Counterexample Guided Inductive Repair of Reactive Contracts

Conference Paper · November 2021

DOI: 10.1109/ASE51524.2021.9678548

5 authors, including:

Soha Hussein
University of Minnesota Twin Cities
11 PUBLICATIONS **84** CITATIONS
SEE PROFILE

Vaibhav Sharma
University of Minnesota Twin Cities
21 PUBLICATIONS **208** CITATIONS
SEE PROFILE

Stephen McCamant
University of Minnesota Twin Cities
75 PUBLICATIONS **4,124** CITATIONS
SEE PROFILE

Sanjai Rayadurgam
University of Minnesota Twin Cities
67 PUBLICATIONS **1,056** CITATIONS
SEE PROFILE

2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678548

Counterexample Guided Inductive Repair of Reactive Contracts

Soha Hussein[§], Vaibhav Sharma, Stephen McCamant, Sanjai Rayadurgam and Mats Heimdahl
University of Minnesota. Minneapolis, MN, USA
soha@umn.edu, vaibhav@umn.edu, mccamant@cs.umn.edu, rsanjai@umn.edu, heimdahl@umn.edu

***Abstract*—Using third-party executable components to build control systems poses challenges for verification. This is because the informal behavior descriptions that typically accompany the components often fall short of the needed rigor. Consequently, there is a need to formalize a component contract that is strong enough to help establish system properties and also weak enough to account for all potential component behaviors in the system's context. In this paper, we present a novel approach that allows an analyst to hypothesize a component contract, explore if the component meets the contract, and, if not, have automated support to help *repair* the contract. Preliminary results show that, in more than 32% of the cases, the repaired contract is logically equivalent to a developer-written one; in a further 63% of cases, it is a distinct, valid, and non-trivial property of the component.**

I. Introduction

Computer-controlled systems, such as in reactive systems, are typically developed by leveraging components developed and supplied by third-parties. Such components may be delivered with only the compiled code made available, and they might lack rigorous requirements (or *contracts*) defining the assumptions made on a component's operating environment and the component's guaranteed behavior. Integrating such components in rigorous software development processes (in formal model-based development [1], [2]) poses obvious challenges. If the system architecture calls for a component with a certain behavior (a specific contract), given the informal nature of the requirements and the lack of source code for a component, predictably integrating that component in the architecture is difficult. For formal architectural verification, one could hypothesize a contract based on the informal description of the component and proceed with the system-level verification. The approach leads to two problems; if the hypothesized contract is too weak, we may fail to verify system-level properties and subsequently discard a perfectly good component because we could not establish a more accurate (more restrictive or *tighter*) contract; if the hypothesized contract is erroneous or too strong, we may succeed with the system-level verification, but the system may fail in operation since the component does not meet its hypothesized contract. Thus, establishing a valid and tight enough (does not over-approximate the component behavior to a point where verification of system-level properties fail) contract is crucially important.

[§]Lecturer on leave of absence
Ain Shams University. Cairo, Egypt.
soha.hussein@cis.asu.edu.eg

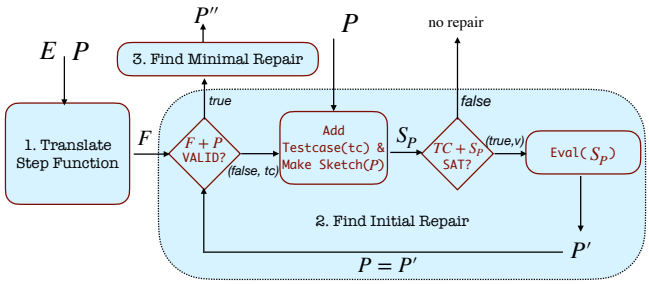

Fig. 1. Overview of the approach with a focus on Find Initial Repair step. E, S, F, are the executable, sketch, and the SDF. P, P', and P'' are the hypothesized, candidate, and repaired properties. S_p, tc, and TC are the sketch for property P, a single test case, and the set of all test cases, respectively.

Our proposed technique takes as inputs a hypothesized contract and an executable reactive component as Java bytecode. To repair an invalid contract, we use a Counter Example Guided Inductive Repair (CEGIR) algorithm constrained by a time budget. Our approach consists of three steps. The *translation* step attempts to translate the executable component and the contract into a common representation. The *initial repair* step attempts to find a variant of the contract that is valid. Finally, the *minimal repair* step attempts to find the tightest repair derived from the initial repair. Our approach adapts the sketching [3] technique to allow analysts control over the sub-expression to be repaired and the grammar of the expected repair. This can be exercised in the manual mode. In the evaluation we used the automated mode for generating repair sketches. We classify generated repairs into *matching repairs*, repairs that are logically equivalent to the developer-written one, and *relevant repairs*, repairs that are not implied by the original unmutated property, yet they are distinct valid and non-trivial. Our preliminary results show the technique's effectiveness with a success rate of 96.5%, where 32.64% are matching repairs and 63.89% are relevant repairs.

II. Technique

Our approach in Fig. 1 consists of three steps. In the *translation step* the executable component E and the hypothesized input property[1] are translated P into a common representation. We use synchronous data flow (SDF) as the common representation, which allows us to translate the executable E into a

[1]In what follows, we use the terms *contract* and *property* interchangeably.

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00131

function F that maintains the executable stateful behavior by defining an additional input/output pair representing the pre- and post-states.

In the *initial repair* step, indicated in the large rounded blue rectangle in Fig. 1, our approach attempts to find a valid variant of P that we call an *initial repair*. This component utilizes CEGIR algorithm and alternates between verification (`VALID?`) and bounded model checking queries (`SAT?`) queries. If the hypothesized property P is not true for F (the SDF representation of the executable), then we extract a test case (tc) from the counterexample, and we use it together with the automatically generated sketch S_p to find valuations for the sketch such that the set of all test cases (*TC*) pass. Note that the sketch specifies the grammar of the repaired expression. At this point, one of two things could happen: either no valuation for the sketch is possible, in which case our technique cannot find a repair, or there exists some valuations v for the sketch that satisfies the test cases in *TC*. In the latter case, the sketch S_p is evaluated ($Eval(S_p)$) using valuations v to obtain the *candidate* property P'. Note that at this point, P' is only true on the sampled test cases; this requires our technique to check whether P' is, in fact, `VALID?` on all inputs.

In the *minimal repair* step (Find Minimal Repair), our approach attempts to find a minimal/tightest repair than the initial repair found by the previous step. In this step our technique attempts to find, within a given time budget, a strongest property P'' matching the repair sketch that is stronger than P (i.e., $P'' \Rightarrow P$). To carry out the above `VALID?` and `SAT?` queries, we use JKind [4], an open-source industrial-strength infinite-state model-checker for safety properties for models expressed in the SDF language Lustre [5].

III. Evaluation

To evaluate our approach we used the Java implementation of Wheel Braking System (WBS) [6], [7], [8]. WBS is implemented in 265 lines of code and it includes two passing assert statements that we used as valid properties. We simulated faults by introducing up to 2 mutations on the valid properties, using logical and relational operator replacement [9].

We ran our technique on 70 repair attempts per property (280 in total), where each repair attempt consists of a mutated property and a repair location. We generated repair sketches automatically for each repair attempt. Each repair attempt tries to repair a subexpression of the mutated property. The grammar of the generated sketch uses standard logical operations, integer comparison operators, and integer constants.

We used the default engines of JKind for the `VALID?` queries and we only enabled the bounded model checking engine for the `SAT` queries, with 10 minutes time out for each query. We allowed up to 5 iterations for finding an initial repair, and 30 for finding a minimal repair. We used Z3 [10], [11] as the back-end SMT solver and imposed a 1-hour overall timeout per mutant. The experiment was conducted on a machine running Ubuntu 16.04.6 on a 3.6 GHz Intel Core i7-7700 CPU processor with 32 GB RAM.

TABLE I: Repair results for WBS, times in seconds

all attempts #	280		
unsolvable # / solvable #	136 / 144		
repaired # / %	139 / **96.53%**		
results classes	**match**	**relevant**	**unrepaired**
attempts	32.64%	63.89%	3.47%
minimal reached %	97.9%	80.4%	N/A
TO %	2.1%	19.6%	100%
median/avg execution	17.52 / 118.23	60.87 / 529.16	67.47 / 283.80
median/avg SAT?	8.54 / 107.07	34.25 / 507.44	52.02 / 270.49
median/avg VALID?	9.18 / 10.38	17.58 / 18.56	14.60 / 12.87

Table I shows the results of our technique. Among the 280 randomly selected repair attempts, we disregarded 136 because they were unsolvable (either the mutated property did not violate the component or no valid non-trivial repair exists using the automatically generated grammar). The remaining 144 were solvable attempts that our technique should repair, and finally, 139 (96.53%) were the repaired (repaired#) attempts.

Generally, relevant repairs were much more likely to be generated than matching repairs (63.89% vs 32.64%); however, matching repairs were much faster (118.23s vs 52.16s). In both cases, termination due to finding a minimal repair (minimal reached %), i.e., no tighter repair can be found, was the dominant termination reason (97.9%, and 80.4%). Timeouts (TO) complements the minimal reached results. A timeout can either be due to a query timing out or an internal loop bound reached. In all three result classes, the synthesis step (`SAT?`) was the most time-consuming operation, the median execution time for it was around a minute. These results indicate that our approach can successfully generate useful (non-trivial) repairs in a reasonable time budget.

IV. Related Work

Unrealizable specification repair [12], [13], [14], [15] is the closest to our work, though it only repairs specifications for which no implementation can exist, and only by adding assumptions. Program repair [16], [17], [18], [19], [20], [21] uses techniques similar to ours, though it searches for modifications to improve imperative software instead of repairing specifications. Invariant discovery tools [22], [23], [24], [25], [26], [27], [28], [29] can generate valid properties, but they may not directly relate to a user's intended properties.

V. Conclusion

We presented a novel specification repair technique using CEGIR and sketching that repairs hypothesized contracts of executable components. Results show that, in more than 32% of cases, the repaired contract matched a developer-written one, and in a further 63% of cases, the repair was relevant. In the future, we plan to further investigate this technique by trying it on various benchmarks with various complexities. Also, we plan to study the effect of the sketch, as well as the effect of repairing a poor specification on the generated repair.

VI. Acknowledgment

The research described in this paper has been supported in part by the National Science Foundation under grant 1563920.

REFERENCES

[1] D. Arney, R. Jetley, P. Jones, I. Lee, and O. Sokolsky, "Formal methods based development of a PCA infusion pump reference model: Generic infusion pump (GIP) project," in *2007 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability (HCMDSS-MDPnP)*, 2007, pp. 23–33.

[2] M. Whalen, D. Cofer, S. Miller, B. H. Krogh, and W. Storm, "Integration of formal analysis into a model-based software development process," in *Formal Methods for Industrial Critical Systems*, S. Leue and P. Merino, Eds. Springer Berlin Heidelberg, 2008, pp. 68–84.

[3] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: Association for Computing Machinery, 2006, p. 404–415. [Online]. Available: https://doi.org/10.1145/1168857.1168907

[4] A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani, "The JKind model checker," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 20–27.

[5] "Lustre," http://www-verimag.imag.fr/The-Lustre-Programming-Language-and, 2020 (accessed May 8, 2020).

[6] SAE International, "Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment," SAE International, Tech. Rep. ARP4761, Dec. 1996. [Online]. Available: https://www.sae.org/standards/content/arp4761/

[7] N. Leveson, C. Wilkinson, C. Fleming, J. Thomas, and I. Tracy, "A comparison of STPA and the ARP 4761 safety assessment process," MIT PSAS, Tech. Rep., 2014. [Online]. Available: http://sunnyday.mit.edu/papers/ARP4761-Comparison-Report-final-1.pdf

[8] R. Qiu *et al.*, "SynergiSE: Using test ranges to improve symbolic execution," [Online; accessed 30-July-2021]. [Online]. Available: https://userweb.cs.txstate.edu/~g_y10/synergise/

[9] P. E. Black, V. Okun, and Y. Yesha, "Mutation operators for specifications," in *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, 2000, pp. 81–88.

[10] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008, pp. 337–340.

[11] "Z3," https://rise4fun.com/z3/tutorial, 2020 (accessed June 5, 2020).

[12] S. Maoz, J. O. Ringert, and R. Shalom, "Symbolic repairs for GR(1) specifications," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 1016–1026. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00106

[13] R. Alur, S. Moarref, and U. Topcu, "Counter-strategy guided refinement of gr(1) temporal logic specifications," in *2013 Formal Methods in Computer-Aided Design*, 2013, pp. 26–33.

[14] D. G. Cavezza and D. Alrajeh, "Interpolation-based gr(1) assumptions refinement," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Legay and T. Margaria, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 281–297.

[15] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011)*, 2011, pp. 43–50.

[16] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 772–781.

[17] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *Computer Aided Verification*, K. Etessami and S. K. Rajamani, Eds. Springer Berlin Heidelberg, 2005, pp. 226–238.

[18] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 166–178. [Online]. Available: https://doi.org/10.1145/2786805.2786811

[19] M. Martinez and M. Monperrus, "ASTOR: A program repair library for Java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 441–444. [Online]. Available: https://doi.org/10.1145/2931037.2948705

[20] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Sketchfix: A tool for automated program repair approach using lazy candidate generation," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 888–891. [Online]. Available: https://doi.org/10.1145/3236024.3264600

[21] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 129–139.

[22] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1–3, p. 35–45, Dec. 2007. [Online]. Available: https://doi.org/10.1016/j.scico.2007.01.015

[23] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "DIG: A dynamic invariant generator for polynomial and array invariants," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, Sep. 2014. [Online]. Available: https://doi.org/10.1145/2556782

[24] S. Padhi, R. Sharma, and T. Millstein, "Data-driven precondition inference with learned features," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 42–56. [Online]. Available: https://doi.org/10.1145/2908080.2908099

[25] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid, "Feedback-driven dynamic invariant discovery," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*. New York, NY, USA: Association for Computing Machinery, 2014, p. 362–372. [Online]. Available: https://doi.org/10.1145/2610384.2610389

[26] T. Nguyen, T. Antonopoulos, A. Ruef, and M. Hicks, "Counterexample-guided approach to finding numerical invariants," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 605–615. [Online]. Available: https://doi.org/10.1145/3106237.3106281

[27] T. Nguyen, M. B. Dwyer, and W. Visser, "Syminfer: Inferring program invariants using symbolic states," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, 2017, p. 804–814.

[28] C. Ackermann, R. Cleaveland, S. Huang, A. Ray, C. Shelton, and E. Latronico, "Automatic requirement extraction from test cases," in *Runtime Verification*, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–15.

[29] C. Schulze and R. Cleaveland, "Improving invariant mining via static analysis," *ACM Trans. Embed. Comput. Syst.*, vol. 16, p. 167:1–167:20, 2017. [Online]. Available: http://doi.acm.org/10.1145/3126504