Assume, Guarantee or Repair

Student submission

Hadar Frenkel[1] ⋆ ⋆⋆ , Orna Grumberg[1], Corina Pasareanu[2] and Sarai Sheinvald[3]

[1] Department of Computer Science, The Technion, Israel
[2] Carnegie Mellon University and NASA Ames Research Center, CA, USA
[3] Department of Software Engineering, ORT Braude College, Israel

Abstract. We present Assume-Guarantee-Repair (AGR) – a novel framework which not only verifies that a program is error-free, but also *repairs* the program, in case the verification fails. We consider simple C-like programs, extended with synchronous communication actions over communication channels.
Our method uses compositional approach to modularly check the system for errors, and to repair it. We fulfill the two tasks simultaneously: in every iteration of the procedure, we either make another step towards proving safety of the (current) system, or remove the current vulnerability in a way that brings it closer to being safe. We manage to handle infinite-state systems by using a finite abstract representation. We describe our method and demonstrate the effectiveness of AGR on several examples using existing SMT solvers, learning, and reachability analysis tools.

1 Introduction

For large-scale systems, verifying that a system is error-free is a main challenge in the field of formal verification. Often, the verification process of such a system does not scale well. *Compositional verification* aims to verify small components of a system separately, and from the safety of the individual components, to conclude the safety of the entire system. This, however, is not always possible, since the safety of a component often depends on the behavior of its environment.

The Assume-Guarantee (AG) style compositional verification [5,8] suggests a solution to this problem. The simplest AG rule checks if a system composed of components M_1 and M_2 fulfills a safety requirement P by checking that M_1 under assumption A fulfills P, and that any system containing M_2 as a component fulfills the safety assumption A. Several frameworks have been proposed to support this style of reasoning. Finding a suitable assumption A is then a common challenge in such frameworks.

In this work, we present *Assume-Guarantee-Repair* (AGR) – a fully automated framework which applies the Assume-Guarantee rule, and while seeking a suitable assumption A, incrementally repairs the given program in case a vulnerability is found. Our framework is inspired by [6], which presented a learning-based method to finding an assumption A, using the L^* [1] algorithm for learning regular languages.

Our AGR framework handles *communicating programs*. These are infinite-state C-like programs, extended with the ability to synchronously read and write messages over communication channels. Sending messages over common channels as well as the ability of local computations, make communicating programs a good model for security protocols. We

⋆ **This research was partially supported by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel National Cyber Directorate.**

⋆⋆ Contact information: hfrenkel@cs.technion.ac.il, +972-4-829-4315

model such programs as finite-state automata, similar to a representation as a control-flow graph. Its advantage, however, is in the ability to exploit an automata-learning algorithm such as L^*.

The L^* algorithm aims at learning a (potentially unknown) regular language U. Its entities consist of a *teacher* – an oracle who answers *membership queries* ("is the word w in U?") and *equivalence queries* ("is $\mathcal{A}$ an automaton whose language is U?"), and a *learner*, who iteratively constructs a finite deterministic automaton $\mathcal{A}$ for U by submitting a sequence of membership and equivalence queries to the teacher.

In using the L^* algorithm for learning an assumption A for the AG-rule, membership queries are answered according to the satisfaction of the safety requirement P: If M_1 composed with t is safe, then the trace[4] t in hand should be in the assumption A. Otherwise, t should not be in A. Once the learner constructs a stable system A, it submits an equivalence query. The teacher then checks whether A is a suitable assumption, that is, whether M_1 composed with A is safe according to P, and whether the language of M_2 is contained in the language of A. The learning procedure aims at learning the weakest assumption A_w, which contains all the traces that composed with M_1 fulfill P . The key observation that guarantees termination is that the components in this procedure – M_1,M_2, P and A_w – are all regular.

Our setting is more complicated than the usual, since the traces in the components contain constraints over program variables, which are to be checked semantically and not syntactically. Moreover, A_w above may no longer be regular.

Our method manages to overcome this problem in a way that still guarantees termination in case the system is safe, and progress towards safety, otherwise.

As we have described above, our goal is not only to compositionaly prove that a system is error-free, but also to remove vulnerabilities in case they exist. An AG-rule can either conclude that the system is safe, or return a counterexample in the form of a trace that contains a vulnerability of the system. In our case, instead of returning the counterexample, we repair M_2 in a way that eliminates this vulnerability. In order to do so, we infer new constraints on the system, in a process called *abduction* [7]. We add the learned constraints to the set of actions of M_2 through the learning process to eliminate the erroneous trace. We then return to the verification stage and try to prove that the repaired program is safe, and so on.

Thus, AGR operates in a verify-repair loop, where each iteration runs a learning-based process to determine whether the (current) system is safe according to the property P, and if not, eliminates errors from M_2 while enriching the set of constraints derived from these errors, which often leads to quicker convergence.

Implementation To demonstrate the effectiveness of AGR, we implemented our AGR framework and ran it on 16 examples designed to exercise different aspects of the approach. We used Spacer [3] as a model checker to answer verification questions, and Z3 [2] to infer new constraints by abduction. We also integrated L^* implementation from the LTSA tool [4] in our AGR framework. The experiments provide proof of concept to our algorithm. In the future we plan to robustify our implementation and to evaluate it extensively in the context of autonomous systems and security protocols.

[4] We refer to program behaviors as traces, as they are represented by traces of the automaton.

2 Example

We now present an example to demonstrate the process. Figure 1 presents a simple program, M_2, which reads a password as long as it has less than four digits. Once the password is long enough, the program encrypts the password and sends it through a communication channel. The figure demonstrates the program and its representation as an automaton.

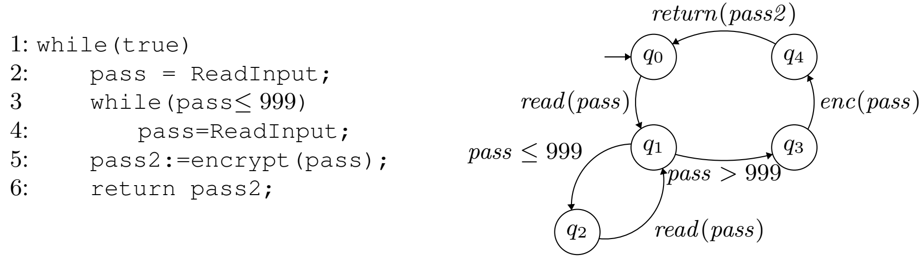

Fig. 1: Modeling a communicating program as an automaton M_2

As mentioned in Section 1, we use a compositional approach in order to handle large systems. In Figure 2 we present the environment of M_2, namely M_1, which preforms the encryption of the password read by M_2. The components M_1 and M_2 synchronize over common channels, i.e., *enc* and *return*. The channel *enc* is used to communicate the value of the password from M_2 to M_1, while the channel *return* is used to return the value of the encrypted password (i.e. *pass2*) from the whole program.

Since we use automata learning in order to detect errors and repair them, the safety property is too given as an automaton. The property P presented in Figure 2 requires that if the password $pass$ was entered and $pass2$ is the password returned after encryption, then $pass \neq pass2$. That is, the original password is not exposed. The property also requires the encrypted password to be of at most 64 bits to avoid overflow. Nothing in the composition of M_1 and M_2 enforces the password to be of at most 64 bits, and thus the systems does not satisfy the property. A possible violation is the concrete trace $t = read(2^{63}), 2^{63} > 999, encrypt(2^{63}), pass2 = 2^{63} \cdot 2, return(2^{64})$. The trace t violates the property P since *pass2* overflows with 64 bits. Therefore, we wish to eliminate t. We then use logical abduction in order to learn the new constraint $pass < 2^{63}$ and locate it in M_2 in a way that ensures that for the encrypted password it holds that $pass2 < 2^{64}$.

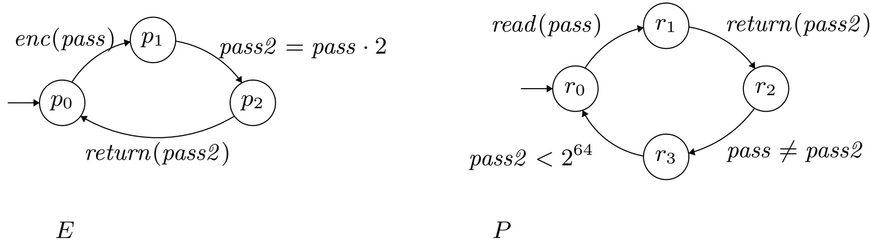

Fig. 2: The system environment E and the property P given as automata

References

1. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2).
2. L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.
3. A. Komuravelli, A. Gurfinkel, and S. Chaki. Smt-based model checking for recursive programs. In *CAV*, 2014.
4. J. Magee and J. Kramer. *Concurrency - state models and Java programs*. Wiley, 1999.
5. J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.
6. C. S. Pasareanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 2008.
7. C. Peirce and C. Hartshorne. *Collected Papers of Charles Sanders Peirce*. Belknap Press, 1932.
8. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems, NATO ASI Series*, 1985.