Soft Verification for Actor Contract Systems

Bram Vandenbogaerde
Vrije Universiteit Brussel
Belgium

Abstract

Design-by-contract is a software engineering practice where programmers annotate program elements with contract specifications that make expectations towards the user and supplier of the program element explicit. This practice has been applied in various contexts such as higher-order programming languages. However, support for contracts in distributed actor programs is limited. Unfortunately, contract specifications need to be checked while executing the program which introduces a substantial overhead. To counter this, soft verification techniques have been proposed to verify (parts of) contract specifications, but have only been applied in the context of sequential programs. The goal of our research is therefore twofold: designing contract languages for distributed actor programs and developing techniques for their soft verification. In this context, we present a work plan and method, and show our preliminary results.

CCS Concepts

• **Software and its engineering → Domain specific languages**; **Specification languages**; **Automated static analysis**; *Software verification*.

Keywords

design-by-contract, actors, distributed programming languages, static program analysis

ACM Reference Format:
Bram Vandenbogaerde. 2024. Soft Verification for Actor Contract Systems. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3650212.3685551

1 Introduction

Design-by-contract [9] is a software engineering practice in which program elements (e.g., classes, functions, methods, ...) are annotated with *contract specification*. Usually, these specifications express pre-conditions, post-conditions and invariants on the program element itself. For example, a contract on a function could specify that it may only be called if its argument is a positive number. Higher-order contracts enable specifications on higher-order languages. Findler et al. [5] propose a contract system for higher-order functional languages. Such a system allows contract specifications on functions that return or accept functions as arguments. Blame assignment is a central concept in these contract systems: upon a contract violation, the application part responsible for the violation should be identified.

In the context of distributed programs, some contract systems have been proposed too [11, 16] but lack expressive power. For example, current contract system cannot express dynamic properties about message receivers, neither can it express dynamic constraints on communication effects over multiple message chains.

Unfortunately, contract systems can introduce significant overhead during a programs run time. This is because contract validity needs to be checked every time a function is called, a class field is updated, a message is sent to an actor… A common solution to this problem is to disable run-time contract checking altogether. However, this approach can result in unsafe code, especially in gradually typed systems were some redundant type safety checks are removed for efficiency. Nguyen et al. [10] instead propose *soft verification* (*SCV*) in the form of a static analysis that checks as many of these contracts as possible before running the program, leaving those that cannot be verified as residual contract checks in the program.

Unfortunately, their approach is conceived as a *whole-program* analysis which makes their analysis more difficult to scale. In the context of actor programs, such an approach is infeasible, as their state space is substantially larger due to the non-determinism inherent in actor turn interleavings. Furthermore, contracts for actor systems introduce a new *temporal* or *behavioral* type of contract [4, 6, 11]. These contracts express constraints on the behavior of an actor in the system, for instance by constraining what messages an actor can send and in what order. Such contract types are currently not supported by the verification system proposed by Nguyen et al. The overall objective of our research is threefold:

- **Rendering *SCV* compositional.** A common approach to speed up static analyses is to render them *modular compositional.* Such an analysis analyses smaller parts of the application first and then combines these results to compute the analysis result of the entire program. Its main instrument are *summaries* which are abstract descriptions of the behavior of the analysed part. These summaries speed up the analysis because they can be reused across different paths in the program. We propose to render *SCV* compositional by summarizing each component of the program with constraints leading to potential contract violations.
- **Designing and implementing an actor contract language.** We propose a novel contract system in the style of Findler et al. Its main contributions are expressing constraints on the communication effects of an actor's message handler, while allowing for more flexibility compared to session types. In tandem, we develop a novel blame theory for

this contract language for which we formulate and proof a blame correctness theorem.

- **Transposing *SCV* into the actor contract setting.** We will transpose soft contract verification into the actor setting. This requires designing an analysis that predicts the behavior of the program in order to verify its temporal properties.

The remainder of this Doctoral Symposium submission is structured as follows. First, we introduce our approach and how we plan to validate and evaluate its contributions. Then, we present our preliminary results for the compositional verifier and contract language. Finally, we conclude with an overview of future research direction and a conclusion.

2 Method

In this section we discuss how we aim to achieve our goals and how we evaluate them.

2.1 Compositional Soft Contract Verification

Compositionality has been investigated for a wide variety of analyses [1, 13]. A compositional analysis splits the program into multiple smaller parts (e.g., functions) and analyses them separately. Its goal is to speed up the analysis by enabling re-use of analysis' results. To obtain the analysis result of the whole program, a compositional analysis needs a view of the *information flow* through the program. In a functional programming language, information flows through function calls, from caller to callee. Thus, the analysis proceeds with the callees first and then propagates the analysis results to the callers.

The analysis results are called *summaries* as they summarize the behavior of the program part. The challenge lies in constructing a precise, but reusable summary. It should be sufficiently precise in order to have precise overall analysis results, but not too precise to preclude reusing results in a similar but non-identical context.

Existing soft-contract verifiers [10] support analysing function contracts in higher-order programming languages. In higher-order languages, control flow is also influenced by the data flow, since call targets flow as values through the program, making call graph construction more expensive.

We developed a static analysis for compositional soft verification of Racket contracts [15]. This analysis operates in two phases. The first phase analyses the program in a whole-program fashion for constructing the call graph, but does not perform soft contract verification. The second phase analyses the program for verifying its contracts in a compositional manner. We evaluated our approach on the following aspects:

- **Empirical evaluation.** The main purpose of rendering soft contract verification compositional is to improve the running time of its analysis. Therefore, our empirical evaluation consist of benchmarking our prototype against a large set of programs containing various types of contracts. Furthermore, to validate the correctness of our implementation, we compare it against existing work and check whether its results are the same or more precise.
- **Formal validation.** We also validated our approach against two formal criteria: *soundness* and *termination*. The soundness criterion requires that our analysis marks contracts as *safe* only when they can never be violated. The termination criteria requires that our analysis terminates on any program input, even if that program input does not terminate in a concrete run.

For the former aspect, we constructed a suite of benchmark programs based on existing work on soft contract verification. Moreover, we also use this benchmark suite for *soundness testing* by manually marking the program (and its parts) as *safe* or *unsafe* with respect to its contracts, and checking whether the analysis agrees with these annotations. We also measured the *degree* of summary reuse in order to explain potential performance improvements.

2.2 Contract Language for Distributed Actor programs

Next, we proceed with investigating contract languages for distributed systems. We identify four locations in which a contract system could intervene. (1) tag , (2) payload, and (3) receiver of messages and (4) communication effects of message handlers. Moreover, a contract language for actor systems should have precise blame assignment. The problem is that actor systems and the microservice architectures often put "internal actors" or services behind a publicly exposed service. For example, a service might be exposed through a load balancer which promises that replies will not be sent through the load balancer but instead directly to the client. Correct blame assignment should assign blame to the load balancer and not to the client when the client fails to do so. Thus, contract systems where blame labels align with actor boundaries is not sufficient and a more precise blame assignment method is needed.

Moreover, constraining communication effects requires changes to existing blame assignment theory, as current contract systems do not take the behavior of the constrained program elements into account other than their input and output effects. Thus, we validate our contract system on the following aspects:

- **Expressive power.** The contract system should be sufficiently expressive to express constraints on real-world programs and patterns emerging from these programs.
- **Formal validation.** Finally, we also validate our approach formally by formulating and proving theoretical properties about the designed systems. To this end we focus on *blame correctness* based on a property formulated by Dimoulas et al. [2, 3]. Their correctness property correlates blame assignment with provenance of values through the program. If both independent sources of information align then the blame is correct. Put differently, a party is to blame when the violating value originated from that party. To validate our approach we formulate a similar blame correctness property for actor contracts and provide a corresponding proof.

2.3 Verification of Distributed Actor Contracts

We aim to bring soft contract verification to the distributed actor setting. Existing soft contract verification techniques for sequential programs rely on *abstract symbolic execution*. In this setting, an abstract interpreter is adapted to work with both abstract values and symbolic values. The analysis then computes a set of reachable paths and associates symbolic path constraints with them. These path constraints are subsequently used to eliminate program paths that have unsatisfiable path constraints.

Thus, to transpose existing soft contract verification techniques novel symbolic representations are needed for representing actor-specific values. For example, our contracts on the receiver of messages need a symbolic representation of an actor reference and its identity. Contract systems for actor languages usually introduce a *behavioural* type of contracts. This type of contract usually puts constraints on the communication effects of an actor. Existing soft verification techniques do not take these communication contracts into account and cannot reason about them.

A final concern is *scalability*. We already rendered the analysis compositional for verifying contracts in sequential programs, but additional steps are needed to render the analysis of communication contracts compositional.

To evaluate our approach we take similar steps as our first research goal: empirical evaluation and formal validation. As a set of benchmark programs we consider the following sources:

- **Reactive design patterns [8].** These patterns describe a set of recurring strategies for implementing certain aspects of a distributed system. Thus, they are an ideal candidate to test our analysis on real-world use cases. For each pattern, we derive a software contract in our contract language. Then, we annotate programs implementing the patterns with these contracts and verify their validity.
- **Savina [7] benchmark suite.** The Savina benchmark suite contains a set of actor programs which are frequently used for testing the performance and correctness of actor system implementations. We annotate the programs in this benchmark suite with contracts that encode the desired properties of the program. To test our analysis, we then systematically inject contract violations in these program in order to test whether our verifier detects them correctly.

3 Preliminary Results

In this section we discuss the preliminary results for each goal of our research. We do so by formulating a set of research questions and answering them with our preliminary results.

3.1 Compositional Analysis

For our first research goal we answer the following research questions:

- **RQ1: does the proposed compositional analysis improve the performance of the analysis?** We measure this using the running time of the analysis. To enable fair comparisons, we reimplemented the verification techniques from existing work, which were written in Racket, into our own analysis framework, which is written in Scala.
- **RQ2: does the proposed compositional analysis improve the precision of the analysis results?** Again, this is measured using our own reimplementation of existing work. Here, we compare the number of contracts marked as *safe* by both variants of the analysis. This approach is sound because we only consider programs that contain no contract violations for our evaluation.
- **RQ3: to what degree are summaries reused?** To measure the degree of summary reuse, we measured the number of function calls to the same function. We also investigated the relation between analysis speed and the number of components (a unit of work in the analysis, usually a function call) in the analysis.

RQ1. Our results [15] indicate that the compositional analysis improves the performance of the soft contract verifier substantially (up to 3 times faster). Measuring the two phases of our analysis separately, we notice that the first phase (call graph construction) takes the majority of the analysis' time. This indicates that our second phase (summary-based soft verifier) is efficient, but that the call-graph construction efficiency could be improved.

RQ2. Our results also show that the precision of the analysis is improved. Individual examples show that this could be because the analysis propagates conditions under which contract violations could occur as opposed to entire program paths.

RQ3. Finally we investigated how summaries are propagated through the analysis. To this end, we measured the in-degree of each function in the call graph. We found that, on our set of benchmark programs, the in-degree was on average between 1 and 2, meaning that most functions have only one or two call sites. Thus, the opportunity of summary reuse at different call-sites seems rather low. Nonetheless, summaries from the same call-sites but at different program paths can be reused. This causes the analysis to become faster since the number of components can be reduced drastically, resulting in less work for the fixpoint algorithm and a faster convergence rate.

3.2 Contract System for Actor Programs

To achieve our second research goal we designed and implemented a contract system for actor languages by formulating contract types for the four identified locations (cf. supra). We answered [14] the following research questions which directly relate to our method:

RQ1: Can our contract language express contracts for real-world applications? Instead of focussing on real-world implementations appearing in public repositories, we decided to focus on common patterns emerging from them. Kuhn et al. [8] propose a catalog of so-called *reactive design patterns* which are patterns that are frequently used for developing distributed applications. We have shown that our contract language can be used to verify that a set of actors follows these patterns correctly by implementing a set of contracts for them. To show the practical feasibility of our approach we implemented our contract language on top of an actor system for Racket. This language was chosen since it contains a powerful contract system for sequential programs, but not for distributed or concurrent ones.

RQ2: Is our contract system formally correct? To validate the formal correctness of our approach we focus on blame correctness. Blame correctness was first formulated by Dimoulas et al. [2] to compare different types of blame assignment semantics in sequential programs. We used this property as a foundation to formulate our own blame correctness theorem and subsequently described its proof. The major difference is the addition of contracts on *communication effects* which require additional rules for blame assignment and their treatment in the blame correctness theorem.

RQ3: What type of constraints can our contract system express that others cannot? We theorize that our contract system can express strictly more types of constraints than existing work. To show this, we aim to simulate existing work such as multiparty session types in our contract language. Our hypothesis is that we can perform this simulation mechanically by adding contract boundaries to each actor within a session. As previously discussed, our contract boundaries are more flexible to account for differences between public-facing actors and internal actors.

3.3 Soft Verification for Actor Contracts

Below we formulate three research questions for our third research goal. Note that we have not answered these research questions entirely as of yet, but provide some preliminary insights already.

RQ1: What novel symbolic representations are needed for verifying contracts in the actor setting? Actors and their contracts introduce a set of new values compared to sequential programs. A value of interest for contract checking is the *receiver* of a message. To check them, actor references need to be symbolically represented and importantly, their *cardinality* needs to be represented. We expect that similar techniques as mailbox abstractions and abstract counting are applicable in this context, but further research is needed.

RQ2: How does reducing the symbolic path constraints affect the performance of the analysis? In contrast to sequential programs, interactions with other actors happen in an asynchronous fashion. This decouples the control flow of the sender of a message from the control flow of the receiver actor. Thus, messages are rarely sent with specific program paths on the receiver in mind, rather the actor is seen as an independent process that has its own separate execution paths. This poses a challenge for soft verification as it relies on constraints on these paths, thus a subset of these constraints relevant to the message should be extracted from path path condition at the send-site and attached to the message itself. We think that this approach provides a more scalable means of checking the validity of actor contracts.

RQ3: Can trace-based semantics be abstracted to verify communication contracts? Contracts on communication effects are usually implemented by keeping track of a message trace. For the purpose of verification, this message trace needs to be abstracted to account for all possible program behavior. Existing work [12] already provides some abstractions for representing sets of messages, their order and multiplicity. However, verification of communication contracts requires an underapproximation of the actual communicaton effects, as it aims to *rule out* contract violations. Existing work provides an overapproximation and can express *may-send* relations on set of messages. Instead, a *must-send* relation on the set of messages is needed.

4 Conclusion

Design-by-contract is a powerful software engineering practice safeguarding the robustness of programs. This approach has been explored in a distributed actor setting to some extent, but most approaches are imprecise with regards to blame assignment. Unfortunately, applying design-by-contract in practice results in substantial run-time overhead. This is because each contract needs to be checked every time some interaction between two contracted parties in the program happens. Thus, this PhD research is focussed on (1) addressing shortcomings in existing contract systems for distributed actor programs, and (2) designing and implementing static analyses for verifying these contracts.

To achieve these goals, we developed a contract language, implemented it in Racket with actor support, and proved its blame correctness. Furthermore, we improved the performance of existing soft verification techniques for sequential programs by rendering them compositional. The current goal is to transpose these techniques to an actor contract setting. To this end, we need to formulate symbolic representations for actor and contract values as well as to design a novel analysis for checking communication contracts.

References

[1] P. Cousot and R. Cousot. 2001. Compositional Separate Modular Static Analysis of Programs by Abstract Interpretation. In *Proceedings of the Second International Conference on Advances in Infrastructure for E-Business, E-Science and E-Education on the Internet, SSGRR 2001*. Scuola Superiore G. Reiss Romoli, Compact disk, L'Aquila, Italy.

[2] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct blame for contracts: no more scapegoating. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 215–226. https://doi.org/10.1145/1926385.1926410

[3] Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. 2016. Oh Lord, please don't let contracts be misunderstood (functional pearl). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP '16)*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 117–131. https://doi.org/10.1145/2951913.2951930

[4] Tim Disney, Cormac Flanagan, and Jay McCarthy. 2011. Temporal higher-order contracts. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 176–188. https://doi.org/10.1145/2034773.2034800

[5] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 48–59. https://doi.org/10.1145/581478.581484

[6] Hannah Gommerstadt, Limin Jia, and Frank Pfenning. 2022. Session-typed concurrent contracts. *J. Log. Algebraic Methods Program.* 124 (2022), 100731. https://doi.org/10.1016/J.JLAMP.2021.100731

[7] Shams Mahmood Imam and Vivek Sarkar. 2014. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014*, Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela (Eds.). ACM, 67–80. https://doi.org/10.1145/2687357.2687368

[8] Roland Kuhn, Brian Hanafee, and Jamie Allen. 2017. *Reactive Design Patterns* (1st ed.). Manning Publications Co., USA.

[9] Bertrand Meyer. 1998. Design by Contract: The Eiffel Method. In *TOOLS 1998: 26th International Conference on Technology of Object-Oriented Languages and Systems*. IEEE Computer Society, 446. https://doi.org/10.1109/TOOLS.1998.711043

[10] Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2018. Soft contract verification for higher-order stateful programs. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 51:1–51:30. https://doi.org/10.1145/3158139

[11] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. 2015. Computational contracts. *Sci. Comput. Program.* 98 (2015), 360–375. https://doi.org/10.1016/J.SCICO.2013.09.005

[12] Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. 2017. Mailbox Abstractions for Static Analysis of Actor Programs. *Dagstuhl Artifacts Ser.* 3, 2 (2017), 11:1–11:2. https://doi.org/10.4230/LIPICS.ECOOP.2017.25

[13] Quentin Stiévenart and Coen De Roover. 2020. Compositional Information Flow Analysis for WebAssembly Programs. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide,*

Australia, September 28 - October 2, 2020. IEEE, 13–24. https://doi.org/10.1109/SCAM51674.2020.00007

[14] Bram Vandenbogaerde, Quentin Stiévenart, and Coen De Roover. 2024. Blame-Correct Support for Receiver Properties in Recursively-Structured Actor Contracts. *Proc. ACM Program. Lang.* 8, ICFP, Article 254 (aug 2024), 29 pages. https://doi.org/10.1145/3674643

[15] Bram Vandenbogaerde, Quentin Stiévenart, and Coen De Roover. 2022. Summary-Based Compositional Analysis for Soft Contract Verification. In *22nd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Limassol, Cyprus, October 3, 2022*. IEEE, 186–196. https://doi.org/10.1109/SCAM55253.2022.00028

[16] Lucas Waye, Stephen Chong, and Christos Dimoulas. 2017. Whip: higher-order contracts for modern services. *Proc. ACM Program. Lang.* 1, ICFP (2017), 36:1–36:28. https://doi.org/10.1145/3110280