Unified Compositional Formal Methods: Exact Separation Logic and the Gillian Platform for Correctness and Incorrectness Reasoning

Andreas Lööw
Imperial College London, UK

Daniele Nantes Sobrinho
Imperial College London, UK

Sacha-Élie Ayoun
Imperial College London, UK

Nat Karmios
Imperial College London, UK

Seung Hoon Park
Imperial College London, UK

Petar Maksimović
Imperial College London, UK
Runtime Verification Inc., USA

Philippa Gardner
Imperial College London, UK

1 Introduction

Our recent work has focussed on *unified compositional formal methods*, meaning formalisms and tools that support both over-approximate (OX) compositional reasoning about program correctness (that is, verification) and under-approximate (UX) compositional reasoning about program incorrectness (that is, true bug-finding).

In the area of program logics, we have introduced a *unified compositional program logic*, exact separation logic [6] (ESL), which unifies traditional OX separation logic (SL) [10, 13] and the more recent UX incorrectness separation logic (ISL) [5, 9, 12]. In the area of automated and semi-automated program analysis tools, we have developed a formalism for *unified compositional symbolic execution* and used this formalism to guide the development of a new unified version of the program analysis platform Gillian [1, 7]—this work is currently under review. Gillian previously only supported OX analyses; in its new version, we have added support for UX reasoning.

2 Unified Compositional Program Logics: Exact Separation Logic

In our work on ESL, we provide a semantics and proof system for exact separation logic quadruples of the form

$$(\mathsf{x} = x \star P)\ f(\mathsf{x})\ (ok : Q_{ok})\ (err : Q_{err})$$

where P is a pre-condition, Q_{ok} a success post-condition, and Q_{err} an error post-condition. Whereas SL quadruples

$$\{\mathsf{x} = x \star P\}\ f(\mathsf{x})\ \{ok : Q_{ok}\}\ \{err : Q_{err}\}$$

are only guaranteed to capture an over-approximation of the function behaviour (making them incompatible with true bug-finding), and ISL quadruples

$$[\mathsf{x} = x \star P]\ f(\mathsf{x})\ [ok : Q_{ok}]\ [err : Q_{err}]$$

only guarantee to capture an under-approximation of said behaviour (making them incompatible with verification), ESL quadruples capture the exact function behaviour in the sense that they neither over-approximate or under-approximate it, making them compatible with both verification and true bug-finding. By definition, the following relationship holds between these three types of quadruples:

$$\begin{aligned}(\mathsf{x} = x \star P)\ f(\mathsf{x})\ (ok : Q_{ok})\ (err : Q_{err}) \iff \\ \{\mathsf{x} = x \star P\}\ f(\mathsf{x})\ \{ok : Q_{ok}\}\ \{err : Q_{err}\} \land \\ [\mathsf{x} = x \star P]\ f(\mathsf{x})\ [ok : Q_{ok}]\ [err : Q_{err}]\end{aligned}$$

One use case of ESL is for library developers who would like to specify their (e.g., data-structure) libraries in such a way that these specifications are useful both in verification-based and testing-based application development. Before ESL, providing such function specifications required two separate proofs, one written in SL and another in ISL. These two proofs, as they would in effect be exact, would be almost identical and would result in duplicated work, as no OX- or UX-specific shortcuts could be taken. Importantly, application developers using an ESL-specified library would not be restricted to exact reasoning: for example, if they were only interested in verification only, they could use the provided ESL quadruples as if they were SL quadruples (as all valid ESL quadruples are also valid SL quadruples).

Abstraction. Perhaps surprisingly, despite ESL quadruples capturing exact function behaviour, they still allow for abstraction: e.g., the following quadruple for the list-length function of a singly-linked list is provable using ESL:

$$\begin{gathered}(\mathsf{x} = x \star \mathsf{list}(x, n)) \\ \mathsf{LLen}(\mathsf{x}) \\ (ok : \mathsf{list}(x, n) \star \mathsf{ret} = n)\ (err : \mathsf{False})\end{gathered}$$

It should however be noted that abstraction in UX logics, such as ISL and ESL, as we explore in our work on ESL, due to the inherent limitations of UX reasoning, cannot be used as freely as in OX logics.

Function call rule for ISL and ESL. Another contribution of the work on ESL is that we provide and prove sound, for the first time, a function call rule compatible with UX

compositional program logics, including ISL and ESL. To do so, we differentiate between external and internal function specifications, which gives us a mechanism to forget information about function-local variables at function boundaries. This is required since in UX reasoning, which features only backward consequence, it is not possible to lose any information, unlike in OX reasoning, where this can be easily accomplished using forward consequence.

3 Unified Compositional Symbolic Execution and Unified Gillian

Since its inception, Gillian has supported both compositional OX verification and compositional OX bug-finding. In its new unified version, the different analyses hosted by Gillian can now be run in the mode most appropriate for each analysis: e.g., true bug-finding with bi-abduction can be carried out in UX mode and verification can be carried out in OX mode. In summary, unified Gillian can now perform the whole-program symbolic testing of CBMC [4], the OX verification of VeriFast [2], and the UX true bug-finding of Meta's Pulse [5].

Unified compositional symbolic execution. Our formal development reaches beyond Gillian: we have introduced a new unified formalism for the consume-produce symbolic execution engine paradigm, a paradigm followed by multiple state-of-the-art OX symbolic execution engines [2, 8], including Gillian. At a high level, a consume-produce engine is based on two functions consume and produce, which, respectively, given an SL assertion, adds and removes the corresponding symbolic state to and from the current symbolic state. The two functions are used for implementing the compositional reasoning features of the engine, such as using function specifications for symbolically executing function calls and folding/unfolding of user-defined predicates.

Our unified formalism pinpoints the difference between OX and UX reasoning in the consume-produce paradigm down to the consume function, which is allowed to drop paths in UX reasoning, but not in OX reasoning.

We have proven that our engine formalism is both OX- and UX-sound. Moreover, we introduce an axiomatic interface for the consume and produce functions, specifying properties for the two functions that ensure that they fit into the engine and its soundness result. This interface modularises our soundness proof and decouples the consume and produce function implementations from the rest of the engine. In particular, we show that the full engine is sound if the associated consume and produce functions satisfy our interface, and separately provide example implementations of consume and produce which provably satisfy this interface.

The axiomatic interface also enables the function call rule of the engine to use OX, UX, and exact function specifications (quadruples), proved both inside and outside of the engine. For example, the engine can reason about the example library discussed in the previous section, which comes with exact specifications established using ESL.

Unified Gillian. As our formalism is based on the consume-produce paradigm already implemented by Gillian, and as we have identified the differences between OX and UX engines in this paradigm to be small, porting Gillian from an OX-only tool to a unified tool was relatively straightforward, and took approximately two weeks of work.

One of the main selling points of Gillian is its support for multiple programming languages, enabled by by Gillian being parametric on the programming language and memory model under analysis. Today, Gillian has been instantiated to C and JavaScript, with CHERI-C [11] and Rust instantiations in development. The instantiation process for unified Gillian requires is of the instantiation developers to ensure both OX and UX properties of the instantiation components (such as memory-model specific implementations of the consume and produce functions) to ensure OX and UX soundness of the full engine. This is analogous to instantiation for previous Gillian versions, where only OX properties were required.

Lastly, we are also in the process of developing a debugger GUI for Gillian [3] compatible with both OX and UX analyses.

References

[1] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part I: A Multi-language Platform for Symbolic Execution. In *Programming Language Design and Implementation*. https://doi.org/10.1145/3385412.3386014

[2] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods Symposium*. https://doi.org/10.1007/978-3-642-20398-5_4

[3] Nat Karmios, Sacha-Élie Ayoun, and Philippa Gardner. 2023. Symbolic Debugging with Gillian. In *International Workshop on Future Debugging Techniques*. https://doi.org/10.1145/3605155.3605861

[4] Daniel Kroening and Michael Tautschnig. 2014. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*. https://doi.org/10.1007/978-3-642-54862-8_26

[5] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022). https://doi.org/10.1145/3527325

[6] Petar Maksimović, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner. 2023. Exact Separation Logic. In *European Conference on Object-Oriented Programming*. https://doi.org/10.4230/LIPIcs.ECOOP.2023.19

[7] Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *Computer Aided Verification*. https://doi.org/10.1007/978-3-030-81688-9_38

[8] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation*. https://doi.org/10.1007/978-3-662-49122-5_2

[9] Peter W. O'Hearn. 2019. Incorrectness Logic. *Proceedings of the ACM on Programming Languages* 4, POPL (2019). https://doi.org/10.1145/3371078

[10] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer*

Science Logic. https://doi.org/10.1007/3-540-44802-0_1

[11] Seung Hoon Park, Rekha Pai, and Tom Melham. 2023. A Formal CHERI-C Semantics for Verification. In *Tools and Algorithms for the Construction and Analysis of Systems*. https://doi.org/10.1007/978-3-031-30823-9_28

[12] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification*. https://doi.org/10.1007/978-3-030-53291-8_14

[13] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science*. https://doi.org/10.1109/LICS.2002.1029817