The E-ACSL Perspective on Runtime Assertion Checking

Julien Signoles

▶ **To cite this version:**

Julien Signoles. The E-ACSL Perspective on Runtime Assertion Checking. International Workshop on Verification and mOnitoring at Runtime EXecution, Jul 2021, (online), Denmark. https://doi.org/10.1145/3464974.3468451, 10.1145/3464974.3468451 . cea-03313535

HAL Id: cea-03313535

https://cea.hal.science/cea-03313535

Submitted on 4 Aug 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The E-ACSL Perspective on Runtime Assertion Checking

Julien Signoles
Université Paris-Saclay, CEA, List
Palaiseau, France
Julien.Signoles@cea.fr

ABSTRACT

Runtime Assertion Checking (RAC) is the discipline of verifying program assertions at runtime, i.e. when executing the code. Nowadays, RAC usually relies on Behavioral Interface Specification Languages (BISL) *à la* Eiffel for writing powerful code specifications. Since now more than 20 years, several works have studied RAC. Most of them have focused on BISL. Some others have also considered combinations of RAC with others techniques, e.g. deductive verification (DV). Very few tackle RAC as a verification technique that soundly generates efficient code from formal annotations. Here, we revisit these three RAC's research areas by emphasizing the works done in E-ACSL, which is both a BISL and a RAC tool for C code. We also compare it to others languages and tools.

CCS CONCEPTS

• **Software and its engineering** → **Specification languages**; **Dynamic analysis**; **Source code generation**; *Software verification*; *Automated static analysis.*

KEYWORDS

Runtime Assertion Checking, Behavioral Interface Specification Language, Source Code Generation, Combination of Static and Dynamic Analyses

ACM Reference Format:
Julien Signoles. 2021. The E-ACSL Perspective on Runtime Assertion Checking. In *Proceedings of the 5th ACM International Workshop on Verification and mOnitoring at Runtime EXecution (VORTEX '21), July 12, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3464974.3468451

1 INTRODUCTION

Runtime Assertion Checking (RAC) is the discipline of verifying program assertions at runtime, i.e. when executing the code. It takes its roots in the late seventies when *assert* primitives have been added to several programming languages, including C [13]. However, such assertions were, and still are, limited to Boolean expressions. In the eighties, Eiffel [44] has included assertions into its Behavioral Interface Specification Language (BISL), which also includes others kinds of formal annotations, such as function contracts and invariants. Yet, at that time, RAC approaches remained very practical and were not studied from a theoretical perspective.

Since now more than 20 years, several works have studied RAC. Most of them have focused on BISL. Some others have also considered combinations of RAC with other techniques, e.g. deductive verification (DV). Very few tackle RAC as a verification technique that soundly generates efficient code from formal annotations. Here, we revisit these three RAC's research areas by emphasizing the works done in E-ACSL, which is both a BISL [16] and a RAC tool [52] for C code. We also compare it to others languages and tools.

2 BEHAVIORAL INTERFACE SPECIFICATION LANGUAGES

After the success of Eiffel, new BISL were designed for mainstream programming languages [25] at the turn of the millennium: first JML [38] for Java, then Spec# [3] for C#, VCC [14], ACSL [4] and E-ACSL [16] for C, CodeContract [17] for .NET, Spark2014 [26] for Ada and, more recently, GOSPEL [11] for OCaml. Dedicated BISL such as Boogie [40] and WhyML [22] were also designed in the meantime: they do not target a mainstream specification language but, instead, propose their own general-purpose specification and programming framework.

Even if formal specifications provided by BISL are useful in their own rights as precise code documentation, their main usage consists in checking them with verification tools. RAC is particularly suitable and lightweight for this purpose. Yet, DV *à la* Dijkstra [24] is the tool of choice for proving them for all possible executions. Abstract interpretation *à la* Cousot [49] is also possible [18, 45], as well as model checking [6], even if several logical properties cannot be easily verified with these techniques. Nowadays, most tools based on BISL support both RAC and DV, e.g. OpenJML [15] for JML, Spec#, Spark2014, GOSPEL and, since recently, Why3 for WhyML. Frama-C [31] relies on ACSL for DV and its subset E-ACSL for both RAC and abstract interpretation, E-ACSL being compatible with ACSL.

2.1 Core Features

Beyond code assertions, the core feature of BISL are function contracts that allow the user to specify what a function assumes from its callers (its preconditions), and what its implementation guarantees (its post-conditions), assuming the preconditions are satisfied.

Figure 1 shows a function contract for a function `binary_search` written in the E-ACSL language. E-ACSL, being defined as a syntactic subset of ACSL, it is also an ACSL specification. Preconditions (resp. post-conditions) are introduced with the keyword `requires` (resp. `ensures`). The second precondition shows the use of the built-in predicate `\valid`, stating that a range of memory locations

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VORTEX '21, July 12, 2021, Virtual, Denmark
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8546-6/21/07…$15.00
https://doi.org/10.1145/3464974.3468451

(here, the memory cells of a between 0 and len-1) has been properly allocated. Such memory predicates are specific to ACSL and E-ACSL that handles C code: the other BISL tackle higher-level programming languages that ensure memory safety by design. Yet, such properties are critical for C code. The third precondition calls the user-defined predicate sorted: most BISL provide the ability to define such predicates and logic functions. This one relies on an universal quantification: BISL are usually based on first-order logic. This specification also defines two behaviors (exists and not_exists) guarded by assumes clauses that specify when each of them are activated: behaviors provide a convenient way to distinguish different cases in a specification. complete and disjoint behaviors specifies that the behaviors covers all possible cases and do not overlap, respectively. Most BISL, including JML, Spark2014, ACSL and E-ACSL, provide such a notion of behavior or, at least, provide a way to specify exceptional cases, e.g. Spec# and WhyML. The assigns clause specifies a frame condition, i.e. the modifications in the program memory made by the function. Here, \nothing means that the function is effect-free (from an observational point of view).

```
/*@ predicate sorted(int* a, int len) =
   \forall integer i, j; 0 <= i <= j < len ==> a[i] <= a[j]; */

/*@ requires len >= 0;
    requires \valid(a + (0 .. len-1));
    requires sorted(a, len);
    assigns \nothing;

    behavior exists:
      assumes \exists integer i; 0 <= i < len && a[i] == key;
      ensures 0 <= \result < len && a[\result] == key;
    behavior not_exists:
      assumes \forall integer i; 0 <= i < len ==> a[i] != key;
      ensures \result == -1;

    complete behaviors;
    disjoint behaviors; */
int binary_search(int* a, int len, int key);
```

Figure 1: Example of an E-ACSL function contract.

Common features of most BISL also include data (or object) invariants, loop invariants, loop variants (i.e., decreasing measures inside loops), and ghost code. Ghost code is a powerful way to write code as specification without interfering with the original code [21]. Another common feature consists in referring to the *old* value of some memory location, i.e. its value in the the initial state of the contract from a post-condition. For instance, writing ensures G == \old(G)+1; for a function f means that the global variable G is incremented by one after having executed f. ACSL and E-ACSL generalize this feature to any program point thanks to the \at keyword: \at(x,L) refers to the value of x at program point L. They are the only BISL providing this generalized form. In the following, we will refer to such properties as *multi-state properties*.

2.2 Main Semantic Differences between BISL

BISL are usually based on typed first order logic. Their semantic differences mainly come from whether the languages target more RAC (e.g., JML, Spec#, Spark2014 and E-ACSL) or DV (e.g., WhyML and ACSL).

The first difference is about numbers. All BISL support bounded numbers (machine integers and floating-point numbers), with two possible flavors when they overflow: either raising an error, or reducing into the bounds with the appropriate modulo operation. For instance, JML provides both mode, while ACSL and E-ACSL use the latter. Yet, even if convenient for formal reasoning, providing mathematical numbers (integers over $\mathbb{Z}$ and real numbers) were historically not provided for BISL targeting RAC because they are hard or even impossible (for real numbers) to be computed in finite time. Nowadays, JML and Spark2014 provide a mode that supports them, while they are natively provided in E-ACSL. They are also provided in all BISL that target DV.

A related difference is about quantifiers that must be bounded for RAC in order to be executable in finite time. Indeed, iterating over all natural numbers is not possible at runtime. Even if bounded quantifiers restrict the BISL's expressive power, it is not an issue in practice since they affect only very few mathematical properties that rarely occur in real code.

A key difference is related to partial functions (e.g., a division): in classical first order logic implemented in theorem provers used by DV tools, such functions are represented by under-specified total functions, meaning that terms such as 1/0 are well defined, even if their values are not specified. However, such terms and predicates cannot be evaluated without error at runtime and so are problematic for RAC. This issue is known as the *undefinedness problem* [12]. For solving it, Chalin have proposed [10] to rely on the *strong validity* principle [33], which considers that a property is valid if and only if it is valid *and* defined. This semantics is consistent with the classical semantics, since any well-defined predicate has the same truth value in both. Nowadays, RAC-oriented BISL that also support deductive verification (e.g. JML, Spark2014, and E-ACSL) follow this semantics.

Another difference is *pure functions*, which are functions from the underlying programming language that can be safely used in specifications. For that purpose, they must always terminate and be side-effect free. Such functions are usually supported by RAC-oriented BISL (JML, Spark2014, and Spec#, whilst the E-ACSL tool has a mode that supports them), while they are not supported by DV-oriented BISL (ACSL, Why3, and Krakatoa [20], a variant of JML based on Why, the ancester of Why3). Indeed, pure methods are easy to execute for runtime assertion checkers, but lead to several issues for consistently verify them for deductive verifiers [35].

Last, one more semantic difference between BISL is actually independent from being RAC oriented or not. Indeed, frame conditions may have two flavors: *assigns* (or *writes*), that make explicit the memory locations that may be written to, and *modifies*, that make explicit the memory locations whose values may be changed. The former is stricter than the latter since, in the latter case, the other memory locations may change if (and only if) their value is restored before the end of the scope of the contract. There is currently no consensus between both semantics. For instance, JML relies on the *assigns* flavor, while ACSL and E-ACSL relies on the *modifies* flavor, even though they use the keyword `assigns`.

3 USING RUNTIME ASSERTION CHECKING IN PRACTICE

Checking annotations at runtime has various applications. First, when directly applied, RAC helps to strengthen code [9, 32] by discovering bugs upstream, when a well-identified assertion is violated, as opposed to downstream, by observing a consequence of the bug in the form of an unexpected outcome, typically a crash or a weird output.

Second, combining it with testing techniques, e.g. fuzzing, allows the user to detect more issues than using these techniques alone, since formal annotations make more invalid behaviors (e.g., broken invariants or invalid API uses) observable. For instance, a buffer overflow that would overwrite data in a supposedly-unchanged memory area $\mathcal{A}$ would usually remain undetected with standard testing techniques if the test verdict would not depend on $\mathcal{A}$.

Third, RAC may be combined with other verification techniques, notably DV, in various ways [35, 43], in particular when they are provided within a shared framework, such as Frama-C, OpenJML, Spark2014, Spec#, Why3, or GOSPEL. An usual combination with any static technique (typically, abstract interpretation, model checking, or DV) consists in discharging as many properties as possible statically, and relying on RAC to verify the remaining ones at runtime: it lowers the amount of effort and the level of expertise required by the static verifications, while reducing the runtime overheads induced by RAC. When combined with DV, RAC also helps debug formal annotations before proving them, find counter-examples in case of proof failure and, more generally, help to understand why a property is not automatically proven [5, 48].

Besides, even if not new, a recent trend consists in seeing BISL as low-level code-oriented specification languages which high-level properties may be generated to. It leads to enhance the area of applications of tools based on such languages, including RAC. Such high-level properties include liveness properties [23], non-interference properties [2], security automata [27], relational properties [7], and system-wide properties [50]. In the same spirit, implicit properties can be made explicit thanks to BISL. An example is security weaknesses [46] for which the implicit semantics depends on the underlying programming language. It worth noting that these approaches usually combine RAC with other verification techniques.

4 COMPILING FORMAL ASSERTIONS

Checking assertions at runtime requires to compile them into executable code. This process has only received little attention [12, 39] up to now. It might be explained by two reasons. First, it looks like most BISL's designers and tool providers do not consider it as challenging enough as expressed by the authors of Spec#: who state that "The run-time checker is straightforward" [3], while dedicating the rest of their paper to DV. Second, even if designing procedure for checking properties at runtime is at the heart of Runtime Verification, the relevant community is more appealed by checking temporal properties [19] than behavioral properties.

Yet, generating *sound* and *efficient* code from formal properties as powerful as the ones allowed in BISL is *not* straightforward. The rest of this section considers a few concrete examples for supporting this claim. All of them are related to research works that have been carried out within E-ACSL [52], the runtime assertion checker of Frama-C [31]. While Frama-C supports ACSL as specification language, its E-ACSL plug-in focuses on E-ACSL, its executable subset whose semantics is based on strong validity.

Three other properties are usually important for RAC tools: trustworthyness, transparency and isolation [30]. *Trustworthyness* means that, if a failure is detected, then there indeed should be one. *Transparency* means that the monitored program should functionally behave as the original one if no failure is detected. *Isolation* means that a failure, if any, must be reported at the point where the annotation is written, and not later.

4.1 Mathematical Properties

Checking mathematical properties efficiently and soundly at runtime is hard: relying on machine numbers is unsound if BISL supports mathematical numbers, while relying on dedicated libraries that encode mathematical numbers in the target programming language, such as GMP[1] for C code, is not efficient.

For being both efficient and sound, E-ACSL relies on a dedicated static type system [36] that automatically infers a precise type in which it is sound to compute a mathematical operation over mathematical integers in $\mathbb{Z}$ or rationals in $\mathbb{Q}$. For instance, on a 64-bit architecture, if `x` is a variable of type `int`, it infers that `x+1` may be soundly computed in type `long`: `x+1` might overflow if computed over `int`, but it would be unnecessary inefficient to use a GMP addition in such a context. Spark2014 has adapted the E-ACSL's type system to its context, but does not support rational numbers yet: as far as we know, E-ACSL is the only RAC tool that support properties over rational numbers.

Even if the code generated thanks to this type system has been proven to be very efficient in practice, it is worth noting that additional research should be carried out for handling definitions of recursive logic functions and inductive predicates in a precise and sound way. Also, no RAC tool currently supports irrational numbers. Actually, it is not possible to get a verdict at runtime in finite time in general in that context (for instance, for equalities over irrational numbers). Yet practical solutions based on incomplete verdicts should be possible.

4.2 Multi-State Properties

Checking multi-state properties, such as `P: x == \at(x,L)+1`, at runtime might seem straightforward: it just requires to save the value of `x` at program point L and use it when evaluating P. The first versions of E-ACSL followed this way. Yet, it is unsound in the general case. Consider for instance the predicate `Q`: `\forall integer i,j; 0<=i<LEN ==> 0<=j<i ==> t[i][j] == \at(t[i][j],L)+1`: the variables `i` and `j` are local to predicate Q and so cannot be used at L. Soundness requires to copy the relevant memory cells of `t`. Yet, knowing at compile time the exact memory cells that must be copied out is undecidable in the general case. For instance, if `LEN` is a variable, it might be unbound at L, or its value might have changed between L and the definition of Q. To circumvent this issue, existing approaches copy the whole memory

[1] https://gmplib.org/

block that contains t [34, 47]. Yet, it is not always efficient since it often copies too many memory cells. Therefore, the current version of E-ACSL implements an hybrid approach that statically computes an over-approximation of the memory cells that need to be copied out. In the best case, it is similar to the former E-ACSL implementation (but sound), whilst, in the worst case, it is similar to the other approaches. Therefore, it reconciles soundness and efficiency. Yet, it is worth noting, that no approach solves efficiently the problem in the general case: there is still room for improvements here.

4.3 Memory Properties

As already explained, memory properties, such as the predicate valid(p+(0..len-1)) used in Figure 1, are specific to ACSL and E-ACSL. Yet, checking such properties at runtime is the purpose of memory debuggers, which focus in detecting memory errors at runtime, such as buffer and heap overflows, and accesses to uninitialized data in the program memory.

In this context, the most efficient technique is *memory shadowing* that stores 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*, and accessible in constant time. The way of structuring these shadow values is refered to as *shadow state encoding* and may vary from one tool to another. One of the most efficient memory debuggers is currently AddressSanitizer [51]. Despite their efficiency, classical memory shadowing techniques are not able to detect all memory defects, in particular block-level memory errors (typically, accessing to an allocated block from another block) [28] or temporal memory errors (typically, accessing to a pointed-to object that is not the same as when the pointer was created) [53, 55]. Even if cleverly using the space left between aligned allocated blocks help circumvent partially this issue [29, 51], it is not enough for all memory properties.

Beyond memory shadowing, others techniques do exist, such as fat pointers [1] or dictionary-based approaches [29]. The former extends the pointer representation with information about memory block bounds. Yet, it suffers from modifying the size of pointed data, which makes this technique very hard to use safely in practice. The latter associates to each allocated memory address the necessary pieces of information and storing them in a separated dictionary. While powerful, this technique has efficiency issue since the dictionnary is not accessed in constant time.

E-ACSL has first tried a dictionary based approach [37]. Yet, it was not efficient enough in practice. Then, an hybrid approach mixing a dictionary and a shadow memory has been experimented with better results [28]. However, its implementation was hardly maintainable. Finally, we design a shadow-based technique relying on two new shadow state encodings (for the heap and the stack). It is efficient enough in practice [56], while more expressive than the existing shadow-based techniques in most cases [54]. In particular, it is able to detect that the program incorrectly accesses to a memory block from another memory block, even if both have been properly allocated earlier. It is worth noting that the instrumentation is independent from the underlying representation [42].

Static analysis can also optimize the instrumentation by preventing monitoring irrelevant memory blocks [41]. Thanks to the collaborative facilities provided by Frama-C, it is also possible to discharge most memory properties before running E-ACSL [46], typically by using Eva [8], the abstract interpretation-based static analyzer of Frama-C. Both techniques makes the code generated by E-ACSL much more efficient.

5 CONCLUSION AND FUTURE WORK

BISL are now well understood and more and more used in frameworks that combine RAC and DV, such as Frama-C, OpenJML, Spark2014 and Why3. Yet, the process of generating executable code from formal assertions has not been extensively studied. The works done on E-ACSL, the RAC tool of Frama-C, show that being both sound and efficient is often challenging and several open questions are still to be addressed, including efficient monitoring of properties over irrational numbers, monitoring of advanced multi-state properties, and monitoring of advanced memory properties.

ACKNOWLEDGMENTS

We would like to thank Thibaut Benjamin who proofreads an early version of this paper.

We would also like to thank all the persons who have contributed to E-ACSL: Nikolai Kosmatov, Kostyantyn Vorobyov, Fonenantsoa Maurica, Basile Desloges, Thibaut Benjamin, Guillaume Petiot, Arvid Jakobsson, Dara Ly, Romain Maliach-Auguste, and Félix Ridoux.

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N°883242, project ENSURESEC.

REFERENCES

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/178243.178446

[2] G. Barany and J. Signoles. 2017. Hybrid Information Flow Analysis for Real-World C Code. In *Int. Conf. on Tests and Proofs (TAP)*. https://doi.org/10.1007/978-3-319-61467-0_2

[3] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. 2011. . *Commun. ACM* (2011). https://doi.org/10.1145/1953122.1953145

[4] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. [n.d.]. *ACSL: ANSI/ISO C Specification Language*. http://frama-c.com/acsl.html.

[5] B. Becker, C. Lourenço, and C. Marché. 2021. Explaining Counterexamples with Giant-Step Assertion Checking. In *Workshop on Formal Integrated Development Environments (F-IDE)*.

[6] B. Beckert, M. Kirsten, J. Klamroth, and M. Ulbrich. 2020. Modular Verification of JML Contracts Using Bounded Model Checking. In *Int. Symp. On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*.

[7] L. Blatter, N. Kosmatov, P. Le Gall, V. Prevosto, and G. Petiot. 2018. Static and Dynamic Verification of Relational Properties on Self-composed C Code. In *Int. Conf. on Tests and Proofs (TAP)*. https://doi.org/10.1007/978-3-319-92994-1_3

[8] S. Blazy, D. Bühler, and B. Yakobowski. 2017. Structuring Abstract Interpreters through State and Value Abstractions. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'17)*. https://doi.org/10.1007/978-3-319-52234-0_7

[9] C. Casalnuovo, P. T. Devanbu, A. Oliveira, V. Filkov, and B. Ray. 2015. Assert Use in GitHub Projects. In *Int. Conf. on Software Engineering (ICSE)*. https://doi.org/doi10.5555/2818754.2818846

[10] P. Chalin. 2007. A Sound Assertion Semantics for the Dependable Systems Evolution Verifying Compiler. In *Int. Conf. on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2007.9

[11] A. Charguéraud, J.-C. Filliâtre, C. Lourenço, and M. Pereira. 2019. GOSPEL – Providing OCaml with a Formal Specification Language. In *Int. Conf. on Formal Methods (FM)*. https://doi.org/10.1007/978-3-030-30942-8_29

[12] Y. Cheon. 2003. *A runtime assertion checker for the Java Modeling Language*. Ph.D. Dissertation. Iowa State University.

[13] L. A. Clarke and D. S. Rosenblum. 2006. A Historical Perspective on Runtime Assertion Checking in Software Development. *SIGSOFT Software Engineering Notes* (2006). https://doi.org/10.1145/1127878.1127900

[14] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Int. Conf. on Theorem Proving in Higher Order Logics (TPHOL)*. https://doi.org/10.1007/978-3-642-03359-9_2

[15] D. R. Cok. 2011. OpenJML: JML for Java 7 by Extending OpenJDK. In *Int. Symp. on NASA Formal Methods (NFM)*. https://doi.org/10.1007/978-3-642-20398-5_35

[16] M. Delahaye, N. Kosmatov, and J. Signoles. 2013. Common Specification Language for Static and Dynamic Analysis of C Programs. In *Symp. on Applied Computing (SAC)*. https://doi.org/10.1145/2480362.2480593

[17] M. Fähndrich, M. Barnett, and F. Logozzo. 2010. Embedded Contract Languages. In *Symp. on Applied Computing (SAC)*. https://doi.org/10.1145/1774088.1774531

[18] M. Fähndrich and F. Logozzo. 2010. Static Contract Checking with Abstract Interpretation. In *Formal Verification of Object-Oriented Software (FoVeOOS)*. https://doi.org/10.1007/978-3-642-18070-5_2

[19] Y. Falcone, K. Havelund, and G. Reger. 2013. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*. https://doi.org/10.3233/978-1-61499-207-3-141

[20] J.-C. Filliâtre and C. Marché. 2007. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Int. Conf. on Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-540-73368-3_21

[21] J.-C. Filliâtre, L. Gondelman, and A. Paskevich. 2014. The Spirit of Ghost Code. In *Int. Conf. on Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-319-08867-9_1

[22] J.-C. Filliâtre and A. Paskevich. 2013. Why3 — Where Programs Meet Provers. In *European Symp. on Programming (ESOP)*. https://doi.org/10.1007/978-3-642-37036-6_8

[23] A. Giorgetti, J. Groslambert, J. Julliand, and O. Kouchnarenko. 2008. Verification of class liveness properties with Java Modeling Language. *IET Software* 2, 6 (2008). https://doi.org/10.1049/iet-sen:20080008

[24] R. Hähnle and M. Huisman. 2019. *Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools*. https://doi.org/10.1007/978-3-319-91908-9_18

[25] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. 2012. Behavioral Interface Specification Languages. *Computing Surveys* 44, 3 (2012). https://doi.org/10.1145/2187671.2187678

[26] D. Hoang, Y. Moy, A. Wallenburg, and R. Chapman. 2015. SPARK 2014 and GNATprove – A competition report from builders of an industrial-strength verifying compiler. *Software Tools for Technology Transfer* (2015). https://doi.org/10.1007/s10009-014-0322-5

[27] Marieke Huisman and Alejandro Tamalet. 2009. A Formal Connection between Security Automata and JML Annotations. In *Int. Conf. on Fundamental Approaches to Software Engineering (FASE)*. https://doi.org/10.1007/978-3-642-00593-0_23

[28] A. Jakobsson, N. Kosmatov, and J. Signoles. 2016. Fast as a Shadow, Expressive as a Tree: Optimized Memory Monitoring for C. *Science of Computer Programming* (2016). https://doi.org/10.1016/j.scico.2016.09.003

[29] R. W. M. Jones and P. H. J. Kelly. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Int. Workshop on Automatic Debugging (AADEBUG)*. https://doi.org/10.1145/1134285.1134309

[30] J. Kandziora, M. Huisman, C. Bockisch, and M. Zaharieva-Stojanovski. 2015. Runtime assertion checking of JML annotations in multithreaded applications with e-OpenJML. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*. https://doi.org/10.1145/2786536.2786541

[31] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. 2015. Frama-C: A Software Analysis Perspective. *Formal Aspects of Computing* (2015). https://doi.org/10.1007/s00165-014-0326-7

[32] P. S. Kochhar and D. Lo. 2017. Revisiting Assert Use in GitHub Projects. In *Int. Conf. on Evaluation and Assessment in Software Engineering (EASE)*. https://doi.org/10.1145/3084226.3084259

[33] B. Konikowska, A. Tarlecki, and A. Blikle. 1988. A three-valued logic for software specification and validation. In *VDM '88 VDM — The Way Ahead*. https://doi.org/10.1007/3-540-50214-9_19

[34] P. Kosiuczenko. 2010. An Abstract Machine for the Old Value Retrieval. In *Int. Conf. on Mathematics of Program Construction (MPC)*. https://doi.org/10.1007/978-3-642-13321-3_14

[35] N. Kosmatov, C. Marché, J. Signoles, and Y. Moy. 2016. Static vs Dynamic Verification in Why3, Frama-C and SPARK 2014. In *Int. Symp. On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*.

[36] N. Kosmatov, F. Maurica, and J. Signoles. 2020. Efficient Runtime Assertion Checking for Properties over Mathematical Numbers. In *Int. Conf. on Runtime Verification (RV)*. https://doi.org/10.1007/978-3-030-60508-7_17

[37] N. Kosmatov, G. Petiot, and J. Signoles. 2013. An Optimized Memory Monitoring for Runtime Assertion Checking of C Programs. In *Int. Conf. on Runtime Verification (RV)*. https://doi.org/10.1007/978-3-642-40787-1_10

[38] G. T. Leavens, A. L. Baker, and C. Ruby. 1999. *JML: A Notation for Detailed Design*. https://doi.org/10.1007/978-1-4615-5229-1_12

[39] H. Lehner. 2011. *A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking*. Ph.D. Dissertation. ETH Zurich.

[40] K. R. M. Leino. 2008. This is Boogie 2. (2008).

[41] D. Ly, N. Kosmatov, F. Loulergue, and J. Signoles. 2018. Soundness of a Dataflow Analysis for Memory Monitoring. In *Workshop on Languages and Tools for Ensuring Cyber-Resilience in Critical Software-Intensive Systems (HILT)*. https://doi.org/10.1145/3375408.3375416

[42] D. Ly, N. Kosmatov, F. Loulergue, and J. Signoles. 2020. Verified Runtime Assertion Checking for Memory Properties. In *Int. Conf. on Tests and Proofs (TAP)*. https://doi.org/10.1007/978-3-030-50995-8_6

[43] F. Maurica, D. R. Cok, and J. Signoles. 2018. Runtime Assertion Checking and Static Verification: Collaborative Partners. In *Int. Symp. On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. https://doi.org/10.1007/978-3-030-03421-4_6

[44] B. Meyer. 1988. Eiffel: A language and environment for software engineering. *Systems and Software* (1988). https://doi.org/10.1016/0164-1212(88)90022-2

[45] A. Ouadjaout and A. Miné. 2020. A Library Modeling Language for the Static Analysis of C Programs. In *Static Analysis Symp. (SAS)*. https://doi.org/10.1007/978-3-030-65474-0_11

[46] D. Pariente and J. Signoles. 2017. Static Analysis and Runtime Assertion Checking: Contribution to Security Counter-Measures. In *Symp. sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*.

[47] G. Petiot, B. Botella, J. Julliand, N. Kosmatov, and J. Signoles. 2014. Instrumentation of Annotated C Programs for Test Generation. In *Int. Conf. on Source Code Analysis and Manipulation (SCAM)*. https://doi.org/10.1109/SCAM.2014.19

[48] G. Petiot, N. Kosmatov, B. Botella, A. Giorgetti, and J. Julliand. 2018. How testing helps to diagnose proof failures. *Formal Aspects of Computing* (2018). https://doi.org/10.1007/s00165-018-0456-4

[49] X. Rival and K. Yi. 2020. *Introduction to Static Analysis: An Abstract Interpretation Perspective*.

[50] V. Robles, N. Kosmatov, V. Prevosto, L. Rilling, and P. Le Gall. 2019. Tame Your Annotations with MetAcsl: Specifying, Testing and Proving High-Level Properties. https://doi.org/10.1007/978-3-030-31157-5_11

[51] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Annual Technical Conf. (ATC)*. https://doi.org/doi/10.5555/2342821.2342849

[52] J. Signoles, N. Kosmatov, and K. Vorobyov. 2017. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. In *Int. Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*. https://doi.org/10.29007/fpdh

[53] M. S. Simpson and R. K. Barua. 2012. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. *Software: Practice and Experience* (2012). https://doi.org/10.1002/spe.2105

[54] K. Vorobyov, N. Kosmatov, and J. Signoles. 2018. Detection of Security Vulnerabilities in C Code using Runtime Verification. In *Int. Conf. on Tests and Proofs (TAP)*. https://doi.org/10.1007/978-3-319-92994-1_8

[55] K. Vorobyov, N. Kosmatov, J. Signoles, and A. Jakobsson. 2017. Runtime Detection of Temporal Memory Errors. In *Internation Conf. on Runtime Verification (RV)*. https://doi.org/10.1007/978-3-319-67531-2_18

[56] K. Vorobyov, J. Signoles, and N. Kosmatov. 2017. Shadow State Encoding for Efficient Monitoring of Block-level Properties. In *Int. Symp. on Memory Management (ISMM)*. https://doi.org/10.1145/3092255.3092269