Experience Report: OCaml for an Industrial-Strength Static Analysis Framework

Pascal Cuoq * Julien Signoles
CEA LIST, Software Reliability Labs,
Boite 65, 91191 Gif-sur-Yvette Cedex, France
First.Last@cea.fr

with Patrick Baudin, Richard Bonichon,
Géraud Canet, Loïc Correnson,
Benjamin Monate, Virgile Prevosto,
Armand Puccetti

Abstract

This experience report describes the choice of OCaml as the implementation language for Frama-C, a framework for the static analysis of C programs. OCaml became the implementation language for Frama-C because it is expressive. Most of the reasons listed in the remaining of this article are secondary reasons, features which are not specific to OCaml (modularity, availability of a C parser, control over the use of resources…) but could have prevented the use of OCaml for this project if they had been missing.

Categories and Subject Descriptors D1.1 [*Programming techniques*]: Applicative (Functional) Programming

General Terms Design, Languages, Verification

1. Introduction

Frama-C is a framework that allows static analyzers, implemented as plug-ins, to collaborate towards the study of a C program. Although it is distributed as Open Source, Frama-C is very much an industrial project, both in the size it has already reached and in its intended use for the certification, quality assurance, and reverse-engineering of industrial code. Frama-C is written in OCaml, and this article reports on the most noticeable consequences of this choice on the human (section 2) and technical (section 3) levels, as well as providing an overview of the implementation of Frama-C (section 4).

2. Human Context

Frama-C is developed collaboratively between the ProVal team (a joint laboratory of INRIA Saclay Île-de-France and LRI) and CEA LIST. This article describes our (CEA LIST) own analysis of this collaborative development. The Open Source nature of the software and other partnerships involving CEA LIST mean that Frama-C is in fact developed at three different sites, by around ten full-time programmers, with infrequent inter-site face-to-face meetings.

2.1 Recruiting OCaml programmers

It is typical for an article of this nature to include a few words to the effect that it is harder to find people who can program in functional language Y (Minsky 2007; Nanavati 2008) than in C++, sometimes nuanced by more words pointing out that this is balanced by the higher quality of Y candidates. The first proposition did not apply for us in the case of Frama-C and OCaml. CEA LIST is an applied research laboratory that recruits almost exclusively PhDs. When the choice is restricted to candidates with a PhD in the field of formal methods, it is not harder to find a candidate with the motivation to program in OCaml than in C++.

2.2 Objectives of the Frama-C project

Although it is developed by research institutes, Frama-C tries to fulfill focused, specific needs expressed by industrial partners. It aims past the R&D departments and into the hands of the engineers who develop embedded code in any industry with criticality issues[1]. Frama-C is structured as a kernel to which different analysis plug-ins are connected. It is composed as a whole of 100 to 200 thousands of lines of OCaml[2]. All this OCaml code provides a wide range of functionalities, but the plug-ins do not just sit side by side. It is more accurate to think of them as built on top of each other. To give an example, a value analysis plug-in computes supersets of possible values for the variables of the program (Canet et al. 2009), indexed by statement of the original program. Unions, structs, arrays and casts thereof are handled with the precision necessary for embedded code (Cuoq 2008). These values, especially the values of pointers and expressions used as indices in arrays, are used by another plug-in to compute synthetic functional dependencies between the inputs and the outputs of each analyzed function. These synthetic dependencies are in turn used by a slicer plug-in which produces simplified, compilable C programs that are guaranteed to be equivalent to the original for the slicing criterion. And the building blocks of this slicer are used by one of Frama-C's most sophisticated plug-ins to date, a security-aware slicer that preserves the confidentiality of information: the (functional) confidentiality is guaranteed to be exactly the same in the sliced program as in the original program (Monate and Signoles 2008). This means that it is safe to study functional confidentiality on the (smaller) sliced program, for instance for a security audit of the source code. This would not be possible with a traditional slicer, because a traditional

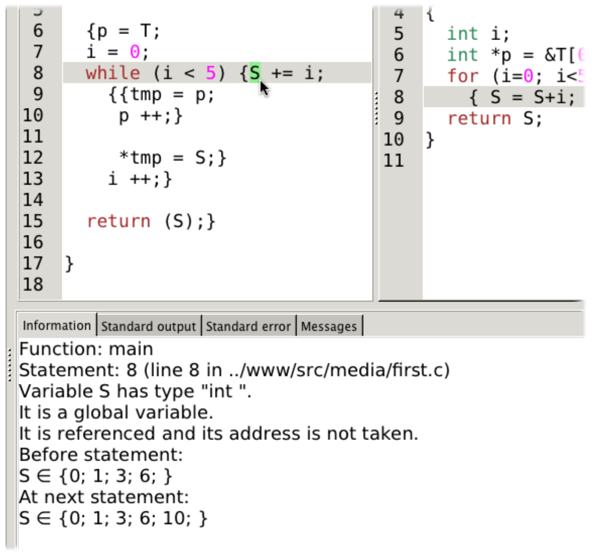
* This work has been supported by the french RNTL project CAT ANR05RNTL00301

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.
Copyright © 2009 ACM 978-1-60558-332-7/09/08…$5.00

[1] or even without criticality issues, but so far the industrial interest has come from critical embedded systems

[2] The command `wc ‘find . -name \*.ml -o -name \*.mli‘` reports 220000 lines as of this writing, which include comments, the CIL and ocamlgraph libraries, and some testing scripts

slicer might remove information leaks — in particular, a malicious programmer could insert information leaks that he knows the traditional slicer used for the audit will remove.



```
    {p = T;
    i = 0;
    while (i < 5) {S += i;
      {{tmp = p;
       p ++;}

       *tmp = S;}
      i ++;}

    return (S);}

}
```

```
{
  int i;
  int *p = &T[
  for (i=0; i<
    { S = S+i;
  return S;
}
```

```
Function: main
Statement: 8 (line 8 in ../www/src/media/first.c)
Variable S has type "int ".
It is a global variable.
It is referenced and its address is not taken.
Before statement:
S ∈ {0; 1; 3; 6; }
At next statement:
S ∈ {0; 1; 3; 6; 10; }
```

Figure 1. Frama-C's value analysis displaying results in the GUI

2.3 History of Frama-C

At the Software Reliability Labs, Frama-C was initially seen as a major evolution to Caveat (Baudin et al. 2002; Randimbivololona et al. 1999; Delmas et al. 2008), a software verification tool for C programs based on Hoare logic. Caveat is used by Airbus for part of the verification of part of the software embedded in the A380. As the DO-178B standard mandates, Caveat has been qualified by Airbus as a verification tool to be used for the certification of this particular software. Caveat is supported by the Software Reliability Labs but new ideas are now developed within Frama-C.

OCaml was pushed as the implementation language to choose for Frama-C by the new hires, but the actual reason it was accepted is that OCaml was not completely unheard of to the senior researchers there. Indeed, OCaml was already used in the (predominantly C++) Caveat project as the scripting language that allows an interactive validation process to be re-played in batch mode.

3. Technical context

3.1 Expressivity

OCaml's expressivity was crucial in the adoption phase of the language. An initial one-person internal prototype was able to produce results that convinced management to extend the experiment to two programmers. Eventually, this prototype was able to persuade industrial users to get involved in a national project.

To be precise, industrial partners agreed to be part of the project because of their previous experiences with the providers of static analyzers in the project. Some of the tools that they were familiar with were written in OCaml (Caduceus by the ProVal team), and some of them weren't (Caveat). The time it takes to build these relationships should not be underestimated, and we are not saying that the choice of any programming language can shorten it.

However, the progress made by Frama-C after the project had started, which can at least partly be attributed to OCaml, convinced the industrial participants to become involved beyond expectations. At each phase of the bootstrap process, OCaml's expressivity was important in quickly writing the proof-of-concept that took the project to the next stage.

3.2 Control over the use of resources

One of Frama-C's first plug-ins was a value analysis based on abstract interpretation. This plug-in computes supersets of possible values for expressions of the program. Among other things, these over-approximated sets are useful to exclude the possibility of a run-time error. In contrast with the heuristic techniques used in other static analysis tools, which may be very efficient but solve a different problem, shortcuts do not work when the question is to correctly — that is, without false negatives — find all possible run-time errors in large embedded C programs. The analysis has to start from the entry point of the analyzed program and unroll function calls (and often loops). In addition, the modular nature of Frama-C and the interface that the value analysis aimed at providing to other plug-ins meant that abstract states had to be memorized at each statement, which dictated the choice of persistent data structures, with sharing between identical sub-parts of states (Cuoq and Doligez 2008). This meant at the very least that a garbage-collected language had to be used. While there are popular imperative languages that are garbage-collected nowadays, and some of these languages have huge libraries of data structures ready to use, persistent data structures are often under-represented in these libraries, and are slightly annoying to write in these languages. For writing a static analyzer, one is not worse off with OCaml and a few select libraries (Conchon et al. 2008; Filliâtre and Conchon 2006) than with any of Python, the .NET framework or the Java platform, although it is by no means impossible to write static analyzers in any of these.

By contrast, in the development of Caveat in C++, there were explicit deallocation functions, and sanity checks that warned at the end of a session if deallocation had been forgotten for some nodes. Programming time that could have been spent usefully was spent writing the calls to the deallocation primitives and the source code's readability was diminished, but this is neglectable compared to the time that had to be spent debugging (the warnings told the developper that the deallocation had been forgotten, not where it should have been done). The solution to a subtle deallocation problem was often to make copies of existing nodes (although it is not meaningful to compare the memory consumption of Caveat to that of Frama-C's value analysis, because they work on different principles).

To conclude, in Frama-C, garbage collection paradoxically enables a tighter control of memory usage than explicit deallocation because it makes sharing possible in practice. OCaml is a strict language. We do not know what the influence of lazy evaluation would be on memory use.

3.3 Existence of CIL

CIL (Necula et al. 2002) is an OCaml library that provides a parser and Abstract Syntax Tree (AST)-level linker for C code. CIL is well documented and provides ready-made generic analyses that are very useful to get a prototypal analyzer started. Unlike the earlier mentioned data structures that are nice to find in OCaml but could be written quickly if they were missing, having to write a C parser when one's goal is to write a static analyzer would be a major time-sink. It would have been a significant counter-argument to the choice of OCaml if such a library had not existed. We were probably a little bit lucky, and OCaml is certainly at a disadvantage with respect to other languages from this point of view. We weren't perhaps lucky to find a C parser so much as to be working in a field that is also of interest to academia. So many researchers in software engineering use OCaml for their experiments that despite the amount of work involved, such a parser could be expected to get written someday. On the other hand, finding a library, or even bindings to an existing library, for an OCaml project in a field that does not interest students and researchers could be a problem.

This drawback (less developers implies less available libraries) is mitigated by higher code re-usability when you do find code to re-use, the existence of the Caml Hump[3] and the fact that the grapevine works well between OCaml developers. OCaml does not have anything that begins to compare for instance with CPAN, the Comprehensive Perl Archive Network, but it does have a healthy community.

3.4 Portability

It would have been an annoyance if the development platform for Frama-C had allowed it to be used only on Unix variants, because many potential users only have access to Windows stations. Unix being the system used by the majority of the researchers at the Software Reliability Labs, the switch to a Windows-only platform was not considered. Motivated users — and at this time actually deploying formal methods requires motivation anyway — have found their way past this limitation for previous projects developed at the Labs.

From this point of view, the choice of OCaml (and later of GTK+ as the toolkit for the graphical interface, through the lablgtk bindings) was an excellent compromise, with Unix clearly being the primary platform of the compiler, and Win32 robustly supported through either Cygwin or Visual C++. Compiling a large OCaml project on Windows+Cygwin is slow. This is probably caused one way or the other by the use of Unix compilation idioms (configuration script, makefile) on an OS where things are usually done differently, and is not a limitation in this context.

It should be noted that many OCaml developments are referenced in the source distribution GODI, with dependency lists and automated compilation scripts. All of those Frama-C dependencies that are written in OCaml are referenced in GODI, and Frama-C itself is. Some Frama-C users who have no interest in OCaml outside of Frama-C have found that this was the most convenient installation path for them.

Frama-C has been tested under Windows, Mac OS X, Solaris (32-bit), OpenBSD and Linux. Binaries are also distributed for some of these platforms.

64-bit readiness A 64-bit address space is available to OCaml programs for many 64-bit platforms. Frama-C compiles in both 32- and 64-bit mode, and the resulting versions are functionally identical. This was not a big effort, as OCaml encourages to write high-level code, for instance by providing bignums. For some 64-bit-aware platforms (Mac OS X), it is a simple `configure` option to choose between 32-bit or 64-bit pointers at the time of compiling OCaml. For others, it is troublesome to go against the default size (Linux). With Linux, getting an OCaml compiler with a word size different from the distribution default is akin to building a cross-compiler. However, efforts are under way in the OCaml community to improve support for cross-compilation, including from Linux to Win32. We are looking forward to the maturation of such initiatives.

Availability of a graphical toolkit Frama-C uses the GTK+ toolkit for its graphical user interface. This section does not discuss the merits or demerits of this toolkit with respect to others. The question it tries to answer is “If I choose OCaml for a software project, will I find one satisfactory toolkit to design the user interface with?”. The choice of GTK+ for Frama-C was somewhat arbitrary, but it allows to give a positive answer to the question above, without prejudice to other available toolkits.

Our experience is that for some Unix variants (Solaris, Mac OS X, very old Linux distributions), it is necessary to obtain and compile the missing GTK+ libraries manually, or semi-manually with the help of source distribution systems such as GARNOME or MacPorts. In this case, retrieving and installing the dependencies of the gtksourceview1 library (a GTK+ widget for displaying source code) is a pain. This is not directly an OCaml problem, but another development platform could have made it possible to use the modern gtksourceview2 (which solves the dependencies problem) or provided more toolkits to choose from initially (to the best of our knowledge, OCaml only offers Tk and GTK+ at this time). Now that gtksourceview2 has become stable, there is talk on the lablgtk development list about including it in lablgtk. This is anyway a very minor quibble. It should be kept in mind for comparison that Java or .NET/Mono do not come pre-installed on every platform either. Again, we have no reason to regret the choices of OCaml and GTK+ from the standpoint of portability.

OCaml as a scripting language It should be noted that while this is not its strongest point, OCaml is an acceptable scripting language. In other words, when OCaml is chosen as the main language for a new project, the project may be saved the introduction of additional dependencies towards various dedicated scripting languages down the road. For instance, the HTML pages of the Frama-C web site are processed with the yamlpp preprocessor[4], which is written in OCaml. For comparison, in its 15 years of development, Caveat had at one point accumulated dependencies towards Perl, Python, Bash and Zsh (in addition to C++, and in addition to OCaml, used for journalizing and re-playing). Some of these dependencies have since be removed, by replacing some tools with OCaml equivalents. Whatever the main language, discipline is obviously the foremost factor in avoiding “dependency creep”.

3.5 Module system

OCaml’s module system (Leroy 1996) has direct advantages: it creates separate namespaces and, when the modules are in separate files and interfaces have been defined, fully type-checked separate compilation. It is easy to underestimate the importance of these features in the management of a big project because they make the compiler transparent, but when they are missing, their absence is unpleasantly noticeable. We discuss these, and the (theoretically more interesting) functor system per se.

Separate compilation With OCaml, in bytecode, separate compilation has the same meaning as everywhere: compilation is parallelizable and only modified files need to be recompiled, with a quick final link phase. With native compilation, all the ancestors of the modified modules in the dependency graph must be recompiled, and the compilation of two files with a parenthood relationship can not be parallelized. Depending on the structure of an OCaml project, recompilation after an incremental change in a low-level module may sometimes feel longish, but in truth, it is much faster to recompile Frama-C with `ocamlopt` than to recompile Caveat with `g++`.

The existence of two OCaml compilers, one with blazingly fast compilation in general, the other with acceptable recompilation time and producing reasonably fast code, allows very short modify-recompile-test cycles. Again, it is easy to take short recompilation times for granted but with other languages, when a software project grows in size, this can sometimes be lost, and sorely missed.

The OCaml compiler tries very hard not to get in the way between a programmer and his program, and it does not force the programmer to write interfaces. However, if the interface `m.mli` is missing for module `M`, the compiled interface is generated from `m.ml`. This means that any change to `m.ml` changes the compiled interface and forces the recompilation of every module that uses `M`, even in bytecode. In a large project, modules should always have

[3] The Caml Hump an is informal central repository for OCaml libraries

[4] `http://www.lri.fr/~filliatr/yamlpp.en.html`

interfaces, if only for the sake of separate compilation. OCaml has an option to generate automatically the interface `m.mli` that exports everything from `M`.

Separate namespaces for compilation units Orthogonally to separate compilation, but as importantly for big projects, OCaml's module system provides separate namespaces. Better yet, the compiler option `-pack` allows to group several compilation units into a namespace. As a consequence, compilation units that have been put into different packs may safely use the same name for a type, variable or module. For instance the types `tree` in files both called `m.ml` in packs `lib1` and `lib2` are seen as `Lib1.M.tree` and `Lib2.M.tree`.

This feature is very useful for libraries because libraries may use very common filenames (`util.ml`) with the guarantee that there will not be a clash at link-time for users of this library (on condition that the pack name itself is unique).

In Frama-C, plug-ins are independent from each other: each plug-in only interfaces with the Frama-C kernel, and does not see the implementation details of other plug-ins. In order to implement this separation, the Frama-C system automatically packs each plug-in. Thus, two different plug-ins may use files with identical names and still be linked together within Frama-C.

Interfaces and functors The possibility to write functors (modules that are parameterized by other modules or functors), introduced before objects (at the time of Caml Special Light), has proved a workable, and completely statically checked, alternative to object-oriented programming. We use OCaml objects only when interfacing with existing OCaml code that uses objects (CIL and lablgtk), and use functors for the rest.

Some very structural idioms seem destined to be expressed with objects (or, for that matter, class types): equality, pretty-printing, hashconsing or marshaling functions[5]. Most of our data structure are complicated enough that automatically produced pretty-printers or equality functions would not fit the bill. Consequently, it is in our case neither more nor less tedious to write modules (and interfaces) that sport, in addition to a type `t`, functions such as `pretty: Format.formatter -> t -> unit` and `equal: t -> t -> bool`, and for unmarshaling values of a hashconsed type, `rehash: t -> t`. But, speaking of equality, it should be noted on the other hand that OCaml's polymorphic comparison functions (including =, >= and even ==) are dangerous pitfalls. The type-checker does not complain when they are applied wrongly instead of, for instance, `equal` above.

In OCaml, the module system allows to encapsulate the definitions of data structures, and in particular to give a purely functional interface to a sophisticated data structure that uses mutable values internally for optimization. With this compromise, the amount of stateful information that the programmer has to keep in mind is limited by the module boundaries, and the implementation's algorithmic complexity may be better than that of all known pure implementations. Some in the OCaml community call such an impure module "persistent" (Conchon and Filliâtre 2007). In fact, some positive reports on the industrial use of Haskell (Nanavati 2008) resonate deeply with our own programming experience, except that we attribute to OCaml's module system the advantages attributed there to Haskell's purity.

3.6 Labels and optional arguments

OCaml allows to use labels for function arguments. This feature does not make anything possible that was not already, but in practice, labels provide a concise way to remove the risk of confusion when a function takes several arguments of the same type with no obvious normal order between them. The only language that we know of with a feature vaguely similar to OCaml's labels is Objective C's infix notation for function calls.

Syntax begets style. The "OCaml style" is to write pure functions unless an exception needs to be made because the syntax *rewards* the use of immutable definitions, as seen in the following:

```
let x = 2 in ... x ...
let x = ref 2 in ... !x ...
```

We argue that using labels rewards consistent naming schemes in a similar fashion. When it is common for an argument to be passed repeatedly as-is from caller to callee without any computations actually happening to it (and in a persistent setting as much of Frama-C is, this happens with a lot of arguments), the labels syntax rewards the consistent choice of a unique label and eponymous variable name for this argument by a very concise syntax. In this example, the function `f` is being defined and calls functions `g` and `eval`.

```
let f ~mode ~env x y  =
  let context = ... in
  ... g ~mode ~context (x+y) ...
  ... eval ~mode ~context ~env ...
```

If the programmer deviates from this style by using different label names or variable names for `mode`, `context`, or `env`, he receives a gentle slap on the wrist in the form of the awkward `~context:computation_context` syntax. This changes the way of reading labels-enabled OCaml programs, too. The reader can put more trust in the names of variables, without having to look for context all the time. The level of obtrusiveness of the label syntax is exactly the same as with the definition of mutable values, and it is exactly right, too. Good style is encouraged but the system can be circumvented when it needs to.

Optional arguments (a syntax for giving a labeled argument a default value if it is omitted) are convenient when the consequences of the omission (and subsequent use of the default value) are visible and traceable (for instance, to provide a toolkit interface that is both powerful and beginner-friendly). It is in general a bad idea to use an optional argument to add a new mode to an existing function, both because of all the existing calls to this function — that the compiler would be glad to help the programmer inspect if s/he did not use an optional argument — and because of all the calls to be written in the future where the optional argument will be omitted by accident.

4. Development of Frama-C

There are a number of features in Frama-C's architecture that any Frama-C developer must be aware of. The goal of this section is not to provide a complete list — which can be found in the Frama-C plug-in development guide (Signoles 2008) — but to give a mildly technical overview of each interesting one, with reference to the OCaml feature(s) that make its implementation possible.

4.1 Software architecture

The software architecture of Frama-C is plug-in-oriented. This architecture allows fine-grained collaboration of analysis techniques (as opposed to the large-grain collaboration that happens when one technique is used in a first pass and another in a second pass). As a consequence, mutual recursion between plug-ins must be possible: a plug-in A must be able to use a plug-in B that uses A. A Frama-C plug-in is a packed compilation unit (see Section 3.5) but, unfortunately, OCaml does not support mutually-recursive compilation units. This problem is circumvented pragmatically by using references to functions placed in a module that all plug-ins are allowed to depend on.

[5] In the presence of hashconsing, not only do you have to write your own unmarshaling functions, but they are extremely tricky to get right

This “central directory” module is called db.ml and a snippet of it may look like:

```
/* db.ml: kernel database of plug-in stubs */
module Plugin1: sig val run: (unit->unit) ref end
module Plugin2: sig ... end
```

During its initialization, a plug-in registers each of its exported functions in the appropriate stub in Db — another OCaml feature is that each compilation unit can have its own initialization effects.

```
/* plugin1_register.ml */
let run () = ... (* the analysis goes here *)

(* registration of [run] in the kernel *)
let () = Db.Plugin1.run := run
```

Thought this solution is the most common way to break mutual recursion between compilation units, it has trade-offs. Firstly, polymorphic functions may not be registered this way. This is not an issue here: each plug-in is a static analyzer, and none of the analyzers we have written so far wanted to provide polymorphic functions. Secondly, the types of the registered functions have to be known by the Frama-C kernel. Here again, that is not a big issue in our context, especially because Frama-C encourages the use of ACSL (Baudin et al. 2008), a common specification language, as the lingua franca to transmit knowledge between plug-ins. Finally, the most significant trade-off with this solution is that any plug-in that wishes to provide an interface to other plug-ins (as opposed to interacting with the user only) needs to modify some well-identified parts of the Frama-C kernel. This has not been a problem so far because, for now, plug-ins written outside Frama-C’s development team have all been dedicated to answering a specific problem, as opposed to providing computations to help other plug-ins.

4.2 Dynamic loading of plug-ins

OCaml has allowed dynamic linking of bytecode compilation units (through module Dynlink) for a long time. In OCaml 3.11, dynamic linking of native code became available for a large number of target architectures.

Frama-C uses dynamic linking where available in order to provide dynamic loading of plug-ins. This is an alternative way to plug analyzers into the Frama-C kernel. When dynamic linking is used for the plugging, the plug-in’s functions are registered in a global table in the kernel at load-time. Because all functions do not have the same ML type, phantom types (Rhiger 2003) are used in order to dynamically ensure the program’s safety (see Section 4.6). Dynamic linking solves two out of three issues of static linking: it ceases to be necessary for the kernel to be aware of the types of all plug-ins’ exported functions, and it becomes more convenient to distribute a plug-in separately from Frama-C (in particular a plug-in no longer needs to patch the kernel).

4.3 Impure functional programming

Most analyses in Frama-C are written in a functional style. However Frama-C’s value analysis (whose results are used by many other plug-ins) relies on hashconsing (Filliâtre and Conchon 2006) and memoization, which are both implemented with mutable data structures. More generally, Frama-C makes use of imperative features in order to improve efficiency. For instance, the abstract syntax tree (inherited from CIL) contains many mutable fields. Besides, Frama-C has a global state which is composed of many global tables.

4.4 Multi-project framework

Frama-C is able to handle several ASTs simultaneously. This allows to build slicing plug-ins where an original AST is navigated through while a reduced AST is being built. Each of these ASTs has its own state (containing for instance the results of the analyses that have been run on this AST). The AST and corresponding state form what is called a project (Signoles 2009). The desirable “safety property” of projects is the absence of interference between two distinct projects. To enforce this property, each global mutable value of Frama-C must be “projectified”. A set of functors are provided to this effect (these functors add a project-aware indirection to any mutable data that is used by any of the functions made visible by the plug-in). We wish OCaml’s type system helped us enforce this rule, but we plan to move to dynamic tags to detect at least at analysis-time when a variable that should have been projectified wasn’t.

4.5 Journalization

During an interactive session, Frama-C journalizes most of the actions which modified its global state. This means that, like Caveat, it generates an OCaml script retracing what happened during the session. The journal may be compiled and statically or dynamically linked with the Frama-C kernel in order to replay the same actions. Furthermore, the journal can be used to grasp Frama-C’s internals (translating GUI actions into function calls), and it is possible to modify it before compiling and replaying it. As for dynamic loading, phantom types allow to safely implement this feature (see Section 4.6).

4.6 Phantom types for dynamic typing in a static setting

Both dynamic loading and journalization rely on phantom types (Rhiger 2003). Phantom types — parameterized types which employ their type variables for encoding meta-information — are used in both cases to ensure the dynamic safety of function calls which cannot be checked by the OCaml type system. Indeed we provide a library of dynamic typing. Its implementation requires the use of unsafe features (through OCaml standard library’s module Obj) but phantom types allow to provide a safe interface: the use of the library cannot break type safety (as long as there is no implementation error in the library).

5. Conclusion

We have not yet considered the point of view of the external Frama-C plug-in developer. We hope to see in the future many useful plug-ins written outside the circle of the initial developers. It is too early to draw conclusions on the consequences of the choice of OCaml as the platform’s language for this goal. Responses so far have ranged from the enthusiastic (“and it’s even written in OCaml”) to the rejection (“[...]drawback that the extensions have to be written in Ocaml[sic]”), with in the middle at least one person who decided to learn OCaml because there was something s/he wanted to do with Frama-C.

Acknowledgments

We would like to acknowledge the help of our colleagues at ProVal, at INRIA Sophia Antipolis’ projects Everest and now Marelle, and at CEA LIST, in the building of Frama-C. The feedback of users of Frama-C within the CAT project, at Fraunhofer FIRST or elsewhere has been great. The anonymous referees suggested various improvements to this experience report. Special thanks go to the developers of the OCaml system.

Keywords OCaml, software architecture, plug-ins, static analysis

References

Patrick Baudin, Anne Pacalet, Jacques Raguideau, Dominique Schoen, and Nicky Williams. Caveat: a tool for software validation. In *Dependable Systems and Networks, 2002*, pages 537+, 2002.

Patrick Baudin, Jean-Christophe Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI C Specification Language (preliminary design V1.4)*, preliminary edition, October 2008. URL `http://frama-c.cea.fr/acsl.html`.

Géraud Canet, Pascal Cuoq, and Benjamin Monate. A value analysis for C programs, 2009. To appear in the proceedings of SCAM2009.

Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In *ML '07: Proceedings of the 2007 workshop on Workshop on ML*, pages 37–46, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-676-9. doi: http://doi.acm.org/10.1145/1292535.1292541.

Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a generic graph library using ML functors. In Marco T. Morazán, editor, *Trends in Functional Programming*, volume 8 of *Trends in Functional Programming*, pages 124–140. Intellect, UK/The University of Chicago Press, USA, 2008. ISBN 978-1-84150-196-3.

Pascal Cuoq. Documentation of Frama-C's value analysis plug-in, 2008. URL `http://frama-c.cea.fr/download/frama-c-manual-Lithium-en.pdf`.

Pascal Cuoq and Damien Doligez. Hashconsing in an incrementally garbage-collected system: a story of weak pointers and hashconsing in OCaml 3.10.2. In *ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 13–22, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-062-3.

David Delmas, Stéphane Duprat, Patrick Baudin, and Benjamin Monate. Proving temporal properties at code level for basic operators of control/command programs. In *4th European Congress on Embedded Real Time Software*, 2008.

Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 12–19, New York, NY, USA, 2006. ACM. ISBN 1-59593-483-9.

Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6:1–32, 1996.

Yaron Minsky. Caml trading: Experiences in functional programming on Wall Street. In Wouter Swierstra, editor, *The Monad.Reader*, April 2007.

Benjamin Monate and Julien Signoles. Slicing for security of code. In Peter Lipp, Ahmad-Reza Sadeghi, and Klaus-Michael Koch, editors, *TRUST*, volume 4968 of *Lecture Notes in Computer Science*, pages 133–142. Springer-Verlags, March 2008.

Ravi Nanavati. Experience report: a pure shirt fits. *SIGPLAN Not.*, 43(9): 347–352, 2008. ISSN 0362-1340.

George C. Necula, Scott Mcpeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.

Famantanantsoa Randimbivololona, Jean Souyris, Patrick Baudin, Anne Pacalet, Jacques Raguideau, and Dominique Schoen. Applying formal proof techniques to avionics software: A pragmatic approach. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II*, pages 1798–1815, London, UK, 1999. Springer-Verlag. ISBN 3-540-66588-9.

Morten Rhiger. A foundation for embedded languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(3):291–315, 2003. ISSN 0164-0925.

Julien Signoles. Plug-in development guide, 2008. URL `http://frama-c.cea.fr/download/plug-in_development_guide.pdf`.

Julien Signoles. Foncteurs impératifs et composés: la notion de projets dans Frama-C. In *Actes des Journées Francophones des Langages Applicatifs*, pages 37–54, January 2009. In French.