




Projet Décysif — Livrable 3.1

Memory Models for Pointer Programs: a State of the Art from the Point of View of Décysif Partners

January 2025

Claude Marché (Inria & Université Paris-Saclay), Guillaume Cluzel (TrustInSoft), Claire Dross (AdaCore), Jean-Christophe Filliâtre (CNRS & Université Paris-Saclay), Jacques-Henri Jourdan (CNRS & Université Paris-Saclay), Andrei Paskevich (Université Paris-Saclay), Raphaël Rieu-Helft (TrustInSoft)



Le projet Décysif est financé par la Région Île-de-France et par le Gouvernement Français dans le cadre du Plan France 2030

Contents

1 Introduction

The term *Deductive Verification* denotes the methodology for verifying that a computer program is conforming to a specification of its intended behavior, with the use of techniques based of automated theorem proving. In this context, the specification is written in some formal mathematical language, and the conformity of the code with respect to the specification is automatically reduced into a set of mathematical formulas than must proved valid. When the computer programs considered are expressed in a programming language making use of pointers to access their memory heap, the interpretation of programs into mathematical formulas must rely on a *logical description* of such pointers and of the memory heap: the so-called *memory model*.

When reasoning mathematically on pointer programs, the suitability of the memory model is crucial. One major challenge is the potential *aliasing* of memory contents: when using pointers, the same memory cell can be denoted by multiple "names", or "access paths". Modification of the memory content through one of these names should be taken into account when accessing the same memory cell via another name. More generally, *blocks* of memory cells can be read or written through such access paths, and these blocks may *overlap*. The property of non-overlapping for memory blocks is called their *separation*. Separation assumptions play an important role when reasoning about pointer programs, an early reference publication on that subject is due to Bornat in 2000 [10]. Since then, many different approaches for dealing with memory aliasing and separation have been proposed, including the landmark proposal *separation logic* [47]. We discuss them more in Section 6.

A memory model cannot be described uniformly among different programming languages. On the contrary, a specific memory model must be defined for each programming language considered. It is even possible to consider several memory models for different supported subsets of a given programming language, as in the WP plug-in of the Frama-C environment for analysis of C code [6]. The Décysif project gathers partners involved in the design of verification environments for several programming languages, involving different approaches for memory access. They develop multiple tools and each of them contains a different approach for dealing with aliasing and separation: the tool Why3 for WhyML programs, the SPARK environment for Ada code, the J3 analyzer for C and C++ code, and the Creusot tool for Rust code.

Generally speaking, there are numerous technical challenges related to the design of memory models. This is not the purpose of this document to review them. Instead we recommend to the interested reader the reading of a survey on that purpose by Leavens *et. al.* [37] published in 2007. The purpose of this document is to summarize the different memory models implemented in the tools of the Décysif partners, focusing on the important design choices, and briefly compare them with each other and with other existing approaches in the international community of deductive verification. From such a summary, we derive and expose some future work we want to perform in the context of the Décysif project.

This document is organized as follows. Section 2 presents the modeling of memory in the Why3 tool (developed by Inria), Section 3 presents the memory model of the SPARK environment for Ada (developed by AdaCore), Section 4 presents the design choices for the memory model in the J3 plugin for TIS-Analyzer (developed by TrustInSoft), and Section 5 presents the design choices for the memory model of the Creusot tool for Rust programs (developed by Inria). The latter sections already contains some element of positioning in the current state of the art, and Section 6 expands such positioning. Section 7 concludes by enumerating the different axis of work we planned within Décysif concerning memory models.

2 The Memory Model of Why3

The Why3 environment [25, 9, 7, 26] is designed and developed by the Inria partner of Décysif. It is not an environment for a particular mainstream programming language such as C or Java. Instead, it is designed as a *generic* environment for deductive verification. This genericity comes into different aspects. First, Why3 proposes its own programming language, called WhyML, dedicated to deductive verification, in the sense that it is designed to ease the generation of verification conditions, making the proof of these formulas as automatic as possible. This relies on a second generic aspect of Why3: its ability to communicate with a large set of provers to discharge the verification conditions [8]. A third generic aspect of Why3 is its suitability to be used as an intermediate language for deductive verification: as a matter of fact, WhyML is used as an intermediate language by the other tools mentioned below: SPARK, J3 and Creusot.

In the WhyML language, mutability of memory is supported, but to control the difficulties coming from aliasing and separation, mutation of memory is controlled by a dedicated powerful *static* typing system. This typing system is expressed on top of important notions of *ownership* and *regions* [24]. Thanks to this typing system, the potential issues with aliasing and memory separation are handled *during* the generation of VCs, so that the generated formulas do not need to take aliasing anymore into account. Roughly speaking, the generated VCs are as simple as if they were coming from a purely functional programming language, with no memory mutability at all. One must understand in particular that this approach differs fundamentally with other well-known approaches in the literature, such as *separation logic* [50] or *dynamic frames* [49], where non-aliasing and separation properties are still present in the logic formulas to be proved valid, which makes the work of provers significantly more difficult. The powerful region-based mechanism of Why3 comes at a cost though: the set of programs that one is writing in WhyML must pass the static type checking, which means the programmers must think carefully about how the side effects in their program must be expressed.

The research work around Why3 is still active nowadays, e.g. regarding the design of better constructs for writing specifications, the improvement of the ratio of proof automation, or the generation of counterexamples when proofs fail [18, 5]. In the context of Décysif, the efforts are planned on two directions: the design of an alternative to WhyML, and the improvement of counterexamples generation to better suit the need of front-ends of the partners. These are detailed in Section 7 below.

3 The Memory Model of SPARK

During the 1990s, SPARK emerged as a tool used to formally verify a subset of the Ada language [13]. In 2014, AdaCore released SPARK2014 [43] as a complete re-implementation of SPARK, made compatible with Ada 2012, a version of Ada that introduced specification contracts as part of Ada. Since that time, SPARK has been using WhyML as an intermediate language, and Why3 as a verification condition generator.

The Ada subset supported by SPARK highly restricts aliasing. It enforces a policy to ensure that a mutable object can only be accessed through a single name. This restriction allows translating SPARK programs into WhyML with a simple *flat* memory model, where all mutable locations are identified as WhyML mutable variables. Such a choice makes the generation of VCs simple enough to obtain a satisfactory level of automation of proofs, and also allows generating meaningful counterexamples [18].

Because they easily introduce aliasing, pointers (called access types in Ada) and memory allocation have been disallowed for a long time. In 2020, the SPARK tool was extended to support them. To control aliasing, a technique inspired by the Rust borrowing policy and the Prusti tool was designed and

implemented [23, 33]. An essential idea behind that technique is that each alias is statically known by the type system so that programs can continue to be translated into an aliasing-free WhyML program. Later on, the method implemented was slightly modified to follow the idea of *prophetic encoding* introduced originally by Matsushita *et al* [42].

Nowadays, the major issues with SPARK remain the need to increase the ratio of automation of proofs, and to produce better counterexamples. The objectives for SPARK in the Décysif project are detailed in Section 7 below.

4 The Memory Model of J3

Since its creation in 2013, TrustInSoft develops TIS-Analyzer, an environment for formal analysis of C and C++ source code. The central method for analyzing code is an abstract interpretation engine, that is aimed at detecting undefined behaviors in any kind of C programs. To complement the use of abstract interpretation when alarms are signaled, it was decided in 2020 to design and implement a new plug-in, called J3, to apply deductive verification techniques on such C and C++ code.

The initial design of J3 included important decisions regarding the supported subset of C. In essence, it was decided that any kind of code should be supported. That includes low-level C code, making use of complex, low-level, architecture-dependent, bit-wise manipulation, but also arbitrary use of pointers, and conversion (cast) of such pointers. Memory allocation and freeing must be supported too, so as any kind of aliasing and memory overlap. To reach such an ambitious goal, but not starting everything from scratch, it was decided to reuse some existing memory model, namely it was decided to use the CompCert V2 [40] memory model, a model already formalized in Coq and the basis of the C semantics on which the verified C compiler CompCert operates.

The formalization of the memory model for J3 is done using WhyML, it makes heavy use of bit-vectors [27]. It explicitly defines the notions of memory blocks and permissions, so that at the end, generated verification conditions for a C code contains explicit pointers, predicates expressing separation (or not separation) and other memory-related concepts. A consequence of this design is that in its current state, the proofs done using J3 suffer from a poor ratio of automation. It is one of the objective in Décysif to increase this ratio.

Another important design requirement for J3 was the ability to obtain a counterexample when a proof fails. In its current state, J3 is able to produce such counterexamples in some cases, but overall the methodology needs significant improvement. Even more, it is a central goal in Décysif to be able to turn such a counterexample into an *exploit*, that is, an explicit test case that exposes a vulnerability regarding safety or security.

The Décysif objectives for J3 are detailed below in Section 7.

Positioning J3 is far from being the only tool for deductive verification of C code. Historically, Frama-C/WP [6] is a deductive verification plugin for Frama-C, the environment from which TIS-Analyzer was derived. WP design was guided by different choices than those of J3, in particular it does not attempt to support any kind of memory aliasing. It also does not support low-level constructs such as unions or pointer casts. On the contrary, it provides a set of variants of memory models where separation is partially or even totally assumed. Also, it wasn't designed with the need of counterexamples in mind, and only very recently such a feature has been added as experimental. Comparison between J3 and WP on that matter will be interesting to investigate in the future.

Jessie is another historic deductive verification plugin for Frama-C, designed on top of the predecessor of Why3. It is not maintained anymore. J3 design inherits in part from Jessie. It is in particular planned to resurrect inside J3 a static separation analysis similar to the one of Jessie [30, 31]

VCC [17] was historically another tool for deductive verification of C, inspired by Spec# ownership (see Section 6 below). Not any kind of memory aliasing was supported. It is not maintained anymore. VST [1] is an environment built on top of CompCert and a Coq formalization of separation logic. To perform a proof of some C code, it is mandatory to use the Coq system, thus having a low level of automation. An advantage though is that the underlying memory model is low-level and precise. VeriFast [32] is a tool for deductive verification of C (and Java) code. It relies on separation logic and/or dynamic frames. To perform proofs, the code must be annotated by the user to explicit the reasoning regarding opening or closing definitions of memory-related representation predicates. Isabelle/C [51] is an environment for reasoning on C code on top of Isabelle/HOL and its layer AutoCorres for imperative programs. To perform proofs it is mandatory to use the Isabelle system interactively. Generally speaking, the tools above are not designed to produce exploits in case of proof failure, or even counterexamples.

Notice finally that J3 has the ambition to be applied on C++ code, which is hardly supported by any other tool.

5 The Memory Model of Creusot

Creusot is a tool for deductive verification of Rust programs, developed by Inria. The Rust programming language has a unique feature: it is equipped with a very elaborated static typing system, so that when a program passes this type checker, it can be considered *safe* in the sense that it is free from undefined behavior related to memory access.

Creusot leverages this property to produce verification conditions for safe Rust programs, that do not need to express anymore any property regarding aliasing and memory separation. To achieve this, Creusot uses a memory model involving a so-called prophetic encoding of mutable borrows, which roughly amounts to representing a borrow by a pair of its current value and its future value at the end of its lifetime [20, 19]. Technically, Creusot translates Rust into Why3 where memory reads and writes are encoded in WhyML mutable variables. Thanks to that approach, the obtained verification conditions are significantly easier to discharge by automated provers. The soundness of the translation based on prophetic encoding has been proved valid using Coq by extending the RustBelt development [41].

Positioning Historically, Prusti [3] is the first tool to implement a verifier for Rust programs, leveraging on the borrow checker. A significant difference with Creusot is that Prusti encodes Rust code into Viper, which is an intermediate language based on separation logic. Technically, this means that Prusti is not limited to safe Rust as Creusot is, so supports a larger subset of Rust. On the other hand, the generated verification are significantly harder to discharge by automated provers, as shown by the benchmarks we did [19].

Several other teams develop verification tools for the Rust programming language. Verus [36] is a verification tool for a language close to Rust, using SMT solvers as backend, developed jointly at CMU, Microsoft Research and MPI-SWS, and mostly targeting system code. Aeneas [29] is a verification tool developed jointly at Microsoft Research and Inria team Prosecco, which translates Rust programs into purely functional programs in proof assistants (like Lean or Coq), and mostly targeting cryptographic code. The Automated Reasoning Group at Amazon Web Service is interested in verification of Rust programs, they even proposed a crowd-sourced verification effort (`https://aws.amazon.com/blogs/`

opensource/verify-the-safety-of-the-rust-standard-library/). The latter web page lists of few other tools for verification of Rust code.

6 Additional Comparison and Positioning

There are other mainstream programming languages for which deductive verification tools exist. Historically, even before 2000, Java was among the first of these languages, starting with the tool ESC-Java [21] for which the specification language JML [12] was invented. At that time there were very few techniques for handling the heap memory conveniently. In the early 2000s, JML featured a notion of *data groups* [11] for such a purpose, coming with early notions of *ownership*, as it was implemented in the Spec# [4] tool for C#, or similar notions of *universes* [22].

Separation Logic [47] appeared around the same time in 2002, but at first there were hardly any automated tools using it, because of the lack of automated reasoning techniques dedicated to this special form of logic.

In 2006 was proposed the concept of *dynamic frames* [34] which was seen as an alternative to separation logic more amenable to automated reasoning, in particular with the variant of *implicit dynamic frames* [49] which was implemented in new tools like VeriFast [32] (2009, for Java and C) and Dafny [38, 39] (2010, featuring its own object-oriented programming language).

Separation Logic obtained a more obvious success in the context of interactive theorem proving tools. It is exemplified by the Iris [35] library for Coq. Separation Logic implemented in Coq gave birth to tools like Ynot [45] and CFML [15], both dealing with their own ML-like language. As already mentioned above in Section 4, this line of approach is applied to the C programming language nowadays: the Verified Software Toolchain (VST) builds upon the CompCert memory model and separation logic to allow (interactive) proofs of C programs in Coq, and Isabelle/Simpl [48] provides an infrastructure for reasoning about imperative programs in Isabelle/HOL, on top of which is built the Isabelle/C [51] environment and the AutoCorres [28] tool.

More recently in 2016 appeared the environment Viper [44], which is meant as a generic environment for automated verification using separation logic/dynamic frames. It is used as an intermediate language for the Prusti tool for Rust (already mentioned in Section 5) and Vercors [2].

Tools exist for the deductive verification of OCaml programs, namely the Gospel [14] environment which combines different techniques including separation logic in Coq, and Cameleer [46] which proceeds by translation into Why3 (for OCaml programs where an alias-free memory model is enough) and Viper (when a more complex memory model is required). The Lab for Automated Reasoning and Analysis (http://lara.epfl.ch/w/) at EPFL develops methods and tools for the verification of Scala programs. This includes the front-end tool Stainless [16] and a backend called Inox which implements a logic and its translation to SMT solvers, in a way similar to Why3.

To conclude with this section of overall positioning, let's emphasize that with the tools we develop in Décysif (in Why3, SPARK, and Creusot but with the notable exception of J3), we adopt a general approach where aliasing and separation are handled in an early stage of static typing based on ownership-like concepts, allowing to produce verification conditions that do not include any further memory-related notions. This makes the generated verification conditions more amenable to automation. J3 is an exception, since there is a requirement to support all C features even to most low-level ones. This is why improving the memory model of J3 remains an important future work in Décysif.

7 Planned Future Work within Décysif

We conclude this document with a review of the research actions that are planned with Décysif related to the memory models and the tools support.

Flexible Abstraction Barriers The on-going PhD thesis of Paul Patault is aimed at the design of an intermediate language in Why3, as an alternative of WhyML. The goal is to relax the standard approach where function calls act as a mandatory abstraction barrier: classically, when reasoning about a function call, only the contract of that function is visible. The new intermediate language under design, now called Coma, is based on a continuation passing style. It is intended to use Coma as a target from Creusot, as Coma offers new possibilities in particular regarding the translation of Rust closures.

Memory model for Coma In the context of the Coma verification language (see above), we want to explore a new abstraction/refinement mechanism for programs with mutable state. On the implementation side, this mechanism would admit arbitrary pointer aliasing, allowing us to verify algorithms with complex pointer structure. This verification would rely on memory models, which grant us greater expressiveness, at the price of more complex specification and reasoning. On the interface side, these memory models would be hidden and a simple alias-free specification would allow us to apply simpler Hoare-style reasoning to verify the client code. A typical example would be the design of a reusable software component for mutable maps, whose implementation uses hash tables and mutable singly-linked lists for buckets, but whose specification only requires a single mutable variable containing a dictionary. This work is part the ongoing PhD thesis of Paul Patault at Inria.

Future Work in SPARK The main pain points for users of pointers in SPARK currently are the multiple restrictions enforced to ensure statically recognizable aliases. We plan to work on investigating which restrictions can be relaxed to improve the expressivity of the language without losing the simple and efficient encoding of the memory model inside WhyML.

Increasing the level of automation inside the tool is also an axis of improvement. In particular, pointer-based recursive data-structures are currently encoded in a way which does not always give statisfactory results. We plan to work on improving our translation to WhyML and its underlying solvers.

Finally, we would like to investigate relaxing some of the non-aliasing constraints imposed by SPARK in non-executable ghost code. Indeed, such constraints are generally necessary to ensure compatibility between the executable semantics of the Ada code and how it is encoded for verification inside Why3. On code purely meant for analysis, it might be possible to relax some of these constraints without compromising soundness, like is done in Creusot.

J3 So far, the memory model of J3 does not give satisfactory results regarding the ratio of automated proofs (see benchmarks from deliverable 2.1), and there are plans to make it better. One important axis is to implement a static analysis of separation, similar to the one of the former plugin Jessie [30, 31], so that the generated verification conditions become simpler, with less need for memory-related predicates.

The main axis of work on J3 is towards the generation of exploits. It means that the potential counterexamples that are obtained from Why3 when proofs fail (see benchmarks from deliverable 1.1) must be turned into concrete test cases for the original C code.

Another axis of work for J3 is the generation of exploits not only on failed functional verification proofs, but also on alarms reported by the abstract interpretation engine. In that context the program is not fully annotated and the tool must instead constrain the counterexamples based on the values computed through abstract interpretation.

Extensions of Creusot The on-going PhD thesis of Arnaud Golfouse is aimed at extending the subset of Rust that is supported by Creusot. This includes support for concurrent programs, and support for the so-called ghost aliasing.

Supporting this extended set of Rust features need to reconsider the prophecy-based modeling, providing adequate extensions, show them correct, implement them, and evaluate them on concrete Rust code.

From Counterexamples to Exploits As already said above for J3, one major goal in the Décysif project is the generation of exploits from alarms obtained in the analyzers. When such an alarm is raised at a given location of a source, it means that the analyzer was not able to guarantee that the corresponding piece of code is safe regarding safety and security. Constructing an exploit is a way to be able to decide if such an alarm is a real bug or a false alarm.

The current objective is to start from potential counterexamples suggested by provers (see benchmarks in deliverable 1.1) and turn them into real test cases to be replayed with a concrete execution of the suspicious code.

This objective is shared between J3 and SPARK.

Counterexamples in Creusot So far Creusot is not designed with the ability to produce counterexamples in case of proof failure. This is thus another plan within Décysif to make this possible, including the fact that, in a second step, these counterexamples should be turned into exploits to be executed on the Rust original code.

References

[1] Andrew Appel. Verified software toolchain. In *European Symposium on Programming*, volume 6602 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2011. Tool page at `https://vst.cs.princeton.edu/`. `doi:10.5555/1987211.1987212`.

[2] Lukas Armborst, Pieter Bos, Lars B. van den Haak, Marieke Huisman, Robert Rubbens, Ömer Şakar, and Philip Tasche. The vercors verifier: A progress report. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification*, pages 3–18, Cham, 2024. Springer Nature Switzerland. `doi:10.1007/978-3-031-65630-9_1`.

[3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA):147:1–147:30, 2019. `doi:10.1145/3360573`.

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[5] Benedikt Becker, Cláudio Belo Lourenço, and Claude Marché. Explaining counterexamples with giant-step assertion checking. In José Creissac Campos and Andrei Paskevich, editors, *6th Workshop on Formal Integrated Development Environments (F-IDE 2021)*, Electronic Proceedings in Theoretical Computer Science, May 2021. URL: https://hal.inria.fr/hal-03217393, doi:10.4204/EPTCS.338.10.

[6] Allan Blanchard, François Bobot, Patrick Baudin, and Loïc Correnson. *Formally Verifying that a Program Does What It Should: The Wp Plug-in*, pages 187–261. Springer International Publishing, 2024. doi:10.1007/978-3-031-55608-1_4.

[7] Sandrine Blazy. Teaching deductive verification in Why3 to undergraduate students. In *FM Tea (Formal Methods Teaching)*, pages 52–66, Porto, Portugal, 2019. URL: https://hal.inria.fr/hal-02362306, doi:10.1007/978-3-030-32441-4_4.

[8] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. https://hal.inria.fr/hal-00790310. URL: http://hal.inria.fr/hal-00790310.

[9] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6):709–727, 2015. See also http://toccata.gitlabpages.inria.fr/toccata/gallery/fm2012comp.en.html. URL: http://hal.inria.fr/hal-00967132/en, doi:10.1007/s10009-014-0314-5.

[10] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[11] C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java-like Programs (FTFJP'03)*, 2003.

[12] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe 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*, 2004.

[13] Bernard Carré and Jonathan Garnsworthy. SPARK—an annotated Ada subset for safety-critical programming. In *Proceedings of the conference on TRI-ADA'90*, TRI-Ada'90, pages 392–402, New York, NY, USA, 1990. ACM Press. URL: http://doi.acm.org/10.1145/255471.255563, doi:http://doi.acm.org/10.1145/255471.255563.

[14] Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Belo Lourenço, and Mário Pereira. GOSPEL — providing OCaml with a formal specification language. In Annabelle McIver and Maurice ter Beek, editors, *FM 2019 23rd International Symposium on Formal Methods*, Porto, Portugal, October 2019. URL: https://hal.inria.fr/hal-02157484.

[15] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008. URL: http://gallium.inria.fr/~fpottier/publis/charguer aud-pottier-capabilities.pdf, doi:10.1145/1411204.1411235.

[16] Samuel Chassot and Viktor Kunčak. Verifying a realistic mutable hash table. In Christoph Benzmüller, Marijn J.H. Heule, and Renate A. Schmidt, editors, *International Joint Conference on Automated Reasoning*, volume 14739, pages 304–314. Springer, 2024. doi:10.1007/978-3-031-63498-7_18.

[17] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 429–430. IEEE Comp. Soc. Press, 2009.

[18] Sylvain Dailler, David Hauzar, Claude Marché, and Yannick Moy. Instrumenting a weakest precondition calculus for counterexample generation. *Journal of Logical and Algebraic Methods in Programming*, 99:97–113, 2018. URL: https://hal.inria.fr/hal-01802488, doi:10.1016/j.jlamp.2018.05.003.

[19] Xavier Denis. *Deductive Verification of Rust Programs*. Phd thesis, Université Paris-Saclay, 2023. URL: https://hal.science/tel-04517581.

[20] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: a foundry for the deductive verication of Rust programs. In *International Conference on Formal Engineering Methods - ICFEM*, Lecture Notes in Computer Science, Madrid, Spain, 2022. Springer. URL: https://hal.inria.fr/hal-03737878.

[21] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, December 1998. See also http://research.compaq.com/SRC/esc/.

[22] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[23] Claire Dross and Johannes Kanig. Recursive data structures in spark. In *Computer Aided Verification*, 2020.

[24] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. A pragmatic type system for deductive verification. Research report, Université Paris Sud, 2016. https://hal.archives-ouvertes.fr/hal-01256434v3. URL: https://hal.archives-ouvertes.fr/hal-01256434v3.

[25] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013. URL: http://hal.inria.fr/hal-00789533.

[26] Jean-Christophe Filliâtre and Andrei Paskevich. Abstraction and genericity in Why3. In Tiziana Margaria and Bernhard Steffen, editors, *9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 12476 of *Lecture Notes in Computer Science*, pages 122–142, Rhodes, Greece, October 2020. Springer. See also https://usr.lmf.cnrs.fr/~jcf/isola-2020/. URL: https://hal.inria.fr/hal-02696246.

[27] Clément Fumex, Claire Dross, Jens Gerlach, and Claude Marché. Specification and proof of high-level functional properties of bit-level programs. In Sanjai Rayadurgam and Oksana Tkachuk, editors, *8th NASA Formal Methods Symposium*, volume 9690 of *Lecture Notes in Computer Science*, pages 291–306, Minneapolis, MN, USA, June 2016. Springer. URL: `https://hal.inria.fr/hal-01314876`.

[28] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 429–439, Edinburgh, UK, June 2014. ACM. `doi:10.1145/2594291.2594296`.

[29] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages*, 6(ICFP):711–741, 2022. URL: `https://hal.science/hal-03931572`, `doi:10.1145/3547647`.

[30] Thierry Hubert. *Analyse Statique et preuve de Programmes Industriels Critiques*. Thèse de doctorat, Université Paris-Sud, June 2008. URL: `http://www.lri.fr/~marche/hubert08these.pdf`.

[31] Thierry Hubert and Claude Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, pages 81–93, Braga, Portugal, March 2007. URL: `https://hal.inria.fr/hal-03630177`.

[32] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[33] Georges-Axel Jaloyan, Claire Dross, Maroua Maalej, Yannick Moy, and Andrei Paskevich. Verification of programs with pointers in SPARK. In *Formal Methods and Software Engineering (ICFEM)*, pages 55–72, 2020. URL: `https://hal.inria.fr/hal-03094566`, `doi:10.1007/978-3-030-63406-3\_4`.

[34] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *14th International Symposium on Formal Methods (FM'06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283, Hamilton, Canada, 2006.

[35] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *26th European Symposium on Programming Languages and Systems*, volume 10201 of *Lecture Notes in Computer Science*, pages 696–723. Springer, 2017. `doi:10.1007/978-3-662-54434-1_26`.

[36] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. In *Proceedings of the ACM on Programming Languages*, volume 7 of *OOPSLA1*. ACM Press, 2023. `doi:10.1145/3586037`.

[37] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, pages 159–189, 2007.

[38] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[39] K. Rustan M. Leino and Valentin Wüstholz. The Dafny integrated development environment. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014.*, volume 149 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–15, 2014. doi:10.4204/EPTCS.149.2.

[40] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert memory model, version 2. Research Report RR-7987, INRIA, 2012. URL: https://hal.inria.fr/hal-00703441.

[41] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *International Conference on Programming Language Design and Implementation*, pages 841–856. ACM, 2022. URL: https://hal.inria.fr/hal-03777103, doi:10.1145/3519939.3523704.

[42] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. RustHorn: CHC-based verification for Rust programs. In Peter Müller, editor, *Programming Languages and Systems*, pages 484–514, Cham, 2020. Springer International Publishing.

[43] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015. doi:10.1017/CBO9781139629294.

[44] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[45] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *Proceedings of ICFP'08*, 2008.

[46] Mário Pereira and António Ravara. Cameleer: A Deductive Verification Tool for OCaml. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 677–689. Springer, 2021. doi:10.1007/978-3-030-81688-9_31.

[47] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

[48] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[49] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009 — Object-Oriented Programming*, Lecture Notes in Computer Science, pages 148–172. Springer Berlin / Heidelberg, 2009.

[50] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, January 2007.

[51] Frédéric Tuong and Burkhart Wolff. Deeply integrating C11 code support into Isabelle/PIDE. In *5th Workshop on Formal Integrated Development Environments (F-IDE 2019)*, volume 310, pages 13–28, 2019. doi:10.4204/eptcs.310.3.