Test Case Generation with PathCrawler/LTest: How to Automate an Industrial Testing Process

Sébastien Bardin[1], Nikolai Kosmatov[1](✉), Bruno Marre[1], David Mentré[2], and Nicky Williams[1]

[1] CEA, List, Software Reliability and Security Lab, PC 174, Gif-sur-Yvette, France
{sebastien.bardin,nikolai.kosmatov,bruno.marre,nicky.williams}@cea.fr
[2] Mitsubishi Electric R&D Centre Europe (MERCE), Rennes, France
d.mentre@fr.merce.mee.com

Abstract. Automatic white-box testing based on formal methods is now a relatively mature technology and operational tools are available. Despite this, and the cost of manual testing, the technology is still rarely applied in an industrial setting. This paper describes how the specific needs of the user can be taken into account in order to build the necessary interface with a generic test tool. We present PathCrawler/LTest, a generator of test inputs for structural coverage of C functions, recently extended to support labels. Labels offer a generic mechanism for specification of code coverage criteria and make it possible to prototype and implement new criteria for specific industrial needs. We describe the essential participation of the research branch of an industrial user in bridging the gap between the tool developers and their business unit and adapting PathCrawler/LTest to the needs of the latter. We present the excellent results so far of their ongoing adoption and finish by mentioning possible improvements.

1 Introduction

In current software engineering practice, testing [3,25,27,34] is the primary approach to find errors in a program. Testing all possible program inputs being intractable in practice, the software testing community has long worked on the question of *test selection*: which test inputs to choose in order be confident that most, if not all, errors have been found by the tests. This work has resulted in proposals of various *testing criteria* (a.k.a. *adequacy criteria*) [3,34], including *code-coverage criteria*. A coverage criterion specifies a set of *test requirements* or *test objectives*, which should be fulfilled by the *test suite* (i.e., the set of test-cases). Typical requirements include for example covering all statements (statement coverage) or all branches (decision coverage) in the source or compiled code. Code coverage criteria present two advantages. Firstly, the obtained coverage can be quantified. Secondly, code coverage criteria facilitate automated testing: they can be used to guide the selection of new test inputs, decide when

© Springer Nature Switzerland AG 2018
T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11247, pp. 104–120, 2018.
https://doi.org/10.1007/978-3-030-03427-6_12

testing should stop and assess the quality of a test suite. This is notably the case in *white-box* (a.k.a. *structural*) software testing, in which the tester has access to the source code—as is the case, for example, in unit testing. Tools for the generation of test input values for code coverage are often based on *program analysis and formal methods* for reasoning about the structure and semantics of the source code.

Code coverage criteria are widely used in industry. In regulated domains such as aeronautics, code coverage criteria are strict normative requirements that the tester must satisfy before delivering the software. In other domains, they are recognized as good practice for testing.

However, automatic tools for the generation of test inputs to satisfy code coverage criteria have not yet made it into widespread industrial use. This despite the maturity of the underlying technology and the promise of significant gains in time, manpower and accuracy. This reticence is probably cultural in part: an automated test process can be very different to a manual one and test engineers who are used to functional testing have to accept the idea that an automatic tool can generate test inputs to respect a code coverage criterion but cannot provide the oracle. It can no doubt also be explained by the very importance of the test process: businesses may be reluctant to conduct experiments in such a crucial part of the development cycle. Finally, we have to suppose that existing test tools do not correspond closely enough to the needs of industrial users and cannot easily be integrated into existing processes.

This is the gap which has to be closed in order for automatic structural testing tools to be used in an industrial setting and this paper describes how one such tool is currently being integrated into industrial practice thanks to a successful experience of collaboration between academia and industry. The present work was done in collaboration between CEA List, a research institute, and MERCE, a research center of Mitsubishi Electric. First, we describe the functionality of the main components of the tool, resulting from several years of academic research and selected by the industrial user as being the most appropriate for its needs. Then we describe the crucial role played by the research branch of the industrial user in refining the definition of the needed functionality and building the interface between the tool and the end users in the business unit. Finally, we present the benefits of the proposed solution and provide some lessons learnt from this experience.

2 Overview of the Tool Architecture

The structure of the complete business-oriented test solution is illustrated by Fig. 1. The generic test generation tool PathCrawler/LTest provided by the CEA List institute contains three main ingredients. A concolic testing tool, PathCrawler, is used to generate test-cases for a given C program. The generation of concrete test inputs for a given program path relies on a constraint solver, Colibri. The specification mechanism of *labels* and a specific label-oriented strategy allow an efficient support of a desired test coverage criterion expressed as labels.

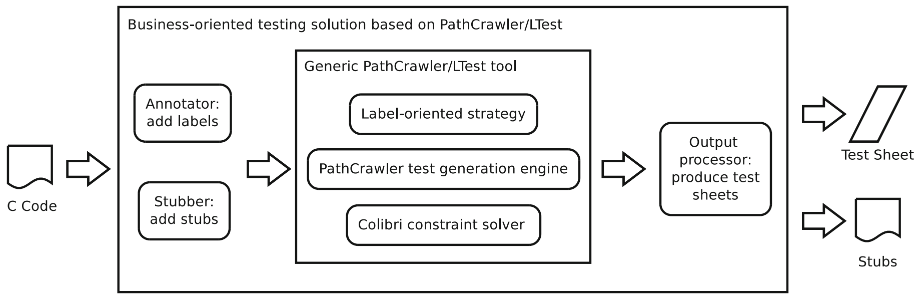

Business-oriented testing solution based on PathCrawler/LTest
Generic PathCrawler/LTest tool
Annotator: add labels
Stubber: add stubs
Label-oriented strategy
PathCrawler test generation engine
Colibri constraint solver
Output processor: produce test sheets
C Code
Test Sheet
Stubs

Fig. 1. Tool architecture

To adapt PathCrawler/LTest for a specific industrial context, additional modules were developed by MERCE, the research branch of the industrial partner. They include Annotator (that expresses the specific target criterion in terms of labels), Stubber (that produces necessary stubs) and Output processor (that creates the required test reports).

The paper is organized as follows. First, Sect. 3 presents the PathCrawler testing tool and its main features. Then, Sect. 4 presents the Colibri constraint solver used by the considered testing tool. Next, Sect. 5 introduces the notion of labels, a recent specification mechanism for coverage criteria, and describes their benefits. Section 6 presents the support of labels in the LTest toolset developed on top of PathCrawler. The ongoing adoption of PathCrawler/LTest by an industrial partner is described in Sect. 7. Finally, Sect. 8 provides a conclusion and future work.

3 PathCrawler Test Generation Tool

PathCrawler [10,32] is a test generation tool for C programs which was initially designed to automate structural unit testing by generating test inputs for full structural coverage of the C function under test.

PathCrawler has been developed at CEA List since 2002. Over the years it has been extended to treat a larger subset of C programs and applied to many different verification problems, most often on embedded software [14,28,33,35]. In 2010, it was made publicly available as an online test server [1], for evaluation and use in teaching [19].

PathCrawler is based on a method [32] which was subsequently baptized *concolic* or *Dynamic Symbolic Execution* [11,31], i.e. it performs symbolic execution along a concrete execution path. The user provides the compilable files containing the complete ANSI C source code of the function under test, f, and all other functions which may be directly or indirectly called by f. He also selects the coverage criterion and any limit on the number of loop iterations in the covered paths as well as an optional precondition to define the test context. He may finally provide an oracle in the form of C code or annotate the code with assertions. Test generation is then carried out in two major phases.

In the first phase, PathCrawler extracts the inputs of f and creates a *test harness* used to execute f on a given test-case. The test harness is basically an instrumented version of the code that outputs a trace of the path covered by each test-case. The extracted inputs include the formal parameters of f and the non-constant global variables used by f. Each test-case will provide a value for each of these inputs. This phase uses the Frama-C platform [18], developed at CEA List.

The second phase generates test inputs to respect the selected coverage criterion. This phase is based on symbolic execution, which generates constraints on symbolic input values, and constraint solving to find the solution, in the form of new concrete input values, to a new set of constraints. Indeed, symbolic execution is used to analyse the trace of the execution path followed when the harness executes f on the concrete input values of each generated test-case, and produce the *path predicate* defining the input variables which cause that path to be covered.

PathCrawler differs in two main ways from other tools based on this concrete/symbolic combination.

Like other tools, PathCrawler runs the program under test on each test-case in order to recover a trace of the execution path. However, in PathCrawler's case actual execution is chosen over symbolic execution merely for reasons of efficiency and to demonstrate that the test does indeed activate the intended execution path. Unlike tools designed mainly for bug-finding, PathCrawler does not use actual execution to recover the concrete results of calculations that it cannot treat. This is because these results can only provide an incomplete model of the program's semantics and PathCrawler aims for complete coverage of a certain class of programs rather than for incomplete coverage of any program.

Indeed, even with incomplete coverage many bugs can often be detected, but PathCrawler was designed for use in more formal verification processes where coverage must be quantified and justified so that and it can also be used in combination with static analysis techniques [12,29]. If a branch or path is not covered by a test, then unreachableness of the branch or infeasibility of the path must be demonstrated. Soundness and completeness are necessary for 100% satisfaction of a coverage criterion. Test-case generation is *sound* when each test-case covers the test objective for which it was generated, and *complete* when absence of a test-case for some test objective means this objective is infeasible or unreachable.

The soundness of the PathCrawler method is verified by concrete execution of generated test-cases on the instrumented version of the program under test. The trace obtained by the concrete execution of a test-case confirms that this test-case really executes the path for which it was generated.

Completeness can only be guaranteed when the objectives can all be covered by a reasonable number of test-cases, symbolic execution correctly represents the semantics of C and constraint solving (which is combinatorially hard) always terminates in a reasonable time. Note that completeness and the verification of soundness on the instrumented code actually require symbolic execution of program features to be adapted to the target platform (compiler optimisations, libraries, floating-point unit, etc.) and also PathCrawler's execution of the tests on the instrumented code to be carried out in the same environment. PathCrawler is currently only adapted to a Linux development environment and Intel-based platform. The search strategy of the PathCrawler method ensures iteration over all feasible paths of the program, which is necessary for completeness, for all terminating programs with finitely many paths. Programs containing infinite loops cannot be tested in any case in the way we describe here, as the execution of the program on the test inputs would never terminate. Any infinite loop which has been introduced as the result of a bug can only be detected by a timeout on the execution of each test-case on the instrumented code. Terminating programs with an infinite number of paths must have an infinite number of inputs and this is another class of programs that cannot be tested using the PathCrawler method.

The second main difference between PathCrawler and other similar tools is that PathCrawler is based not on a linear arithmetic or SMT solver but on the finite domain constraint solver Colibri, also developed at CEA List (see Sect. 4). PathCrawler and Colibri are both implemented in Constraint Logic Programming, which facilitates low-level control of constraint resolution and the development of specialized constraints, as well as providing an efficient backtracking mechanism. Within PathCrawler, specialized constraints have been developed to treat bit operations, casts, arrays with any number of variable dimensions and array accesses using variable index values. The attempt to correctly treat all C instructions is ongoing but PathCrawler can already treat a large class of C programs.

PathCrawler outputs detailed results in the form of XML files. These include overall statistics on the test session, including results in terms of coverage, whether the session ended normally or timed out or crashed and start and end times. For each test-case, the input values, result (according to the user's oracle or assertions, if provided, or run-time error, timeout or detection of an unitialised variable), covered path and concrete output values are provided. The result is either the verdict according to the user's oracle or assertions, if provided, or maybe a run-time error, timeout or detection of an unitialised variable. The symbolic (i.e. expressed as a formula over input variables) output values are also given. Moreover, for each path prefix which could not be covered, the reason is given: demonstration of infeasability, constraint resolution timeout, limit on the number of loop iterations, or untreated C language construction. The predicate on the input variables of each covered path and uncovered prefix is also given. In the case of path prefixes found to be infeasible, the predicate can be used to explain the infeasibility to the user and in the case of constraint resolution timeout, it can be used to determine manually the feasability of the path.

4 Colibri Constraint Solver

Constraint solving techniques are widely recognized as a powerful tool for Validation and Verification activities such as test data generation or counter-example generation from a formal model [23], program source code [15,16] or binary code [7]. A constraint solver maintains a list of posted constraints (*constraint store*) over a set of variables and a set of allowed values (*domain*) for each variable, and provides facilities for constraint propagation (*filtering*) and for instantiation of variables (*labeling*) in order to find a solution.

In this section we present the Colibri library (COnstraint LIBrary for veRIfication) developed at CEA List since 2000 and used inside the PathCrawler tool for test data generation purposes. The variety of types and constraints provided by Colibri makes it possible to use it in other testing tools at CEA List like GATeL [23], for model based testing from Lustre/SCADE, and Osmose [7], for structural testing from binary code.

General Presentation. Colibri provides basic constraints for arithmetic operations and comparisons of various numeric types (integers, reals and floats). Cast constraints are available for cast operations between these types. Colibri also provides basic procedures to instantiate variables in their domains making it possible to design different instantiation strategies (or *labeling procedures*). These implement specific heuristics to determine the way the variables should be instantiated during constraint resolution (e.g. a particular order of instantiation) and the choice of values inside their domain (e.g. trying boundary or middle values first). Thus the three aforementioned testing tools have designed their own labeling procedures on the basis of Colibri primitives.

The domains of numerical variables are represented by unions of disjoint intervals with finite bounds: integer bounds for integers; double float bounds for reals; and double/simple float bounds, infinities or NaNs for double/simple floating point formats. These unions of intervals make it possible to accurately handle domain differences. For each numeric type and each basic unary/binary operation or comparison, Colibri provides the corresponding constraint.

Moreover, for each arithmetic operation, additional filtering rules apply algebraic simplifications, which are very similar for integer and real arithmetics, whereas floating arithmetics uses specific rules.

Bounded and Modular Integer Arithmetics. Colibri provides two kinds of arithmetics for integers: bounded arithmetics for ideal finite integers and modular arithmetics for signed/unsigned computer integers.

Bounded arithmetics is implemented with classical filtering rules for integer interval arithmetics. These rules are managed in the projection functions of each arithmetic constraint. Moreover, a congruence domain is associated to each integer variable. Filtering rules handle these congruences in order to compute new ones and maintain the consistency of interval bounds with congruences (as in [20]). The congruences are introduced by multiplications by a constant and propagated in the projection functions of each arithmetic constraint.

Modular arithmetics constraints are implemented by a combination of bounded arithmetics constraints with modulus constraints as detailed in [17]. Thus they benefit from the mechanisms provided for bounded integer arithmetics. Notice that using unions of disjoint intervals for the domain representation makes it possible to precisely represent the domain of signed/unsigned integers.

Real and Floating Point Arithmetics. Real arithmetics is implemented with classical filtering rules for real interval arithmetics where interval bounds are double floats. In real interval arithmetics each projection function is computed using different rounding modes for the lower and the upper bounds of the resulting intervals. The lower bound is computed by rounding downward, towards $-1.0\mathit{Inf}$ (i.e. $-\infty$), while the upper bound is computed by rounding upward, towards $+1.0\mathit{Inf}$ (i.e. $+\infty$). This enlarging ensures that the resulting interval of each projection function is the smallest interval of doubles including all real solutions.

Floating point arithmetics is implemented with a specific interval arithmetics as introduced by Michel in [26]. Notice that properties like associativity or distributivity do not hold in floating point calculus. The projection functions in this arithmetics have to take into account *absorption* and *cancellation* phenomena specific to floating point computations. These phenomena are handled by specific filtering rules allowing to further reduce the domains of floating point variables. For example, the constraint $A +_F X = A$ over floating point numbers means that X is absorbed by A. The minimal absolute value in the domain of X can be used to eliminate all the values in the domain of A that do not absorb this minimum. Thus, in double precision with the default rounding mode (called *nearest to even*), for $X = 1.0$ the domain of A is strongly reduced to the union of two interval of values that can absorb X:

$$[\mathit{MinDouble} \,..\, -9007199254740996.0,\ 9007199254740992.0 \,..\, \mathit{MaxDouble}].$$

Colibri uses very general and powerful filtering rules for addition and subtraction operations as described in [24]. For example, for the constraint $A + B = 1.0$ in double precision with the *nearest to even* rounding mode, such filtering rules converge to the same interval for A and B

$$[-9007199254740991.0 \,..\, 9007199254740992.0].$$

Implementation Details. Colibri is implemented in ECLiPSe Prolog [30]. Its suspensions, generic unification and meta-term mechanisms make it possible to easily design new abstract domains and associated constraints. Incremental constraint posting with on-the-fly filtering and automatic backtracking to a previous constraint state provided by Colibri are important benefits for search-based state exploration tools, and in particular, for test generation tools.

To conclude this short presentation of Colibri, let us remark that the accuracy of its implementation relies a lot on the use of unions of intervals and the combination of abstract domain filtering rules with algebraic simplifications. Experiments in [4,9,13] using SMT-LIB benchmarks show that Colibri can be competitive with powerful SMT solvers. In 2017 and 2018, Colibri was the winner of the floating point category at the 12th and 13th International Satisfiability Modulo Theories Competitions (SMT-COMP 2017 and 2018).

5 Generic Specification of Coverage Criteria with Labels

In 2014, a previous paper introduced *labels* [8], a code annotation language to encode concrete test objectives, and showed that several common coverage criteria can be simulated by label coverage. In other words, given a program P and a coverage criterion $\mathbf{C}$, the concrete test objectives instantiated from $\mathbf{C}$ for P can always be encoded using labels. In this section, we recall some basic results about labels.

Labels. Given a program P, a *label* ℓ is a pair (loc, φ) where loc is a location of 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 given in Fig. 2.

```
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;
```

Fig. 2. Examples of labels

We say that a test datum t *covers a label* $\ell = (loc, \varphi)$ in P, denoted $t \overset{\mathrm{L}}{\rightsquigarrow}_P \ell$, 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 Fig. 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.

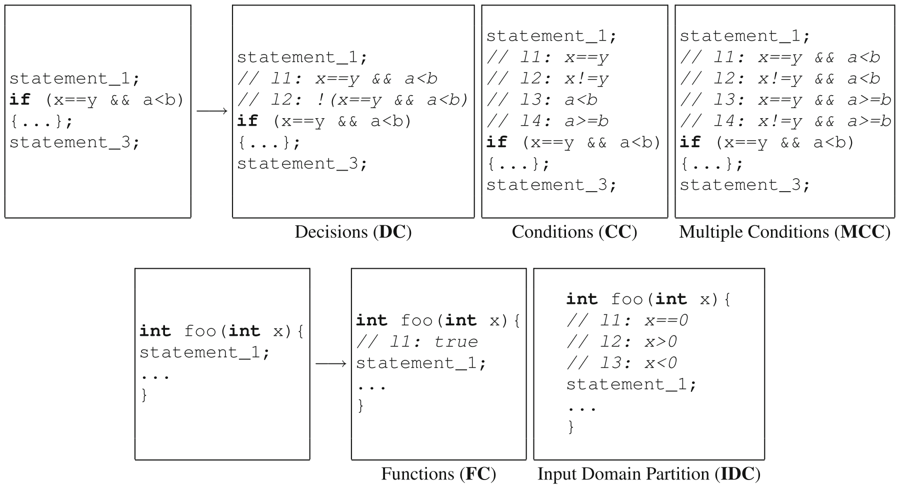
An *annotated program* is a pair $\langle P, L\rangle$ where P is a program and L is a set of labels defined in P. Figure 2 shows an example of an annotated program with four labels.

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-case t in TS such that $t \overset{\text{L}}{\rightsquigarrow}_P \ell$. This is denoted $TS \overset{\text{L}}{\rightsquigarrow}_{\langle P,L\rangle}$ **LC**.

Criterion Encoding. 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 test objectives instantiated from **C** for P. We call *annotation* (or *labeling*) *function* such a procedure automatically adding test objectives to a given program for a given coverage criterion.

It is shown in [8] that label coverage can notably simulate basic-block coverage (**BBC**), branch coverage (**BC**) and decision coverage (**DC**), function coverage (**FC**), condition coverage (**CC**), decision-condition coverage (**DCC**), multiple condition coverage (**MCC**), **GACC** [2], as well as the side-effect-free fragment of weak mutations (**WM'**) in which the considered mutation operators are not allowed to introduce side-effects. 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 other, more specific criteria.

Figure 3 illustrates the simulation of some common criteria with labels on sample code. The resulting annotated code is automatically produced by the corresponding annotation functions. 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. The encoding of **GACC** (General Active Clause Coverage) [2] is shown in Fig. 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 Fig. 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 first clause would indeed change the decision.



```
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;
...
}
```

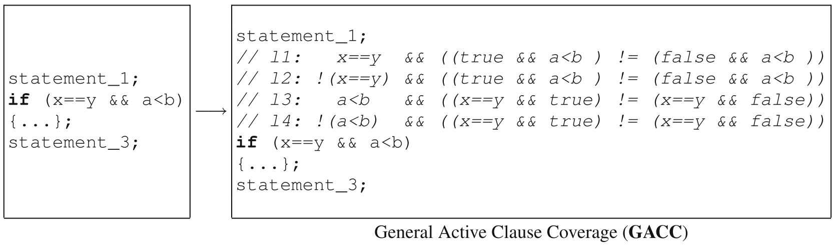
```
int foo(int x){
// l1: true
statement_1;
...
}
```

Functions (**FC**)

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

Input Domain Partition (**IDC**)

Fig. 3. Simulating standard coverage criteria with labels



```
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**)

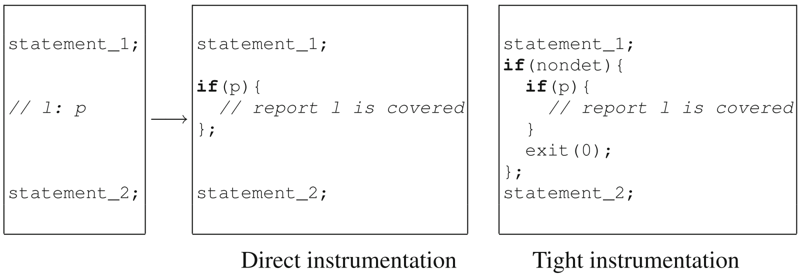
Fig. 4. Simulating the GACC coverage criterion with labels

6 Efficient Test-Case Generation for Labels in LTest

Labels appear to be not only convenient to express various testing criteria, but also amenable to efficient support in various testing tasks. Previous efforts [6,8,21] showed that labels can be 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 [5]. In this section, we detail the label-oriented strategy for test-case generation used in the PathCrawler/LTest tool and implemented in top of PathCrawler.

The label-oriented 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 Dynamic Symbolic Execution (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. Fig. 5). In direct instrumentation the



```
statement_1;

// l: p

statement_2;
```

```
statement_1;

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

statement_2;
```

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

Fig. 5. Two ways to instrument a label: direct and tight instrumentation

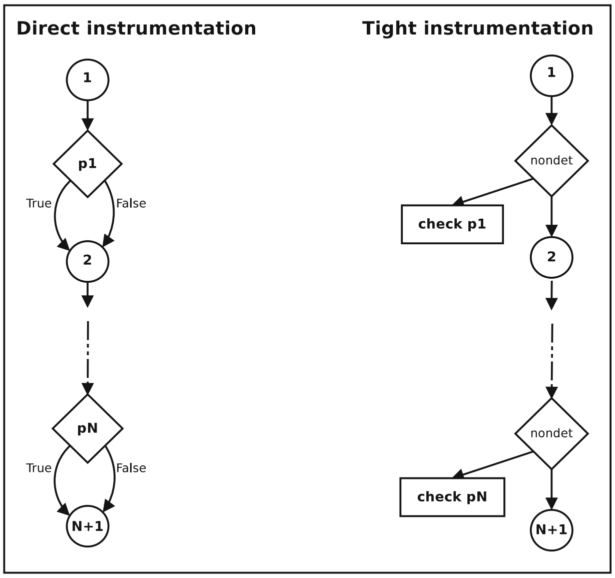


Fig. 6. Comparison of direct and tight instrumentation for a sequence of N labels

label is replaced by a conditional statement that checks the label predicate p and reports that the label is covered whenever the predicate is satisfied. In tight instrumentation, the conditional statement is reached only when a nondeterministic operation `nondet` returns true. Moreover, the execution exits after the evaluation of the label predicate, whenever it is true or not. Note that any DSE engine can simulate non-deterministic choices via an additional input array of (symbolic) boolean values.

In the resulting instrumented program, direct instrumentation leads to creating two paths[1] for each path in the non-instrumented program, while tight instrumentation makes DSE consider only one additional program path each

[1] And sometimes even more, if the label was inside a loop or a function called several times.

time a label is traversed. This situation is schematically illustrated for a sequence of N labels in Fig. 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 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 previously traversed labels. Thus, tight instrumentation yields only a linear growth of the path space without any complexification of path predicates.

The main idea behind iterative label deletion is to ignore a label that has been covered while continuing the test generation session. It can be easily implemented by introducing a status for each label and considering that the `nondet` operation never returns true for an already covered label. The label-oriented test generation strategy is further detailed in [8].

7 Ongoing Adoption of PathCrawler/LTest in an Industrial Setting

Mitsubishi Electric is a global group having a wide range of activities from Home Products to Space Systems including Automotive Equipment, Transportation Systems, Energy Systems and many others. A lot of those products are software intensive, are developed in C language and are safety critical, like train control systems or automotive components. They thus require a high quality level, typically meeting railway EN 50128 SIL4 or automotive ISO 26262 ASIL D certification criteria. To reach such quality, extensive and diverse testing is needed. This testing is very costly, due to the effort needed to reach such very stringent testing criteria: design adequate test sheets satisfying the criteria with adequate test-cases, fill inputs and expected outputs of those test-cases, apply those test sheets on the developed code, compare actual and expected outputs, determine actual coverage, compare actual results to expected one, determine missing coverage and rework the test sheets and the code accordingly. On a typical safety critical software, 65% of the cost is due to testing and associated rework.

Mitsubishi Electric R&D Centre Europe (MERCE) is the advanced European research laboratory of Mitsubishi Electric group. From MERCE knowledge of business unit test process, MERCE identified that PathCrawler/LTest could accelerate it. More specifically given a C source code as input, PathCrawler/LTest can automatically produce a set of test-cases satisfying a coverage criterion, thus opening the door to automatic structural test generation. The only manual step is to encode as labels the coverage criterion through annotation on the tested source code. MERCE knows that to be usable by engineers, a new technology should be as automated and as integrated as possible within the existing development process. Thus MERCE decided to focus on unit testing which seems amenable to full automation. Therefore, the question studied by MERCE was simple: is it possible to design a fully automatic structural unit test generation tool that can be easily integrated into the current development process used in the business unit?

To answer this question MERCE started to evaluate PathCrawler/LTest technology, first on a few examples provided by the business unit. MERCE manually encoded a business unit coverage criterion by adding labels on the source code samples, a few functions ranging from a few hundreds to one thousand lines of code. PathCrawler/LTest was able to successfully cover all the labels, in a few seconds for small functions to a few tens of minutes for the biggest one having 2^{145} paths[2]. One interesting outcome of PathCrawler technology is that it is possible to determine when a test objective (i.e. a label) is *impossible* to cover due to the structure of the code, which seemed a quite important feedback to give to the tester and potentially a crucial information to be used in a certification process.

From this first very positive step, MERCE decided to start the design and the implementation of the desired test generation solution. This solution works as follows (cf. Fig. 1): take as input the original, unmodified source code, automatically add labels satisfying the business unit coverage criterion through Annotator, automatically produce stubs suitable for unit testing through Stubber, find actual test-cases using PathCrawler/LTest, process its output in Output processor to produce final test sheets in Excel and CSV (Comma Separated Value) formats for human and machine use in the remaining part of the test process. MERCE developed an OCaml plug-in of about 1,500 lines within Frama-C to do the annotation and stub generation parts, reusing Frama-C capabilities to parse and modify C source code. An additional program of 2,000 lines was also written to coordinate the call to the annotation plug-in, the call to PathCrawler, the parsing of PathCrawler's output and production of ready-to-use test sheets.

MERCE conducted experiments with this new tool on real industrial code. This code is about 80,000 lines of C code (without headers), making about 1,300 functions to unit test distributed over about 150 files. The MERCE tool was able to parse and annotate 100% of the files, and to successfully apply PathCrawler/LTest for generating test sheets for 86% of the functions and covering about 14,000 test objectives, of which 17% are structurally impossible test objectives. The total test generation time is about 8 h on a regular PC, i.e. less than a day, taking on average 26 s per function. MERCE roughly estimated the total manual generation of those tests to 230 work days[3], therefore bringing an effective benefit factor of more that 230 for test input generation. Those very good results are very encouraging for pushing the technology in business units.

Developing such a tool requested a non negligible engineering effort from MERCE. Despite Frama-C providing all the needed framework, understanding and applying the Frama-C toolbox, moreover in the non mainstream OCaml

[2] Recall that path exploration stops as soon as all labels are covered.

[3] This time does not include the time to elaborate an oracle whose elaboration remains manual.

language, took some time. On the benefit side, PathCrawler provides all the information needed to produce the test sheets and thus creating them was relatively easy.

8 Conclusion and Future Work

We have described an example of how to transfer new technology based on formal methods to industrial use. PathCrawler is a mature test generation tool and *labels* offer an easy way to adapt it to the user's own code coverage criterion.

Several lessons can be learned from this experience. First of all, this work demonstrates that *a close collaboration between the tool developers and the industrial user* is vital. Having an efficient tool developed by a research laboratory is necessary, but often not sufficient for its integration into an industrial testing process. The role of MERCE in adapting the tool to the specific needs of the business units has been crucial.

Changing habits in an industrial process is always difficult and that is why, when trying to industrialize PathCrawler/LTest technology, MERCE focused on *a fully automated tool* that would integrate well in the current testing process. Of course such automation is done at the expense of richer functionalities: in this case MERCE focused on unit testing (while PathCrawler/LTest could probably handle more elaborate testing). And beyond the technological core, there is still a lot of mundane integration work to adapt the tool to the real process (e.g. with other tools or legacy test material) and let testers be at ease with it.

An important factor is related to *the completeness of the tool*, or its capacity to justify the absence of a test input for a given test objective. This feature can be particularly appreciated in an automated testing process since it can be very difficult (or even impossible) to achieve manually. *Soundness and completeness of the tool* are also particularly important in the context of certification. They help to rigorously justify the coverage of each test-case and the whole test suite, and to provide the certification authority with a proof of best-achievable coverage.

The performance of the tool is another crucial factor for its integration. While current speed of PathCrawler/LTest has already shown an astonishing possible increase in productivity (a factor of 230 on a real-life example), having even higher performance would allow interactive use and direct integration into developers' IDE, thus allowing even greater productivity by merging the testing phase into the development phase.

Regarding future work, extension of PathCrawler/LTest to currently unhandled coverage criteria (like MCDC) is certainly a strong requirement as those criteria are requested by standards like ISO 26262. Other examples of test criteria of interest are related to rigorous boundary testing and coverage of function outputs. Efficient support of hyperlabels [22], a recent generalization of labels to a larger class of criteria, is another future work direction.

References

1. The PathCrawler online test generation service (2010–2018). http://pathcrawler-online.com/
2. Ammann, P., Offutt, A.J., Huang, H.: Coverage criteria for logical expressions. In: Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE 2003), pp. 99–107 (2003)
3. Ammann, P., Offutt, J.: Introduction to Software Testing, 1st edn. Cambridge University Press, Cambridge (2008)
4. Bardin, S., Herrmann, P., Perroud, F.: An alternative to SAT-based approaches for bit-vectors. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 84–98. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_7
5. Bardin, S., Chebaro, O., Delahaye, M., Kosmatov, N.: An all-in-one toolkit for automated white-box testing. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570, pp. 53–60. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09099-3_4
6. Bardin, S., et al.: Sound and quasi-complete detection of infeasible test requirements. In: Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015), pp. 1–10. IEEE (2015)
7. Bardin, S., Herrmann, P.: Structural testing of executables. In: Proceedings of the First International Conference on Software Testing, Verification, and Validation (ICST 2008), pp. 22–31. IEEE (2008)
8. Bardin, S., Kosmatov, N., Cheynier, F.: Efficient leveraging of symbolic execution to advanced coverage criteria. In: Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014), pp. 173–182. IEEE (2014)
9. Bobot, F., Chihani, Z., Marre, B.: Real behavior of floating point. In: Proceedings of the 15th International Workshop on Satisfiability Modulo Theories (SMT 2017), Part of CAV 2017 (2017)
10. Botella, B., et al.: Automating structural testing of C programs: experience with PathCrawler. In: Proceedings of the 4th International Workshop on the Automation of Software Test (AST 2009), Part of the 31st International Conference on Software Engineering (ICSE 2009), pp. 70–78. IEEE (2009)
11. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), pp. 209–224. USENIX Association (2008)
12. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, Software Verification and Testing Track (SAC-SVT 2012), pp. 1284–1291. ACM (2012)
13. Chihani, Z., Marre, B., Bobot, F., Bardin, S.: Sharpening constraint programming approaches for bit-vector theory. In: Salvagnin, D., Lombardi, M. (eds.) CPAIOR 2017. LNCS, vol. 10335, pp. 3–20. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59776-8_1
14. Dierkes, M., Faivre, A., Le Guen, H., Williams, N.: Completion of test models based on code analysis. In: Proceedings of the Conference on Embedded Real Time Software and Systems (ERTS2 2014) (2014)

15. Gotlieb, A.: Euclide: a constraint-based testing platform for critical C programs. In: Proceedings of the Second International Conference on Software Testing Verification and Validation (ICST 2009), pp. 151–160. IEEE (2009)
16. Gotlieb, A., Botella, B., Watel, M.: INKA: ten years after the first ideas. In: Proceedings of the the International Conference on Software and Systems Engineering and their Applications (ICSSEA 2006) (2006)
17. Gotlieb, A., Leconte, M., Marre, B.: Constraint solving on modular integers. In: Proceedings of the Workshop on Constraint Modelling and Reformulation (ModRef 2010), Part of CP 2010 (2010)
18. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Asp. Comput. **27**(3), 573–609 (2015)
19. Kosmatov, N., Williams, N., Botella, B., Roger, M.: Structural unit testing as a service with pathcrawler-online.com. In: Proceedings of the 7th IEEE International Symposium on Service-Oriented System Engineering (SOSE 2013), pp. 435–440. IEEE (2013)
20. Leconte, M., Berstel, B.: Extending a CP solver with congruences as domains for software verification. In: Proceedings of the Workshop on Constraints in Software Testing, Verification and Analysis (CSTVA 2006), Part of CP 2006 (2006)
21. Marcozzi, M., Bardin, S., Kosmatov, N., Papadakis, M., Prevosto, V., Correnson, L.: Time to clean your test objectives. In: Proceedings of the 40th International Conference on Software Engineering (ICSE 2018), pp. 456–467. ACM (2018)
22. Marcozzi, M., Delahaye, M., Bardin, S., Kosmatov, N., Prevosto, V.: Generic and effective specification of structural test objectives. In: Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST 2017), pp. 436–441. IEEE (2017)
23. Marre, B., Blanc, B.: Test selection strategies for Lustre descriptions in GATeL. Electron. Notes Theor. Comput. Sci. **111**, 93–111 (2005)
24. Marre, B., Michel, C.: Improving the floating point addition and subtraction constraints. In: Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 360–367. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15396-9_30
25. Mathur, A.P.: Foundations of Software Testing. Addison-Wesley Prof (2008)
26. Michel, C.: Exact projection functions for floating point number constraints. In: Proceedings of the 7th International Symposium on Artificial Intelligence and Mathematics (AIMA 2002) (2002)
27. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing, 3rd edn. Wiley, Hoboken (2011)
28. Park, J., Pajic, M., Lee, I., Sokolsky, O.: Scalable verification of linear controller software. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 662–679. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_43
29. Petiot, G., Kosmatov, N., Giorgetti, A., Julliand, J.: How test generation helps software specification and deductive verification in Frama-C. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570, pp. 204–211. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09099-3_16
30. Schimpf, J., Shen, K.: ECLiPSe - from LP to CLP. Theory Pract. Log. Program. **12**(1–2), 127–156 (2011)
31. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005), pp. 263–272. ACM (2005)

32. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: Dal Cin, M., Kaâniche, M., Pataricza, A. (eds.) EDCC 2005. LNCS, vol. 3463, pp. 281–292. Springer, Heidelberg (2005). https://doi.org/10.1007/11408901_21
33. Williams, N., Roger, M.: Test generation strategies to measure worst-case execution time. In: Proceedings of the 4th International Workshop on Automation of Software Test (AST 2009), pp. 88–96 (2009)
34. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. **29**(4), 366–427 (1997)
35. Zutshi, A., Sankaranarayanan, S., Deshmukh, J.V., Jin, X.: Symbolic-numeric reachability analysis of closed-loop control software. In: Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control (HSCC 2016), pp. 135–144 (2016)