The proved mutants respect the specification and are classified as correct. Second, we apply the $\mathfrak{D}^{\text{NC}}$ method on the remaining mutants. It classifies proof failures for some mutants as non-compliances and indicates a failing annotation. The third step applies the $\mathfrak{D}^{\text{SW}}$ method on remaining mutants, classifies some of them as subcontract weaknesses and indicates a weak subcontract. If no counterexample has been found by the $\mathfrak{D}^{\text{SW}}$, the mutant remains unclassified.

Experimental results. For the 26 considered programs, 2036 mutants have been generated. 462 of them have been proved by WP. Among the 1574 unproven mutants, $\mathfrak{D}^{\text{NC}}$ has detected a non-compliance induced by the mutation in 1347 mutants (85.6%), leaving 227 unclassified. Among them, $\mathfrak{D}^{\text{SW}}$ has been able to exhibit a counterexample (either an NCCE or an SWCE) for 157 of them (69.1%), finally leaving 70 programs unclassified.

To evaluate the capacity of STADY to generate NCCEs or SWCEs even with a partial testing coverage, we set a timeout for any test generation session to 5 seconds (including one session for the $\mathfrak{D}^{\text{NC}}$ step, and possibly several sessions for $\mathfrak{D}^{\text{SW}}$ steps), and limit the number of explored program paths using the *k-path* criterion (cf. Section 1.3.1) with $k = 4$. Both the session timeout and *k-path* heavily limit the testing coverage but STADY still detects 95.5% of faults in the generated programs. That demonstrates that the proposed method can efficiently classify proof failures and generate counterexamples even with a partial testing coverage and can therefore be used for programs where the total number of paths cannot be easily limited.

Regarding execution time, on the considered programs WP needs on average 2.2 sec. per mutant (at most 3.4 sec.) to prove a program, and spends 12.5 sec. on average (at most 39.4 sec.) when the proof fails. The total execution time of STADY is comparable: it needs on average 2.2 sec. per unproven mutant (at most 43.5 sec.). Most of this time is used by test generation performed by PATHCRAWLER. The time required for the specification-to-code translation performed by STADY is negligible. The results of the experiments are presented in more detail in [9, 33].

Summary of Experiments. The experiments show that the proposed method can automatically classify a significant number of proof failures within an analysis time comparable to the time of an automatic proof and for programs for which only a partial test coverage is possible. The $\mathfrak{D}^{\mathrm{SW}}$ technique offers an efficient complement to $\mathfrak{D}^{\mathrm{NC}}$ for a more complete and more precise diagnosis of proof failures.

3.2.7 Related Work

Assisting program verification and generation of counterexamples have been addressed in different research efforts (e.g. [111, 126, 130, 133, 141, 150, 155, 157, 169, 172, 199, 210, 219]). We detail below a few projects most closely related to the STADY method.

Understanding proof failures. When SMT solvers fail on some verification conditions and provide a counter-model to explain that failure, the counter-model can be turned into a counterexample for the program under verification. This non-trivial task was designed by Hauzar et al. [111] and implemented for SPARK, a subset of Ada targeted for formal verification. This static analysis does not require to restrict the specification language, as we do, but it is not guaranteed that the provided models are real counterexamples and when they are, it does not allow the user to distinguish non-compliances from specification weaknesses. It is complementary to our combination of static and dynamic analyses and it would be useful to adapt it to C/ACSL programs. For C programs, SMT models are already exploited, for instance by the CBMC model checker [214].

A two-step verification of Tschannen et al. [145] compares the proof failures of an EIFFEL program with those of its variant where called functions are inlined and loops are unrolled. It reports code and contract revision suggestions from this comparison. This approach allows to detect specification weaknesses. The difference to our approach is that after a proof failure, they need to be able to prove a program variant (for example, replacing by the unrolled loop a loop contract that may be too weak). In our case, we need to be able to find a counter-example for a different variant (in the case of a loop contract weakness, replacing all other subcontracts except this loop contract by the real code). In the two-step verification approach, inlining and unrolling are respectively limited to a given number of nested calls and explicit iterations. If that number is too small the semantics is lost and a warning of unsoundness is reported. A bigger number of inlinings can overpass the capacity of the prover, while DSE, focusing on one path at a time, can be expected to be more efficient, but can suffer from a combinatorial explosion of the number of paths. Another benefit of DSE is the possibility to use concrete values (e.g. discovered in a previous execution) even when the constraints become very complex and the solver cannot generate a counterexample.

DAFNY has also been recently extended with tools for diagnosing proof failures [123]. When the proof times out, an algorithm decomposes it and tries to diagnose on which part the user has to focus to prevent the timeout. Then, if the proof fails, following the approach we proposed in our previous work [41], a DSE tool is used to try to find counterexamples demonstrating non-compliance between program and specification. But, when no counterexample is found, the user must manually try to find the reason of the proof failure (with the Boogie Verification Debugger), whereas we extend the approach by further exploiting DSE to automatically identify subcontract weaknesses. The notions of global and single SW and their comparison are also new.

Notice that our method assumes that the program (or its part over which the proposed method should be applied to diagnose a proof failure) is annotated in an executable specification language. While this assumption is not a limitation for most C programs, it can make the proposed method unsuitable for object-oriented programs if the properties to be verified use a (non-executable) quantification over all objects.

Proof tree analysis. More precision can be statically obtained by analyzing the unclosed branches of a proof tree. The work of Gladisch [171] is performed in the context of KEY and its verification calculus that applies deduction rules to a dynamic logic formula mixing a program and its specification. It proposes *falsifiability preservation checking* that helps to distinguish whether the branch failure comes from a programming error or from a contract weakness. However this technique can detect bugs only if contracts are strong enough. Moreover it is automatic only if a prover (typically, an SMT solver) can decide the non-satisfiability of the first-order formula expressing the falsifiability preservation condition.

The test generation proposed by Engel and Hähnle [190] exploits the proof trees built by the KEY prover during a proof attempt. The relevance of generated tests depends on the quality of the provided specification, and it does not allow to distinguish non-compliances from specification weaknesses.

Combination of static and dynamic analysis. As already discussed in Section 2.3.4, static and dynamic analysis can work better when used together, as in SYNERGY [198], its interprocedural and compositional extension in SMASH [167], the SANTE method and the STADY method. Static analysis maintains an over-approximation that aims at verifying the correctness of the system, while dynamic analysis maintains an under-approximation trying to detect an error. Both abstractions help each other in a way similar to the counterexample guided abstraction refinement method (CEGAR) [217]. Chen et al. [157] combine symbolic execution, testing and automatic debugging, through the identification of counterexamples violating metamorphic relations for the program under test. The debugging builds a cause-effect chain to a failure, by analysis of some path conditions. Comparatively, our method focuses on deductive verification rather than on symbolic execution, and aims at verifying behavioral pre-post specifications rather than metamorphic relations. Another framework, FOCALIZE [99, 197], provides an environment for both certified program development and testing [115, 147].

Counterexamples for non-inductive invariants. Counterexamples can be generated to show that invariants proposed for transition systems are too strong or too weak [180]. Differences with our work are the focus on invariants, the formalism of transition systems, and the use of random testing (with QUICKCHECK).

Other verification feedbacks. Our goal was to find input data to illustrate proof failures. A complementary work by Müller and Ruskiewicz [160] proposed to extend a runtime assertion checker to use it as a debugger to help the user understand complex counterexamples. For NC errors in the code, Christ et al. [140] proposed to analyze a trace formula to identify the fragments of code that can cause them. Our approach is complementary on two points. First, we detect either NC or SW errors. Second, we consider that the origin of an NC can be either in the code or in the specifications. Combining our method with such a localization of causes of NC errors, extended to specifications, would be another contribution.

Checking prover assumptions. Axioms are logic properties used as hypotheses by provers and thus usually not checked. Model-based testing applied to a computational model of an axiom can permit to detect errors in axioms and thus to maintain the soundness of the axiomatization [163]. This work is complementary to ours because it tackles the case of deductive verification trivially succeeding due to an invalid axiomatization, whereas we tackle the case of inconclusive deductive verification.

Christakis et al. [148] proposed to complete the results of static checkers with dynamic symbolic execution using PEX. The explicit assumptions used by the verifier (absence of overflows, non-aliasing, etc.) create new branches in the program's control flow graph which PEX tries to explore. This approach permits to detect errors out of the scope of the considered static checkers, but does not provide counterexamples in case of a specification weakness.

More recently, KEYTESTGEN [118] also proposed to automatically generate test cases from partial proofs developed in the deductive verifier KEY [117, 187].

Our work continues previous efforts to simplify deductive verification by generating counterexamples. We propose a detection technique of three categories of proof failure that gives a more precise diagnosis than in the previous work using testing. That is due to dedicated detection methods for non-compliances and subcontract weaknesses, as well as the definition and detection of single and global subcontract weaknesses. Such a testing-based methodology, automatically providing the verification engineer with a precise feedback on proof failures was not studied, implemented and evaluated before.

The different techniques of assisting deductive verification (in particular, by generating counterexamples using test generation or using counter-models produced by solvers) being relatively recent and intrinsically incomplete, further work is still required to better compare them and understand in which cases which technique is more practical.

3.3 Verification of Relational Properties

The possibility of applying both static and dynamic verification techniques appeared in another, at first glance independent research project, realized in the context of the PhD work of Lionel Blatter. Its original goal was to develop a support of more complex properties during deductive verification in FRAMA-C. It was motivated by several projects, case studies and industrial feedback. This section briefly presents its motivation and main results. These results are published in our papers in TACAS'17 [23] and TAP'18 [16].

3.3.1 Motivation

Modular deductive verification allows the user to prove that a function respects its formal specification. For a given function f, any individual call to f can be proved to respect the *contract* of f, that is, basically an implication: if the given *precondition* is true before the call, the given *postcondition* is true after it. However, some kinds of properties are not reduced to one function call. Indeed, it is frequently necessary to express a property that involves several functions or relates the results of several calls to the same function for different arguments. We call them *relational properties*.

Different theories and techniques have been proposed to deal with relational properties in different contexts. Algebraic specifications [231] can provide a theoretical framework to deal with such properties, and some specific specification languages to express them were proposed [223]. They include Relational Hoare Logic to show the equivalence of program transformations [209] or Cartesian Hoare Logic for k-safety properties [125]. Self-composition [153] is a theoretical approach to prove relational properties relating two execution traces. It reduces the verification of a relational property to a standard verification problem for a new function. Self-composition techniques have been applied for verification of information flow properties [121, 153] and properties of two equivalent-result object methods [184]. Relational properties can be expressed on Java pure methods [196] using the JML specification language. OPENJML [135] offers a partial support for deductive verification of relational properties.

The necessity to deal with relational properties in FRAMA-C has been faced in various verification projects. In a recent work, Bishop et al. [139] report on verification of continuous monotonic functions in an industrial case study on smart sensor software. The authors write:

> After reviewing around twenty possible code analysis tools, we decided to use FRAMA-C, which fulfilled all our requirements (apart from the specifications involving the comparison of function calls).

The relational property in question is the monotonicity of a function f:

$$\forall x,\ \forall y,\ x \leqslant y \Rightarrow f(x) \leqslant f(y).$$

To deal with it in FRAMA-C, Bishop et al. apply a variation of self-composition consisting in a separate verification of an additional, manually created wrapper function including the calls to be compared.

Relational properties can often be useful to give an expressive specification of library functions or hardware-supported functions, when the source code is not available. In this case, relational properties are only specified and used to verify client code, but are not verified themselves. For instance, in the PISCO project [12], an industrial case study on verification of software using hardware-provided cryptographic primitives (PKCS#11 standard) required tying together different cryptographic functions with properties such as

$$\mathrm{Decrypt}(\mathrm{Encrypt}(Msg, PrivKey), PubKey) = Msg.$$

Other examples include properties of data structures, for instance, matrix transformations such as

$$\begin{gathered}(A+B)^{\intercal} = A^{\intercal} + B^{\intercal},\\ \det(A) = \det(A^{\intercal}),\end{gathered}$$

the specification of Push and Pop over a stack [102], or parallel program specification e.g.

$$\mathrm{map}(\mathrm{append}(l_1, l_2)) = \mathrm{append}(\mathrm{map}(l_1), \mathrm{map}(l_2))$$

[12] http://www.systematic-paris-region.org/en/projets/pisco

in the MapReduce approach. A subclass of relational properties, *metamorphic properties*, relating multiple executions of the same function [143], are also used in a different context in order to address the oracle problem in software testing [257].

The lack of support for relational properties can be a major obstacle to a wider application of deductive verification in academic and industrial projects. Manual application of self-composition or possible workarounds (like in the work of Bishop et al. [139]) reduce the level of automation, can be error-prone and do not provide a fully automatic solution. A desirable solution should provide an automated link between three key components:

(i) specification of a relational property,

(ii) proof of a relational property,

(iii) usage of a relational property as a hypothesis in other proofs.

Furthermore, it would be beneficial to find a solution compatible with dynamic analysis techniques (such as runtime assertion checking with E-ACSL2C and counterexample generation with STADY).

The main purpose of the present work is to extend self-composition for specification and verification of relational properties in the context of the ACSL specification language [109] and various (static and dynamic) verification tools of FRAMA-C [11].

3.3.2 Approach and Main Results

We consider a large class of relational properties that include universally quantified properties invoking any finite number of calls of possibly dissimilar functions with possibly nested calls. We propose a novel technique inspired from self-composition for specification and proof of relational properties for C programs in FRAMA-C that meets the three aforementioned requirements *(i)-(iii)* for deductive verification, and appears also to be compatible with dynamic analysis. We implemented it in a FRAMA-C plugin RPP and illustrated its capacity to treat a large range of examples coming from various industrial and academic projects that were suffering from the impossibility to express relational properties.

One benefit of this approach is its capacity to rely on sound and mature verification tools like FRAMA-C/WP, thus allowing for automatic or interactive proof of the specified properties. Thanks to an elegant transformation into auxiliary C code and logic definitions accompanied by a property status propagation, the user can treat complex relational properties and observe the results in a transparent and fully automatic manner. Another key benefit is that this approach is suitable for verification of programs relying on library or hardware-provided functions whose source code is not available. Finally, thanks to a self-composition based approach compatible with dynamic analysis, the user can verify properties at runtime (with E-ACSL2C) or find counterexamples (with STADY). In our experiments with an independent benchmark proposed by Sousa and Dillig [125], RPP was able to prove all 65 correct properties (100%), whereas STADY was able to successfully generate counterexamples for 34 out of the 37 remaining (unproven) properties (92%).

The contributions of this work include:

— a new specification mechanism to formally express a relational property in ACSL;

— a fully-automated transformation into ACSL-annotated C code based on (an extension of) self-composition, that allows the user to prove such a property and to use it as a hypothesis for the proof of other properties in a completely automatic and transparent way;

— an implementation of this approach in a FRAMA-C plugin RPP with a sound integration of proof statuses of relational properties,

— experiments on several examples from earlier case studies and from existing benchmarks illustrating the benefits of the approach.

These results are presented in our papers in TACAS'17 [23] and TAP'18 [16]. Although the support of relational properties is still a very recent project and RPP is very young tool, it is already being used in an ongoing European H2020 project VESSEDIA to express and verify some security related properties.

3.4 Summary

The ambition to combine a large range of analysis techniques, including deductive verification, motivated the design of the executable specification language E-ACSL suitable for static and dynamic analysis, and a runtime assertion checking tool E-ACSL2C. This work fostered a lot of innovative research and tool development activities in Software Reliability Lab in 2013–2018, leading to several internships and postdocs, ten publications and two patents. These efforts dramatically improved the performances of the E-ACSL2C tool and resulted in several optimizations. Some of them rely on a prelimilary static analysis step that alone brought a speed-up between 63% and 72% [10].

These efforts also allowed the development of a combination of deductive verification with testing in the STADY method presented in the second part of this chapter. We proposed a new approach to improve the user feedback in case of a proof failure. Our method relies on test generation and helps to decide whether the proof has failed or timed out due to a non-compliance (NC) between the code and the specification, a subcontract weakness (SW), or a prover weakness. This approach is based on a spec-to-code program transformation that produces an input program for the test generation tool. Our experiments show that our implementation — in a FRAMA-C plugin, STADY— was able to diagnose over 95% of unproven programs. In particular, the non-compliance detection ($\mathfrak{D}^{\mathrm{NC}}$) was able to diagnose 85% of the unproven programs and the subcontract weakness detection ($\mathfrak{D}^{\mathrm{SW}}$) was able to diagnose 67.4% of the remaining proof failures.

In a more recent project — on support of relational properties in FRAMA-C, presented in the last part of this chapter — STADY was successfully applied to generate counterexamples for automatically generated annotated programs that model relational properties using self-composition [16]. Despite a relatively high complexity of the generated (self-composed) programs, STADY was able to produce counterexamples for almost all considered programs.

We are convinced that the STADY method facilitates the verification task and lowers the level of expertise required to conduct deductive verification, removing one of the major obstacles for its wider use in industry. One benefit of the proposed approach is the ability to provide the verification engineer with a precise reason and a counterexample that facilitate the processing of proof failures. Generated counterexamples illustrate the issue on concrete values and help to find out more easily why the proof fails. The method is fully automatic, relies on the existing specification and does not require any additional manual specification or instrumentation task. As a consequence, this method can be adopted by less experienced verification engineers and software developers.

Chapter 4

A Proof-Friendly View of Test Coverage Criteria

This chapter presents our work on test coverage criteria and their automated support performed since 2013. We propose a new, generic and proof-friendly view of several common coverage criteria based on the notion of *labels*, as well as efficient techniques for test generation, test optimization and coverage evaluation for labels implemented in the LTEST toolset. Combinations of static and dynamic analysis techniques appear to be strongly beneficial in this context. We finish by presenting the recent extension of the label specification mechanism to express a very large class of coverage criteria including **MCDC**[1] and dataflow criteria.

Different parts of this work were conducted in collaboration with Sébastien Bardin, Virgile Prevosto, Nicky Williams, Bruno Marre and Loïc Correnson (CEA List), David Mentré (MERCE), as well as Mike Papadakis and Yves Le Traon (Univ. of Luxembourg). Significant contribution to this project was brought by two postdocs at CEA List, Mickaël Delahaye[2] in 2014–2015 (that I co-supervised with Sébastien Bardin) and Michaël Marcozzi[3] in 2015–2017 (that I co-supervised with Sébastien Bardin and Virgile Prevosto). Contribution to this project was also made at its early stages by François Cheynier, an intern at CEA List in 2013, and Robin David, a PhD student at CEA List in 2013–2016, who were mainly supervised by Sébastien Bardin, and more recently in 2018 by another intern, Thibault Martin, that I co-supervised with Sébastien Bardin and Virgile Prevosto.

The criterion specification mechanism based on labels and the definition of a dedicated test generation technique for labels were first described in our ICST'14 paper [39], while the LTEST testing toolkit [78] for labels was presented in our TAP'14 paper [38]. Techniques for

[1] **MCDC** requires to demonstrate that each atomic condition alone can influence the whole branch decision (it will be further discussed in Section 4.6.1).

[2] currently, engineer in software verification

[3] currently, postdoc at Imperial College London

detection of infeasible, duplicate and subsumed test objectives were presented in our papers at ICST'15 [34] and ICSE'18 [19]. Finally, the recent extension of labels to a larger set of criteria, called Hyperlabel Test Objective Language (HTOL) and initial experiments were presented at ICST'17 in a research paper [25] and a tool paper [24]. An ongoing industrial adoption of the the PATHCRAWLER tool enriched with the label-based technology is described in our ISOLA'18 paper [69].

4.1 Motivation

Context. In current software engineering practice, testing [161, 176, 185, 233] remains the primary approach to find bugs in a piece of code. This work focuses on *white-box software testing*, in which the tester has access to the source code — as it is the case for example in unit testing. Testing all the possible program inputs being intractable in practice, the software testing community has notably defined *code-coverage criteria* (a.k.a. *adequacy criteria* or *test criteria*) [176, 233], to select an appropriate set of test inputs. In regulated domains such as aeronautics, these coverage criteria are strict normative requirements that the tester must satisfy before delivering the software. In other domains, coverage criteria are recognized as a good practice for testing, and a key ingredient of test-driven development.

A coverage criterion fundamentally specifies a set of *test requirements* or *objectives*, which should be fulfilled by the selected test inputs. Typical requirements include for example covering all statements (statement coverage) or all branches in the code (decision coverage). These requirements are essential to an automated white-box testing process, as they are used to guide the selection of new test cases, decide when testing should stop and assess the quality of a *test suite* (i.e., a set of test cases including test inputs).

Problem. Dozens of code-coverage criteria have been proposed in the literature [176, 233], from basic control-flow or data-flow [254] criteria to mutations [261] and **MCDC** [239], offering notably different ratios between testing thoroughness and effort. However, from a technical standpoint, these criteria are seen as very dissimilar bases for automation, so that most testing tools are restricted to a very small subset of criteria and that supporting a new criterion is time-consuming. *Hence, the wide variety and deep sophistication of coverage criteria in academic literature is barely exploited in practice. In addition, academic criteria have only a weak penetration into industry.*

The global motivation of this work is to bridge the gap between the potentialities offered by the huge body of academic work on (code-)coverage criteria on one side, and their limited use in the industry on the other side.

Indeed, a large number of automatic testing tools are available today. However, they often of-

Tool name	BBC	FC	DC	CC	DCC	GACC	MCDC	MCC	BP	Other
Gcov	✓	✓	✓							0/19
Bullseye		✓			✓					0/19
Parasoft	✓	✓	✓	✓			✓		✓	0/19
Semantic Designs		✓	✓							0/19
Testwell CTC++	✓	✓			✓		✓			0/19
Label-based LTest [38]	✓	✓	✓	✓	✓	✓		✓		4/19
HTOL-based LTest [24]	✓	✓	✓	✓	✓	✓	✓	✓	✓	18/19

Figure 4.1 – Support of white-box coverage criteria in coverage measurement tools

fer a limited scope of services (test coverage measurement or automatic test generation) and are restricted to a small number of coverage criteria. For example, regarding coverage measurement tools, in 2007 a survey [175] found ten tools for programs written in the C language: Bullseye [98], CodeTEST [4], Dynamic [97], eXVantage [5], Gcov (part of GCC) [96], Intel Code Coverage Tool [95], Parasoft [93], Rational PurifyPlus [106], Semantic Designs [91], TCAT [90]. To this date, there are even more tools, such as COVTOOL [103], LDRAcover [94], and Testwell CTC++ [89]. These tools support though only a limited number of test criteria in a hard-coded, non-generic manner. The upper part of Figure 4.1 summarizes which of the criteria defined in Amman and Offut's book [176] are implemented for some popular tools.

Another important point is the need for a clear and rigorous description of test criteria. Good examples for that are **MCDC** (Modified Condition-Decision Coverage) and **ACC** (Active Clause Coverage) whose original definitions were somewhat ambiguous and were interpreted differently by different testers for years (see [216]). Having a mechanism for a clear and precise definition of testing criteria would avoid such issues.

In practice, the task of the tester is tedious, not only because she has to generate test data to reach the criterion expectations but also because she must justify why a certain test requirement cannot be covered. Indeed it is likely that some requirements cannot be covered due to the semantics of the program. We refer to these requirements as *infeasible*. An important problem in testing is to make this justification automatic, that is, to detect infeasible test objectives automatically.

4.2 Approach and Main Results

In this work, we aim at proposing a *well-defined and unifying specification mechanism for these criteria*, enabling a clear separation of concerns between the precise declaration of test re-

[4] website does not exist any more

[5] website does not exist any more

quirements on one side, and the automation of white-box testing on the other side. This approach appeared to be fruitful and was successfully applied, for example, with SQL for databases and with temporal logics for model checking. This is also a *challenging* task as such a mechanism should be, at the same time:

(i) well-defined and precise to facilitate criterion documentation and comparison,

(ii) expressive enough to encode test requirements from most existing criteria, and

(ii) amenable to automation — coverage evaluation, test generation and infeasible objective detection.

Summary of Main Results. The main contributions of our work on this topic are the following.

- We define a new specification mechanism to describe coverage criteria based on *labels*. It appears to be both expressive and amenable to efficient automation, allowing to handle different coverage criteria in a unified way.
- We show that Dynamic Symbolic Execution can be extended for label coverage with only a slight overhead.
- To address the problem of infeasible test objectives, we propose a heuristic relying on static analysis for an efficient detection of infeasible labels. Recently, we extended this work for support of other kinds of polluting test objectives (infeasible, duplicate and subsumed).
- To express a larger class of criteria that cannot be expressed by labels, we recently proposed an extension of labels to the Hyperlabel Test Objective Language (HTOL), capable to express for instance dataflow criteria (such as all-uses or all-def-use-paths) and criteria relating several executions (such as **MCDC**).
- Finally, our work has lead to an ogoing adoption of the PATHCRAWLER tool with the label-based test generation strategy by MERCE, the research branch of a major industrial actor, Mitsubishi Electric. The label-based technology allows the industrial partner to easily express necessary (standard and custom) coverage criteria and efficiently generate test cases for the resulting test objectives.

4.3 Criterion Encoding with Labels

This section defines the notion of *labels*, a code annotation language to encode concrete test objectives, and explains how several common coverage criteria can be simulated by label coverage. In other words, given a program P and a coverage criterion $\mathbf{C}$, the idea is to encode the concrete test objectives instantiated from $\mathbf{C}$ for P using labels.

```
1 statement_1;
2 // l1: x == 5
3 // l2: x == y ∧ a < 3
4 statement_2;
5 // l3: x == 5
6 // l4: x ≠ y ∧ a ⩾ b
7 statement_3;
```

Figure 4.2 – Examples of labels

Labels. Given a program P, a *label* ℓ is a pair (loc, φ) where loc is a location in P and φ is a predicate over the internal state at loc, that is, such that:

— φ contains only variables and expressions (in the same language as P) defined at location loc in P, and

— φ contains no side-effect expressions.

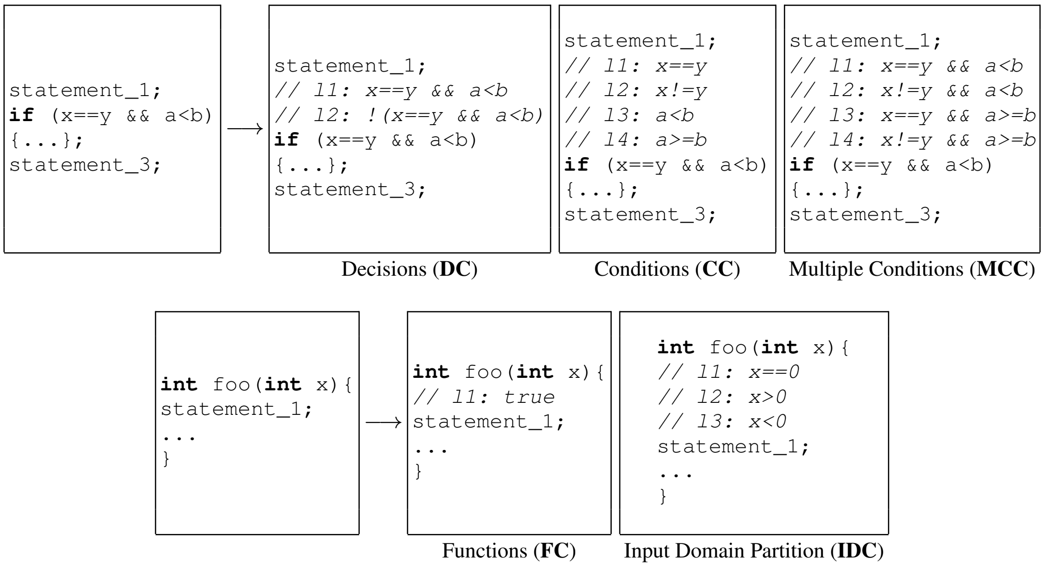
There can be several labels defined at a single location, which can possibly share the same predicate. More concretely, our notion of labels can be compared to labels in the C language, decorated with a pure C expression. Some examples of labels, named $l_1, \ldots, l_4$, are shown in Figure 4.2.

We say that a test datum t *covers a label* $\ell = (loc, \varphi)$ in P, denoted $t \rightsquigarrow_P \ell$, if the execution of P on t reaches loc on some program state s such that s satisfies φ. For example, for the program given in Figure 4.2, label l_1 is covered by test datum t if the execution of the program for this test datum reaches line 2 (or, more precisely, the program location between statements 1 and 2) with a program state in which $x = 5$. If statement 2 does not modify variable x and its execution does not change control flow, label l_3 will be covered by the same test datum. However, if statement 2 can modify x or change control flow, a simultaneous coverage of both labels is not guaranteed.

An *annotated program* is a pair $\langle P, L\rangle$ where P is a program and L is a set of labels defined in P. Figure 4.2 shows an example of an annotated program with a set of four labels $L = \{l_1, \ldots, l_4\}$.

Given an annotated program $\langle P, L\rangle$, we say that a test suite TS satisfies the *label coverage criterion* **LC** for $\langle P, L\rangle$ if TS covers every label of L, that is, for any label ℓ in L, there is a test datum t in TS, such that $t \rightsquigarrow_P \ell$. This is denoted $TS \rightsquigarrow_{\langle P,L\rangle}$ **LC**.

Criterion Encoding. The goal of the encoding is to express (or simulate) test objectives required by a given coverage criterion using labels. We say that label coverage *simulates* a given coverage criterion **C** if any program P can be *automatically* annotated with a set of labels L in such a way that any test suite TS satisfies **LC** for $\langle P, L\rangle$ if and only if TS covers all the concrete



```
statement_1;
if (x==y && a<b)
{...};
statement_3;
```

```
statement_1;
// l1: x==y && a<b
// l2: !(x==y && a<b)
if (x==y && a<b)
{...};
statement_3;
```

Decisions (**DC**)

```
statement_1;
// l1: x==y
// l2: x!=y
// l3: a<b
// l4: a>=b
if (x==y && a<b)
{...};
statement_3;
```

Conditions (**CC**)

```
statement_1;
// l1: x==y && a<b
// l2: x!=y && a<b
// l3: x==y && a>=b
// l4: x!=y && a>=b
if (x==y && a<b)
{...};
statement_3;
```

Multiple Conditions (**MCC**)

```
int foo(int x){
statement_1;
...
}
```

```
int foo(int x){
// l1: true
statement_1;
...
}
```

Functions (**FC**)

```
int foo(int x){
// l1: x==0
// l2: x>0
// l3: x<0
statement_1;
...
}
```

Input Domain Partition (**IDC**)

Figure 4.3 – Simulating standard coverage criteria with labels

test objectives instantiated from **C** for P. Such a procedure automatically adding test objectives to a given program for a given coverage criterion will be called *annotation* (or *labeling*) *function*.

Figure 4.3 illustrates the simulation of some common criteria with labels on sample code. The resulting annotated code is automatically produced by the corresponding annotation function. For example, consider decision coverage (**DC**). It is easy to see that a test suite covers **DC** for the initial program (on the left) if and only if this test suite covers **LC** for the annotated program produced for the **DC** criterion. It is ensured by the systematic insertion of labels for all branches of the code. Similar equivalences hold for other criteria encoded by labels in Figure 4.3.

We have shown [39] that label coverage can notably simulate basic-block coverage (**BBC**), decision coverage (**DC**), function coverage (**FC**), condition coverage (**CC**), decision-condition coverage (**DCC**), multiple condition coverage (**MCC**) as well as the side-effect-free fragment of weak mutations (**WM'**).

The **GACC** (General Active Clause Coverage) criterion — a weak version of **MCDC**— can be encoded in labels as well. We illustrate it for a sample code in Figure 4.4. In **GACC**, each clause in a decision should become true for some test case and false for some test case. In addition, the clause should affect the decision: changing the value of this clause should change the whole decision. For example, labels named l_1, l_2 in Figure 4.4 simulate these requirements for the first clause x==y: label l_1 ensures that it can become true, while label l_2 ensures it can become false. The second part of the predicates of these labels ensures that changing only the

```
statement_1;
if (x==y && a<b)
{...};
statement_3;
```

⟶

```
statement_1;
// l1:   x==y  && ((true && a<b ) != (false && a<b ))
// l2: !(x==y) && ((true && a<b ) != (false && a<b ))
// l3:   a<b   && ((x==y && true) != (x==y && false))
// l4: !(a<b)  && ((x==y && true) != (x==y && false))
if (x==y && a<b)
{...};
statement_3;
```

General Active Clause Coverage (**GACC**)

Figure 4.4 – Simulating **GACC** criterion with labels

first clause would indeed change the decision. The difference with stronger versions of **MCDC** (that cannot be encoded using labels) will be discussed below in Section 4.6.

Moreover, these encodings can be fully automated: the corresponding labels can be inserted automatically into the program under test. Similarly, labels can be used to encode some other, more specific criteria. In total, 11 out of 28 criteria defined in Amman and Offut's book [176] can be expressed using labels.

Despite a very simple definition, labels appear to be not only convenient to express various testing criteria, but also amenable to efficient support in various testing tasks. We showed [19, 34, 39] that labels can be very efficiently supported during test-case generation, coverage evaluation and detection of polluting (e.g. infeasible) test objectives. This support was originally implemented in 2013–2014 in the LTEST toolset [38] and was continuously improved since that time. The LTEST toolkit is open-source (except for the test generation module that depends on a close-source tool PATHCRAWLER).

It is important to emphasize that, thanks to the unified specification mechanism provided by labels, a testing service implemented for labels will directly support all criteria expressed by labels. For example, the coverage evaluation module of LTEST directly supports coverage measurement for 11 out of 28 criteria defined in Amman and Offut's book [176]. Line "Label-based LTEST " of Figure 4.1 illustrates some of these criteria.

In the next section, we detail the label-oriented strategy for test-case generation.

4.4 Test Generation for Labels

Dynamic Symbolic Execution (DSE) follows an exhaustive exploration of the path space of the program under test, that can often be inefficient for objective-oriented testing. Labels are defined as predicates attached to program instructions through a labeling function. A label is covered if a test execution reaches it and satisfies the predicate. This idea underlies former work [144, 49, 181, 236]. We generalize these results and propose efficient ways of taming the

```
statement_1;


// l: p


statement_2;
```

→

```
statement_1;

if(p){
  // report l is covered
};


statement_2;
```

Direct instrumentation

```
statement_1;
if(nondet){
  if(p){
    // report l is covered
  }
  exit(0);
};
statement_2;
```

Tight instrumentation

Figure 4.5 – Two ways to instrument a label: direct and tight instrumentation

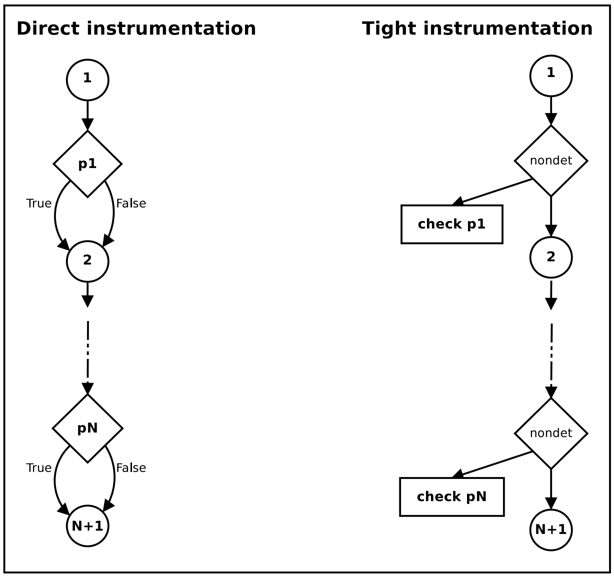


Figure 4.6 – Comparison of direct and tight instrumentation for a sequence of N labels

potential blow-up.

The label-oriented test generation strategy is based on two main principles, *tight instrumentation* and *iterative label deletion*. They can be implemented in a dedicated manner or used in a black-box manner on top of a DSE tool. We follow the second approach to present them here, and assume we have an existing DSE tool used to cover program paths.

Let us illustrate tight instrumentation in comparison with a simple approach, referred to as direct instrumentation (cf. Figure 4.5). In direct instrumentation the label is replaced by a conditional statement that checks the label predicate p and records that the label is covered whenever the predicate is satisfied.

In tight instrumentation, the conditional statement is reached only when a non-deterministic variable `nondet` is true (cf. Figure 4.5). This variable is supposed to be true or false in a non-deterministic manner. Note that any DSE engine can simulate non-deterministic choices via an additional input array of (symbolic) boolean values. Indeed, they can easily be simulated by

— an additional sufficiently large global array **int** `nondet_array[MAXLEN]` of symbolic inputs, and

— a global index variable `nd_index` initially set to zero.

Then the condition

```
if( nondet_array[nd_index++] )
```

can be used to model a non-deterministic choice instead of **if**`(nondet)` in Figure 4.5. In this way, each array value will be used at most once to decide whether symbolic execution should choose to evaluate a label or to skip it. Moreover, the execution exits after the evaluation of the label predicate, whenever it is true or not.

In the resulting instrumented program, direct instrumentation leads to creating two paths [6] for each path in the non-instrumented program, while tight instrumentation makes DSE consider only one additional program path each time a label is traversed. This situation is schematically illustrated for a sequence of N labels in Figure 4.6. We see that tight instrumentation leads to $N + 1$ paths to be considered by DSE, while direct instrumentation results in 2^N paths.

Along with a smaller number of paths to consider, tight instrumentation brings another important benefit: conditions coming from labels are added to path predicates only during the evaluation of the label predicate, while in direct instrumentation path predicates always contain conditions on all previously traversed labels. Thus, tight instrumentation yields only a linear growth of the path space without any complexification of path predicates [7].

The main idea behind iterative label deletion is to ignore a label that has been covered while continuing the test generation session. It consists in (conceptually) erasing a label constraint as soon as it is covered, so that it will not affect the subsequent path search. It is easily implemented by introducing a status for each label and considering that the label is to be checked if the `nondet` returns true and the label under consideration has not been covered yet. The label-oriented test generation strategy is further detailed in our ICST'14 paper [39]. In the LTEST toolset, I was the author of the first implementation of this strategy on top of PATHCRAWLER.

Experiments. Our experiments [39] show that the proposed test generation technique for labels offers a *dramatic improvement of performances*. For the tested examples, DSE with direct instrumentation yields a significant overhead with respect to standard DSE: several timeouts (with a timeout set to 90 min.) and an average overhead of 122x (excluding timeouts) and >144x (including timeouts). These results are comparable to previous results with a similar instrumentation-based approach used in the APEX tool built on top of Microsoft's PEX tool:

[6] and even more, e.g. if the label was inside a loop or a function called several times.

[7] If `nondet` is simulated via an additional array of symbolic inputs, the conditions on the array values are still added but there can be only one condition per value and these conditions are trivial to solve.

```
int main() {
  int a = nondet(0, 20);
  int x = nondet(0, 1000);
  return g(x,a);
}
int g(int x, int a) {
  int res;
  if( x+a ⩾ x )
    res = 1; // possible
  else
    res = 0; // impossible
  // l1: res == 0
  // l2: res == 2
  return res;
}
```

(a) A sample program with two infeasible labels l_1, l_2

```
int main() {
  int a = nondet(0, 20);
  int x = nondet(0, 1000);
  return g(x,a);
}
int g(int x, int a) {
  int res;
  if( x+a ⩾ x )
    res = 1; // possible
  else
    res = 0; // impossible
  //@ assert res ≠ 0; // unproved
  //@ assert res ≠ 2; // proved by Eva, WP
  return res;
}
```

(b) The same program where the labels are replaced with assertions

Figure 4.7 – A program with two infeasible labels and their translation into assertions

a recent paper of Jamrozik et al. reports a 272x average time-overhead, with a worst case of 2,000x [144, Table 2]. Our label-oriented DSE technique yields only a very reasonable overhead with respect to the standard DSE: no timeouts are reported, and the time-overhead is kept under 7x in all tested examples, with an average overhead of 2.4x.

4.5 Detection of Infeasible Test Objectives

Infeasible test requirements have long been recognized as one of the main cost factors of software testing [243, 249, 260]. Weyuker [243] identified that such cost should be leveraged and reduced by automated detection techniques. This issue is due to the following three reasons. First, resources are wasted in attempts to improve test cases while there is in fact no hope to cover these requirements. Second, the decision to stop testing is made impossible if the knowledge of what could be covered remains uncertain. Third, since identifying such infeasible test objectives is an undecidable problem [240], they require time consuming manual analysis because the tester must justify why she could not fulfill these requirements. In short, the effort that should be spent in testing is wasted in understanding why a given requirement cannot be covered.

By identifying the infeasible test requirements, such as equivalent mutants, testers can accurately measure the coverage of their test suites. Thus, they can decide with confidence when they should stop the testing process. Additionally, they can target full coverage. According to Frankl and Iakounenko [228] this is desirable since the majority of the faults are triggered when covering higher coverage levels, i.e., from 80% to 100% of decision coverage.

Despite the recent achievements with respect to the test generation problem [138], the infeasible requirements problem remains open. Indeed, very few approaches deal with this issue. Yet, none suggests any practical solution to it.

The label specification mechanism appears to be *proof-friendly:* it is particularly well-adapted for combinations with static analysis techniques to prove the infeasibility of test objectives. In our ICST'15 paper [34], we proposed a heuristic method to deal with the infeasible requirements for several structural testing criteria. This approach is generic since we assume that the test requirements are expressed using labels. Thus, it can be applied to all criteria that can be simulated by labels. This approach is also sound since it relies on sound static analysis techniques — if a test objective is detected as infeasible, it really is.

The main idea of the approach is that the problem of detecting infeasible requirements can be transformed into an assertion validity problem. By using static program verification techniques, it becomes then possible to address and solve this problem.

Let us illustrate the principles of the approach on a simple example. Figure 4.7a presents a program with a function `g` called from the main function. We assume that the argument `a` always belongs to the interval $[0, 20]$ while the argument `x` always belongs to the interval $[0, 1000]$. In this toy example, it is assumed to be achieved artificially by the calls to a `nondet` function on lines 2–3. In real-life programs, reduced domains of variables can follow from the program logic and computation.

The body of function `g` checks the condition `x+a`$\geqslant$`x` and assigns 1 to `res` if this condition is true, and 0 otherwise. For the given intervals of arguments, the condition on line 8 is always satisfied, so `res==1` is the only possible outcome after the conditional statement. The function contains two infeasible labels l_1, l_2 on lines 12–13.

In order to check whether a label is infeasible, our approach proposes to transform it into an assertion at the same location with the negated predicate. Now, the initial label is infeasible if and only if the assertion is always true. Static analysis techniques can then be used to prove the validity of the assertion. In our example, Figure 4.7b shows the resulting program where the labels are translated into the corresponding assertions.

The second assertion in function `g` can be proven either by value analysis or by weakest precondition calculus (using, respectively, the EVA or the WP plugins of FRAMA-C). Indeed, the outcome `res==2` is impossible, and each tool manages to show that.

The first assertion can be proven neither by value analysis nor by weakest precondition calculus. Value analysis with EVA is unable to prove that `x+a`$\geqslant$`x` is always true because it lacks relational domains and does not take into account that both occurrences of `x` refer to the same variable and have the same value. Working on a per function level, WP ignores possible values of `x` and `a` in the function `g` and cannot prove the validity of the assertion because of possible

```
int main() {
  int a = nondet(0, 20);
  int x = nondet(0, 1000);
  return g(x,a);
}
// Step 1: insert domains computed by EVA as assumptions
/*@ requires 0 ⩽ a ⩽ 20;
    requires 0 ⩽ x ⩽ 1000; */
int g(int x, int a) {

  int res;
  if( x+a ⩾ x )
    res = 1; // possible
  else
    res = 0; // impossible
  //@ assert res ≠ 0; // Step 2: now proved by WP
  //@ assert res ≠ 2; // proved by Eva, WP
  return res;
}
```

Figure 4.8 – The program of Figure 4.7b after insertion of assumptions

overflows.

We propose a detection technique consisting in two steps:

— first run value analysis on the whole program, compute the domains of variables and then insert the computed domains in the beginning of each function as assumptions,

— second, run the weakest precondition calculus to try to show the assertions with these additional assumptions.

Figure 4.8 shows the program of Figure 4.7b where the first step runs value analysis with EVA and inserts as preconditions additional assumptions, so that the deductive verification plugin WP can rely on them. At the second step, the WP plugin can indeed prove that the first assertion is valid. Since both assertions are now proven valid, it follows that the original labels l_1, l_2 in Figure 4.7a are both infeasible.

Experiments. Our experiments demonstrate that the combined static analysis approach can detect almost all the infeasible requirements of the condition coverage, multiple condition coverage and weak mutation testing criteria. In particular, the combined approach identifies on average more than 95% of the infeasible requirements, while value analysis detects on average 63% and weakest precondition detects on average 82%.

Our experiments also show that by identifying infeasible requirements before the test generation process we can speed up the automated test generation tool since it does not waste time trying to cover infeasible test objectives Thus, the automated test generation technique can be more than ~55× faster in the best case and approximately ~3.8× faster on the average case (taking into account infeasibility detection time). These technique and experiments are presented in more detail in our ICST'15 paper [34].

Our most recent work also proposed an extension of this technique for detection of two other kinds of polluting test objectives: duplicate and subsumed objectives. These results are presented in our ICSE'18 paper [19].

4.6 More Complex Criteria: From Labels to Hyperlabels

While labels can express a large range of criteria (including a large part **WM'** of weak mutations **WM**, and **GACC**, a weak variant of **MCDC**), they are still too limited in terms of expressiveness. For instance, labels cannot express strong variants of **MCDC** [239] or most path and dataflow criteria [254].

To address this limitation, we recently introduced the Hyperlabel Test Objective Language (HTOL) based in the notion of *hyperlabels*, a major extension of labels. HTOL makes it possible to emulate all white-box criteria defined in Ammann and Offutt's book [176], except strong mutations. Compared to labels, HTOL notably adds support for all the variants of the **MCDC** criterion, call coverage and all the dataflow and path-based criteria. Moreover, HTOL enables encoding test objectives to find violations of important software security properties, such as non-interference [165].

4.6.1 Hyperlabel Test Objective Language

Concretely, HTOL introduces five operators to combine labels (now referred to as *atomic labels*) into *hyperlabels*, specifying more complex test objectives:

- Bindings $\ell \triangleright \{v_1 \hookleftarrow e_1; \dots; v_k \hookleftarrow e_k\}$ store in *meta-variables* $v_1, \dots, v_k$ the value of well-defined expressions $e_1, \dots, e_k$ at the state at which atomic label ℓ is covered;
- Sequence $\ell_1 \xrightarrow{\phi} \ell_2$ requires two atomic labels ℓ_1 and ℓ_2 to be covered sequentially by a single test run, possibly constraining the whole path section between them by predicate ϕ;
- Conjunction $h_1 \cdot h_2$ requires two hyperlabels h_1, h_2 to be covered by (possibly distinct) test cases, enabling to express *hyperproperties* about sets of tests;
- Disjunction $h_1 + h_2$ requires covering at least one of hyperlabels h_1, h_2. This enables to simulate criteria involving disjunctions of objectives;
- Guard $\langle h \mid \psi \rangle$ expresses a constraint ψ over meta-variables observed (at different locations and/or during distinct executions) when covering labels underlying h.

Our ICST'17 paper provides a formal definition of these constructs and a formal semantics of hyperlabels. Let us illustrate here these operators through examples of criteria encoding.

The MCDC Criterion. This example illustrates conjunction, bindings and guards. Consider the following code snippet:

```
statement_0;
// loc_1
if ( x == y ∧ a < b )
  {...};
statement_2;
```

The (strong) **MCDC** criterion requires demonstrating that each atomic condition $c_1 \triangleq$ `x==y` and $c_2 \triangleq$`a<b` alone can influence the whole branch decision $d \triangleq c_1 \wedge c_2$. Some testers say in this case that each condition can independently affect the whole decision. For c_1, this requirement comes down to providing two tests where the truth value of c_2 at loc_1 remains the same, while values of c_1 and d change. Notice that the executions of both tests are related. The requirement for c_2 is symmetric.

This can be directly encoded with hyperlabels h_1 and h_2 as follows:

$$\begin{aligned} l &\triangleq (loc_1, d) \rhd \{v_1 \mapsfrom \texttt{x==y}; v_2 \mapsfrom \texttt{a<b}\} \\ l' &\triangleq (loc_1, \neg d) \rhd \{v_1' \mapsfrom \texttt{x==y}; v_2' \mapsfrom \texttt{a<b}\} \\ h_1 &\triangleq \langle l \cdot l' \mid v_1 \neq v_1' \wedge v_2 = v_2' \rangle \\ h_2 &\triangleq \langle l \cdot l' \mid v_1 = v_1' \wedge v_2 \neq v_2' \rangle \end{aligned}$$

Labels l and l' express the requirements to reach location loc_1 with decision d being, respectively, true and false. In addition, the bindings of l and l' record the values of conditions in metavariables v_1, v_2 and v_1', v_2', respectively. Hyperlabel h_1 requires (through the conjunction operator "$\cdot$") that the test suite covers both labels (hence, with different values for decision d) so that the recorded values satisfy the guard $v_1 \neq v_1' \wedge v_2 = v_2'$, demonstrating that c_1 alone influences the decision. Similarly, h_2 ensures the desired test objective for c_2.

The **MCDC** encoding essentially relies on the capacity of hyperlabels to express hyperproperties, that is, to relate the executions of several tests, unlike the **GACC** criterion that can be expressed in labels, as we saw in Section 4.3. This strong version of **MCDC** is also called Restricted Active Clause Coverage (**RACC**) [216]. Finally, the intermediate form of **MCDC**, called Correlated Active Clause Coverage (**CACC**) [216], also requires hyperproperties.

Call Coverage. Let us continue by showing the interest of the disjunction operator. Consider the following code snippet where `f` and `g` are two functions.

```
int f() {
  if (...)
    { /* loc_1 */ g(); }
  if (...)
    { /* loc_2 */ g(); }
}
```

The function call coverage criterion (**FCC**) requires a test case going from `f` to `g`, i.e. passing either through loc_1 or loc_2. Covering both calls is not required by this criterion, one is sufficient. This is exactly represented by hyperlabel h_3 below using the disjunction operator "$+$":

$$h_3 \triangleq (loc_1, \textit{true}) + (loc_2, \textit{true})$$

The all-defs Criterion. We illustrate now the sequence and disjunction operators. Consider the following code snippet.

```
/* loc_1 */ x := a;
if (...)
  { /* loc_2 */ res := x+1; }
else
  { /* loc_3 */ res := x-1; }
```

In order to meet the **all-defs** dataflow criterion for the definition of variable `x` at line loc_1, a test suite must cover at least one of the two def-use paths from loc_1 to loc_2 and to loc_3. This objective is represented by the hyperlabel:

$$h_4 \triangleq ((loc_1, \textit{true}) \rightarrow (loc_2, \textit{true})) + ((loc_1, \textit{true}) \rightarrow (loc_3, \textit{true})).$$

The all-uses Criterion. Consider the same code snippet. In order to meet the **all-uses** dataflow criterion for the definition of variable `x` at line loc_1, a test suite must cover both def-use paths from loc_1 to loc_2 and to loc_3. These two objectives are represented by hyperlabels:

$$h_5 \triangleq (loc_1, \textit{true}) \rightarrow (loc_2, \textit{true}) \quad \text{and} \quad h_6 \triangleq (loc_1, \textit{true}) \rightarrow (loc_3, \textit{true}).$$

Violation of Non-interference. Last, we present a more demanding example that involves bindings, sequences and guards. *Non-interference* is a strict security policy model which prescribes that information does not flow from sensitive (*high*) data towards non-sensitive (*low*) data. This is a typical example of hypersafety property [134, 165]. Hyperlabels can express the violation of such a property in a straightforward manner. Consider the code snippet below.

```
int flowcontrol(int  high,  int  low) {
  // loc_1
  ...
  // loc_2
  return res;
}
```

Non-interference is violated here if and only if two executions with the same `low` input exhibit different output (`res`) — it would mean that a difference in the `high` input is observable. This can be encoded with hyperlabel h_7:

$$
\begin{aligned}
l_1 &\triangleq (loc_1, \mathit{true}) \rhd \{v_{low} \mapsfrom \texttt{low}\} \rightarrow (loc_2, \mathit{true}) \rhd \{v_{res} \mapsfrom \texttt{res}\}\\
l_2 &\triangleq (loc_1, \mathit{true}) \rhd \{v'_{low} \mapsfrom \texttt{low}\} \rightarrow (loc_2, \mathit{true}) \rhd \{v'_{res} \mapsfrom \texttt{res}\}\\
h_7 &\triangleq \langle l_1 \cdot l_2 \mid v_{low} = v'_{low} \wedge v_{res} \neq v'_{res} \rangle
\end{aligned}
$$

Tool Support for Hyperlabels. LTest has been introduced as a toolkit for white-box testing for labels. It can be seen as a coherent all-in-one combination of various advanced techniques within a unique toolset, able to automate most aspects of testing for various coverage criteria.

In our recent work on tool support for hyperlabels, we have upgraded the core design of LTest to support hyperlabels. For the moment, the extended capabilities basically include code instrumentation (for encoding the criteria using hyperlabels) and coverage measurement (to evaluate the coverage of a given test suite for a given criterion). They can handle all existing white-box criteria but strong mutations, and do it in a generic way (cf. the last line of Figure 4.1). The implementation and experiments with the upgraded version of LTest are described in our ICST'17 tool paper [24].

4.6.2 A New Taxonomy of Coverage Criteria: The Cube

One interesting result of this work on specification of coverage criteria is a new classification of coverage criteria according to features that are necessary to express them. We propose (in our ICST'17 research paper [25]) a new taxonomy for code coverage criteria, based on the semantics of the associated reachability problem (that is, the reachability problem of the test requirements associated to the coverage criterion). We take standard reachability constraints expressible by labels as an origin, and consider three orthogonal extensions:

Origin location-based reachability, constraining a single program location for a single test execution at a time,

Ext1 reachability constraints relating several executions of the same program (hyperproperties),

Ext2 reachability constraints along a whole execution path (sequences),

Ext3 reachability constraints involving choices between several objectives (disjunctions).

The origin corresponds to criteria that can be encoded with labels. Extensions 1, 2 and 3 can be seen as three euclidean axes that spawn from the origin and add new capabilities to labels

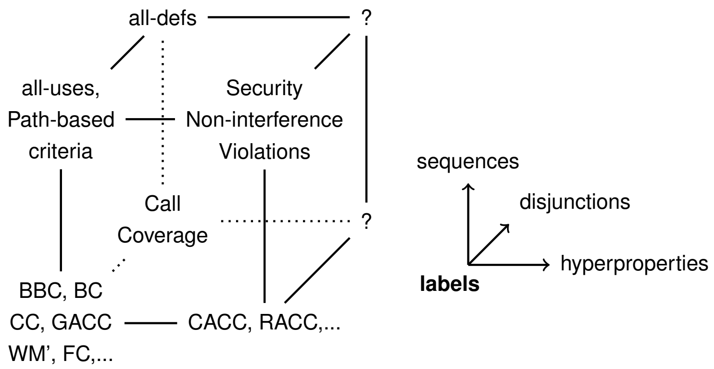


Figure 4.9 – The "cube" taxonomy of coverage criteria

along three orthogonal directions. This gives birth to a visual representation of our taxonomy as a cube, depicted in Figure 4.9, where all coverage criteria (but full mutations) can be arranged on one of the cube vertices, depending on the expressiveness of its associated reachability constraints. Intuitively, the strong mutation criterion falls outside the cube because it relates two executions on *two programs*, the program under test and the mutant. Yet, we can classify test objectives corresponding to the violation of security properties such as non-interference.

This taxonomy is interesting in several respects. First, it is *semantic*, in the sense that it refers to the reachability problems underlying the test requirements rather than to the artifact which the test requirements are drawn from. In that sense it represents progress toward abstraction compared to the older taxonomies [176, 233], the one of [176] being already more abstract than [233]. Second, it is very concise (only three basic parameters) and yet almost comprehensive, yielding new insights on criteria, through their distance to basic reachability. Interestingly, while many criteria require two extensions, we are not aware of any criterion involving the three extensions. More generally, no criterion seems to use a disjunction of constraints over several executions of the same program. These missing combinations are shown by "?" in the corresponding vertices of Figure 4.9.

4.7 Related Work

The FQL Language. The *Fshell Query Language* (FQL) by Holzer et al. [168] for test suite specification is closely related to our work. FQL enables encoding code coverage criteria into an extended form of regular expressions, whose alphabet is composed of elements from the control-flow graph of the tested program. The Fshell tool takes advantage of an off-the-shelf model-checker to automatically generate from a C program a test suite satisfying a given FQL specification. The scope of criteria that can be encoded in FQL is incomparable with the one offered by HTOL, as FQL handles complex safety-based test requirements but no hyperpropertybased requirement. Moreover, FQL is limited to syntactic elements of the program under analysis. As a consequence, FQL can encode neither **MCDC** nor **WM'**. Yet, FQL offers the ability to encode, in a standardized way, generic coverage criteria (independently of any concrete program), where HTOL encodes concrete test objectives (i.e. particular instantiations of coverage criteria for given programs). In future work, a promising research direction would be to use HTOL and FQL in a complementary way, by enabling the instantiation of the generic (extended) FQL specification of a criterion into a set of hyperlabels for the program to test.

HyperLTL and HyperCTL*. Test requirements from the strongest **MCDC** variants can be seen as examples of hyperproperties [165], i.e. properties over several traces of the system to verify. Testing hyperproperties is a rising issue, notably in the frame of security [131]. However, research in the topic still remains exploratory. Clarkson et al. [134] have introduced HyperLTL and HyperCTL*, which are extensions of temporal logics for hyperproperties, as well as an associated model-checking algorithm. This work makes no reference to test criterion encoding, but the proposed languages could be used to encode criteria like **MCDC**. However, the complexity results and first experiments [134] indicate that the approach faces strong scalability limits. HTOL being *a priori* less generic (yet, sufficient in practice), it is likely to be more amenable to *efficient* automation. In future work, we intend to explore how HTOL formally compares to HyperLTL and HyperCTL*.

Test Description Languages. Some languages have been designed to support the implementation of test harnesses at the program level (TSTL [129], UDITA [166]) or model level (TTCN-3 [221], UML Testing Profile [222]). A test harness is the helper code that executes the testing process in practice, which notably includes test definition, documentation, execution and logging. These languages offer general primitives to write and execute easily test suites, but independently of any explicit reference to a coverage criterion.

A large number of automatic testing tools are available. However, they often offer a limited scope of services (test coverage measurement or automatic test generation) and are restricted to few coverage criteria.

Coverage Measurement Tools. Code coverage is used extensively in the industry. As a result, there exists a lot of tools that embed some sort of coverage measurement. Some of the tools were mentioned in Section 4.1. Figure 4.1 summarizes supported criteria for some popular tools. It should be noticed, to be fair, that these code coverage tools also aim at causing as little overhead as possible. In contrast, as a first step, we only aim at getting a reasonable overhead.

Test Generation Tools. Many test generation tools only handle basic coverage criteria, such as bounded path coverage or decision coverage. Three interesting exceptions to be compared with our work are Fshell [183] (already described above), FAJITA [137] and APEX [144].

The FAJITA tool [137] proposes to encode a testing criterion as a set of disjoint boolean constraints, which partition the input space of the tested program. These constraints are then solved using a SAT solver to build a test suite satisfying the criterion. The paper demonstrates how basic white-box criteria (statement, branch and path coverage) can be handled in this way.

As LTEST, APEX [144] also relies on dynamic symbolic execution, but adds additional predicates to the path conditions, where LTEST annotates the code itself. However, LTEST makes use of a DSE tailored to labels, which dramatically improves the overhead observed with APEX.

4.8 Summary

The activities on advanced test coverage criteria reflect my permanent concerns of developing efficient verification techniques, based on well-defined artifacts, providing sound and applicable in practice results.

This chapter has presented our efforts to propose a unified framework for rigorous definition and efficient support of test coverage criteria. We described the label specification mechanism, an efficient label-oriented test generation method and a heuristic for sound detection of uncoverable test objectives. This approach enables in particular implementing generic tools that can be used for a wide range of criteria. This work gives an excellent illustration how static analysis techniques can be useful for testing. We presented the most recent extension of labels to hyperlabels, capable to encode all coverage criteria defined by Amman and Offut [176] except strong mutations.

This line of work inspired very active research activities in the Software Reliability Lab of CEA List in the recent five years: it involved two postdocs, two interns, one PhD student and several senior colleagues from CEA List, University of Luxembourg and MERCE.

The ongoing adoption of the label-based technology at MERCE, a research center of Mitsubishi Electric, confirms the interest of this work for the industry as we report in our ISOLA'18 paper [69]. After an experiment on a real-life industrial code of about 80,000 lines with about 1,300 functions, MERCE reported that the automatic test generation using PATHCRAWLER with labels took only about 8 hours instead of 230 work days the total manual generation would require, therefore bringing an effective benefit factor of more that 230x. Those very good results are very encouraging for further pushing the technology into industry.

Chapter 5

Conclusion and Research Perspectives

It is difficult to exhaustively describe 15 years of active research within one document. In the previous chapters, I made the choice to focus on some selected topics with a various level of detail and to leave some other activities and results out of the scope of this thesis. This final chapter will therefore first summarize some other results not described above. Then it will provide some concluding remarks, and finish with some future research perspectives.

5.1 Other Research Activities

This section provides a brief summary of research activities that have not been covered in this thesis in detail.

5.1.1 Earlier Activities in Mathematics

My research interests moved from Mathematics to Computer Science rapidly after my PhD Defense (in 2001) and a few years of temporal teaching and research positions (ATER) and postdocs (in 2001–2004). For that reason, my previous research work in Mathematics until 2003 is not at all described in this thesis. This earlier work lead to several journal publications [81–86] and a tool development [87].

5.1.2 Verification of Microkernels for the Cloud

The expansion of Cloud computing has clearly shown the need to ensure reliability, safety and security of cloud environments. In a position paper at HPCS'12 [52], we described research challenges in this domain, going from verification of applications to verification of cloud hypervisors and microkernels through verified compilation and execution.

In the context of the Carnot project SecureCloud that I coordinated for Software Reliability Lab of CEA List in 2011–2013, we have conducted several case studies where formal verification has been applied to the memory paging system, one of the most critical modules of the Anaxagoros hypervisor developed at CEA List. These results were presented in our papers at TAP'14 [40] and FMICS'15 [35].

5.1.3 Verification of Concurrent Programs

As it often happens in applied research, in my work, practical applications and case studies often guided the directions for future research efforts. Our verification case studies of Cloud hypervisor modules [35] demonstrated the need to extend FRAMA-C/WP to automatic deductive verification of concurrent code. This topic has become the main purpose of the PhD work of Allan Blanchard that I co-supervised with Frédéric Loulergue (Univ. of Orléans and later Nothern Arizona Univ.) and in collaboration with Matthieu Lemerre (CEA List). Allan Blanchard [1] defended his thesis [122] on December 6, 2016.

We proposed an automatic code transformation technique translating a given annotated concurrent program into an equivalent sequential program with an accordingly adapted specification. This work is described in our SCAM'16 paper [29], while a partial proof of the approach was presented in our VPT'17 paper [22]. The sequential program simulates the interleavings of different threads and is well-adapted for verification by an existing deductive verification tool like FRAMA-C/WP. The proposed method has been implemented in the FRAMA-C plugin CONQ2SEQ [101].

To ensure that the behavior of the concurrent program under a given memory model can be analyzed in terms of interleavings of instructions of different threads, we proposed a constraint solver developed in Prolog and Constraint Handling Rules (CHR) that determines all admissible executions of a program under a given memory model. These results are described in our CSTVA'16 paper [30] and its journal version in *Computer Languages, Systems & Structures* [7].

5.1.4 Verification of IoT Software

While formal verification is traditionally applied to embedded software in many critical domains (avionics, energy, rail, etc.), its usage for the Internet of Things (IoT) has not yet become common practice, probably because the first IoT applications were not considered critical. However, with the emergence of IoT, security concerns become of utmost importance as IoT applications deal with sensitive data and act on the physical world.

In the European Artemis DEWI project that I coordinated for Software Reliability Lab of CEA List in 2014–2017, we formally verified one of the most critical modules — the memory

[1] currently, postdoc in software verification at Inria Lille

allocation module — of Contiki, a popular OS for IoT. This verification was performed using the FRAMA-C/WP tool and presented in our CRISIS'16 paper [32].

This successful case study partly encouraged the submission of the H2020 VESSEDIA project on verification of IoT software. Later, in the context of this project, we performed other successful verification studies, on the AES-CCM* cryptography module and the linked list module of Contiki. These results are presented in our papers at RED-IoT'18 [20], NFM'18 [15] and TAP'18 [18]. In particular, the last two papers use ghost code to verify linked lists and propose some early solutions to make the underlying specification executable. These solutions will create a stronger link between deductive verification and runtime verification and facilitate their collaborative applications on real-life code.

5.2 Concluding Remarks

This document presented some research projects and results to which I contributed, in collaboration with several colleagues and students, during the last 15 years. The great variety of collaborative projects on which our lab at CEA List had the opportunity to work, and the wide range of possible applications of the FRAMA-C platform, guided to a great extent the choice of topics and problems I studied during these years. I was particularly interested in studying combined static and dynamic analyses, for which Software Reliability Lab of CEA List offers an ideal environment and FRAMA-C provides a remarkably rich tool development platform.

In these activities, I always tried to design efficient and sound analysis techniques, based on expressive and well-defined specification mechanisms, and tried to achieve, or at least to target, their practical applicability. These concerns correspond to the challenges stated in very general terms in Section 1.2 in the introduction. Let me briefly recall these challenges and underline their link to my research work.

Efficiency. The efficiency concern implies a careful design of the analysis technique and its implementation, creation of a suitable experimental protocol and rigorous evaluation of the obtained results. The benefits can include detecting more defects, or leaving less unknowns, or making the analysis faster, or reducing the number of situations that should be analyzed manually, or providing additional or more precise information to facilitate such manual analysis, etc. This concern was carefully taken into account for all techniques considered in this thesis.

Soundness. The soundness concern was also seriously considered, and sometimes even inspired new research activities. Examples of completed formal machine-checked proofs of soundness were given in Sections 2.4, 2.5 for program slicing [8, 17]. Significant efforts of formalization and proof were realized in several other projects [22, 122, 132, 39, 41, 49, 156], some of which are not detailed in this document, and these efforts should be continued.

Specification Mechanisms. Modern software verification problems continue to give rise to new kinds of properties to consider, and imply the need to specify and verify them. In my work it often appeared necessary to find new specification mechanisms to express such properties and/or to support them by different analysis techniques. Examples of such new specification mechanisms described in this document include an executable specification language [45] (Section 3.1), relational properties [16] (Section 3.3), and advanced test coverage criteria [25, 39] (Chapter 4).

Practical Applicability. This is an extremely difficult goal since it implies modification of existing practices, especially in industry, and the achievement of this goal can take significant time and effort. While several projects presented in this thesis were motivated by, or realized with this concern in mind, the goal has not been achieved in all cases. The most successful example — an ongoing industrial adoption by a major industrial actor — is given for label-based test generation [69] (Section 4.8). Other successful examples of ongoing practical applications so far include the combinations of value analysis, slicing and dynamic analysis inspired by SANTE (Section 2.3.5) and the support of relational properties (Section 3.3) which are used in several collaborative projects, in particular, for detection of security issues. The online testing service PATHCRAWLER-online (Section 2.2.2) is actively used for evaluation and teaching. Finally, I am optimistic that these and other projects will find even more users in the future. This goal requires active dissemination of research results and regular contacts between researchers and practitioners.

I actively participate in dissemination, training and teaching activities in which I try to use recent research results. In addition to regular teaching activities[2] using FRAMA-C analyzers at several French universities, I co-organized more than 15 tutorials on structural testing (using the PATHCRAWLER-online testing service), runtime assertion checking, deductive verification, verification of IoT software, combinations of static and dynamic analyzes, and advanced test coverage criteria. They were presented at various international conferences and summer schools such as ASE, ISSRE, QSIC, SAC, iFM, HPCS, TAP, SecDev, RV, ZINC, TAROT. These tutorials were often accompanied by invited tutorial papers, for instance, at TAP'12 [76, 51], QSIC'12 [77], TAP'13 [74], RV'13 [75], TAP'14 [73], RV'16 [72], HPCS'18 [70]. The presentations and exercises for many of them are also available on my webpage [88], and I am

[2] Even if my current position does not include any teaching obligations and the number of hours of teaching I am allowed to give is limited.

always happy to share them with colleagues who use them for teaching and with industrial partners who wish to learn more about these techniques. Finally, to encourage dissemination and exchanges between researchers and practitioners, I had the opportunity to co-organize *Frama-C and SPARK Days* workshops in 2017 and 2018, a few other workshops, international and national conferences [2, 3], served as Tutorial Chair at ISSRE'17 and co-edited a special journal issue [1].

5.3 Research Perspectives

Several directions of future research arise from the aforementioned projects. Combinations of different static and dynamic approaches can make verification techniques both more efficient and more easily accessible for non-expert engineers. However, being based on different methods and assumptions, such combinations often need solid theoretical foundations.

On the other hand, the increasing amount of critical software and the requirements of modern certification standards (e.g. MCDC criterion required by the DO178 norm in avionics, various specific coverage criteria used in industry, specific security requirements, blockchains, etc.) bring into focus another challenge: the need for a better support of complex properties. This includes their formalization and their sound and efficient tool support.

A few research directions are presented below in more detail.

5.3.1 A Generic Program Slicer for Unstructured Programs

Providing a dedicated slicer for a new programming language can be a difficult task, especially for non-structured programs (with goto, break etc.), where control dependencies are more complex to formalize than for a structured program. This research direction proposes to provide a generic program slicing, in the presence of possible runtime errors and non-termination, formalized in COQ. Using a CFG-based representation of an input program, it will allow to compute control and data dependencies, and to formalize and prove the soundness of the program slice computation. The benefits of this approach will include providing a sound, certified and generic slicer, extracted from COQ, that can be connected to different programming languages. As a next step, integrating an optimization by the results of other analyses (e.g. value analysis) will help to further improve its precision.

5.3.2 Formal Proof of Soundness of Combined Analyses in Coq

Previous attempts to establish the soundness of (combinations of) analysis techniques have demonstrated both the need and the difficulty to formally represent them in a proof assistant like COQ. This research perspective will require first to create a program analysis framework prototype for a representative programming language in COQ, where various analyses can be formalized, first by specifying the expected results, then by a concrete verified implementation. While significant research efforts have been invested to formalize separate analyses (e.g. value analysis, slicing, test generation, constraint solver) in COQ, they are currently not connected to the same (input or intermediate) language and cannot be easily used together. The objective of this project is to unify these efforts within a unique analysis platform in COQ and to facilitate the development and verification of future analysis techniques. In particular, it will allow to formally establish soundness of combined methods used in SANTE, STADY, E-ACSL2C, etc.

As a first step in this direction, the proof of the combination of dataflow analysis and runtime verification in E-ACSL2C was started in October 2017 in the context of the PhD project of Dara Ly that I supervise in collaboration with Frédéric Loulergue (Northern Arizona Univ.) and Julien Signoles (CEA List). To continue this work, I also participate in the submission of a collaborative research project on related topics.

5.3.3 Testing Techniques for Advanced Test Coverage Criteria

Our previous work provided generic mechanisms to specify a large class of source-code based test coverage criteria using labels and hyperlabels and proposed efficient test generation and infeasibility detection techniques for labels [25, 34, 38, 39]. However, more complex test objectives expressed in hyperlabels such as dalaflow criteria or hyperproperties are not yet efficiently supported by the tools.

Future work directions include an efficient lifting of automatic test generation technologies to hyperlabels and an efficient detection of uncoverable hyperlabels. Support of the large class of coverage criteria expressible in the HTOL language (using hyperlabels) in major testing tasks remains an important challenge. This class includes important criteria, such as **MCDC** and dataflow criteria (e.g. def-use). In addition, it is worth investigating extensions to HTOL for yet unexplored industrially relevant criteria.

Overall, unifying, formalizing and evaluating test criteria and their "optimized" use, requires the identification and formalization of the particular industrial needs. For instance, the practical use of test criteria under a continuous deployment model is still unknown. Indeed, integration into a continuous development cycle raises many questions, notably how to integrate automated test input generation, when should testers measure coverage, what they should cover and to what extend coverage is sufficient. To continue the work on these topics, in 2018 I coordinated the submission of a joint ANR-FNR (France-Luxembourg) collaborative research project on these topics. This project was granted a support of 670 000 € and will start in January 2019.

Chapter 6

References

Publications of Nikolai Kosmatov

The references of the author are organized by discipline (Computer Science and Mathematics) and category, and sorted by year. All publications in Computer Science present contributions realized and published after the PhD thesis of the author defended in 2001 (in Mathematics).

Publications in Computer Science

Edited Conference Proceedings and Special Journal Issue

[1] Jasmin Christian Blanchette, Francis Bordeleau, Nikolai Kosmatov, Alfonso Pierantonio, Gabi Taentzer, and Manuel Wimmer, eds. *Revised Selected Papers of STAF 2015, Special Issue, Software & Systems Modeling (SoSyM)*. Springer. DOI: 10.1007/s10270-018-0686-1. Rank B.

[2] Akram Idani and Nikolai Kosmatov, eds. *Actes des 16èmes journées AFADL: Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 2017)*. Published online. URL: http://afadl2017.imag.fr/AFADL_Procs_2017.pdf.

[3] Jasmin Christian Blanchette and Nikolai Kosmatov, eds. *Proc. of the 9th International Conference on Tests and Proofs (TAP 2015), Held as Part of STAF 2015*. Vol. 9154. LNCS. Springer, 2015. ISBN: 978-3-319-21214-2. DOI: 10.1007/978-3-319-21215-9. Rank B.

Patent Applications

[4] Paul Dubrulle, Stéphane Louise, Christophe Gaston, Mathieu Jan, Nikolai Kosmatov, and Arnault Lapitre. *Outil et procédé de conception et de validation d'un système flots de données par un modèle formel (Tool and procedure for design and validation of a dataflow system by a formal model).* Patent application. In French. Submitted on August 31, 2018 to INPI (French Patent Office). 2018.

[5] Kostyantyn Vorobyov, Nikolai Kosmatov, and Julien Signoles. *A computer-implemented method and a system for encoding a heap application memory state using shadow memory.* Patent application. Submitted on September 27, 2016 to European Patent Office. Submitted on September 15, 2017 to US Patent Office. 2016. URL: `https://goo.gl/qkzGKp`.

[6] Kostyantyn Vorobyov, Nikolai Kosmatov, and Julien Signoles. *A computer-implemented method and a system for encoding a stack application memory state using shadow memory.* Patent application. Submitted on December 7, 2016 to European Patent Office. Submitted on November 29, 2017 to US Patent Office. 2016. URL: `https://goo.gl/hBt7MB`.

Peer-Reviewed International Journals

[7] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. "MMFilter: A CHR-Based Solver for Generation of Executions under Weak Memory Models." In: *Computer Languages, Systems & Structures* 53 (2018), pp. 121–142. ISSN: 1477-8424. DOI: `10.1016/j.cl.2018.03.002`. Rank C.

[8] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. "Cut branches before looking for bugs: certifiably sound verification on relaxed slices." In: *Formal Asp. Comput.* 30.1 (2018), pp. 107–131. ISSN: 0934-5043. DOI: `10.1007/s00165-017-0439-x`. Rank A.

[9] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. "How Testing Helps to Diagnose Proof Failures." In: *Formal Asp. Comput.* 30.6 (2018), pp. 629–657. ISSN: 0934-5043. DOI: `10.1007/s00165-018-0456-4`. Rank A.

[10] Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. "Fast as a shadow, expressive as a tree: Optimized memory monitoring for C." In: *Sci. Comput. Program.* 132 (2016). Special Issue, Revised Selected Papers of SAC-SVT 2015, pp. 226–246. ISSN: 0167-6423. DOI: `10.1016/j.scico.2016.09.003`. Rank A.

[11] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. “Frama-C: A software analysis perspective.” In: *Formal Asp. Comput.* 27.3 (2015), pp. 573–609. ISSN: 0934-5043. DOI: 10.1007/s00165-014-0326-7. Rank A.

[12] Omar Chebaro, Pascal Cuoq, Nikolai Kosmatov, Bruno Marre, Anne Pacalet, Nicky Williams, and Boris Yakobowski. “Behind the scenes in SANTE: a combination of static and dynamic analyses.” In: *Autom. Softw. Eng.* 21.1 (2014), pp. 107–143. ISSN: 0928-8910. DOI: 10.1007/s10515-013-0127-x. Rank A.

Peer-Reviewed Book Chapters

[13] Nikolai Kosmatov. “Chapter XI: Concolic Test Generation and the Cloud: Deployment and Verification Perspectives.” In: *Software Testing in the Cloud: Perspectives on an Emerging Discipline*. Ed. by Scott Tilley and Tauhida Parveen. IGI Global, 2013. ISBN: 978-1-46662-536-5. DOI: 10.4018/978-1-4666-2536-5.ch011.

[14] Nikolai Kosmatov. “Chapter XI: Constraint-Based Techniques for Software Testing.” In: *Artificial Intelligence Applications for Improved Software Engineering Development: New Prospects*. Ed. by Farid Meziane and Sunil Vadera. IGI Global, 2010, pp. 231–251. ISBN: 978-1-60566-758-4. DOI: 10.4018/978-1-60566-758-4.ch011.

Peer-Reviewed International Conferences and Workshops

[15] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. “Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C.” In: *Proc. of the 10th NASA Formal Methods Symposium (NFM 2018)*. Vol. 10811. LNCS. Springer, Apr. 2018, pp. 37–53. ISBN: 978-3-319-77934-8. DOI: 10.1007/978-3-319-77935-5.

[16] Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, Virgile Prevosto, and Guillaume Petiot. “Static and Dynamic Verification of Relational Properties on Self-Composed C Code.” In: *Proc. of the 12th International Conference on Tests and Proofs (TAP 2018), Held as Part of STAF 2018*. Vol. 10889. LNCS. Springer, June 2018, pp. 44–62. ISBN: 978-3-319-92993-4. DOI: 10.1007/978-3-319-92994-1_3. Rank B.

[17] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. “Fast Computation of Arbitrary Control Dependencies.” In: *Proc. of the 21st International Conference on Fundamental Approaches to Software Engineering (FASE 2018), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2018)*. Vol. 10802. LNCS. Springer, Apr. 2018, pp. 207–224. ISBN: 978-3-319-89362-4. DOI: 10.1007/978-3-319-89363-1_12. Rank B.

[18] Frédéric Loulergue, Allan Blanchard, and Nikolai Kosmatov. “Ghosts for Lists: from Axiomatic to Executable Specifications.” In: *Proc. of the 12th International Conference on Tests and Proofs (TAP 2018), Held as Part of STAF 2018*. Vol. 10889. LNCS. Springer, June 2018, pp. 177–184. ISBN: 978-3-319-92993-4. DOI: `10.1007/978-3-319-92994-1_11`. Rank B.

[19] Michaël Marcozzi, Sébastien Bardin, Nikolai Kosmatov, Mike Papadakis, Virgile Prevosto, and Loïc Correnson. “Time to clean your test objectives.” In: *Proc. of the 40th International Conference on Software Engineering (ICSE 2018)*. ACM, June 2018, pp. 456–467. ISBN: 978-1-4503-5638-1. DOI: `10.1145/3180155.3180191`. Rank A*.

[20] Alexandre Peyrard, Nikolai Kosmatov, Simon Duquennoy, and Shahid Raza. “Towards Formal Verification of Contiki OS: Analysis of the AES-CCM* Modules with Frama-C.” In: *Proc. of the 2nd International Workshop on Recent advances in secure management of data and resources in the IoT (RED-IoT 2018), part of the International Conference on Embedded Wireless Systems and Networks (EWSN 2018)*. ACM, Feb. 2018, pp. 264–269. URL: `http://dl.acm.org/citation.cfm?id=3234910`.

[21] Kostyantyn Vorobyov, Nikolai Kosmatov, and Julien Signoles. “Detection of Security Vulnerabilities in C Code using Runtime Verification: an Experience Report.” In: *Proc. of the 12th International Conference on Tests and Proofs (TAP 2018), Held as Part of STAF 2018*. Vol. 10889. LNCS. Springer, June 2018, pp. 139–156. ISBN: 978-3-319-92993-4. DOI: `10.1007/978-3-319-92994-1_8`. Rank B.

[22] Allan Blanchard, Frédéric Loulergue, and Nikolai Kosmatov. “From Concurrent Programs to Simulating Sequential Programs: Correctness of a Transformation.” In: *Proc. of the 5th International Workshop on Verification and Program Transformation (VPT 2017) co-located with the 2017 European Joint Conferences on Theory and Practice of Software (ETAPS 2017)*. Vol. 253. EPTCS. Apr. 2017, pp. 109–123. DOI: `10.4204/EPTCS.253.9`.

[23] Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, and Virgile Prevosto. “RPP: Automatic Proof of Relational Properties by Self-Composition.” In: *Proc. of the 23th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017)*. Vol. 10205. LNCS. Springer, Apr. 2017, pp. 391–397. ISBN: 978-3-662-54576-8. DOI: `10.1007/978-3-662-54577-5_22`. Rank A.

[24] Michaël Marcozzi, Sébastien Bardin, Mickaël Delahaye, Nikolai Kosmatov, and Virgile Prevosto. “Taming Coverage Criteria Heterogeneity with LTest.” In: *Proc. of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017)*. IEEE, Mar. 2017, pp. 500–507. ISBN: 978-1-5090-6031-3. DOI: `10.1109/ICST.2017.57`. Rank A.

[25] Michaël Marcozzi, Mickaël Delahaye, Sébastien Bardin, Nikolai Kosmatov, and Virgile Prevosto. "Generic and Effective Specification of Structural Test Objectives." In: *Proc. of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017)*. IEEE, Mar. 2017, pp. 436–441. ISBN: 978-1-5090-6031-3. DOI: 10.1109/ICST.2017.48. Rank A.

[26] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. "E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper)." In: *In Proc. of the 1st International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES 2017), part of the 17th International Conference on Runtime Verification (RV 2017)*. Vol. 3. Kalpa Publications in Computing. EasyChair, Sept. 2017. DOI: 10.29007/fpdh.

[27] Kostyantyn Vorobyov, Nikolai Kosmatov, Julien Signoles, and Arvid Jakobsson. "Runtime Detection of Temporal Memory Errors." In: *Proc. of the 17th International Conference on Runtime Verification (RV 2017)*. Vol. 10548. LNCS. Springer, Sept. 2017, pp. 294–311. ISBN: 978-3-319-67530-5. DOI: 10.1007/978-3-319-67531-2_18. Rank C.

[28] Kostyantyn Vorobyov, Julien Signoles, and Nikolai Kosmatov. "Shadow state encoding for efficient monitoring of block-level properties." In: *Proc. of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM 2017)*. ACM, June 2017, pp. 47–58. ISBN: 978-1-4503-5044-0. DOI: 10.1145/3092255.3092269. Rank A.

[29] Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre, and Frédéric Loulergue. "Conc2Seq: A Frama-C Plugin for Verification of Parallel Compositions of C Programs." In: *Proc. of the 16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2016)*. IEEE, Oct. 2016, pp. 67–72. ISBN: 978-1-5090-3848-0. DOI: 10.1109/SCAM.2016.18. Rank C.

[30] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. "A CHR-Based Solver for Weak Memory Behaviors." In: *Proc. of the 7th International Workshop on Constraint Solvers in Testing, Verification, and Analysis (CSTVA 2016) co-located with the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. Vol. 1639. CEUR Workshop Proceedings. CEUR-WS.org, July 2016, pp. 15–22.

[31] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. "Cut Branches Before Looking for Bugs: Sound Verification on Relaxed Slices." In: *Proc. of the 19th International Conference on Fundamental Approaches to Software Engineering (FASE 2016), Held as Part of the European Joint Conferences on Theory and Practice of Soft-*

ware (ETAPS 2016). Vol. 9633. LNCS. Springer, Apr. 2016, pp. 179–196. ISBN: 978-3-662-49664-0. DOI: 10.1007/978-3-662-49665-7_11. Rank B.

[32] Frédéric Mangano, Simon Duquennoy, and Nikolai Kosmatov. "Formal Verification of a Memory Allocation Module of Contiki with Frama-C: A Case Study." In: *Proc. of the 11th International Conference on Risks and Security of Internet and Systems (CRiSIS 2016).* Vol. 10158. LNCS. Springer, Sept. 2016, pp. 114–120. ISBN: 978-3-319-54875-3. DOI: 10.1007/978-3-319-54876-0_9. Rank C.

[33] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. "Your Proof Fails? Testing Helps to Find the Reason." In: *Proc. of the 10th International Conference on Tests and Proofs (TAP 2016), Held as Part of STAF 2016.* Vol. 9762. LNCS. **Best paper award**. Springer, July 2016, pp. 130–150. ISBN: 978-3-319-41134-7. DOI: 10.1007/978-3-319-41135-4_8. Rank B.

[34] Sébastien Bardin, Mickaël Delahaye, Robin David, Nikolai Kosmatov, Mike Papadakis, Yves Le Traon, and Jean-Yves Marion. "Sound and Quasi-Complete Detection of Infeasible Test Requirements." In: *Proc. of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015).* IEEE, Apr. 2015, pp. 1–10. ISBN: 978-1-4799-7125-1. DOI: 10.1109/ICST.2015.7102607. Rank A.

[35] Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre, and Frédéric Loulergue. "A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C." In: *Proc. of the 20th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2015).* Vol. 9128. LNCS. Springer, June 2015, pp. 15–30. ISBN: 978-3-319-19457-8. DOI: 10.1007/978-3-319-19458-5_2. Rank C (ranked as a conference).

[36] Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. "Fast as a shadow, expressive as a tree: hybrid memory monitoring for C." In: *Proc. of the 30th Annual ACM/SIGAPP Symposium on Applied Computing, Software Verification and Testing Track (SAC-SVT 2015).* ACM, Apr. 2015, pp. 1765–1772. ISBN: 978-1-4503-3196-8. DOI: 10.1145/2695664.2695815. Rank B.

[37] Balázs Kiss, Nikolai Kosmatov, Dillon Pariente, and Armand Puccetti. "Combining Static and Dynamic Analyses for Vulnerability Detection: Illustration on Heartbleed." In: *Proc. of the 11th International Haifa Verification Conference (HVC 2015).* Vol. 9434. LNCS. Springer, Nov. 2015, pp. 39–50. ISBN: 978-3-319-26286-4. DOI: 10.1007/978-3-319-26287-1_3.

[38] Sébastien Bardin, Omar Chebaro, Mickaël Delahaye, and Nikolai Kosmatov. "An All-in-One Toolkit for Automated White-Box Testing." In: *Proc. of the 8th International Conference on Tests and Proofs (TAP 2014), Held as Part of STAF 2014.* Vol. 8570.

LNCS. Springer, July 2014, pp. 53–60. ISBN: 978-3-319-09098-6. DOI: 10.1007/978-3-319-09099-3_4. Rank B.

[39] Sébastien Bardin, Nikolai Kosmatov, and François Cheynier. "Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria." In: *Proc. of the 7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*. IEEE, Mar. 2014, pp. 173–182. ISBN: 978-0-7695-5185-2. DOI: 10.1109/ICST.2014.30. Rank A.

[40] Nikolai Kosmatov, Matthieu Lemerre, and Céline Alec. "A Case Study on Verification of a Cloud Hypervisor by Proof and Structural Testing." In: *Proc. of the 8th International Conference on Tests and Proofs (TAP 2014), Held as Part of STAF 2014*. Vol. 8570. LNCS. Springer, 2014, pp. 158–164. ISBN: 978-3-319-09098-6. DOI: 10.1007/978-3-319-09099-3_12. Rank B.

[41] Guillaume Petiot, Bernard Botella, Jacques Julliand, Nikolai Kosmatov, and Julien Signoles. "Instrumentation of Annotated C Programs for Test Generation." In: *Proc. of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014)*. IEEE, Sept. 2014, pp. 105–114. ISBN: 978-0-7695-5304-7. DOI: 10.1109/SCAM.2014.19. Rank C.

[42] Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. "How Test Generation Helps Software Specification and Deductive Verification in Frama-C." In: *Proc. of the 8th International Conference on Tests and Proofs (TAP 2014), Held as Part of STAF 2014*. Vol. 8570. LNCS. Springer, 2014, pp. 204–211. ISBN: 978-3-319-09098-6. DOI: 10.1007/978-3-319-09099-3_16. Rank B.

[43] Mickaël Delahaye and Nikolai Kosmatov. "A Late Treatment of C Precondition in Dynamic Symbolic Execution." In: *Proc. of the 5th International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA 2013), co-located with the 6th IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*. IEEE, 2013, pp. 230–231. ISBN: 978-1-4799-1324-4. DOI: 10.1109/ICSTW.2013.34.

[44] Mickaël Delahaye and Nikolai Kosmatov. "A Late Treatment of C Precondition in Dynamic Symbolic Execution Testing Tools." In: *Proc. of the 4th International Conference on Runtime Verification (RV 2013)*. Vol. 8174. LNCS. Springer, 2013, pp. 328–333. ISBN: 978-3-642-40786-4. DOI: 10.1007/978-3-642-40787-1_20. Rank C.

[45] Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. "Common specification language for static and dynamic analysis of C programs." In: *Proc. of the 28th Annual ACM Symposium on Applied Computing, Software Verification and Testing Track (SAC-SVT*

2013). ACM, Mar. 2013, pp. 1230–1235. ISBN: 978-1-4503-1656-9. DOI: 10.1145/2480362.2480593. Rank B.

[46] Nikolai Kosmatov, Guillaume Petiot, and Julien Signoles. “An Optimized Memory Monitoring for Runtime Assertion Checking of C Programs.” In: *Proc. of the 4th International Conference on Runtime Verification (RV 2013).* Vol. 8174. LNCS. Springer, 2013, pp. 167–182. ISBN: 978-3-642-40786-4. DOI: 10.1007/978-3-642-40787-1_10. Rank C.

[47] Nikolai Kosmatov, Nicky Williams, Bernard Botella, and Muriel Roger. “Structural Unit Testing as a Service with PathCrawler-online.com.” In: *Proc. of the 7th IEEE International Symposium on Service-Oriented System Engineering (SOSE 2013).* IEEE, 2013, pp. 435–440. ISBN: 978-1-4673-5659-6. DOI: 10.1109/SOSE.2013.78.

[48] Omar Chebaro, Mickaël Delahaye, and Nikolai Kosmatov. “Testing Inexecutable Conditions on Input Pointers in C Programs with SANTE.” In: *Proc. of the 24th International Conference on Software and Systems Engineering and their Applications (ICSSEA 2012).* Oct. 2012, pp. 1–7.

[49] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. “Program slicing enhances a verification technique combining static and dynamic analysis.” In: *Proc. of the 27th Annual ACM Symposium on Applied Computing, Software Verification and Testing Track (SAC-SVT 2012).* ACM, Mar. 2012, pp. 1284–1291. ISBN: 978-1-4503-0857-1. DOI: 10.1145/2245276.2231980. Rank B.

[50] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. “Frama-C - A Software Analysis Perspective.” In: *Proc. of the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012).* Vol. 7504. LNCS. Springer, Oct. 2012, pp. 233–247. ISBN: 978-3-642-33825-0. DOI: 10.1007/978-3-642-33826-7_16. Rank B.

[51] Nikolai Kosmatov, Nicky Williams, Bernard Botella, Muriel Roger, and Omar Chebaro. “A Lesson on Structural Testing with PathCrawler-online.com.” In: *Proc. of the 6th International Conference on Tests and Proofs (TAP 2012).* Vol. 7305. LNCS. Springer, May 2012, pp. 169–175. ISBN: 978-3-642-30472-9. DOI: 10.1007/978-3-642-30473-6_15. Rank B.

[52] Frédéric Loulergue, Frédéric Gava, Nikolai Kosmatov, and Matthieu Lemerre. “Towards verified cloud computing environments.” In: *Proc. of the 2012 International Conference on High Performance Computing & Simulation (HPCS 2012).* IEEE, July 2012, pp. 91–97. ISBN: 978-1-4673-2359-8. DOI: 10.1109/HPCSim.2012.6266896. Rank B.

[53] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. “The SANTE Tool: Value Analysis, Program Slicing and Test Generation for C Program Debugging.” In: *Proc. of the 5th International Conference on Tests and Proofs (TAP 2011).* Vol. 6706. LNCS. Springer, June 2011, pp. 78–83. ISBN: 978-3-642-21767-8. DOI: 10.1007/978-3-642-21768-5_7. Rank B.

[54] Nikolai Kosmatov, Bernard Botella, Muriel Roger, and Nicky Williams. “Online Test Generation with PathCrawler: Tool Demo.” In: *Proc. of the 3rd International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA 2011), co-located with the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST 2011).* **Best tool demo award**. IEEE, Mar. 2011, pp. 316–317. ISBN: 978-0-7695-4345-1. DOI: 10.1109/ICSTW.2011.85.

[55] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. “Combining Static Analysis and Test Generation for C Program Debugging.” In: *Proc. of the 4th International Conference on Tests and Proofs (TAP 2010).* Vol. 6706. LNCS. Springer, July 2010, pp. 94–100. ISBN: 978-3-642-21767-8. DOI: 10.1007/978-3-642-13977-2_9. Rank B.

[56] Bernard Botella, Mickaël Delahaye, Stéphane Hong Tuan Ha, Nikolai Kosmatov, Patricia Mouy, Muriel Roger, and Nicky Williams. “Automating Structural Testing of C Programs: Experience with PathCrawler.” In: *Proc. of the 4th International Workshop on the Automation of Software Test (AST 2009), part of the 31st International Conference on Software Engineering (ICSE 2009).* IEEE, 2009, pp. 70–78. ISBN: 978-1-4244-3711-5. DOI: 10.1109/IWAST.2009.5069043.

[57] Nikolai Kosmatov. “On Complexity of All-Paths Test Generation. From Practice to Theory.” In: *Proc. of Testing: Academic and Industrial Conference on Practice and Research Techniques (TAIC PART 2009).* Vol. 0. IEEE, Sept. 2009, pp. 144–153. ISBN: 978-0-7695-3820-4. DOI: 10.1109/TAICPART.2009.26.

[58] Nikolai Kosmatov. “All-Paths Test Generation for Programs with Internal Aliases.” In: *Proc. of the 19th International Symposium on Software Reliability Engineering (ISSRE 2008).* IEEE, 2008, pp. 147–156. ISBN: 978-0-7695-3405-3. DOI: 10.1109/ISSRE.2008.25. Rank A.

[59] Nikolai Kosmatov. “A constraint solver for sequences and its applications.” In: *Proc. of the 21th Annual ACM Symposium on Applied Computing, Constraint Solving and Programming Track (SAC-CSP 2006).* ACM, Apr. 2006, pp. 404–408. ISBN: 1-59593-108-2. DOI: 10.1145/1141277.1141369. Rank B.

[60] Nikolai Kosmatov. “Constraint Solving for Sequences in Software Validation and Verification.” In: *Revised Selected Papers of the 16th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2005)*. Vol. 4369. LNCS. Springer, 2006, pp. 25–37. ISBN: 978-3-540-69233-1. DOI: 10.1007/11963578_3.

[61] Jean-François Couchot, Alain Giorgetti, and Nikolai Kosmatov. “A uniform deductive approach for parameterized protocol safety.” In: *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*. ACM, 2005, pp. 364–367. ISBN: 1-58113-993-4. DOI: 10.1145/1101908.1101971. Rank A.

[62] Nikolai Kosmatov. “A Constraint Solver for Sequences.” In: *Proc. of the 1st International Workshop on Constraint Programming Beyond Finite Integer Domains (BeyondFD 2005), part of the 11th International Conference on Principles and Practice of Constraint Programming (CP 2005)*. Oct. 2005, pp. 49–54.

[63] Nikolai Kosmatov, Bruno Legeard, Fabien Peureux, and Mark Utting. “Boundary Coverage Criteria for Test Generation from Formal Models.” In: *Proc. of the 15th International Symposium on Software Reliability Engineering (ISSRE 2004)*. Nov. 2004, pp. 139–150. ISBN: 0-7695-2215-7. DOI: 10.1109/ISSRE.2004.12. Rank A.

Peer-Reviewed National Conferences and Workshops

[64] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. “Why3 a dit : gardez le contrôle en toute situation.” In: *Actes des 29mes Journées Francophones des Langages Applicatifs (JFLA 2018)*. In French. 7 pages. Jan. 2018.

[65] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. “Coq a dit : fromage tranché ne peut cacher ses trous.” In: *Actes des 27mes Journées Francophones des Langages Applicatifs (JFLA 2016)*. In French. 3 pages. Jan. 2016.

[66] Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. “Rester statique pour devenir plus rapide, plus précis et plus mince.” In: *Actes des 26mes Journées Francophones des Langages Applicatifs (JFLA 2015)*. In French. 15 pages. Jan. 2015.

[67] Nikolai Kosmatov. “Génération de tests "tous-les-chemins" : quelle complexité pour quelles contraintes?” In: *Actes des Sixièmes Journées Francophones de Programmation par Contraintes (JFPC 2010)*. In French. June 2010.

[68] Yohan Boichut, Nikolai Kosmatov, and Laurent Vigneron. “Validation of Prouvé protocols using the automatic tool TA4SP.” In: *Proc. of the 3rd Taiwanese-French Conference on Information Technology (TFIT 2006)*. Mar. 2006, pp. 467–480.

Invited and Tutorial Papers

[69] Sébastien Bardin, Nikolai Kosmatov, Bruno Marre, David Mentré, and Nicky Williams. "Test Case Generation with PathCrawler/LTest: How to Automate an Industrial Testing Process." In: *Proc. of the 8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Part IV. Industrial Practice. (ISOLA 2018).* Vol. 11247. LNCS. Springer, Nov. 2018, pp. 104–120. ISBN: 978-3-030-03426-9. DOI: 10.1007/978-3-030-03427-6_12. Rank C.

[70] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. "A Lesson on Verification of IoT Software with Frama-C." In: *Proc. of the 2018 International Conference on High Performance Computing & Simulation (HPCS 2018).* IEEE, July 2018, pp. 21–30. ISBN: 978-1-5386-7878-7. DOI: 10.1109/HPCS.2018.00018. Rank B.

[71] Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. "Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014." In: *Proc. of the 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2016).* Vol. 9952. LNCS. Springer, Oct. 2016, pp. 461–478. ISBN: 978-3-319-47165-5. DOI: 10.1007/978-3-319-47166-2_32. Rank C.

[72] Nikolai Kosmatov and Julien Signoles. "Frama-C, A Collaborative Framework for C Code Verification: Tutorial Synopsis." In: *Proc. of the 7th International Conference on Runtime Verification (RV 2016).* Vol. 10012. LNCS. Springer, Sept. 2016, pp. 92–115. ISBN: 978-3-319-46981-2. DOI: 10.1007/978-3-319-46982-9_7. Rank C.

[73] Nikolai Kosmatov and Julien Signoles. "Runtime Assertion Checking and Its Combinations with Static and Dynamic Analyses – Tutorial Synopsis." In: *Proc. of the 8th International Conference on Tests and Proofs (TAP 2014), Held as Part of STAF 2014.* Vol. 8570. LNCS. Springer, 2014, pp. 165–168. ISBN: 978-3-319-09098-6. DOI: 10.1007/978-3-319-09099-3_13. Rank B.

[74] Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles. "A Lesson on Proof of Programs with Frama-C. Invited Tutorial Paper." In: *Proc. of the 7th International Conference on Tests and Proofs (TAP 2013).* Vol. 7942. LNCS. Springer, June 2013, pp. 168–177. ISBN: 978-3-642-38915-3. DOI: 10.1007/978-3-642-38916-0_10. Rank B.

[75] Nikolai Kosmatov and Julien Signoles. "A Lesson on Runtime Assertion Checking with Frama-C." In: *Proc. of the 4th International Conference on Runtime Verification (RV 2013).* Vol. 8174. LNCS. Springer, 2013, pp. 386–399. ISBN: 978-3-642-40786-4. DOI: 10.1007/978-3-642-40787-1_29. Rank C.

[76] Nikolai Kosmatov and Nicky Williams. "Tutorial on Automated Structural Testing with PathCrawler – Extended Abstract." In: *Proc. of the 6th International Conference on Tests and Proofs (TAP 2012)*. Vol. 7305. LNCS. Springer, May 2012, p. 176. ISBN: 978-3-642-30472-9. DOI: `10.1007/978-3-642-30473-6_16`. Rank B.

[77] Nicky Williams and Nikolai Kosmatov. "Structural Testing with PathCrawler: Tutorial Synopsis." In: *Proc. of the 12th International Conference on Quality Software (QSIC 2012)*. IEEE, Aug. 2012, pp. 289–292. ISBN: 978-1-4673-2857-9. DOI: `10.1109/QSIC.2012.24`. Rank B.

Software

[78] Sébastien Bardin, Omar Chebaro, Robin David, Mickaël Delahaye, Nikolai Kosmatov, Thibault Martin, and Virgile Prevosto. *The LTest test generation toolset*. 2014–2018. URL: `https://github.com/ltest-dev/LTest/`.

[79] Nikolai Kosmatov et al. *The PathCrawler online test generation service*. 2010–2018. URL: `http://pathcrawler-online.com/`.

[80] Nikolai Kosmatov. *The CHR solver for sequences*. 2005. URL: `http://nikolai.kosmatov.free.fr/sequences/`.

Publications in Mathematics

Peer-Reviewed International Journals

[81] Alexander Generalov and Nikolai Kosmatov. "Projective resolutions and Yoneda algebras for algebras of dihedral type." In: *Algebras and Repr. Theory* 10.3 (2007), pp. 241–256. ISSN: 1572-9079. DOI: `10.1007/s10468-006-9012-7`.

Peer-Reviewed National Journals

[82] Alexander Generalov and Nikolai Kosmatov. "Computation of the Yoneda Algebras for Algebras of Dihedral Type"." In: *J. Math. Sci.* 130.3 (2005), pp. 4699–4711. ISSN: 1573-8795. DOI: `10.1007/s10958-005-0364-z`.

[83] Alexander Generalov and Nikolai Kosmatov. "Projective resolutions and Yoneda algebras for algebras of dihedral type: the family D(3Q)." In: *Fund. Appl. Math.* 10.4 (2004), pp. 65–89.

[84] Nikolai Kosmatov. "Bounds on Homological Dimensions of Pullbacks. II." In: *J. Math. Sci.* 116.1 (2003), pp. 3014–3015. ISSN: 1573-8795. DOI: `10.1023/A:1023551328399`.

[85] Nikolai Kosmatov. “Bounds for the Homological Dimensions of Pullbacks.” In: *J. Math. Sci.* 112.4 (2002), pp. 4367–4370. ISSN: 1573-8795. DOI: 10.1023/A:1020351104689.

[86] Nikolai Kosmatov. “An estimate for the global dimension of pullback rings.” In: *Fund. Appl. Math.* 5.4 (1999). In Russian., pp. 1251–1253.

Software

[87] Nikolai Kosmatov. *The Resolut program to compute projective resolutions*. 2002. URL: http://nikolai.kosmatov.free.fr/resolut/.

Personal Webpage

[88] Nikolai Kosmatov. *Personal webpage*. 2018. URL: http://nikolai.kosmatov.free.fr/.

Other References

The remaining references are sorted by year (in the inversed order).

[89] *Testwell CTC++: Test Coverage Analyzer for C/C++*. URL: http://www.testwell.fi/ctcdesc.html.

[90] *Testworks: TCAT C/C++*. URL: http://www.testworks.com/Products/Coverage/tcat.html.

[91] *Semantic Designs: C Test Coverage Tool*. URL: http://semanticdesigns.com/Products/TestCoverage/.

[92] Polyspace Development Team. *The Polyspace Software verification tool*. URL: http://mathworks.com/products/polyspace/.

[93] *Parasoft C/C++test: Comprehensive dev. testing tool for C/C++*. URL: https://www.parasoft.com/product/cpptest/.

[94] *LDRA – LDRACover*. URL: http://www.ldra.com/en/ldracover.

[95] *Intel Code Coverage Tool in Intel C++ compiler*. URL: https://software.intel.com/en-us/node/522743.

[96] *GCC's Gcov*. URL: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[97] *Dynamic Code Coverage*. URL: http://dynamic-memory.com/.

[98] *Bullseye Testing Technology: BullseyeCoverage*. URL: http://bullseye.com/.

[99] François Pessaux et al. *The FoCaLiZe project*. URL: http://focalize.inria.fr.

[100] Patrick Baudin, François Bobot, Loïc Correnson, and Zaynah Dargaye. *WP Plug-in Manual*. URL: http://frama-c.com/wp.html.

[101] Allan Blanchard. *The Conc2Seq verification plugin for Frama-C*. URL: https://github.com/AllanBlanchard/Frama-C-Conc2Seq.

[102] Jochen Burghardt and Jens Gerlach. *ACSL by Example*. URL: https://github.com/fraunhoferfokus/acsl-by-example.

[103] *COVTOOL - Free test coverage analyzer for C++*. URL: http://covtool.sourceforge.net/.

[104] Frama-C Development Team. *The Frama-C website*. URL: https://frama-c.com.

[105] Guillaume Petiot. *The StaDy verification plugin for Frama-C*. URL: https://github.com/gpetiot/Frama-C-StaDy.

[106] *PurifyPlus: Run-Time Analysis Tools for Application Reliability and Performance*. URL: http://teamblue.unicomsi.com/products/purifyplus/.

[107] Microsoft Research. *The Pex4Fun online testing service*. URL: `http://www.pex4fun.com`.

[108] Why3 development team. *Why3, a tool for deductive program verification*. URL: `http://why3.lri.fr`.

[109] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. 2018. URL: `http://frama-c.com/acsl.html`.

[110] Coq Development Team. *The Coq Proof Assistant Reference Manual*. 2018. URL: `http://coq.inria.fr/`.

[111] Sylvain Dailler, David Hauzar, Claude Marché, and Yannick Moy. "Instrumenting a weakest precondition calculus for counterexample generation." In: *J. Log. Algebr. Meth. Program.* 99 (2018), pp. 97–113.

[112] Jean-Christophe Léchenet. "Certified Algorithms for Program Slicing." PhD thesis defended on July 19, 2018. PhD thesis. CentraleSupélec, 2018.

[113] Julien Signoles. *From Static Analysis to Runtime Verification with Frama-C and E-ACSL*. Habilitation thesis defended on July 9, 2018. 2018.

[114] Sandrine Blazy, David Bühler, and Boris Yakobowski. "Structuring Abstract Interpreters Through State and Value Abstractions." In: *Proc. of the 18th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2017)*. Vol. 10145. LNCS. Springer, 2017, pp. 112–130.

[115] Raphaël Cauderlier and Catherine Dubois. "FoCaLiZe and Dedukti to the Rescue for Proof Interoperability." In: *Proc. of the 8th International Conference on Interactive Theorem Proving (ITP 2017)*. Vol. 10499. LNCS. Springer, 2017, pp. 131–147.

[116] Reiner Hähnle and Marieke Huisman. "24 Challenges in Deductive Software Verification." In: *Proc. of the 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements (ARCADE 2017)*. Vol. 51. EPiC Series in Computing. EasyChair, 2017, pp. 37–41. To appear in LNCS, vol. 10000, Springer.

[117] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, eds. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. LNCS. Springer, 2016.

[118] Wolfgang Ahrendt, Christoph Gladisch, and Mihai Herda. "Proof-based Test Case Generation." In: *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. LNCS. Springer, 2016, pp. 415–451.

[119] Romain Aïssat, Marie-Claude Gaudel, Frédéric Voisin, and Burkhart Wolff. “A Method for Pruning Infeasible Paths via Graph Transformations and Symbolic Execution.” In: *Proc. of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS 2016)*. IEEE, 2016, pp. 144–151.

[120] Romain Aïssat, Frédéric Voisin, and Burkhart Wolff. “Infeasible Paths Elimination by Symbolic Execution Techniques: Proof of Correctness and Preservation of Paths.” In: *Archive of Formal Proofs* 2016 (2016).

[121] Gilles Barthe, Juan Manuel Crespo, and César Kunz. “Product programs and relational program logics.” In: *J. Log. Algebr. Meth. Program.* 85.5 (2016), pp. 847–859.

[122] Allan Blanchard. “Aide à la vérification de programmes concurrents par transformation de code et de spécifications.” (In French). PhD thesis. Univ. d’Orléans, 2016.

[123] Maria Christakis, K. Rustan M. Leino, Peter Müller, and Valentin Wüstholz. “Integrated Environment for Diagnosing Verification Errors.” In: *Proc. of the 22th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016)*. Vol. 9636. LNCS. Springer, 2016, pp. 424–441.

[124] Marieke Huisman, Wolfgang Ahrendt, Daniel Grahl, and Martin Hentschel. “Formal Specification with the Java Modeling Language.” In: *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. LNCS. Springer, 2016, pp. 193–241.

[125] Marcelo Sousa and Isil Dillig. “Cartesian Hoare Logic for Verifying k-safety Properties.” In: *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*. ACM, 2016, pp. 57–69.

[126] Stephan Arlt, Sergio Feo Arenis, Andreas Podelski, and Martin Wehrle. “System Testing and Program Verification.” In: *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI)*. Vol. 239. LNI. GI, 2015, pp. 71–72.

[127] Sandrine Blazy, André Maroneze, and David Pichardie. “Verified Validation of Program Slicing.” In: *Proc. of the 2015 Conference on Certified Programs and Proofs (CPP 2015)*. ACM, 2015, pp. 109–117.

[128] Jesús Mauricio Chimento, Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. “StaRVOOrS: A Tool for Combined Static and Runtime Verification of Java.” In: *Proc. of the 6th International Conference on Runtime Verification (RV 2015)*. Vol. 9333. LNCS. Springer, 2015, pp. 297–305.

[129] Alex Groce, Jervis Pinto, Pooria Azimi, and Pranjal Mittal. “TSTL: A Language and Tool for Testing (Demo).” In: *Proc. of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, 2015, pp. 414–417.

[130] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. “Assertion guided symbolic execution of multithreaded programs.” In: *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, 2015, pp. 854–865.

[131] Johannes Kinder. “Hypertesting: The Case for Automated Testing of Hyperproperties.” In: *Proc. of the 3rd Workshop on Hot Issues in Security Principles and Trust (HotSpot 2015)*. 2015.

[132] Guillaume Petiot. “Contribution à la vérification de programmes C par combinaison tests et preuves.” (In French). PhD thesis. Univ. de Franche-Comté, 2015.

[133] Maria Christakis, Patrick Emmisberger, and Peter Müller. “Dynamic Test Generation with Static Fields and Initializers.” In: *Proc. of the 5th International Conference on Runtime Verification (RV 2014)*. Vol. 8734. LNCS. Springer, 2014, pp. 269–284.

[134] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. “Temporal Logics for Hyperproperties.” In: *Proc. of the Third International Conference on Principles of Security and Trust (POST 2014)*. Springer Berlin Heidelberg, 2014, pp. 265–284.

[135] David R. Cok. “OpenJML: Software Verification for Java 7 using JML, OpenJDK, and Eclipse.” In: *Proc. of the 1st Workshop on Formal Integrated Development Environment, (F-IDE 2014)*. Vol. 149. EPTCS. 2014, pp. 79–92.

[136] Loïc Correnson. “Qed. Computing What Remains to Be Proved.” In: *Proc. of the 6th International Symposium on NASA Formal Methods (NFM 2014)*. Vol. 8430. LNCS. Springer, 2014, pp. 215–229.

[137] Pablo Abad, Nazareno Aguirre, Valeria S. Bengolea, Daniel Ciolek, Marcelo F. Frias, Juan P. Galeotti, Tom Maibaum, Mariano M. Moscato, Nicolás Rosner, and Ignacio Vissani. “Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving.” In: *Proc of the IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST 2013)*. IEEE, 2013, pp. 21–30.

[138] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. “An orchestrated survey of methodologies for automated software test case generation.” In: *Journal of Systems and Software* 86.8 (2013), pp. 1978–2001.

[139] Peter G. Bishop, Robin E. Bloomfield, and Lukasz Cyra. "Combining testing and proof to gain high assurance in software: A case study." In: *Proc. of the 24th International Symposium on Software Reliability Engineering (ISSRE 2013).* IEEE, 2013, pp. 248–257.

[140] Jürgen Christ, Evren Ermis, Martin Schäf, and Thomas Wies. "Flow-Sensitive Fault Localization." In: *Proc. of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2013).* Vol. 7737. LNCS. Springer, 2013, pp. 189–208.

[141] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. "Automatic Inference of Necessary Preconditions." In: *Proc. of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2013).* Vol. 7737. LNCS. Springer, 2013, pp. 128–148.

[142] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 – Where Programs Meet Provers." In: *Proc. of the 22nd European Symposium on Programming (ESOP 2013).* Vol. 7792. LNCS. Springer, 2013, pp. 125–128.

[143] Zhan-Wei Hui and Song Huang. "A Formal Model for Metamorphic Relation Decomposition." In: *Proc. of the Fourth World Congress on Software Engineering (WCSE 2013).* ACM, 2013, pp. 64–68.

[144] Konrad Jamrozik, Gordon Fraser, Nikolai Tillman, and Jonathan de Halleux. "Generating Test Suites with Augmented Dynamic Symbolic Execution." In: *Proc of the 7th International Conference on Tests and Proofs (TAP 2013).* Springer, 2013, pp. 152–167.

[145] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. "Program Checking with Less Hassle." In: *Proc. of the 5th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2013).* Vol. 8164. LNCS. Springer, 2013, pp. 149–169.

[146] José Bernardo Barros, Daniela Carneiro da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. "Assertion-based slicing and slice graphs." In: *Formal Asp. Comput.* 24.2 (2012), pp. 217–248.

[147] Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. "A First Step in the Design of a Formally Verified Constraint-Based Testing Tool: FocalTest." In: *Proc. of the 6th International Conference on Tests and Proofs (TAP 2012).* Vol. 7305. LNCS. Springer, 2012, pp. 35–50.

[148] Maria Christakis, Peter Müller, and Valentin Wüstholz. "Collaborative Verification and Testing with Explicit Assumptions." In: *Proc. of the 18th International Symposium on Formal Methods (FM 2012).* Vol. 7436. LNCS. Springer, 2012, pp. 132–146.

[149] Loïc Correnson and Julien Signoles. “Combining Analyses for C Program Verification.” In: *Proc. of the 17th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2012)*. Vol. 7437. Springer, 2012, pp. 108–130.

[150] Rayna Dimitrova and Bernd Finkbeiner. “Counterexample-Guided Synthesis of Observation Predicates.” In: *Proc. of the International Conference on Formal Modeling and Analysis of Timed Systems - 10th International Conference (FORMATS 2012)*. Vol. 7595. LNCS. Springer, 2012, pp. 107–122.

[151] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. “AddressSanitizer: A Fast Address Sanity Checker.” In: *Proc. of the USENIX Annual Technical Conference*. USENIX Association, 2012, pp. 309–319.

[152] Josep Silva. “A vocabulary of program slicing-based techniques.” In: *ACM Comput. Surv.* 44.3 (2012), p. 12.

[153] Gilles Barthe, Pedro R. D’Argenio, and Tamara Rezk. “Secure information flow by self-composition.” In: *J. Mathematical Structures in Computer Science* 21.6 (2011), pp. 1207–1252.

[154] Derek Bruening and Qin Zhao. “Practical memory checking with Dr. Memory.” In: *Proc. of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE Computer Society, 2011, pp. 213–223.

[155] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. “Integrating Testing and Interactive Theorem Proving.” In: *Proc. 10th International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 2011)*. Vol. 70. EPTCS. 2011, pp. 4–19.

[156] Omar Chebaro. “Classification de menaces d’erreurs par analyse statique, simplification syntaxique et test structurel de programmes.” (In French). PhD thesis. Univ. de Franche-Comté, 2011.

[157] Tsong Yueh Chen, T. H. Tse, and Zhiquan Zhou. “Semi-Proving: An Integrated Method for Program Proving, Testing, and Debugging.” In: *IEEE Transactions on Software Engineering* 37.1 (2011), pp. 109–125.

[158] Sebastian Danicic, Richard W. Barraclough, Mark Harman, John Howroyd, Ákos Kiss, and Michael R. Laurence. “A unifying theory of control dependence and its application to arbitrary program structures.” In: *Theor. Comput. Sci.* 412.49 (2011), pp. 6809–6842.

[159] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. “DyTa: dynamic symbolic execution guided with static verification results.” In: *Proc. of the 33rd International Conference on Software Engineering (ICSE 2011)*. ACM, 2011, pp. 992–994.

[160] Peter Müller and Joseph N. Ruskiewicz. "Using Debuggers to Understand Failed Verification Attempts." In: *Proc. of the 17th International Symposium on Formal Methods (FM 2011)*. Vol. 6664. LNCS. Springer, 2011, pp. 73–87.

[161] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd ed. Wiley, 2011.

[162] Daniel Wasserrab. "From formal semantics to verified slicing: a modular framework with applications in language based security." PhD thesis. Karlsruhe Inst. of Techn., 2011.

[163] Ki Yung Ahn and Ewen Denney. "Testing First-Order Logic Axioms in Program Verification." In: *Proc. of the 4th International Conference on Tests and Proofs (TAP 2010)*. Vol. 6143. LNCS. Springer, 2010, pp. 22–37.

[164] Richard W. Barraclough, David Binkley, Sebastian Danicic, Mark Harman, Robert M. Hierons, Akos Kiss, Mike Laurence, and Lahcen Ouarbya. "A trajectory-based strict semantics for program slicing." In: *Theor. Comp. Sci.* 411.11–13 (2010), pp. 1372–1386.

[165] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties." In: *J. Comput. Secur.* 18.6 (2010), pp. 1157–1210.

[166] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. "Test Generation Through Programming in UDITA." In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)*. ACM, 2010, pp. 225–234.

[167] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. "Compositional may-must program analysis: unleashing the power of alternation." In: *POPL*. ACM, 2010, pp. 43–56.

[168] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. "How Did You Specify Your Test Suite." In: *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*. ACM, 2010, pp. 407–416.

[169] Andreas Podelski and Thomas Wies. "Counterexample-guided focus." In: *Proc. of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*. ACM, 2010, pp. 249–260.

[170] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. "Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software." In: *Proc. of the 14th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2009)*. Vol. 5825. LNCS. Springer, 2009, pp. 53–69.

[171] Christoph Gladisch. “Could We Have Chosen a Better Loop Invariant or Method Contract?” In: *Proc. of the Third International Conference on Tests and Proofs (TAP 2009).* Vol. 5668. LNCS. Springer, 2009, pp. 74–89.

[172] Laura Kovács and Andrei Voronkov. “Finding Loop Invariants for Programs over Arrays Using a Theorem Prover.” In: *Proc. of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE 2009), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2009).* Vol. 5503. LNCS. Springer, 2009, pp. 470–485.

[173] Xavier Leroy. “Formal verification of a realistic compiler.” In: *Commun. ACM* 52.7 (2009), pp. 107–115.

[174] Härmel Nestra. “Transfinite Semantics in the Form of Greatest Fixpoint.” In: *J. Log. Algebr. Program.* 78.7 (2009), pp. 573–592.

[175] Qian Yang, J Jenny Li, and David M Weiss. “A survey of coverage-based testing tools.” In: *The Computer Journal* 52.5 (2009), pp. 589–597.

[176] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 1st ed. Cambridge University Press, 2008. ISBN: 0521880386, 9780521880381.

[177] Torben Amtoft. “Slicing for modern program structures: a theory for eliminating irrelevant loops.” In: *Inf. Process. Lett.* 106.2 (2008), pp. 45–51.

[178] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. “Proofs from tests.” In: *Proc. of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008).* ACM, 2008, pp. 3–14.

[179] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs.” In: *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008).* USENIX Association, 2008, pp. 209–224.

[180] Koen Claessen and Hans Svensson. “Finding Counter Examples in Induction Proofs.” In: *Proc. of the Second International Conference on Tests and Proofs (TAP 2008).* Vol. 4966. LNCS. Springer, 2008, pp. 48–65.

[181] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. “Active property checking.” In: *Proc. of the 8th ACM & IEEE International conference on Embedded software (EMSOFT 2008).* 2008, pp. 207–216.

[182] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. “Automated Whitebox Fuzz Testing.” In: *Proc. of the Network and Distributed System Security Symposium (NDSS 2008).* 2008.

[183] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. "FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement." In: *Proc. of the 20th International Conference on Computer Aided Verification (CAV 2008)*. Springer, 2008, pp. 209–213.

[184] K. Rustan M. Leino and Peter Müller. "Verification of Equivalent-Results Methods." In: *Proc. of the 17th European Symposium on Programming (ESOP 2008), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2008)*. Vol. 4960. LNCS. 2008, pp. 307–321.

[185] Aditya P. Mathur. *Foundations of Software Testing*. Addison-Wesley Professional, 2008.

[186] Nikolai Tillmann and Jonathan de Halleux. "Pex-White Box Test Generation for .NET." In: *Proc. of the 2nd International Conference on Tests and Proofs (TAP 2008)*. Vol. 4966. LNCS. Springer, 2008, pp. 134–153.

[187] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, eds. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, Heidelberg, 2007.

[188] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "The Software Model Checker Blast: Applications to Software Engineering." In: *Int. J. Softw. Tools Technol. Transfer* 9.5–6 (2007), pp. 505–525.

[189] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. "Designing a Generic Graph Library Using ML Functors." In: *Proc. of the 8th Symposium on Trends in Functional Programming (TFP 2007)*. Vol. 8. Intellect, 2007, pp. 124–140.

[190] Christian Engel and Reiner Hähnle. "Generating Unit Tests from Formal Proofs." In: *Proc. of the First International Conference on Tests and Proofs (TAP 2007)*. Vol. 4454. LNCS. Springer, 2007, pp. 169–188.

[191] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. "The Daikon system for dynamic detection of likely invariants." In: *Sci. Comput. Program.* 69.1–3 (2007), pp. 35–45.

[192] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. "A new foundation for control dependence and slicing for modern program structures." In: *ACM Trans. Program. Lang. Syst.* 29.5 (2007).

[193] Yannis Smaragdakis and Christoph Csallner. "Combining Static and Dynamic Reasoning for Bug Detection." In: *Proc. of the First International Conference on Tests and Proofs (TAP 2007)*. Vol. 4454. LNCS. Springer, 2007, pp. 1–16.

[194] David Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. "Theoretical foundations of dynamic program slicing." In: *Theor. Comput. Sci.* 360.1-3 (2006), pp. 23–41.

[195] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: automatically generating inputs of death." In: *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS 2006).* ACM Press, 2006, pp. 322–335.

[196] Adám Darvas and Peter Müller. "Reasoning About Method Calls in JML Specifications." In: *Journal of Object Technology* (2006).

[197] Catherine Dubois, Thérèse Hardin, and Véronique Donzeau-Gouge. "Building certified components within FOCAL." In: *Revised Selected Papers from the Fifth Symposium on Trends in Functional Programming (TFP 2004).* Vol. 5. Trends in Functional Programming. Intellect, 2006, pp. 33–48.

[198] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. "SYNERGY: a new algorithm for property checking." In: *Proc. of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2006).* ACM, 2006, pp. 117–127.

[199] Sam Owre. "Random Testing in PVS." In: *Workshop on Automated Formal Methods (AFM).* 2006.

[200] Koushik Sen and Gul Agha. "CUTE and jCUTE: concolic unit testing and explicit path model-checking tools." In: *Proc. of the 18th International Conference on Computer Aided Verification (CAV 2006).* Vol. 4144. LNCS. Springer, 2006.

[201] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. "The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications." In: *Proc. of the 17th International Conference on Computer Aided Verification (CAV 2005).* Vol. 3576. LNCS. Springer, 2005, pp. 281–285.

[202] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "The ASTRÉE Analyzer." In: *Proc. of the 14th European Symposium on Programming (ESOP 2005), part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2005).* Vol. 3444. LNCS. Springer, 2005, pp. 21–30.

[203] Christoph Csallner and Yannis Smaragdakis. "Check'n'crash: combining static checking and testing." In: *Proc. of the 27th International Conference on Software Engineering (ICSE 2005).* ACM, 2005, pp. 422–431.

[204] Corina S. Pasareanu, Radek Pelánek, and Willem Visser. “Concrete Model Checking with Abstract Matching and Refinement.” In: *Proc. of the 17th International Conference on Computer Aided Verification (CAV 2005)*. Vol. 3576. LNCS. Springer, 2005, pp. 52–66.

[205] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C.” In: *the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005)*. ACM, 2005, pp. 263–272.

[206] Julian Seward and Nicholas Nethercote. “Using Valgrind to Detect Undefined Value Errors with Bit-Precision.” In: *Proc. of the USENIX Annual Technical Conference*. USENIX, 2005, pp. 17–30.

[207] Nicki Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. “PathCrawler: automatic generation of path tests by combining static and dynamic analysis.” In: *Proc. of the 5th European Dependable Computing Conference (EDCC 2005)*. Vol. 3463. LNCS. Springer, 2005, pp. 281–292.

[208] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. “A brief survey of program slicing.” In: *ACM SIGSOFT Software Engineering Notes* 30.2 (2005), pp. 1–36.

[209] Nick Benton. “Simple relational correctness proofs for static analyses and program transformations.” In: *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*. 2004, pp. 14–25.

[210] Stefan Berghofer and Tobias Nipkow. “Random Testing in Isabelle/HOL.” In: *Proc. of the 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*. IEEE Computer Society, 2004, pp. 230–239.

[211] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development; Coq’Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[212] David Binkley and Mark Harman. “A survey of empirical results on program slicing.” In: *Advances in Computers* 62 (2004), pp. 105–178.

[213] David R. Cok and Joseph R. Kiniry. “ESC/Java2: Uniting ESC/Java and JML.” In: *Proc. of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2004)*. Vol. 3362. LNCS. Springer, 2004, pp. 108–128.

[214] Alex Groce, Daniel Kroening, and Flavio Lerda. "Understanding Counterexamples with explain." In: *Proc. of the 16th International Conference on Computer Aided Verification (CAV 2004)*. Vol. 3114. LNCS. Springer, 2004, pp. 453–456.

[215] Matthew Allen and Susan Horwitz. "Slicing Java programs that throw and catch exceptions." In: *Proc. of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM 2003)*. 2003, pp. 44–54.

[216] Paul Ammann, A. Jefferson Offutt, and Hong Huang. "Coverage Criteria for Logical Expressions." In: *Proc. of the 14th International Symposium on Software Reliability Engineering (ISSRE 2003)*. 2003, pp. 99–107.

[217] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided Abstraction Refinement for Symbolic Model Checking." In: *J. ACM* 50.5 (2003), pp. 752–794.

[218] Alain Deutsch. *Static verification of dynamic properties*. Tech. rep. White paper. Nov. 2003.

[219] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. "Combining Testing and Proving in Dependent Type Theory." In: *Proc. of the International Conference on Theorem Proving in Higher Order Logics, 16th International Conference (TPHOLs 2003)*. Vol. 2758. LNCS. Springer, 2003, pp. 188–203.

[220] Roberto Giacobazzi and Isabella Mastroeni. "Non-Standard Semantics for Program Slicing." In: *Higher-Order and Symbolic Computation* 16.4 (2003), pp. 297–339.

[221] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. "An Introduction to the Testing and Test Control Notation (TTCN-3)." In: *Comput. Netw.* 42.3 (2003), pp. 375–403.

[222] Ina Schieferdecker, Zhen Ru Dai, Jens Grabowski, and Axel Rennoch. "The UML 2.0 Testing Profile and Its Relation to TTCN-3." In: *Proc. of the 15th IFIP International Conference on Testing of Communicating Systems (TestCom 2003)*. Springer, 2003, pp. 79–94.

[223] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. "CASL: the Common Algebraic Specification Language." In: *Theoretical Computer Science* 286.2 (2002). Current trends in Algebraic Development Techniques, pp. 153–196.

[224] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. "How the Design of JML Accomodates Both Runtime Assertion Checking and Formal Verification." In: *Proc. of the 1st International Symposium on Formal Methods for Components and Objects (FMCO 2002)*. Vol. 2852. LNCS. Springer, 2002, pp. 262–284.

[225] Bruno Marre and Agnès Arnould. “Test sequences generation from Lustre descriptions: GATeL.” In: *Proc. of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*. IEEE, 2000, pp. 229–237.

[226] Robert M. Hierons, Mark Harman, and Sebastian Danicic. “Using Program Slicing to Assist in the Detection of Equivalent Mutants.” In: *Softw. Test., Verif. Reliab.* 9.4 (1999), pp. 233–262.

[227] Famantanantsoa Randimbivololona, Jean Souyris, Patrick Baudin, Anne Pacalet, Jacques Raguideau, and Dominique Schoen. “Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach.” In: *Proc. of the World Congress on Formal Methods in the Development of Computing Systems (FM 1999)*. Vol. 1709. LNCS. Springer, 1999, pp. 1798–1815.

[228] Phyllis G Frankl and Oleg Iakounenko. “Further empirical studies of test effectiveness.” In: *ACM SIGSOFT Softw. Eng. Notes* 23.6 (1998).

[229] Alain Deutsch. “On the Complexity of Escape Analysis.” In: *Proc. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997)*. ACM, 1997, pp. 358–371.

[230] Susanne Graf and Hassen Saïdi. “Construction of Abstract State Graphs with PVS.” In: *Proc. of the 9th International Conferenceon Computer Aided Verification (CAV 1997)*. Vol. 1254. LNCS. Springer, 1997, pp. 72–83.

[231] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types*. John Wiley & Sons, Inc., 1997.

[232] Bertrand Meyer. *Object-oriented Software Construction, Second Edition*. New York: Object-oriented Series, Prentice Hall, 1997.

[233] Hong Zhu, Patrick A. V. Hall, and John H. R. May. “Software Unit Test Coverage and Adequacy.” In: *ACM Comput. Surv.* 29.4 (1997), pp. 366–427.

[234] Gianfranco Bilardi and Keshav Pingali. “Generalized Dominance and Control Dependence.” In: *Proc. of the ACM SIGPLAN’96 Conference on Programming Language Design and Implementation (PLDI 1996)*. ACM, 1996, pp. 291–300.

[235] Mark Harman, Dan Simpson, and Sebastian Danicic. “Slicing programs in the presence of errors.” In: *Formal Asp. Comput.* 8.4 (1996), pp. 490–497.

[236] Bogdan Korel and Ali M. Al-Yami. “Assertion-Oriented Automated Test Data Generation.” In: *Proc. of the 18th International Conference on Software Engineering*. IEEE Computer Society, 1996, pp. 71–80.

[237] Mark Harman and Sebastian Danicic. “Using Program Slicing to Simplify Testing.” In: *Softw. Test., Verif. Reliab.* 5.3 (1995), pp. 143–162.

[238] Frank Tip. “A survey of program slicing techniques.” In: *J. Prog. Lang.* 3.3 (1995).

[239] John Joseph Chilenski and Steven P. Miller. “Applicability of modified condition/decision coverage to software testing.” In: *Software Engineering Journal* (1994).

[240] Allen Goldberg, T C Wang, and David Zimmerman. “Applications of feasible path analysis to program testing.” In: *Proc. of the International Symposium on Software Testing and Analysis (ISSTA 1994)*. ACM, 1994.

[241] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. “Debugging with Dynamic Slicing and Backtracking.” In: *Softw., Pract. Exper.* 23.6 (1993), pp. 589–616.

[242] Thomas Ball and Susan Horwitz. “Slicing Programs with Arbitrary Control-flow.” In: *Proc. of the First International Workshop on Automated and Algorithmic Debugging (AADEBUG 1993)*. 1993, pp. 206–222.

[243] Elaine J. Weyuker. “More experience with data flow testing.” In: *IEEE Trans. Softw. Eng.* 19.9 (1993).

[244] Patrick Cousot and Radhia Cousot. “Abstract Interpretation Frameworks.” In: *J. Log. Comput.* 2.4 (1992), pp. 511–547.

[245] Reed Hastings and Bob Joyce. “Purify: Fast detection of memory leaks and access errors.” In: *Proc. of the Winter USENIX Conference*. San Francisco, California, USA, Jan. 1992, pp. 125–136.

[246] Andy Podgurski and Lori A. Clarke. “A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance.” In: *IEEE Trans. Software Eng.* 16.9 (1990), pp. 965–979.

[247] Robert Cartwright and Matthias Felleisen. “The Semantics of Program Dependence.” In: *Proc. of the ACM SIGPLAN’89 Conference on Programming Language Design and Implementation (PLDI 1989)*. ACM, 1989, pp. 13–27.

[248] Thomas W. Reps and Wuu Yang. “The Semantics of Program Slicing and Program Integration.” In: *Proc. of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT 1989), Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Current Issues in Programming Languages (CCIPL)*. Vol. 352. LNCS. Springer, 1989, pp. 360–374.

[249] Derek F. Yates and Nicos Malevris. “Reducing the effects of infeasible paths in branch testing.” In: *ACM SIGSOFT Softw. Eng. Notes* 14.8 (1989), pp. 48–54.

[250] Susan Horwitz, Thomas Reps, and David Binkley. “Interprocedural slicing using dependence graphs.” In: *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1988)*. ACM, 1988, pp. 35–46.

[251] Thomas W. Reps and Wuu Yang. *The Semantics of Program Slicing*. Tech. rep. Univ. of Wisconsin, 1988.

[252] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The program dependence graph and its use in optimization.” In: *ACM Trans. Program. Lang. Syst.* 9.3 (1987), pp. 319–349.

[253] Karl J. Ottenstein and Linda M. Ottenstein. “The program dependence graph in a software development environment.” In: *Proc. of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1984)*. ACM Press, 1984, pp. 177–184.

[254] Janusz W. Laski and Bogdan Korel. “A Data Flow Oriented Program Testing Strategy.” In: *IEEE Trans. Software Eng.* 9.3 (1983).

[255] Jean-Pierre Queille and Joseph Sifakis. “Specification and verification of concurrent systems in CESAR.” In: *Proc. of the International Symposium on Programming*. Vol. 137. LNCS. Springer, 1982, pp. 337–351.

[256] Mark Weiser. “Programmers Use Slices When Debugging.” In: *Commun. ACM* 25.7 (1982), pp. 446–452.

[257] Elaine J. Weyuker. “On Testing Non-Testable Programs.” In: *Comput. J.* 25.4 (1982), pp. 465–470.

[258] Mark Weiser. “Program slicing.” In: *Proc. of the 5th International Conference on Software Engineering (ICSE 1981)*. IEEE Computer Society, 1981, pp. 439–449.

[259] E. Allen Emerson and Edmund M. Clarke. “Characterizing Correctness Properties of Parallel Programs Using Fixpoints.” In: *Proc. of the 7th Colloquium on Automata, Languages and Programming*. Vol. 85. LNCS. Springer, 1980, pp. 169–181.

[260] Martin R. Woodward, David Hedley, and Michael A. Hennell. “Experience with Path Analysis and Testing of Programs.” In: *IEEE Trans. Softw. Eng.* SE-6.3 (1980), pp. 278–286.

[261] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. “Hints on Test Data Selection: Help for the Practicing Programmer.” In: *Computer* (1978).

[262] Patrick Cousot and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.” In: *Proc. of the 4th ACM Symposium on Principles of Programming Languages (POPL 1977)*. ACM, 1977, pp. 238–252.

[263] Dorothy E. Denning and Peter J. Denning. “Certification of Programs for Secure Information Flow.” In: *Commun. ACM* 20.7 (1977), pp. 504–513.