

Non-Intrusive Structural Coverage for Objective Caml

Philippe Wang¹, Adrien Jonquet², Emmanuel Chailloux³

*Équipe APR
Laboratoire d'Informatique de Paris 6 (CNRS UMR 7606)
Université Pierre et Marie Curie (Paris 6)
4 place Jussieu, 75005 Paris, France*

Abstract

This paper presents a non-intrusive method for Objective Caml code coverage analysis. While classic methods rewrite the source code to an instrumented version that will produce traces at runtime, our approach chooses not to rewrite the source code. Instead, we use a virtual machine to monitor instructions execution and produce traces. These low-level traces are used to create a machine code coverage report. Combined with control-flow debug information, they can be analyzed to produce a source code coverage report. The purpose of this approach is to make available a method to generate code coverage analysis with the same binary for testing and for production. Our customized virtual machine respects the same semantics as the original virtual machine; one of its original aspects is that it is implemented in the Objective Caml, the very language we build the tool for. This work is part of the *Coverage* project, which aims to develop open source tools for safety-critical embedded applications and their code generators.

Keywords: Certification Tools Design, Code Coverage, Objective Caml Virtual Machine

1 Introduction

One of the most demanding development process for safety-critical software was defined a couple of decades ago by the civil avionics authorities as the DO-178B standard [17]. This standard notably contains all constraints ruling aircraft software development. A very precise development process is imposed, and its preponderant activity is independent verification of each development step. Product specifications are written by successive refinements, from high-level requirements to design and then to implementation. Each step owns an independent verification activity, which must provide a complete traceability of the requirements appearing at this stage.

¹ Email: Philippe.Wang@lip6.fr

² Email: Adrien.Jonquet@lip6.fr

³ Email: Emmanuel.Chailloux@lip6.fr

The certification process, required before actually using such a software, mainly consists in making a specification and testing process document reporting that software specifications and implementation are tested by another entity to show that its behaviour conforms to its specifications.

Code coverage reports are part of the documents required by the certification process. They are generated from the program’s source code, and the execution traces of the compiled program. The classic approach to obtain the latter is to add instructions in the source code to produce them, while keeping the same semantics otherwise. For instance, the Esterel Technologies company developed such a tool for Objective Caml [14], a multiparadigm programming language [12], which is not widely used in the safety-critical domain. However it has already been successfully used for safety-critical development tools, e.g., a code generator written in Objective Caml is used in Esterel Technologies’ SCADE Suite [15].

A different approach consists of keeping the same program without adding instructions but instead to run it in a modified execution context. This approach means that the code coverage tools do not instrument the original program, so that the binary executed for code coverage testing purpose can be the exact same binary as for the final product. This is the core of the Coverage project ⁴ which interests on the QEMU virtualizer and the ZINC machine [11], the Objective Caml virtual machine, called ZAM ⁵ here after.

Both approaches should produce the same reports, but the non-intrusive way should shorten the traceability process because the exact same code can be executed for both functional testing and coverage testing.

In this paper, we focus on this Objective Caml multiparadigm programming language, which is distributed in an open source package that contains – among other things – a compiler and a virtual machine. One major motivation for using Objective Caml is that it has already been used with success in a certification framework.

We present ZAMCOV, a new ZAM implementation in Objective Caml, which produces traces at runtime for future code coverage analysis. This work will be compared to Esterel Technologies’ approach.

This paper is organized as follows: section 2 describes a past experiment on using Objective Caml in the safety-critical software domain; section 3 details the project in which our work takes part; section 4 presents the ZAM machine and our implementation, first step in non-intrusive code coverage process; section 5 shows non-intrusive code coverage process at the machine code level and how to go from machine code coverage to source code coverage with our tool available at <http://www.algo-prog.info/zamcov> and section 6 describes related work and announces our future work in this project.

⁴ This project is supported in part by the SYSTEM@TIC PARIS-REGION Cluster in the Free and Open Source Software thematic group (<http://www.projet-couverture.com/>). Two companies are involved in the development: AdaCore and OpenWide, together with two academic partners: Telecom ParisTech and University Pierre et Marie Curie (Paris 6).

⁵ ZAM stands for ZINC Abstract Machine

2 Structural Coverage in Objective Caml by Esterel Technologies

The French company Esterel Technologies ⁶ decided in 2006 to base its new SCADE SUITE 6™ ⁷ [4,5] certifiable code generator on Objective Caml. Esterel Technologies markets SCADE SUITE 6™, a model-based development environment dedicated to safety-critical embedded software. The code generator (KCG ⁸) 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 use it.

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 should be effective soon.

2.1 Code Coverage and MC/DC (Modified Condition/Decision Coverage)

Among the numerous testing activities, one is making reports on code coverage. This activity has a set of constraints other than just showing whether some code is alive or dead: for instance, if a result is a complex Boolean expression, it is not enough to show that it has been evaluated (to any value). Neither is it enough to show it has taken both true and false values. Indeed, a complex Boolean expression is composed with sub Boolean expressions, and these also have to have taken both true and false values. Plus, if two subexpressions always return the same value, it is suspicious: are they duplicated?

As any activity during a DO-178B compliant development process, the verification activities are evaluated. Some criteria must be reached to decide that the task has been completed. One of these criteria is the activation of any part of the code during a functional test. On this particular point, more than a complete structural exploration of the code, the DO-178B standard requires that a complete exploration of the control flow has to be achieved following the MC/DC measurement that we explain below.

- A *decision* is the Boolean expression evaluated in a test instruction to determine the branch to be executed. It is covered if there exist tests in which it is evaluated to **true** and **false**.
- A *condition* is an atomic subexpression of a decision. It is covered if there exist tests in which it is evaluated to **true** and **false**.
- MC/DC requires that, for each condition *c* of a decision, there exist two tests which must change the decision value while keeping the same valuations for all

⁶ <http://www.esterel-technologies.com>

⁷ SCADE stands for *Safety Critical Application Development Environment*; *Scade* is the programming language provided by SCADE SUITE 6™.

⁸ KCG stands for *qualifiable Code Generator*.

conditions but c . It ensures that each condition can affect the outcome of the decision and that all contribute to the implemented function (no dead code is wanted).

MC/DC is properly defined on an abstract Boolean data flow language [10] with a classical automata point of view. The measure is extended to imperative programming languages, especially the C language, and is implemented in verification tools able to compute this measure.

2.2 *MLcov: an Objective Caml Code Coverage Tool*

MLcov [1] is an open source code coverage measurement tool for Objective Caml developed by Esterel Technologies. MLcov only treats the functional and imperative features of Objective Caml, which correspond to the subset allowed by the coding rules of the Scade-to-C compiler. This subset remains quite large, for instance, it is sufficient to compile the standard library of the Objective Caml distribution.

Coverage is measured by instrumenting the source code of the program. With respect to Objective Caml, we state that an expression is covered as soon as its evaluation ends. The main idea of the instrumentation algorithm is to replace each expression `expr` with `(let aux = expr in mark(); aux)`, where 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.

A program is structurally covered when every call to `mark()` in the instrumented source code has been reached. This instrumentation algorithm, detailed in [14] and consisting in adding a side-effect after each expression, systematically breaks tail calls, thus forbids this optimization.

2.3 *New certified KCG*

The new developed-in-Objective-Caml KCG is certified with respect to IEC 61508 and EN 50128 norms. It is used in several civil avionics DO-178B projects (e.g., for the A380 Airbus plane) and will be qualified simultaneously to the project qualifications (with DO-178B, the tools are not qualified by themselves, but by their usage in a project).

3 Code Coverage with Non-Intrusive Tools: The Coverage Project

The Coverage project, which started a year ago, aims at providing non-intrusive coverage tools in a free software/open source context for safety-critical applications.

In the Coverage project, the main idea is not to instrument the code directly but instead to instrument the runtime environment which executes the code as shown in figure 1. This execution produces some traces which can be analysed offline (i.e. after the execution) and mapped back to the original program source. In this case the final machine code will be executed in a special runtime context.

Two (language \times target machine) approaches are studied: the first is (Ada language \times PowerPC processors family), the second is (Objective Caml language \times

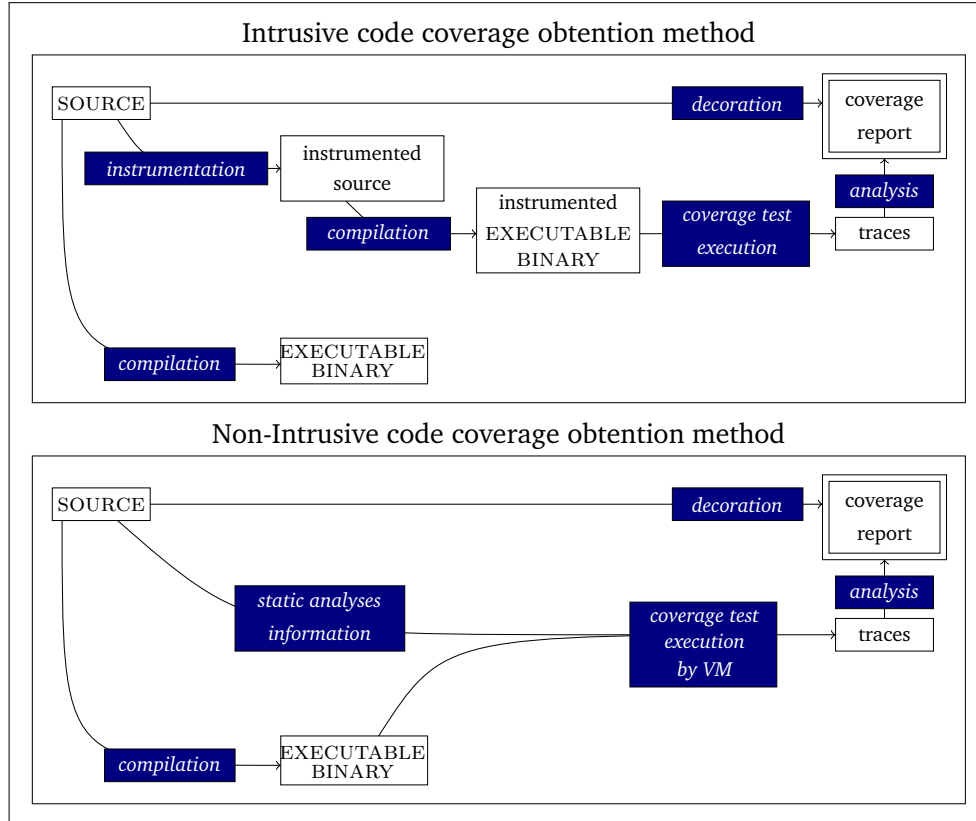


Figure 1. Code Coverage Obtaining Methods Comparison

ZAM its own virtual machine). These two couples are used for safety-critical embedded applications (Ada) and code generators (Objective Caml), including avionics projects using the DO-178B standard.

For common traditional processors, QEMU [2] is used as a free-software emulator (Power-PC, ...) which can generate traces. This allows non-intrusive analysis on final target code with emulators running on development hosts. In this part, the Adacore⁹ company develops tools which are independent from the language, like Ada or C, and from the compiler, by using source DWARF [3] debug info. This independance implies additional yet several restrictions for MC/DC.

For virtual machines, we have studied Objective Caml virtual machine to produce traces. These traces are analyzed after the execution to produce a structural coverage report for machine code and source code. To make the link between machine code and source code, we use debug information added by the Objective Caml compiler in the debug mode. This information, called events, indicates the beginning or the end of an expression. With these events, the control flow graph of a program can be rebuilt during or after an execution. For MC/DC analysis more information is needed, so the original Objective Caml compiler has to be modified. This will be discussed in section 6.

At mid-term of the project, we can present our progression in the two following

⁹ <http://www.adacore.com>

sections on machine code and source code coverage.

4 An Objective Caml Virtual Machine in Objective Caml

Generating a machine code coverage report first means to know the compiled program's binary format and semantics to interpret it, this work is done by a virtual machine. Then, during interpretation, one needs to keep the execution flow for further analysis; this work is done by a component plugged into the virtual machine. In our case, we implemented an Objective Caml virtual machine such that it is easy to extend with pluggable components. For this, we chose Objective Caml as the implementation language, for several reasons:

- it is interesting to implement a virtual machine in the very language it is designed for;
- it brings the bootstrap challenge, so the resulting tool can be used for itself;
- Objective Caml is strongly statically typed, so the interpreter does not use unsafe type casts
- and it permits to build applications that are naturally robust to components plugins.

4.1 The Objective Caml Virtual Machine (ZAM)

ZAM is a stack-based virtual machine for a functional-based multiparadigm programming language. It only uses 7 registers: an accumulator to store a value, a code pointer (next instruction to interpret), a stack pointer, another stack pointer for the highest exception handler, an extra arguments counter, an environment (a value array) and a global data (a value array). ZAM interprets 146 different instructions, about 60% of which are shortcuts for several instructions combinations. 18 instructions are for arithmetic and Boolean operations.

Values are uniformly represented, it makes exploring a value easy, notably for the garbage collection system. A value is either a integer encoded on 32-1=31 bits or 64-1=63 bits depending on the architecture, or a block value whose header encodes a block tag on one byte (*e.g.*, closure tag, string tag, double tag or variants) and a block size. This integer representation is actually an optimization: since it is often sufficient to have 31 or 63-bit (signed) integers, and since the weakest bit of an address is never set to 1, it is possible to use 32 or 64-bit integers to represent immediate integers. As a consequence, the compiler will automatically convert some int values: *e.g.*, 0 becomes 1, 1 becomes 3, n becomes $n \times 2 + 1$, and arithmetic operations are modified in consequence. Since increasing arithmetic operations has a lower cost than having a pointer dereferencing, the loss of performance is acceptable. An instruction is dedicated to recognizing an integer from a block: `IS_INT`.

Objective Caml has functional values that are encoded as closures (functions \times environment), some specific instructions handle to them (*e.g.*, `APPLY`, `CLOSURE`, `CLOSUREREC`, `GRAB`, `OFFSETCLOSURE`).

4.2 Existing Objective Caml Virtual Machine Implementations

The Objective Caml virtual machine has several implementations, the original one is in C, one is in Java and another is in JavaScript. They are quickly described as follows.

- **In C code:** The INRIA standard distribution provides a virtual machine implemented in C code, it is likely the most efficient implementation available. Its written-in-C runtime library is the same as the one used with hardware machine programs that are produced by the native compiler.
- **In Java code:** The Java implementation, called Cadmium [7], allows an Objective Caml program to be executed on any machine that has a Java Virtual Machine, without having to install the whole Objective Caml system. For instance, this can be used to easily run Objective Caml programs on a web page. Parts of its runtime library rely on Java runtime library such as garbage collection, the other part is in Java.
- **In JavaScript:** The JavaScript implementation, called O'Browser [6], gives the possibility to write dynamic web page components (that are usually written in JavaScript) in Objective Caml. As it is not relevant to have exactly the same runtime library as the original distribution, an alternative version provides an interface with web page related functions.

4.3 Our New Implementation in Objective Caml itself

It is important to note that whereas Objective Caml is strongly statically typed, its virtual machine is untyped. This design was motivated by the guarantee that static type checking process frees the runtime process from making any type checks. Writing an Objective Caml virtual machine in Objective Caml implies writing an untyped virtual machine for a strongly typed programming language in a strongly typed programming language. It is analog with the runtime library: Objective Caml runtime library has two parts: the low-level part is a set of C functions that may access low-level data representations, and the high-level part is a set of Objective Caml functions that may use functions implemented in C code.

We chose to implement ZAMCOV values type as follows:

```

type tag =
  | Structured_tag of int
  | Closure_tag
  | Object_tag
  | Abstract_tag
  | ...

type value =
  | Int of int
  | Float of float
  | String of string
  | Block of block
  | ...

and block = { tag : tag; data : value array; ... }

```

From a bytecode binary, ZAMCOV initializes a virtual machine data structure with the instructions section, the global data section and the set of external functions. Then the interpreter function is linked to this data structure to make it able to run instruction by instruction: it is easy to plug a component into the interpreter.

The original standard library is a set of Objective Caml functions, some of which call some C code. For instance, operations on files are implemented with C functions encapsulated for Objective Caml. When compiling an Objective Caml function to bytecode instructions, there are two cases:

- the underlying functionality is in C code: the bytecode will contain C_CALL instructions ¹⁰;
- it is fully in “pure” Objective Caml: bytecode instructions do not contain C_CALL instructions.

The first case is not trivial with an alternative bytecode interpreter. In our case, with ZAMCOV, C_CALL instructions will be interpreted by an Objective Caml function, and will mean calling an Objective Caml function. Thus, for instance, when calling a I/O operation (or any operation that cannot be directly represented by some bytecode instructions), an indirection is added. The source code in Objective Caml is compiled to bytecode, which is then interpreted by an Objective Caml program.

For instance, to call a C function `foo` from the original virtual machine, the C_CALL instruction is used with “foo” as first argument and it will call the C function. This C function cannot be called directly at the interpretation of a C_CALL instruction, because our value representation is different.

Implementing the runtime library is a weird constraint: the original runtime library is implemented in low-level C code, and this is a behaviour that has to be reproduced in our Objective Caml runtime library. For instance, comparison functions (which compare data structures in depth) are based on the comparison function (`val compare : 'a -> 'a -> int`) which is implemented in C code as part of the runtime library, and since our data representation is not exactly the same, we cannot use it directly when the machine code invokes function `compare` as a C call. This means we had to implement in Objective Caml the comparison function for our data representation that emulate the original data representation.

¹⁰ There is a set of “C call” instructions (C_CALL[1-5], C_CALL_N) that allows the bytecode to call external functions (i.e. functions that are not to be compiled to bytecode).

Indeed, one constraint was not to break the type checker because otherwise implementing first draft of the virtual machine would have been quicker but its debugging would have been a nightmare.

5 ZAMCOV's code Coverage Tools

5.1 Execution trace generation

The first component plugs itself in the instructions interpreter and keeps trace of which instructions are executed. Thanks to the design of our virtual machine implementation, it is quite easy to write and plug a component into it. The trace is an array whose length equals the code section's length of the bytecode. When an instruction of the code section is executed, the trace's element whose index is the address of the instruction in the bytecode is marked as "covered". The Objective Caml array, which contains the execution trace, is serialized in an external file when ZAMCOV ends the interpretation of the program.

5.2 Machine Code Coverage

Traces are analyzed after the execution to generate an instruction coverage report (in HTML format). The first report is a machine code coverage report. This report represents a list of all non-covered (never executed) bytecode instructions.

Here is an example of the machine code coverage report of a simple factorial program:

```
let rec fact x =
  if x = 0 then 1
  else if x = 1 then 1
  else x * (fact (x-1));;

fact 5;;
```

This program is a simple function computing the factorial. Application `fact 5` does not allow the first test (`x = 0`) to become true, therefore the first branch, which returns 1, is not taken.

```
ocamlc fact.ml -o fact
zamcov-run -trace fact.trace fact
zamcov-cover fact.trace fact
```

First we need to compile the `fact.ml` file with standard `ocamlc` compiler. Then, with `zamcov-run` we interpret and generate the execution trace of the program `fact` and we build the coverage report with `zamcov-cover`.

00000002	ACC0
00000003	BNEQ
00000006	CONST1
00000007	RETURN
00000009	ACC0
00000010	BNEQ
00000013	CONST1
00000014	RETURN

The code sample on the left shows part of the machine code coverage result for program factorial. There are some boxed instructions: they are never executed and correspond to the code which returns the constant 1, which indeed is not executed in the factorial example.

Then one question was to know whether there was an equivalence between machine code coverage and source code coverage, as such an equivalence would

remove the need for source code coverage. We will see in the rest of the paper that they are not equivalent.

5.3 Source Code Coverage

The central role of ZAMCOV’s virtual machine is to interpret Objective Caml programs compiled to bytecode by standard distribution’s compiler `ocamlc`. This section presents a ZAMCOV component that generates the execution trace (representing all executed bytecode) instructions during the interpretation.

This trace only contains information about bytecode instructions, such as instructions names or their addresses in the bytecode. Hence, we need a mean to link this information with the source code.

Using Debug Events to Generate a Code Coverage Report

Debug events are debug information added by the compiler when using debug option “-g”. They are used by `ocamldebug`, the Objective Caml debugger. These events are not in program’s code section. There are not differences in this section for a program compiled with or without the debug option. This is important because ZAMCOV is a non-intrusive code coverage tool, so it is not supposed to modify the source code or the bytecode of the program. Debug events are located in an independent section of the program binary, not in the code to be interpreted, so they could be in a separate file only visible by the virtual machine if needed.

A debug event is a data structure linked with an Objective Caml expression during the compilation with the standard compiler (`ocamlc`). Debug events are located strategically in an Objective Caml program as shown in figure 2.

A debug event contains a lot of information about its expression:

- the location of the expression in the source code;
- the first bytecode instruction address corresponding to the Objective Caml expression.

The address in the bytecode recorded by the debug event is the missing link with the execution trace.

Coverage of Objective Caml expressions

ZAMCOV is also a source code coverage measurement tool. First coverage level is statement coverage. Objective Caml is a functional-based language (a program is an expression evaluation), so every “statement” is actually an “expression”. In Objective Caml, it is more appropriate to report an “expression coverage”. This kind of source code coverage checks if all Objective Caml expressions written in the source code are evaluated at least once. An expression coverage tool must show in coverage reports which expressions are not evaluated in the source code. These expressions are called “non-covered expressions” (or dead code), and evaluated expressions are called “covered expressions”.

ZAMCOV uses debug events and execution traces to check which expressions in the source code are covered or non-covered. Debug events contain information to associate bytecode instructions addresses to their corresponding source code. So,

We define F and G recursively as the functions that respectively place debug events in expressions and pattern-matching branches.

- P represents a pattern-matching branches set, unfolded as $p_i [\text{when } c_i] \rightarrow e_i$ which is a shortcut for $p_0 [\text{when } c_0] \rightarrow e_0 \mid \dots \mid p_n [\text{when } c_n] \rightarrow e_n$;
- p or p_n represent a pattern, which is a variable identifier or a structural accessor;
- e, e_n or c_n represent an expression;
- and $\$ \$$ represents a debug event.

$$F(\text{atom}) = \text{atom} \text{ (constant value or identifier)}$$

$$F(e_0 \ e_1) = (F(e_0)) \ (F(e_1)) \ \$ \$$$

$$F(\text{let } [\text{rec}] \ p = e_0 \text{ in } e_1) = \text{let } [\text{rec}] \ p = F(e_0) \text{ in } F(e_1)$$

$$F(\text{fun } p_0 \dots p_n \rightarrow e) = \text{fun } p_0 \dots p_n \rightarrow \$ \$ F(e)$$

$$F(\text{function } P) = \text{function } G(P)$$

$$F(\text{match } e \text{ with } P) = \text{match } F(e) \text{ with } G(P)$$

$$F(\text{try } e \text{ with } P) = \text{try } F(e) \text{ with } G(P)$$

$$F(e_0; \ e_1) = (F(e_0); \ \$ \$ F(e_1))$$

$$F(\text{if } e_0 \text{ then } e_1 \ [\text{else } e_2]) = \text{if } F(e_0) \text{ then } \$ \$ F(e_1) \ [\text{else } \$ \$ F(e_2)]$$

$$F(\text{while } e_0 \text{ do } e_1 \text{ done}) = \text{while } F(e_0) \text{ do } \$ \$ F(e_1) \text{ done}$$

$$F(\text{for } i = e_0 \text{ to } e_1 \text{ do } e_2 \text{ done}) = \text{for } i = F(e_0) \text{ to } F(e_1) \text{ do } \$ \$ F(e_2) \text{ done}$$

$$F(e_0 \# m) = F(e_0) \# m \ \$ \$$$

where m represents the name of a method

$$G(P) = p_i [\text{when } F(c_i)] \rightarrow \$ \$ F(e_i)$$

Figure 2. Debug Events Placement in Expressions

to report source code coverage, for each debug event, if its associated bytecode instructions have been activated according to the execution traces, then its associated source code is covered. If a debug event cannot be related with the execution trace, it means that the associated expression is non-covered.

```
ocamlc -g fact.ml -o fact
zamcov-run -trace fact.trace fact
zamcov-cover fact.trace fact
```

This time, we compile the .ml file with option “-g” given to the standard ocamlc compiler to generate debug events for zamcov-cover to produce the source code coverage. Then, as for machine code coverage, with zamcov-run we interpret and generate the execution trace of the program fact and we build the coverage report with zamcov-cover.

Coverage report	
Trace Filename	Program Name
fact.trace	fact
Source Filename	Expression coverage
fact.ml	85% 6/ 7
Object code coverage report	
fact	

There is on the left an expression coverage report generated by ZAMCOV of a simple Objective Caml program (factorial). This report shows the source code files list of the coverage-measured program. Each file has a expression coverage rate which shows the difference between the number of non-covered debug events and the total number of debug events. For each file, there is a link to its source code coverage. There is also an HTML link to the machine code coverage of the program.

ZamCov: Expression Coverage (fact.ml)
<pre> let rec fact x = if x = 0 then 1 else if x = 1 then 1 else x * (fact (x-1));; fact 5;; </pre>

Link “fact.ml” allows to obtain its source code coverage page. In the screenshot on the left, text highlighted in green (light gray) is executed code (covered), and in red (dark gray) is code that has never been executed (non-covered).

ZamCov: Expression Coverage (fact.ml)
<pre> let rec fact x = if x = 0 then 1 else if x = 1 then 1 else x * (fact (x-1));; fact 5;; fact 0;; </pre>

$x = 0$ is never executed when calling fact 5, that is why the first 1 is non-covered. The structural coverage of this example is not complete. For function fact to be fully covered (expression coverage), we need to add more tests as shown in the screenshot on the left.

6 Related and Future Work

6.1 Other coverage tools in Objective Caml

The official Objective Caml distribution provides two profiler tools. The first one – for bytecode programs – instruments original source and counts each computed expression. The second one modifies the native code generator to produce information which can be used by gprof [9].

Ocamlviz [16] is a new graphical tool for real-time profiling in Objective Caml. It uses alarms to collect and send data. These data can be processed by a graphical interface during execution.

These different tools cannot produce MC/DC report, the only MC/DC coverage tool for Objective Caml is MLCOV that we described in section 2.2. All these tools are intrusive.

6.2 Other coverage tools for different virtual machines

For more classical virtual machines, as the Java Virtual Machine (JVM) or the Common Language Runtime (CLR) of the .NET environment, we find a lot of libraries to build debug tools. They offer a set of services which exposes runtime events that occur during the execution. In Java, JVMTI (Java Virtual Machine Tool Interface) ¹¹ allows to write agents which can be notified of interesting occurrences through events. In .NET, the CLR Profiling API can provide notification of many activities within the CLR and managed code.

A good overview describing different ways to instrument Java code is presented in [8]. This bibliographical study compares different static and dynamic instrumentation techniques at source or bytecode level, including hybrid combinations, for Java. Examples using a specialized virtual machine are scarce, mainly for portability and efficiency criteria which can be important for monitoring or optimizing tools.

In our case, portability is guaranteed because we use the same runtime with and without execution traces. The loss of performance efficiency with ZAMCOV is acceptable for this kind of tools.

6.3 MC/DC for ZAMCOV

ZamCov: Expression Coverage (abs.ml)

```
let abs x =
  let y = ref 0 in
    if x < !y then y := -x;
    !y;;
abs (-5);;
```

The next objective of ZAMCOV is to offer a Decision, Condition and MC/DC measurement tool. ZAMCOV will need to identify Boolean expressions evaluation at run-time. Notably, a complete *statement* coverage is not equivalent to a decision coverage. In the example on the left, all the statements are covered but the decision only takes value false.

ZAMCOV needs to analyse these values to generate a report that shows in the source code which decisions satisfy MC/DC and which don't. The operation needs to go back to the source.

The main issue is that we need to recognize Boolean expressions in the machine code, and this is not possible without specific source code analysis information. Indeed, as the machine code is untyped, it is not possible to know the difference between an integer and a Boolean value, it is neither possible to know in all cases if a branch is introduced by a conditional expression or by a Boolean operator, or even a pattern-matching filter.

Adding new debug events requires the modification of Objective Caml's bytecode compiler. Indeed, we need to be able to identify &&, || and not Boolean operations in the source code and link them with the machine code to produce Boolean vectors at run-time.

¹¹ JVMTI has replaced the JMVPI (JVM Profiler interface) [13] and JVMDI (JVM Debug Interface).

7 Conclusion

In this paper, we have presented a new approach for structural code coverage analysis without code instrumentation but only runtime environment instrumentation. This approach has been used to build ZAMCOV, a tool dedicated to Objective Caml's virtual machine. Our criteria of success is to produce the same reports as MLCOV, the open source code coverage measurement tool developed by Esterel Technologies for their certifiable code generator (KCG) written in Objective Caml. It will be reached for expression coverage (statement coverage) without any change in the Objective Caml compiler. But for MC/DC coverage, it will be mandatory to add new debug information systematically around Boolean expressions to check the condition/decision coverage to produce traces which can be analysed to measure the MC/DC coverage.

This approach can be used for any compiler that generates ZAM code with the appropriate events, if need be. This indicates a strong link between the compiler schemes and the debug events to map back to the original source.

It can be surprising to associate Objective Caml and bytecode for safety-critical software development tool. But the Esterel experiment has opened this way by using Objective Caml in a complete certification process. The introduction of virtual machine to build certifiable development tool is interesting for its non-intrusive approach: real code is analysed and not an equivalent but instrumented code.

This work takes place in the Coverage Project which studies non-intrusive coverage tools for Ada (to Power-PC) and Objective Caml (to ZAM). In the first case the QEMU emulator is used and in the second the ZAM virtual machine. But the compiler information needed by the modified runtime environment for the MC/DC measurement are similar for both languages.

Finally this work makes the link between two communities: DO-178B world and free open source software, by building the first part of a non-intrusive structural coverage tool. It joins the effort for openDO ¹², towards a cooperative and open framework for the development of certifiable software.

References

- [1] *MLcov*, <http://www.algo-prog.info/mlcov>.
- [2] *Qemu documentations*, <http://www.qemu.org>.
- [3] *The DWARF Debugging Standard* (2007), <http://dwarfstd.org>.
- [4] Berry, G., *The Effectiveness of Synchronous Languages for the Development of Safety-Critical Systems*, Technical report, Esterel-Technologies (2003).
- [5] Camus, J.-L. and B. Dion, *Efficient Development of Airborne Software with SCADE Suite™*, Technical report, Esterel-Technologies (2003).
- [6] Canou, B., V. Balat and E. Chailloux, *O'Browser : Objective Caml on Browsers*, in: *Proceedings of the 2008 ACM SIGPLAN Workshop on ML The 2008 ACM SIGPLAN Workshop on ML*, 2008, pp. 69–78, <http://www.pps.jussieu.fr/~canou/obrowser/tutorial/>.
- [7] Clerc, X., *Cadmium* (2007), <http://cadmium.x9c.fr>.

¹² <http://www.open-do.org/>

- [8] Delahaye, M., *Instrumentation of Java code : bibliographical study (in French)* (2007), <ftp://ftp.irisa.fr/local/caps/DEPOTS/BIBLIO2007/biblio.delahaye.mickael.pdf>.
- [9] Graham, S. L., P. B. Kessler and M. K. McKusick, *gprof: a call graph execution profiler* (1982).
- [10] Hayhurst, K. J., D. S. Veerhusen, J. J. Chilenski and L. K. Rierison, *A Practical Tutorial on Modified Condition/Decision Coverage*, Technical report, NASA/TM-2001-210876 (2001).
- [11] Leroy, X., *The ZINC Experiment : an Economical Implementation of the ML Language*, Technical Report 117, INRIA (1990).
- [12] Leroy, X., *The Objective Caml system release 3.10 : Documentation and user's manual*, Technical report, Inria (2008), <http://caml.inria.fr>.
- [13] O'Hair, K., *The JVMPI Transition to JVMTI* (2004), <http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition>.
- [14] Pagano, B., O. Andrieu, B. Canou, E. Chailloux, J.-L. Colaço, T. Moniot and P. Wang, *Certified Development Tools Implementation in Objective Caml*, in: P. Hudak and D. S. Warren, editors, *Practical Aspects of Declarative Languages (PADL 08)*, Lecture Notes in Computer Science **4902** (2008), pp. 2–17.
- [15] Pagano, B., O. Andrieu, T. Moniot, B. Canou, E. Chailloux, P. Wang, P. Manoury and J.-L. Colaço, *Experience Report: Using Objective Caml to Develop Safety-Critical Embedded Tools in a Certification Framework*, in: *International Conference of Functional Programming (ICFP 09)*, 2009.
- [16] Robert, J. and G. V. Tokauski, *Ocamlviz : reference manual* (2009), <http://ocamlviz.forge.ocamlcore.org>.
- [17] RTCA/DO-178B, *Software Considerations in Airborne Systems and Equipment Certification* (1992), Radio Technical Commission for Aeronautics RTCA.