Certified Development Tools Implementation in Objective Caml

Bruno Pagano[1], Olivier Andrieu[1], Benjamin Canou[2,3], Emmanuel Chailloux[3], Jean-Louis Colaço[4,⋆], Thomas Moniot[1], and Philippe Wang[3]

[1] Esterel Technologies, 8, rue Blaise Pascal, 78890 Elancourt, France
{Bruno.Pagano,Olivier.Andrieu,Thomas.Moniot}@esterel-technologies.com
[2] ENS Cachan, antenne de Bretagne Campus Ker Lann, F-35170 Bruz, France
Benjamin.Canou@eleves.bretagne.ens-cachan.fr
[3] Laboratoire d'informatique de Paris 6 (LIP6 - UMR 7606),
Université Pierre et Marie Curie, Paris 6,
104, avenue du Président Kennedy, 75016 Paris, France
{Emmanuel.Chailloux,Philippe.Wang}@lip6.fr
[4] Siemens VDO Automotive, 1, avenue Paul Ourliac, BP 1149, 31036 Toulouse, France
Jean-Louis.Colaco@siemens.com

Abstract. This paper presents our feedback from the study on the use of Objective Caml for safety-critical software development tools implementation. As a result, Objective Caml is now used for the new Scade™ certified embedded-code generator. The requirements for tools implementation are less strict than those for the embedded code itself. However, they are still quite demanding and linked to imperative languages properties, which are usually used for this kind of development. The use of Objective Caml is outstanding: firstly for its high level features (functional language of higher order, parametric polymorphism, pattern matching), secondly for its low level mechanisms needed by the runtime system (GC, exceptions). In order to develop the tools to check the safety-critical software development rules, it is necessary to reinterpret them for this language, and then to adapt Objective Caml so that it satisfies them. Thus, we propose a language restriction and a simplified runtime library in order that we can define and measure the coverage of a program written in Objective Caml according to the MC/DC criteria. Then we can look forward to seeing this kind of languages spread out the industrial environment, while raising the abstraction level in the conception and implementation of tools for certified programs production.

Keywords: Code coverage, Tests measurement, Functional programming, Objective Caml, Civil avionics.

1 Introduction

Safety-critical softwares are traditionally associated to embedded control systems but some other areas need them. Standards for software development have been

⋆ This work started while the author was at Esterel-Technologies.

P. Hudak and D.S. Warren (Eds.): PADL 2008, LNCS 4902, pp. 2–17, 2008.

defined with levels determined from the safety assessment process and hazard analysis by examining the effects, on the final users, of a failure condition in the system. Among the most common applications, we hold up as examples flight commands, railway traffic lights, the control system of a nuclear power plant, but also medical equipment or a car ABS[1]. They share the particularity that their dysfunctions can cause catastrophes with lethal consequences for those in relation with such a system.

The civil avionics authorities defined a couple of decades ago the certification requirements for aircraft embedded code. The DO-178B standard [17] defines all the constraints ruling the aircraft software development. This procedure is included in the global certification process of an aircraft, and declines now for other industrial sectors concerned by critical software (FDA Class III for medical industry, IEC 61508 for car industry, etc).

The DO-178B standard imposes a very precise development process, which preponderant activity is independent verification of each development step. In this paper, we focus on software development and mistakes hunting procedures, whereas DO-178B's scope goes further. Code development as it is recognised by certification authorities follows the traditional V-Model dear to the software engineering industry. Constraints are reinforced but the principles stay the same: the product specifications are written by successive refinements, from high level requirements to design and then implementation. Each step owns an independent verification activity, which must provide a complete traceability of the requirements appearing in this step.

The followed process to realize embedded code satisfying such a certification requires the definition of a "coding standard". This standard must define a set of strict rules for the specifications' definition, for the implementation and for the traceability between specifications and realizations. In particular, the coding standard must put forward the obligation to cover the entire code. The DO-178B certification imposes this coverage to be done according to the MC/DC [10] measure (Modified Condition/Decision Coverage).

The DO-178B standard applies to embedded code development tools with the same criteria as the code itself. This means that the tool development must follow its own coding standard. The certification standard originally targeted only embedded software, so its application for a development tool must be adapted. For instance, for a code generator it is accepted to use dynamic allocation and have recursive functions. The specificity of the certification process for tools is under discussion to be explicitly addressed by the forthcoming DO-178C standard that will be effective in a few years.

In this context, tools development in a particular language must comply with DO-178B constraints, which means having an MC/DC coverage of the program's source. Likewise, the runtime library, if there is one, must be certified. For the C language, this translates to the control of `libc` calls and compiler mechanisms verification. For more modern languages, such as Java, it would demand the certification of the whole library.

[1] Anti-lock Braking System.

Objective Caml (OCaml) is particularly suitable for compiler conception and formal analysis tools. As well as itself [13], it is used in Lucid Synchrone [15], the à la Lustre language for reactive systems implementation, or the Coq [16] proof assistant implementation. Even ten years ago, the use of the OCaml language in software engineering for safe real-time programs development interested some major avionics industries (Dassault). The experience of Surlog with AGFL shows that OCaml can be integrated in a critical software development process and that it brings in its well-founded model. With Astrée [8], OCaml proves its adequacy for critical software tools realization.

The Esterel-Technologies company markets Scade [2,4], a model-based development environment dedicated to safety-critical embedded software. The code generator of this suite that translates models into embedded C code is DO-178B compliant and allows to shorten the certification process of avionics projects which make use of it. The first release of the compiler was implemented in C and was available in 1999 (version 3.1 to 5.1 were based on this technology), but since 2001, Esterel-Technologies has prepared its next generation code generator based on a prototype written in OCaml. This work allowed to test new compiling techniques [7] and language extensions [6]. It has now appeared that OCaml allowed to reduce the distance between the specifications and the implementation of the tool, to have a better traceability between a formal description of the input language and its compiler implementation.

In a more classical industrial environment, where C or Ada languages dominate, and where development processes use intensive tests, the introduction of OCaml changes the formulations of qualification problematics. Many of its language features surprise control theory engineers or imperative languages programmers, first because OCaml is an expression language, but also because it provides higher level features such as parametric polymorphism, pattern matching, exception handling and automatic memory management [11] (*Garbage Collector or GC*).

Conversely, code coverage and conditions/decisions notions are defined and well understood for imperative languages like the C language. So we need to adapt this notion to OCaml Boolean expressions. Functional programming and parametric polymorphism are the main concerns in this evolution of MC/DC code coverage specification. It is also necessary to adapt the runtime library to fit the coding standards, and advisable to bring control points between specifications and runtime library for the control (general apply mechanism, exceptions handling) and for automatic memory management. This makes the use of the Inria original language runtime library difficult and militates for the building of an alternate compatible runtime library.

The rest of this paper is organized as follows. Section 2 exposes the validation process in an industrial context. Section 3 explains the adaptation of code coverage for OCaml programs and describes our implementation called *mlcov*. Section 4 shows how to certify OCaml programs and then details how the runtime library must be modified. Section 5 compares different approaches to use OCaml and presents our future work.

2 Code Verification in a Certified Context

The American federal aviation administration (FAA) requires any computer program embedded in an aircraft to respect the DO-178B standard to allow it to fly. Several levels of criticity are defined (A, B, C, etc.) and applied according to the impact of a software bug on the whole system and passengers. For instance comfort application like entertainment or air-conditioning are less critical than flying command system.

The DO-178B is highly rigorous about the development process but does not give any constraint, neither on the programming language to use nor even about the paradigms it has to implement. However rules exist to precise this standard and drastically restrain the type of accepted programs. At level A, which applies to the most critical programs, an embedded program cannot dynamically allocate memory, use recursive function and generally speaking has to work in bounded time and space with known bounds. For this kind of development, using OCaml or any other high level language is not an option. Usually, only assembly language and limited subsets of C and Ada are used.

Nevertheless, it is allowed, and becoming more and more necessary, to use code generators and/or verifiers to help writing these programs, if these tools are themselves certified at the same level of the standard. For example, the code coverage measurement, about which we will speak later, can be done by human code reviewers or by a software if it is itself certified at the appropriate level. This level is a bit relaxed for verification tools as they cannot directly affect the embedded application.

When it comes to tools development, some of the most constraining rules can consensually be broken, given that the fundamental demands are fulfilled. For example, if recursion or dynamic memory allocation are allowed, it must be restrained to memory configurations where the stack and the heap are large enough not to interfere with the ongoing computation. Even if, unlike an embedded software, a tool can fail, it must provably never give a false result. Therefore, the verification activities take a preponderant amount of time in the development process.

Tests: coverage measurement criteria: During an industrial process, the code verification stage takes place after the development and is meant to show that the product matches its specifications. Testing software requires a stoping criteria to state that the behavior of the program is reasonably explored as it is well known that exhaustivness is unreachable. Coverage measurement is the traditional answer from the software engineering community to define the good compromise between loose verification and theoretical limits. On this particular point, the DO-178B standard has a very precise requirement by demanding a complete exploration of the code structure in the MC/DC sense. The DO-178B defines several verification activities and among these a test suite has to be constituted to cover up the set of specifications of the software to verify and thus its implementation. The standard requires each part of the code to be used in the execution of at least one test and conform to its specifications.

The structural coverage criteria can be separated into the data flow ones and the control flow ones. The data flow analysis measures the relation between the

assignments and uses of variables. The DO-178B only defines criteria over the control flow. The control flow is measured on executed instructions, Boolean expressions evaluated and branches of control instruction executed. We will now present the main measurements.

- *Statement Coverage:* It is the simplest criterion, to understand as well as to apply. It consists in verifying that each instruction of the tested program is executed by at least one test. Such a criterion is considered fragile as shown in the next example.
- *Decision Coverage:* A *decision* is the Boolean expression evaluated in a test instruction to determine the branch to be executed. This coverage requires each decision to be evaluated to the two possible values during the tests, ensuring that all code branches are taken.

  The following C code example, defining the absolute function over integers, exposes the difference between these two criteria:

  ```
  int abs(int x) {int y; if (x<0) y = -x; return y;}
  ```

  A unique test with a negative value for x is sufficient to cover all the instructions, however the decision coverage needs a second one with a positive value. This little code is sufficient to prove that decision coverage can detect more incorrect programs, since with a positive value, a random value is returned by the function instead of the identity whereas such a test is not needed by the statement coverage.
- *Condition Coverage:* A *condition* is an atomic subexpression of a decision. For example, the decision `x && (y<0) || f(z)` contains the three conditions `x`, `y<0` and `f(z)`. A condition is covered if tests exist in which it is evaluated to `true` and `false`.
- *Condition/Decision Coverage:* The C/DC is the combination of the two previous criteria.
- *Modified Condition/Decision Coverage:* The MC/DC extends the C/DC criterion by requiring each condition to independently modify the decision value. In other words, for each condition c, two tests have to exist which must change the decision value while keeping the same valuations for all conditions but c.
- *Multiple Condition Coverage:* For this criterion, the tests must generate every Boolean combinations of the conditions of each decision.

Let us now illustrate these definitions by showing the tests required by each of the criteria for the test instruction `if ((a || b ) && c) { ... }`. Eight tests exist for this instruction which are the different valuations of the Boolean variables a, b and c. We shall name these valuations *test vectors* and use the notations `[TTT T]`, `[FTT T]`, `[TFT T]`, `[FFT F]`, `[TTF F]`, `[FTT F]`, `[TFT F]` and `[FFF F]`. They correspond to the truth table of the expression.

The required tests vary according to the coverage criterion:

- Statement Coverage: a unique test giving the value T to the condition is necessary

- Decision Coverage: Two tests are necessary, one of them giving T and the other F, for example `[TTT T]` et `[FTT F]`.
- Condition Coverage: Each condition has to take the two values, therefore two tests which give different valuations to every condition are sufficient, for example `[TTF F]` et `[FFT F]` (note that this example does not satisfy the decision coverage criterion).
- C/DC: As in the previous case, we must provide two tests which give different valuations to every condition but now they must give a different value to the decision too, for example `[TTT T]` et `[FFF F]`.
- MC/DC: For each condition, we must exhibit two tests in which only this condition and the decision result differ. For example, the two test vectors `[TFT T]` and `[FFT F]` show the independence of the condition a. The test vectors may be used to show the independence of more than one condition. Usually, N+1 test vectors are necessary for a decision with N conditions. For this example, the four test vectors `[TFT T]`, `[FTT T]`, `[TFF F]` and `[FFT F]` are sufficient.
- Multiple Condition Coverage : By definition, the eight vectors of the truth table detailed before has to be be provided.

The DO-178B level A certification requires the whole program code to have a 100% MC/DC measurement. The MC/DC criterion turned out to be a reasonable compromise between a too weak requirement of two tests and an unreachable one of 2^n tests.

The relevance of the MC/DC criterion has been profusely discussed [9,12]. Our aim is to show the meaning of this measurement in OCaml since it is required by the civil avionics agencies. An important point to understand is that the MC/DC analysis of the code is one element of the validation process of every development step. Therefore, even if it is possible to work around the coverage analyses by coding tricks in theory, these tricks will be rejected by the persons in charge of reviewing the code or validating the MC/DC measurement in practice.

3 Code Coverage of OCaml Programs

According to Chilenski *et al.* [10], code coverage is not a test technique: it should be considered as a measure describing the degree to which the source code of a program has been exercised. In this section, we give a definition of the MC/DC criteria from the viewpoint of OCaml programs. We restrict to the functional and imperative features of OCaml, which correspond to the subset allowed by the coding rules of the Scade to C compiler. This subset remains quite large (*cf.* paragraph 3.3), for instance, it is sufficient to compile the standard library of the OCaml distribution.

3.1 Coverage of Expressions

We need to adapt the definition of code coverage to a functional language like OCaml. With respect to imperative languages, the notion of coverage is related

to the statements of a program. Since OCaml is an expression language, we will be interested in the coverage of the expressions evaluation.

In the imperative paradigm, coverage shall pinpoint that every execution branch in the program has independently been exercised. The same is encountered in the OCaml language, since some sub-expressions (in the case of the conditional expression, for instance) may remain unevaluated.

`if (x<y) {`
`  min = f(x);`
`} else {`
`  min = f(y);`
`}`
	`let min =`
	`  if x<y`
	`  then f x`
	`  else f y`
		As well as the coverage of the C program shows which branch of the `if` control structure has been executed, coverage of the OCaml program examines which sub-expression of the `if` operator has been evaluated.

Coverage is measured by instrumenting the source code of the program. With respect to OCaml, we state that an expression has been covered as soon as its evaluation has ended. The main idea of the instrumentation algorithm is to replace each expression `expr` with `(let aux = expr in mark(); aux)`, where the variable `aux` is not free in `expr` and `mark()` is a side-effect allowing to record that this point of the program has been reached.

Some constructions of the OCaml language (such as `if then else`) may introduce several execution branches. Coverage of expressions entails to trace the evaluation of each one of the branches independently. In order to avoid overmarking, we split the instrumentation algorithm into two mutually recursive translation functions $\mathcal{F}$ and $\mathcal{G}$. Both $\mathcal{F}$ and $\mathcal{G}$ instrument the execution branches of the program, but only $\mathcal{F}$ marks the end of evaluation of expressions. Here is the definition of the instrumentation functions, together with some explanation of the interesting cases:

$$\mathcal{F}(k) = \texttt{mark(); } k \quad \text{if } k \text{ is a constant or a constant constructor}$$

Since we (statically) know that the evaluation of a constant value or constructor never fails, we can simplify the translation and write $\mathcal{F}(k) = \texttt{mark(); } k$.

$$\mathcal{F}(id) = \begin{cases} \texttt{fun } x \texttt{ -> mark(); } id\ x & \text{if } id \text{ has a functional type} \\ \texttt{mark(); } id & \text{otherwise} \end{cases}$$

Note that $\mathcal{F}$ η-expands every top-level functional value ($\mathcal{F}(id) = \texttt{fun } x \texttt{ -> mark(); } id\ x$) so that the algorithm is still type-preserving. Otherwise, it would produce weak type variables.

$$\mathcal{F}(f\ x) = \begin{cases} \texttt{mark(); } \mathcal{G}(f)\ \mathcal{G}(x) & \text{if } f \text{ does not return} \\ \mathcal{G}(f)\ \mathcal{G}(x)\texttt{; mark()} & \text{if } f\ x \text{ has type } unit \\ \texttt{let } aux = \mathcal{G}(f)\ \mathcal{G}(x) \texttt{ in mark(); } aux & \text{otherwise} \end{cases}$$

A heuristics is implemented in order not to trace the end of evaluation of a family of functions that do not return, such as `failwith` and `exit`: we look for functions with type $\forall\alpha.\tau \rightarrow \alpha$, where the polymorphic type variable α

does not appear in τ. Indeed, the application of any of those functions does not terminate, which implies that structural coverage would never be reached if they were instrumented in the usual way. Instead of the normal case, we write $\mathcal{F}(f\ x) = \texttt{mark()};\ \mathcal{G}(f)\ \mathcal{G}(x)$. Unfortunately, this heuristics suffers from both false positives and false negatives, since it may be fooled by type annotations.

As a shortcut, we define $\mathcal{F}(f\ x) = \mathcal{G}(f)\ \mathcal{G}(x);\ \texttt{mark()}$ when the type of $f\ x$ is *unit*, since the type of `mark()` is *unit* too.

$$
\begin{aligned}
\mathcal{F}(\texttt{fun}\ x\ \rightarrow\ e) &= \texttt{fun}\ x\ \rightarrow\ \mathcal{F}(e)\\
\mathcal{F}(e_1;\ e_2) &= \mathcal{G}(e_1);\ \mathcal{F}(e_2)\\
\mathcal{F}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ [\texttt{else}\ e_3]) &= \texttt{if}\ \mathcal{G}(e_1)\ \texttt{then}\ \mathcal{F}(e_2)\ [\texttt{else}\ \mathcal{F}(e_3)]\\
\mathcal{F}(\texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{done}) &= \texttt{while}\ \mathcal{G}(e_1)\ \texttt{do}\ \mathcal{F}(e_2)\ \texttt{done};\ \texttt{mark()}\\
\mathcal{F}(\texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2) &= \texttt{let}\ x\ =\ \mathcal{G}(e_1)\ \texttt{in}\ \mathcal{F}(e_2)\\
\mathcal{F}(\texttt{match}\ e_1\ \texttt{with}\ p_i\ [\texttt{when}\ e_2]\ \rightarrow\ e_i) &= \texttt{match}\mathcal{G}(e_1)\texttt{with}\ p_i\\
&\qquad [\texttt{when}\mathcal{G}(e_2) \rightarrow \mathcal{F}(e_i)]\\
\mathcal{F}(\texttt{try}\ e\ \texttt{with}\ p_i\ \rightarrow\ e_i) &= \texttt{try}\ \mathcal{F}(e)\ \texttt{with}\ p_i\ \rightarrow\ \mathcal{F}(e_i)\\
\mathcal{F}((e_1,\ e_2)) &= \texttt{mark()};\ (\mathcal{G}(e_1),\ \mathcal{G}(e_2))\\
\mathcal{F}(C(e)) &= \texttt{mark()};\ C(\mathcal{G}(e))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{G}(x) &= x \text{ if } x \text{ is a constant value or an identifier}\\
\mathcal{G}(f\ x) &= \mathcal{G}(f)\ \mathcal{G}(x)\\
\mathcal{G}(\texttt{fun}\ x\ \rightarrow\ e) &= \texttt{fun}\ x\ \rightarrow\ \mathcal{F}(e)\\
\mathcal{G}(e_1;\ e_2) &= \mathcal{G}(e_1);\ \mathcal{G}(e_2)\\
\mathcal{G}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ [\texttt{else}\ e_3]) &= \texttt{if}\ \mathcal{G}(e_1)\ \texttt{then}\ \mathcal{F}(e_2)\ [\texttt{else}\ \mathcal{F}(e_3)]\\
\mathcal{G}(\texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{done}) &= \texttt{while}\ \mathcal{G}(e_1)\ \texttt{do}\ \mathcal{F}(e_2)\ \texttt{done}\\
\mathcal{G}(\texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2) &= \texttt{let}\ x\ =\ \mathcal{G}(e_1)\ \texttt{in}\ \mathcal{G}(e_2)\\
\mathcal{G}(\texttt{match}\ e_1\ \texttt{with}\ p_i\ [\texttt{when}\ e_2]\ \rightarrow\ e_i) &= \texttt{match}\ \mathcal{G}(e_1)\ \texttt{with}\ e_1\\
&\qquad [\texttt{when}\mathcal{G}(e_2)] \rightarrow \mathcal{F}(e_i)\\
\mathcal{G}(\texttt{try}\ e\ \texttt{with}\ p_i\ \rightarrow\ e_i) &= \texttt{try}\ \mathcal{G}(e)\ \texttt{with}\ p_i\ \rightarrow\ \mathcal{F}(e_i)\\
\mathcal{G}((e_1,\ e_2)) &= (\mathcal{G}(e_1),\ \mathcal{G}(e_2))\\
\mathcal{G}(C(e)) &= C(\mathcal{G}(e))
\end{aligned}
$$

Correction of the instrumentation. Since `mark()` has type *unit* (computes by side-effect), the translations defined by functions $\mathcal{F}$ and $\mathcal{G}$ do not alter the types of the expressions being instrumented. Furthermore, they do not alter the value computed by this expression.

A program is structurally covered when every call to `mark()` in the instrumented source code has been reached.

Tail recursion: Tail recursion is not a feature of OCaml or of functional languages. It is a property of a function, in which the last operation is a recursive

call. Such recursions can be easily transformed into iterations: this is known as the tail call optimization. Replacing recursion with iteration can drastically decrease the amount of stack space used and improve efficiency.

Our instrumentation algorithm, consisting in adding a side-effect after each expression, systematically breaks tail calls, thus forbidding the optimization mentioned above. In pratice (with the Scade compiler typically), we were not confronted with cases in which the instrumentation of the program led to a stack overflow.

3.2 MC/DC Coverage

According to the DO-178B standard, MC/DC is fulfilled when every point of entry and exit in the program has been invoked at least once, every condition in a decision has taken on all possible outcomes at least once, and each condition has been shown to affect that decision's outcome independently.

With respect to the OCaml language, we chose to define an MC/DC decision for each expression of type *bool* (except the Boolean constants `true` and `false`). Then, MC/DC conditions are determined by syntactically looking for the Boolean operators `not`, `&&` and `||`. We propose to transform every Boolean expression into a bunch of nested `if then else`. Here is the translation scheme:

$$\begin{aligned}
\mathcal{F}_b(\texttt{not}\ e,\ l,\ r) &= \mathcal{F}_b(e,\ r,\ l)\\
\mathcal{F}_b(e_1\ \texttt{\&\&}\ e_2,\ l,\ r) &= \mathcal{F}_b(e_1,\ \mathcal{F}_b(e_2,\ l,\ r),\ r)\\
\mathcal{F}_b(e_1\ \texttt{||}\ e_2,\ l,\ r) &= \mathcal{F}_b(e_1,\ l,\ \mathcal{F}_b(e_2,\ l,\ r))\\
\mathcal{F}_b(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ l,\ r) &= \mathcal{F}_b(e_1,\ \mathcal{F}_b(e_2,\ l,\ r),\ \mathcal{F}_b(e_3,\ l,\ r))
\end{aligned}$$

$$\mathcal{F}_b(e,\ l,\ r) = \texttt{if set\_condition();}\ e\ \texttt{then}\ l\ \texttt{else}\ r \quad \text{otherwise}$$

Thence, the MC/DC instrumentation $\mathcal{M}$ of a Boolean expression e can be defined straightforwardly:

$$\mathcal{M}(e)\ =\ \mathcal{F}_b(e,\ \texttt{set\_outcome();}\ \mathit{true},\ \texttt{set\_outcome();}\ \mathit{false})$$

where `set_condition()` and `set_outcome()` are side-effects (of type *unit*) allowing to update respectively the value of the current condition and the value of the decision's outcome.

Example. The following Boolean expression is composed of four independent conditions. Here they are single variables, but could also be replaced with more complex expressions.

```
let pred a b c d = (a || b) && (c || d)
```

There are 2^4 possible tests. Coverage of expressions requires 2 tests, whereas MC/DC needs 5. The $\mathcal{F}_b$ translation reveals 7 calls to `set_outcome()` ; , thus the 2^4 test cases fall into 7 classes according to the way they affect our counters.

```
let pred a b c d =              (*       a  b  c  d  -> p *)
  if a then
    if c then true              (* 1 : T _  T _  -> T *)
    else if d then true         (* 2 : T _  F T  -> T *)
    else false                  (* 3 : T _  F F  -> F *)
  else if b then
    if c then true              (* 4 : F T  T _  -> T *)
    else if d then true         (* 5 : F T  F T  -> T *)
    else false                  (* 6 : F T  F F  -> F *)
  else false                    (* 7 : F F  _ _  -> F *)
```

Let us find, for each condition, which test pairs are sufficient to prove that the decision's outcome is independently affected:

a : (1)+(7) b : (4)+(7) c : (1)+(3) or (4)+(6) d : (2)+(3) or (5)+(6)

which leads us to the following minimal sets:

{(1), (2), (3), (4), (7)} or {(1), (4), (5), (6), (7)}

As a consequence, full MC/DC coverage can be achieved with 5 tests, which confirms the theoretical result. Mind that our translation is only required to measure MC/DC coverage, it isn't a method to derive a minimal set of test cases from the source code: hence the discrepancy between the 5 tests required for full MC/DC coverage and the 7 possible tests. In other words, it is not necessary to cover the translated version of the decision in its entirety to fulfill the MC/DC criterion.

3.3 Implementation

We developed a tool capable of measuring the MC/DC rate of OCaml programs. The tool first allows to create an instrumented version of the source code, together with a trace file. Then, the user has to build the instrumented code with the Inria OCaml compiler. Running the instrumented executable leads to (incrementally) updating the counters and structures of the trace file. Finally, the coverage results are presented through HTML reports, that are generated from the information collected in the trace file.

Our tool is built on top of the front-end of the INRIA OCaml compiler. A first pass is always done, prior to the instrumentation stage, in order to reject OCaml programs that do not comply with the coding rules related to the Scade compiler. For instance, we do not support objects, polymorphic variants, labels and optional arguments, nor the unconventional extensions (multi-threading, support for laziness, recursive values and modules).

Performance Results. Performances are quite good with respect to programs that contain a lot of pattern-matching and a few recursive calls. Thus, the Scade to C compiler has been successfully instrumented and used to compile non trivial

Scade programs. The instrumented version of Scade compiler runs almost as fast as the original one.

	Scade compiler	instrumented Scade compiler
number of lines	30 000	53 500
execution time on a large Scade model (8 120 lines)	27.6 s	28.1 s

On the contrary, performances are very degraded with OCaml programs that use recursion intensively, such as a naive implementation of `fibonacci`. Indeed, in those cases, counters can be hit several millions (even billions) of times, whereas in the case of Scade they are updated a few hundreds or thousands of times only.

The η-expansion of polymorphic variables can introduce a lack of performance with a program that uses intensively the higher-order features of the language. This lack has not been measured, but the cost of abusive closure constructions is well known for any functional language developer.

The lack of performance is not a point to our purpose because the instrumented code is only used to measure the coverage of the code. So it will be used in a large set of tests; but it is used on real application cases. Most of cases, including the Scade compiler, the tests needed by the coverage of the source code are small and the instrumented code performs quite well (less than 5% of overcost). Furthermore, the code coverage analysis is a heavy process: tests building, test validation, coverage results analysis, ... In this context, the slight lack of performance is not relevant.

4 Certification of OCaml Programs

The DO-178B certification of an application applies on the final executable. Thus the analysis must be applied to the source code of the progam itself but also on the library code used by the program. A typical OCaml program such as the Scade compiler uses two kind of library code: the OCaml *standard library* which is itself written in OCaml, and the *runtime library*, written in C and assembler; both libraries are shipped with the OCaml compiler and are automatically linked with the final executable.

The standard library contains the interfaces manipulating the datatypes predefined by the OCaml compiler (integers, strings, etc.), the implementation of some commonly used data structures (hash tables, sets, maps, etc.) and some utilitary modules (command line parsing, `printf`-like formatting, etc.). The runtime library contains:

- the implementation of some library functions that cannot be written in pure OCaml because they need access to the in-memory representation of OCaml values: the polymorphic comparison function, the polymorphic hashing function, the serialization/deserialization functions;

– library functions that need to interact with the host OS, most notably the I/O functions;
– low-level support code that is required for the execution of OCaml programs: memory management functionality, exceptions management, support for some function calls, etc. Use of this code is not always traceable to the original source in OCaml, it is often introduced by the OCaml compiler.

The difficulty of specifying and testing such low-level library code (as required by the DO-178B process) lead us to adapt the runtime library so as to simplify it.

4.1 Modifications to the Runtime Library

The bulk of the modifications was to remove features unessential to our specific application, the Scade compiler. This is a program with a relatively simple control flow and very little interaction with the OS: its inputs consist only of command line arguments and text files, its outputs are also text files.

First, the concurrency support was removed from the runtime. OCaml programs can use POSIX signals and multi-threading but this feature is dispensable when writing a compiler.

Similarly, the support for serialization and deserialization of OCaml values was removed. Furthermore these functions are not type-safe and thus can compromise the safety guarantees of the OCaml type system.

Most of the work consisted in simplifying the automatic memory management subsytem. Indeed the garbage collector (GC) of OCaml is renowned for its good performances; however it is a large and complex piece of code. It is a generational GC with Stop&Copy collection for the young generation and incremental Mark&Sweep collection for the older generation; it supports compaction, weak pointers and finalization functions. We successfully replaced it by a plain Stop&Copy collector, thus eliminating features unnecessary to our compiler such as weak pointers and finalization. The collector is no longer incremental, which implies that the execution of the program may be interrupted for a large amount of time by the collector, however this is of no concern for a non-interactive application such as a compiler.

Simplifying this part of the runtime library was difficult because of its tight coupling with the OCaml compiler. Indeed, both this memory manager code and the compiler must agree on the in-memory representation of OCaml values and on the entry points of the memory manager. Furthermore, the OCaml compiler inlines the allocation function of the memory manager for performance reasons. All in all, we had little leeway in replacing this code: it practically had to be Stop&Copy collector and we had to keep some of the symbol names. However we were able to obtain complete coverage of this simplified GC, despite the fact that it is difficult to test since most of the calls are not explicit in the original OCaml source code.

The OCaml standard library is less problematic concerning certification. Most of it is written in plain OCaml and certification of this library code is no more difficult than that of the application code. Some of the more complex modules such as the `printf`-like formatters were simply removed.

The only notable modification of the standard library is the support of overflows in integer arithmetics. The built-in integers in OCaml implement a signed 31 bit (or 63-bit, depending on the platform) wrap-around arithmetic. To be able to detect overflows, the basic arithmetic functions were replaced by wrapper function that check the result and raise an exception in case of overflow.

4.2 Performance Results

The modifications of the runtime library that can impact the program's performance are the new GC and the overflow-checking arithmetic operations; other modifications are merely removal of unused code and do not alter performance. Tests were done on the (non-instrumented) Scade compiler running with the same large Scade model as in section 3.3.

To check the impact of overflow checks, we tested our modified runtime library with and without overflow detection. No measurable difference could be seen. This is expected as the Scade compiler does very few arithmetic computations.

To measure the performance of the GC, we measured the total running time of the program and the top memory consumption; individual collection time was not measured (as mentioned earlier, "pauses" in the execution due to the GC are not a concern for a compiler). We found the Scade compiler to be approximately 30% slower than with the regular OCaml GC. The memory footprint reached 256 MB vs. 150 MB with the regular GC. This was expected: Stop&Copy GC are not very memory-efficient: due to their design, no more than half the memory used by the GC is available to store program data.

5 Discussion

5.1 Approaches

Our approach in this article is to focus directly on OCaml programs and on the OCaml compiler from Inria. To ensure compatibility of this approach with the Scade compiler, we have restricted the OCaml language to its functional and imperative core language, including a basic module system. The runtime library pointers, serialization ...) has also been simplified. One pending difficulty is to explain compilation schemes for language features and their composition.

Another approach to certificate OCaml programs would be to use a compiler from ML to C [19,5] and then to certify the generated C code by using tools for C code coverage. Once again the main difficulty is to check the GC implementation of the runtime library; GC with ambiguous roots using [3] or not [5] the C stack may "statistically" fail the certification. The simple GC as Stop&Copy [11] are not appropriate to the C language because they move their allocated values, mainly GC regarding the C stack as roots set.

A third approach, which can be compatible with the first, is to use a byte-code interpretor. Its strengh is to improve control to manage stack and exceptions. Moreover, an interpretor gives the possibility to analyse the program coverage

during execution and not only by its instrumentation. A Just in Time translator can be added to improve performances [18]. A JIT transformation is easier to explain and to describe during the certification process than an entire compiler, mainly because its optimisations are less complex.

These three approaches allow the use of high level languages to develop tools for embedded softwares. This will reduce the development life cycle and simplify the certification process.

5.2 Future Work

The pattern matching is one of the most important features of the OCaml language. It can be considered both as a control structure and as the only way to build accessors to complex values. Moreover, the static analysis [14], used by the OCaml compiler, ensure some good properties. In this paper, we consider that a pattern matching instruction is covered by a single test for each pattern of the filter. This is sufficient with respect to the definition of MC/DC requirements which are only applicable on Boolean expressions. An extension of the coverage principles is to consider a pattern matching as multiple conditions and to require to cover the independance between any of the condition. For instance, the pattern `x::y::l ->` matches any list of at least two elements; intuitively, it expresses two conditions: the list is not empty and the the tail of the list is not empty too. A more precise coverage measure can ask to have two different tests for this pattern.

The more modern features of OCaml [1] are not necessarily wished by the certification organizations to design critical softwares. For instance the object programming, *à la C++*, is not yet fully accepted by the DO-178B; and the row polymorphism from the OCaml object extension may not satisfy all their criteria. In the same way, polymorphic variants bring a concept of extensibility that is not compatible with the critical software development, which requires comprehensive specifications for all used data structures.

On the other hand, the genericity of functors (parametric modules) is valuable to build this kind of tools, but when a functor is applied, the same types constraints than parametric polymorphism have to be checked These restrictions are under study. A simple solution to properly cover parametric modules is to consider independently any of its monomorphic instance. But this solution leads to demand more tests than the necessary ones: when a part of a functor does not use some arguments, it can share the same tests to ensure the coverage.

6 Conclusion

For the community of statically typed functional languages, usual arguments on quality, safety and efficiency about code written in OCaml are well known and accepted for a long time. Nevertheless, convincing the authorities of certification requires to respect their measuring criteria of quality. This development has shown that the concepts of MC/DC coverage could be used for a functional/imperative subset of OCaml and its simplified runtime. Although it is not

applicable to embbed code written in OCaml, satisfying criteria from DO-178B gives to OCaml the capabilities to specify and to implement tools for design of critical softwares.

The Scade compiler of *Esterel Technologies* is such a tool, it has been certified DO-178B level A by the American and the European civil aviation administrations; it is used for instance by Airbus, Pratt and Whitney and many others. Previously implemented with the C language, the compiler of the version 6 of Scade has been written in OCaml and will be submitted to the qualification procedures. The code coverage analysis will be performed by the *mlcov* tool described in this paper. Notice that *mlcov* needs to be DO-178B level C certified which is a necessary condition to be used in a DO-178B level A cycle development.

References

1. Aponte, M.-V., Chailloux, E., Cousineau, G., Manoury, P.: Advanced Programming Features in Objective Caml. In: 6th Brazilian Symposium on Programming Languages(June 2002)
2. Berry, G.: The Effectiveness of Synchronous Languages for the Development of Safety-Critical Systems. Technical report, Esterel-Technologies (2003)
3. Boehm, H., Weiser, M., Bartlett, J.F.: Garbage collection in an uncooperative environment. Software - Practice and Experience (September 1988)
4. Camus, J.-L., Dion, B.: Efficient Development of Airborne Software with SCADE Suite^{TM}. Technical report, Esterel-Technologies (2003)
5. Chailloux, E.: An Efficient Way of Compiling ML to C. In: Workshop on ML and its Applications. ACM SIGPLAN (June 1992)
6. Colaço, J.-L., Pagano, B., Pouzet, M.: A Conservative Extension of Synchronous Data-flow with State Machines. In: ACM International Conference on Embedded Software (EMSOFT 2005), Jersey city, New Jersey, USA (September 2005)
7. Colaço, J.-L., Pouzet, M.: Clocks as First Class Abstract Types. In: Third International Conference on Embedded Software (EMSOFT 2003), Philadelphia, Pennsylvania, USA (October 2003)
8. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astrée analyser. In: European Symposium on Programming. LNCS (April 2005)
9. Dupuy, A., Leveson, N.: An empirical evaluation of the mc/dc coverage criterion on the hete-2 satellite software. In: Digital Aviations Systems Conference (DASC), Philadelphia, Pennsylvania, USA (October 2000)
10. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., Rierson, L.K.:A Practical Tutorial on Modified Condition/Decision Coverage. Technical report, NASA/TM-2001-210876 (May 2001)
11. Jones, R., Lins, R.: Garbage Collection. Wiley, Chichester (1996)
12. Kapoor, K., Bowen, J.P.: Experimental evaluation of the variation in effectiveness for dc, fpc and mc/dc test criteria. In: ISESE, pp. 185–194. IEEE Computer Society, Los Alamitos (2003)
13. Leroy, X.: The Objective Caml system release 3.10 : Documentation and user's manual (2007), `http://caml.inria.fr`
14. Maranget, L.: Warnings for pattern matching. Journal of Functional Programming (2007)

15. Pouzet, M.: Lucid Synchrone version 3.0 : Tutorial and Reference Manual (2006), www.lri.fr/~pouzet/lucid-synchrone
16. T.C.D.T.L. Project: The Coq Proof Assistant Reference Manual (2006), http://coq.inria.fr/V8.1beta/refman
17. RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics RTCA (December 1992)
18. Starynkevitch, B.: OCamljit - a faster Just-In-TIme Ocaml implementation. In: Workshop MetaOcaml (June 2004)
19. Tarditi, D., Lee, P., Acharya, A.: No assembly required: Compiling standard ML to C. ACM Letters on Programming Languages and Systems 1(2), 161–177 (1992)