From Static Analysis to Runtime Verification with Frama-C and E-ACSL

Julien Signoles

ORSAY
N° d'ordre : 2097

UNIVERSITÉ DE PARIS-SUD 11

CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

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

PAR

Julien Signoles

—×—

SUJET :

From Static Analysis to Runtime Verification with Frama-C and E-ACSL

soutenue le 9 juillet 2018 devant la commission d'examen

Klaus	Havelund	senior researcher	NASA/JPL	rapporteurs
Marie-Laure	Potet	professeure	ENSIMAG	
Mihaela	Sighireanu	maître de conférence, HDR	Université Paris Diderot	
Wolfgang	Ahrendt	professor	Chalmers University	examinateurs
Sylvain	Conchon	professeur	Université Paris Sud	
Gilles	Dowek	directeur de recherche	Inria, ENS Paris-Saclay	
Xavier	Leroy	directeur de recherche	Inria	
Claude	Marché	directeur de recherche	Inria	

From Static Analysis to Runtime Verification with Frama-C and E-ACSL

Julien Signoles

start of writting: 2016/9/2
end of writting: 2018/2/28

Acknowledgements / Remerciements

Acknowledgements

First, I would like to thank my reviewers Klaus Havelund, Marie-Laure Potet, and Mihaela Sighireanu who accepted to read this memoir and write a report about it.

I am also grateful to the jury president Sylvain Conchon, as well as the jury examiners Wolgang Ahrendt, Gilles Dowek, Xavier Leroy, and Claude Marché. Your various questions were interesting and I really enjoyed answering them. Claude also helped me in the different administrative steps from the start to the end: thank you also for this support.

In addition to the official reviewers, four persons also proofread this memoir and helped me to improve it a lot: Zaynah Dargaye was my first reader, but also my first supporter by regularly asking me my progress. Jean-Christophe Filliâtre, and Florent Kirchner also proofread intermediate versions, while David R. Cok helped me to improve the final version. Many thanks to each of you!

Remerciements

Je passe maintenant dans ma langue maternelle pour des remerciements plus larges.

Cela fait maintenant douze années, depuis 2006, que je suis au LSL. Je voudrais profiter de cet espace pour remercier ses membres de contribuer chaque jour à faire de ce laboratoire un lieu dans lequel il est agréable de ~~grignotter des gourmandises aux pauses cafés~~ travailler.

Tout particulièrement, je voudrais remercier les contributeurs principaux à Frama-C, à commencer par l'«équipe noyau» actuelle – Virgile Prévosto, François Bobot, David Bühler, Loïc Correnson, et André Maroneze –, mais aussi Patrick Baudin, Valentin Pérelle et Boris Yakobowski, ainsi que ses contributeurs historiques Benjamin Monate, Pascal Cuoq et Anne Pacalet. Sans eux, cette plateforme ne serait pas ce qu'elle est, dans toute sa complexe simplicité.

Je voudrais également remercier Nikolai Kosmatov avec qui je collabore très régulièrement depuis plusieurs années maintenant, Fonenantsoa Maurica, Viet-Hoang Le, et Dara Ly que j'ai la chance d'encadrer en ce moment et Michele Alberti, Gergö Barany, Kostyantyn Vorobyov, Mounir Assaf, Guillaume Petiot, Arvid Jakobsson, Quentin Bouillaguet, Antonin Butant et Kharam Youness Kharraz que j'ai récemment encadrés. Par nos riches échanges et vos contributions, chacun d'entre vous apporte, ou a apporté, sa pierre (voire plusieurs) à mes développements aussi bien scientifique que personnel.

Par ailleurs, je souhaite également saluer l'extraordinaire aide quotidienne apportée par Frédérique Descréaux pour soulager chacun, au DILS, des contraintes administratives et logistiques, même dans des situations apparemment désespérées.

Enfin, merci à ma famille pour son soutien continu.

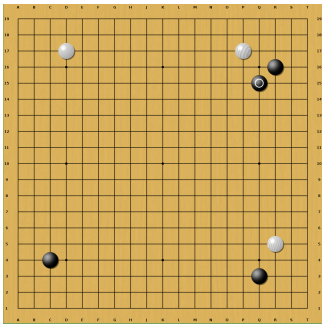
Saclay, le 7 septembre 2018

Contents

List of Figures

1
Introduction

Fuseki (Opening)



Shusaku fuseki invented by Honinbo Shusaku (1829–1862).

I got a PhD in Computer Science from University Paris 11 in 2006 [Sig06]. This document presents my main research activities that have followed. It should be readable by everyone who got a Masters in Computer Science and followed at least one introductory course in software formal methods. Indeed it is my research area. In computer science, formal methods is [1] a set of techniques based on logic, mathematics, and theoretical computer science which are used for specifying, developing and verifying software and hardware systems. By relying on solid theoretical foundations, formal methods is able to provide strong guarantees, so it is of primary importance for critical systems whose a failure could lead to dramatic consequences like deaths, as well as economical or environmental collapses. Among formal methods, I am particularly interested in software verification techniques that focus on verifying software code *after* it has been written and even compiled. Sometimes one also names this set of techniques *a posteriori* verification, in opposition to *a priori* correct-by-construction techniques which aim to

1. I am not a native English speaker: my English writing is unfortunately certainly far from being perfect, so there are certainly English mistakes in this document despite my efforts. At least this agreement is fine according to NASA (`http://shemesh.larc.nasa.gov/fm/fm-is-vs-are.html`).

derive correct programs from specifications. They also do not include programming language-based techniques, such as typing.

1.1 Applied Software Verification

Even if formal methods in general and software verification in particular is more and more successfully used in critical industries all around the world [Bou12b, Bou12a], their adoptions are still the exception rather than the rule for several reasons, including their ignorance by engineers and managers, their difficulty of use, the changes they often require in system life cycles and their costs. Being expensive is almost a consequence of the other reasons. Ignorance can be solved by teaching even if it certainly takes decades. Simplifying their uses and making them compliant to current system life cycle require to improve formal method based *tools*.

My research activities are tools oriented: I aim at developing cutting-edge tools containing the most innovative software verification techniques implemented in the most effective way to help industry verify and validate their software and systems efficiently. Yet I have no pretension to develop such tools alone. In 2006, after getting my PhD, I moved to the Software Reliability and Security Laboratory, LSL for short, at CEA LIST. Since 2006, all my research activities have been done in this research laboratory in close relationship with the other engineer-researchers of the team [2], in accordance with CEA LIST and LSL strategy which can be synthesized in a single sentence: take the best academic results, put them in efficient tools and transfer them to the industry, first in France, then in Europe, later all around the world. Coincidentally and luckily this strategy matches my personal scientific goal.

1.2 Developing a Code Analyses Framework

When joining LSL, a tool had been emerging there for about one year: Frama-C [CKK+12, KKP+15, CCK+]. I have been continuously contributing to this tool ever since. It aims at providing several analyzers of C code in a single collaborative and extensible framework. *Collaborative* means that analyses can collaborate with each others to solve together a particular task, while *extensible* means that everyone can extend the framework with new analyses [SAC+]. Yet it was more a wish than a reality in 2006. A few months after joining LSL, I began to modify and extend the Frama-C kernel in order to transform this young promising tool into an industrial-strength collaborative extensible code analysis framework [CSB+09]. I devoted most of my time from 2007 to 2011 and several additional months from 2012 to 2015 to this foundational task. In particular I designed and implemented its software architecture [Sig15], different ways of providing analyzer collaborations [CS12], and

2. Several of them will be explicitly named all along this document.

several kernel libraries [Sig09, Sig11, CDS11, Sig14], which implement extensively used services all along the Frama-C codebase. Some of them require innovative programming techniques. During that time, I have also developed a few small plug-ins in order to demonstrate the extensibility of the framework. Chapter 2 is dedicated to this part of my work.

1.3 Runtime Verification

In 2011, I began to be interested in *runtime verification* and more particularly in *online* runtime verification. Online runtime verification is a lightweight software verification technique that consists in checking formal properties at runtime, that is when the analyzed program is being executed. It does not provide as strong guarantees as formal static analyses because it does not check property validity for every possible program execution but only for a few of them. However, it is automatic and easily usable by any software engineer, so less expensive than other formal methods, while still being able to check non-trivial program properties on particular executions of interest. It is indeed a way to introduce formal techniques and tools in traditionally reluctant applicative domains.

My work on runtime verification focuses on an online monitor generator named E-ACSL and implemented as a Frama-C plug-in [KS13, SKV17, SV]. It converts a C program p extended with annotations written in a formal specification language also named E-ACSL [Siga] into a new program p' which inlines an embedded monitor m_p: the program p' functionally behaves as the original program p, but fails at runtime whenever the inlined monitor m_p detects that a property denoted by an annotation does not hold [DKS13]. E-ACSL aims at being compliant with both the Frama-C ecosystem and C intricacies. First, its specification language is a conservative subset of Frama-C's ACSL formal specification language [BFM+] that includes all the constructs that are translatable to C code. Second, it must deal with logical constructs that are usually considered to be complicated to handle at runtime, like mathematical integers [JKS15b], memory-related properties (*e.g.* validity of pointers and initialization of memory locations) [KPS13a, JKS15a, VSK17, VKSJ17], and specifications that are undefined at runtime (*e.g.* division by zero, or out-of-bound array index access). Chapter 3 is dedicated to E-ACSL.

1.4 Information Flow Analysis

I initially joined LSL in 2006 on a one-year postdoc position. My research goal was to design and implement an information flow analysis for Frama-C. I first planned to benefit from the abstract interpretation front-end of Frama-C by implementing an abstract interpretation based taint analysis in order to mark (or "taint")

every memory location with a security label (*e.g.* public or secret). However, I was quickly confronted with important Frama-C limitations. First, it was not possible at that time to develop a new analyzer without modifying Frama-C itself. Second, it was not possible to implement this taint analysis without duplicating most efforts already done to develop the main abstract interpreter of Frama-C, namely Value [CYL+]. Trying to fix the first issue has been the starting point of my efforts to improve the Frama-C kernel. Circumventing the second issue eventually led to the PhD of Mounir Assaf [Ass15] that I supervised in collaboration with Éric Totel and Frédéric Tronel at CentraleSupélec Rennes. Mounir designed a program transformation that weaves the information flows inside the source code in a way that checking their validity is equivalent to checking a standard E-ACSL assertion [ASTT13b, ASTT13a]. Consequently, every standard verification technique can be used to check them, including runtime verification through E-ACSL or other static ones provided through other Frama-C analyzers (*e.g.* Value). Mounir developed a prototype Frama-C plug-in named Secure Flow. After Mounir's PhD, I pursued this work through the supervision of the 18-month postdoc of Gergö Barany, who contributed to maturing Secure Flow [Bar16, BS17]. I do not dedicate a full chapter to this topic in this document, though Section 3.4.2 provides a few details about this work.

1.5 Support

Financing more than ten years of Research and Development activities required fundings in the current economical model of French research institutes in general and CEA LIST in particular. During these years, I have participated in numerous academic research projects supported by national or European funding and a few industrial bilateral projects directly supported by industrial partners.

1.5.1 Academic Projects

***Vessedia*,** Europe, H2020, 2017-2019, with Dassault Aviation, Fraunhofer Fokus [3]
Verification Engineering of Safety and Security Critical Dynamic Industrial Applications: improving and proposing new analyses collaborations [4].

***S3P*,** France, PIA, 2015-2018, with Thales, TrustInSoft
Smart and Secured Platform: improving E-ACSL.

***ARVI*,** Europe, ICT Cost Action, 2015-2018, with most European academic researchers in runtime verification

3. I only indicate the partners which I have personally collaborated with.
4. I indicate both the global goal of the project and the action(s) I have contributed to.

Runtime Verification beyond Monitoring: participating in the European community of researchers in runtime verification.

***AnaStaSec*,** France, ANR, 2015-2018, with Airbus, Inria
Static Analysis of Security Properties: improving Secure Flow.

***Aurochs*,** France, DGA Rapid, 2015-2017, with TrustInSoft
Source Code Analyzers for Cyber-Security: improving Frama-C capabilities on security-oriented libraries and improving E-ACSL.

***Chekofv*,** United States, Darpa, 2012-2015, with SRI International, University of Santa Cruz
Crowd Sourced Formal Verification: developing dedicated plug-ins which help to improve Frama-C capabilities on analyses collaborations.

***Stance*,** Europe, FP, 2012-2015, with Dassault Aviation, Fraunhofer Fokus, Thales
Source Code Analysis Toolbox for Software Security Assurance: developing Secure Flow.

***Hi-Lite*,** France, FUI, 2010-2013, with Adacore, Inria
High Integrity Lint Integrated with Testing and Execution: creating the E-ACSL formal specification language and developing the E-ACSL plug-in.

ADS+ , France, FUI, 2010-2012, with Atos Worldline, Gemalto
Opened and Secured Architecture for POI: adapting Frama-C to banking security-oriented applications.

***U3CAT*,** France, ANR, 2008-2011, with Airbus, Dassault Aviation, Inria
Unification of Critical C Code Analysis Techniques: improving the Frama-C kernel, in particular services related to combination of analyzers.

***€-Confidential*,** Europe, ITEA, 2006-2009, with EADS, Gemalto, VTT
Trusted Security Platform to secure multiple kinds of application and to provide a trustworthy execution environment: designing Frama-C's very first information flow analysis.

***OpenTC*,** Europe, FP, 2006-2009
Open Trusted Computing: designing and developing the Security Slicing plug-in of Frama-C.

***PFC*,** France, DGE, 2006-2009, with EADS, Gemalto, Inria
Trustworthy Platform: designing and developing the Impact analysis plug-in of Frama-C.

***Cat*,** France, ANR, 2005-2008, with Airbus, Dassault Aviation, Inria
Toolbox for Analysis of C Programs: developing the Frama-C kernel, in particular its software architecture and several services.

1.5.2 Industrial Projects

joint lab CEA LIST – Thales, 2015-2016 & 2018

2018 *applying E-ACSL on numerical programs.*

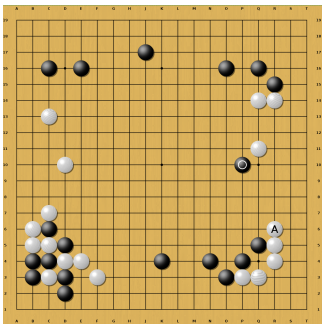
2015–2016 *providing expertise in formal specifications and formal methods.*

joint lab CEA LIST – TrustInSoft, 2014-2015

designing and developing two dedicated Frama-C plug-ins, one of them being Cfp.

2
Frama-C
a Framework for Analyses of C Code

Chuban (Middle Game)



AlphaGo 's extraordinary move 37 starts the chuban
(Game 2 of the match AlphaGo – Lee Sedol, 2016/3/10).

Frama-C, born in 2004, is a free open source scalable extensible collaborative plug-in-based kernel-centred framework developed in OCaml, that provides analyzers for C99 source code annotated with ACSL specifications. Every single word of this sentence is important. They are explained in the subsequent sections of this chapter in order to give to the reader a journey into the Frama-C world, from its past to its possible future through its current main features, while highlighting my own contributions in the meantime.

2.1 Frama-C, a Framework Based on Software Verification History

Context This section is novel, but takes ideas from Chapter 1 of Jean-Christophe Filliâtre's habilitation thesis [Fil11] and from Claude Marché's talk at Frama-C Day 2015 about the history of Frama-C. [1]

1. Most sections of this thesis are based on my own previous publications. A few are novel, but usually take inspiration from existing sources. To make things crystal clear, each section starts by

This section summarizes the main evolution of software verification up to the birth of Frama-C in 2004 in order to explain the historical foundations Frama-C is built upon.

2.1.1 Big Bang

A very long time ago, in the 1930s, there was nothing but Alonzo Church's lambda-calculus [Chu33], Alan M. Turing's machine [Tur36, Goo60], and a pair of negative results from Kurt Gödel [Gö31, vH76].

2.1.2 Invention of Writing

Then, right after a few years of chaos, from 1946 to 1948, Hermann H. Goldstine and John von Neumann wrote a report in two parts for the U.S. Army Ordnance Department. It may be seen as foundations of both hardware and software. Indeed the first part [BGvN46] is the first widely-circulated document about computers built upon what is now known as the von Neumann's machine [vN45], while the second part [GvN47] introduces the foundations of programming techniques [Knu70], refered as "methods of coding" by the report's authors. In particular, they introduce the well-known notion of *flowchart* as a way to describe programs. The vertices of flowcharts are named *boxes*. Interestingly they define only three kinds of them: *operation boxes* that correspond to computational expressions (*e.g.* $x + 1$, x being a bound variable), *substitution boxes* that are now known as variable assignments, and *assertion boxes*. Let me quote the authors about the latter.

> "It may be true, that whatever [the code] actually reaches a certain point in the flow diagram, one or more bound variables will necessarily possess some certain specified values, or possess certain properties, or satisfy certain relations with each other. Furthermore, we may, at such a point, indicate the validity of these limitations. For this reason we will denote each area in which the validity of such limitations is being asserted, by a special box, which we call an *assertion box*."

It is amazing to have such historical evidence that coding cannot go without specifying in the minds of the inventors of coding methodologies. In particular, one of their recommendations is to include such an assertion box after every loop: "At the exit from an induction loop, the induction variable usually has a (final) value which is known in advance, or for which a mathematical symbol has been introduced. [...] Hence this is usually the place for an assertion box". However, their methods of coding do not include program verification. Actually they follow a

indicating where its contents comes from.

correct-by-construction approach which consists in deriving correct code from its (mathematical) specification.

Nevertheless, program verification was clearly introduced as early as 1949 by Alan M. Turing who tries to answer this question [Tur49, MJ84] [2]:

> "How can one check a routine in the sense of making sure that it is right?"

As an example, he provides a rigorous mathematical proof of a program computing the factorial by repeated additions. The interested reader may refer to Jean-Christophe Filliâtre's habilitation thesis which contains nice and didactic explanations of Alan M. Turing's original proof [Fil11, Chapter 1, page 2].

However, at the time of these pioneering computer scientists, verifying a program was a pure mathematical activity which was pen and paper, as well as brain, consuming. It could be theoretically done for any program, but it suffers from the same problems as any mathematical proof: it requires mathematical skills, may be tedious, and may contain subtle hard-to-catch errors.

2.1.3 Birth of Monotheistic Religions

Removing these drawbacks requires more systematic approaches based on formal representations of programs. In 1969, Tony Hoare understood this necessity [Hoa69]: he built upon an earlier work of Robert W. Floyd [Flo67] to construct what is now known as *Hoare Logic* (sometimes also called *Floyd-Hoare Logic*) in order to "[evaluate] the possible benefits to be gained by adopting this approach both for program proving and for formal language definition".

However, Hugh G. Rice had already proved in 1953 that no automatic exact static analysis can verify any non trivial program property [Ric53]. Therefore some compromises are required in practice to prove programs. An idea is to relax (at least) one of the important constraints of Hugh G. Rice' statement. In the 1970s, it eventually lead to three different formal verification methods —*weakest precondition calculus*, *abstract interpretation* and *model checking*— in addition to program *testing* which relaxes the constraint of being computed statically and is known from the earliest days of programming.

Edsger W. Dijkstra's weakest precondition calculus [Dij75] (also known as WP calculus) may be seen as a computable version of Hoare Logic which computes the least constrained (or weakest) predicate which is sufficient to ensure that a given predicate is satisfied after executing a given program statement. Even if computable, it is not a fully automatic method, since it requires in practice to manually write loop invariants and loop variants of every loop of the program. Quoting Edsger W.

2. This work is only the oldest one than I am aware of and that refers to program verification. That does not necessarily imply that there are no older ones.

Dijkstra, "[the design] of a repetitive construct requires what [he] regard[s] as the "invention" of an invariant relation and a variant function". Such an "invention" is indeed challenging and arbitrarily hard, actually as hard as finding an induction hypothesis strong enough to establish a proof by induction in mathematics.

Patrick Cousot's abstract interpretation framework [CC77] relaxes the *exact* nature of the analyzer by computing a correct over-approximation of the program semantics. Therefore it is automatic but unconclusive whenever the approximation contains both a potential execution state that satisfies the property to be verified and a potential execution state that does not satisfy it. In this context, one challenge is to remain precise enough to be able to check the properties of interest, while relaxing precision enough to scale up.

Model checking, simultaneously introduced by E. Allen Emerson and Edmund M. Clarke [EC80] and Jean-Pierre Queille and Joseph Sifakis [QS82], substitutes the problem of verifying a program by the one of verifying a model, typically an automaton represented by a Kripke structure [Kri63]. Therefore, an important question is how to ensure code's correctness with respect to the proven model. Depending on the property and the code, the model is possibly automatically extractable from the code, but the well known state explosion problem may make this approach difficult to apply on large programs manipulating a large amount of data because of scalability issues.

At the beginning of the eighties, thanks to these seminal works, the theoretical foundations of software verification techniques were established. However, practical tools were still missing.

2.1.4 Industrial (R)evolution

Two additional decades were necessary to create the first industrial applications of software formal methods in general, and software verification in particular. The first significant industrial applications of software formal methods was the MéTéor project which was initiated in the beginning of the 1990s and terminated in 1998 [BBFM99, Bou12b]. It used the B method [Abr96] in order to build the automation system of the line 14 of the Paris's métro. The B method ensures correctness of the system by deriving the code from a high level specification, while guaranteeing its correctness. Nevertheless, it is not a *program verification* technique, since it performs *a priori* proof of correctness and no *a posteriori* one.

Let us come back to the three above-mentioned techniques of program verification. A hackneyed example of critical failure is the crash of the first Ariane 5 flight in 1996. I have no intention to explain this story one more time [3]. However, it remains of particular importance for program verification in general and

3. It is still possible to read the full report of this failure at `http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html`.

abstract interpretation in particular because the post-crash investigations allowed Alain Deutsch to discover errors in the Ariane 5's embedded code by means of an abstract interpretation tool. This tool became PolySpace [Deu04] when Alain Deutsch founded PolySpace Technologies in 1999. This company was acquired by The Mathworks in 2006. PolySpace still exists today. It is specialized in runtime error detection by over-approximating the possible behaviors of Ada programs (and also, nowadays, C and C++ programs).

In 2001, Airbus also decided to operationally use abstract interpretation techniques for its A380 program [SWDD09] in order to compute worst-case execution times thanks to the tool aiT [FHL+01] and upper bounds of the stack memory actually used by the program thanks to the tool Stackanalyzer [4]. Airbus also integrated two other abstract interpretation tools, namely Astrée [CCF+05] and Fluctuat [DGP+09]. The former is similar to PolySpace but is particularly efficient on (avionic) programs generated from Scade models. The latter verifies that the program parts using floating-point arithmetic can only generate negligible rounding errors. All these tools are still used at Airbus today.

Also in 2001, Airbus also decided to use WP calculus through the Caveat tool [RSB+99] for its A380 program. The Caveat project emerged in 1993 at LSL, CEA. The developpement of the tool really started in 1995. In 2001, Airbus transferred Caveat to the teams that developed A380 software in order to replace unit testing by *unit proof* [SWDD09] on C code: within the development process of the most safety-critical A380 program, unit proof was used for achieving most DO-178B objectives related to the verification of the executable code with respect to the Low-Level Requirements (LLR) which were written in the Caveat formal specification language. Since the A380 program, Caveat has been used in the same way for the A400M and A350 programs. It is currently being replaced by Frama-C [BDH+18].

Most industrial uses of model checking focus on hardware verification [GV08]; so they are outside of my topic. Here I present only one of the first and most significant successful industrial applications of software model checking, namely the SLAM project, that originated at Microsoft Research in early 2000 [BCLR04]. This project was used at Microsoft to automatically verify that Windows device drivers properly interact with the Windows kernel at the heart of the Windows operating system. It relies on the model checker Bebop [BR00] in order to detect whether or not a Boolean program reaches an error state. It is worth remembering that the success of this project does not rely on model checking only but also on other techniques, notably predicate abstractions, symbolic executions, and theorem proving, so it examplifies collaboration of analysis techniques.

But let us come back from Redmond, United States, to the *plateau de Saclay* near Paris, France. Indeed the LSL team is located here and so is the Caveat tool.

4. `http://www.absint.com/stackanalyzer/`

In 2001, at two kilometers from the LSL team, Jean-Christophe Filliâtre and Claude Marché from Inria also begin to develop a verification tool for C programs, named Caduceus [5]. It was based on Why which was a multi-language multi-prover verification platform [FM07]. Both Caveat and Caduceus implemented the same techniques for solving the same kind of problems, but had different advantages and drawbacks: the former benefited from its industrial usage but was hard to maintain and became outdated, while the latter was a cutting-edge tool but remained a prototype suffering from lack of manpower. In 2004, both teams decided to learn from the past experiences and joined their strengths in order to develop a new tool from scratch: Frama-C. Here we are. Figure 2.1 resumes the main periods and dates outlined in this section.

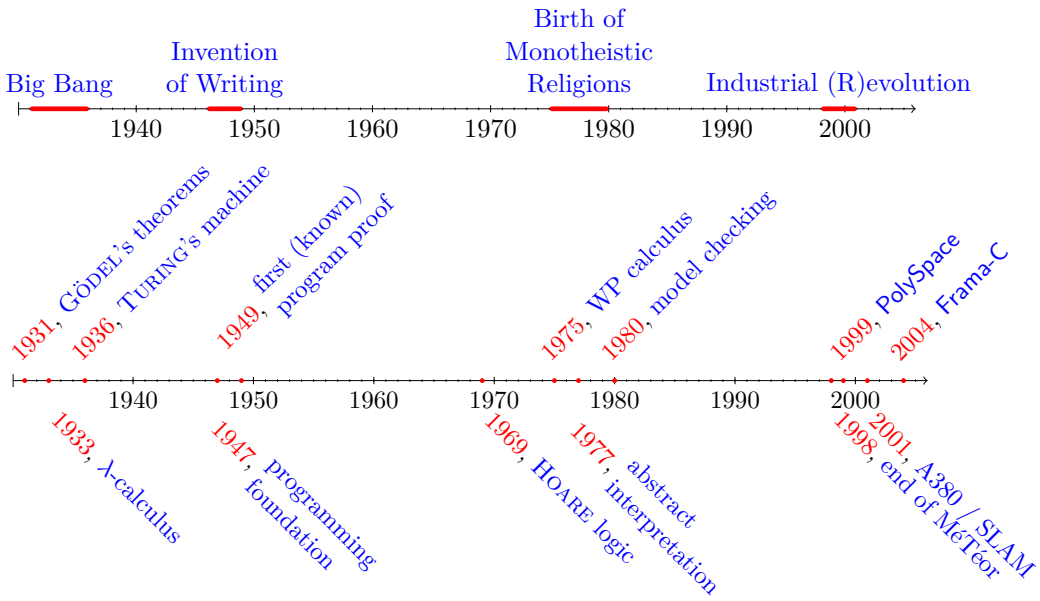


Figure 2.1: Software Verification Timeline up to Frama-C's Birth.

2.2 Frama-C, a Free Open Source Framework

Context This section is novel.

Frama-C is released under the GNU Lesser General Public License (LGPL), version 2.1. This license was carefully chosen before its first public release in 2008:

5. Maybe it is worth stating that I did my PhD under the supervision of Jean-Christophe Filliâtre in this team from 2002 to 2006, even if my PhD research was not related to C program verification.

this choice is of particular importance because it allows Frama-C to match CEA's dissemination objectives. Indeed being freely available under an open source license overcomes a few of the Caveat's limitations. Caveat is actually a non free closed source tool. That, combined with too few academic publications, prevents potential users to try it easily (a trial license is required), while its closed internal technology is almost unknown in academia.

After 10 years of free open sourcing of Frama-C, I think that this choice is a great success: today Frama-C has a wide community all over the world. Open sourcing really allows Frama-C to be easily tried by potential partners, while it makes easier to disseminate it in academia through publications and tutorials. It is worth noting that Frama-C was a CEA pioneer since it was the very first major free open source tool to be released by CEA. Its success has encouraged other tools from CEA LIST to be also open sourced, for instance Papyrus[6]. The choice of LGPL among the numerous open source licenses will be justified in Section 2.6.

2.3 Frama-C, a Scalable Framework

Context This section takes a few ideas from a few of my publications [CSB+09, CKK+12, KKP+15], but is also inspired by an article about CompCert's semantics [KLW14].

Frama-C analyzes C programs from their source code. The C programming language was created in 1972 by Denis Ritchie and Ken Thompson. It is defined by a norm that has had several evolutions since its first version ratified as ANSI X3.159-1989 and known as C89. The latest version of the standard is ISO/IEC 9899:2011 and is known as C11. Frama-C aims at being compliant with ISO/IEC 9899:1999, known as C99 because this standard is still the most widely adopted in the software industry. In the rest of this document, I always consider C99 programs, unless otherwise specified.

C is a general-purpose programming language that allows for close control over the machine and for high runtime efficiency. This made C among the most popular programming languages in the world[7]. In particular, for the above-mentioned reasons, most operating and embedded systems are still written in C and could hardly be written as efficiently in another language, including the most safety- and security-critical ones.

However, C is almost one of the most dangerous programming language because of unsafe constructs like casts and its absence of runtime checks: it is extremely easy for C programs to have bugs that make the program crash or behave badly in other

6. `https://eclipse.org/papyrus/`

7. C is the second most popular programming language in August 2017 according to TIOBE index (`https://www.tiobe.com/tiobe-index/`).

ways. Its semantics is particularly tricky. Several impressive efforts have been made to formalize it, either with pen and paper through abstract state machines [GH93] or monadic denotational semantics [Pap98], or in a mechanized way in HOL [Nor98], Coq [BL09], or $\mathbb{K}$ [ER12, Ell12].

Nevertheless, in program analysis, the semantics of the underlying programming language usually remains implicit. Indeed it is expressed in accordance with the program properties to be verified. For instance, abstract interpreters over-approximate the program semantics in their abstract domains, while weakest precondition calculi encode it in their models (*e.g.* arithmetic and memory models). Frama-C follows this approach and must respect the C99 semantics when implementing analyzers. Frama-C also aims at being usable on large industrial systems: it must handle the largest possible part of the standard while scaling up. In that respect, Frama-C also supports some non-standard extensions which are used by lots of pieces of code, or by a particular customer.

2.4 Frama-C, a Specification Framework

Context This section is inspired by the introduction of the ACSL manual [BFM+] and the introduction about ACSL in the Frama-C tutorial paper that I gave with Nikolai Kosmatov in 2016 [KS16]. It is also based on some of my recent teaching material.

The primary goal of Frama-C is to verify programs with respect to their specifications. These specifications are expressed either implicitly or explicity. Implicit specifications are directly encoded in the analyzer together with the program semantics. Absence of undefined behaviors [8] is certainly the most common implicit specification.

Explicit specifications must be provided as input to Frama-C. For this purpose, Frama-C analyzes not only C programs, but C programs annotated with ACSL specifications. ACSL [BFM+] is a formal specification language for C programs designed by LSL together with Inria. It is a behavioral interface specification language [Win87, HLL+12] that allows developpers to write code contracts, a concept originally introduced in Eiffel [Mey92b]: each function f may be annotated with preconditions and postconditions that enforce a contract between the callee f and its callers. If a caller g satisfies the preconditions when calling the function f, then the callee f must guarantee that the postconditions hold when leaving f for returning into g.

8. C99 standard defines an undefined behavior as "a behavior, upon use of nonportable or erroneous program construct or erroneous data, for which [C99] imposes no requirements" (p.16, §3.4.3).

ACSL is inspired by the specification languages of Caduceus and Caveat which both rely on contracts. The former was inspired itself by JML [LBR99]. Consequently, users who know both C and JML should be able to easily understand ACSL specifications. ACSL logic is a typed polymorphic first-order logic whose terms are pure (*i.e.* side-effect free) C expressions extended with specific keywords and built-ins to handle language specificities. For instance, `\result` denotes the result of a function (in a postcondition). Predicates may be (possibly inductive) user-defined ones, or built-ins. For instance `\valid` is a built-in predicate that holds if and only if its argument is a valid pointer, *i.e.* a non-null pointer that points to an address that the program is allowed to access.

Additional technical details about ACSL will be provided in Section 3.2, but let us introduce here an illustrative example based on the C function provided in Figure 2.2. It will be our companion for the rest of this chapter. This function implements Kadane's algorithm which solves the maximum subarray problem with an optimal linear complexity [Ben84]. In 2016, Claude Marché and Andrei Paskevich made me discover this algorithm that had been proven [9] in Why3 [FP13], the successor of the Why tool. I translated it in C and proposed it as an exercice to master students during a deductive verification training session.

```
int max_subarray(int *a, int len)  {
  int max = 0, cur = 0;
  for(int i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) cur = 0;
    if (cur > max) max = cur;
  }
  return max;
}
```

Figure 2.2: A C Implementation of Kadane's Algorithm.

Here is the specification of this function in natural language that I gave to my students. It is one possible definition for the maximum subarray problem.

Definition 2.1 (Maximum Subarray Problem) *A sub-array* b *of an array* a *is a subsequence of contiguous elements of* a. *For instance, if* a = { 0, 3, -1, 4 }, *some possible sub-arrays of* a *are* ∅, { 0 }, { 3, -1 }, { 0, 3, -1, 4 }.

A sub-array of a *is* maximum *if the sum of its elements is at least as big as the sum of any other sub-array of* a. *The unique maximum sub-array of the previous*

9. `http://toccata.lri.fr/gallery/maximum_subarray.en.html`

example is { 3, -1, 4 }. *Since 0-length sub-arrays are allowed, arrays that only contain negative values have a maximum subarray of sum 0.*

The function call max_subarray(a, len) *returns the sum of one maximum subarray of the array* a *of length* len.

From this informal definition (and two additional meaningful hints in order to help them but omited here), the students must first specify the function in ACSL, and then prove it. I will prove it later on, but Figure 2.3 provides a solution for the specification.

```
#include <limits.h>

/*@ axiomatic Sum {
  logic integer sum(int *a, integer low, integer high, integer len)
    reads a[low..high];

  axiom base: \forall integer low, high, len; \forall int *a;
    low > high ==> sum(a, low, high, len) == 0;

  axiom ind: \forall integer low, high, len; \forall int *a;
    0 <= low <= high < len ==>
    sum(a, low, high, len) == a[high] + sum(a, low, high-1, len);
  } */

/*@ requires len >= 0;
  @ requires \valid(a+(0..len-1));
  @ requires \forall integer l, h;
  @   0 <= l <= h-1 <= len ==> sum(a,l,h,len) <= INT_MAX;
  @ ensures \forall integer l, h;
  @   0 <= l <= h <= len ==> sum(a,l,h-1,len) <= \result;
  @ ensures \exists integer l, h;
  @   0 <= l <= h <= len && sum(a,l,h-1,len) == \result;
  @ assigns \nothing;
  @ */
int max_subarray(int *a, int len);
```

Figure 2.3: ACSL Specification of Kadane's Algorithm.

ACSL annotations are formatted as special comments with a leading `@` character. This way, they do not interfere with the C implementation; and they are not interpreted by standard tools for C programs, notably compilers.

The specification is constituted of two parts enclosed in separated comments: an axiomatic introduced by the keyword `axiomatic` and a function contract on top of the function declaration. The axiomatic describes the behavior of a logic function `sum` that computes the sum of the elements of the array `a` (represented by a pointer) of length `len`, between its indices `low` and `high`, by reading the contents of the array between these bounds. They are declared as `integer` which means mathematical integer. Indeed ACSL types includes C types but also `boolean`, `integer` and `real` as well as user defined types. Implicit coercion rules automatically promote C types to integer when necessary while explicit casts may be used to convert mathematical types to C types. I am going to discuss this design choice in Section 3.2.4. The logic function is inductively defined through an axiomatic: the axiom `base` defines the sum of 0-length arrays while the axiom `ind` defines the sum from `low` to `high` from the sum from `low` to `high-1`. Both logic functions and predicates may be defined either in an axiomatic way (like here), or by providing a (possibly recursive) definition. The former are better understood by some logic-based tools (*e.g.* Wp, see page 29), while the latters are better handled by tools that execute specifications, even just symbolically, as soon as they are able to deal with recursive definitions (*e.g.* StaDy, see page 27).

The function contract of this example contains three kind of clauses: `requires`, `ensures` and `assigns`. The first one introduces the function preconditions that the function callers must satisfy, the second one are the function postconditions that this function ensures if the preconditions are satisfied, while the third one indicates the memory locations that the function may modify: here `\nothing` means that the function is pure. Let me explain now the preconditions and postconditions. The two first preconditions indicate that the length `len` of the array must be non negative and that the pointer `a` (actually an array) must be valid between the indices 0 and `len-1`. The last precondition specifies that no sum of sub-arrays overflows. That is a necessary condition to guarantee the absence of integer overflows in this function [10]. The postconditions state that the result is greater or equal than the sum of any sub-array, and is necessarily the sum of one sub-array.

Even if introduced only through a single still-unproved example, I hope that this section has provided enough evidence that ACSL is a powerful formal specification language. In practice, it has proved powerful enough to specify the functional properties of interest of safety-critical code of our industrial partners.

2.5 Frama-C, an OCaml Framework

Context This section is an updated summary of the ICFP experience report that

10. Proving the absence of integer overflows is actually optional for the students because this condition is actually hard to infer from their limited experience.

I wrote with Pascal Cuoq in 2009 [CSB+09].

Frama-C is developed in OCaml. The choice of the programming language(s) for developing a tool which is expected to be maintained during decades should never be made lightly. For instance, one of the issues of Caveat is the difficulty to maintain it and make it evolve, because it was developed in C++ complemented by other additional languages (namely Awk, OCaml, Shell and Smalltalk). This choice has both human and technical impacts.

In 2004, OCaml was pushed as the implementation language to choose for Frama-C by the latest hires of the Caveat team. But the actual reason it was accepted is that OCaml was not completely unheard of to Caveat's senior researchers since it was already used in this tool as the scripting language that allows an interactive validation process to be automatically re-played in batch mode. OCaml was also the development language of both Caduceus and Why at Inria. Another important human factor is that our team mostly hires PhD candidates in formal methods. Most such candidates already know OCaml (at least in France where OCaml is taught to most students in mathematics and computer science) or another functional language. Even better, some of them are already expert programmers at the end of their PhD and our team is lucky enough to attract a few of them from time to time, despite the attractive power of *GAFAM*[11] and the existence of excellent academic research teams. I am fully persuaded that our daily use of OCaml is actually one of our main convincing arguments in that regard and could partially explain why our team is so lucky.

Switching from human factors to technical reasons, OCaml was also chosen because it is particularly suited to writing programs with lots of symbolic manipulations like compilers or program analyzers. But maybe the most important reason was the existence of Cil [NMRW02], an OCaml library that provides a parser and Abstract Syntax Tree (AST)-level linker for C code. Cil APIs are (mostly) well documented and provide ready-made generic analyses that are very useful to get a prototype analyzer started. It would have been a significant counter-argument to the choice of OCaml if such a library had not existed. The existence of LablGtk2, an OCaml binding for Gtk2, was also helpful for designing a Graphical User Interface (GUI) for Frama-C. Once again, it could have been an important counter-argument if no good graphical library had been available. However, it is worth noting that the absence of upgrade of this library to Gtk3 has now become more and more problematic for Frama-C and we are currently investigating solutions to replace the old LablGtk2 GUI by a new one.

Frama-C also benefits from almost every OCaml programming feature, including those which were not present when choosing Frama-C but were introduced in recent years. I do not provide additional details here, but technical arguments may be

11. Google, Apple, Facebook, Amazon and Microsoft.

found in a related paper [CSB+09], while some interesting features will be highlighted later on in this document. In short, I am sure that no researcher of the Frama-C team regrets the 13-year old choice of OCaml for developing Frama-C. In my opinion, it is largely enough to conclude that this was an excellent choice.

2.6 Frama-C, a Plug-in Based Framework

Context This section is based on the Frama-C tutorial paper written with Nikolai Kosmatov [KS16] and several introductary talks I gave about Frama-C. It is however more complete than these related materials.

Frama-C is not a single tool, but a framework which groups together several tools, most of which are code analyzers. Each tool is provided as a plug-in. The official open source distribution of Frama-C currently comes with 24 plug-ins [12]. A few other source plug-ins are also open source but distributed independently. One strength of Frama-C's LGPL license is to allow everyone to develop additional plug-ins under different licenses, even not necessarily open source. Indeed the license of each plug-in is carefully chosen according to our dissemination strategy. Usually they remain close source as long as we consider they are not mature enough for industrial experimentation. Then, we open source as many plug-ins as possible, in particular if they are not specific to a particular industrial usage. Figure 2.4 shows the gallery of LSL's Frama-C plug-ins.

To our knowledge, it is the most comprehensive snapshot of these plug-ins ever presented [13]. It highlights open source plug-ins (the circled ones) and their current maturity on a scale from 1 (used by at least one industrial partner) to 4 (still a research prototype) [14]. The figure also emphasizes my own contribution. A major contribution means that either I have developed most of the plug-in, or I have

12. The provided information about Frama-C is based on Frama-C Silicon-20161101.

13. Nevertheless, I voluntarly omit plug-ins which are now almost unmaintained, or which have currently no practical interest. Also I could have missed some interesting plug-ins that were developed very recently and on which no communications have been done yet.

14. The current most common tool for measuring maturity level is *Technology Readiness Level* (TRL) which defines a scale from 1 to 9 (see for instance `http://www.nasa.gov/pdf/458490main_TRL_Definitions.pdf`). But, I think that such a scale is far too precise, while it does not fit how R&D software is actually developed (at least in my working context). Anyway, in case of need, here is a possible correlation table (overlappings emphasize that there is actually no easy direct correlation):

local scale	TRL
1	6-9
2	4-6
3	3-5
4	1-3

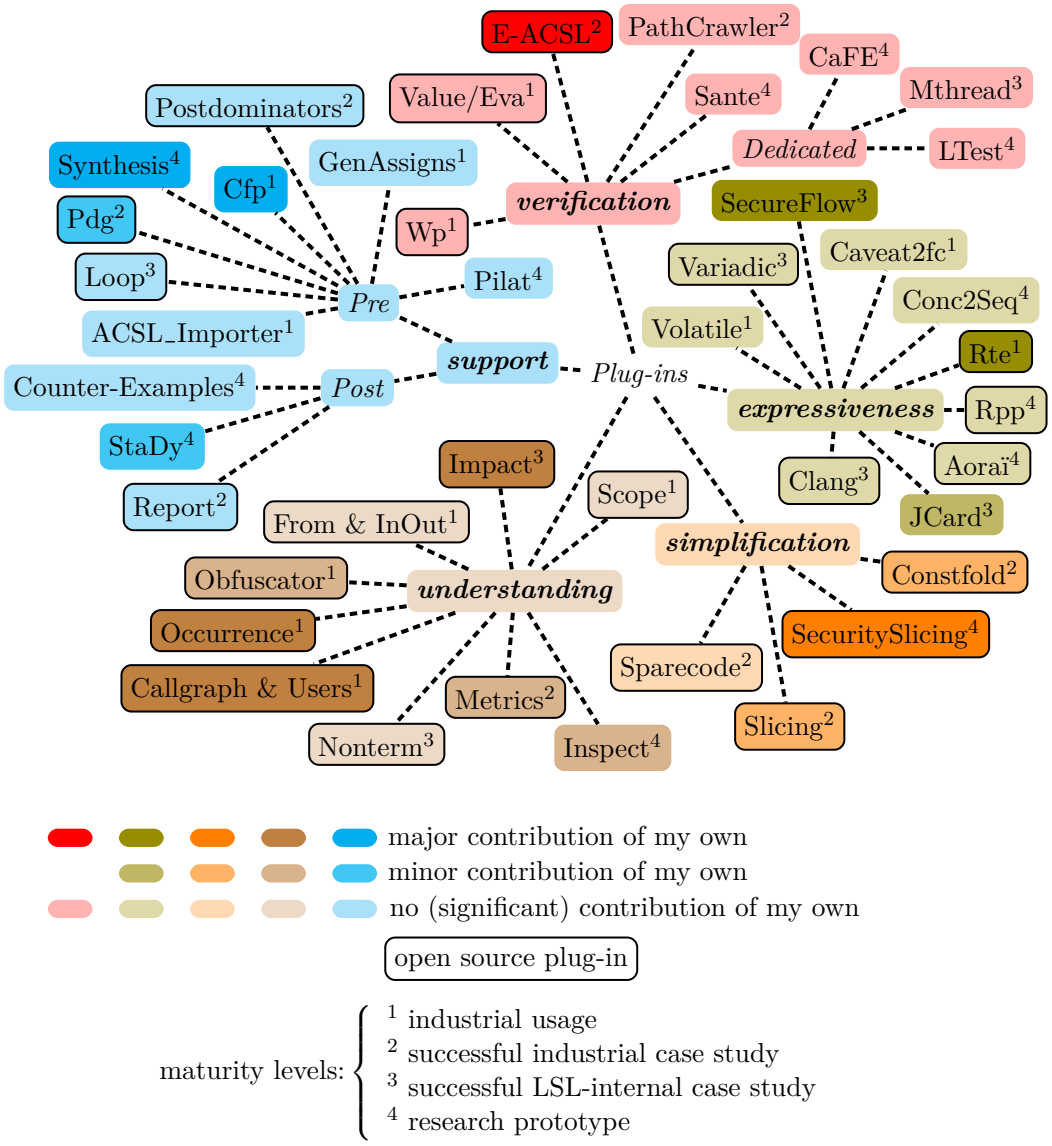


Figure 2.4: LSL's Frama-C Plug-in Gallery.

supervised the related R&D effort and reviewed the code. A minor contribution means that I have modified or extended a part of the plug-in, or provided expertise to its developer and reviewed some pieces of code. The plug-ins are organized in five main categories:

verification plug-ins verify program properties. The sub-category *dedicated* contain verification plug-ins which verify a specific kind of (non functional) property.

expressiveness plug-ins extend the variety of programs or properties that can be handled by the framework.

simplification plug-ins are program transformations that simplify the input code according to some criteria.

understanding plug-ins are plug-ins that help the verification engineer to understand a program.

support plug-ins are plug-ins that ease the use of verification plug-ins. They are split to *pre* and *post* support plug-ins. The former are used before a verification task, while the latter are used after a verification task in order to better understand the provided results.

I briefly introduce below each plug-in, in alphabetical order. This ordering has the drawback of introducing short forward references, but allows the reader to quickly search for the description of a particular plug-in. Indeed this plug-in enumeration should be seen as a dictionary in which pick up information about particular plug-ins. It could be a bit tedious to read it from the beginnning to the end. I also indicate the sizes of the plug-ins in number of lines of code. [15] All in all, they constitute a codebase of about 180 kloc and 110 kwloc in other languages (mainly C++ and Coq). That provides a numerical evidence of the LSL investment in the Frama-C ecosystem. My major contributions are about 20 kloc plus 5 kwloc of C, while my minor contributions concern about 22 kloc.

ACSL_Importer (2.3 kloc) allows the user to add ACSL annotations in external files in order to not modify the original C source code. A dedicated mechanism links these annotations to program points. This plug-in is used by industrial customers whose the normalized development process forbids any modification of source files (even for just inserting comments) at verification time.

Aoraï (8 kloc)[GS09, KKP+15, SP] takes as inputs a C program together with a Büchi automaton which specifies acceptable sequences of function calls. It generates a C program annotated with ACSL specifications that encodes the

15. The measurements are based on a Frama-C Silicon-20161101 compatible version and are in OCaml unless otherwise specified. 1 kloc is one thousand non empty lines of code (computed with the tool `ocamlwc`), while 1 kwloc is one thousand (possibly blank) lines of code (computed with the tool `wc`).

automaton in the input code. This way, any Frama-C verification tool may verify such temporal properties over function calls.

CaFE (6 kloc) [dOPB15] verifies temporal logic properties over C programs by model checking. They are expressed in the CaRet language [AEM04] (an extension of LTL [Pnu77]).

Callgraph and Users (0.6+0.3 kloc and 0.1 kloc respectively) respectively compute the callgraph of a C program and the function callees inside its functions. If Value has already been computed, function pointers are taken into account when computing the callgraph, while Users automatically runs Value if not already computed. I developed this later plug-in with Pascal Cuoq in 2009 in order to illustrate how easy reusing Value's results was (and still is). An interesting feature of the Frama-C callgraph (implemented in an additional 0.3 kloc outside the core plug-in) is its so-called services that group together functions that contribute to the very same functionality (*a.k.a.* service). Services are displayed in the Frama-C GUI. For instance, Figure 2.5 indicates the callgraph with its services for the program `gzip` (version 1.2.4). The detail is not important here but, displaying four large services as boxes (named `zip`, `unzip`, `treat_file`, and `unlzh`) helps the user to understand that they respectively contain functions for compressing, uncompressing, file manipulations and bit-level manipulations (even if the last name is not really helpful, the names of the functions inside the service — that the user can then inspect — are explicit).

This notion of service originates from Caveat. In this tool, its implementation was cubic in the number of vertices and its specification was unclear —to say the least— in particular in the presence of mutually recursive functions (cycles in the callgraph). In 2009, I formalized the specification and implemented a $O(n \log n \times e)$ algorithm, n being the number of C functions or, equivalently, the number of graph vertices, and e being the maximal number of callers for a C function.

I have never tried to publish this work because I would like to formally prove it first. Indeed several bugs have been found in the past, not directly in the implementation of the algorithm (only one bug found in a corner case as far as I remember) but in the OCamlGraph [CFS07] implementation of the topological traversal it relies on. Actually topological graph ordering usually assumes directed acyclic graphs (DAG) while cycles are possible in our case. Jean-Christophe Filliâtre and I eventually rewrote the OCamlGraph implementation from scratch to correctly deal with cycles. We also wrote its invariants to convince ourselves of its correctness. However, a formal proof of the topological traversal and of the Frama-C implementation of the service algorithm is still a challenge in the domain of the verification of higher-order

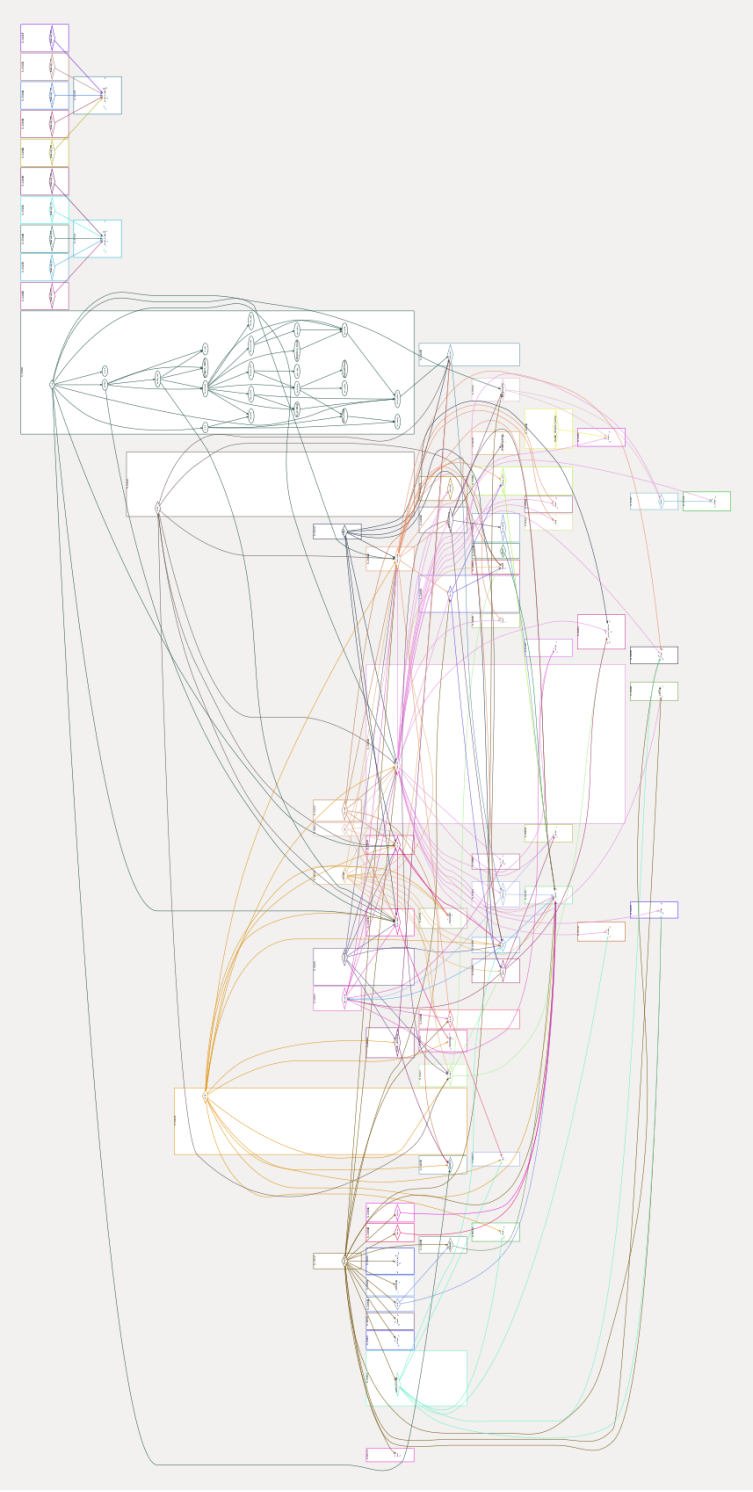


Figure 2.5: Callgraph and services for program `gzip` version 1.2.4. By default, the services only present their root function. The large service on the top is `zip` and has been unrolled: the internal vertices are functions that are actually functions only useful for function `zip`.

not-purely-functional programs. For the sake of completeness, the algorithm is presented in Appendix A.

Caveat2fc (4.6 kloc + 0.6 C-kwloc + 0.4 Coq-kwloc + 0.1 Why3-kwlocls) replaces annotations written in the Caveat specification language by equivalent ACSL annotations. It is particularly useful for industrial customers who used Caveat extensively and have switched to Frama-C.

Cfp (3.5 kloc)[AS17] automatically generates a function `main` from an ACSL function precondition. This function initializes the context of Value with respect to the precondition expressed in ACSL, while expressing it in the most understandable way both from Value and from a human point of view. It may be seen as solving a dual problem of functional synthesis [KMPS10], which aims at generating function bodies from postconditions. However, even if the formulation of both problems is dual, the desired properties and the proposed solutions are truly different. In particular, functional synthesis aims at generating executable function bodies, while Cfp's generated context aims at being executed symbolically and may contain non-deterministic inputs. This plug-in has been implemented in the context of the joint lab with TrustInSoft which has integrated it in its verification methodology. This work is *not* presented in this document.

Clang (7 kloc + 45 C++-kwloc) takes as input a C++ program and generates an equivalent C program. This way, every Frama-C plug-in may analyze C++ code. The majority of the development is actually a LLVM plug-in implemented in C++, not a Frama-C plug-in. The LLVM plug-in generates an Abstract Syntax Tree (AST) in an intermediate representation in a dedicated format which is then parsed and converted to the Frama-C internal AST from the OCaml side. The necessary information is kept in order to convert results and user messages emitted from Frama-C back to the original C++ code.

Conc2Seq (1.4 kloc + 0.1 C-kwloc) [BKLL16] transforms concurrent C programs into sequential ones in which concurrency is simulated by interleavings. This way, it allows other analyzers to handle concurrency for free.

Constfold (0.4 kloc) is a program transformation that replaces each constant expression — that is, expressions that evaluate to the very same value for every program execution — by that constant value.

Counter-Examples (1.8 kloc) tries to generate counter-examples from SMT solvers' counter-models. More precisely, when a prover fails to prove a proof obligation generated by Wp, this plug-in generates a function `main` that expresses the prover's internal candidate counter-example in term of program's global variables.

E-ACSL (6 kloc + 8 C-kwloc) [KS13, SKV17, SV] generates an instrumented program p' from an annotated program p. This program p' fails at runtime whenever

an annotation is violated, while it is functionally equivalent to p if every annotation is satisfied (for the given program execution). This runtime verification tool is one of my main contributions to Frama-C and is covered in depth in Chapter 3.

From and InOut (1 kloc and 1.2 kloc respectively) [CYL+, Section 6] computes different kinds of dependencies between function inputs and outputs by relying on Value's results.

GenAssigns (1.9 kloc) automatically generates candidate ACSL `assigns` clauses from a program in order to ease the task of the verification engineer. Such clauses indicate memory locations that may be modified by a function (more generally, a piece of code) and must be provided when proving code by Wp. Consequently this plug-in especially helps increase automation when using Wp in industrial settings.

Impact (1.4 kloc) computes statements whose semantics may depend on the semantics of a given statement. It relies on both Pdg (Program Dependence Graph)'s and Value's results. I initially designed and implemented this plug-in from my postdoc's plug-in Security Slicing in the context of the project *Cat*. Later on, at LSL, Boris Yakobowski improved its scalability and proved its usefulness in the context of the verification of structural properties on memory separation and cyclic behaviors of an industrial critical system.

Inspect (0.4 kloc) is a debugging tool that outputs the internal Frama-C representation of C constructs.

JCard (6 kloc) takes as input JavaCard bytecode and generates an equivalent C program. This way, every Frama-C plug-in may analyze JavaCard code. This prototype was developed by Philippe Hermann, with my support, in the context of the project *ADS+*. It is now being used again in the context of the European project *H2O* [16].

Loop (1 kloc) aims at approximating the number of loop iterations in order to automatically infer Value's `slevel` parameters. They are numerical values (possibly different for every loop or every function of the program) which are the most important Value's parameters. Indeed they tune Value's *ad-hoc* trace partitioning [MR05] by indicating the maximum number of states kept by the analyzer at a given moment.

LTest (4.6 kloc) [BCDK14] provides three basic services for test automation: coverage estimation, automatic test generation and detection of uncoverable test objectives. It also proposes several standard coverage criteria, while adding new ones aimed at being straightforward.

16. I do not directly participate to this project, but LSL does and I was slightly involved in the project proposal.

Metrics (2.6 kloc) [BY] computes several purely syntactic metrics and a few metrics related to the coverage ratio of Value.

Mthread (6 kloc + 0.3 C-kwloc) is a Value-based verification tool for multithreaded programs. It detects invalid concurrent access to shared ressources by abstract interpretation techniques.

Nonterm (0.3 kloc) warns when definitively non-terminating pieces of code are detected.

Obfuscator (0.4 kloc) obfuscates a program by replacing any name (variable name, type name, etc) by a meaningless name while preserving program semantics. This kind of transformation is sometimes required by industrial partners before disclosing their code.

Occurrence (0.4 kloc) displays the occurrences of left values in the GUI. It takes aliasing into account by taking advantage of Value's results. I initially developed this plug-in in the context of the project *Cat* in order to demonstrate how easy the development of a Frama-C plug-in was (and still is). Later on, it has been proved particularly useful when investigating analysis results of large pieces of source code in the Frama-C GUI.

PathCrawler (10 kloc + 36 Prolog-kwloc + 1.4 C-kwloc) [WMMR05, BDHTH$^+$09] is a white-box test case generation tool based on several coverage criteria including MCDC. A limited online version of this tool is freely available online at `http://pathcrawler-online.com` [KWB$^+$12].

Pdg (3.5 kloc) computes an over-approximation of the program dependence graph (PDG) by relying on From's and Value's results. This plug-in is usually not directly useful for the end-user, but it provides a convenient graph representation for plug-ins that make use of program dependencies (*e.g.* Impact).

Pilat (3.4 kloc) [dOBP16] automatically generates polynomial loop invariants by relying on linear algebra techniques. Loop invariants are of particular interest for Wp and other verification tools that benefit from summaries of loop semantics.

Postdominators (0.3 kloc) computes the postdominator set $\mathcal{S}$ of a statement s: a statement s' belongs to $\mathcal{S}$ if all execution paths through s also go through s'. Similarly to PDG, while being of no interest for the end user, this notion is sometimes useful when implementing static analyzers.

Report (0.5 kloc) outputs in various textual formats a synthesis of analysis results in term of validity statuses of ACSL properties: which ones are valid, invalid, or have currently no complete proof of (in)validity. The different validity statuses and how Frama-C computes them is explained in Section 2.9.

Rpp (10 kloc) [BKGP17] extends ACSL with relational properties that involve several functions or several calls to the same functions. These extra properties are

then translated into additional code and axiomatics in order to be verified by other means, *e.g.* Wp.

Rte (1.5 kloc) [HS] generates ACSL annotations for different kinds of undefined behaviors including validity of pointers before dereferencing them, array accesses, and arithmetic overflows. It is in particular proved useful in Section 3.3.2.

Sante (1.9 kloc) [Che11, CCK+14] implements a verification technique that combines Value, Slicing and PathCrawler in order to refine Value's results. The method consists of several steps. First, it automatically runs Value. Next, it slices the program once for every potential undefined behavior found by Value. Finally, for every slice, it performs concrete executions on PathCrawler's generated test cases in order to determine whether each potential defect arises.

Scope (1.3 kloc) displays in the Frama-C GUI different left-value scoping information (*e.g* the potential definition points of a left value from a particular use).

Secure Flow (4.1 kloc) [ASTT13b, Ass15, BS17] generates an instrumented program that encodes the information flows of the original program inside its source code. This way, it becomes possible to verify (termination insensitive) non-interference [GM82] of the input program with standard verification tools like E-ACSL, Value or Wp run on the generated code. Section 3.4.2 provides some details about this tool.

Security Slicing (0.5 kloc) [MS08] is a specific slicing algorithm whose criterion is to preserve confidentiality of data. I implemented it in the project *€-Confidential*. Indeed, one of the contributions of my postdoc demonstrated that the standard (backward) slicing algorithms cannot preserve confidentiality of information because it may require to keep forward statements (that is, statements that are forward to the considered program point in the control flow graph). I proposed an alternative algorithm implemented in this small plug-in. This work is *not* presented in this document.

Slicing (5 kloc) is a program transformation technique that removes program constructs that do not impact a given semantic criterion (*e.g.* preserving the semantics of some statements): the generated program is equivalent to the original one with respect to this criterion, but is shorter. Consequently it helps the verification engineer or other analysis tools to focus on the desired properties of interest.

Sparecode (0.6 kloc) is a specific slicing whose criterion is to preserve statement reachability. Said otherwise, this plug-in removes pieces of dead code.

StaDy (3.4 kloc) [Pet15, PKB+16] explains proof failures of Wp by means of testing through PathCrawler. Indeed, when an automatic theorem prover is not able to discharge a proof obligation emitted by Wp, it may be because the code and its specification are inconsistent (at least one of them is wrong), or because a

required annotation is missing (*e.g.* a loop invariant), or because the property is out-of-scope of prover capabilities. StaDy helps the user to understand which reason is the cause of the failure. It relies on a translation from annotations to C code similar to E-ACSL but guided by test case generation [PBJ+14].

Synthesis (4 kloc) automatically generates a function body from ACSL postconditions by applying Viktor Kunčak's functional synthesis principle [KMPS10]. This research work was done through the postdoc of Michele Alberti that I supervise. It is *not* presented in this document, although if I discuss its potential applications for runtime verification in Section 3.5.2.

Value (18 kloc) [KKP+15, CYL+] computes over-approximations of the possible values of each program variable at each program point by abstract interpretation *à la* Cousot. A major side-effect of this analyis is to raise an alarm for any potential undefined behavior of the program and any ACSL annotation whenever it is not able to conclude that such a behavior can never occur (or that such an annotation can never be invalid). For instance, assuming we provide a suitable analysis context, Value is usually able to demonstrate the first two preconditions of function `max_subarray` in Figure 2.3, while it could prove the absence of runtime errors (including arithmetic overflows) of its implementation of Figure 2.2. However, it cannot verify the last precondition and the postconditions since it currently always topifies quantified predicates [17].

Value has hard-coded abstract domains that cannot be easily changed. Even if not as specific as the domains of Astrée for avionic programs generated from Scade, they have been chosen for their good compromise between precision and efficiency and rely on heavily optimized datastructures and algorithms [BC11]. Since the Frama-C release Aluminium released in May 2016, an evolved version of Value, called Eva (shares 10 kloc with Value + 8 kloc) [BBY17], is out. It improves the legacy Value in term of expressivity, genericity, precision and performance. Indeed Eva transforms Value from a monolithic analyzer with hard-coded domains to a generic extensible analysis parameterized by cooperating abstract domains. Case studies also demonstrate that Eva gets comparable analysis time for better results in term of precision, while being fully backward compatible with Value.

It is worth noting that Value has been removed from Frama-C Phosphorus-20170501 released in May 2017.

Variadic (1.7 kloc) replaces variadic function calls by non-variadic function calls by means of a monomorphisation technique [TO98] [18]. This way, the other plugins do not need to take care of variadic function calls.

17. "Topify" means that it associates the abstract value $\top$ (the least precise element of the abstract domain) to a concrete element (here, predicates).

18. The authors of the referenced paper named this technique *"code specialization"* or *"polymorphism removal"*. It appears that the name *"monomorphisation"* becomes popular a bit later when

Volatile (0.6 kloc) replaces accesses to volatile memory locations by deterministic function calls. This way, the other plug-ins do not need to take care of volatiles when analyzing the code. While volatiles are rather rare in academic examples, they are frequent in embedded applications and need to be handled specifically.

Wp (29 kloc + 26 Coq-kwloc + 3 Why3-kwloc) [KKP+15, BBCD] verifies function bodies with respect to ACSL annotations through weakest precondition calculus *à la* Edsger W. Dijkstra. The WP engine is parameterized by integer and real models as well as memory models. It includes a library called Qed, dedicated to the manipulation and simplification of proof obligations [Cor14] which eventually may be discharged by any prover supported by Why3. While Value is the Frama-C plug-in that automatically proves the largest number of properties, Wp is the Frama-C plug-in that is able to prove the most complex functional properties as soon as the user provides enough code annotations, including the necessary loop invariants, loop assigns and loop variants. Loop invariants are properties that are true before entering the loop and remain true at the end of each loop iteration [19], loop assigns are the memory locations that may be modified by the loop, while a loop variant is a measure that strictly decreases at each loop iteration, guaranteeing loop termination.

For instance, Figure 2.6 shows the code annotations that are necessary [20] to automatically discharge, with Wp and the Alt-Ergo prover [21], Kadane's algorithm of Figure 2.2 with respect to its specification of Figure 2.3. It is worth noting that the `loop assigns` clause cannot be omitted in order to prove this program with Wp, even if most deductive verification tools do not usually require this information. It can nevertheless be automatically generated by the GenAssigns plug-in. The proof also demonstrates the absence of undefined behaviors, including arithmetic overflows. Interestingly, this example illustrates the need for ghost variables and ghost code to encode the necessary pieces of information that are required to explain *why* the algorithm is correct but are useless to implement it. Here the ghost variables `low` and `high` respectively store the lower and the upper bounds of the maximum subarray found so far, while `cur_low` stores the lower bound of the currently visited subarray. The ghost statements in the loop body update these variables whenever required. Thanks to these extra variables, in addition to expressing boundaries in a standard way (the first four invariants), the loop invariants express that `cur` and `max` store the sum of the current subarray and the sum of the sub-

introduced in the SML optimizing compiler MLton (`http://mlton.org`).

19. In practice, they should be informative enough to prove the desired properties, so `true` is pretty rarely a satisfactory loop invariant.

20. Arguably alternative versions are certainly possible, but the proposed solution is minimal in the sense that you cannot remove an annotation without breaking the proof.

21. `http://alt-ergo.lri.fr`

array from `low` to `high` − 1, respectively (fifth and sixth invariants). They also indicate that `cur` is the maximum sum ending at the last visited index (seventh invariant), while `max` is the maximum sum ever found up to now (eight invariant). Actually the beauty of Kadane's algorithm is that it only requires maintaining the intermediate sums `cur` and `max` without exhibiting the subarrays *per se*. These subarrays are nevertheless required to explain its correctness; thus the use of ghost variables to exhibit them.

This description demonstrates that we have developed over the years plenty of tiny (< 1 kloc) or small (< 5 kloc) plug-ins, several medium-size plug-ins and a few large (> 20 kloc) plug-ins. If we assume that there are lots of features easy to implement, while only a few complicated ones, I think that this is an empirical evidence that Frama-C succeeds in fulfilling Alan C. Kay's maxim [Dav93] that is displayed on the Frama-C website:

"Simple things should be simple, complex things should be possible".

2.7 Frama-C, an Extensible Framework

Context This section is inspired from the conclusion of the Frama-C reference paper [KKP+15] with an updated related work.

In the introduction of the previous section, I wrote that the LGPL license allows anyone to release additional plug-ins with the license of their choice. Maybe I did not insist that much on the word *anyone*. Indeed, in addition to allow LSL to distribute plug-ins through different licenses, another positive consequence of the Frama-C LGPL license is that several Frama-C plug-ins have been developed outside LSL by both academic and industrial partners. Some of them are freely available, while some others are not (and often used for industrial purposes).

Academia usually develops open source Frama-C plug-ins in order to experiment new ideas. Using Frama-C to develop their research prototypes allows researchers to rapidly obtain a usable tool that works for an important subset of C without too much effort. Claude Marché and Yannick Moy have developed Jessie [Moy09, MM], which is another plug-in based on weakest precondition calculus. Shubash Shankar [SP16] integrated the model checker Cegar in Frama-C to verify statement contracts. Antoine Ferlin *et al.* [FBCDW15] develop a Value-based plug-in to set observation points necessary for generating traces dedicated to offline monitoring of temporal properties. Several prototypes aim at verifying security properties. Jonathan-Christofer Demay *et al.* generate security monitors based on fine-grained feedback from Value [DTT09]. Dumitru Ceara *et al.* implemented a taint analysis [CMP10], producing explicit dependency chains with accompanying risk quantifiers. Xavier Kauffmann-Tourkestansky *et al.* [BHKL10] propose a

```
int max_subarray(int *a, int len)  {
  int max = 0, cur = 0;
  /*@ ghost int cur_low = 0, low = 0, high = 0; */

  /*@ loop invariant 0 <= i <= len;
    @ loop invariant 0 <= low <= high <= i;
    @ loop invariant 0 <= cur_low <= i;
    @ loop invariant 0 <= cur;
    @
    @ loop invariant cur == sum(a,cur_low,i-1,len);
    @ loop invariant max == sum(a,low,high-1,len);
    @
    @ loop invariant \forall integer l;
    @   0 <= l <= i ==> sum(a,l,i-1,len) <= cur;
    @
    @ loop invariant \forall integer l, h;
    @   0 <= l <= h <= i ==> sum(a,l,h-1,len) <= max;
    @
    @ loop assigns i, cur, max, cur_low, low, high;
    @ loop variant len - i; */
  for(int i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) {
      cur = 0;
      /*@ ghost cur_low = i+1; */
    }
    if (cur > max) {
      max = cur;
      /*@ ghost low = cur_low; */
      /*@ ghost high = i+1; */
    }
  }
  return max;
}
```

Figure 2.6: Required Code Annotations to Prove the Implementation of Figure 2.2 *w.r.t.* the Specification of Figure 2.3.

source-code model for verifying physical attacks on smart cards, and use Value to verify it. Finally, Maria Christofi [Chr13] automatically injects faults into the code to formally verify implementations of cryptographic protocols. Some other prototypes focus on generating annotations: Ahmed Bouajjani *et al.* [BDES11] automatically synthesize invariants of sequential programs with singly-linked lists, while Nicolas Ayache *et al.* [AARG12] automatically infer trustworthy ACSL assertions about the concrete worst-case execution cost of programs from so-called "cost annotations" generated by a custom C compiler. Also, Rigel Gjomemo *et al.* [GNP+15] automatically insert assertions in LLVM to improve the effectiveness of several optimizations.

On the industrial side, some companies also develop Frama-C plug-ins. At Airbus, David Delmas *et al.* verify compliance to domain-specific coding standards [DDMLS10] and to expected control and data flows [CDDM12] through home-made plug-ins. At Dassault Aviation, Dillon Pariente develops two security-related plug-ins Gena-Taint and Gena-CWE. The former is a taint analysis plug-in based on Value's results. The latter is an extension of Value that detects more CWEs (Common Weakness Enumerations) [22] than Value alone [PS17]. Adelard, a UK company that performs safety and security audit for the nuclear industry, also develops a plug-in to analyze concurrent programs in a complementary manner to Mthread: it trades off soundness for simplicity [23]. SafeRiver is another SME that develops Frama-C extensions, *e.g.* Carto-C for cartography of C source code [24]. Also, TrustInSoft develops two industrial versions of Frama-C pre-installed with proprietary extensions designed to facilitate the analysis of complex programs, namely TIS-Analyzer [CRH17] and TIS-Interpreter [25].

It is possible that I have omited some external Frama-C plug-ins. Indeed it becomes more and more complicated to be aware of the projects that use Frama-C throughout the world, even for developing plug-ins that arguably require often more effort than just using parts of the framework. That is particularly the case in industrial settings where the users and developers have no particular need to communicate on their business practices.

2.8 Frama-C, a Kernel-Centred Framework

Context This section summarizes a few papers [Sig09, Sig11, CDS11, Sig14, Sig15] that I wrote about the Frama-C kernel.

22. `http://cwe.mitre.org/`
23. `https://bitbucket.org/adelard/simple-concurrency`
24. `http://www.safe-river.com/index.php?id=carto-c&setlang=en`
25. http://trust-in-soft.com/tis-interpreter/

As explained in Section 2.6, Frama-C consists of a set of plug-ins that may be seen as independent tools. Indeed using a runtime verification tool based on program transformation technique like E-ACSL is not the same as using a program proof tool like Wp and both are different from an abstract interpreter like Value in many respects. This *heterogeneity of usage* means that Frama-C must be customizable enough to fit the analyzer needs. However, Frama-C is not only a collection of independent tools but aims at being easily identifiable as a single tool. That means primarily *homogeneity*. For instance, user interactions (*e.g.* inputs/outputs) should be as uniform as possible in order to make the tool easier and faster to learn. Being both heterogeneous and homogeneous while being user friendly implies that the user should identify quickly what is common to all analyzers and what is specific to each of them. For instance, a code analysis tool usually has many parameters that allow the users to tweak the tool according to their current use case: they must be able to quickly understand which parameters are common to all analyzers (*e.g.* system-dependent information like the size of C types) and which ones are relevant only for specific analyzer(s) (*e.g.* the slicing criteria for the Slicing plug-in). One way to provide homogeneity is *inversion of control* which gives the control of any tool t that uses a framework f to f and not to t. For instance, the framework often plays the role of the main program in coordinating and sequencing application activity. Interestingly, inversion of control is usually considered as a defining characteristic of frameworks [JF88].

To reach this goal, the Frama-C software architecture [Sig15] is based on a *kernel* that plug-ins are connected to. This fundational design was defined when Frama-C was created and has never been changed. It was not new however. For instance, Eclipse [26] and Gimp [27] are based on a similar architecture usually refered to as *plug-in based architecture*. The Frama-C kernel has several fundamental objectives. First, it really controls the execution of Frama-C from its beginning to its end, notably when plug-ins are loaded, parameterized and executed. Second, it parses C programs into a normalized representation shared by all plug-ins. Third, it provides several general services for helping plug-in development [SAC$^+$] and providing convenient features to Frama-C's end-user in a homogeneous and consistent way.

Figure 2.7 details Frama-C's software architecture and emphasizes the central role of the kernel. It also indicates my own contribution to its development. I do not indicate a maturity level because (almost) all the kernel is mature enough to be used — and is currently used — in industrial settings. All in all the size of the kernel is about 88 kloc. The size of my major contributions are about 23 kloc. That is about the same than the size of my minor contributions. In addition to plug-ins, the kernel is split into three main parts which correspond to directories in the source and shown with different colors on Figure 2.7 (blue, orange and green). They contain several

26. `http://eclipse.org`
27. `http://www.gimp.org/`

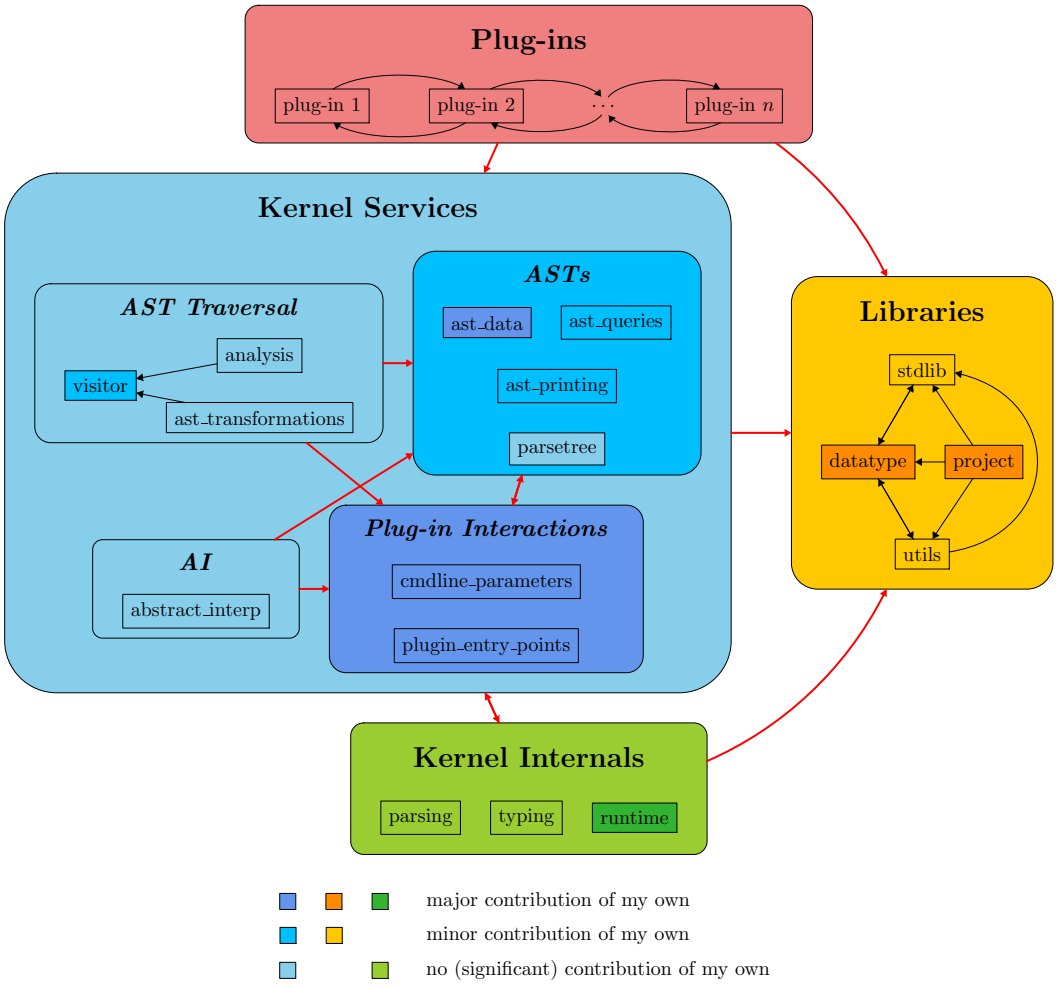


Figure 2.7: Frama-C Software Architecture.

services which correspond to subdirectories in the source and group together several compilation units. This organization aims at helping a developer, particularly a plug-in developer, to find what (s)he is looking for more easily. The edges between services indicate dependencies: one edge from a (group of) service(s) A to a (group of) service(s) B means that A uses B.

The three main parts of the kernel are the following:

libraries groups together libraries that are not related to the interpretation of C programs: theoretically, they could have been put outside Frama-C.

kernel internals are the compilation units that are only necessary inside the kernel. Plug-in developers should not use them.

kernel services really correspond to the kernel API and propose several kinds of services which can be useful to plug-in developers. The most important kind of services is the operations related to the AST (Abstract Syntax Tree), which is the internal representation of a C program annotated with ACSL. We also propose several ways to traverse it. The kernel API includes several possibilities of interactions with plug-ins and a toolkit dedicated to abstract interpretation.

Each service is described below.

libraries/stdlib (1.9 kloc) contains a few extensions to the OCaml standard libraries like additional operations over sets and maps.

libraries/utils (5 kloc) contains general-purpose libraries (different kinds of maps, vectors, bit vectors, floats, system commands, filepath managements, pretty printing, etc).

libraries/project (2.3 kloc) contains the project library of Frama-C [Sig09] [28]. It is the very first large service that I developed in Frama-C (in 2007-2008) and it is one of the most used and stable parts of the tool (only three small extensions and one minor modification in the last 2 years). The initial requirement expressed by Benjamin Monate, who was leading the development of Frama-C at that time, was to be able to analyze several programs (or the same program with different parameter sets) in the same Frama-C run (called a *session*). For instance, it should be possible to generate a new AST a' from an input AST a and let the user choose to work with a or a'. A naive solution would have been to parameterize almost everything that contributes to an analysis by the analyzed AST. That would have been not tractable. In fact the second and third requirements were that the solution should remain light for the developer (at least the plug-in developer), while having no visible cost at runtime.

The implemented solution consists in grouping together all the internal states related to one AST: the AST itself and its associated states inside the kernel

28. This paper [Sig09] was the very first article about Frama-C ever published.

as well as the parameters and the results of the analyzers. An internal state (or, shortly, a state) is usually provided as an OCaml global imperative value such as a (global) hashtable or a (global) reference. Such a group of all states is called a *project*. Several projects can co-exist at the same time, but one and only one is active: the *current project*. This way, the developer and the end-user only works with one AST and a single set of parameters and results. They can also switch from one project to another instantaneously. The project library stores all the existing projects: that is the centralized global state of Frama-C. Indeed the relationship between the library and a single state is a client-server model as shown in Figure 2.8. Each single state must be registered in the project library through an OCaml functor called `State_builder.Register`. It only knows its current value (that is, its value in the current project), while the library stores its values in the different project. When needed, the library can broadcast a query to the states (*e.g.* to get or set its current local values).

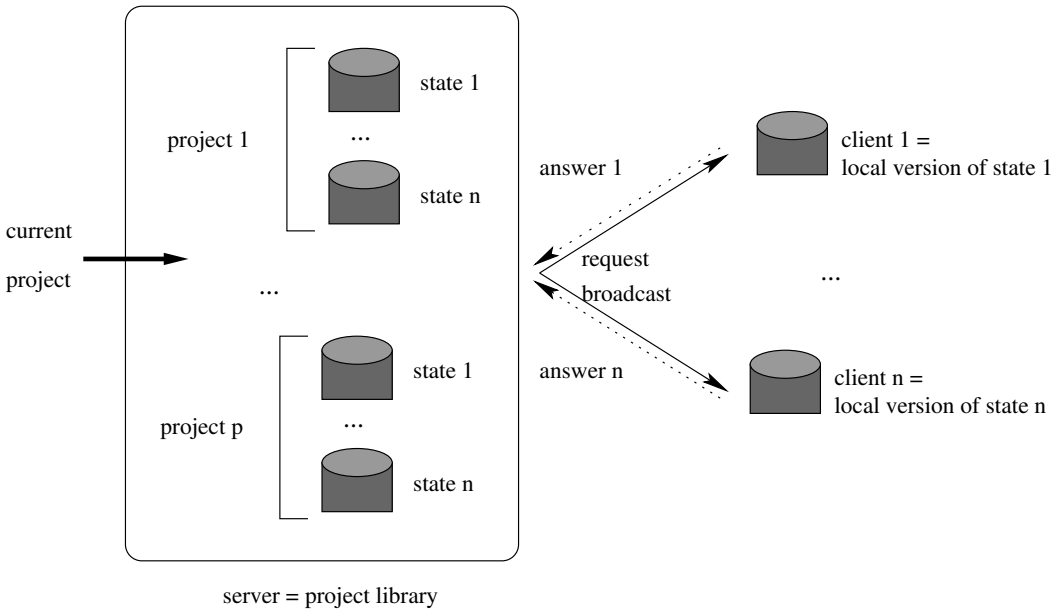


Figure 2.8: Client-Server Model of the Project Library.

Two key properties of the library ensure non-interference between projects: when modifying a state of a project, it is not possible to modify any state of another project. They can be informally expressed as follows. More precise formulations and their associated proofs can be found in the above-mentioned paper [Sig09].

Theorem 2.1 (Project Isolation) *The state of a project p is not shared with any state of another project p' ($p \neq p'$).*

Theorem 2.2 (Local Isolation) *The local version of a state is not shared with any state of any non-current project.*

These theorems actually rely on hypotheses which must be satisfied when applying `State_builder.Register`. One of these hypotheses is a non-aliasing property which is unfortunately tricky to implement safely because it usually requires one extra level of indirection. For instance, creating a state of type `int ref` requires to handle a value of type `int ref ref` and to modify one reference or the other when necessary. To prevent correctness issues, the library provides high-level functors which ensure the expected properties for standard mutable types and prevent the developer from applying directly the default (low-level) functor: the current open source distribution of Frama-C contains 1543 registered states, but there are only 4 direct applications of `State_builder.Register` (only 1 in a plug-in). By the way, those 1543 registered states also demonstrate the importance of this library in Frama-C.

Another important feature of this library is state dependency. The developer may (and must) declare that a state depends on another one (for instance, an analysis result usually depends on the AST). This way, when modifying a state, it is possible to clear its dependencies in order to force them to be recomputed (in combination with memoisation [Mic68]). It also provides a simple way to consistently apply an operation to a set of states (*e.g.* copying the state s and all its dependencies). This mechanism allows Frama-C to remain consistent almost for free when modifying one of its numerous states.

Last but not least, since the project library contains the Frama-C global state, it is also responsible for saving and loading a session. This is actually not as trivial as one might think at a first glance because of the heavy use of hash consing [Ers58, Got74] by Frama-C [CD08]. Even projects are themselves hashconsed. However, if nothing is done, unmarshalling (the underlying operation when loading) breaks the fundamental maximal sharing invariant of hashconsing: the saved hashconsed values need to be re-hashconsed when loading them. To do so, I worked with Damien Doligez from Inria, who implemented in C the unmarshalling algorithm of the OCaml standard library, and Pascal Cuoq, the main developer of Value at that time. We implemented in OCaml a custom efficient algorithm that allows the developer to apply safe transformers when unmarshalling values [CDS11]. Safety is ensured by the datatype library.

libraries/datatype (4.4 kloc) contains the datatype library of Frama-C [Sig11]. This library implements dynamic types that lift OCaml monomorphic types toward

first-class expression. It also provides heterogeneous hash tables, which are hash tables containing values of different types. Even if the library internally uses the unsafe module `Obj`, its API relies on phantom types [Rhi03] to ensure its correctness, informally expressed as follows:

Theorem 2.3 (Strong Correctness of the Datatype Library) *If no unsafe value is used outside the datatype library, well-typed expressions do not go wrong.*

The proof of this property as well as other proved properties of the library can be found in the above-mentioned paper [Sig11].

This library was primarily developed to provide safe APIs to plug-ins dynamically linked against the kernel. It is not used anymore for this purpose since the Frama-C compilation toolchain has been improved. Still it would be the only way to implement mutually recursive plug-ins but there are currently none. Rapidly, the library was also used to provide two other features to Frama-C. The first one is safe unmarshalling as already discussed. The second one is safe *journalisation* [Sig14] which is a way to automatically generate an OCaml script from the user actions for later replay: if loaded by Frama-C, this script provides the exact same output as the initial user session. The journalisation algorithm also uses introspection primitives provided by the datatype library.

Last but not least, this library also provides types together with standard operations (comparisons, hash function, pretty printers, *etc*) through modules implemented in a compilation unit called `Datatype`. Such types with their default operations are called *datatypes*. They are used throughout the framework (1845 occurrences of `Datatype`) and implemented for newly defined types. For instance, an extension called `Cil_datatype` provides datatypes for the types of the Frama-C AST (1266 occurrences, even though it is one of the two modules — with the module `Cil_types` containing the types of the AST — that good Frama-C programming practice recommends opening).

Features introduced in OCaml in the last few years could possibly be used to replace most usage of this library. However, a one-to-one replacement would be a lot of work. In particular, some custom functionality could be hard to replace, like pretty printing OCaml values as valid OCaml code. For instance, they are usually not provided as such by existing alternative implementations (*e.g* the `dyntype` library [29] or Alain Frisch's implementation at Lexifi [30]).

29. `https://github.com/mirage/dyntype`
30. `https://www.lexifi.com/blog/dynamic-types`

A more realistic improvement would be to automatically generate datatypes from type definitions through `ppx` rewriters[31].

kernel_internals/parsing (4.8 kloc) contains the parser of C files annotated with ACSL annotations. It generates an untyped AST, called `Cabs`, per C file.

kernel_internals/typing (12 kloc) types and links `Cabs` AST in order to generate a single Frama-C AST.

kernel_internals/runtime (0.6 kloc) contains pieces of code that are executed when starting Frama-C.

kernel_services/parsetree (0.5 kloc) contains the untyped AST `Cabs`.

kernel_services/ast_data (7 kloc) contains the typed Frama-C AST as well as associated tables which must be kept synchronized altogether, such as tables of globals, functions, annotations and property statuses. I will introduce property statuses in more details in Section 2.9.

kernel_services/ast_queries (19 kloc) contains standard operations over the AST. For instance, the module `Cil_datatype` belongs to this service.

kernel_services/ast_printing (4.7 kloc) contains the pretty printers of the AST.

kernel_services/cmdline_parameters (2.9 kloc) implements the parsing and the interpretation of the Frama-C command line, which is extraordinarily tricky. Indeed initializing Frama-C is composed of four stages, each of them being decomposed into several steps and substeps, for a total of 19 (see plug-in development guide [SAC$^+$, Section *Initialization Steps*]). Such a complexity comes from plug-ins. For instance, a few Frama-C options allow the user to load a plug-in, while loading plug-ins often leads to registering new options which must be detected on the command line. Furthermore, loading a previously saved Frama-C session requires loading the plug-ins which have saved states, but additional options on the command line should be able to modify the loaded states. This complexity also partly comes from the fact that Frama-C allows the user to put (most) command line arguments in any order. It offers more flexibility to the user but it has a cost in term of complexity which was certainly underestimated at the very beginning when almost no features were present. Being stricter would certainly make the parsing and the interpretation of the command line simpler. However, it could complicate a bit the interpretation of such parameters in the GUI. Indeed, it is not always appropriate to apply immediately the effect of modifying a parameter there, but then it would require to save the modification sequence to apply them in the correct order. As often, there is no perfect solution but, still, I would probably choose another command line semantics for a new similar tool.

31. `https://caml.inria.fr/pub/docs/manual-ocaml/extn.html`

kernel_services/plugin_entry_points (6 kloc) contains the most important registration points for plug-ins. For instance, plug-in developers should register their plug-ins into the kernel and be provided operations allowing user feedback, error messages and new (automatically projectified and journalized) parameters as command line options.

kernel_services/visitor (1.4 kloc) implements the object-oriented pattern known as *visitor* [GHJV95]. It allows the developers to "visit" only a few nodes of the AST instead of writing themselves a complete recursive traversal of the AST. This way, when the operation of interest only focuses on some constructs of the AST (of which there are plenty in the case of an ACSL-annotated C program), it is possible to implement it in a few lines of code. The drawback is that, unlike a standard recursive traversal, the type checker does not warn the developer when some cases are missing.

Importantly, the visitor is the tool of choice in Frama-C for implementing program transformations. It has two flavors: the *in-place* visitor, which visits the AST in place, is dedicated to get information from the AST [32], while the *copy* visitor generates a new project that contains by default a deep copy of the original AST and lets the developer modify the AST. The visitor takes care of preserving the kernel internal invariants between the AST and its associated tables. It would be almost impossible for a developer to preserve them correctly because they are many, almost undocumented, and not always easy to preserve.

kernel_services/analysis (4.5 kloc) implements a few static analyses directly provided by the kernel. The most important ones are certainly generic backward and forward dataflow functors.

kernel_services/ast_transformations (0.8 kloc) implements a few program transformations provided by the kernel.

kernel_services/abstract_interp (11 kloc) implements a generic toolbox for abstract interpretation, in particular generic lattices.

This presentation of the Frama-C kernel is almost complete but still not fully done. Even without mentioning its LablGtk2 graphical user interface (9 kloc) which is as extensible as the rest of the framework, another important element for extensibility is often ignored but is worth highlighting: **the compilation toolchain**. Actually the effort spent on this obscure part of the tool is impressive even if I have little evidence to prove that claim. In addition to the Git activities on this part of the code, perhaps the only quantifiable evidence is the number of lines, which is

32. The Plug-in Development Guide [SAC+, Section *Project Management System*] actually indicates that one should never modify the AST in place because the kernel cannot guarantee that all its invariants are preserved. To be accurate, the only standard case where it is always safe to modify an AST in-place is just after being inserted in a newly created project.

about 1.8 kwloc of m4 for both configure of the Frama-C kernel and about 4.8 kwloc for all its `Makefile`s (a few of them being automatically generated). Indeed this part of Frama-C has been the most buggy part of the whole framework for sure. I think that make is a powerful language with no current equivalent, but it has poor genericity mechanisms and some important parts of this language have a complicated unintuitive semantics. Consequently it is difficult to provide to the Frama-C developers an easy way to implement their own plug-in `Makefile`, assuming that (s)he has only basic knowledge of make. However, I currently see no other build system for OCaml programs mature enough to be used.

Other difficulties of the compilation toolchain arise from the fact that it must support several complicated features (parallel compilation, static or dynamic linking, compilation of a plug-in inside or outside Frama-C, deactivation of some plug-ins, verbose and silent compilation mode, *etc*) [33]. The Frama-C `Makefile` is a program of its own with a dedicated architecture shown in Figure 2.9. Despite the difficulties of maintaining this large piece of code, I would like to highlight that it almost succeeds in its original goal: it is easy to implement a `Makefile` for a simple plug-in by setting a few variables and including a `Makefile` called `Makefile.dynamic`. I will nevertheless not detail it deeper. The interested reader may refer to the Plug-in Development Guide [SAC+, Section *Makefiles*] for additional details.

2.9 Frama-C, a Collaborative Framework

Context This section is mainly based on the Frama-C reference paper [KKP+15, Section 3.4] and my publication with Loïc Correnson [CS12]. The last subsection about the project *Chekofv* is novel.

Because of the Hugh G. Rice theorem, which roughly states that program analysis aims at solving undecidable problems (see Section 2.1), no program analysis technique is perfect by nature. Consequently, even if it is really important to continuously improve particular analysis technique in order to successfully solve more and more problems, no technique can ever claim to be *the* universal solution to program analysis in general and to program verification in particular. To mitigate this issue, the key Frama-C feature is analyzer collaboration. It allows the user to combine several analyzers altogether to solve one particular problem that none of them is able to solve alone. Analyzer collaboration comes in two different flavors in Frama-C: either *sequentially*, by chaining analysis results to perform complex operations; or *in parallel*, by combining partial analysis results into a full program verification.

33. In an effort to reduce the maintenance effort of the compilation toolchain, the less used features have been discarded in the latest releases.

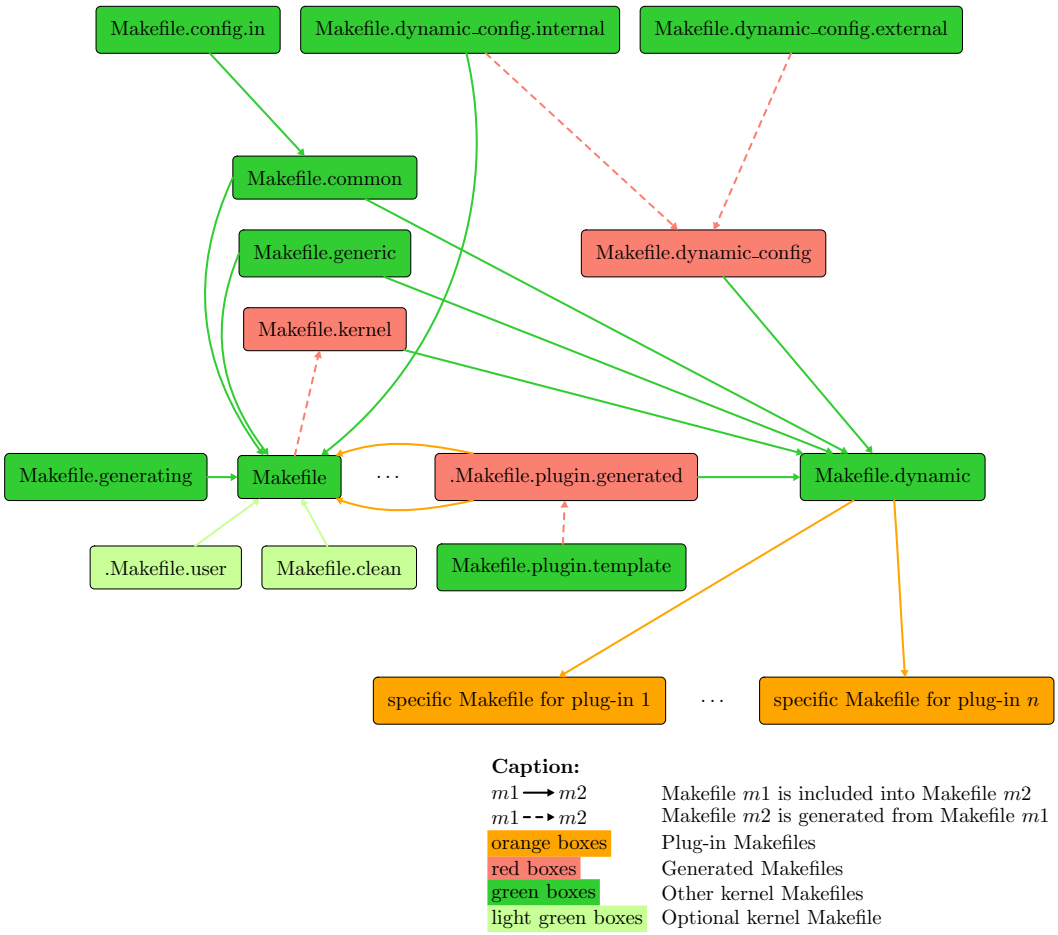


Figure 2.9: Relationship between the Frama-C Makefiles.

2.9.1 Sequential Collaboration

Sequential collaboration consists in using the results of an analyzer as input to another one thanks to the plug-in APIs. For instance, it is easy to use the Eva results through its API in order to handle at least function pointers and aliasing in a safe and easy way. Actually all the abstract interpretation based plug-ins of the open source distribution of Frama-C proceed that way: Callgraph, Constfold, From, Impact, InOut, Nonterm, Occurrence, Scope, Security Slicing, Slicing, Sparecode and Users.

Sequential collaboration can also consist in generating annotated C code, which encodes a verification problem in such a way that it can be understood by the next analyzers in the chain. It is actually why ACSL is often refered to as the *lingua franca*

of Frama-C analyzers. For instance, the plug-in Aoraï generates ACSL annotations which encode a Büchi automaton denoting a LTL formula. These annotations can in turn be verified by, let's say, Wp.

More generally, sequential collaboration is the *raison d'être* of all the plug-ins classified in categories *"support"* and *"expressiveness"* in the LSL's Frama-C plug-in Gallery of Figure 2.4. Indeed support plug-ins aim at either computing data to simplify the job of verification tools (*"pre-support"*), or simplifying their results (*"post-support"*), while expressiveness plug-ins aim at handling more complex programs or properties by encoding them in simpler C code extended with ACSL annotations. A more complex example of sequential collaboration will be provided at the end of this section.

2.9.2 Parallel Collaboration

Parallel collaboration consists in using several analyzers to verify program properties, each analyzer verifying a strict subset of the whole goal. For instance, some ACSL properties can be proved by Eva while most of the others are discharged by Wp and the few remaining goals are validated by E-ACSL. This kind of collaboration fully relies on an algorithm, named *consolidation of property statuses* [CS12], that I have designed and implemented in the Frama-C kernel. It ensures the consistency of several partial results. The algorithm is independent of any verification techniques and particular programming and specification languages. Its correctness requires a *blocking semantics* [GGJK08, HMM12] which expresses that, if an execution trace leads to an invalid property, then the program stops. In particular, it does not evaluate the next properties of the execution flow.

Overview of the Consolidation Algorithm

The consolidation algorithm takes as inputs property statuses emitted by the analyzers after trying to prove a property (*i.e.* a subpart of an ACSL clause for Frama-C): *true*, *false*, or *don't know*. These statuses are called *local statuses*. A local status may depend on the validity of some other properties — called its *hypotheses* — which are indicated by the analyzer when emitting it [34]. For instance, Wp may indicate that its proof of the postcondition of a function f depends on the validity of the loop invariants of f. The algorithm first builds a consolidation graph whose nodes denote properties and edges represent dependencies towards their hypotheses. Analyzers are represented by another kind of nodes with edges towards the properties they established. A partial view of such a graph is represented in

34. Hypotheses over *false* properties are restricted to reachability of the associated program point of the property, otherwise the algorithm is unsound. Details are omited here but may be found in [CS12].

Figure 2.10. It will be detailed later in this section. Yet one can see two analyzers (Value and Wp, at the bottom) which try to prove several properties, those proofs relying on (mutually dependent) hypotheses.

From this graph, the second step of the algorithm computes the *consolidation status* of each property. It notably indicates if the property is fully proved, or not proved at all, or only only locally valid (the consolidation status is then called *valid under hypotheses*). In that latter case, the algorithm also computes what hypotheses remain to be proven to conclude the proof. It also reports the invalid properties and which analyzers prove which properties with which set of parameters, if several have been tried (for example, in the case of plug-in Eva, different `slevel` values, `slevel` being the parameter controlling the maximum number of states that Eva keeps at the same time). Frama-C has actually additional statuses which refine either the status *valid* or *unknown* (*valid under hypotheses* being one of them) in order to improve user feedback.

Correctness and Completeness

Altogether this algorithm is both correct and complete in the following sense.

Theorem 2.4 (Correctness of the Consolidation Algorithm)
Assume that analyzers emit true *(resp.* false*) for valid (*resp. *invalid) properties. Then, if the consolidation algorithm returns* valid *(resp.* invalid*) for a property π, then π is valid (resp. invalid) in the underlying logic model.*

Theorem 2.5 (Completeness of the Consolidation Algorithm)
Assume that analyzers emit true *(resp.* false*) for valid (resp.* invalid*) properties and also indicate all the hypotheses the proofs rely on. Then, if a property π is valid (resp. invalid) in the underlying logic model and an analyzer emits a local status different from* don't know *under recursively valid hypotheses, then the consolidation algorithm computes the consolidation status* valid *(resp.* invalid*) for π.*

Both theorems rely on two distinct notions of correctness of the analyzers that emit local statuses. The one for the correctness of the algorithm corresponds to the standard notion of analyzer correctness: analyzers must emit correct verdicts. The notion of correctness for the completeness of the algorithm is stronger: they must emit correct verdicts but also specify all their hypotheses. It is in fact not true in practice. For instance, Eva currently indicates almost no hypotheses because it would take too much computing time for too small benefit. However, informally, the more complete the hypotheses indicated by the analyzers are, the more complete the results of the algorithm are (even if there is no proof of such a relationship). The formal statements of these theorems as well as their proofs are in the above-mentioned article [CS12]. Surprisingly the proofs are not trivial and proceed by induction on a

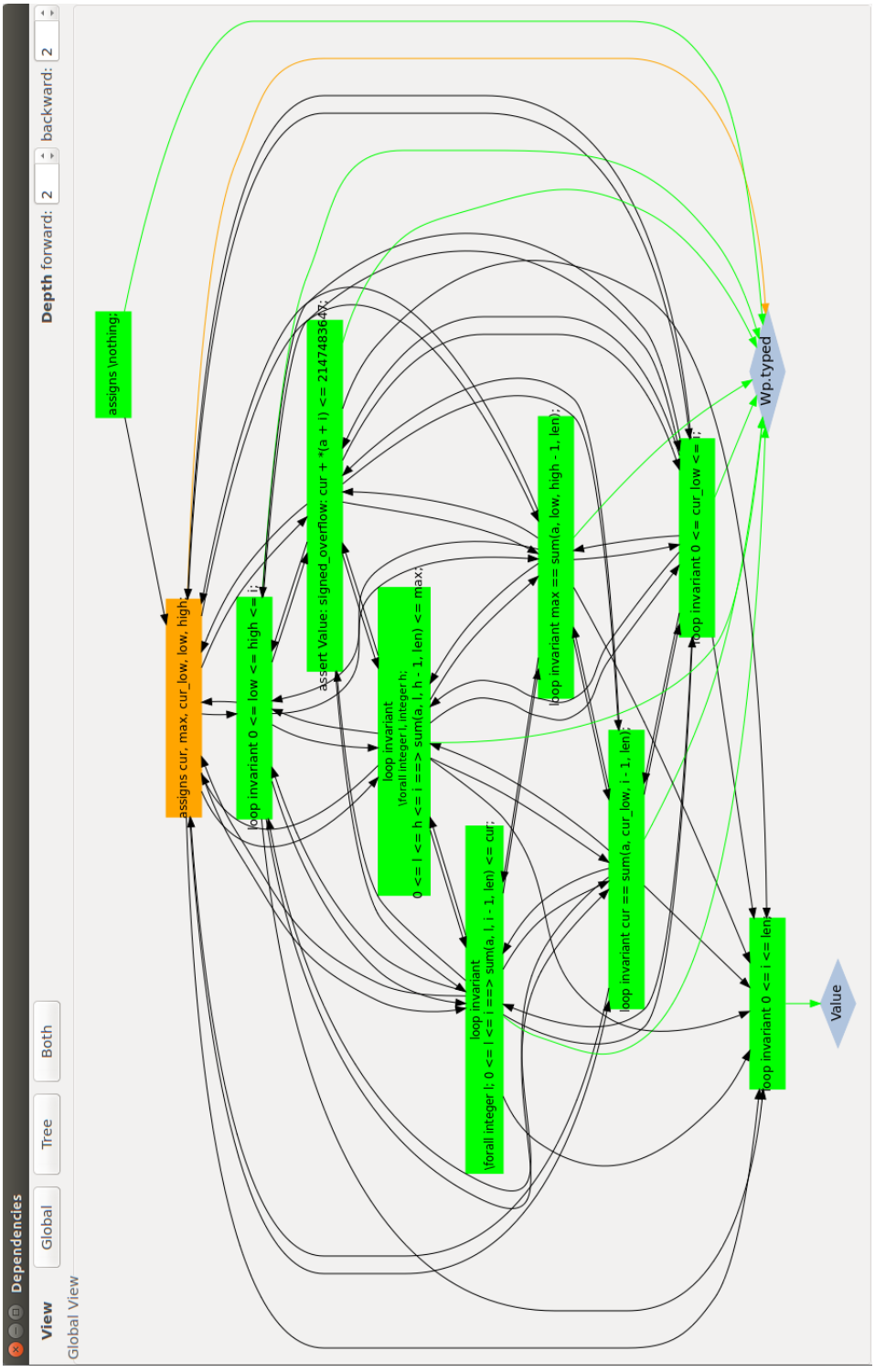


Figure 2.10: Consolidation Graph for the `assigns` Clause of an Unprovable Version of Kadane's Algorihm.

dedicated well-founded relation over properties that express a topological ordering over vertices of the (possibly cyclic) consolidation graph [35].

Handling Inconsistency

Interestingly, the consolidation algorithm is able to detect inconsistency. Indeed it is able to compute a special status named *inconsistent*. Actually, an extended statement of the correctness theorem 2.4 handles this status as follows (the addition to theorem 2.4 is the last sentence).

Theorem 2.6 (Extended Correctness of the Consolidation Algorithm) *Assume that analyzers emit* true *(resp.* false*) for valid (*resp. *invalid) properties. Then, if the consolidation algorithm returns* valid *(resp.* invalid*) for a property π, then π is valid (resp. invalid) in the underlying logic model. If it returns* inconsistent*, then π is both valid and invalid.*

If we assume that the underlying logic model is consistent [36] and so a property π cannot be both valid and invalid, then the last sentence means that computing a status *inconsistent* implies that the assumption of the theorem, that is correctness of analyzers, is wrong. Consequently, the consolidation algorithm may detect incorrectness of analyzers. That is a very nice feature because correctness of Frama-C analyzers is by far the most important property of the framework: incorrectness should never ever happen. However, it already happened in a very rare circumstances that the status *inconsistent* was emitted. Everytime, the reason was the same: incompatible *implicit hypotheses*. Indeed analyzers usually rely on hypotheses that guarantee correctness of their results but are not explicit. For instance, depending on its memory model, Wp may rely on non-aliasing of the arguments of the proved function while Value and Eva make some assumptions about their initial state. It is the responsability of the users to enforce these hypotheses through the verification methodology. When combining analyzers, it is usually tricky and error prone since the user must also combine hypotheses. That is why it is so important to make explicit such analyzer hypotheses, usually called explicit assumptions [CMW12, KCC+14]. However, even though Frama-C provides a (arguably

35. Beside this paper proof of the core of the algorithm, there is rigorous unit testing of its OCaml implementation which covers all the 234 different cases of the consolidation algorithm when an analyzer has emitted a status with 0, 1 or 2 hypotheses. Considering the criticality and the non-triviality of this algorithm, I think that a formal proof would be a valuable effort even if no bugs have been found for a while. Actually, as far as I remember, only one bug in a corner case was found since unit testing has been done (as usual, the implementation did not exactly match the paper algorithm). Yet it is the case that a paper proof combined with rigorous unit testing is not enough.

36. That is unprovable in a provably consistent logic model from Kurt Gödel's second theorem [Gö31, vH76].

primitive) way to express such properties and associate validity statuses to them, plug-ins unfortunately do not emit them. Providing a simpler way to make such explicit assumptions would be a nice improvement for Frama-C global correctness.

Example

In order to make this algorithm more concrete, let us go back to our running example, that is the annotated function `max_subarray` presented in Figure 2.2. Assume that one wrongly modifies the `loop assigns` clause, then tries to prove the program with Value and Wp in that order. Then the proof of the `loop assigns` clause fails. Otherwise said, Wp reports the local status *don't know*, while Value does not even try to prove any `loop assigns` and so emits no local status. Then all the properties are proved by Wp (so they have a local status *true*) while relying on the validity of the `loop assigns`: they get a consolidation status *valid under hypotheses* and are displayed with a specific bullet containing both colors green and orange in the GUI[37] as shown in Figure 2.11. That is for example the case for the `assigns` clause in the function contract of `max_subarray`. The Frama-C GUI is able to display the consolidation graph of a particular property p, which is the transitive closure of the nodes reachable from p in the (global) consolidation graph. Figure 2.10 shows this graph for the `assigns` clause[38]. It indicates that all properties are proved either by Value or Wp, but the `loop assigns` clause is orange. Consequently the user knows that (s)he must concentrate his (her) effort on this particular property. The watchful reader might notice that all properties but the `assigns` clause are mutually dependent. That is because they belong to the same loop and so both are goals to be proven and are induction hypotheses.

Related Work

It is worth noting that Frama-C is not the only tool which provides parallel collaboration but, as far as I know, it provides the most fine-grained collaboration taking care of correctness. For instance, the Eve verification environment [TFNM11] for Eiffel programs combines a deductive verification tool and a testing tool. In contrast to Frama-C analyzers, these tools are not supposed to be correct. Consequently, Eve computes a so-called *correctness score* for each property. Spark2014, the tool suite dedicated to the verification of a formal subset of Ada, also integrates both a verification tool and a testing tool but the collaboration is more coarse because it is made at function level, not at property level [CKM12]: for each function f, the

37. Since PolySpace, the meaning of colors green and orange in code analyzers is respectively *valid* and *don't know*, while the color red means *invalid*. Frama-C extends this panel of colors with additional bullets for its extra statuses.

38. For the sake of clarity, the properties guaranteeing the absence of undefined behaviors are omited in this figure.

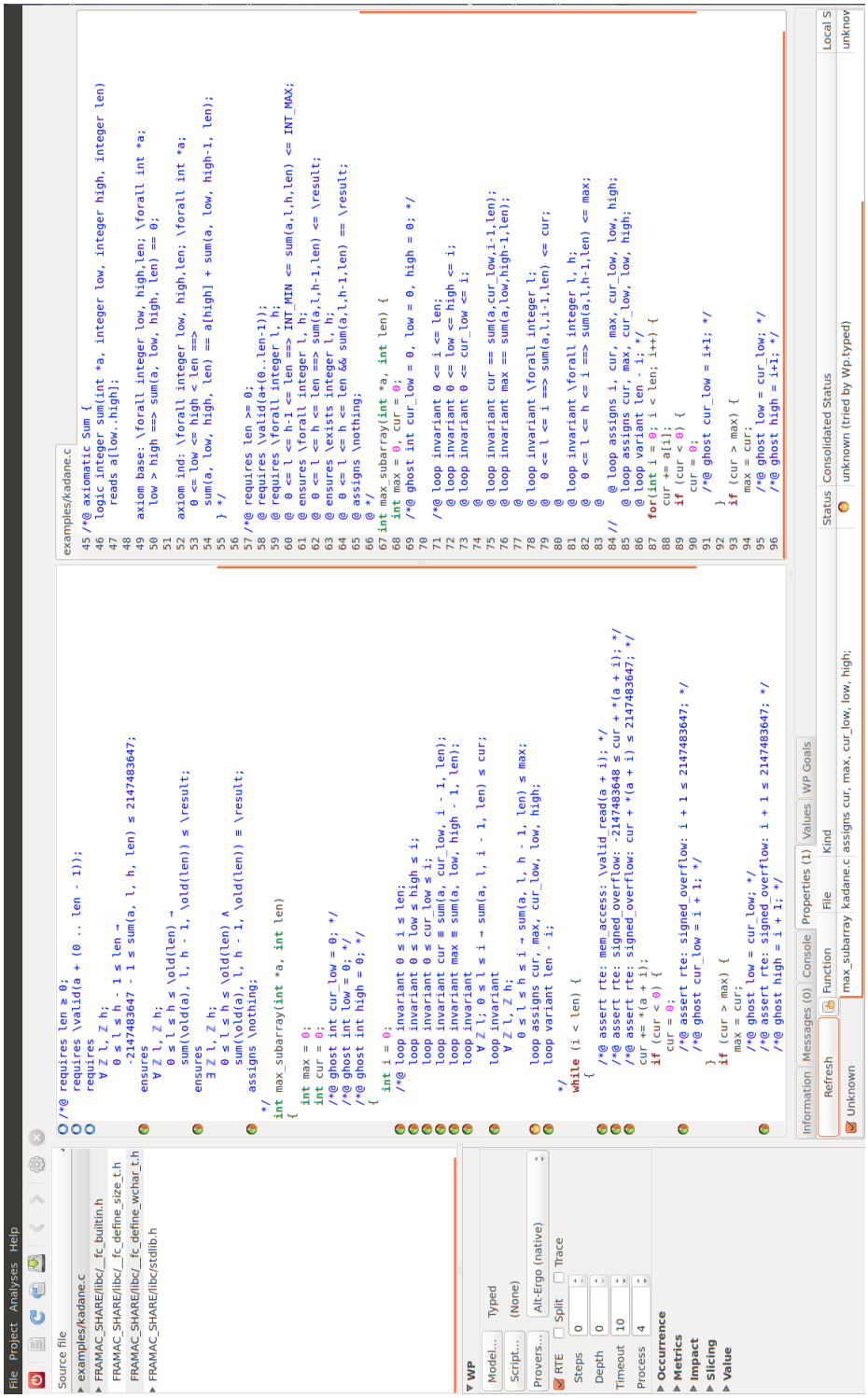


Figure 2.11: Incomplete Proof of the Kadane's Algorithm in the Frama-C GUI.

user can choose whether its properties must be proved or tested, but (s)he cannot decide to prove some of the properties of f and test the others. I limit myself to these two large-scale maintained tools but a few other (arguably not up-to-date) related works may be found in the related paper [CS12].

2.9.3 The Example of Project *Chekofv*

DARPA project *Chekofv*, in which I participated from 2012 to 2015, examplifies the advanced collaboration capabilities of Frama-C. Indeed, it is one of the examples, if not *the* example, that directly uses the largest number of different plug-ins to solve a common goal. The goal of *Chekofv* was to crowdsource the inference of loop invariants by means of computer games (and their gamers) [FSL+15]. In this context, I developed three Frama-C plug-ins named Fusy, Invmerger and Hardcheck which contribute to either produce game input data or to verify the solutions provided by the gamers. They exchange data and are also directly built upon the plug-ins Slicing, Value, Wp and Counter-Examples.

Fusy first slices the program according to each target loop. Then, for each slice (so for each loop), it runs Value in a precise way (similarly to a symbolic execution). During this run, it intercepts the precise computed values for each loop variable at each loop iteration through hooking facilities provided by the Value API. Finally, it converts collected data to game inputs. From these inputs, a game is built (this part was done by University of Santa Cruz). The solution provided by a gamer is then a candidate loop invariant expressed by a logic formula. Invmerger inserts such a formula as an ACSL loop invariant into the slice previously computed by Fusy. Finally Hardcheck tries to verify them. For this purpose, after a few program transformations like function inlining to automate the process, it runs Wp to verify the candidate invariant. In case of failure, Counter-Examples is run in order to propose a candidate counter-example, which in turn, Value attempts to verify. If not verified (meaning that it might be a real counter-example), Hardcheck converts it back in terms of Fusy's game input data in order to produce feedback to the gamer who has proposed a wrong answer. All in all, Hardcheck directly needs to use the API of six plug-ins (both Fusy and Invmerger as well as of Slicing, Value, Wp and Counter-Examples) in order to validate the answer from the gamer and giving it some feedback.

Even if the results of the project were unconclusive because the initial target application (the `bind` program) was arguably too ambitious, these three plug-ins helped to improve and demonstrate Frama-C collaboration capabilities: a powerful tool was developed which only relies on three small plug-ins (about 0.7 kloc for Hardcheck for a total of about 3.1 kloc).

2.10 Frama-C, an Evolving Framework

Context This section is novel.

Most of my contributions to the Frama-C kernel were done from 2007 to 2011, even if I continued more and more sparsely to add new features up to 2015. I think they have helped Frama-C to evolve from a young promising prototype to a mature industrial-strength tool. Nowadays my contributions to the Frama-C kernel in terms of new lines of code is small, mainly consisting of small patches. I focus more on code reviews and disseminating my knowledge of the platform internals to newcomers. However, the platform is still evolving. This section summarizes what could be learnt from the past twelve years and what could still be done in Frama-C and beyond.

2.10.1 Lessons Learnt

Frama-C is now 13 years old. I worked on this tool since 2006, about 11 years ago. At the origin, in 2004–2005, Frama-C was mainly developed by two intrepid OCaml programmers, namely Pascal Cuoq and Benjamin Monate. When I arrived, only a handful of engineer-researchers at LSL worked on the tool. Nowadays at LSL, fifteen engineer-researchers contribute to the platform as well as several PhD students and postdocs constituting a team with size varying from twenty-five to thirty persons. During these years, I think that all the lab including myself have learnt a lot about software development. It is almost impossible to report all of these lessons here, but I will summarize here the most important ones that I have personally learnt. I would like to emphasize that this section contains claims that are built upon my personal experience, but are not supported by formal proofs: they are debatable.

Lesson 1: A piece of code is rewritten at least twice before being mature.

I wrote several libraries for Frama-C and a development scheme has emerged: after the original development of a new library, its users require new features that are often added in a rush. At the time it becomes necessary to clean up the API: that is the first rewriting. Then, after a new period of uses including bug fixes and small extensions, another rewriting is usually necessary to design a better library architecture based on better data structures. That is the second rewriting. Then either the library becomes stable, or a new cycle of features and bug fixes come depending on the context.

One could claim that the proper library architecture, data structures and API could have been done in a proper way from the start. However, I do not think so: in a research project, the final needs rarely match the initial needs, while the

implementation regularly relies on original solutions for which it is hard to elaborate from scratch without a bit of experimentation.

This statement is actually in line with Frederick P. Brooks which explains in a wonderful way that one should "plan to throw one [system] away; you will, anyhow" because "for most projects, the first system built is barely usable: too slow, too big, too hard to use, or all three." [Bro75].

Lesson 2: Major central evolution is a pain.

Modifying an API that is used everywhere requires propagating the changes all along the codebase, including in parts that you do not maintain yourself. Even if the API remains stable, a major evolution takes months, literally (particularly when you do it part-time). During that time, conflicts usually arise. They can be hard to fix and certainly consume lots of time. They do not necessarily occur because two commits modify the same part of the code, but because side-effect modifications impact common parts like Makefiles (see Figure 2.9) and oracles of non-regression tests which must be kept up-to-date.

For instance, it was possible to introduce rather smoothly the project library (see Section 2.8) in 2007-2008, because the codebase was relatively small at that time. It would not be possible anymore on the current codebase (which now has thousands of uses of this library as already explained in Section 2.8).

Lesson 3: Historical mistakes are hard to overcome.

This is a corollary of the previous lesson. Indeed wrong design choices are sometimes made, in particular when developing a research tool for which you have no clear idea of what you really want to do. However, modifying such bad choices with large impacts is really hard.

For instance, a module named `CilE` was introduced in the first days as a dedicated extension of Cil for Frama-C. This file grew regularly in the beginning. It was a mistake because Frama-C became so large and so different from Cil that it was not possible to put all the extensions in this single file. Even if the file has been almost cleared out since 2009, it still exists today as part of Eva, and newcomers do not even know what its filename means.

Lesson 4: Knowledge of the code must be shared.

In small teams, a piece of code is often developed by a single developer who is then in charge of maintaining it. However, during the project lifetime, this person may leave for a while or permanently. He/she may also be overworked, while modifications have to be done. Then someone else has to maintain the code. To do so efficiently, (s)he must understand it well. It includes the understanding of both

the technical details with the help of the code comments, and of the overall code organization together with why it is designed as it is. The latest changes are often not documented directly in the code because code comments do not fit that need. So they must be documented elsewhere. Generally it is only in the head of the developer because writing a global code design document is a time-consuming task. That is why such code knowledge should always be shared between at least two persons for every piece of code. Unfortunately that is not the case in Frama-C, where many pieces of code are known by either no or one developer. Actually one of the main *raison d'être* of several of my publications [Sig09, Sig11, Sig14, JKS15b, AS17] is to share the core design of some important pieces of code.

Lesson 5: Code review does increase the quality of the code.

Up to the end of 2014, Frama-C developers directly committed in the master branch of the revision control system whenever they think the development was ready. Even if everyone did his (her) best, mistakes occur including bugs, compatibility issues and misfeatures.

At the end of 2014, the Frama-C development version was migrated to Git and Gitlab together with a major evolution of the Frama-C development process. It includes peer code review. Every single commit which modifies more than a few bits is reviewed by at least one engineer-researcher, the choice of the person being made according to which part(s) of the code is (are) being modified [39].

It has lots of benefits:

1. the code is less buggy because the reviewer is often able to detect bugs not caught by testing;
2. the code is much more commented because the reviewer really needs to understand it;
3. the code knowledge is shared, fulfilling the previous lesson;
4. reviewing includes recorded discussions on Gitlab about design and feature choices. Recording is particularly important to save the tool history and remember why particular choices were made.

All in all, the quality of the recent pieces of code is much higher than the older ones. It is definitively worth the many hours spent in reviewing.

Lesson 6: Single code review fails to build expertise.

Another goal of code reviewing was to build double expertise for the pieces of code with no or only one expert. Unfortunately that has not worked up to now.

39. Code review does not concern prototype plug-ins written by interns and PhD students, except if it is considered general purpose and good enough to be integrated in the mainstream version.

Indeed the developers tend to assign merge requests to be reviewed to an expert of the corresponding piece of code, not to a newcomer who would need to learn it. It is certainly better for the quality of the review but it does not help at all to reach the double expertise goal. To solve this issue, it would be necessary to assign merge requests to two reviewers, one expert and one newcomer. However, code review is costly and we decide not to spend additional manpower for such a task for the time being. The most visible drawback is that code review is almost always done by the very same group of Frama-C experts: without double reviewing, it takes very long to build expertise upon a large codebase.

Lesson 7: Non-regression testing takes both CPU and human time, but it is worth it.

From its earliest days, Frama-C has a larger and larger non-regression test suite (organized by plug-in). The golden rule is:

> *Never ever commit (or merge) as long as at least one oracle of the non-regression test suite is broken.*

We develop tools to support this rule: ptests.opt allows running the tests in parallel, to check the differences between the current results and the saved oracles and to update the oracles when necessary [SAC$^+$, Sections *Testing* and *Ptests*]. Another tool has been used for years to run the tests and additional checks on different systems each night with a morning report. It has been replaced by another Gitlab-based tool which performs many checks before allowing the reviewer to merge commits.

It really helps the developers not to reintroduce old bugs, or not to introduce new ones in the tool. However, running them take a lot of time (several minutes per run) and so make servers busy and force the developers and reviewers to learn patience. To reduce this drawback while keeping its benefits, a strong test management strategy must be defined to add meaningful minimal test cases for bug fixes and new features without compulsively integrating useless computations. We — the Frama-C development team — certainly has room for improvement in that respect.

Lesson 8: Industrial quality for research development platforms is nearly impossible.

A mature industrial-strength tool requires stability, notably backward compatibility. When the tool aims to be used as a development platform, that includes stability of its API. However, a tool developed from research results requires major evolution to breakthrough. It means whole re-implementations of some pieces of code with API rewritten from time to time (see Lesson 1). Clearly both objectives are *not* compatible. It also (partly) explains why **a research lab should never try to do what a company does: put and maintain a product on market**. As soon as a product has many users with high expectations (particularly in terms of stability), it is hard to make it strongly evolve and, consequently, it may slow

down research-guided development drastically. For several years now, Frama-C has been really caught in the middle: sometimes major necessary evolutions are not done or reported for a while because of the expected backward incompatibilities, while incompatibilities are introduced without paying attention enough to possible consequences some other times.

A solution could be to clearly indicate the maturity level of the different (parts of) APIs in a way that users could expect stability or evolution in future releases [40]. Of course that is easier written than done, since evolution is difficult to predict by nature. However, it would provide useful indications anyway and it is still roughly doable: if an API is used, well documented and remains unchanged during two releases, we might expect it will not be broken in the next ones, while a newly introduced API might be modified in an incompatible way in the next release.

Lesson 9: Keep small groups of contributors for a task, or let it die.

Put lots of people in a room with a decision to be taken, close the door and wait for a consensus. Almost always either no final decision is finally taken, or a bad compromise is made [41]. If you want that things to proceed, just select at most five or six persons. Only then, you have a chance to see strong not-too-bad actions initiated. The main difficulty of this process is certainly to convince possibly strong-minded persons to *not* participate in a decision process without offending them. Keeping groups small is also beneficial for global productivity because people who do not participate in one particular decision save time for something else, possibly even more important.

Lesson 10: A 10+-developer team is hard to keep on track of.

As explained in the beginning of this section, the Frama-C team grew from a handful of persons in the beginning to nearly thirty persons. That is really not the same in terms of management. At the beginning, everyone developed more or less what (s)he would like even if (s)he should let the others know of major upcoming changes. It is not possible anymore because of the large impact of such changes (see Lesson 2) and because of the many concrete objectives that must be fulfilled for industrial contracts and project deadlines. Planning is required. However, even if the team goals are the same, no individual has exactly the same, neither in terms of development nor research. Also both the team objectives and the individual goals may move quickly. It makes planning difficult.

40. There is almost no issue with end-user features: either they remain backward compatible, or they are almost useless and eventually die.

41. Another possibility is that some participants are actually irrelevant and are not involved in the decission process, but then they just waste time.

Since a minimum of global planning is nevertheless necessary in a large team to reach its most important goals, it is not possible anymore to let everyone fully decide his (her) own priorities. However, I do not think that a unique leader is a good solution because everyone should keep a certain amount of liberty of action in order to fulfill individual objectives as long as they are in line with the global project. Since about three years ago, consistent with Lesson 9, we are trying a new light organization with small groups of persons working on important parts of the framework: the kernel, Eva, Wp, and E-ACSL. Each group is led by one senior researcher but they are organized in different ways (as far as I understand how each of them works, particularly those which I am not involved in). Groups help us to set priorities for the most important parts of the framework without disturbing too many individuals. For instance, the kernel group has improved the Frama-C release process with a 6-month agenda for each release. Even if this organization works rather well for the time being, I think that this experiment is a bit too recent to draw definitive conclusions. Slight modifications might be required. However, one should not forget that **planning is mandatory to reach milestones in time, but freedom is the mother of high-quality research**.

2.10.2 Frama-C Perspectives and Beyond

Frama-C is a more mature tool today than ten years ago, but it is still far from being perfect. Even without talking about specific plug-ins and limiting myself to the kernel, a lot remains to do from both an engineering point of view and research perspective.

Software Engineering

From an engineering point of view, Frama-C's architecture and API still suffer from several major drawbacks despite many improvements over the years. They both show the burden of history and a few bad compromises in desperate attempts to satisfy everyone and eventually make everyone frustrated. In particular, the cumbersome heritage of Cil is still present. Even if its existence was of primary importance at the beginning, its monolithic API in the middle of other badly-structured files is a daily pain for any Frama-C developer: if you search for a particular function related to the AST, it could belong to about ten large files and their "organizations" just let you feel very alone. Even worse, this large API does not consistently provide the most important feature for writing program transformations: smart constructors, that is small functions enforcing the necessary invariants while building pieces of AST [42]. Unfortunately, because of the difficulty of central evolution (see previous

42. There are many smart constructors, but several are missing, while others are not really *smart*. Also they are randomly disseminated in several files and so may be hard to find.

section, lesson 2), this architectural defect is really hard to overcome. That explains why it is still here.

Not only did Frama-C evolve over the years. OCaml did too. We regularly upgraded the minimal OCaml supported version to take benefit from its latest improvements (and bug fixes), but still one major recent feature remains unused: ppx extensions which provide a way to write powerful syntax extensions understood by the OCaml compilers. They could possibly allow Frama-C to automatically generate standard sets of functions. For instance, datatypes could rely on Deriving [43] while visitors could be based on François POTTIER's Visitors library [44]. Once again, it would lead to a difficult major evolution of the framework.

Analyzer Collaboration

At the boundary between engineering and research, there is analyzer collaboration, the most important feature of Frama-C as a whole (see Section 2.9). Currently, Frama-C analyzer collaboration is a reality but it remains difficult to apply at a wide scale because of several deficiencies. First, invalid properties are hard to deal with because of the consolidation algorithm: they cannot depend on another property (except the reachability of its own program point) because it would open the door to incorrectness [CS12]. Reachability is usually proved by exhibiting an execution trace through dynamic or symbolic computation. However, there is no such notion of trace in Frama-C. Also, combined with the semantics of ACSL, the notion of (property) reachability is surprisingly tricky because properties are not only attached to program points as usual, but could also be attached to ACSL behaviors [45]. The semantics of property reachability needs to be clarified before hard-coding it in Frama-C. Nevertheless being able to handle traces directly inside Frama-C would probably help dynamic and symbolic analyzers report invalid properties in a more precise way.

At the architectural level, it seems that a new form of collaboration is emerging: basic blocks implementing generic static analysis to be used in larger analyzers, but not directly interesting for the end-users (*e.g.* light alias analysis or postdominator calculus). In the current Frama-C architectural design, the status of such generic analyzers is not clear. At a first glance, the kernel service directories *analysis* and *ast_transformations* seem dedicated to them, but it is not the spirit of the framework to integrate such extensions directly in the kernel. Furthermore, a few others are either provided through plug-ins like Postdominators, or are directly part of a larger plug-in like E-ACSL or Wp (and often not as generic as possible in this last case).

43. `https://github.com/whitequark/ppx_deriving`
44. `https://gitlab.inria.fr/fpottier/visitors`
45. The private issue # 1256 on that topic is opened on the Frama-C BTS since August 2012.

However, even if it is now possible to export a nice API for these plug-ins [46], a smoother way to integrate them in the Frama-C ecosystem remains necessary to make such plug-ins popular.

Interestingly the new Eva extension mechanism [BBY17] is also a new promising way to provide collaboration through abstract domains: developing specific Eva abstract domains, which take benefit from other Eva domains is currently a hot topic which seems to be not limited to LSL. The best trade-off between hard-coded information and possibility of customization (at the price of more code to be written) is not yet known and will certainly evolve in the next years. It is not just limited to abstract interpretation since, for instance, it opens the door to new direct collaborations between Eva and Wp (beyond the mutual exchange of proved properties) [BBYS18].

At system level, analysis collaboration also raises interesting questions. Section 2.9 already introduced how important, but hard to deal with, are implicit hypotheses. More generally, issues arise when trying to combine different techniques to validate a system, particularly when they do not provide the same level of confidence (*e.g.* when mixing tests and proofs) because it is not clear what is really guaranteed by such a combination of techniques. The basis of a background theory has been defined during the master internship of Antonin Butant [But16] that I supervised in 2016, together with Benoît Boyer and David Mentré from Mitsubishi Electric, Catherine Dubois from ENSIIE, and Virgile Prevosto at LSL. However, this promising work is not yet complete. Another research direction consists in defining coverage criteria that could be applied to determinate which parts of the code and of the specification is covered by a verification process involving both dynamic and static techniques. Such criteria would be particularly useful when combining verification techniques in normative domains like avionics with the standard DO-178C. That is the purpose of the ongoing PhD of Viet Hoang Le that I currently supervise with Loïc Correnson at LSL and Virginie Wiels at Onera [LCSW18].

Systemwide Verification

Remaining at the system level, how to link properties verified on a model and proved ACSL specifications on C code is still an open question. Refinement-based approaches that automatically generate annotated code from a verified model is one possibility but it is not always doable. Research in that direction could unify Frama-C and Papyrus [47], the Eclipse-based modeling environment for model-driven engineering also developed at CEA. It could also involve other tools, such as EasyCrypt and ProVerif when interested in security protocol verification. The former has

46. It was not possible before Frama-C Neon, released in March 2014.
47. `https://eclipse.org/papyrus/`

already been tried [ABBD15] (even if the link between the EasyCrypt model and the ACSL properties is unfortunately not detailed in the reference), while the latter is ongoing work in the context of the French ANR project *AnaStaSec*.

On the practical side, applying Frama-C analyzers to new applicative domains (*e.g.* robotics or medical devices) is usually necessary to improve Frama-C because it becomes exposed to unexpected pieces of code or properties possibly difficult to handle. Among these domains is learning: with the recent achievements of deep learning (*e.g.* the recent advents of AlphaGo in the game of go [SHM+16, SSS+17] to take a single example outside the traditional applicative domains of formal methods), neural networks are more and more used in critical systems to improve their level of automation. This raises the question of verifying such Artificial Intelligence based systems [SSS16]. That is a difficult challenge particularly because neural networks are hard to specify. For instance, their specifications may require probabilistic models. We could also consider the other way around: how these new achievements in machine learning could increase the automation level of program verification tools like Frama-C and so help their dissemination. Together with three European partners and colleagues at LSL, I submitted a European project proposal on that topic in January 2017.

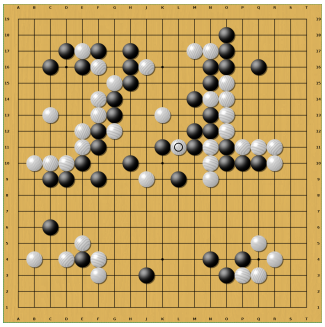
On the other end of the spectrum, Frama-C analyzers and proved ACSL annotations could also be used in optimizing compilers. This idea is not new and has even been already explored using ACSL annotations [GNP+15]. However, the static analysis currently embedded in compilers is either light or incorrect: when they are too aggressive, they do not necessarily preserve the semantics of the input program. I think that there is room for additional research in that direction in order to propose safe aggressive optimizations based on both analysis results and proved formal properties.

Last but not least, C is not the only programming language used in critical systems. Frama-C plug-ins JCard and Clang already deal with (large subsets of) JavaCard and C++ respectively. Another (rather unconclusive) try was also made with Python. In each case, the approach consists in generating a Frama-C AST from an internal representation of the input programming language, so as to compile a program from a given language $\mathcal{L}$ to C. It has the huge benefit to prevent the development of new analyzers for each programming language: "just" analyze the generated code and trace back the results to the original code and that's it. Unfortunately, it also has three main drawbacks. First, the Frama-C analyzers usually have a hard time understanding the generated code, for instance the encoding of a virtual method table through a table of function pointers. Next ACSL does not usually fit all the needs of the input language and so needs to be adapted. It is planned work for C++ with the design of ACSL++, but it is not an easy task. Finally dealing with the runtime of the language and its standard library is also difficult and requires extra effort. To solve these issues (at least the first two ones), a

better approach might be to develop a new AST able to natively express constructs of mainstream programming languages. However, it would increase significantly the number of constructs to be supported by analyzers. To circumvent this issue, dedicated transformations could be developed to remove particular constructs. This is actually what plug-ins Variadic and Volatile already do for C programs. Interestingly, when developing the Clang plug-in, exceptions were added to the Frama-C AST to simplify the translation from the C++ AST. They are automatically removed by the Frama-C kernel when building the AST. It may be seen as a proof that this approach is fully doable. If generalized to other difficult language constructs, it would allow every analyzer to decide to support them natively for better results, or let the front-end replace them by simpler constructs. The effort to develop such a generic AST and the associated transformations is however so large that it might require starting a new project to replace Frama-C by a new tool. Developing such a new tool would also have the benefit to set up a clean architecture.

3
E-ACSL
a Runtime Verification Tool

Tesuji (Skillful Finesse)



Lee Sedol's winning tesuji against AlphaGo
(Game 4 of the match AlphaGo – Lee Sedol, 2016/3/16).

E-acsl was born on 21 February 2011. It is the runtime verification plug-in of Frama-C and certainly one of my most important contributions to this framework. It is currently my main research topic. This chapter details this tool, from its research context (Section 3.1) to its perspectives (Section 3.5) through its specification language (Section 3.2), its main current features (Section 3.3), and its usages (Section 3.4).

3.1 E-ACSL, a Tool on a Young Research Domain

Context This section is novel, but takes inspiration from existing surveys on runtime verification [HG05, FHR13], runtime assertion checking [CR06] and dynamic memory analysis [LCH+06].

I present here a short overview of runtime verification, runtime assertion checking and dynamic memory analysis, as well as the relationship between these topics.

It does not aim at being complete, but at introducing the context in which E-ACSL evolves.

3.1.1 Runtime Verification

As a research domain, runtime verification is the study of rigorous techniques that analyze the behavior of programs or systems during their executions. The overall goal of this research domain is to improve the confidence in such programs and systems by discovering defects with respect to their intended behaviors expressed as formal specifications. Runtime verification may be seen as *lightweight formal verification*. Indeed runtime verification techniques are usually based on formal reasoning, but they are often lighter (so less expensive) than static formal program analysis techniques because they only focus on executing some program traces without checking all of them. They are also able to find more defects than standard testing techniques — at similar costs — because the extra properties checked at runtime are often invisible from test oracles.

According to Klaus Havelund and Allen Goldberg [HG05], the terminology *runtime verification* was introduced in 2001 through the first workshop on Runtime Verification [1]. This workshop may be seen as the starting point of runtime verification as an independent research domain with its own community of researchers and practitioners. The workshop series on Runtime Verification was converted into an international conference series in 2010. This attests to the development of the field which is still ongoing: since 2014, the European ICT Cost Action *ARVI* [2] — which I am involved in — aims at developing the European expertise in the field by grouping together its active European researchers. Notably, this project already contributes to the International Runtime Verification Competition (CRV) [BFB+17] since 2014, and initiates the first Summer School on Runtime Verification in 2016 [CF16]. *Ergo* Runtime Verification is still a young growing research domain.

3.1.2 Runtime Assertion Checking

Despite runtime verification being a young scientific discipline, verifying properties at runtime is definitively not a recent idea with its origin in the seventies, at the time of the birth of the monotheistic religions of static program analysis. Indeed, as explained in Section 2.1, assertions are as old as programs [GvN47]. At that pioneering time, assertions were just Boolean expressions. They were later extended to first-order logic by Robert W. Floyd [Flo67]. Since infinitely many terms of first-order logic cannot be evaluated in finite time by a computer, that is an indication that the primary goal of assertions was *not* runtime evaluation, but

1. `http://runtime-verification.org/`
2. `https://www.cost-arvi.eu/`

program documentation and formal reasoning for establishing program correctness as stated by Lori A. Clarke and David S. Rosemblum [CR06]. The wonderful historical survey of Cliff B. Jones [Jon03] provides additional evidences about the initial need of assertions for mathematical reasoning instead of runtime verification.

Lori A. Clarke and David S. Rosemblum explain that the idea of monitoring properties at runtime emerged in the mid seventies in the programming language and compiler communities as a simpler and more practical alternative than formal proofs of correctness [CR06]. Since runtime checks were usually introduced through a dedicated *assert* language construct or a specific *assert* preprocessor macro, the wording *runtime assertion checking* became popular even if the expression *self-checking programs* was also used for a while [YC75]. Such runtime assertions were regularly introduced in academic programming languages and systems in the late seventies and during the eighties, notably through extensions of Fortran [SF75] and Ada [LVH85, Luc90]. They were eventually added to the C programming language through the macro `assert` and were popularized to support *defensive programming* by the commercial programming language Eiffel [Mey88, Mey92b]. Eiffel was also one of the first language to propose Bertrand Meyer's *design by contract* approach [Mey92a], which lifts assertions from statements to functions and larger components like objects and modules. At the turn of the millennium, this approach was adopted by dedicated specification languages for mainstream programming languages like JML [LBR99] for Java, Spec# [BLS04] for C# and, of course, ACSL [BFM+] for C. For the latter, it is worth noting that it goes back to sources of assertions by using them for formal reasoning only. The implications of this fundamental design choice will be discussed in Section 3.2.

3.1.3 Dynamic Memory Analysis

While checking monitoring properties at runtime concerns both the compiler and the programming language communities, the former is not that much involved in runtime assertion checking, which is more related to the design of specification languages as explained above. It is more concerned with monitoring properties induced by the programming language semantics that do not involve a manual process such as writing assertions. In low level programming languages such as C, these are often memory-related properties. Indeed, in such languages, most programmer mistakes come from trying to access an invalid memory location, *e.g.* when dereferencing the `null` pointer. This important concern led to the development of *dynamic memory analyzers*, also known as *memory debuggers*, which are tools specialized to find defects when accessing memory at runtime by instrumenting the program code either at source code level or at binary level.

In the seventies, Jung-Chang Huang provided one of the first methods (if not the first method) to detect memory errors through source code instrumenta-

tion [Hua79]. Since that time, lots of tools have been released [LCH+06], almost all of them for debugging C and C++ programs. There are about as many commercial tools as free software. It may be seen as an evidence of the need for such tools in the software industry. Today the commercial tools Purify [HJ92] and Insure++ [3], as well as the free software MemCheck [SN05] embedded in Valgrind [NS07], Dr. Memory [BZ11], and AddressSanitizer [SBPV12] are certainly among the most famous dynamic memory analyzers. The latest one, provided by Google, is currently considered to be the most efficient one and is integrated within Clang and, recently, Gcc.

3.1.4 Runtime Verification *vs* Runtime Assertion Checking *vs* Dynamic Memory Analysis

From the above definitions of runtime verification, runtime assertion checking and dynamic memory analysis, it should be clear that the latter two are sub-fields of the former. However, there are currently major differences between them which has led to three almost separate communities.

First, there is no intersection between runtime assertion checking and dynamic memory analysis: as far as I know, no general-purpose formal specification language for runtime assertion checking (except E-ACSL, see Section 3.2.9) provides support for verifying memory properties, while dynamic memory analyzers do not require their users to write annotations. Beyond this, researchers on runtime assertion checking are mainly concerned with expressiveness and semantics of specification languages, while researchers on dynamic memory analysis are more concerned with time and memory efficiency in terms of overhead with respect to the original (uninstrumented) program. Runtime verification stands in the middle, being concerned with finding the right balance between efficiency and expressiveness [FHR13].

Next, researchers of these fields are usually not concerned with the same properties: runtime assertion checking is concerned with functional (safety) properties, dynamic memory analysis is concerned with memory properties (several of them leading to common security vulnerabilities), while the runtime verification community leans towards temporal properties because lots of its researchers come from the model checking community. To support my latest claim, I could rely one more time upon Lori A. Clarke and David S. Rosemblum who write that "assertions [...] are an important element of model checking in which [...] a program's execution is checked against logical assertions expressing temporal safety and liveness properties" [CR06], but also upon Martin Leucker and Christian Schallhart in their brief account of runtime verification [LS09b]:

> As runtime verification has its roots in model checking, often some variant of linear temporal logic, such as LTL [Pnu77], is employed.

3. `https://www.parasoft.com/product/insure/`

Finally, runtime verification relies on notions of *events* and *traces* in addition to properties to be checked. Indeed, the general goal of a runtime verifier may be expressed by its capability to check that a runtime *trace* τ of a program, τ being represented by a finite sequence of (possibly parametric) *events*, satisfies a specification ϕ which denotes a set of correct behaviours of the program. One current challenge of the runtime verification community in this respect is to provide definitions of events and traces that would be general enough to capture most usages [RH16]. However, these notions usually remain totally implicit in runtime assertion checkers and dynamic memory analyzers for which an event is a program state associating a value to every program variable (or, more generally, memory location) and the trace of interest is the sequence of states from the entry point of the program up to the current execution point.

However, the above-mentioned differences between the three communities tend to keep them relatively separated and make exchanges and discussions sometimes more complicated than they should be. Nevertheless the ongoing effort of the runtime verification community for uniformizing languages and definitions, in particular through the ICT action *ARVI*, is definitively a step in the right direction. Since E-ACSL is both a runtime assertion checker and a dynamic memory analyzer, while also being able to deal with temporal properties and even more (as the following sections will hopefully demonstrate), I hope that the ongoing research effort on this tool might contribute to making collaborations between these communities easier and fruitful.

3.2 E-ACSL, an Executable Formal Specification Language

Context This section is not as formal as the ACSL and E-ACSL reference manuals [BFM+, Siga] and does not provide as many examples as the excellent Fraunhofer Fokus' reference tutorial *ACSL by Example* [BCGP]. However, as far as I know, there is no overview of ACSL and E-ACSL that provides as many explanations about the design of these languages than this section. It is inspired by existing shorter presentations of the E-ACSL language that I contributed to [DKS13, KS13, KMSM16].

The E-ACSL tool [KS13, SV] is a Frama-C plug-in that translates annotations written in a dedicated formal specification language to C code in order to check them at runtime. This section presents this specification language, also named E-ACSL [DKS13, Siga]. In the rest of this chapter, I will simply write E-ACSL to denote either the E-ACSL tool or the E-ACSL specification language and let the context discriminate the former from the latter.

In 2011, I designed the E-ACSL specification language as an executable subset of ACSL. Indeed, unlike JML, ACSL was primarily designed for program proof

and thus contains logical constructs that cannot be executed in finite time (*e.g.* unbounded quantification over mathematical integers or reals). Therefore, such constructs should be removed from the executable subset that aims to be translated to C code to be evaluated (in finite time) at runtime. Consequently, E-ACSL is a specification language for dynamic verification which comes from a specification language for static verification. It is worth noting that it was designed in the context of the *Hi-Lite* project in which Adacore also designed the Spark2014 specification language by extending its former 2005 version, in order to prove formal properties of Spark2014 programs and also test them by runtime assertion checking. Indeed both E-ACSL and Spark2014 share the common goal of filling the gap between static analysis and runtime verification tools. For both, the first step in that direction was to design a specification language that could be used for both program proof and runtime assertion checking [KMSM16]. It also worth noting that E-ACSL seems also to be a reasonable ACSL subset to be analyzed by abstract interpretation (*e.g.* *via* Eva).

3.2.1 Semantics

As already explained, E-ACSL aims at being a subset of ACSL in which each construct should be executable in finite time. In particular, one could expect both languages to share a common semantics for their common subset. However, this is not quite the case because their interpretations of partial functions differ.

Indeed, ACSL is based on standard (two-valued) mathematical logic in which partial functions are modeled by underspecified total functions. For instance, integer division is a total function that returns some unspecified value when the denominator is 0. This interpretation was introduced by David Gries and Fred B. Schneider [GS95] and is the most standard approach for specification languages dedicated to static verification like ACSL. Its nicest property is to preserve standard mathematical properties, notably reflexivity of equality. For instance, the predicate $1/0 \equiv 1/0$ holds with such an interpretation. It helps deductive verifiers like Wp and their associated theorem provers to soundly deal with partial functions: nothing wrong can be proved. At worst, underspecified terms lead to unprovable properties, or tautologies. For instance, no interesting properties containing $1/0$ can be proved because the semantic value of this term is unspecified.

However, the term $1/0$ and consequently the predicate $1/0 \equiv 1/0$ would result into runtime failures if one tries to evaluate them dynamically. This problem is well known and named the *undefinedness problem* by Yoonsik Cheon who developed the first [4] JML runtime assertion checker in 2003 [Che03, Section 3.2]. To solve this foundational issue, E-ACSL relies on Patrice Chalin's *strong validity* [Cha07]: instead of representing partial functions by underspecified total functions, E-ACSL

4. As far as I am aware of.

relies on a three-valued logic that introduces undefinedness as truth value. When a partial function is applied outside its definition domain, the result is undefined. Consequently the semantics of $1/0$ is undefined. By extension, any term or predicate that contains an undefined term is also undefined. For instance, the predicate $1/0 \equiv 1/0$ is not valid anymore but undefined. More generally, a predicate is (strongly) valid if and only if (1) it is well defined and (2) it holds. This semantics is actually consistent with the ACSL one as stated below.

Conjecture 3.1 (Mutual consistency of ACSL and E-ACSL) *For any E-ACSL predicate p, if p is valid (resp. invalid) in ACSL then p is either valid (resp. invalid) or undefined in E-ACSL. Conversely, if p is valid (resp. invalid) in E-ACSL then p is also valid (resp. invalid) in ACSL.*

A proof of this claim would require formalizing ACSL and E-ACSL in the same setting. This is not done yet, although Paolo Herms provided a formalization of a subset of ACSL in Coq [HMM12, Her13] and Guillaume Petiot formalized a language very close to E-ACSL [Pet15]. In its own context, Patrice Chalin formally proved a similar claim in HOL (only for the implication corresponding to E-ACSL to ACSL in Conjecture 3.1) [Cha09] [5]. This fundamental property ensures tool compatibility between ACSL and E-ACSL since any well-defined term has the same meaning in both languages.

An indirect consequence of strong validity is laziness of most binary and ternary logical operators (namely, conjunction `&&`, disjunction `||`, exclusive disjunction `^^`, conditional `_?_:_` and implication `==>`). Indeed, laziness introduces fewer undefined terms since the dynamic semantics of these operators relies on fewer (possibly undefined) subterms. For instance, the E-ACSL predicate `\true || 1/0 != 1/0` is (strongly) valid thanks to laziness. This semantics is also consistent with the C semantics of binary logical operators. Thus, it makes the translation from annotation to C easier. Another advantage of this semantics is that it is more natural for practitioners as pointed out by Patrice Chalin [Cha05].

The rest of this section presents the most important E-ACSL constructs with illustrative examples. Some of these examples aim at explaning a few pitfalls of the E-ACSL semantics, in addition to making it more concrete. Also, the last subsection compares E-ACSL with other formal specification languages for programming languages.

5. More precisely, this 2009 paper refers to a HOL proof that should have been accessible at that time at a given URL. Unfortunately, I was unable to freely access to it eight years later in order to check what was precisely proved. I can only trust what is written in the paper.

3.2.2 Assertions

Assertions, introduced through the keyword `assert` right before C statements, are the simplest way to express E-ACSL properties as shown in the two code snippets of Figure 3.1. They just ensure that a given predicate holds at a particular program point.

```
int x = gcd(y, z);
/*@ assert x != 0; */

/*@ assert y % x == 0; */
prime = y / x;
```

```
int x = gcd(y, z);
/*@ assert x != 0; */
; /* skip */
/*@ assert y % x == 0; */
prime = y / x;
```

Figure 3.1: Annotation Evaluation Orderings.

Here, the two annotated pieces of code look equivalent but they are not. Indeed, in the first snippet, both annotations are attached to the program point right before the very same statement. It means that their order of evaluation is left unspecified; so there is no guarantee that `x` is different from 0 when evaluating it as the divisor of the modulo operator. To ensure that an annotation is evaluated before another one, it must be attached to an earlier program point in the program's execution flow as in the second code snippet thanks to the extra semi-column that introduces a C skip statement between both annotations. Thanks to laziness, another solution is to write a single annotation with a conjunctive predicate `x != 0 && y % x == 0`. This behavior is consistent with the C notion of sequence point that precisely defines where C side effects are applied [6].

3.2.3 Contracts

E-ACSL being a behavioral interface specification language, one of its most important features is the notion of *contract* between a service provider and its clients: whenever a client guarantees some *preconditions* to be true before using the service, the provider ensures some *postconditions* to hold when leaving it. E-ACSL contracts are available at function level and at statement level. In the first case, contracts bind function callees to their callers. In the second case, they bind C statements (usually blocks) to their surrounding blocks. E-ACSL logical preconditions are introduced through keyword `requires`, while E-ACSL logical postconditions are introduced through keyword `ensures`. Clause `assigns` is another kind of postcondition that is known as a framing specification [HLL+12]. It specifies the memory locations that a function (or a statement) *may* modify. It is particularly convenient to describe in this manner changes in the program state, since most parts of the

6. See C99 standard, §5.1.2.3, p.13 and Annex C, p.438.

memory remain unchanged when calling a function [BMR95] (or, more generally, when executing a small piece of code). It may be extended with a `from` subclause that specifies the memory locations that *may* be involved in the computation. For instance, `assigns x,*p from y,*p,p` specifies that (1) `x` and `*p` are the only possibly modified memory locations and (2) their new values may only be computed from the values of `y`, `*p` and `p`. Function contracts have already been exemplified in Figure 2.3: if one ignores its axiomatized function for the moment (discussed later), this ACSL contract is also an E-ACSL contract.

E-ACSL also provides the notion of *behaviors*. It is a convenient way to split a contract into several cases. In addition to increasing the readability of postconditions, using behaviors allows the user to specify completeness and disjointness of behaviors which are most of the time desired properties over behaviors. Completeness indicates that no possible behavior is missed, while disjointness expresses that the behaviors are pairwise disjoint: if the behaviors are both complete and disjoint, the specification of the given function or statement for a particular input is defined by exactly one behavior.

Figure 3.2 shows an example of a function contract containing three behaviors. An informal specification of this function is as follows:

> `swap(a, len, idx, value)` takes as inputs an array `a` of length `len`, an index `idx` and a given `value`. If `value` belongs to `a` at some index `i`, then the function swaps the values at indices `idx` and `i` and returns `i`. Otherwise, this function just returns `-1`.

The two preconditions introduced by the clause `requires` are common to all behaviors and express that the cells of the array `a` must be valid up to `len` (excluded) and the index `idx` must be in bounds. Next, two different cases can be directly derived from the informal specification: either the given `value` belongs to `a`, or not. The latter case is formalized by the behavior `not_found` activated whenever its clause `assumes` is valid (that is, when no cell of array `a` contains the given `value`). The former case is split into two different subcases that are not explicit in the informal description: either the searched index `i` is equal to `idx`, or not. The former subcase is the behavior `found_same_index` in which nothing happens but returning `idx`. The latter subcase is the behavior `found_and_replace` which corresponds to the standard case in which a swap occurs. Its third postcondition uses the E-ACSL keyword `\old` that refers to the value of its argument (here `a[idx]`) in the pre-state of the function. This construct is usual in behavioral specification language and is necessary in this particular example since `a[idx]` is a memory location modified by the function: one must distinguish its value before entering the function (*i.e.* in the pre-state) and when leaving it (*i.e.* in the post-state). In E-ACSL, `\old(t)` is just a syntactic sugar for `\at(t, Pre)`: in the general case, the term `\at(t, L)` denotes the value of `t` at label `L`, `L` being either a standard C label, or a logical label. Here, `Pre` is the logical label denoting the pre-state. Finally, the clauses `complete`

```
/*@ predicate exists_val(int *a, integer len, integer value) =
  @   \exists integer i; 0 <= i < len && a[i] == value;
  @*/

/*@ requires \valid(a+(0..len-1));
  @ requires 0 <= idx < len;
  @
  @ behavior not_found:
  @   assumes ! exists_val(a, len, value);
  @   assigns \nothing;
  @   ensures \result == -1;
  @
  @ behavior found_same_index:
  @   assumes exists_val(a, len, value);
  @   assumes a[idx] == value;
  @   assigns \nothing;
  @   ensures \result == idx;
  @
  @ behavior found_and_replace:
  @   assumes exists_val(a, len, value);
  @   assumes a[idx] != value;
  @   assigns a[idx], a[\result];
  @   ensures a[idx] == value;
  @   ensures 0 <= \result < len;
  @   ensures a[\result] == \old(a[idx]);
  @
  @ complete behaviors;
  @ disjoint behaviors;
  @*/
int swap(int* a, int len, int idx, int value);
```

Figure 3.2: Example of E-ACSL behaviors.

`behaviors` and `disjoint behaviors` of the function contract specify that the three behaviors must be both complete and pairwise disjoint as previously explained.

3.2.4 Integer Arithmetic

E-ACSL is based on mathematical numbers and operators. It means that each E-ACSL integer constant represents a $\mathbb{Z}$ mathematical integer constant, while E-ACSL integer operations are computed over $\mathbb{Z}$ and do not rely on any (bounded) machine arithmetic. The E-ACSL type of mathematical integer is `integer`. A subtyping system automatically coerces C integral types to `integer` whenever necessary. For instance, if `x` is a C variable of type `unsigned short` and `y` is another C variable of type `char` in the E-ACSL term t defined by `(x + 1) / y`, `x` and `y` are both automatically coerced to `integer` in order to compute the addition and the division over $\mathbb{Z}$. They would be promoted to `int` if t would be a C expression instead of an E-ACSL term[7]. In particular, neither the addition nor the division can overflow in t (but dividing by zero is still undefined according to the E-ACSL's three-valued semantics). Also, the constant 1 is of type `integer`, while it would be of type `int` in a C expression.

This important design choice is inherited from ACSL and has several advantages. First, automatic theorem provers usually work better with mathematical arithmetic than with bounded arithmetic because the latter requires additional hypotheses for the bounds. That is the main *raison d'être* of using mathematical arithmetic in ACSL. Arguably, this feature is not as important for a specification language dedicated to runtime verification such as E-ACSL, but it is important for consistency between ACSL and E-ACSL as stated by Conjecture 3.1. This argument is not only theoretical since it comes into play when combining static analysis (*e.g.*, with plug-in Wp) and dynamic analysis (*e.g.*, with plug-in E-ACSL). Second, it is usually considered good engineering practice to write specifications without any implementation detail in mind, but dealing with potential overflows in specification can be seen as an implementation detail. Third, it is still possible to use bounded arithmetic if necessary by using explicit casts. For instance, rewriting the term t of the previous paragraph to `(int)((int)(x + 1) / y)` would enforce coercing the mathematical results of the addition and of the division to `int` values. If there is no overflow, the result is equivalent to computing the operation over `int` as a C compiler does. Last but not least, the choice of relying on mathematical operators simplifies specifying overflows. For instance, the assertion `/*@ assert INT_MIN <= x + y <= INT_MAX; * /` specifies in the easiest way that the C operation `x + y` over `int` will not overflow by computing it over $\mathbb{Z}$ and checking that the result is in the bounds of the C type `int`. Also, Patrice Chalin introduced several concrete examples motivating the use of mathematical integers in a formal specification

7. See C99 standard, §6.3.1.1, p.43.

language [Cha04]. There is however one major drawback to using mathematical operators instead of (bounded) machine operators: they are more complicated and less efficient to compute at runtime. This issue is discussed in Section 3.3.3.

In addition to `integer`, E-ACSL also includes the type `boolean` with its constants `\true` and `\false`. Integers are automatically coerced to Booleans whenever required, while an explicit cast is required to convert `boolean` to a C integral type or `integer` (`\true` [resp. `\false`] being converted to 1 [resp. 0]). For instance, assuming `x` and `y` of type `int`, the predicate `1 <= (char) (x && y) + 1 <= 2` is a tautology, equivalent to `1 <= (char)(x != 0 && y != 0) + 1 <= 2`.

3.2.5 Real Numbers and Floating-Point Numbers

Similarly to integers, E-ACSL includes the type `real` to denote the set $\mathbb{R}$ of mathematical reals. The C floating-point types `float` and `double`, as well as `integer` and its subtypes, are subtypes of `real` with implicit casts from these types to `real` when necessary. The C floating-point constants and operations used in E-ACSL are interpreted over $\mathbb{R}$. For instance, the E-ACSL term `2 * 0.1` is interpreted as the real `0.2` and not to any of its floating-point approximations.

The casts from reals to floating-point types depend on the rounding mode which is one of `\Up`, `\Down`, `\ToZero`, `\NearestAway` and `\NearestEven` (the latter by default). These E-ACSL constants correspond to the floating-point rounding modes defined by the IEEE 754 standard. In case of large reals, the casts could also lead to `+infinity` or `-infinity`.

This design choice comes from ACSL with similar benefits and drawbacks as using mathematical integers. It is even worse to verify them at runtime (see Section 3.3.3 for a discussion). E-ACSL also introduces built-in predicates for comparing floating-point numbers such as `\eq_float` and for checking whether an argument is a finite number or `NaN` such as `\is_finite`.

3.2.6 Quantifications

ACSL and E-ACSL are based on first-order logic: their predicates may include universal and existential quantifications. Such quantifications have already been illustrated in Figures 2.3, 2.6 and 3.2. There is yet anothere difference between the two languages: while an ACSL predicate may contain unbounded quantifications, E-ACSL quantifications are restricted to bounded quantifications. For instance, the ACSL predicate `\forall integer x, \exists integer y, y == x + 1` does not belong to E-ACSL because the quantified variables `x` and `y` range over the infinite set of mathematical integers. This restriction is necessary to evaluate quantifications at runtime in finite time.

More precisely, in E-ACSL, quantified variables must be of type `integer` or one of its subtypes and must be syntactically guarded: the general form of an universal quantification is

```
\forall τ x1, ..., xn;
  a1 <= x1 <= b1 ... && an <= xn <= bn
  ==> p
```

while the general form of an existential quantification is

```
\exists τ x1, ..., xn;
  a1 <= x1 <= b1 ... && an <= xn <= bn
  && p
```

In these formulæ, τ denotes a subtype of `integer` and `p` is a predicate that may use the quantified variables `x1`, ..., `xn`. Each variable `xi` must be syntactically bounded by a guard `ai <= xi <= bi` [8] in which `ai` and `bi` are E-ACSL terms. It is worth noting that these terms are not necessarily constants and may contain other variables, in particular other quantified variables already guarded. This syntactic restriction is not an issue in practice: almost all quantifications over integers in C programs are naturally bounded (for instance, when ranging over the index of an array). It might be not the case for a few mathematical properties, but artificial bounds can usually be added rather easily in practice, by taking into account the global context of the application.

As an exception to this syntactic restriction, E-ACSL actually accepts unguarded quantifications over Booleans and characters, since these types do not contain many values and may be efficiently enumerated on any computer.

Since quantifying over non-integer types is often useful, E-ACSL introduces a notion of *iterators*. Iterators can only serve as guards for (typically non-integer) quantified variables. An iterator over a type τ is defined by a prototype of a predicate that takes two arguments of type τ. This prototype is associated with two fields `nexts` and `guards`. The former is a set of terms of type τ representing the successors of the second parameter. The latter is a set of predicates over τ. This set must have the same cardinal as `nexts`: the iteration continues over the i-th successor if and only if the i-th predicate of `guards` holds. Consider for instance the iterator `access` over binary trees shown in Figure 3.3: its field `nexts` contains two terms in order to access the left and right subtrees when iterating, while its field `guards` specifies that the iteration shall continue as long as the next subtree is valid (so in particular when it is not the `NULL` pointer).

The first parameter of the iterator is useless when defining it (so it is replaced by a wildcard in our example), but, when using an iterator π in a quantification, it

8. Actually each relational operator `<=` may be replaced by `<` or `==`.

```
struct btree {
  int val;
  struct btree *left, *right;
};

/*@ iterator access(_, struct btree *t):
  @   nexts t->left, t->right;
  @   guards \valid(t->left), \valid(t->right); */
```

Figure 3.3: E-ACSL iterator over binary trees.

binds the quantified variable to π. For instance, assuming that `t` is an existing valid binary tree, the following predicate holds if and only if every value of `t` is even.

```
\forall struct btree *subtree;
  access(subtree, t) ==> subtree->val % 2 == 0;
```

It is worth noting that this definition does not prevent infinitely iterate over cyclic trees (so, graphs, stricly speaking). It is left to future work to add either syntactic restrictions, or dynamic checks to prevent infinite recursions at runtime when using such iterators.

These iterators are actually the only extension of E-ACSL to ACSL since they do not exist in this language. However, nothing prevents extending ACSL with them.

Quantifications are not only useful when writing contracts, but also when writing loop invariants, in particular when the loop iterates over a collection such as an array or a linked list. Notably, Bertrand Meyer "feels that the lack of quantification has meant that Eiffel programmers rarely use Eiffel's loop invariant" [CR06].

3.2.7 Loop Invariants

Loop invariants are necessary properties when proving program properties whose (partial) correctness relies on a loop computation. They are properties that must hold before entering the loop and remain true at the end of each loop iteration. For program proving, they make explicit the induction hypotheses that are necessary for proving the global properties of interest. In particular, ACSL loop invariants are inductive properties: in order to establish their preservation at the end of the loop body, one may assume that they hold at the beginning of the loop body. Their usage for program proving has already been illustrated in Figure 2.6.

In E-ACSL, loop invariants lose their inductive nature (useless at runtime) in the nominal case: they are strictly equivalent to two assertions; one written just before entering the loop condition, and one written at the end of the loop body.

3.2.8 Sets of Terms

Quantifications are often required but may be quite verbose to write. They can actually be avoided many times thanks to ranges. Ranges have already been illustrated in Figures 2.3 and 3.2. For instance, in the latter example, the first precondition `\valid(a+(0..len-1))` uses a range `0..len-1` to express that all the cells of the array `a` between `0` and `len-1` are valid. It is equivalent to the predicate `\forall integer i; 0 <= i < len ==> \valid(a+i)`, but shorter.

More generally, E-ACSL allows the user to write a set of terms (called a *tset*) instead of a term t whenever t denotes a memory location. That is for instance the case in `assigns` clauses and in built-in memory logic functions and predicates such as `\valid`. A tset may be the `\empty` set, a simple term denoting a singleton, ranges as previously shown, a union (resp. intersection) of sets through the built-in logic functions `\union` and `\inter` with a finite list of tsets as arguments. It can also be defined by comprehension. For instance the tset `{ 2 * n | integer n; 0 <= n < 128 && n % 3 == 0 }` denotes the set of even natural numbers divisible by 6 and smaller than 2×128, that is 256. It is worth noting that the local binder of the comprehension (here `n`) is properly guarded to be sure to define a finite set. Tsets may also be defined by extension of terms. For instance the tset `a[0..n-1].b` denotes the set of fields `b` of every array cell of `a` between 0 and `n-1`, or equivalently `{ a[i].b | integer i; 0 <= i <= n-1 }`.

E-ACSL also provides constructs to support finite sequences of elements (also known as lists) over some type `A` (through parametric polymorphism). Since this part of the language is currently almost not used, I do not cover this part in more details here. The interested reader can refer to the ACSL and E-ACSL reference manuals [BFM+, Siga, Section 2.8.2].

3.2.9 Memory Properties

Verifying memory properties in a programming language such as C is of the utmost importance since incorrectly accessing a memory location is one of the primary causes of bugs in C programs. Many of them are also considered security vulnerabilities. For instance, looking for pattern "C memory" on the MITRE's CVE [9] search engine [10] returns 20,400 results [11]. This pattern is arguably imprecise and I have not verified each result individually but, at least, the first page of results [12] seems to refer to invalid memory accesses in C programs in tools ImageMagic [13], Graphic-

9. Common Vulnerabilities and Exposures.
10. `https://cve.mitre.org/find/`
11. Search done the 1st of December 2017, at 19:08, UTC+1.
12. CVE-2016-8862, CVE-2016-8866, CVE-2016-9830, CVE-2017-14042, CVE-2016-8684, CVE-2016-5114, CVE-2017-5505, CVE-2017-5503, and CVE-2009-2626.
13. `http://www.imagemagick.org/`

smagick [14], PHP [15], and JasPer [16]. Consequently E-ACSL needs to be able to easily express memory properties preventing such vulnerabilities.

I already introduced several constructs dedicated to memory properties, such as `assigns` and `loop assigns` clauses, `\at` and `\old`, as well as the built-in predicate `\valid` expressing that its argument must be a valid pointer (that is, a non-null pointer that refers to a memory location that the program can safely read or write). Most memory properties are described through built-in predicates and logic functions (or constants) in practice. They are presented in Figure 3.4 (by order of importance to my eyes). As explained in the previous subsection, these built-ins are often used with sets of pointers as arguments instead of single memory locations. Also, each of them is parameterized by an optional label `L` (but `\null` takes none, while `\fresh` takes two labels as explained below). For instance, in a contract, `\valid{Pre}(p)` refers to the validity of pointer `p` in the pre-state, so it is equivalent to `\old(\valid(p))`.

predicates	logic functions and constant
`\valid`	`\null`
`\valid_read`	`\block_length`
`\initialized`	`\base_addr`
`\separated`	`\offset`
`\freeable`	`\allocation`
`\allocable`	
`\dangling`	
`\fresh`	

Figure 3.4: E-ACSL memory built-in logic functions and predicates.

Predicate `\valid_read` is similar to `\valid` but only checks that the given pointer argument is readable, so that the pointed value is not modifiable. Literal strings (of type `char *`) are the typical example of read-only memory locations. For instance, a function that takes a literal string as argument is not allowed to modify it but may safely read it. Predicate `\initialized` indicates that its pointer argument points to a fully initialized memory location. For instance, `\initialized(&a[0])` holds if and only if the first cell of the C array `a` has been initialized. It is worth noting that it is not equivalent to `\initialized(a)` which would be incorrect. Indeed, `a` is a C array implicitly coerced to a logic array when used as a term, while there is no implicit coercion from logic arrays to C pointers.

14. `http://www.graphicsmagick.org/`
15. `http://php.net/`
16. `https://jasperproject.github.io/`

Predicate `\separated` holds if and only if its two arguments, which are memore locations, are different. It is particularly interesting when its arguments are memory regions defined by sets of memory locations in order to express memory separation. For instance, `memcpy` assumes that its source `src` and its destination `dest` (both of type `void *`) do not overlap before copying `n` bytes from `src` to `dest`. It can be specified by the precondition `\separated(((char *)src)+(0..n-1), ((char *)dest)+(0..n-1))` [17]. Predicate `\freeable` holds if and only if its pointer argument may be successfully released by calling the C function `free`: it is actually used as a precondition of this function. Similarly, predicate `\allocable` holds if and only if its pointer as argument points to a memory location that is a base address of an unallocated memory block (in the program memory). It is actually a postcondition of the very same function `free`: after releasing a freeable pointer `p` through a call to `free`, `\allocable(p)` holds since `p` may be safely allocated again. Predicate `\dangling` indicates that its argument is a pointer that points to a dangling pointer (note the extra level of indirection). A dangling pointer is a pointer that was valid at the program point in the past but now points to a memory location that is not accessible anymore. For instance, it could point to a local variable that is now out of scope or to a dynamically-allocated memory block that has been deallocated in the meantime. Figure 3.5 exemplifies both cases with a small C program containing two valid assertions that use `\dangling`. Finally, predicate `\fresh` takes two labels L_1 and L_2 (in a contract, they are optional and respectively denote the pre- and the post-states by default), a pointer `p` and a size `n`. It indicates that `p` was allocable at L_1, is freeable at L_2, and that the length of the memory block pointed to by `p` at L_2 is equals to `n`. Such a behavior maps most actual behaviors of C allocation functions such as `malloc` when they succeed (that is, when they do not return the `NULL` pointer).

Let us now explain the built-in logic constant and functions. Logic constant `\null` is equivalent to the C constant `NULL`. Now, assume some pointer `p` that can be safely read and points to some memory block m. Then `\block_length(p)` denotes the length of m, `\base_addr(p)` denotes the base address of m, that is the very first memory location of m, and `\offset(p)` denotes the difference between `p` and its base address. Finally, `\allocation(p)` returns the allocation status of m, that is one constant in the set { `\static`, `\register`, `\automatic`, `\dynamic`, `\unallocated` }. They refer to all the possible ways to (un)allocate a memory block in C programs.

Verifying at runtime predicates using such memory built-ins is hard. This will be covered in Section 3.3.4.

17. In this example, both casts to `char *` ensure that the size of each memory cell is exactly one byte.

```
#include <stdlib.h>

int main(void) {
  int *p;

  {
    int a = 0;
    p = &a;
  } /* p becomes dangling when exiting this scope */
  /*@ assert \dangling(&p); */

  int *q = (int *)malloc(sizeof(int) * 10);
  p = q + 2;
  free(q);  /* p becomes dangling when releasing q */
  /*@ assert \dangling(&p); */

  return 0;
}
```

Figure 3.5: Examples of C dangling pointers.

3.2.10 Data Invariants

Data invariants allow the user to specify predicates over a particular set of memory locations, in particular those of a common datatype, that must hold during the whole program execution. They come in $2 \times 2 = 4$ flavors in E-ACSL: strong *versus* weak invariants on one hand, and global *versus* type invariants on the other hand.

Strong and weak invariants indicate the program points where the invariant must hold: a strong invariant must hold at any sequence point, while a weak invariant must only hold at function boundaries, that is when entering and leaving a function, but may be temporary broken inside function bodies.

A global invariant is an invariant over particular global variables, while a type invariant is an invariant that applies over all the variables of some specified static types.

Figure 3.6 illustrates a weak global invariant and a strong type invariant. The former specifies that global variable `cash` must stay nonnegative and below a given `MAX_AMOUNT`, while the latter indicates that the field `age` of every person `p` must be nonnegative.

```
#define MAX_AMOUNT 1000
long cash;
/*@ weak global invariant always_has_cash: 0 <= cash <= MAX_AMOUNT; */

typedef struct { char *name; short age; } person;
/*@ strong type invariant is_born(person p) = p.age >= 0; */
```

Figure 3.6: Examples of E-ACSL data invariants.

3.2.11 Termination Properties

It is not possible to verify in finite time whether a program terminates (or not) at runtime. Consequently, the ACSL clause `terminates` in a contract, that specifies a sufficient condition guaranteeing the function (or statement) termination, does not exist in E-ACSL.

However, it is still possible to specify a `loop variant` in order to verify that each loop iteration makes the variant strictly decrease. Most of the time, the variant is a positive integer as illustrated in Figure 2.6 (page 31). It may also be more general through the syntax `loop variant t for R;` that allows specifying a general measure R for t of type τ. For instance, the default variant for integers assumes that R is defined by $R(\mathtt{x},\mathtt{y}) \equiv \mathtt{x >= 0 \&\& x > y}$. For recursive functions, it is also possible to specify a clause `decreases` in their contract similarly to loop variants for loops.

3.2.12 Logic Specifications

E-ACSL also provides several constructs to enrich specifications with high-level features. For instance, E-ACSL includes many functional features *à la* OCaml: lists and parametric polymorphism (already mentioned), lambda terms and built-in high-order functions `\sum`, `\product`, `\min`, `\max` and `\numof`, pattern matching, local bindings, tuples and functional updates of structures and arrays. They help write more readable and expressive specifications.

E-ACSL also provides user-defined, possibly-recursive logic functions and predicates. They can be parameterized by labels similarly to built-ins such as `\valid`. For instance, Figure 3.7 shows a recursive logic function `rec_sum` that is similar to the axiomatized function `sum` of Figure 2.3, which computes the sum of the elements of an array between two of its indices. It is worth noting that `rec_sum` does not require an extra integer for the number `len` of elements of the array `a` (unlike its axiomatized version), but its behavior is undefined if `high` is negative or too large (since `a[high]` would be undefined according to the E-ACSL's three-valued semantics). Figure 3.7 also introduces a predicate `diff_sum_positive` that holds

if and only if the sum of the elements of an array between two of its indices is strictly greater at a given label L1 than at another given label L2. Note that there is also no way to ensure that recursive definitions are well defined, so that their calls terminate in finite time. This is a current design issue of ACSL (and so E-ACSL).

```
/*@ logic integer rec_sum(int *a, integer low, integer high) =
  @   high < low ? 0 : a[high] + rec_sum(a, low, high-1);
  @
  @ predicate diff_sum_positive{L1, L2}
  @   (int *a, integer low, integer high) =
  @   rec_sum{L1}(a, low, high) > rec_sum{L2}(a, low, high);
  @ */
```

Figure 3.7: Example of E-ACSL logic functions and predicates.

ACSL axiomatized logic functions and predicates, as well as inductive predicates, are not part of E-ACSL since, when defining the language in 2011, there was *a priori* no way to execute the axioms (or the inductive formulæ for inductive predicates) that define them. However, during his PhD, Pierre-Nicolas Tollitte was able to soundly translate some Coq inductive definitions into recursive functions (written in OCaml or Coq) under well-defined hypotheses [TDD12, Tol13]. In collaboration with Catherine Dubois from ENSIIE who supervised this PhD, I am currently considering adapting this work to E-ACSL in order to add an executable subset of axiomatized logic functions and predicates and inductive predicates to this language. For instance, the axiomatized logic function `sum` of Figure 2.3 (page 16) should be part of this subset. Preliminary work has already been done by Kharam Youness Kharraz during his master internship [Kha17] but it needs to be continued.

3.2.13 Ghost Code

Ghost code is similar to C code but is only visible in annotations. It is introduced by the `ghost` keyword when entering an E-ACSL comment. It must not interfere with the program: the program behavior must be the same with or without executing its ghost code [FGP16]. For instance, it must not modify the program memory. An example was already introduced in Figure 2.6 in which ghost variables are used to model pieces of information that are not directly computed by the program but are necessary to explain why the algorithm is correct. More generally, ghost code offers a very convenient way to compute extra information about the program that helps reasoning.

Ghost code is often a key feature when transforming non-functional properties to a set of functional properties in E-ACSL. In such a context, E-ACSL may be seen

as a low-level specification language in which high-level properties are encoded. For instance, Frama-C plug-ins Aoraï and Secure Flow use ghost code to respectively convert LTL properties expressed as Büchi automata and information flow properties as E-ACSL properties [18].

3.2.14 Comparison with Other Specification Languages

I voluntarily restrict this comparison to general-purpose specification languages for mainstream languages comparable to C. For instance, I do not include specification languages for separation logic [Rey02], dynamic logic [HTK00], or programming languages such as Praspel [EDGBO11] for PHP.

First, as already explained in this section, E-ACSL is directly derived from ACSL [BFM+] and both languages are consistent (see Conjecture 3.1), even if they have a few differences. The most important one is semantical: E-ACSL is based on a three-valued logic, while ACSL is based on a logic of underspecified total functions (see Section 3.2.1). Also, ACSL includes a few additional constructs that do not belong to E-ACSL, either because they are not executable in finite time (*e.g.* lemmas, model variables and fields, and `terminates` clauses), or they are still experimental in ACSL and I prefer to not include them in E-ACSL for the time being (*e.g.* specification modules, or well-typed pointers).

The specification language of VCC [CDH+09] is another specification language for C. VCC is a Microsoft's deductive verifier and the semantics of its specification language is similar to ACSL. These languages also share a lot of constructs, including assertions, function and block contracts, framing specifications, loop invariants and variants, object invariants, and ghost code. Actually these constructs are now rather standard and most languages cited in this section include them in one form or another: I will not discuss them for the other languages. It is worth noting that VCC targets verification of concurrent programs (such as Microsoft's Hyper-V hypervisor) [LS09a], and so its specification language also includes several functions and predicates related to concurrent accesses to program memory. That is not the case in E-ACSL (neither in ACSL). However, there is no construct dedicated to low-level memory properties such as `block_length`, `base_addr`, `initialized` or `valid_read`. More generally, this kind of construct is very specific to ACSL and E-ACSL.

ACSL and so E-ACSL are directly inspired from JML [LBR99] for Java, and both the authors of JML and E-ACSL have runtime assertion checking in mind. Even if it was not the case in its first versions [Che03], JML has eventually been based on Patrice Chalin's strong validity principle itself based on three-value seman-

18. Actually their current implementation uses standard C code instead of ghost code, because of a limited support of ghost code by the Frama-C kernel but, in an ideal world, ghost code would be used.

tics [Cha07] as is E-ACSL (see Section 3.2.1). For mathematical numbers, JML provides several modes [Cha04]. One of them consists in using mathematical integers (and reals) similarly to E-ACSL. The differences in their respective core language mainly come from the differences of their respective underlying programming language. I might however point out that JML allows the user to call pure methods (*i.e.* terminating Java methods without side effects), while it is not the case in E-ACSL (with functions instead of methods) since they cannot be used in ACSL. Indeed, ACSL chooses to strictly separate the logic from the program, because enforcing consistency of specifications including pure functions is not trivial [LM09], while it is critical to introduce no inconsistency in the underlying proof system. However, the recent versions of the E-ACSL tool allow the user to explicitly declare pure C functions usable in specifications. It has already been proved useful in a few E-ACSL case studies, while there is no inconsistency issue with runtime assertion checking of annotations containing pure methods (even if the verification that these pure methods terminate cannot be done at runtime, but is still required to ensure runtime check termination).

Spec# [BLS04] for C# is very similar to JML for the subset where E-ACSL, JML and Spec# are comparable. However, I have surprisingly found no reference about Spec#'s underlying logical and mathematical model. From a few experiments done from Spec#'s online version [19], it looks like it is based on a three-valued logic *à la* Chalin and bounded machine integers.

Code Contract [FBL10] is another Microsoft's formal specification language, but for .NET. It has been designed to support the Clousot abstract interpreter [FL10], even if a runtime verifier is also proposed to test the specifications before verifying them by abstract interpretation. Compared to E-ACSL and the other specification languages mentioned in this section, Code Contract is rather limited for, I think, three reasons. First, it must only contain predicates that can be verified by abstract interpretation. Second, it is independent of any programming language that can be compiled to .NET (for example, C# or VB.NET). Finally, it also aims at staying very close to the underlying programming language. In particular, predicates are just Boolean expressions and, for instance, (bounded) quantifications are just helper methods that make use of lambda expressions to express the bounded predicate.

Last but not least, Spark2014 also embeds a contract-based specification language [20]. As already explained, it was designed at the same time as E-ACSL, during the *Hi-Lite* project led by Adacore (from 2010 to 2013). It also relies on strong validity to solve the undefinedness problem. Bounded integer arithmetic semantics is the default but may be switched to unbounded. Pure methods are also supported. Also dedicated effort has been apported to developing a library of verified containers

19. `https://rise4fun.com/specsharp/`
20. `http://docs.adacore.com/spark2014-docs/html/lrm/`

easily usable in specifications [DFM11]. Such a library is currently missing in E-ACSL (and ACSL). For the interested reader, a deeper comparison between E-ACSL and Spark2014, as well as Why3, is provided in [KMSM16].

All these langagues are inspired by Eiffel [Mey88, Mey92b]. This language is still evolving [21]. Its specification language is based on preconditions, postconditions, `assigns`-like clauses, assertions, class invariants (corresponding to E-ACSL's strong type invariants) and loop annotations. Terms are Boolean expressions extended with `\result`- and `\old`-like terms. There is in particular no quantification. Eiffel annotations raise exceptions when evaluating undefined terms at runtime.

3.3 E-ACSL, a Tool for Generating Monitors

Context This section is based on several technical papers that I contributed to about the E-ACSL tool [DKS13, JKS15b, JKS16, VSK17, VKSJ17]. However, it aims at being less technical in order to provide a detailed global overview of the most interesting parts of the tool.

Writing a runtime assertion checker for a formal specification language such as E-ACSL may be considered as an easy task: just directly translate each term and predicate from the specification language to the corresponding expression of the underlying programming language and that's it. For instance, the E-ACSL assertion `/*@ assert x == 0; */` may be easily translated to the C assertion `assert(x == 0);`. Nothing complicated. The authors of Spec# seem to be in line with that idea since one of their papers [BFL+11] contains about one double-column page of explanations about enforcing Spec# contracts but, within it, only a short paragraph is dedicated to the Spec# runtime checker (all the others being dedicated to static verifications). Here it is *in extenso*:

> The run-time checker is straightforward: each contract indicates some particular program points at which it must hold. A run-time assertion is generated for each, and any failure causes an exception to be thrown.

But, if so straightforward, how can one explain that Yoonsik Cheon dedicated his whole PhD thesis to runtime assertion checking of JML [Che03], a language of similar complexity as Spec#? A first explanation might be that enough good research has been done in the meantime between 2003 and 2011. For instance, the undefinedness problem pointed out by Yoonsik Cheon is now well understood, particularly thanks to the works of Patrice Chalin [Cha04, Cha05, Cha07, Cha09] that I already mentioned several times. Another reason is perhaps a need for *efficiency*: it is rather easy to implement a quite naive runtime verifier, but it is harder

21. The lattest revision of the Eiffel standard was in 2006 and is freely available at `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-367.pdf`.

to efficiently generate efficient code. It could explain why the authors of Spec# pointed out that "the runtime overhead [of dynamically checking Spec# contracts] is prohibitive" [BFL+11].

This section aims at supporting this claim, while (informally) explaining how E-ACSL deals with this efficiency issue for the most interesting parts of the translation.

3.3.1 Overview

According to the taxonomy of Yliès Falcone *et al* [FHR13], and in common with all the runtime assertion checkers for Eiffel-like languages that I know, E-ACSL is an *(online) inline runtime verification tool*. It means that the monitor checks the annotations during the program execution (*online monitoring*) and is deeply embedded in the input program to do so (*inline monitoring*). Other possibilities could be to verify the annotations *after* the program execution on some traces (*offline monitoring*), or to run the monitor in parallel to the program under verification during its execution (online *outline monitoring*).

Fundamental Principles

More precisely, E-ACSL is a Frama-C plug-in that takes as input an E-ACSL-annotated C program and aims at generating another C program that has the following two fundamental properties.

Conjecture 3.2 (Preservation of Program Behaviors) *If all the annotations of the input program p are valid for a particular execution trace σ, then the generated program behaves in the same way than program p: its execution trace includes σ and, for each program point of σ, the value associated to each C variable from p remains the same.*

Conjecture 3.3 (Detection of Invalid Annotations) *If (at least) one annotation of the input program is invalid for a particular execution trace, then the generated program detects it and enters a special mode.*

The first property means that the generated monitor should have no (functionally observable) effects on the monitored program as long as the checked annotations are valid [22]. It is worth remarking that the generated trace is usually larger than the original trace since it notably includes additional checks of annotations. It also involves bigger memory states since the monitor may have its own memory locations

22. However, it might have non-functional undesirable effects. For instance, since the instrumented program consumes additional memory, it might lead to out of memory, while the original program would not.

to store intermediate results. Both explain why the generated program is usually slower and consumes more memory than the input one.

The second property means that the generated monitor is able to detect any invalid annotation. According to the blocking semantics of the annotation language (see Section 2.9.2, page 43), the monitor should abort when detecting the very first invalid annotation. This is actually the default "special mode" of E-ACSL. It is however customizable in practice (see Section 3.4.1).

In other words, both properties state that the generated monitor has neither false positives nor false negatives for a particular execution. One should also observe that both properties are actually only *conjectures*: they are currently —and unfortunately— unproved, and not even formalized. Dara Ly has just begun in November 2017 a PhD on that topic under my supervision in collaboration with Nikolai Kosmatov from my lab and Frédéric Loulergue from Northern Arizona University. This work should lead in three years to a formalization and a Coq proof of these properties for a subset of C and E-ACSL.

First Example

To make more concrete what E-ACSL is, let us illustrate what it does on the following tiny example:

```
int main(void) {
  /*@ assert 0 == 0; */
  ;
  /*@ assert 0 == 1; */
  return 0;
}
```

This program does nothing, but contains one valid assertion and one invalid one. Let say that this code is written in a file `foo.c`. It is possible to generate a monitored version of this code in file `monitored_foo.c` through the following command:

```
$ frama-c -e-acsl foo.c -then-last -print -ocode monitored_foo.c
```

This command first runs E-ACSL on the input file. By its own, E-ACSL only generates a new AST in a new Frama-C project [23]. The Frama-C kernel itself handles the second part of the command line. It switches to the project generated by E-ACSL through option `-then-last`, then pretty prints the generated AST as C code in file `monitored_foo.c`. Figure 3.8 presents a simplified version of the generated code.

In this code, after each assertion, a call to function `__e_acsl_assert` has been introduced to be checked at runtime. This function is provided by the E-ACSL C runtime library (RTL for short) and is similar to function `assert` from the C standard

23. The notion of Frama-C project is explained in Section 2.8, page 35.

```
void __e_acsl_assert(int pred, char *kind, char *fct,
                     char *pred_txt, int line);

int main(void) {
  int __retres;
  /*@ assert 0 == 0; */
  __e_acsl_assert(0 == 0,(char *)"Assertion",(char *)"main",
                  (char *)"0 == 0", 2);
  /*@ assert 0 == 1; */
  __e_acsl_assert(0 == 1,(char *)"Assertion",(char *)"main",
                  (char *)"0 == 1", 3);
  __retres = 0;
  return __retres;
}
```

Figure 3.8: Simplified version of the code generated by E-ACSL from a simple program. A few unused generated declarations have been removed for clarity.

library: it aborts the execution if the value of its first argument is 0. The function does nothing otherwise. The other arguments are only necessary to produce a nice localized error message when it occurs. Here, in each call, the first argument corresponds to the E-ACSL predicate of the translated assertion. The translation is straightforward in this trivial example. In order to be able to execute the generated file, it must be linked to the E-ACSL RTL. It is not very easy for the end-user to write the necessary command line in the right way. Consequently, to ease the whole process (invoking Frama-C, then compiling the generated code with a C compiler), E-ACSL provides a shell script named `e-acsl-gcc.sh`. It is thus possible to replace the above invokation of Frama-C by the following command:

```
$ e-acsl-gcc.sh -c foo.c
```

It creates three files:

- `a.out` is the binary without instrumentation compiled from `foo.c` by Gcc [24];
- `a.out.frama-c` is the instrumented C file called `monitored_foo.c` above;
- `a.out.e-acsl` is the instrumented binary generated by the compilation of `monitored_foo.c` by Gcc [25].

Running `a.out` just does nothing but exit with exit code 0 as expected, while running `a.out.e-acsl` aborts with the following error message, demonstrating that the monitor generated by E-ACSL is able to detect the violated assertion:

24. Gcc is the default compiler for `e-acsl-gcc.sh`. It is however customizable.
25. All these filenames are customizable.

```
Assertion failed at line 3 in function main.
The failing predicate is:
0 == 1.
Abandon
```

Architectural Design

As explained, E-ACSL only transforms a program into another one, so it is a program transformation tool. It may also be seen as a compiler from C+E-ACSL to C. Its architectural design, presented in Figure 3.9 reinforces this idea since it is rather close to the one of a standard compiler.

E-ACSL is a Frama-C plug-in and so works on the AST generated by the Frama-C kernel. Since the generated code relies on the E-ACSL RTL, and also the E-ACSL memory model (see Section 3.3.4), the headers of these libraries are automatically added to the Frama-C compilation process from the very beginning in order to easily generate self-contained code. The most important and the largest part of the plug-in is the instrumentation engine which concretely generates the monitor. It relies on several static analyses in order to ensure its correctness and to soundly improve its efficiency. For instance, the exit point computation deals with statements like `goto`, `break` and `continue` which are statements modifying the normal control flow of the program: these jumps may enter or leave a block that should initialize or free E-ACSL local variables and/or contain pre- or post-conditions to be verified. The other analyses will be explained later. All of them are done on demand, once. Furthermore, before generating the monitor, several intermediate transformations are also required.

First, a few modications are done to ease the monitor generation. For instance, the local variables declared in the bodies of `switch` statements are moved out. Also, this step computes necessary information to preserve the property status [26] of each annotation up to the end of the transformation. That is particularly important since, by default, E-ACSL does not generate any code for proved properties in order to improve the monitor efficiency (see Section 3.4.1 for a concrete example).

Second, during the *function duplication* step, a wrapper is generated for each function with an E-ACSL contract. The wrapper calls the original function and will contain the monitor in charge of verifying the function contract. Clearly separating the pieces of code verifying the contract from the original code is a useful feature which was originally demanded during the very first E-ACSL case study in 2012. It actually partially implements the "wrapper approach" described by Yoonsik Cheon [Che03, Section 4.1.3] in which the pre- and post-conditions are put in separate methods.

26. The notion of property status in Frama-C is explained in Section 2.9.2, page 43.

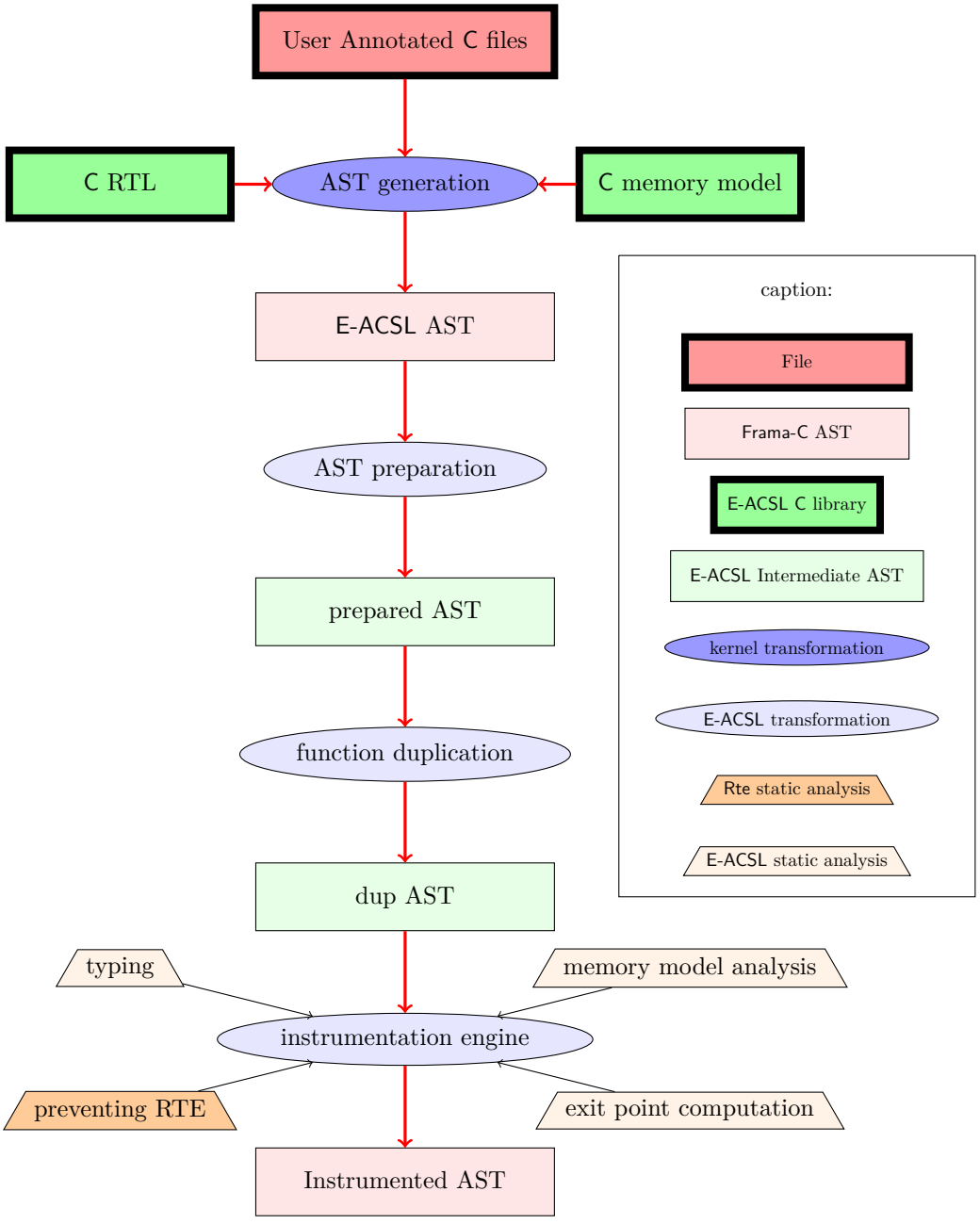


Figure 3.9: E-ACSL Architectural Overview.

I do not detail precisely here what each step does. It would be far too technical compared to the rest of this document. I prefer to highlight the most interesting parts of the transformation in the next subsections: handling undefinedness (Section 3.3.2), mathematical numbers (Section 3.3.3), and memory-related properties (Section 3.3.4).

3.3.2 Preventing Additional Undefined Behaviors

As explained in Section 3.2.1, the semantics of the E-ACSL specification language is three-valued: the semantics of some terms and predicates may be undefined. However, it is the responsibility of the tools implementing this language to take care of not evaluating such undefined entities.

To do so, the E-ACSL tool generates a runtime error each time an undefined term or predicate would be evaluated. It is one of the possible approaches described by John Hatcliff *et al* [HLL+12, Section 2.1.3], while Patrice Chalin argues that it is the one that practionners want [Cha05].

For implementing such a feature, E-ACSL relies on the Rte plug-in of Frama-C, shortly introduced in Section 2.6 page 27. This plug-in generates ACSL assertions —that are fully compatible with E-ACSL— for a large variety of possible undefined behaviors: for each C expression e, if the generated assertions are valid, then e does not contain any undefined behavior of the considered kinds. This property is ensured for C statements as well. For instance, this plug-in generates the following assertions for the expression `(int)(*(*p) + 1) / r <= 2147483647; */` (assuming a 64-bit architecture [27], a global variable `p` of type `int **`, and another global variable `r` of type `int`):

1. `/*@ assert rte: mem_access: \valid_read(p); */`
2. `/*@ assert rte: initialization: \initialized(p); */`
3. `/*@ assert rte: mem_access: \valid_read(*p); */`
4. `/*@ assert rte: initialization: \initialized(*p); */`
5. `/*@ assert rte: signed_overflow: -2147483648 <= *(*p) + 1; */`
6. `/*@ assert rte: signed_overflow: *(*p) + 1 <= 2147483647; */`
7. `/*@ assert rte: division_by_zero: r != 0; */`
8. `/*@ assert rte: signed_overflow:`
   `(int)(*(*p) + 1) / r <= 2147483647; */`

These assertions notably cover pointer validity (assertions 1 and 3), memory location initialization (assertions 2 and 4) [28], arithmetic overflow (assertions 5, 6, and 8) and

27. Each code fragment assumes a 64-bit architecture from now on, unless otherwise specified.
28. The initialization assertions are not generated by default, but it is optionally possible to get them through the Rte option `-rte-initialized`, available since Frama-C Sulfur-20171101.

division by zero (assertion 7). The Rte plug-in also may perform quick syntactic checks on demand in order to not generate assertions that are trivially valid. On the above example, annotation 5 always holds and is not generated on demand.

In addition to providing such services to end-users, the Rte plug-in also offers an API to be used by other Frama-C plug-ins. Continuing our above example, it is possible to get the list of predicates corresponding to the above assertions (ordered in the very same way). Consequently, when generating the new C expression corresponding to a predicate to be verified at runtime, E-ACSL also generates the E-ACSL predicates corresponding to its possible undefined behaviors through the Rte plug-in and translates them in turn to C code. That corresponds to trapezium "preventing RTE" in Figure 3.9. There is even no need to compute a fixpoint, since Rte presents the predicates in the right order such that the n first generated predicates guarantee that, if every of them holds, the $(n+1)$-th generated predicate contains no undefined terms or predicates (with respect to the E-ACSL semantics). For instance, on the above example, the well-definedness of term `*p` in assertions 3, 4, 5, 6, and 8 is guaranteed by the first two assertions which state that `p` is valid (assertion 1) and its contents `*p` is properly initialized (assertion 2). Such a usage of Rte by E-ACSL is an example of sequential collaboration in Frama-C, presented in Section 2.9.1.

As a final illustration of handling undefinedness by E-ACSL, here is the block of C code generated by E-ACSL for the assertion `assert x / y >= 0;` in which `x` and `y` are two global variables of type `int`:

```
/*@ assert x / y >= 0; */
{
  __e_acsl_assert(y != 0L,(char *)"RTE",(char *)"main",
                  (char *)"division_by_zero: (long)y != 0",20);
  __e_acsl_assert(x / y >= 0U,(char *)"Assertion",(char *)"main",
                  (char *)"x / y >= 0",20);
}
```

The first function call prevents any division by zero when runtime checking the assertion in the second call. It comes from the predicate `y != 0L` computed by Rte from the C expression `x / y >= 0L` generated by E-ACSL from the original assertion [29]. This Rte intermediate predicate is not exposed to the end-user but the error message generated at runtime when the value of `y` is 0 should be crystal clear:

```
RTE failed at line 20 in function main.
The failing predicate is:
division_by_zero: y != 0L.
```

29. The constant `0L` instead of `0` is infered by E-ACSL in order to evaluate soundly integer operations as explained in Section 3.3.3.

```
Abort
```

In this example, the watchful reader might have noticed that an unsigned zero `0U` was generated from the original assertion in which it was a signed zero `0`. It comes from the way that E-ACSL translates mathematical numbers and operations such as `0` in the above assertion.

3.3.3 Executing Mathematical Numbers and Operations

The E-ACSL specification language relies on mathematical numbers and operations ($\mathbb{Z}$ for integers, and $\mathbb{R}$ for reals) as explained in Sections 3.2.4 and 3.2.5. However, translating mathematical integers to C integral types would be incorrect, as well as translating reals to floating point numbers. This section presents the E-ACSL solution to the integer problem. Reals are not yet handled by the tool and is left to future work.

Illustrative Example

Let us first illustrate the issue of translating mathematical integers *soundly* and *efficiently* on the two following simple assertions.

```
/*@ assert x + 1 <= 127; */
/*@ assert c + 1 == 0; */
```

Let us also assume that they are evaluated on a rather limited 8-bit architecture for which the values of `x` of type `int` belong to the interval $[-128; 127]$ and values of `c` of type `char` belong to $[-32; 31]$ [30]. Also assume that there is no other types that `int` and `char`.

Consequently, the first assertion is invalid if and only if the value of `x` is 127. However, a direct translation of its predicate to the C expression `x + 1 <= 127` (so of type `int`) would be unsound since an overflow would occur whenever `x` would be 127. To solve this issue, the E-ACSL translation relies on the GNU Multiple Precision Arithmetic Library, also known as Gmp [31], which offers arbitrary precision integer arithmetic. Figure 3.10 shows how E-ACSL translates the first assertion above using Gmp. In this figure, the C comments have been manually added to make the generated code almost self-explaining, while a few (unmandatory) casts as well as the extra arguments to `__e_acsl_assert` have been removed for clarity. The translation consists in declaring, allocating, and, if necessary, initializing one Gmp variable

30. The only goal of this assumption is simplicity, but one gets the same results with a more realistic 64-bit architecture and standard intervals for `int` and `char`, as soon as one replaces the constant 127 in the first assertion by 2147483647 (*i.e.* `INT_MAX`).

31. `https://gmplib.org/`

for each mathematical integer (here, the values of `x`, 1, the result of the addition, and `127`), computing the mathematical operations through specific Gmp functions (here `__gmpz_add` and `__gmpz_cmp`) before checking the predicate. At the end, the allocated memory should be cleared through calls to function `__gmpz_clear` (Gmp values are pointers).

```
/*@ assert x + 1 <= 127; */
{
  /* declaration of necessary temporary Gmp variables */
  __e_acsl_mpz_t __gen_e_acsl_x;
  __e_acsl_mpz_t __gen_e_acsl;
  __e_acsl_mpz_t __gen_e_acsl_add;
  __e_acsl_mpz_t __gen_e_acsl_2;
  int __gen__e_acsl_le;
  /* computing x+1 */
  __gmpz_init_set_si(__gen_e_acsl_x, x);
  __gmpz_init_set_si(__gen_e_acsl, 1);
  __gmpz_init(__gen_e_acsl_add);
  __gmpz_add(__gen_e_acsl_add, __gen_e_acsl_x, __gen_e_acsl);
  /* comparing x+1 to 127 */
  __gmpz_init_set_si(__gen_e_acsl_2, 127);
  __gen_e_acsl_le = __gmpz_cmp(__gen_e_acsl_add, __gen_e_acsl_2);
  __e_acsl_assert(__gen_e_acsl_le <= 0);
  /* freeing the previously allocated memory */
  __gmpz_clear(__gen_e_acsl_x);
  __gmpz_clear(__gen_e_acsl);
  __gmpz_clear(__gen_e_acsl_add);
  __gmpz_clear(__gen_e_acsl_2);
}
```

Figure 3.10: Example of Gmp-based translation by E-ACSL.

It is worth noting that a similar translation scheme could be adopted for computing correct approximations of operations over reals (for instance, by using the MPFR library [32]). It is even possible to be exact for a few arithmetic operators over rationals. Such a support is also provided in Gmp.

Let us come back to integers. While sound, this translation scheme is nevertheless much more complex and expensive (in terms of time and memory consumption) than directly (but unsoundly) using C integral types such as `int`. Executing a sim-

32. `https://www.mpfr.org/`

ilar block of code for each E-ACSL mathematical operation is actually prohibitive. However, most of the time, it is fortunately possible to be sound without using Gmp. Let us consider for instance the second assertion `/*@ assert c + 1 == 0; */`. Here, integer constants `0` and `1` can safely be represented with machine integers, as well as `c + 1` whose values vary between -31 and 32, so can be represented by an `int` (but not by a `char` in our *ad-hoc* setting). Consequently, it would be perfectly sound to directly translate the predicate to the C expression `c + 1 == 0` of type `int`. However, how to distinguish the first example that requires the sound yet inefficient Gmp encoding from the second one that can soundly rely on efficient C integral types?

Type System

E-ACSL answers this question using a type system [JKS15b] that corresponds to the trapezium "typing" in Figure 3.9. Before translating terms and predicates, each of them is fully typed on the fly. If the type system is able to infer $\Gamma \vdash t : \tau_1 \rightsquigarrow \tau_2$, it means that, in an environment Γ, the C expression resulting from the translation of term t may soundly be encoded in type τ_1 while the head operator of the term (if any) may be translated to an operator over type τ_2. There is a similar type judgment for predicates.

It is a bit unusual to compute two types instead of only one, but the second one helps to recover precision when typing antimonotonic operators such as integer divisions. Also, this type system relies on interval arithmetic [MY59, MKC09] to infer an interval that a given term belongs to. In this context, types are just intervals defined by their bounds. For instance, type `int` may be seen as the interval $[-128;\ 127]$ on our 8-bit architecture used in our examples. Thanks to these intervals, the type system comes with two subtyping rules: one of them is the very standard subsumption rule [Pie02] which allows the type system to lose precision by converting a type to a bigger one, while the other one (called the coercion rule) allows recovering precision by coercing a type to a smaller type τ when the interval infered for the given term fits into type τ.

Let us illustrate all of this by typing the term `1 + (x + 1) / (y - 64)` in which `x` and `y` are respectively of types `int` and `char` in our 8-bit architecture. Figure 3.11 [33] presents its derivation tree in which rules right-tagged with $\preccurlyeq$ are instances of the subsumption rule, and the rule with $[-3;3] \preccurlyeq$ `int` as hypothesis is an instance of the coercion rule. Also, computing a single type τ means $\tau \rightsquigarrow \tau$. This derivation safely allows us to assign a type for each C expression and operation

33. In this figure, it might look erroneous that variable `y` gets type `int`, while its C type is `char`. It is not a bug but a feature^{TM}of the type system because it is actually not necessary to compute a type more precise than `int` thanks to the C99 promotion rule (see C99 standard, §6.3.1.1, p.43). Therefore, whenever all the possible values of any variable v fit in an `int`, v gets type `int`. Details about the typing rules are omitted here but can be found in the original paper (in French) [JKS15b].

generated from this term. For instance, the generated addition from term `x + 1` must be a Gmp one since its type is $\mathbb{Z}$ (third line of the derivation). Indeed this operation might overflow if computed with a 8-bit C type. However, the generated substraction from `y - 64` may be safely computed in type `int`. Indeed, its result may only range from $-32 - 64 = -96$ to $31 - 64 = -33$. However, its results must be converted to Gmp in order to compute the division through a Gmp operation, since the dividend `x + 1` must be translated to a Gmp value anyway. Nevertheless, the result of this division ranges between $128/-33 = -3$ and $-127/-33 = 3$ which fits into `int`. Consequently it can safely be coerced to this type in order to compute the first increment also in `int`.

$$\dfrac{\Gamma \vdash 1 : \texttt{int} \qquad \dfrac{\dfrac{\dfrac{\Gamma \vdash x : \texttt{int}}{\Gamma \vdash x : \mathbb{Z}} \preccurlyeq \dfrac{\Gamma \vdash 1 : \texttt{int}}{\Gamma \vdash 1 : \mathbb{Z}} \preccurlyeq}{\Gamma \vdash x + 1 : \mathbb{Z}} \qquad \dfrac{\dfrac{\Gamma \vdash y : \texttt{int} \quad \Gamma \vdash 64 : \texttt{int}}{\Gamma \vdash y - 64 : \texttt{int}}}{\Gamma \vdash y - 64 : \mathbb{Z}} \preccurlyeq}{\Gamma \vdash (x+1)/(y-64) : \mathbb{Z} \rightsquigarrow \mathbb{Z}} \qquad [-3;3] \preccurlyeq \texttt{int}}{\Gamma \vdash (x+1)/(y-64) : \texttt{int} \rightsquigarrow \mathbb{Z}}}{\Gamma \vdash 1 + (x+1)/(y-64) : \texttt{int}}$$

Figure 3.11: Example of derivation tree from the E-ACSL type system.

On this small example, it is clear that the type system helps us to safely replace several Gmp operations and values by C integral counterparts. This type system is actually very effective since it removes almost all Gmp in practice, except for monotonic operations over large types such as `long long`. Last but not least, it is worth noting that Adacore has adapted to Spark2014 the idea of such a type system in order to allow its users to specify mathematical properties while soundly minimizing the runtime overhead. From my point of view, it is an excellent illustration of what I try to do in E-ACSL: adapt standard static analysis and compilation techniques (here, a type system) in order to soundly optimize the generated code (see the original paper [JKS15b] for the correctness theorem).

3.3.4 Monitoring Memory Properties

Between 2012 and 2017, the most active E-ACSL research area was related to supporting the memory built-in logic functions and predicates of the E-ACSL specification language exposed in Figure 3.4. As a numerical evidence of this research effort, two patents [VKS16a, VKS16b] and five research papers [KPS13a, JKS15a, JKS16, VSK17, VKSJ17] were entirely dedicated to this topic in that period of time. It is joint work with Nikolai Kosmatov at LSL, with major contributions from Guillaume Petiot, Arvid Jakobsson, and Dara Ly during their master in-

ternships in 2012 [Pet12], 2014 [Jak14], and 2017 [Ly17] respectively, and from Kostyantyn Vorobyov during his 2-year postdoc from 2015 to 2017.

Short State of the Art

First, it is worth noting that the memory-related properties described in Figure 3.4 are very specific to the E-ACSL specification language: no other Eiffel-like specification languages support similar constructs as explained in Section 3.2.14.

In order to verify them at runtime, E-ACSL must have a deep knowledge of the structure of the program memory. For instance, it may need to know if some memory address pointed to by some pointer `p` is writable by the program (otherwise, the predicate `\valid(p)` would not hold), or if it has been initialized at some moment during the execution in order to check `\initialized(p)`.

Such dynamic verifications are actually performed by a category of tools named *dynamic memory analyzers* (or *memory debuggers*, or *memory error detectors*). These tools are specialized in detecting memory errors such as buffer and heap overflows, and accesses to uninitialized data in the program memory before they really occur. In order to perform the required verifications, they instrument the program code either at source code level (like E-ACSL), or at binary level.

Most of them rely on an efficient technique called *memory shadowing* to keep track of pieces of information about the program memory in order to detect memory issues. Memory shadowing usually associates addresses from program memory to values stored in a disjoint memory space called *shadow memory*. Accessing the shadow memory from the program memory is performed in constant time, just by adding some fixed offset from the program memory. The way of structuring these shadow values is refered to as *shadow state encoding* and may vary from one tool to another. However, they almost always store bit-level information about individual bits or bytes of the monitored program memory. For instance, Dr. Memory [BZ11] and Purify [HJ92] shadow every byte from the program memory by two bits representing its allocation and initialization statuses, while Valgrind [NS07]'s MemCheck [SN05] and MemorySanitizer [SS15] use bit-to-bit shadowing to track the initialization status of every single bit. For its part, Google's AddressSanitizer [SBPV12] (probably the most famous memory debugger currently since it is embedded in Clang and Gcc) ensures that memory blocks are allocated at 8-byte boundaries by customizing memory allocation, and tracks aligned 8-byte sequences by one shadow byte in order to store which bytes of such sequences are addressable.

However, even if efficient enough to be of practical interest, tools based on memory shadowing that focus on byte-level (or bit-level) information fail to detect *block-level properties*[34]. Such properties are related to the bounds of memory blocks.

34. This name was introduced during Arvid Jakobsson's internship [Jak14, JKS15a].

Consider for instance the following code snippet that contains a buffer overflow at its second line:

```
char a[1], b[4];
a[1] = '0';      /* buffer overflow */
```

Variables `a` and `b` are local variables, usually allocated on the stack one after the other. Consequently, at line 2, trying to modify the memory block m_a corresponding to variable `a` may actually modify the memory block m_b corresponding to variable `b`. Traditional memory shadowing techniques as described above do not detect any error, since memory block m_b has been properly allocated. To detect this error, it is necessary to detect that accessing `a[1]` crosses the boundaries of memory block m_a. A similar detection is also required by E-ACSL, for example for detecting that property `\valid(a+1)` does not hold right before line 2. More generally, some E-ACSL predicates and logic functions such as `\offset` and `\block_length` are block-level memory properties that cannot be computed by traditional memory shadowing techniques for the very same reason.

To overcome this issue, it is possible to cleverly use the space left between aligned allocated blocks, as pioneered by Richard W. M. Jones and Paul H. J. Kelly [JK97]. For instance, AddressSanitizer [SBPV12] uses this alignment property through a technique named *red zoning* that creates poisoned red zones around allocated memory blocks: if a buffer overflow is small enough to go to the red zone, it is detected. Typically, it is able to detect off-by-one errors such as the one of the above example, but it is not able to detect the error if line 2 is replaced by the assignment `a[3] = '0';` since the 3-offset goes outside the redzone of `a` but still in the allocated area m_b. This example also illustrates that this technique, even if highly practical, is unable to detect all possible overflows.

Another issue with memory shadowing comes from the fact that they usually miss *temporal memory errors* [SB10, VKSJ17], which are errors when trying to access a pointed-to object that is not the same as when the pointer was created. Consider for instance the following code snippet where `p` and `q` are two pointers to type `int`:

```
p = malloc(sizeof(int));
*p = 1;
free(p);
q = malloc(sizeof(int)); /* may reuse the freed memory block */
*p = 2;                  /* illegal access */
```

Once pointer `p` is freed, it cannot be dereferenced anymore. However, allocating a new memory block may reuse the very same freed memory block (depending on the system's allocation strategy): traditional shadow memory techniques would then

miss an illegal access to `*p` since the pointed-to memory block is properly allocated (yet unallowed to be accessed through `p`).

To fully deal with the limitations of shadow memory techniques, one could rely on *fat pointers* [ABS94], as done for instance by Yutaka Oiwa [Oiw04, Oiw09] in order to extend the pointer representation with bounds information. While offering a way to check all the desired properties about pointers, the main drawback of this technique is the modification of the size of pointed data: preserving the original program behavior with such a modification of the program memory structure is tricky, to say the least. It is also not possible to verify properties about non-pointer values, for instance checking that an integer local variable has been properly initialized before being used.

Yet another approach solves the fat pointer issue by associating to each allocated memory address the necessary pieces of information and storing them in separated dictionaries, usually implemented through splay trees [ST85]. This approach was pionnered by Richard W. M. Jones and Paul H. J. Kelly at the same time as using alignment of allocated blocks [JK97]. This approach is however slower than the other ones, despite many efforts to improve it, such as those of Dinakar Dhurjati [DA06].

In the following, whatever the way used to store at runtime the necessary pieces of information about program memory, I will call it a *runtime memory model* by analogy to the notion of memory model used in compilation [LB08] or static program verification as "the way that the analysis tool models the storage of the underlying machine on which the code runs" [XKZ10].

Patricia-Trie Memory Model

The very first E-ACSL memory model [Pet12, KPS13a] was created in 2012 by Guillaume Petiot, during his master internship mainly supervised by Nikolai Kosmatov with my help. Next he has continued to improve it during his PhD up to 2015 [Pet15]. My main contributions were some advice and the proper integration of the memory model within E-ACSL.

In 2012, the focus was put more in expressiveness of the memory model than its efficiency, in order to support the desired constructs of the E-ACSL specification language. Consequently, the choice was made to rely on a dictionary-based memory model. The most usual dictionary datastructure is hashtable, but it does not fit here because one needs to access to the closest base address of some address a smaller than a (and to check its bounds afterwards): such an operation would be of linear complexity with hashtables. We choose to implement it through a *Patricia trie* [Mor68, Szp90], also known as *radix tree* or *compact prefix tree*. As far as I know, this is the only memory model implemented with such a data structure. Its efficiency seems comparable to splay trees (see the last part of this section).

Each leaf of the trie contains a *block metadatum*. Each metadatum includes a

base address a of thirty two or sixty four bits which is the key of the dictionary, the size (in bytes) of the memory block b starting from a, its validity status (whether reading or writing block b is safe) and the initialization status for each byte of block b.

Each internal node of the trie contains the greatest common prefix of all base addresses stored in its subtrees. This way, it is easy and fast, through a few bit-level operations, to go from the root of the trie to a base address and its associated information stored in a leaf.

The block metadata contain enough information to compute the E-ACSL memory-related constructs, including all the block-level properties. More precisely, Figure 3.12, based on Figure 3.4, shows which E-ACSL memory-related constructs are actually supported by the tool and which ones are not yet implemented. It is worth noting that the support of `\null` is independent from the memory model, while the support of the missing constructs is doable without any extension of the memory model: it only requires additional code in the instrumentation engine. Consequently, the expressive power of the memory model is strong enough.

predicates	logic functions and constant
`\valid`	`\null`
`\valid_read`	`\block_length`
`\initialized`	`\base_addr`
~~`\separated`~~	`\offset`
`\freeable` [35]	~~`\allocation`~~
~~`\allocable`~~	
~~`\dangling`~~	
~~`\fresh`~~	

Figure 3.12: Memory built-in logic functions and predicates implemented in E-ACSL.

This memory model has been implemented in C and corresponds to the box "C memory model" of Figure 3.9. Its API provides the necessary functions to evaluate the E-ACSL memory-related logic functions and predicates and to register and modify metadata in the trie. These functions are called by the monitor generated by the E-ACSL instrumentation engine. For instance, Figure 3.13 shows a slightly simplified version of the code generated by E-ACSL for the following function `main`:

```
int main(void) {
  int p[10];
  /*@ assert \block_length(&p[0]) == 10 * sizeof(int); */
```

35. The predicate `\freeable` was not implemented in the very first versions of E-ACSL. Its support was added in 2015.

```
  p[2] = 0;
  /*@ assert \initialized(p+2); */
  return 0;
}


int main(void) {
  int p[10];
  __e_acsl_store_block((void *)(p),40);
  /*@ assert \block_length(&p[0]) == 10 * sizeof(int); */
  {
    unsigned long __gen_e_acsl_block_length;
    __gen_e_acsl_block_length = __e_acsl_block_length((void *)(p));
    __e_acsl_assert(__gen_e_acsl_block_length == 10 * 4);
  }
  __e_acsl_initialize((void *)(& p[2]),sizeof(int));
  p[2] = 0;
  /*@ assert \initialized(p+2); */
  {
    int __gen_e_acsl_initialized;
    __gen_e_acsl_initialized =
      __e_acsl_initialized((void *)(& p[2]),sizeof(int));
    __e_acsl_assert(__gen_e_acsl_initialized);
  }
  __e_acsl_delete_block((void *)(p));
  __e_acsl_memory_clean();
  return 0;
}
```

Figure 3.13: Example of E-ACSL instrumentation based on its runtime memory model.

This function first checks the length of array `p`, before initializing its third cell (at index 2) to 0 and verifying that this initialization has been correctly performed. The generated code first stores, through a call to function `__e_acsl_store_block` provided by the runtime memory model, that a memory block started at address `p` and containing 40 bytes has been allocated (assuming that the size of `int` is 4, 40 is the size of an array of `int` of length 10). The block of code following the first assertion checks the length of `p` by calling function `__e_acsl_block_length` exported by the memory model. Next, the generated code stores that `sizeof(int)` bytes have been initialized from address `&p[2]` (which is equivalent to `p+2`), through a

call to `__e_acsl_initialize`. Eventually, it allows the generated code to successfully check that the last assertion holds through a call to `__e_acsl_initialized`. Before leaving function `main`, the generated code removes its local variable `p` from the memory model through a call to `__e_acsl_delete_block`. Finally, before leaving the program (since this code monitored function `main`), the memory model frees what it may have internally allocated for its own purpose through a call to function `__e_acsl_memory_clean`.

It is worth noting that the calls to the memory model functions usually cast the address to type `void *`. Indeed, the E-ACSL runtime memory model is a byte-level untyped model. When required, the size in bytes of the considered memory chunk is provided as an extra argument.

Hybrid Memory Model

The Patricia trie model is expressive enough, but it is too inefficient and does not scale well on large-size programs. A first step towards solving the efficiency issue has been made in 2014 by Arvid Jakobsson during his master internship [Jak14], supervised by Nikolai Kosmatov and myself. Arvid designed a *hybrid memory model* that mixes the Patricia trie model with a shadow memory model [JKS15a, JKS16]. As far as I know, it is the only attempt to combine a dictionary-based model and a shadow model: only hybrid approaches combining dictionary-based models and fat pointers have been studied [SB10, YJ12].

The underlying idea of Arvid Jakobsson's hybrid model is to use an efficient shadow store whenever possible and to rely on the less efficient Patricia trie store whenever the shadow model is not expressive enough, in particular when verifying block-level properties. The hybrid model comes with seven principles explaining in which store the monitor should register, delete or look for metadata. For instance, principle P1 indicates that metadata associated with memory locations that require block-level properties must be stored in the trie, while (conversely) principle P5 explains that evaluation of block-level predicates should always query the trie. Applying principle P1 requires to statically know, at the time of the instrumentation, whether a block-level property of a particular memory location may eventually be computed. This issue will be addressed a bit later in this section.

Last but not least, the hybrid model is fully compatible with the Patricia trie model since both share the same API: when changing the memory model, it is not necessary to change the E-ACSL instrumentation engine that generates the monitor.

Shadow Memory Model

The hybrid memory model was a step towards improving efficiency but it does not solve this issue for block-level properties. Also, it was quite heavy since it embeds two different stores, increasing in particular the risk of implementation

bugs: if only one of the store implementations is buggy, then so is the hybrid model implementation.

In 2016, Kostyantyn VOROBYOV, during his postdoc supervised by Nikolai KOSMATOV and myself, found a way to express the desired block-level properties in a shadow memory model [VSK17]. I should better write that he found two different ways since the encodings of metadata in the heap [VKS16a] and in the stack [VKS16b] actually differ, the former being simpler than the latter.

The key idea of this block-level shadow memory model consists in splitting memory blocks and their shadow representations in memory segments of a fixed size of s bytes, the first segment being distinguished from the other and named the *meta-segment*. I voluntarily simplify the presentation here to keep things simple, but the technical details may be found in the above-mentioned papers. For instance, I assume here that program segments and shadow segments have the same size s, but this is actually not mandatory since different segment sizes only impact memory overhead.

The heap shadow encoding assumes that memory blocks are aligned at a boundary divisible by s in order to guarantee that memory segments do not overlap and their base addresses may be found through a single modulo operation: for some address a, the base address of its segment is $a - a \bmod s$. The encoding also assumes that every memory block is padded with (at least) s bytes in order to reserve enough space for the meta-segment. Last, it also assumes that the minimal size of s is twice the number of bytes necessary to represent a physical address (*i.e* $s \geq 64/8 \times 2 = 16$ for 64-bit addressing). It is worth noting that all these assumptions have no practical issue. For instance, it is easy to wrap the C allocation functions to fit them.

Let us now explain the heap encoding through an example shown in Figure 3.14. It consists in shadowing a memory block B of 40 bytes starting at physical address 0x100 [36]. We assume here a size of segments of 16: address 0x100 is thus properly aligned. We need three 16-byte segments to fully cover the 40-byte block B. Their base addresses are 0x100, 0x110 and 0x120. The 8 trailing bytes of the last segment are unallocated, as well as the meta-segment right before the first segment.

The shadow model of this block follows the same segment partitioning, while we assume that it starts at 0x200 (*i.e.* the offset of the shadow region from the program memory is 0x100). The first 8 bytes of the shadow meta-segment are unused and nullify, while the last 8 bytes encode the block length of the memory block B, here 40. For the other segments, the first 8 bytes encode the offset from the base address of the segment to the base address of the block incremented by one. This non-zero value guaranteed by this final increment indicates that at least the first byte of the corresponding segment of B has been allocated. Then, the 16 following *bits* (*i.e.* 2

36. Hexadecimal notation is used for denoting physical addresses and large offsets.

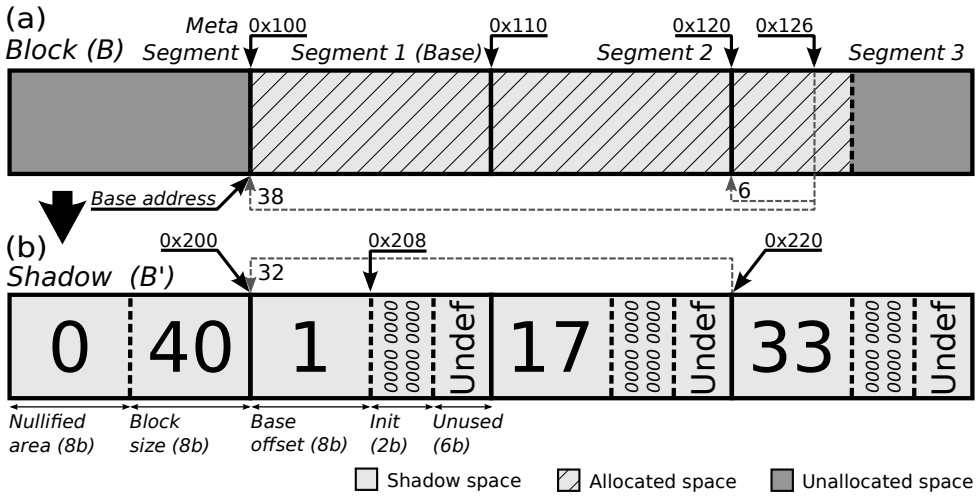


Figure 3.14: Example of E-ACSL heap shadow representation.

bytes) encode the per-byte initialization status of the corresponding segment of B: the i^{th} bit is set to 1 as soon as the i^{th} byte of the corresponding segment of B is initialized. The following bits (here 48 bits, *i.e.* 6 bytes) are left unused and may be used for other purposes (see Section 3.5.1).

Through this encoding, it is easy to retrieve the useful information for a particular byte. For instance, consider the address 0x126, its base segment starts at address $\texttt{0x126} - \texttt{0x126} \bmod 16 = \texttt{0x120}$. Consequently, its initialization bit is the sixth bit from 0x228 (so 0 here). The base address of the memory block that it belongs to (*i.e.* B) is $\texttt{0x120} - 33 + 1 = \texttt{0x100}$, 33 being the number stored in the eight first bytes from 0x220. Also, the size of this block is 40, the number stored on 8 bytes at address $\texttt{0x200} - 16 + 8 = \texttt{0x198}$.

The encoding is a bit more complicated for the program stack. Indeed, blocks on the stack are usually unaligned and often small: introducing a sufficient alignment to apply the above encoding is likely to introduce significant memory overhead. Instead, the stack encoding uses two shadow stores called the *primary* and the *secondary shadows*. The key idea here is to store metadata of small blocks (of size 8 bytes or smaller) in the primary shadow, and large metadata (that is, the block length and the byte offset) of large blocks (of size greater than 8 bytes) in the secondary shadow. In the latter case, the primary shadow is also used to retrieve the location in the secondary shadow where the block length and the byte offset of the monitored address are stored. The primary shadow uses a byte-to-byte shadow encoding, while the secondary shadow relies on a segment-based encoding.

Let us consider the encoding of one small 4-byte block, starting at address 0x100,

shown in Figure 3.15. One byte of the stack is directly shadowed by one byte of the primary shadow. A 0-value in the primary shadow indicates an unallocated byte, while a value between 1 and 36 stored in the first 6 bits of the shadow byte encodes both the size of the memory block that the byte belongs to and the offset of this byte from the base address of its block. For instance, a value of 8 indicates a memory block of 4 bytes and an offset of 1. These 36 possible values allow us to encode any possible sizes and offsets between a 1-byte block (and necessarily a 0-offset byte) and a 8-byte block and a 7-byte offset in this block. The seventh bit of the shadow byte encodes the initialization status of the monitored byte, while the last bit is left unused.

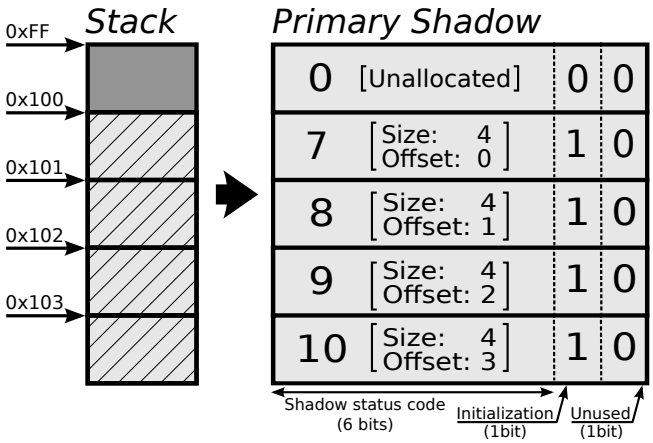


Figure 3.15: Example of E-ACSL stack shadow representation for a small block.

Now consider a larger block B of 18 bytes still starting at address 0x100. The encoding is shown in Figure 3.16. We assume here that offsets of 0x100 and 0x200 allow accessing to the primary shadow store and to the secondary shadow store, respectively. Thus, the primary shadow block tracking block B starts at address 0x200, while its secondary shadow block starts at address 0x300. Let us now remember that, for small blocks, the first six bits of the primary shadow byte are used to encode values between 1 and 36 representing the block length and the byte offset, while a 0-value indicates an unallocated byte. However, sixty-four values may be encoded within six bits. Our encoding does not need this much, but still uses the values between 48 and 63 for large blocks: these values, decremented by 48, indicate the offsets from the corresponding shadow segment base addresses in the secondary shadow. For instance, the first six bits of the primary shadow byte of address 0x10F contains the value 55, so an offset of $55 - 48 = 7$ in the secondary shadow: the segment storing the block length and the offset of address 0x10F starts

at address 0x30F − 7 = 0x308. In this segment, the first four bytes encode the length of the memory block that the original address belongs to (here, 18) and the last 4 bytes encode the offset from the start of the secondary shadow block to the start of this segment (here, 8). From it, it is possible to recover the base address of the memory block that an address such as 0x10F belongs to (here, 0x108 − 8 = 0x100), and its offset in that block (here, 0x0F − 0x100 = 15). Still, the seventh bit of the primary shadow bytes represents the initialization status of the corresponding bytes of a large block.

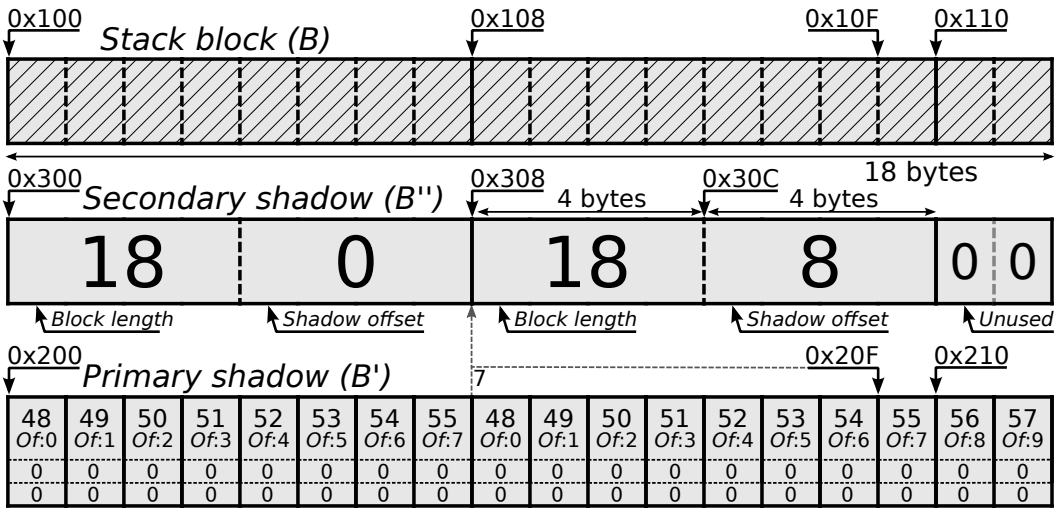


Figure 3.16: Example of E-ACSL stack shadow representation for a large block.

This block-level shadow memory model, with its two versions for the heap and the stack, has the very same expressive power as Guillaume Petiot's Patricia trie model, while being of comparable efficiency as (but more expressive than) traditional shadow models. It is also more efficient and lighter than Arvid Jakobsson's hybrid model. It is now the default E-ACSL runtime memory model since Frama-C Phosphorus-20170501. Its API is fully compatible with the Patricia trie and the hybrid models; thus it is easy to swap from one model to another without changing the E-ACSL instrumentation engine.

As far as I know, the only other work that could express block-level properties with a shadow-like technique is METAlloc [HvdKGB16] but it relies on modern heap [Ghe07] and stack [KSP+14] organizations and a non-trivial alignement of objects sharing a memory page whose implementation is tricky.

Even if the above-presented block-level shadow memory model is more expressive than the traditional shadow models, it is only able to detect spatial memory errors, not temporal ones. In 2017, it has been extended to also detect this additional

kind of error, using an idea that emerged at the end of Arvid Jakobsson's internship [VKSJ17]. While effective in detecting temporal errors, the time overhead is still rather high (see the last part of this section). Consequently, such detections are deactivated by default in E-ACSL, but may be performed on user request. I do not detail the extended shadow model here. The interested reader may refer to the above-mentionned paper [VKSJ17].

Reducing Instrumentation

Even if the E-ACSL block-level shadow memory model is efficient, it is also possible to improve the efficiency of the monitored program by limiting its usage by the monitored code. Consider for instance the function `f` below.

```
int f(void) {
  int x, y, z, *p;
  p = &x;
  x = 0;
  y = 1;
  z = 2;
  /*@ assert \valid(p); */
  *p = 3;
  return x;
}
```

It does nothing very interesting, but declares four local variables, makes a few assignments, and contains one single E-ACSL annotation in order to verify that pointer `p` is valid before dereferencing it.

A complete instrumentation as in Figure 3.13 would generate the monitored function of Figure 3.17. First, it stores in the runtime memory model the 4-byte memory blocks corresponding to the four local variables. Then, before each assignment, it set the initialization status of every byte of the assignee in the runtime memory model through calls to function `__e_acsl_full_init`. Next, it queries the runtime memory model to verify the validity status of pointer `p` through a call to function `__e_acsl_valid`. Finally, it removes from the runtime memory model the local variables (by zeroing out the corresponding memory block for the shadow model). Even if the runtime model is efficient, the execution time of all these function calls is not negligeable compared to the execution time of the original program (even if it is still instantaneous for this small function).

However, if one closely looks at the generated code, one may observe that several generated function calls are actually useless, as indicated on Figure 3.17. Indeed, the only goal of the generated monitor is to verify the E-ACSL assertion about the validity of pointer `p`. For this purpose, there is absolutely no need to register

```
int f(void) {
  int x, y, z, *p;
  __e_acsl_store_block((void *)(&p),4);
  __e_acsl_store_block((void *)(&z),4); /* useless */
  __e_acsl_store_block((void *)(&y),4); /* useless */
  __e_acsl_store_block((void *)(&x),4);
  __e_acsl_full_init((void *)(&p));
  p = &x;
  __e_acsl_full_init((void *)(&x));     /* useless */
  x = 0;
  __e_acsl_full_init((void *)(&y));     /* useless */
  y = 1;
  __e_acsl_full_init((void *)(&z));     /* useless */
  z = 2;
  /*@ assert \valid(p); */
  {
    int __gen_e_acsl_valid;
    __gen_e_acsl_valid = __e_acsl_valid((void *)p,sizeof(int));
    __e_acsl_assert(__gen_e_acsl_valid);
  }
  *p = 3;
  __e_acsl_delete_block((void *)(&p));
  __e_acsl_delete_block((void *)(&z));  /* useless */
  __e_acsl_delete_block((void *)(&y));  /* useless */
  __e_acsl_delete_block((void *)(&x));
  return x;
}
```

Figure 3.17: Example of monitoring reduction through static analysis.

in the runtime memory model pieces of information that are not related to the memory block corresponding to this pointer. In particular, one could safely remove everything instrumenting variables `y` and `z`. However, one should be cautious while doing so. For instance, one cannot remove the instrumentation code about variable `x` since pointer `p` and `&x` are aliased by the very first assignment: their memory blocks are actually the same. Consequently, adding `&x` in the store is still required to get a correct result at runtime. Nevertheless, one could safely remove the call `__e_acsl_full_init((void *)(&x));` when assigning a value to `x`. Indeed, the monitor only needs to check the validity status of `p`, *i.e.* `&x`, not the initialisation status of its contents `x`. One in the other, one could remove about half of the instrumentation on this small example: that is not negligeable at all.

In order to do such an optimization automatically, E-ACSL runs a dataflow analysis before generating code that would call the memory model for the first time. This analysis corresponds to the box "memory model analysis" in Figure 3.9. It computes a sound over-approximation of memory locations that must be monitored at a particular program point in order to be sure that the runtime memory model contains the required information when evaluating E-ACSL memory-related annotations. This analysis does not necessarily need to be very precise since it is only in charge of optimizing the generated code, but should be sound, fully automatic, and pretty fast to be executed. I first implemented this analysis in 2012 [KPS13a]. It relies on an alias analysis which was done at the same time. During his internship, Arvid Jakobsson implemented a new version parameterized with such an alias analysis [Jak14] which was Bjarne Steensgaard's algorithm [Ste96] in practice. Nikolai Kosmatov and I formalized this version in 2015 for a subset of C and E-ACSL [JKS15b, JKS16]. During his postdoc under my supervision, Gergö Barany continued to improve it in 2016. Dara Ly eventually made a more precise formalization and proved it in 2017 during his master internship under my supervision [Ly17, LKLS18].

Despite a large amount of work already dedicated to this analysis, none of the existing implementations is fully satisfactory to my eyes. Indeed, they are pretty slow to run on large programs, while hardly dealing with the most complicated features of the C language. Consequently, this analysis must sometimes be deactivated when generating monitors for large and complex programs. Yet, this analysis is important for reducing E-ACSL's runtime and memory overhead in many, if not most, use cases. In order to improve this situation, one could first try to work on the latest implementation of this analysis, which has been only lightly tested, debugged and profiled. If not sufficient, these additional experiments will certainly help to understand what the issue is. One possible global problem might be that this analysis is currently inter-procedural and so performs a whole-program analysis. If these experiments show that it is problematic for its efficiency, it could lead to the design of a new intra-procedural analysis. Such a study is future work.

Benchmarking

I just concluded the previous paragraph by explaining that additional experiments about the dataflow analysis are necessary. Indeed, a runtime verifier like E-ACSL needs a lot of empirical evaluations in order to determine its expressiveness and its efficiency, usually measured through runtime and memory overheads with respect to the uninstrumented original program. It also needs to be compared to similar tools.

Consequently, a huge amount of time since 2012 has been dedicated to this work. First, Guillaume Petiot compared his Patricia-trie model to models relying on other data structures, including splay trees (the most standard data structure for dictionary-based runtime memory model), and to Valgrind (based on shadow memory encoding). Its evaluation also included the first implementation of the dataflow analysis. Even if the evaluation was only performed on small-size C programs like sorting algorithms, he shows that his runtime memory model was at least as competitive as the other dictionary-based ones but slower than Valgrind (without surprise), and that the static analysis always reduces the runtime overhead, sometimes drastically [KPS13a].

Second, Arvid Jakobsson compared his hybrid model to the Patricia-trie model and to his shadow model on the same benchmark used by Guillaume Petiot [JKS16]. It demonstrated that the hybrid model was hugely faster than the Patricia-trie model and comparable to his shadow model. It also confirmed that the dataflow analysis was very useful for reducing the runtime overhead (by more than 70% on average). The runtime overhead of the hybrid model is almost incomparable to Valgrind: it was faster on some examples and slower on some others. However, in addition to Guillaume Petiot's evaluation, this one also compared memory overhead and concluded that the hybrid model consumed less memory than Valgrind.

In parallel, in 2014, the Competition on Runtime Verification (CRV) was also created in order to "foster the process of comparison and evaluation of software runtime verification tools" [BFB+17]. It was split into three tracks: offline monitoring, monitoring of Java programs, and monitoring of C programs. Arvid Jakobsson, Nikolai Kosmatov and I participated with E-ACSL in the latter one and E-ACSL took the second place with a score slightly behind RiTHM [NJW+13]. I also participated alone with E-ACSL in the next edition in 2015 [FNRT15] and, this time, won the C track[37] with a score about twice that of RV-Monitor [LZL+14], developed by startup Runtime Verification Inc.[38], which took the second place. However, these two editions of the competition, particularly the C track, suffered from many weaknesses, one of the most important being that it was hard to estimate what was actually evaluated (*e.g.* tool expressiveness or efficiency?) [Sig17]. Also, and proba-

37. https://www.rv-competition.org/2015-2/
38. https://runtimeverification.com/

bly related, only a few tools competed in the C track during these two editions and no tool but E-ACSL registered to participate in the C track during the 2016 edition, which was thus cancelled. As a consequence, the competition organizers decided not to organize it in 2017 and to replace it by the workshop RV-CuBES [RH17] in parallel to the RV conference. However, I think that the two organized editions were at least beneficial for E-ACSL since they helped it to be better known in the academic runtime verification community and so achieved one of the competition goals, which was "to enhance the visibility of presented tools" [BFB+17].

The most important benchmarking efforts on E-ACSL were done by Kostyantyn Vorobyov in 2016 and 2017, during his postdoc, in order to evaluate the E-ACSL shadow model. First, 17 C programs ranging from 74 to 36,037 (on average, 9,345) lines of code from SPEC CPU benchmarks [39] were used to evaluate the runtime and memory overhead of E-ACSL with respect to E-ACSL using the Patricia trie model and tools such as AddressSanitizer [SBPV12], MemCheck [SN05] and Dr. Memory [BZ11] (32- and 64-bit). All these tools use a shadow memory model as explained in the state of the art at the beginning of this section. For E-ACSL, the annotations were automatically generated through the plug-in Rte in order to check the properties shown in Figure 3.18. Runtime overheads are shown in Figure 3.19, while memory overheads are presented in Figure 3.20.

	E-ACSL	AddressSanitizer	MemCheck	Dr. Memory 32/64
Heap Tracking	+	+	+	+ +
Stack Tracking	+	+	−	+ +
Global Tracking	+	+	−	+ +
Allocation	+	+	+	+ +
Initialization	±	−	+	+ −
Pointer Init	+	−	−	− −
Bounds Check	+	±	±	± ±
Arith. Overflow	+	−	−	− −
Read-only	+	−	−	− −
Block Properties	+	−	−	− −
Heap Leak	±	+	+	+ +

Figure 3.18: Properties tracked during experimentation.

I let the reader refer to the original paper [VSK17] for detailed explanations, particularly with respect to the exact experimental setup and threats to validity, but I summarize here the results. The E-ACSL shadow model has a runtime overhead of about 19 times on average (ranging from 8 times to 55 times) with respect to

39. https://www.spec.org/cpu/

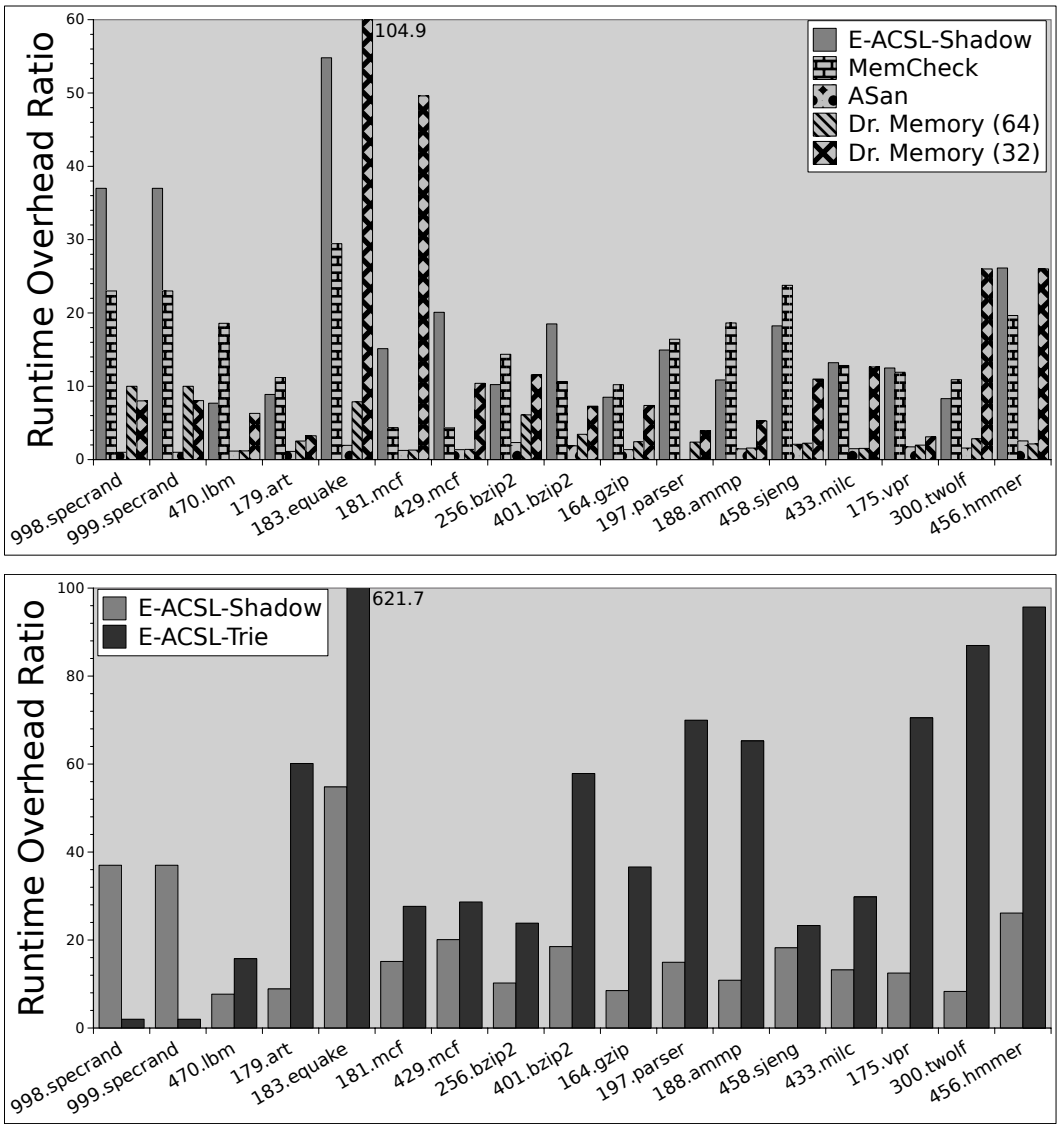


Figure 3.19: Runtime overhead of E-ACSL, AddressSanitizer, MemCheck and Dr. Memory on SPEC CPU programs.

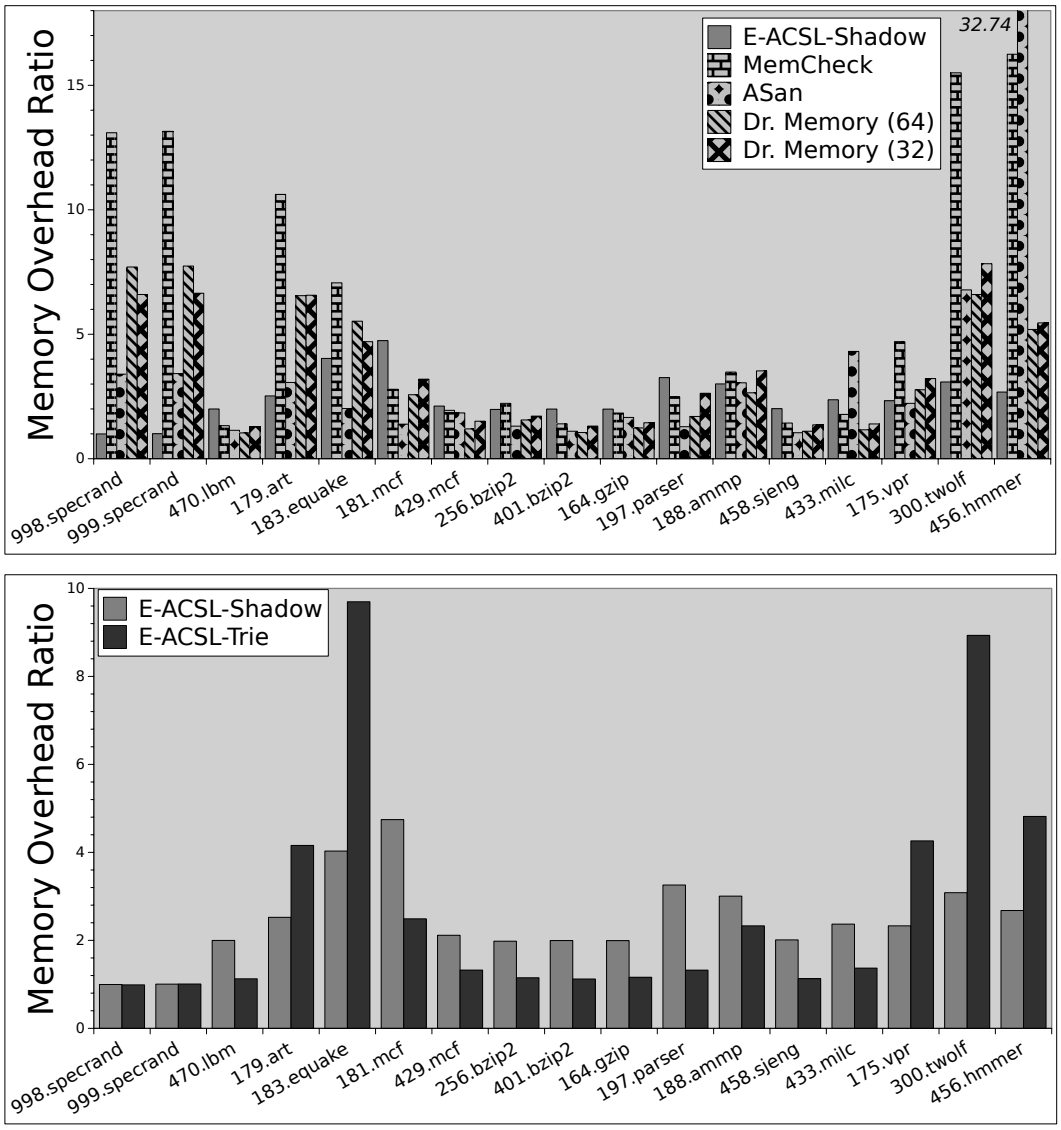


Figure 3.20: Memory overhead of E-ACSL, AddressSanitizer, MemCheck and Dr. Memory on SPEC CPU programs.

the normal execution time without any instrumentation. It has a memory overhead of 2.48 on average (ranging from 1.01 to 4.74) with respect to an uninstrumented execution. The results confirm that it is significantly faster than the Patricia trie model which has an average overhead of 77.5, ranging from 2 to 662. However, it is often slower on small programs because of the constant time initialization of the shadow space which becomes visible in such a context. The memory overhead of the Patricia trie model is slightly higher too, with an average of 2.85, ranging from 1.01 to 9.7. The runtime overhead of E-ACSL (with the shadow model) is slightly higher but in the same order of magnitude as MemCheck (average overhead of 15.49 times) and Dr. Memory 32-bit (average overhead of 18 times), while AddressSanitizer and Dr. Memory 64-bit are significantly faster (average overheads of 1.58 and 3.59, respectively). The difference between Dr. Memory 32-bit and 64-bit comes probably from the fact that the former checks the initialization statuses, while the latter do not. The memory overheads of AddressSanitizer, Dr. Memory 32- and 64-bit and MemCheck are on average 4.22, 3.37, 7.74 and 5.95 respectively, so higher than E-ACSL. To be fair, AddressSanitizer memory overhead comes from a maximum on a particular program. Without taking it into account, its average is 2.44 times, so similar to E-ACSL. All of these tools check fewer properties than E-ACSL, particularly because they do not track block-level properties. Consequently, E-ACSL may be seen as a good replacement for these tools, adding the advantage of tracking more properties at the cost of a higher, yet affordable, runtime overhead.

Last, Kostyantyn Vorobyov also evaluated the detection power of E-ACSL with respect to standard security defects of C programs [VKS18]. This time, the focus was not put on tool efficiency, but on tool expressivity. E-ACSL was compared to Google's sanitizers (AddressSanitizer [SBPV12], MemorySanitizer [SS15], ThreadSanitizer [SPIV11] and UndefinedBehaviorSanitizer [40]) seen as a single tool and RV-Match [GHSR16], the automatic debbuger provided by Runtime Verification Inc. and based on the $\mathbb{K}$ framework [ER12, Ell12]. This debugger is a tool of choice for such an evaluation since it aims at finding as many C undefined behaviors as possible. The evaluation was done on the NIST's SARD-100 test suite [41] and Toyota ITC benchmark [SMM15] [42]. The results are respectively presented in Figures 3.21 and 3.22.

Considered altogether, they show that all tools cannot detect some kinds of security defects that are out of their scopes (*e.g.* SQL injection). This is not surprising since none of them pretends to detect any security defects: they are not at all equipped to detect several of them. Also E-ACSL has currently no support for multi-threaded programs and concurrency defects. Its other current weaknesses pointed out by these benchmarks are the lack of support for detecting incorrect func-

40. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`
41. `https://samate.nist.gov/SRD/view.php?tsID=100`
42. `https://github.com/Toyota-ITC-SSD/Software-Analysis-Benchmark`

	E-ACSL	Google's Sanitizers	RV-Match
Non-memory Defects			
CWE-078: Command Injection	– (0/6)	– (0/6)	– (0/6)
CWE-080: Basic XSS	– (0/5)	– (0/5)	– (0/5)
CWE-089: SQL Injection	– (0/4)	– (0/4)	– (0/4)
CWE-099: Resource Injection	– (0/4)	– (0/4)	– (0/4)
CWE-259: Hard-coded Password	– (0/5)	– (0/5)	– (0/5)
CWE-489: Leftover Debug Code	– (0/1)	– (0/1)	– (0/1)
Memory Defects			
CWE-121: Stack Buffer Overflow	✓(11/11)	91% (10/11)	91% (10/11)
CWE-122: Heap Buffer Overflow	✓(6/6)	✓(6/6)	✓(6/6)
CWE-416: Use After Free	✓(9/9)	✓(9/9)	✓(9/9)
CWE-244: Heap Inspection	– (0/1)	– (0/1)	– (0/1)
CWE-401: Memory Leak	✓(5/5)	80% (4/5)	60% (3/5)
CWE-468: Pointer Scaling	50% (1/2)	50% (1/2)	50% (1/2)
CWE-476: Null Dereference	✓(7/7)	✓(7/7)	✓(7/7)
CWE-457: Uninitialized Variable	✓(4/4)	75% (3/4)	✓(4/4)
CWE-415: Double Free	✓(6/6)	✓(6/6)	67% (4/6)
CWE-134: Format String	✓(8/8)	– (0/8)	– (0/8)
CWE-170: String Termination	✓(5/5)	✓(5/5)	✓(5/5)
CWE-251: String Management	✓(5/5)	✓(5/5)	✓(5/5)
CWE-391: Unchecked Error	– (0/1)	– (0/1)	– (0/1)
Concurrency Defects			
CWE-367: Race Condition	– (0/4)	– (0/4)	– (0/4)
CWE-412: Unrestricted Lock	– (0/1)	– (0/1)	– (0/1)
Overall	67% (67/100)	56% (56/100)	54% (54/100)

Figure 3.21: Detection results of E-ACSL, Google's sanitizers and RV-Match over SARD-100 test suite.

Defect Type	E-ACSL	Google's Sanitizers	RV-Match
Dynamic Memory	94% (81/86)	78% (67/86)	94% (81/86)
Static Memory	✓(67/67)	96% (64/67)	✓(67/67)
Pointer-related	56% (47/84)	32% (27/84)	99% (83/84)
Stack-related	35% (7/20)	70% (14/20)	✓(20/20)
Resource	99% (95/96)	60% (58/96)	98% (94/96)
Numeric	93% (100/108)	59% (64/108)	98% (106/108)
Miscellaneous	94% (33/35)	49% (17/35)	71% (25/35)
Inappropriate Code	– (0/64)	– (0/64)	– (0/64)
Concurrency	– (0/44)	73% (32/44)	66% (29/44)
Overall	71% (430/604)	57% (343/604)	84% (505/604)

Figure 3.22: Detection results of E-ACSL, Google's sanitizers and RV-Match over Toyota ITC benchmark.

tion pointers (notably involved in Toyota ITC's "pointer-related" category), stack overflows (notably involved in Toyota ITC's "stack-related" category) and overflows through bitfield values (notably involved in the Toyota ITC's "numeric" category). Without taking these limitations into account, E-ACSL has better results than the other tools, in particular Google's sanitizers (as expected because of its more expressive memory model). Also it is worth noting that RV-Match detection capabilities are quite good, but the issue with this tool (at least with the academic version that is freely available) is efficiency: trying RV-Match on large programs from SPEC CPU just fails almost always. To be fair, this tool is based on symbolic execution and not on concrete execution, so is necessarily less efficient than standard runtime verifiers.

These evaluations of E-ACSL on several large benchmarks and its comparison with other state-of-the-art tools demonstrate that it is able to handle complex constructs from its specification language with a pretty decent runtime overhead and a good memory overhead. Thus, it is able to detect many security defects that may appear in C programs. Even if much opportunities for improvement are possible as explained in Section 3.5, it emphasizes that E-ACSL may already be considered as a tool of choice for debugging C programs at any step during its development.

3.4 E-ACSL, a Tool with Multiple Usages

Context This section takes ideas from the E-ACSL tool paper [SKV17]. Subsection 3.4.1 is also based on a paper co-published with Dillon Pariente from Dassault Aviation [PS17], while subsection 3.4.2 presents some results from the latest paper about Secure Flow [BS17].

Besides debugging C programs and checking formal annotations, E-ACSL has several other uses introduced in this section.

3.4.1 Finding Undefined Behaviors

As illustrated with the experimentation presented at the end of the previous section, the most important usage of E-ACSL today is certainly debugging C programs by finding their undefined behaviors at runtime. This usage is very similar to AddressSanitizer's and Valgrind's, in particular regarding memory safety. Besides removing bugs, this usage of E-ACSL is of the utmost importance from a security point-of-view since it points out bugs such as memory errors (as illustrated by benchmark SARD-100 in Figure 3.21). In 2012, such memory errors were still among the top 3 most dangerous software errors and account for about half of the reported security vulnerabilities [vdVdSCB12].

In this context, E-ACSL works in sequential collaboration with plug-in Rte which generates E-ACSL annotations for a large class of C undefined behaviors. This way, the process is fully automatic since no user-defined annotations are required. Also, E-ACSL may work in collaboration with plug-in Eva which generates Rte-like annotations each time it cannot soundly statically ensure that an undefined behavior never happens at runtime. This idea is close to Hasan Sözer's proposal to use runtime verification techniques to discharge alerts raised by static analyzers [Sö15, KS17]. This approach has been automatated in method Cursor proposed by Dillon Pariente from Dassault Aviation [PS17] and summarized in Figure 3.23. I contributed to this work through support and expertise in the context of European projects *Stance* [43] (finished in 2016) and *Vessedia* [44] (started in 2017).

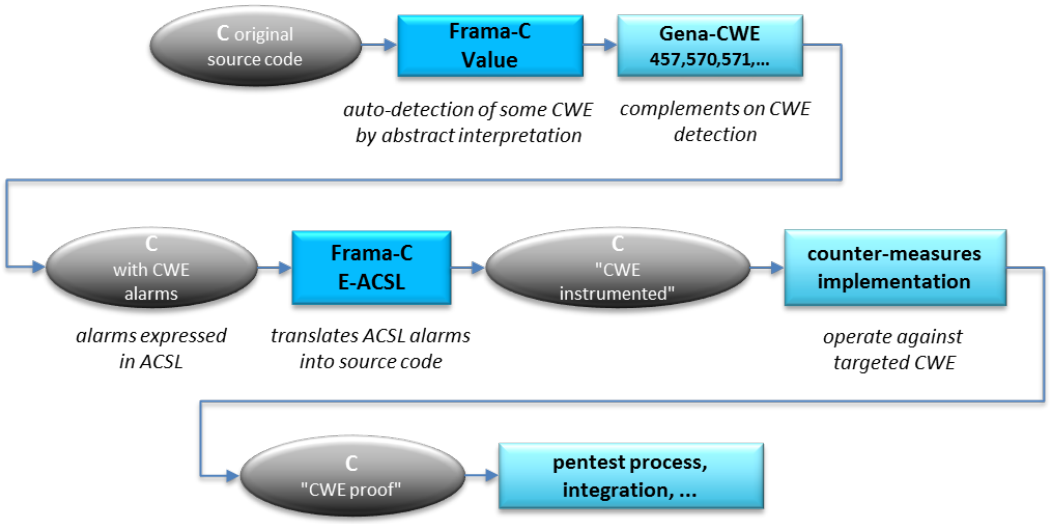


Figure 3.23: Cursor method process proposed by Dassault Aviation.

The main goal of this method consists in limiting as much as possible the amount of work to be done by the engineer, in particular when setting the right Eva parameters. This way, it reduces the global verification cost. Indeed, instead of taking time to set the best Eva parameters to get the fewest possible numbers of Eva's alarms, it is more efficient to spend almost no time on this task, even if it means having many statically-unverified alarms. These alarms are then verified at runtime by E-ACSL. In this method, Eva static verification is also extended with two plug-ins developed by Dassault Aviation, namely Gena-Taint and Gena-CWE. The former is a taint analysis plug-in based on Eva's results. The latter is an Eva extension that detects

43. `http://www.stance-project.eu/`
44. `https://vessedia.eu/`

more CWEs than Eva alone. Last but not least, this method also deploys security counter-measures at runtime by modifying the default behavior of the E-ACSL function `e_acsl_assert`. For instance, when detecting an undefined behavior, it could log what happened, halt a service or the whole system, or apply a dedicated defensive measure.

This method was successfully evaluated on a few modules of Apache and OpenSSL. Interestingly, the runtime overhead of E-ACSL on these pieces of code ($\times 19$) was about the same as those obtained on benchmark SPEC CPU. But, when Eva was run first in order to statically remove potential alarms (even without any precise parameterization), the runtime overhead was only about 3 times the uninstrumented version. This overhead is similar to tools such as AddressSanitizer, at the price of a longer static analysis time. Consequently, even if it still needs additional evaluation, this method is very promising.

3.4.2 Detecting Information Leakage

Another way to use E-ACSL automatically (*i.e.* without writing annotations manually) consists in detecting information flow leakage in combination with plug-in Secure Flow. This plug-in was originally developed by Mounir Assaf during his PhD at LSL under my supervision, in collaboration with Éric Totel and Frédéric Tronel at CentraleSupélec [Ass15, ASTT13a, ASTT13b]. Next, it was extended by Gergö Barany during his postdoc at LSL, still under my supervision [Bar16, BS17].

Secure Flow Overview

Secure Flow is also a program transformation. It converts the original C code extended with a few dedicated security annotations into a new C code containing E-ACSL annotations. This new code encodes the security statuses of data as labels *public* or *private* stored in new C variables. The information flows (how these statuses propagate through the program control flow) are encoded in new C statements. This way, the plug-in detects leakage of private data into public channels. More precisely, the encoded property is *Termination Insensitive Non Interference* (TINI) [BR16]. The original security annotations are also translated into standard E-ACSL annotations that refer to the newly introduced variables corresponding to the security statuses. A small ACSL extension allows the user to check properties about security statuses by using a dedicated logic function `security_status` and dedicated constants `private` and `public`.

Consider for instance the following code snippet.

```
if (user_input == secret) x = 1; else y = 1;
/*@ assert security_status(x) == private; */
```

It checks that variable `x` *does* leak a secret at the point of the assertion. Indeed, there is a so-called implicit information flow from variable `secret` to variable `x`, because the latter is modified in a conditional whose the guard depends on the former. Plug-in Secure Flow generates the code snippet of Figure 3.24 for this piece of code.

```
pc_label_1 = pc_label_0 | user_input_label | secret_label;
if (user_input == secret) {
  x_label = 0 | pc_label_1;
  x = 1;
  y_label |= pc_label_1;
} else {
  y_label = 0 | pc_label_1;
  y = 1;
  x_label |= pc_label_1;
}
/*@ assert x_label == 1; */
```

Figure 3.24: Secure Flow Encoding for a simple conditional.

Here, the security status of a variable v is encoded in the variable v`_label` with values in $\{0, 1\}$ [45]. Value 0 means *public* while value 1 denotes *private*. Bitwise disjunctions propagate the information flow from label to label in order to convert public data to private as soon as the former depends on the latter. From a mathematical point of view, the set of security labels (of cardinal 2) is a lattice whose the join operator is bitwise disjunction. The special labels `pc_label_`i encode the security label of the control flow itself. Here, it propagates the security label of `secret` involved in the conditional to variables `x` and `y` updated in both branches. Prefix `pc` stands for *program counter*. Such program counter labels are quite standard in information flow tracking (see for instance Daniel Hedin and Andrei Sabelfeld's survey [HS12]).

The reader may have noticed that labels for variables `x` and `y` are updated in both branches of the conditional, while each variable is modified only in one branch. Indeed, this is a necessary condition for this kind of analysis to be sound, as demonstrated by Alejandro Russo and Andrei Sabelfeld [RS10]. In practice, Secure Flow relies on Eva to compute an over-approximation of such sets of modified variables. Such a tool that depends on a static analysis while encoding the

45. The encoding is actually more complex for variables of compound types and for pointers and arrays, but I do not provide this level of technical details here. The interested reader may refer to the Secure Flow papers for details (in particular, [ASTT13b, BS17]).

information flows in the original program is usually called a *hybrid information flow monitor* [BR16].

In this context, the main contribution of Mounir Assaf's PhD is to have proposed the first correct hybrid information flow monitor in presence of pointers and aliasing [ASTT13b, Ass15]. Gergö Barany extended the proof to arrays and pointer arithmetic [Bar16], and the approach to most C constructs (without proof) [BS17].

Secure Flow Usage

Similarly to E-ACSL, which is useless without a C compiler, Secure Flow is useless without a verification tool since it only generates code. In order to verify the absence of leakage with Secure Flow, the user *must* use a code analysis tool on the generated program. However, Secure Flow does not require any particular tool: the user *may* use his/her tool of choice, even if (s)he must rely on Eva when generating the code with Secure Flow. This approach is schematized in Figure 3.25. In particular, this figure emphasizes that either static analysis (*e.g.* through Eva or Wp), or runtime verification (*e.g.* through E-ACSL) may be used in practice.

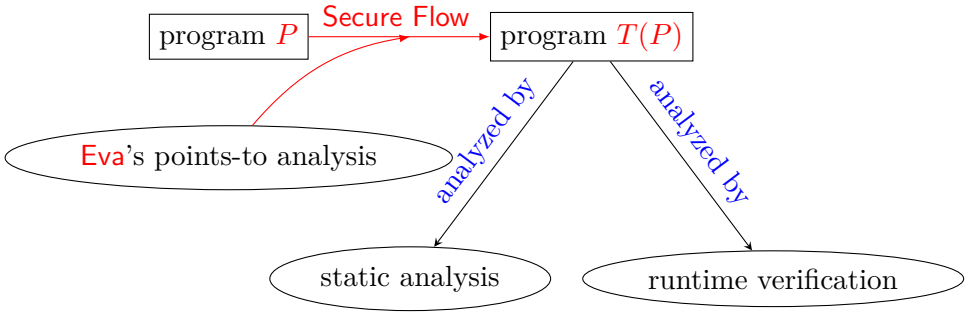


Figure 3.25: Secure Flow Operational Principle.

At the end of his postdoc, Gergö Barany experimented with this approach on the library LibTomCrypt and its dependency LibTomMath [46]. LibTomCrypt provides many cryptographic routines. Secure Flow was in particular used to detect a certain class of timing attacks (or their absence) over all the fourteen symmetric crypto-functions provided by LibTomCrypt (*e.g.*, including AES) [BS17]. More precisely, the chosen security policy was to enforce that no conditional branch depends on the secret key. Indeed, if the executions of the two branches take different amounts of time, an attacker able to measure this time may deduce information about the

46. `http://www.libtom.net`

secret key. Such attacks are known against implementations of several cryptosystems in common use, including RSA and elliptic curve cryptography [GPP+16]. Actually, even if timing differences cannot be measured directly, attackers on the same machine may be able to detect cache misses that allow them to deduce the same kind of information [Per05].

A command line option has been added to Secure Flow to automatically add the necessary assertions to enforce this security policy. Next, on every symmetric cryptosystem, Secure Flow was run in combination to Eva. It was enough to conclude that these implementations are secure with respect to the checked security policy, but Eva was not used to check all the safety properties. In other words, after running Eva, there remained no alarm corresponding to Secure Flow assertions, but several ones corresponding to potential undefined behaviors. Finally, Secure Flow was also tested in combination with E-ACSL on concrete executions in order to validate both the security assertions and the absence of undefined behaviors (of the class supported by plug-in Rte, see Section 3.4.1). Figure 3.26 presents the overhead introduced by Secure Flow alone, and by Secure Flow and E-ACSL together. The former are about twice the execution time of the original program, while the latter varies from 14.0× to 22.2× and so are consistent with the results observed on the SPEC CPU benchmark. The Secure Flow memory overhead is 1.6× the original program in average. E-ACSL adds an additional factor of 3.7 on average.

Program	Secure Flow Overhead	+E-ACSL Overhead
aes	3.0 ×	20.8 ×
anubis	2.8 ×	19.0 ×
blowfish	2.3 ×	15.9 ×
cast5	1.9 ×	16.9 ×
kasumi	1.8 ×	14.8 ×
khazad	2.2 ×	22.0 ×
kseed	1.8 ×	14.3 ×
noekeon	1.9 ×	16.3 ×
rc2	1.5 ×	12.2 ×
rc5	2.3 ×	22.2 ×
rc6	2.0 ×	20.6 ×
saferp	7.0 ×	14.0 ×
twofish	2.4 ×	19.4 ×
xtea	1.6 ×	15.0 ×

Figure 3.26: Runtime overhead of Secure Flow instrumentation and Secure Flow +E-ACSL instrumentation on LibTomCrypt's symmetric cryptofunctions.

3.4.3 Checking Functional Properties

Being a runtime assertion checker for the E-ACSL specification language, the E-ACSL tool is of course able to check functional properties at runtime. The process is usually not fully automatic, since the user must manually write specifications. However, the necessary effort is not as important as verifying the same properties statically. Indeed, such a static verification is usually done through deductive methods (*e.g.* with plug-in Wp in the Frama-C world) which require adding annotations to help the verification process, such as loop annotations, extra assertions and/or lemmas. On the other hand, runtime verification requires an execution context and does not provide a guarantee for all possible executions but only for the tested ones. In that respect, using E-ACSL or Wp for checking functional properties is a matter of choice.

Nevertheless, I think that both tools and techniques may be complementary in several ways. First, runtime verification may help to debug specifications before proving them. Indeed, writing specifications is error-prone, so it is worth testing them by running E-ACSL on unit test cases before trying to prove them. This approach could be particularly interesting when teaching a formal specification language such as ACSL. It is even more fruitful when students also learn program testing and program proving in addition to formal specification, since they may practice all these activities within the same framework. It would advance existing teaching practices [KPS13b] another step forward.

Second, runtime verification with E-ACSL could also help program proving with Wp *during* the proof stage. Indeed, writing correct specifications is difficult, but it is certainly even harder to write the required annotations (in particular, loop invariants) to prove them. When the proof fails (*i.e.* no automatic theorem prover is able to prove a particular proof obligation), it is not always clear *why* the proof fails. It usually comes from several reasons, in particular the code or the specifications (or both) may be wrong, a proof annotation such as a loop invariant may be wrong or missing, or the used automatic theorem prover(s) may be not powerful enough. Actually, this use case is exactly the *raison d'être* of plug-in StaDy which builds a bridge between the test case generation plug-in PathCrawler and Wp in order to help understand proof failure [PKB+16]. E-ACSL could be used for this activity too. Indeed, StaDy transforms ACSL annotations (actually E-ACSL ones) into C code in a way very similar to that of E-ACSL [PBJ+14].

Finally, runtime verification with E-ACSL and program proving with Wp may be complementary, in the same way that E-ACSL and Eva are complementary when verifying the absence of undefined behaviors: some properties may be proven by Wp, while some others may be checked at runtime with E-ACSL. This approach offers more guarantees than runtime verification alone, while being more cost-effective than program proof alone, since the user has no obligation to write the necessary ad-

ditional proof annotations for all his/her specifications. In this context, particularly in order to use this approach for verifying critical software such as DO-178C level-A avionic software, one difficulty is to know when enough verification effort has been spent. Indeed, when using program proof alone, this task is easy: the verification step is complete when all proof obligations are verified. Also, for testing, many structural and functional coverage criteria exist to solve this problem [AO16]. However, there is currently no satisfactory way to detect whether enough verification effort has been spent when combining both techniques. Viet Hoang Le is currently solving this issue during his PhD that I supervise together with Loïc Correnson at LSL and Virginie Wiels at Onera. He proposes a new coverage criterion to combine test and proof. He is assessing it on examples for which some specifications are proved with Wp, while some others are verified by E-ACSL on concrete executions [LCSW18].

3.4.4 Additional Compilation-Based Usages

I already described several usages of E-ACSL in which the tool verifies annotations generated by other plug-ins, such as Rte, Eva or Secure Flow. More generally, the E-ACSL specification language and the C programming language may be seen as low-level languages that a higher-level language is compiled to. This compilation-based approach may be applied to any Frama-C plug-in or external tool that generates E-ACSL annotations. That is for instance the case of Aoraï, the Frama-C plug-in generating annotations from Büchi automata (see Section 2.6, page 21). I already tried such a combination of Aoraï and E-ACSL on toy examples. It works fine. It should now be evaluated on larger examples. I think that this usage partially fills the gap between E-ACSL and runtime verifiers for temporal logics such as LTL [BLS11]. However, the latter usually tackles more complex logics than LTL, such as the recent Metric Dynamic Logic (MDL) [BKT17]. I also think that a dedicated runtime verifier for LTL is certainly more efficient than E-ACSL, which may suffer from the double encoding issue discussed in Section 3.5.1.

Arnaud Dieumegard *et al* [DGK+15] also rely on such a compilation-based approach in order to check that a Simulink model is consistent with synchronous observers. They first compile a Simulink model to Lustre code with a home-made tool. Then, they express properties as synchronous observers at Lustre level. Next, the Lustre code and their observers are compiled to C code annotated with E-ACSL annotations. They finally rely on either Wp, or E-ACSL to verify it.

3.5 E-ACSL, an Evolving Tool

Context This section is novel.

Even if its development started in 2011, I consider that E-ACSL is still a young tool. Indeed, since the beginning, only small effort has been put on its development (less than one person by year). Also, developing a full code analyzer is always a long-term effort in order to improve it from an academic prototype to a tool usable in different industrial contexts. I consider that the development of E-ACSL is now in midstream. It has been successfully applied on its first use cases. However, many research and development activities are still to be done, in particular to extend its usages and its expressiveness, while better understanding what E-ACSL really is.

This section is dedicated to these perspectives. First, Section 3.5.1 investigates what could be new usages for E-ACSL and what are the associated required developments. Next, Section 3.5.2 explains how E-ACSL could be improved. Finally, Section 3.5.3 discusses the possible research actions that would contribute to better understand E-ACSL from a theoretical point of view.

3.5.1 Adding New Usages

First, as already explained, E-ACSL may be used in combination with other plug-ins, notably Rte, Eva, Secure Flow, and Aoraï, that generate E-ACSL annotations or compile high-level specifications to E-ACSL-annotated C source code. Arnaud Dieumegard *et al* also demonstrated that the same approach can be adopted for models and their synchronous specifications [DGK+15]. I think it would be worth studying this approach.

Currently, the Rte plug-in only addresses a (large) class of undefined behaviors, but some of them are not yet handled for different reasons. Furthermore, detecting implementation-defined behaviors is also interesting in some particular industrial contexts. A systematic approach could be adopted to explore what is not yet supported, by exploring the C99 norm from the start to the end. Extending plug-in Rte by generating annotations for new kinds of undefined or implementation defined behaviors will extend the E-ACSL automatic detection power, as soon as the generated annotations are supported by E-ACSL. For instance, using an uninitialized left-value was not detected by Rte up to Frama-C release Sulfur-20171101, because it introduced a too large number of (trivial) assertions that annoy analyzers like Eva. It was eventually implemented — even if deactivated by default — in order to be able to automatically check this kind of property with E-ACSL, increasing *de facto* its automatic detection power.

Another possible usage of E-ACSL is handling C++ programs. Indeed, it would be worth trying E-ACSL on the code generated from plugin Clang that converts C++ code to the Frama-C AST. In particular, since the C generated code is quite complicated, it is hard to use static analysis in combination with plug-in Clang. However, E-ACSL is more concerned by the complexity of the annotations than that of C code, so it should work even on complicated C encoding of C++ constructs.

Yet, in order to express specific properties of C++ programs (in particular, related to objects), extensions of ACSL and E-ACSL for C++ must be designed. The just-started 18-month researcher position of David R. Cok — one of the main JML contributors [BCC+05] in particular through OpenJML [Cok11] — at LSL could help to reach this goal.

In this compilation-based setting, it would be interesting to design a plug-in that would ensure that ghost code (see Section 3.2.13) does not interfere with the original C code. This feature is the key property of ghost code. Why3, for instance, ensures it through a dedicated type system with effects [FGP16]. There is no such verification in Frama-C today. The intricacies of C code (in particular, pointers) make any solution more difficult to design than the one for Why3. It would certainly be possible to restrict usage of ghost code by design in a way that verifying this property statically becomes decidable. However, since it is a non-interference property, another solution would consist in adapting a non-interference analysis to this particular case, such as the Secure Flow program transformation (see Section 3.4.2). If such a program transformation would be designed and implemented, E-ACSL would be a tool of choice to verify the generated code at runtime.

However compilation-based approaches potentially suffer from (what I call) the *double encoding issue*. Indeed, such approaches consist in encoding a problem P_1 into another one P_2 (possibly in another language) before converting it into a third one P_3, suitable for verification. Since each compilation step possibly adds a large amount of new complicated code, the final code becomes huge and difficult to verify. For instance, Secure Flow information flow encoding generates a large amount of new annotated C code. All these pieces of code are in turn compiled to a larger C code by E-ACSL. All in all, the size of the final code is one or two order(s) of magnitude bigger than the original code. If the initial code is already large, the result is just too large. Handling C++ code may lead to the very same problem, since the amount of code generated by plug-in Clang is large. A partial solution consists in implementing optimization techniques in the program transformation steps (see Section 3.5.2 for E-ACSL), but it will not completely fix this double encoding issue.

A better choice could consist in designing a dedicated solution for each analyzer (or verification technique, at least). For instance, it could lead to the design of new dedicated abstract domains for Eva, such as one for security labels to replace using Secure Flow in combination to Eva. Regarding E-ACSL, a possible solution would consist in using the useless bits of its memory model to store dedicated pieces of information, such as security labels. This way, it would generate much more efficient code. The drawback of this approach is the need to design several dedicated solutions (one *per* analysis), while a compilation-based approach provides a general solution and allows the user to choose his/her verification tool of choice for each particular problem.

More generally, the E-ACSL instrumentation engine and the useless bits of its

memory model could be used to compute and store different pieces of information about a program run, its traces and/or its data. They could be useful either later during the same execution, or during a post-processing stage. An example of the former usage would be a runtime concurrency analysis in which the memory model would store resource owners in order to detect incorrect accesses to shared resources. An example of the latter usage could consist in storing pieces of information in a database in order to train a neural network after millions of executions. Then, based on this large sampling, a machine learning algorithm could perform *offline monitoring* or *property synthesis*, such as invariant synthesis.

Instead of *offline monitoring*, another research direction consists in transforming E-ACSL from an inline runtime verification tool to an *outline* runtime verification tool. Indeed, in some industrial settings such as constrained embedded systems, inline monitoring is unusable because the generated monitor modifies the program environment. For instance, E-ACSL keeps for its own use a large part of the program memory. It also introduces new variables stored in the stack and/or the heap, as well as new statements. Outline monitoring may solve this issue by cleanly separating the monitor from the monitored program. A first step in this direction was performed by Quentin Bouillaguet during his master internship [Bou16] in 2016, that I supervised at LSL with Nikolai Kosmatov, Yves Lhuillier, and Gilles Mouchard. He designed a Frama-C plugin that generates an outline monitor that collaborates with Unisim-VP, the framework for virtual platforms developed at LSL [47]: when the simulator runs the monitored program, a communication channel between the monitor and Unisim-VP allowed the former to verify (simple) E-ACSL properties by asking Unisim-VP on the fly for the concrete values of the variables used in the properties while evaluating them. This approach could be adapted by replacing Unisim-VP with any C debugger, such as gdb.

Many more usages of E-ACSL certainly remain to be invented. They often come from industrial needs, and also from academics who invent new ways of using and combining existing tools for their specific needs. This is why industrial partnerships on one hand and academic collaborations on the other hand are often fruitful.

3.5.2 Improving E-ACSL

Another way to extend the application domains of E-ACSL consists in improving the tool itself. Currently, several features of the E-ACSL specification language are not yet implemented [Sigb], though some of them would be very useful. For instance, support of user logic functions and predicates (Section 3.2.12) is still missing, while being required in many use cases, for instance from Dassault Aviation, Fraunhofer Fokus, and Thales. Similarly, ranges (Section 3.2.8) would be useful for Dassault Aviation and Fraunhofer Fokus, while local binding

47. `http://unisim-vp.org/`

(Section 3.2.12) and rational numbers are required to handle a THALES use case. They will therefore be implemented soon. Some other constructs are necessary to handle standard Wp examples, such as `assigns` and `loop assigns` clauses, as well as `loop variant`, `complete behaviors`, and `disjoint behaviors`. Once they are implemented, checking functional properties with E-ACSL will be much easier (see Section 3.4.3). These developments have different levels of complexity. For instance, implementing local binding should be quite easy, while supporting the `assigns` clause is much more complex (for this latter clause, more than half of Hermann LEHNER's PhD is dedicated to runtime checking of a comparable construct in JML [Leh11]). Indeed, since such a clause lists all the memory locations that *may* be modified by a function (or a statement), possible solutions would consist in either collecting all the memory locations whose values change and comparing this set to the list from the clause afterwards, or monitoring all the memory locations that do *not* appear in the clause (possibly many), in order to verify that they are never modified in the considered piece of code. Both solutions are quite expensive. Therefore, implementing a sound-yet-fast static analysis is certainly necessary to reduce what memory locations are collected and compared (first solution) or monitored (second solution). Another current issue is the support of the `\at` construct. Its current support in the E-ACSL tool is efficient but partial. For `\at(t, L)`, it consists in storing in an extra variable the value of term `t` at the program point corresponding to label `L`. This is efficient, but it does not work as soon as the term `t` contains variables with no values at label `L`. The typical example is `\forall integer i; 0 <= i < 10 ==> \old(t[i]) == 0` for some array `t` of length 10: if used in a postcondition, the binder `i` has no meaning in the precondition in which the `\old` construct must be evaluated. Workarounds usually consist in cloning large parts of (if not all) the relevant datastructures [Kos10, PBJ+14]. Here it would soundly clone the whole array. However, this approach is expensive in the general case. Mixing up the current E-ACSL implementation with such a workaround could be a potential solution to an efficient-yet-sound runtime checking of the `\at` construct. Many of these unsupported (or partially supported) features should be implemented by Fonenantsoa MAURICA who started in February 2018 a 2-year postdoc on E-ACSL under my supervision.

Another research direction consists in optimizing the generated code. Indeed, a large amount of time has already been spent to optimize the E-ACSL RTL thanks to the postdoc of Kostyantyn VOROBYOV (see Section 3.3.4), while I have already implemented a type system to prevent generating Gmp code (Section 3.3.3), as well as a static analysis to prevent monitoring useless memory locations (Section 3.3.4). However, much more could be done in that direction. First, as already pointed out, this static analysis is currently not perfect and needs to be improved. Second, the E-ACSL compilation scheme is currently rather naive for most constructs. In particular, it does nothing special to prevent generating plenty of C variables, or to

remove code redundancy. It leads to extra time overhead and memory consumption. The most representative example is perhaps the code generated to prevent undefined behavior through plug-in Rte (see Section 3.3.2): currently, if the original code uses a left-value $*^k p$ (that is, p dereferenced k times), the generated code performs a quadratic number of verifications with respect to k, while a linear number of verifications is enough (to verify the proper access to each level of pointer). Kostyantyn Vorobyov did an experiment: he manually removed all the useless verifications on a use case of a few thousands lines of code. It led to a runtime overhead divided by about two (that is, instead of a 19× overhead, the overhead was below 10×). It is only one possible optimization among several. In particular, I think that E-ACSL could adapt to its own context several standard (and sound) compiler optimizations, such as constant propagation, common sub-expression elimination, branch optimization, and code motion [App04]. Some of these optimizations should also be implemented by Fonenantsoa Maurica within the next two years.

Today, the most promising industrial perspective of E-ACSL is to use it as an automatic debugger, such as AddressSanitizer or Valgrind. For this purpose, it currently relies on the E-ACSL specification language and automatic annotation generation through plug-in Rte. However, the absence of some common C programming errors cannot be (easily) specified, such as absence of format string vulnerabilities when using `printf`-like functions. E-ACSL was already extended by Kostyantyn Vorobyov to (optionally) detect this kind of errors, but it is not complete for all functions (*e.g.* `scanf`). Perhaps, a similar approach could also be adopted for other kinds of errors that are traditionally out-of-scope of runtime assertion checkers, such as *e.g.* SQL injection.

Verifying at runtime complex contracts of low-level functions (*e.g.* `memcpy`) is also not very efficient, in particular when they are heavily used by the monitored program. A more efficient approach consists in providing built-ins for such functions. A built-in is a C function that behaves like the function that it monitors, but also checks its precondition efficiently, while assuming that the low-level system function is correct and so no postconditions are verified [48]. This approach has been recently adopted by E-ACSL for a few libc functions, but it is still possible to extend it to other useful low-level functions. It is worth noting that using built-ins is not E-ACSL-specific. For instance, a LSL closed source extension of plug-in Eva also implements native OCaml built-ins for a few libc functions in order to analyze them faster and more precisely: they are almost mandatory for large industrial use cases to get practical results.

Some current E-ACSL limitations prevent the tool from being used in some contexts. First, the current implementation of its memory model prevents analyzing multi-threaded programs because concurrent accesses to the shadow model may

48. It is still possible to prove them by other means, such as plug-in Wp.

lead to unsoundness. It must be fixed, but it requires major modifications of this implementation. Second, E-ACSL, as any runtime verification tool, requires the whole program to be executed. It may be an issue in some operational contexts, for instance if some third-party libraries are not (yet) available. A potential solution to this problem is function synthesis [KMPS10]. Plug-in Synthesis (Section 2.6, page 28), developed by Michele Alberti during his postdoc under my supervision, has been implemented with this use case in mind, but it is still a research prototype and needs to be improved for practical use.

Last but not least, it is also possible to improve the specification language in some directions. First, as already mentioned, a C++ extension will be necessary in order to tackle C++ programs. Next, it is based on ACSL which is updated regularly, so E-ACSL should be updated accordingly to remain compatible. Finally, research progress may help to add ACSL constructs to the E-ACSL language because new techniques make them monitorable, such as axiomatic definitions of logic functions and predicates; as explained in Section 3.2.12, the work of Pierre-Nicolas Tollitte for compiling Coq inductive definitions to recursive functions [TDD12, Tol13] led to preliminary work in E-ACSL [Kha17] to compile a subset of them to C recursive functions. Yet, this work needs to be continued.

3.5.3 Theoretical Foundation

Up to now, the E-ACSL perspectives were mostly engineering tasks, even if interesting research questions arise for some of them. Also, most of the effort already spent on E-ACSL since its birth were of this nature in order to get a usable tool as fast as possible. However, more theoretical research could — and should — be performed.

First, while the type system has been proved sound [JKS15b], and the soundness proof of the memory model analysis is also almost complete thanks to the internship of Dara Ly [Ly17, LKLS18], no soundness proof — and, even worse, no formalization — of the instrumentation engine currently exists. That is pretty annoying for a tool that aims to check safety and security properties of critical systems. Such a formalization and soundness proof is precisely the topic of Dara Ly's PhD started in November 2017 under my supervision together with Nikolai Kosmatov at LSL and Frédéric Loulergue at Northern Arizona University. However, this effort certainly requires more than one PhD to be complete.

Also, there were already several formalization efforts of subsets of C and ACSL, in order to prove different Frama-C-based analyses in a proof assistant, either in Coq [Bla16, BLK17, LKG18], or in Isabelle [Bar16]. Each of them defines its own subset with slightly different semantics corresponding to its specific need. It is also worth noting the impressive effort already spent on the CompCert certified compiler [Ler09] and the Verasco certified abstract interpreter [JLB+15, Jou16]. I

think that it is maybe the right moment to not reinvent the wheel one more time. Consequently, I would like to develop a formal framework based on the CompCert semantics [KLW14] and extended with ACSL, in which anyone could define a subset of interest in order to formally study his/her favorite analysis. That is the sense of a French ANR project that I am currently submitting together with several French academic partners (including the Verasco team) and Mitsubishi Electric: building a certified code analyzer framework including at least an abstract interpreter, a program prover and a runtime verifier, all of them being based on CompCert's semantics.

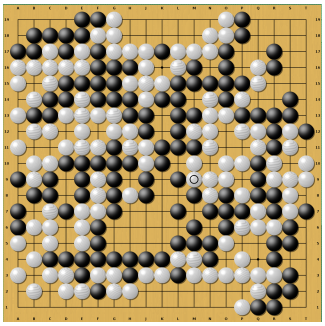
I hope that this effort will lead in particular to a better understanding of E-ACSL and, notably, what property is exactly checked at runtime. First, when running Rte before E-ACSL to generate annotations, it would be worth studying which undefined behaviors are really caught and, most importantly, which ones are not (yet) covered. Second, memory safety is a quite complex property [dAHP17] and it is not yet clear what safety property is precisely checked by the E-ACSL memory model when verifying a memory property such as a `\valid` predicate (at least, it depends whether it tries to detect temporal memory properties [VKSJ17]). This formalization effort could also help to formally compare E-ACSL to other formal tools, such as RV-Match [GHSR16] based on $\mathbb{K}$ [ER12, Ell12].

Beyond what properties the E-ACSL memory model ensures, its C implementation is also quite complicated and would be worth being formally verified (at least for its core part). However, it relies on low-level C programming, such as pointer arithmetic and bitwise operations: its formal proof is very challenging and, for sure, cannot be done with automatic theorem provers only. Nevertheless, several recent works show that verifying complicated low-level pieces of C code is doable [BKLL15, MDK16, RHMM17, BLK18]. Perhaps, the time has come to face this challenge.

I am also interested in expressing E-ACSL in a standard runtime verification formalism. In this context, E-ACSL would become a monitor generator checking properties over events and traces, instead of monitoring properties about program states. However, while the relationship between events and traces on one hand and program states on the other hand seems rather intuitive (see Section 3.1.4), the underlying logic able to express the properties of the E-ACSL specification language remains unclear to my eyes. For instance, despite its merits, I am not yet convinced that the Adrian Francalanza *et al*'s praiseworthy foundational theory of monitors [FAA+17, AAFI18] fits our need.

4
Conclusion

Yose (Endgame)



End of the Game of the Century between Go Seigen and Honinbo Shusai (1933/10/16 – 1934/1/29).

It is now time to conclude this journey into the Frama-C and E-ACSL worlds. Its objective was to present a large overview of both tools, while highlighting my personal contributions.

I joined the Frama-C team in 2006 when Frama-C was still a baby framework. I spent most of my time up to 2011 improving its kernel. I developed several libraries that are now at the heart of this framework. This development raises some nice programming challenges (*e.g.* dynamic typing in OCaml) and research questions (*e.g.* how to soundly combine analysis results). Then I started the development of its runtime verification plug-in E-ACSL. It led to several very interesting research questions about efficient and sound online monitoring of state properties, in particular related to memory safety.

Frama-C is now used worldwide by academia and by industrial partners, in particular in (but not limited to) the avionic domain and more generally transportation, as well as the defense and the nuclear domains. Its development also led to the creation of TrustInSoft, which commercializes two custom versions of Frama-C, namely TIS-Analyzer and TIS-Interpreter.

E-ACSL is not yet as mature as Frama-C and its most advanced plug-ins such as Eva and Wp. However, the large benchmarking effort and the first industrial use cases recently completed are promising: they show that E-ACSL is now comparable to the best commercial tools of the field, such as AddressSanitizer or Valgrind.

Yet many research actions are still to be pursued. For Frama-C, its kernel is still evolving. Analyzer collaborations, its most important feature, could be improved, while new usages are emerging such as integrating new Eva domains, or creating collaborations between Frama-C as a whole and other tools, in order to build bridges between C code and model, or between C code and lower-level code such as assembly or binary code. New domains of application constantly arise, such as robotics, machine learning or quantum computing. Each time, Frama-C must be adapted to new domain specific needs. Finally, targeting new programming languages, such as the ongoing effort on C++ is also a challenge. It may eventually lead to the development of a new framework, powerful enough to natively support most mainstream programming languages.

The engineering effort on E-ACSL is still continuing in order to improve its efficiency and its expressiveness, while adding new usages. It includes adding constructs from its specification language that are not yet supported. The efficient implementation of some of them is actually a challenge. For instance, some of them may require developing dedicated static analyses. Also, optimizing the generated code is required. It would notably require adapting existing compilation techniques to the E-ACSL setting. The E-ACSL memory model contains a few unused bits for each monitored block. It would be worth using them for specific tasks such as taint analysis or machine learning. In addition, using E-ACSL in some industrial settings is not possible because it heavily modifies the program environment. Such a drawback is inherent to inline monitoring tools such as E-ACSL. For these contexts, developing an outline version of E-ACSL would be worthwhile.

I am also interested in exploring the theoretical foundation of E-ACSL. This includes formalizing and proving its instrumentation engine and its memory model, as well as defining precisely the verified properties. I would like to include this effort in a bigger one, in order to develop a Coq framework of certified analyzers including Verasco and based on CompCert's semantics. Besides, I would like to study how to express E-ACSL in a standard runtime verification formalism based on traces and events.

Some of the above-mentioned perspectives are already being explored. In particular, Dara Ly started a PhD about formalizing E-ACSL last November, while Fonenantsoa Maurica is just starting a 2-year postdoc for improving expressiveness and efficiency of E-ACSL.

Beyond these scientific considerations, funding is nowadays of the utmost importance for research in general and applied research in particular: without them, no significant effort can be put on a project. Today, Frama-C is funded by many

academic and industrial projects. Renewing them is a constant effort. E-ACSL, on its side, is much smaller and does not require as many sources of funding. It is currently funded by the French ANR project *AnaStaSec*, the French PIA project *S3P*, the European H2020 project *Vessedia*, and our joint lab with Thales. New academic and industrial research proposals that include R&D actions on E-ACSL are currently being submitted. Together with several industrial interests, it makes me confident in the future of E-ACSL.

Bibliography

This bibliography contains 262 inputs. Figure 1 shows their distribution per publication year.

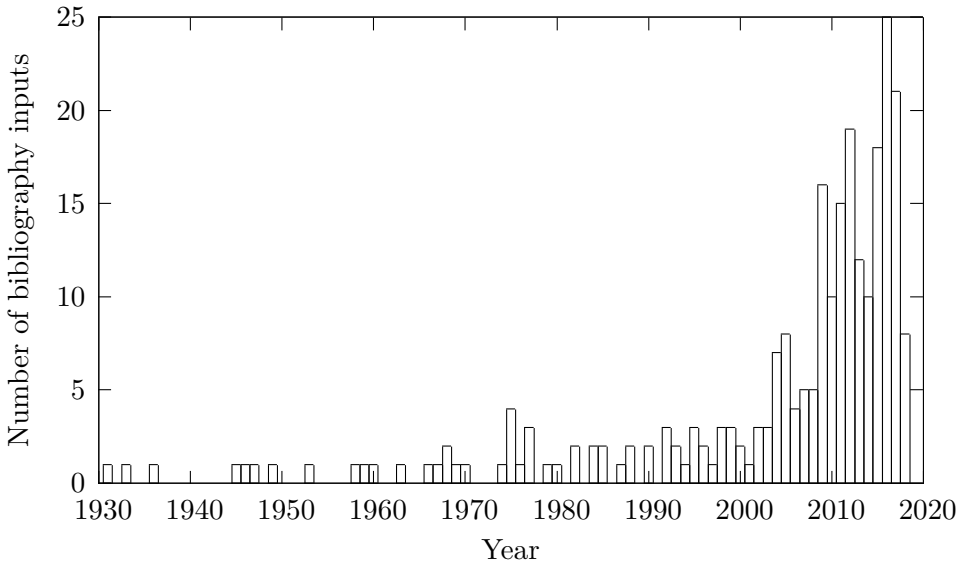


Figure 1: Number of bibliography inputs per publication year.

[AAFI18] Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir.
A framework for parametrized monitorability.
In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'18)*, April 2018.

[AARG12] Nicoals Ayache, Roberto Amadio, and Yann Régis-Gianas.
Certifying and reasoning on cost annotations in C programs.
In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS'12)*, August 2012.

[ABBD15] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir.
Verifiable side-channel security of cryptographic implementations: constant-time mee-cbc.
IACR Cryptology ePrint Archive, December 2015.

[Abr96] Jean-Raymond Abrial.
The B-Book, assigning programs to meaning.
Cambridge University Press, 1996.

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi.
Efficient detection of all pointer and array access errors.
In *Conference on Programming Language Design and Implementation (PLDI'94)*, June 1994.

[AEM04] Rajeev Alur, Kousha Etessami, and Parthasarathy Madhusudan.
A temporal logic of nested calls and returns.
In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, April 2004.

[AO16] Paul Ammann and Jeff Offutt.
Introduction to Software Testing.
Cambridge University Press, second edition, December 2016.

[App04] Andrew W. Appel.
Modern Compiler Implementation in ML.
Cambridge University Press, 2004.

[AS17] Michele Alberti and Julien Signoles.
Context Generation from Formal Specifications for C Analysis Tools.

In *International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'17)*, October 2017.

[Ass15] Mounir Assaf.
From Qualitative to Quantitative Program Analysis: Permissive Enforcement of Secure Information Flow.
PhD Thesis, Université Rennes 1, May 2015.

[ASTT13a] Mounir Assaf, Julien Signoles, Éric Totel, and Frédéric Tronel.
Moniteur hybride de flux d'information pour un langage supportant des pointeurs.
In *Conférence sur la Sécurité des Architectures Réseaux et des Systèmes d'Information (SARSSI'13).*, September 2013.
In French.

[ASTT13b] Mounir Assaf, Julien Signoles, Éric Totel, and Frédéric Tronel.
Program transformation for non-interference verification on programs with pointers.
In *International Information Security and Privacy Conference (SEC'13)*, July 2013.

[Bar16] Gergö Barany.
Hybrid Information Flow Analysis for Programs with Arrays.
In *Workshop on Verification and Program Transformation (VPT'16)*, April 2016.

[BBCD] Patrick Baudin, François Bobot, Loïc Correnson, and Zaynah Dargaye.
WP Plug-in Manual.
`http://frama-c.com/download/frama-c-wp-manual.pdf`.

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier.
MéTéor: A Successful Application of B in a Large Project.
In *Formal Methods in the Development of Computing Systems (FM'99)*, September 1999.

[BBY17] Sandrine Blazy, David Bühler, and Boris Yakobowski.
Structuring Abstract Interpreters through State and Value Abstractions.
In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'17)*, January 2017.

[BBYS18] Quentin Bouillaguet, François Bobot, Boris Yakobowski, and Mihaela Sighireanu.
A value analysis based memory model.
In *Journées Francophones des Langages Applicatifs (JFLA'18)*, January 2018.

[BC11] Richard Bonichon and Pascal Cuoq.
A mergeable interval map.
Studia Informatica Universalis, January 2011.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll.
An overview of JML tools and applications.
International Journal on Software Tools for Technology Transfer, 7(3), June 2005.

[BCDK14] Sébastien Bardin, Omar Chebaro, Mickaël Delahaye, and Nikolai Kosmatov.
An All-in-One Toolkit for Automated White-Box Testing.
In *International Conference on Tests and Proofs (TAP'14)*, July 2014.

[BCGP] Jochen Burghardt, Robert Clausecker, Jens Gerlach, and Hans Pohl.
ACSL by Example. Towards a Verified C Standard Library.
Technical report, Fraunhofer Fokus.

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani.
SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft.
In *Integrated Formal Methods (IFM'04)*, April 2004.

[BDES11] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu.
On inter-procedural analysis of programs with lists and data.
In *Conference on Programming Language Design and Implementation (PLDI'11)*, June 2011.

[BDH+18] Abderrahmane Brahmi, David Delmas, Mohamed Habib Essoussi, Famantanantsoa Randimbivololona, Abdellatif Atki, and Thomas Marie.

Formalise to automate: deployment of a safe and cost-efficient process for avionics software.
In *Embedded Real-Time Software and Systems (ERTS^2'18)*, January 2018.

[BDHTH+09] Bernard Botella, Mickaël Delahaye, Stéphane Hong-Tuan-Ha, Nikolai Kosmatov, Patricia Mouy, Muriel Roger, and Nicky Williams.
Automating structural testing of C programs: Experience with PathCrawler.
In *International Workshop on Automation of Software Test (AST'09)*, pages 70–78, May 2009.

[Ben84] Jon Bentley.
Programming Pearls: Algorithm Design Techniques.
Communications of the ACM, 27(9), September 1984.

[BFB+17] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Felix Klaedtke, Klaus Havelund, Yogi Joshi, Reed Milewicz, Giles Reger, Grigore Roşu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang.
First International Competition on Runtime Verification. Rules, Benchmarks, Tools and Final Results of CRV 2014.
International Journal on Software Tools for Technology Transfer, pages 1–40, April 2017.

[BFL+11] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter.
Specification and Verification: The Spec# Experience.
Communications of the ACM, 54(6):81–91, June 2011.

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

[BGvN46] Arthur W. Burks, Herman H. Goldstine, and John von Neumann.
Preliminary discussion of the logical design of an electronic computing instrument.
Technical report, Institute for Advanced Study, 1946.

[BHKL10] Pascal Berthomé, Karinne Heydemann, Xavier Kauffmann-Tourkestansky, and Jean-François Lalande.
Attack model for verification of interval security properties for smart card C codes.
In *Workshop on Programming Languages and Analysis for Security (PLAS'10)*, June 2010.

[BKGP17] Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, and Virgile Prevosto.
RPP: automatic proof of relational properties by self-composition.
In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*, April 2017.

[BKLL15] Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre, and Frédéric Loulergue.
A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C.
In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS'15)*, June 2015.

[BKLL16] Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre, and Frédéric Loulergue.
Conc2Seq: A Frama-C Plugin for Verification of Parallel Compositions of C Programs.
In *International Working Conference on Source Code Analysis and Manipulation (SCAM'16)*, pages 67–72, 2016.

[BKT17] David Basin, Srđan Krstić, and Dmitriy Traytel.
Almost Event-Rate Independent Monitoring of Metric Dynamic Logic.
In *International Conference on Runtime Verification (RV'17)*, September 2017.

[BL09] Sandrine Blazy and Xavier Leroy.
Mechanized Semantics for the Clight Subset of the C Language.
Journal of Automated Reasoning, 43(3), 2009.

[Bla16] Allan Blanchard.
Aide à la vérification de programmes concurrents par transformation de code et de spécifications.
PhD Thesis, Université d'Orléans, December 2016.
In French.

[BLK17] Allan Blanchard, Frédéric Loulergue, and Nikolai Kosmatov.
From Concurrent Programs to Simulating Sequential Programs: Correctness of a Transformation.
In *Workshop on Verification and Program Transformation (VPT'17)*, April 2017.

[BLK18] Allan Blanchard, Frédéric Loulergue, and Nikolai Kosmatov.
Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C.
In *International Symposium on NASA Formal Methods (NFM'18)*, April 2018.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte.
The Spec# Programming System: An Overview.
In *International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, pages 49–69, March 2004.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart.
Runtime verification for ltl and tltl.
Transactions on Software Engineering and Methodology, 20(4), September 2011.

[BMR95] Alexander Borgida, John Mylopoulos, and Raymond Reiter.
On the frame problem in procedure specifications.
Transactions on Software Engineering, 21(10):785–798, October 1995.

[Bou12a] Jean-Louis Boulanger, editor.
Industrial Use of Formal Methods: Formal Verification.
Wiley-ISTE, June 2012.
320 pages.

[Bou12b] Jean-Louis Boulanger, editor.
Industrial Use of Formal Methods: from Model to the Code.
Wiley-ISTE, May 2012.
384 pages.

[Bou16] Quentin Bouillaguet.
Développement d'un outil de monitoring à base de simulation pour le code embarqué.
Master's thesis, CEA LIST, September 2016.
In French.

[BR00] Thomas Ball and Sriram K. Rajamani.
Bebop: A Symbolic Model Checker for Boolean Programs.
In *SPIN Workshop*, August 2000.

[BR16] Nataliia Bielova and Tamara Rezk.
A taxonomy of information flow monitors.
In *Principles of Security and Trust (POST'16)*, April 2016.

[Bro75] Frederick P. Brooks.
The Mythical Man-Month: Essays on Software Engineering.
Addison-Wesley, 1975.

[BS17] Gergö Barany and Julien Signoles.
Hybrid Information Flow Analysis for Real-World C Code.
In *International Conference on Tests and Proofs (TAP'17)*, July 2017.

[But16] Antonin Butant.
Mixing Proved and Unproved Systems Parts through Contracts to ensure Correct-by-Construction System Design.
Master's thesis, CEA LIST, September 2016.

[BY] Richard Bonichon and Boris Yakobowski.
Frama-C's metrics plug-in.
`http://frama-c.com/download/frama-c-metrics-manual.pdf`.

[BZ11] Derek Bruening and Qin Zhao.
Practical Memory Checking with Dr. Memory.
In *International Symposium on Code Generation and Optimization (CGO'11)*, April 2011.

[CC77] Patrick Cousot and Radhia Cousot.
Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.
In *Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival.
The ASTREÉ Analyzer.
In *European Symposium on Programming (ESOP'05)*, April 2005.

[CCK+] Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski.
Frama-C User Manual.
`http://frama-c.com/download/user-manual.pdf`.

[CCK+14] Omar Chebaro, Pascal Cuoq, Nikolai Kosmatov, Bruno Marre, Anne Pacalet, Nicky Williams, and Boris Yakobowski.
Behind the Scenes in SANTE: A Combination of Static and Dynamic Analyses.
Automated Software Engineering, 21(1), March 2014.

[CD08] 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 *Workshop on ML (ML'08)*, September 2008.

[CDDM12] Pascal Cuoq, David Delmas, Stéphane Duprat, and Victoria Moya Lamiel.
Fan-C, a Frama-C plug-in for data flow verification.
In *Embedded Real-Time Software and Systems (ERTS^2'12)*, February 2012.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies.
VCC: A Practical System for Verifying Concurrent C.
In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, pages 23–42, August 2009.

[CDS11] Pascal Cuoq, Damien Doligez, and Julien Signoles.
Lightweight typed customizable unmarshaling.
In *Workshop on ML (ML'11)*, September 2011.

[CF16] Christian Colombo and Yliès Falcone.
First International Summer School on Runtime Verification.
In *Runtime Verification (RV'16)*, September 2016.

[CFS07] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles.
Designing a generic graph library using ML functors.
In *Trends in Functional Programming (TFP'07)*, April 2007.

[Cha04] Patrice Chalin.
JML Support for Primitive Arbitrary Precision Numeric Types: Definition and Semantics.
Journal of Object Technology, 3(6):57–79, June 2004.

[Cha05] Patrice Chalin.
Logical foundations of program assertions: What do practitioners want?
In *International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 383–393, September 2005.

[Cha07] Patrice Chalin.
A sound assertion semantics for the dependable systems evolution verifying compiler.
In *International Conference on Software Engineering (ICSE'07)*, pages 23–33, May 2007.

[Cha09] Patrice Chalin.
Engineering a sound assertion semantics for the verifying compiler.
Transactions on Software Engineering, 36:275–287, September 2009.

[Che03] Yoonsik Cheon.
A runtime assertion checker for the Java Modeling Language.
PhD Thesis, Iowa State University, 2003.

[Che11] Omar Chebaro.
Classification de menaces d'erreurs par analyse statique, simplification syntaxique et test structurel de programmes.
PhD Thesis, University of Franche-Comté, December 2011.
In French.

[Chr13] Maria Christofi.
Preuves de sécurité outillées d'implémentations cryptographiques.
PhD Thesis, Université of Versailles-Saint Quentin en Yvelines, February 2013.
In French.

[Chu33] Alonzo Church.
A Set of Postulates for the Foundation of Logic, Part II.
Annals of Mathematics, 34(2):839–864, 1933.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski.

Frama-C: a Software Analysis Perspective.
In *International Conference on Software Engineering and Formal Methods (SEFM'12)*, October 2012.

[CKM12] Cyrille Comar, Johannes Kanig, and Yannick Moy.
Integrating formal program verification with testing.
In *Embedded Real-Time Software and Systems (ERTS^2'12)*, February 2012.

[CMP10] Dumitru Ceara, Laurent Mounier, and Marie-Laure Potet.
Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences.
In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW'10*, April 2010.

[CMW12] Maria Christakis, Peter Müller, and Valentin Wüstholz.
Collaborative verification and testing with explicit assumptions.
In *Formal Methods (FM'12)*, pages 132–146, August 2012.

[Cok11] David R. Cok.
OpenJML: JML for Java 7 by Extending OpenJDK.
In *International Symposium on NASA Formal Methods (NFM'11)*.
April 2011.

[Cor14] Loïc Correnson.
Qed. Computing What Remains to Be Proved.
In *International Symposium on NASA Formal Methods (NFM'14)*, May 2014.

[CR06] Lori A. Clarke and David S. Rosenblum.
A Historical Perspective on Runtime Assertion Checking in Software Development.
SIGSOFT Software Engineering Notes, 31(3), May 2006.

[CRH17] Pascal Cuoq and Raphaël Rieu-Helft.
Result graphs for an abstract interpretation-based static analyzer.
In *Journés Francophones des Langages Applicatifs (JFLA'17)*, January 2017.

[CS12] Loïc Correnson and Julien Signoles.
Combining Analyses for C Program Verification.
In *Formal Methods for Industrial Case Studies (FMICS'12)*, August 2012.

[CSB+09] Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto, and Armand Puccetti.
Experience report: OCaml for an industrial-strength static analysis framework.
In *International Conference of Functional Programming (ICFP'09)*, September 2009.

[CYL+] Pascal Cuoq, Boris Yakobowski, Matthieu Lemerre, André Maroneze, Valentin Perelle, and Virgile Prevosto.
Frama-C's value analysis plug-in.
`http://frama-c.com/download/frama-c-value-analysis.pdf`.

[DA06] Dinakar Dhurjati and Vikram Adve.
Backwards-compatible Array Bounds Checking for C with Very Low Overhead.
In *International Conference on Software Engineering (ICSE'06)*, May 2006.

[dAHP17] Arthur Azevedo de Amorim, Catalin Hritcu, and Benjamin C. Pierce.
The Meaning of Memory Safety.
Computing Research Repository (CoRR), abs/1705.07354, May 2017.

[Dav93] Clive Davidson.
The man who made computers personal.
New Scientist, (1878):32–35, June 1993.

[DDMLS10] David Delmas, Stéphane Duprat, Victoria Moya-Lamiel, and Julien Signoles.
Taster, a Frama-C plug-in to enforce Coding Standards.
In *Embedded Real-Time Software and Systems Congress (ERTSS'10)*, May 2010.

[Deu04] Alain Deutsch.
Static verification of dynamic properties.
Technical report, PolySpace, February 2004.
White Paper.

[DFM11] Claire Dross, Jean-Christophe Filliâtre, and Yannick Moy.
Correct Code Containing Containers.

In *International Conference on Tests and Proofs (TAP'11)*, June 2011.

[DGK+15] Arnaud Dieumegard, Pierre-Loïc Garoche, Temesghen Kahsai, Alice Taillar, and Xavier Thirioux.
Compilation of synchronous observers as code contracts.
In *Symposium on Applied Computing (SAC'15)*, April 2015.

[DGP+09] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine.
Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software.
In *Formal Methods for Industrial Critical Systems (FMICS'09)*, November 2009.

[Dij75] Edsger W. Dijkstra.
Guarded commands, nondeterminacy and formal derivation of programs.
Communications of the ACM, 18(8):453–457, 1975.

[DKS13] Mickaël Delahaye, Nikolaï Kosmatov, and Julien Signoles.
Common specification language for static and dynamic analysis of C programs.
In *Symposium on Applied Computing (SAC'13)*, March 2013.

[dOBP16] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto.
Polynomial invariants by linear algebra.
In *International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, November 2016.

[dOPB15] Steven de Oliveira, Virgile Prevosto, and Sébastien Bardin.
Au temps en emporte le C.
In *Journées Francophones des Langages Applicatifs (JFLA'15)*, January 2015.
In French.

[DTT09] Jonathan-Christofer Demay, Éric Totel, and Frédéric Tronel.
SIDAN: a tool dedicated to software instrumentation for detecting attacks on non-control-data.
In *International Conference on Risks and Security of Internet and Systems (CRiSIS'09)*, October 2009.

[EC80] E. Allen Emerson and Edmund M. Clarke.

Characterizing correctness properties of parallel programs using fixpoints.
Automata, Languages and Programming, 85/1980, 1980.

[EDGBO11] Ivan Enderlin, Frédéric Dadeau, Alain Giorgetti, and Abdallah Ben Othman.
Praspel: A Specification Language for Contract-Based Testing in PHP.
In *International Conference on Testing Software and Systems (ICTSS'11)*, pages 64–79, November 2011.

[Ell12] Chucky Ellison.
A Formal Semantics of C with Applications.
PhD Thesis, University of Illinois, July 2012.

[ER12] Chucky Ellison and Grigore Roşu.
An Executable Formal Semantics of C with Applications.
In *Symposium on Principles of Programming Languages (POPL'12)*, 2012.

[Ers58] Andreï P. Ershov.
On programming of arithmetic operations.
Communications of the ACM, 1(8):3–6, August 1958.

[FAA+17] Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir.
A foundation for runtime monitoring.
In *International Conference on Runtime Verification (RV'17)*, September 2017.

[FBCDW15] Antoine Ferlin, Philippe Bon, Simon Collart-Dutilleul, and Virginie Wiels.
Parallel verification of temporal properties using dynamic analysis.
In *International Conference on Industrial Engineering and System Management (IESM'15)*, October 2015.

[FBL10] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo.
Embedded Contract Languages.
In *Symposium on Applied Computing (SAC'10)*, March 2010.

[FGP16] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich.
The spirit of ghost code.

Formal Methods in System Design, 48(3):152–174, June 2016.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm.
Reliable and precise WCET determination for a real-life processor.
In *Embedded Software (EMSOFT'01)*, October 2001.

[FHR13] Yliès Falcone, Klaus Havelund, and Giles Reger.
A tutorial on runtime verification.
In *Engineering Dependable Software Systems*, pages 141–175. IOS Press, June 2013.

[Fil11] Jean-Christophe Filliâtre.
Deductive Program Verification.
Habilitation Thesis, Université Paris-Sud, December 2011.

[FL10] Manuel Fähndrich and Francesco Logozzo.
Static contract checking with abstract interpretation.
In *Formal Verification of Object-Oriented Software (FoVeOOS'10)*, June 2010.

[Flo67] Robert W. Floyd.
Assigning meanings to programs.
In *Mathematical Aspects of Computer Science*, volume 19, 1967.

[FM07] Jean-Christophe Filliâtre and Claude Marché.
The Why/Krakatoa/Caduceus platform for deductive program verification.
In *Computer Aided Verification (CAV'07)*, July 2007.

[FNRT15] Yliès Falcone, Dejan Nickovic, Giles Reger, and Daniel Thoma.
Second International Competition on Runtime Verification CRV 2015.
September 2015.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich.
Why3 — Where Programs Meet Provers.
In *European Symposium on Programming (ESOP'13)*, March 2013.

[FSL+15] Daniel Fava, Julien Signoles, Matthieu Lemerre, Martin Schäf, and Ashish Tiwari.

Gamifying Program Analysis.
In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'15)*, pages 591–605, November 2015.

[GGJK08] Alain Giorgetti, Julien Groslambert, Jacques Julliand, and Olga Kouchnarenko.
Verification of class liveness properties with Java Modeling Language.
IET Software, 2(6), December 2008.

[GH93] Yuri Gurevich and James K. Huggins.
The Semantics of the C Programming Language.
In *Workshop on Computer Science Logic (CSL'92)*, 1993.

[Ghe07] Sanjay Ghemawat.
TCMalloc: Thread-Caching Malloc, February 2007.
`http://pages.cs.wisc.edu/~danb/google-perftools-0.98/tcmalloc.html`.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns: Elements of Reusable Object-oriented Software.
Addison-Wesley, 1995.

[GHSR16] Dwight Guth, Chris Hathhorn, Manasvi Saxena, and Grigore Roşu.
RV-Match: Practical Semantics-Based Program Analysis.
In *Computer Aided Verification (CAV'16)*, July 2016.

[GM82] Joseph A Goguen and José Meseguer.
Security Policies and Security Models.
Symposium on Security and Privacy, April 1982.

[GNP+15] Rigel Gjomemo, Kedar S. Namjoshi, Phu H. Phung, V. N. Venkatakrishnan, and Lenore D. Zuck.
From Verification to Optimizations.
In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'15)*, January 2015.

[Goo60] Richard Goodman, editor.
Annual Review in Automatic Programming, volume 1.
Pergamon Press, 1960.

[Got74] Eiichi Goto.
Monocopy and associative algorithms in extended lisp.
Technical Report Technical Report TR-74-03, University of Toyko, April 1974.

[GPP+16] Daniel Genkin, Lev Packmanov, Itamar Pipman, Adi Shamir, and Eran Tromer.
Physical key extraction attacks on PCs.
Communications of the ACM, 59(6), June 2016.

[GS95] David Gries and Fred B. Schneider.
Avoiding the undefined by underspecification.
In *Computer Science Today: Recent Trends and Developments*, pages 366–373. Springer, 1995.

[GS09] Julien Groslambert and Nicolas Stouls.
Vérification de propriétés LTL sur des programmes C par génération d'annotations.
In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'09)*, January 2009.
In French.

[GV08] Orna Grumberg and Helmut Veith, editors.
25 Years of Model Checking: History, Achievements, Perspectives.
Springer Verlag, 2008.

[GvN47] Hermann H. Goldstine and John von Neumann.
Planning and Coding of Problems for an Electronic Computing Instrument.
Technical Report Part II, volume 1–3, Institute for Advanced Study, 1947.

[Gö31] Kurt Gödel.
Über formal Unentscheidbare sätze der Principia Mathematica und verwandter Systeme I.
Monatshefte für Mathematik und Physik, 38:173–198, 1931.
English translation available in *From Frege to Gödel* [vH76].

[Her13] Paolo Herms.
Certification of a Tool Chain for Deductive Program Verification.
PhD Thesis, Université Paris-Sud, January 2013.

[HG05] Klaus Havelund and Allen Goldberg.
Verify Your Runs.

In *Verified Software: Theories, Tools, Experiments (VSTTE'05)*, October 2005.

[HJ92] Reed Hastings and Bob Joyce.
Purify: Fast detection of memory leaks and access errors.
In *Winter 1992 USENIX Conference*, January 1992.

[HLL+12] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson.
Behavioral interface specification languages.
Computing Surveys, 44(3):16:1–16:58, June 2012.

[HMM12] Paolo Herms, Claude Marché, and Benjamin Monate.
A certified multi-prover verification condition generator.
In *Verified Software, Theories, Tools and Experiments (VSTTE'12)*, January 2012.

[Hoa69] C. A. R. (Tony) Hoare.
An axiomatic basis for computer programming.
Communications of the ACM, 12(10), October 1969.

[HS] Philippe Herrmann and Julien Signoles.
Annotation generation: Frama-C's RTE plug-in.
`http://frama-c.com/download/frama-c-rte-manual.pdf`.

[HS12] Daniel Hedin and Andrei Sabelfeld.
A Perspective on Information-Flow Control.
In *Software Safety and Security*, volume 33 of *NATO Science for Peace and Security Series – D: Information and Communication Security*, pages 319–347. IOS Press, 2012.

[HTK00] David Harel, Jerzy Tiuryn, and Dexter Kozen.
Dynamic Logic.
MIT Press, September 2000.

[Hua79] Jung-Chang Huang.
Detection of data flow anomaly through program instrumentation.
Transactions on Software Engineering, SE-5(3):226–236, May 1979.

[HvdKGB16] István Haller, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos.
METAlloc: efficient and comprehensive metadata management for software security hardening.

In *European Workshop on Systems Security (EUROSEC'16)*, April 2016.

[Jak14] Arvid Jakobsson.
Optimized Support of Memory-Related Annotations for Runtime Assertion Checking with Frama-C.
Master's thesis, CEA LIST, August 2014.

[JF88] Ralph E. Johnson and Brian Foote.
Designing reusable classes.
Journal of Object-Oriented Programming, June 1988.

[JK97] Richard W. M. Jones and Paul H. J. Kelly.
Backwards-compatible bounds checking for arrays and pointers in C programs.
In *International Workshop on Automatic Debugging (AADEBUG'97)*, September 1997.

[JKS15a] Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles.
Fast as a Shadow, Expressive as a Tree: Hybrid Memory Monitoring for C.
In *Symposium On Applied Computing (SAC'15)*, April 2015.

[JKS15b] Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles.
Rester statique pour devenir plus rapide, plus précis et plus mince.
In *Journées Francophones des Langages Applicatifs (JFLA'15)*, Le Val d'Ajol, France, January 2015.
In French.

[JKS16] Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles.
Fast as a Shadow, Expressive as a Tree: Optimized Memory Monitoring for C.
Science of Computer Programming, pages 226–246, October 2016.
Extended version of [JKS15a].

[JLB+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie.
A Formally-Verified C Static Analyzer.
In *Symposium on Principles of Programming Languages (POPL'15)*, January 2015.

[Jon03] Cliff B. Jones.
The early search for tractable ways of reasonning about programs.
Annals of the History of Computing, 25(2):26–49, April 2003.

[Jou16] Jacques-Henri Jourdan.
Verasco: a Formally Verified C Static Analyzer.
PhD Thesis, Université Paris Diderot, May 2016.

[KCC+14] Johannes Kanig, Rod Chapman, Cyrille Comar, Jerôme Guitton, Yannick Moy, and Emyr Rees.
Explicit assumptions – a prenup for marrying static and dynamic program verification.
In *International Conference on Tests and Proofs (TAP'14)*, pages 142–157, July 2014.

[Kha17] Karam Younes Kharraz.
Synthesis of C Recursive Functions from E-ACSL Axiomatic Definitions.
Master's thesis, CEA LIST, September 2017.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski.
Frama-C: A Software Analysis Perspective.
Formal Aspects of Computing, January 2015.
Extended version of [CKK+12].

[KLW14] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk.
Formal C semantics: Compcert and the C standard.
In *International Conference on Interactive Theorem Proving (ITP'14)*, July 2014.

[KMPS10] Viktor Kunčak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter.
Complete functional synthesis.
In *Conference on Programming Language Design and Implementation (PLDI'10)*, June 2010.

[KMSM16] Nikolai Kosmatov, Claude Marché, Julien Signoles, and Yannick Moy.
Static vs Dynamic Verification in Why3, Frama-C and SPARK 2014.
In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'16)*, October 2016.

[Knu70] Donald E. Knuth.
Von Neumann's First Computer Program.
ACM Computing Surveys, 2(4), December 1970.

[Kos10] Piotr Kosiuczenko.
An Abstract Machine for the Old Value Retrieval.
In *International Conference on Mathematics of Program Construction (MPC'10)*, June 2010.

[KPS13a] Nikolaï Kosmatov, Guillaume Petiot, and Julien Signoles.
An optimized memory monitoring for runtime assertion checking of C programs.
In *International Conference on Runtime Verification (RV'13)*, September 2013.

[KPS13b] Nikolaï Kosmatov, Virgile Prevosto, and Julien Signoles.
A lesson on proof of programs with Frama-C.
In *International Conference on Tests and Proofs (TAP'13)*, June 2013.
Tutorial Paper.

[Kri63] Saul Kripke.
Semantical Considerations on Modal Logic.
Acta Philosophica Fennica, 16, 1963.

[KS13] Nikolaï Kosmatov and Julien Signoles.
A lesson on runtime assertion checking with Frama-C.
In *International Conference on Runtime Verification (RV'13)*, September 2013.

[KS16] Nikolai Kosmatov and Julien Signoles.
Frama-C, a Collaborative Framework for C Code Verification. Tutorial Synopsis.
In *International Conference on Runtime Verification (RV'16)*, September 2016.

[KS17] Yunus Kiliç and Hasan Sözer.
Generating runtime verification specifications based on static code analysis alerts.
In *Symposium on Applied Computing (SAC'17)*, April 2017.

[KSP+14] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song.
Code-pointer integrity.
In *Conference on Operating Systems Design and Implementation (OSDI'14)*, pages 147–163, October 2014.

[KWB+12] Nikolai Kosmatov, Nicky Williams, Bernard Botella, Muriel Roger, and Omar Chebaro.
A lesson on structural testing with PathCrawler-online.com.
In *International Conference on Tests and Proofs (TAP'12)*, May 2012.

[LB08] Xavier Leroy and Sandrine Blazy.
Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations.
Journal of Automated Reasoning, 41(1):1–31, July 2008.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby.
JML: A Notation for Detailed Design, chapter 12, pages 175–188.
Springer, October 1999.

[LCH+06] Glenn R. Luecke, James Coyle, Jim Hoekstra, Marina Kraeva, Ying Li, Olga Taborskaia, and Yanmei Wang.
A survey of systems for detecting serial run-time errors.
Concurrency and Computation: Practice and Experience, 18(15):1885–1907, February 2006.

[LCSW18] Viet Hoang Le, Loïc Correnson, Julien Signoles, and Virginie Wiels.
Verification coverage for combining test and proof.
In *International Conference on Tests and Proofs (TAP'18)*, June 2018.
To appear.

[Leh11] Hermann Lehner.
A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking.
PhD Thesis, ETH Zurich, 2011.

[Ler09] Xavier Leroy.
Formal verification of a realistic compiler.
Communications of the ACM, 52(7):107–115, 2009.

[LKG18] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall.
Cut branches before looking for bugs: certifiably sound verification on relaxed slices.
Formal Aspects of Computing, 30(1):107–131, January 2018.

[LKLS18] Dara Ly, Nikolai Kosmatov, Frédéric Loulergue, and Julien Signoles.
Soundness of a dataflow analysis for memory monitoring.
2018.
Submitted.

[LM09] K. Rustan M. Leino and Ronald Middelkoop.
Proving Consistency of Pure Methods and Model Fields.
In *Fundamental Approaches to Software Engineering (FASE'09)*, pages 231–245, March 2009.

[LS09a] Dirk Leinenbach and Thomas Santen.
Verifying the Microsoft Hyper-V Hypervisor with VCC.
In *Formal Methods (FM'09)*, November 2009.

[LS09b] Martin Leucker and Christian Schallhart.
A brief account of runtime verification.
Journal of Logic and Algebraic Programming, 78(5):293 – 303, May 2009.

[Luc90] David C. Luckham.
Programming with Specifications - An Introduction to ANNA, A Language for Specifying Ada Programs.
Springer Verlag, 1990.

[LVH85] David C. Luckham and Friedrich W. Von Henke.
An Overview of Anna, a Specification Language for Ada.
IEEE Software, 2(2), March 1985.

[Ly17] Dara Ly.
Formalisation sémantique d'une analyse statique pour un générateur de code.
Master's thesis, CEA LIST, October 2017.
In French.

[LZL+14] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Serbanuta, and Grigore Roşu.
RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties.
In *International Conference on Runtime Verification (RV'14)*, September 2014.

[MDK16] Frédéric Mangano, Simon Duquennoy, and Nikolai Kosmatov.
Formal Verification of a Memory Allocation Module of Contiki with Frama-C: A Case Study.
In *International Conference on Risks and Security of Internet and Systems (CRiSIS'16)*, September 2016.

[Mey88] Bertrand Meyer.
Eiffel: A language and environment for software engineering.
Journal of Systems and Software, pages 199–246, June 1988.

[Mey92a] Bertrand Meyer.
Applying "Design by Contract".
Computer, 25(10), October 1992.

[Mey92b] Bertrand Meyer.
Eiffel: The Language.
Prentice-Hall, 1992.

[Mic68] Donald Michie.
Memo functions and machine learning.
Nature, 218:19–22, 1968.

[MJ84] F. Lockwood Morris and Cliff B. Jones.
An Early Program Proof by Alan Turing.
Annals of the History of Computing, 6, April 1984.
Reprint of [Tur49].

[MKC09] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud.
Introduction to Interval Analysis.
Society for Industrial and Applied Mathematics, 2009.

[MM] Claude Marché and Yannick Moy.
The Jessie Plug-in for Deductive Verification in Frama-C. Tutorial and Reference Manual.
`http://http://krakatoa.lri.fr/jessie.pdf`.

[Mor68] Donald R. Morrison.
PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric.
Journal of the ACM, 15(4):514–534, October 1968.

[Moy09] Yannick Moy.
Automatic Modular Static Safety Checking for C Programs.
PhD Thesis, Université Paris-Sud, January 2009.

[MR05] Laurent Mauborgne and Xavier Rival.
Trace partitioning in abstract interpretation based static analyzers.
In *European Symposium on Programming (ESOP'05)*, April 2005.

[MS08] Benjamin Monate and Julien Signoles.
Slicing for Security of Code.
In *International Conference on Trusted Computing and Trust in Information Technologies (TRUST'08)*, March 2008.

[MY59] R. E. Moore and C. T. Yang.
Interval analysis I.
Technical Document LMSD-285875, Lockheed Missiles and Space Division, September 1959.

[NJW+13] Samaneh Navabpour, Yogi Joshi, Wallace Wu, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister.
RiTHM: A Tool for Enabling Time-triggered Runtime Verification for C Programs.
In *Joint Meeting on Foundations of Software Engineering (FSE'13)*, August 2013.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer.
CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs.
In *International Conference on Compiler Construction (CC'02)*, 2002.

[Nor98] Michael Norrish.
C formalised in HOL.
PhD Thesis, University of Cambridge, 1998.

[NS07] Nicholas Nethercote and Julian Seward.
Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation.
In *Conference on Programming Language Design and Implementation (PLDI'07)*, June 2007.

[Oiw04] Yutaka Oiwa.
Implementation of a Fail-Safe ANSI C Compiler.
PhD Thesis, University of Tokyo, December 2004.

[Oiw09] Yutaka Oiwa.
Implementation of the memory-safe full ANSI-C compiler.
In *Conference on Programming Language Design and Implementation (PLDI'09)*, June 2009.

[Pap98] Nikolaos S. Papaspyrou.
Formal Semantics for the C Programming Language.
PhD Thesis, National Technical University of Athens, 1998.

[PBJ+14] Guillaume Petiot, Bernard Botella, Jacques Julliand, Nikolai Kosmatov, and Julien Signoles.
Instrumentation of annotated C programs for test generation.
In *International Conference on Source Code Analysis and Manipulation (SCAM'14)*, September 2014.

[Per05] Colin Percival.
Cache missing for fun and profit.
In *The Technical BSD Conference (BSDCan'05)*, May 2005.

[Pet12] Guillaume Petiot.
Étude des combinaisons d'analyse statique et d'analyse dynamique pour la validation de programmes.
Master's thesis, CEA LIST, September 2012.
In French.

[Pet15] Guillaume Petiot.
Contribution à la vérification de programmes C par combinaison de tests et de preuves.
PhD Thesis, University of Franche-Comté, Besançon, France, December 2015.
In French.

[Pie02] Benjamin C. Pierce.
Types and Programming Languages.
MIT Press, January 2002.

[PKB+16] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand.
Your proof fails? testing helps to find the reason.
In *International Conference on Tests and Proofs (TAP'16)*, July 2016.

[Pnu77] Amir Pnueli.
The temporal logic of programs.

In *Symposium on Foundations of Computer Science (FCS'77)*, November 1977.

[PS17] Dillon Pariente and Julien Signoles.
Static Analysis and Runtime Assertion Checking: Contribution to Security Counter-Measures.
In *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'17)*, June 2017.

[QS82] Jean-Pierre Queille and Joseph Sifakis.
Specification and verification of concurrent systems in CESAR.
In *International Symposium on Programming*, volume 137, 1982.

[Rey02] John C. Reynolds.
Separation logic: A logic for shared mutable data structures.
In *Symposium on Logic in Computer Science (LICS'02)*, pages 55–74, July 2002.

[RH16] Giles Reger and Klaus Havelund.
What Is a Trace? A Runtime Verification Perspective.
In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'16)*, October 2016.

[RH17] Giles Reger and Klaus Havelund, editors.
RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, volume 3 of *Kalpa Publications in Computing*, September 2017.

[Rhi03] Morten Rhiger.
A foundation for embedded languages.
Transactions on Programming Languages and Systems (TOPLAS'03), 25(3):291–315, 2003.

[RHMM17] Raphaël Rieu-Helft, Claude Marché, and Guillaume Melquiond.
How to Get an Efficient yet Verified Arbitrary-Precision Integer Library.
In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'17)*, January 2017.

[Ric53] Hugh G. Rice.
Classes of Recursively Enumerable Sets and Their Decision Problems.

Transactions of the American Mathematical Society, 74(2), 1953.

[RS10] Alejandro Russo and Andrei Sabelfeld.
Dynamic vs. Static Flow-Sensitive Security Analysis.
In *Computer Security Foundations Symposium (CSF'10)*, pages 186–199, July 2010.

[RSB+99] Famantanantsoa Randimbivololona, Jean Souyris, Patrick Baudin, Anne Pacalet, Jacques Raguideau, and Dominique Schoen.
Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach.
In *Formal Methods in the Development of Computing Systems (FM'99)*, September 1999.

[SAC+] Julien Signoles, Thibaut Antignac, Loïc Correnson, Matthieu Lemerre, and Virgile Prevosto.
Frama-C Plug-in Development Guide.
`http://frama-c.com/download/plugin-developer.pdf`.

[SB10] Matthew S. Simpson and Rajeev K. Barua.
MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime.
In *Working Conference on Source Code Analysis and Manipulation (SCAM'10)*, September 2010.

[SBPV12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov.
Addresssanitizer: A fast address sanity checker.
In *Annual Technical Conference (ATC'12)*, June 2012.

[SF75] Leon G. Stucki and Gary L. Foshee.
New assertion concepts for self-metric software validation.
In *International Conference on Reliable Software (ICRS'75)*, pages 59–71, April 1975.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis.

Mastering the game of Go with deep neural networks and tree search.
Nature, 529(7587):484–489, January 2016.

[Siga] Julien Signoles.
E-ACSL: Executable ANSI/ISO C Specification Language.
`http://frama-c.com/download/e-acsl/e-acsl.pdf`.

[Sigb] Julien Signoles.
E-ACSL. Implementation in Frama-C Plug-in E-ACSL.
`http://frama-c.com/download/e-acsl/e-acsl-implementation.pdf`.

[Sig06] Julien Signoles.
Extension de ML avec raffinement: syntaxe, sémantiques et système de types.
PhD Thesis, Université Paris-Sud, July 2006.
In French.

[Sig09] Julien Signoles.
Foncteurs impératifs et composés: la notion de projet dans Frama-C.
In *Journées Francophones des Langages Applicatifs (JFLA'09)*, Studia Informatica Universalis, September 2009.
In French.

[Sig11] Julien Signoles.
Une bibliothèque de typage dynamique en OCaml.
In *Journées Francophones des Langages Applicatifs (JFLA'11)*, Studia Informatica Universalis, 2011.
In French.

[Sig14] Julien Signoles.
Comment un chameau peut-il écrire un journal?
In *Journées Francophones des Langages Applicatifs (JFLA'14)*, January 2014.
In French.

[Sig15] Julien Signoles.
Software Architecture of Code Analysis Frameworks Matters: The Frama-C Example.
In *Formal Integrated Development Environment (F-IDE'15)*, June 2015.

[Sig17] Julien Signoles.
Online Runtime Verification Competitions: How To Possibly Deal With Their Issues. Position Paper.
In *International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES'17)*, September 2017.

[SKV17] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov.
E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. Tool Paper.
In *International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES'17)*, September 2017.

[SMM15] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu.
Test suites for benchmarks of static analysis tools.
In *International Symposium on Software Reliability Engineering Workshops (ISSREW'15)*, November 2015.

[SN05] Julian Seward and Nicholas Nethercote.
Using Valgrind to Detect Undefined Value Errors with Bit-precision.
In *Annual Technical Conference (ATC'05)*, April 2005.

[SP] Nicolas Stouls and Virgile Prevosto.
Aoraï plug-in tutorial.
`http://frama-c.com/download/frama-c-aorai-manual.pdf`.

[SP16] Subash Shankar and Gilbert Pajela.
A Tool Integrating Model Checking into a C Verification Toolset.
In *International SPIN Symposium on Model Checking of Software (SPIN'16)*, April 2016.

[SPIV11] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov.
Dynamic Race Detection with LLVM Compiler.
In *International Conference on Runtime Verification (RV'11)*, September 2011.

[SS15] Evgeniy Stepanov and Konstantin Serebryany.
MemorySanitizer: fast detector of uninitialized memory use in C++.

In *International Symposium on Code Generation and Optimization (CGO'15)*, February 2015.

[SSS16] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry.
Towards verified artificial intelligence.
Computing Research Repository (CoRR), abs/1606.08514, July 2016.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis.
Mastering the game of go without human knowledge.
Nature, 550, October 2017.

[ST85] Daniel D. Sleator and Robert E. Tarjan.
Self-adjusting binary search trees.
Journal of the ACM, 32(3):652–686, July 1985.

[Ste96] Bjarne Steensgaard.
Points-to analysis in almost linear time.
In *Symposium on Principles of Programming Languages (POPL'96)*, January 1996.

[SV] Julien Signoles and Kostyantyn Vorobyov.
E-ACSL User Manual.
`http://frama-c.com/download/e-acsl/e-acsl-manual.pdf`.

[SWDD09] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny.
Formal verification of avionics software products.
In *Formal Methods (FM'09)*, November 2009.

[Szp90] Wojciech Szpankowski.
Patricia tries again revisited.
Journal of the ACM, 37(4):691–711, October 1990.

[Sö15] Hasan Sözer.
Integrated static code analysis and runtime verification.
Software: Practice and Experience, 45(10), October 2015.

[TDD12] Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois.

Producing certified functional code from inductive specifications.
In *International Conference on Certified Programs and Proofs (CPP'12))*, pages 76–91, December 2012.

[TFNM11] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer.
Usable verification of object-oriented programs by combining static and dynamic techniques.
In *International Conference on Software Engineering and Formal Methods (SEFM'11)*, November 2011.

[TO98] Andrew P. Tolmach and Dino Oliva.
From ML to Ada: Strongly-Typed Language Interoperability via Source Translation.
Journal of Functional Programming, 8(4), 1998.

[Tol13] Pierre-Nicolas Tollitte.
Extraction de code fonctionnel certifié à partir de spécifications inductives.
PhD Thesis, Conservatoire National des Arts et Métiers, December 2013.
In French.

[Tur36] Alan M. Turing.
On Computable Numbers, with an Application to the Entscheidungsproblem.
Proceedings of the London Mathematical Society, 42(2):230–265, 1936.
reprint in 1960 (see [Goo60], first appendix).

[Tur49] Alan M. Turing.
Checking a Large Routine.
In *Report on a Conference on High Speed Automatic Computation*, pages 67–69, June 1949.

[vdVdSCB12] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos.
Memory errors: The past, the present, and the future.
In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'12)*, September 2012.

[vH76] Jean van Heijenoort, editor.
From Frege to Gödel: A Source Book to Mathematical Logic, 1879–1931.

Harvard University Press, third edition, 1976.

[VKS16a] Kostyantyn Vorobyov, Nikolai Kosmatov, and Julien Signoles.
A computer-implemented method and a system for encoding a heap application memory state using shadow memory, September 2016.
European patent 16306236.7.

[VKS16b] Kostyantyn Vorobyov, Nikolai Kosmatov, and Julien Signoles.
A computer-implemented method and a system for encoding a stack application memory state using shadow memory, December 2016.
European patent 16306629.3.

[VKS18] Kostyantyn Vorobyov, Nikolai Kosmatov, and Julien Signoles.
Detection of Security Vulnerabilities in C Code using Runtime Verification: an Experience Report.
In *International Conference on Tests and Proofs (TAP'18)*, June 2018.
To appear.

[VKSJ17] Kostyantyn Vorobyov, Nikolai Kosmatov, Julien Signoles, and Arvid Jakobsson.
Runtime Detection of Temporal Memory Errors.
In *Internation Conference on Runtime Verification (RV'17)*, pages 294–311, September 2017.

[vN45] John von Neumann.
First draft of a report on the EDVAC.
Technical report, University of Pennsylvania, June 1945.

[VSK17] Kostyantyn Vorobyov, Julien Signoles, and Nikolai Kosmatov.
Shadow State Encoding for Efficient Monitoring of Block-level Properties.
In *International Symposium on Memory Management (ISMM'17)*, pages 47–58, June 2017.

[Win87] Jeannette M. Wing.
Writing larch interface language specifications.
Transactions on Programming Languages and Systems, 9(1):1–24, January 1987.

[WMMR05] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger.
PathCrawler: automatic generation of path tests by combining static and dynamic analysis.
In *European Dependable Computing Conference (EDCC'05)*, April 2005.

[XKZ10] Zhongxing Xu, Ted Kremenek, and Jian Zhang.
A Memory Model for Static Analysis of C Programs.
In *International Conference on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA'10)*, volume Part I, pages 535–548, October 2010.

[YC75] S. S. Yau and R. C. Cheung.
Design of self-checking software.
In *International Conference on Reliable Software (ICRS'75)*, pages 450–455, April 1975.

[YJ12] Jun Yuan and Rob Johnson.
CAWDOR: Compiler Assisted Worm Defense.
In *International Working Conference on Source Code Analysis and Manipulation (SCAM'12)*, September 2012.

Index

A
Callgraph Services

This appendix presents the so-called *service algorithm* of the Frama-C callgraph introduced in Section 2.6. Informally, a service groups together functions that contribute to a common functionality. In order to work properly, it is computed not only from the callgraph but also from a set of functions called *initial roots* that define a first set of (unfilled) services. In Frama-C, by default, these roots are the `main` function and its callees.

More precisely, the service algorithm computes a minimal cluster of the callgraph, that is one of its smallest decompositions into services. Let us now define these notions of services, (initial) roots and (minimal) clusters.

A.1 Definitions

We denote $|S|$, the cardinal of a set S. We also define a directed graph $G = (V, E)$ by its set V of vertices and its set E of edges. First we define the notion of a service S from a root vertex r as a set of vertices that only transitively depend on r.

Definition A.1 (Graph Service) *A service S of a directed graph $G = (V, E)$ with a root $r_S \in V$ is a subset of vertices V such that:*

$$r_S \in S \qquad \textit{(a root is in its own service)}$$
$$\forall s \in S \setminus \{r_S\}, \forall s' \in V, (s', s) \in E \implies s' \in S$$
$$\textit{(no root vertex of a service depends on outside vertices)}$$

The above definition captures what a service is. However it remains imprecise in organizing different services of a graph in a convenient way. That is what the notion of cluster does. It relies on an initial set of roots.

Definition A.2 (Graph Cluster) *A cluster $\mathcal{C}$ of a directed graph $G = (V, E)$ of initial roots $R \subseteq V$ is a set of services such that:*

$$\forall r \in R, \exists S \in \mathcal{C}, r = r_S \qquad \textit{(all initial roots are cluster roots)}$$
$$\forall (S, S') \in \mathcal{C}^2, S \neq S' \implies S \cap S' = \emptyset \qquad \textit{(cluster services are pairwise disjoint)}$$
$$\bigcup_{S \in \mathcal{C}} S = V \qquad \textit{(cluster services are complete)}$$

Minimizing the number of services (and so getting biggest ones) provides better results in practice. Here is its definition.

Definition A.3 (Minimal Graph Cluster) *A minimal cluster of a directed graph $G = (V, E)$ of initial roots $R \subseteq V$ is a cluster $\mathcal{C}$ such that its cardinality is the smallest among all the clusters of G:*

$$\forall \mathcal{C}' \textit{ cluster of } G, |\mathcal{C}| \leq |\mathcal{C}'|.$$

A.2 Algorithm

The service algorithm computes a minimal cluster of a directed graph $G = (V, E)$ of initial roots $R \subseteq V$. Figure 1 presents its pseudo-code. It proceeds in three steps. First, it initializes the *status* of each vertex to *unknown*. Then, it traverses the graph in topological ordering in order to visit the predecessors of a vertex v before v. This way, it is possible to discover roots of services and associate services to vertex through the status *service s* where s is the root of the service a vertex belongs to. Unfortunately, in case of cycles, it cannot always conclude with certainty because, when visiting a vertex v, all its predecessors have not necessarily been visited. Therefore it may compute a weaker status *maybe s* which means that either v belongs to the service of root s, or must be the root of a new service. The final step of the algorithm performs another topological traversal of the graph to precisely handle the status *maybe*.

If we assume that the time complexity of a topological iteration of G is

$$O(|V| \times \log |V|),$$

then the time complexity of our algorithm is

$$O(|V| \times \log |V| \times \max_{v \in V} |\operatorname{pred}(v, G))|.$$

The algorithm is yet unproved. As explained in Section 2.6, I consider the formal proof of its Frama-C implementation (based on OcamlGraph's datastructures and higher-order iterators [CFS07]) as a challenge for verification of higher-order not-purely-functional programs.

Let $G = (V, E)$ be a callgraph, $R \subseteq V$ be the sets of initial nodes and pred(v,g) be the set of predecessors of node v in G. In the pseudo-code below, comments are enclosed in *(* ... *)* like in OCaml.

(first step: initialize status to unknown *)*
for each $v \in V$, status(v) $\leftarrow$ unknown
(second step: discover roots and associate services to vertices *)*
for each $v \in V$ following a topological ordering
  if $v \in R \vee \forall v' \in \text{pred}(v, g), \text{status}(v') = \text{unknown}$
  then $v \leftarrow$ *service* v *(* new service of root* v *)*
  else if $\exists s \in V, \forall v' \in \text{pred}(v, g)$, status($v'$) = *service* s
  then $v \leftarrow$ *service* s *(* v *belongs to the same service than all its predecessors *)*
  else if $\exists s \in V, \forall v' \in \text{pred}(v, g), \text{status}(v') = (\text{service } s \vee \text{maybe } s \vee \text{unknown})$
  then $v \leftarrow \text{maybe } s$ *(* depend on the non-visited vertices *)*
  else if $v \leftarrow \text{service } v$ *(* new connected component *)*
  end if
done
(third step: resolve uncertainty from cycles *)*
for each $v \in V$ following a topological ordering
  if $\exists s \in V, \text{status}(v) = \text{maybe } s$
  then
    if $\forall v' \in \text{pred}(v, g), \text{status}(v') = (\text{service } s \vee \text{maybe } s)$
    (predecessors are either in the same service or in the same cycle *)*
    then $v \leftarrow \text{service } s$
    else $v \leftarrow \text{service } v$
    end if
  end if
done

Figure 1: Graph Service Algorithm.

B
List of Publications

This appendix contains the full list of my publications. Each publication is linked to the corresponding one in the main bibliography when it is cited in this document.

B.1 Patents

B2 Kostyantyn Vorobyov, Nikolai Kosmatov and Julien Signoles.
A computer-implemented method and a system for encoding a stack application memory state using shadow memory.
European patent 16306629.3 2016.
Bibliography entry [VKS16b].

B1 Kostyantyn Vorobyov, Nikolai Kosmatov and Julien Signoles.
A computer-implemented method and a system for encoding a heap application memory state using shadow memory.
European patent 16306236.7, 2016.
Bibliography entry [VKS16a].

B.2 Editor of Conference Proceedings

E2 Julien Signoles and Sylvie Boldo.
Actes des vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA).
Editors, 2017.

E1 Julien Signoles.
Actes des vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA).
Editor, 2016.

B.3 Peer-Reviewed International Journals

J3 Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Felix Klaedtke, Klaus Havelund, Yogi Joshi, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, Yi Zhang.
First International Competition on Runtime Verification. Rules, Benchmarks, Tools and Final Results of CRV 2014.
Software Tools for Technology Transfer (STTT), 2017.
Bibliography entry [BFB+17].

J2 Arvid Jakobsson, Nikolai Kosmatov and Julien Signoles.
Fast as a Shadow, Expressive as a Tree: Optimized Memory Monitoring for C.
Science of Computer Programming (SCP), 2016. Extended version of C13.
Bibliography entry [JKS16].

J1 Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles and Boris Yakobowski.
Frama-C, A Software Analysis Perspective.
Formal Aspects of Computing (FAC), 2015. Extended version of C5.
Bibliography entry [KKP+15].

B.4 Peer-Reviewed International Conferences

C24 Dara Ly, Nikolai Kosmatov, Frédéric Loulergue and Julien Signoles.
Soundness of a Dataflow Analysis for Memory Monitoring.
Submitted.
Bibliography entry [LKLS18].

C23 Kostyantyn Vorobyov, Nikolai Kosmatov and Julien Signoles.
Detection of Security Vulnerabilities in C Code using Runtime Verification: an Experience Report.
International Conference on Tests and Proofs (TAP), 2018, to appear.
Bibliography entry [VKS18].

C22 Viet Hoang Le, Loïc Correnson, Julien Signoles and Virginie Wiels.
Verification Coverage for Combining Test and Proof.
International Conference on Tests and Proofs (TAP), 2018, to appear.
Bibliography entry [LCSW18].

C21 Michele Alberti and Julien Signoles.
Context Generation from Formal Specifications for C Analysis Tools.
International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR), 2017, best paper award.
Bibliography entry [AS17].

C20 Kostyantyn Vorobyov, Nikolai Kosmatov, Julien Signoles and Arvid Jakobsson.
Runtime Detection of Temporal Memory Errors.
International Conference on Runtime Verification (RV), 2017.
Bibliography entry [VKSJ17].

C19 Gergö Barany and Julien Signoles.
Hybrid Information Flow Analysis for Real-World C Code.
International Conference on Tests and Proofs (TAP), 2017.
Bibliography entry [BS17].

C18 Kostyantyn Vorobyov, Julien Signoles and Nikolai Kosmatov.
Shadow State Encoding for Efficient Monitoring of Block-level Properties.
International Symposium on Memory Management (ISMM), 2017.
Bibliography entry [VSK17].

C17 Mounir Assaf, David A. Naumann, Julien Signoles, Éric Totel and Frédéric Tronel.
Hypercollecting Semantics and its Application to Static Analysis of Information Flow.
Principles of Programming Languages (POPL), 2017.

C16 Nikolai Kosmatov, Claude Marché, Yannick Moy and Julien Signoles.
Static vs Dynamic Verification in Why3, Frama-C and SPARK 2014.
International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), invited paper, 2016.
Bibliography entry [KMSM16].

C15 Nikolai Kosmatov and Julien Signoles.
Frama-C, a Collaborative Framework for C Code Verification. Tutorial Synopsis.
International Conference on Runtime Verification (RV), tutorial paper. 2016.
Bibliography entry [KS16].

C14 Daniel Fava, Matthieu Lemerre, Julien Signoles, Martin Schäf and Ashish Tiwari.
Gamifying Program Analysis.
International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), 2015.
Bibliography entry [FSL+15].

C13 Arvid Jakobsson, Nikolai Kosmatov and Julien Signoles.
Fast as a Shadow, Expressive as a Tree: Hybrid Memory Monitoring for C.
Symposium on Applied Computing (SAC), 2015.
Bibliography entry [JKS15a].

C12 Guillaume Petiot, Bernard Botella, Jacques Julliand, Nikolai Kosmatov and Julien Signoles.

Instrumentation of annotated C programs for test generation.
International Conference on Source Code Analysis and Manipulation (SCAM), 2014.
Bibliography entry [PBJ+14].

C11 Nikolai Kosmatov and Julien Signoles.
Runtime assertion checking and its combinations with static and dynamic analyses. Tutorial synopsis.
International Conference on Tests and Proofs (TAP), tutorial paper, 2014.

C10 Nikolai Kosmatov, Guillaume Petiot and Julien Signoles.
An optimized memory monitoring for runtime assertion checking of C programs.
International Conference on Runtime Verification (RV), 2013.
Bibliography entry [KPS13a].

C9 Nikolai Kosmatov and Julien Signoles.
A Lesson on Runtime Assertion Checking with Frama-C.
International Conference on Runtime Verification (RV), tutorial paper, 2013.
Bibliography entry [KS13].

C8 Mounir Assaf, Julien Signoles, Éric Totel and Frédéric Tronel.
Program transformation for non-interference verification on programs with pointers.
International Information Security and Privacy Conference (SEC), 2013. Mounir Assaf won the best student paper award of SEC'13 for this article.
Bibliography entry [ASTT13b].

C7 Nikolai Kosmatov, Virgile Prevosto and Julien Signoles.
A Lesson on Proof of Programs with Frama-C.
International Conference on Tests and Proofs (TAP), invited tutorial paper, 2013.
Bibliography entry [KPS13b]

C6 Mickaël Delahaye, Nikolai Kosmatov and Julien Signoles.
Common Specification Language for Static and Dynamic Analysis of C Programs.
Symposium on Applied Computing (SAC), 2013.
Bibliography entry [DKS13].

C5 Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles and Boris Yakobowski.
Frama-C, A Software Analysis Perspective.
International Conference on Software Engineering and Formal Methods (SEFM), 2012.
Bibliography entry [CKK+12].

C4 David Delmas, Stéphane Duprat, Victoria Moya Lamiel and Julien Signoles.
Taster, a Frama-C plug-in to enforce Coding Standards.
Embedded Real-Time Software and Systems Congress (ERTSS), 2010.
Bibliography entry [DDMLS10].

C3 Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto and Armand Puccetti.
Experience Report: OCaml for an industrial-strength static analysis framework.
International Conference on Functional Programming (ICFP), 2009.
Bibliography entry [CSB+09].

C2 Benjamin Monate and Julien Signoles.
Slicing for Security of Code.
TRUST, 2008.
Bibliography entry [MS08].

C1 Sylvain Conchon, Jean-Christophe Filliâtre and Julien Signoles.
Designing a generic graph library using ML functors.
Trends on Functional Programming (TFP), 2007.
Bibliography entry [CFS07].

B.5 Peer-Reviewed International Workshops

W6 Julien Signoles, Nikolai Kosmatov and Kostyantyn Vorobyov.
E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper).
International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES), 2017.
Bibliography entry [SKV17].

W5 Julien Signoles.
Online Runtime Verification Competitions: How To Possibly Deal With Their Issues (position paper).
International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES), 2017.
Bibliography entry [Sig17].

W4 Mounir Assaf, Julien Signoles, Éric Totel and Frédéric Tronel.
The Cardinal Abstraction for Quantitative Information Flow.
Workshop on Foundations of Computer Security (FCS), 2016.

W3 Julien Signoles.
Software Architecture of Code Analysis Frameworks Matters: The Frama-C

Example.
Workshop on Formal-IDE (F-IDE), 2015.
Bibliography entry [Sig15].

W2 Loïc Correnson and Julien Signoles.
Combining Analyses for C Program Verification.
Formal Methods for Industrial Case Studies (FMICS), 2012. Bibliography entry [CS12].

W1 Pascal Cuoq, Damien Doligez and Julien Signoles.
Lightweight Typed Customizable Unmarshaling.
ML Workshop, 2011.
Bibliography entry [CDS11]

B.6 Peer-Reviewed French Journals

N2 Julien Signoles.
Une bibliothèque de typage dynamique en OCaml.
Studia Informatica Universalis, 2011. In French.
Bibliography entry [Sig11].

N1 Julien Signoles.
Foncteurs impératifs et composés: la notion de projets dans Frama-C.
Studia Informatica Universalis, 2009. In French.
Bibliography entry [Sig09].

B.7 Peer-Reviewed French Conferences

F7 Dillon Pariente and Julien Signoles.
Static Analysis and Runtime Assertion Checking: Contribution to Security Counter-Measures.
Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC), 2017.
Bibliography entry [PS17].

F6 Arvid Jakobsson, Nikolai Kosmatov and Julien Signoles.
Rester statique pour devenir plus rapide, plus précis et plus mince.
Journées Francophones des Langages Applicatifs (JFLA), 2015. In French.
Bibliography entry [JKS15b].

F5 Julien Signoles.
Comment un chameau peut-il écrire un journal?
Journées Francophones des Langages Applicatifs (JFLA), 2014. In French.
Bibliography entry [Sig14].

F4 Mounir Assaf, Julien Signoles, Éric Totel and Frédéric Tronel.
Moniteur hybride de flux d'information pour un langage supportant des pointeurs.
Conférence sur la Sécurité des Architectures Réseaux et des Systèmes d'Information (SARSSI), 2013. In French. Mounir Assaf won the best student paper award of SARSSI'13 for this article.
Bibliography entry [ASTT13a].

F3 Julien Signoles.
Une approche fonctionnelle du Modèle-Vue-Contrôleur.
Journées Francophones des Langages Applicatifs (JFLA), 2005. In French.

F2 Sylvain Conchon, Jean-Christophe Filliâtre and Julien Signoles.
Le foncteur sonne toujours deux fois.
Journées Francophones des Langages Applicatifs (JFLA), 2005. In French.

F1 Julien Signoles.
Calcul statique des applications de modules paramétrés.
Journées Francophones des Langages Applicatifs (JFLA), 2003. In French.

B.8 Other Publications

A10 Florent Kirchner, Julien Signoles and Sara Tucci.
Assurer une confiance sans faille.
Clefs CEA n°64 *Voyage au cœur du big data*, 2017. In French.

A9 Mounir Assaf, Julien Signoles, Éric Totel and Frédéric Tronel.
Moniteur hybride de flux d'information pour un langage supportant des pointeurs.
Research Report 8326, Inria, 2013. In French.

A8 Julien Signoles.
Extension de ML avec raffinement: syntaxe, sémantiques et système de types.
PhD Thesis, Université Paris Sud, 2006. In French.
Bibliography entry [Sig06].

A7 Julien Signoles.
Towards a ML extension with Refinement: a Semantic Issue.
Technical Report 1440, Université Paris Sud, 2006.

A6 Julien Signoles.
Calcul statique des applications de foncteurs en présence d'effets de bord.
Master thesis, Université Paris Sud, 2002. In French.

A5 Julien Signoles, Thibaud Antignac, Loïc Correnson, Matthieu Lemerre, and Virgile Prevosto.
Frama-C Plug-in Development Guide.

User manual.
Bibliography entry [SAC+].

A4 Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles and Boris Yakobowski.
Frama-C User Manual.
User manual.

A3 Julien Signoles.
E-ACSL: Executable ANSI/ISO C Specification Language.
Reference manual.
Bibliography entry [Siga].

A2 Julien Signoles and Kostyantyn Vorobyov.
E-ACSL User Manual.
User manual.
Bibliography entry [SV].

A1 Philippe Herrmann and Julien Signoles.
Annotation Generation: Frama-C's RTE Plug-in.
User manual.
Bibliography entry [HS].