Code Mutation in Verification and Automatic Code Correction

Gal Katz and Doron Peled

Department of Computer Science, Bar Ilan University
Ramat Gan 52900, Israel

Abstract. Model checking can be applied to finite state systems in order to find counterexamples showing that they do not satisfy their specification. This was generalized to handle parametric systems under some given constraints, usually using some inductive argument. However, even in the restricted cases where these parametric methods apply, the assumption is usually of a simple fixed architecture, e.g., a ring. We consider the case of nontrivial architectures for communication protocols, for example, achieving a multiparty interaction between arbitrary subsets of processes. In this case, an error may manifest itself only under some particular architectures and interactions, and under some specific values of parameters. We apply here our model checking based genetic programming approach for achieving a dual task: finding an instance of a protocol which is suspicious of being bogus, and automatically correcting the error. The synthesis tool we constructed is capable of generating various mutations of the code. Moving between them is guided by model checking analysis. In the case of searching for errors, we mutate only the architecture and related parameters, and in the case of fixing the error, we mutate the code further in order to search for a corrected version. As a running example, we use a realistic nontrivial protocol for multiparty interaction. This protocol, published in a conference and a journal, is used as a building block for various systems. Our analysis shows this protocol to be, as we suspected, erroneous; specifically, the protocol can reach a livelock situation, where some processes do not progress towards achieving their interactions. As a side effect of our experiment, we provide a correction for this important protocol obtained through our genetic process.

1 Introduction

Model checking is a successful technique for comparing a model of a system with some formal specification. One of its limitations is that of state space explosion. This is combated by many techniques that avoid the simple enumeration of all the reachable states. Since model checking of concurrent systems is intractable, this is a very challengeable problem, with many interesting heuristics. Another limitation of model checking is that the method is mainly restricted to finite state systems, while the verification of infinite state systems is, in general, undecidable. The problem of synthesizing correct code or attempting to automatically correct errors is considered to be even harder than model checking. In some limited cases where this was shown to be decidable, the complexity was considerably higher than simple model checking.

In recent papers [8,9,10] we demonstrated the approach of combining model checking and genetic programming for the synthesis of correct-by-design programs. This is in particular effective for the automatic generation of code that is hard to program manually. Examples are mutual exclusion problems and various concurrent synchronization problems. In this paper we exploit a related technique, and extend the tool we developed, to assist the programming throughout the code development process. Specifically, we use the ability of mutating code, guided by ranking that is based on model checking, to find errors in some complicated parametric protocol, and, moreover, to correct the errors.

The first main challenge that we tackle here is to check communication protocols that are not limited to a particular number of processes or communication architecture. Although each instance of the protocol is a finite state system, this is a parametric problem, which means that in general, its verification is undecidable [1]. Furthermore, this protocol is not limited to a particular simple communication pattern or network topology (such as in [4]). We thus use code mutation to generate instances of the protocol that we want to check. Since there are several parameters that vary with the code (the number of processes, the communication network, etc.) it is not simple to detect an instance that would manifest the error, even if we suspect there exists one. We seek an alternative to a simple enumeration of the instances of such a protocol. Since the communication architecture is not fixed, the enumeration can easily progress in a direction that will not reveal the existence of an error (e.g., focusing on a particular architecture such as a ring and just extending the number of processes). We thus apply some ranking on the checked instances, based on the model checking results, in order to help direct the search in the space of syntactically constrained programs towards an instance with an error. Then, when the error is revealed, we apply similar techniques to help us correct the code.

In essence, we apply model checking techniques for finite state systems on instances of the code, using the genetic programming approach as a heuristic method to move between different variations of it, first to find the error, then to find a correction. The mutation at the search for errors is limited to the communication architecture and various related parameters. After finding an erroneous instance of the protocol in this way, we reverse the search and allow mutating the code in order to correct it. In this latter case, mutation is allowed on the protocol itself, rather than the architecture. Correcting the code of a parametric protocol is also challenging. When a new candidate (mutation) for the protocol is suggested, it is again impossible to apply a decision procedure to check its correctness. Thus, we have again to check each candidate against various architectures, a problem that is similar to the one we are faced with when trying to find the error in the original protocol. We thus alternate between using mutations for generating new candidate protocols, and using mutations for generating instances for model checking the suggested corrections. However, there is some learning process here, as architectures that were shown to create counterexamples for either the original algorithm or tested mutations can be used for the model checking of subsequent candidates. This alternation between the evolution of architectures and programs code is repeated, gradually adding new architectures against which the candidates for corrected code need to be checked. This process continues until a program that cannot be refuted by the tool is found.

Our running example in this paper is an actual protocol for coordinating multiprocess interaction in a distributed system, named *α-core* [13]. This is quite a nontrivial protocol, which extends the already difficult problem of achieving multiprocess communication in the presence of nondeterminism both on the sender and receiver side. The protocol we check appears both in a conference and in a followup journal paper and is of practical use. Reading the paper, we were suspicious of the correctness of the protocol, but due to its complexity and, in particular, its multiple architecture nature, could not pinpoint the problem with a manually constructed example. Except for the actual debugging of the protocol, we followed some remarks made by the authors of that protocol in the original paper [13] about subtle points in the design of the protocol, and have let our tool discover these problems automatically. Thus, we believe that a framework that performs a genetic model checking driven mutation, as we developed and used here, is very effective as an interactive tool in the process of protocol design.

2 Background

2.1 Model Checking Guided Genetic Programming

Genetic programming [11] is a program synthesis technique based on a search in the state space of syntactically constrained programs. The search is guided by providing fitness to the generated candidate programs, usually based on test cases, but recently also on model checking results [7,8,9].

The genetic search begins with the construction of some random candidate programs (typically, a few hundreds candidates). Then one iterates the following steps. Some candidates Γ are selected at random, then they are syntactically mutated by either erasing, adding or replacing program segments. The code is often represented using a tree, and the mutation operations may also involve adding some code, when the syntax requires it (e.g., just removing a test in a program may result in syntactically erroneous code, hence the missing construct is generated at random). A powerful operator that combines the code of several candidates, called *crossover*[1] may also be used, although there are also some arguments against its utilization. The mutation (and crossover) operations generate some set of candidates Δ. Fitness of the candidates $\Gamma \cup \Delta$ is calculated, trying to rank their potential to further evolve towards correct code. Traditionally, fitness is calculated in genetic programming by checking some test suite. In contrast, we use model checking for calculating this value. Then, instead of the old $|\Gamma|$ candidates selected, we return for the next iteration the $|\Gamma|$ candidates with highest fitness value among $\Gamma \cup \Delta$.

[1] In our work, we did not implement the crossover operation.

The iterative process stops either when a candidate that achieves a fitness value above some level is found, or some limit on the number of iterations expires. In the former case, provided that the fitness is well crafted, the candidate found has a good potential to be a solution of the synthesis problem. In the latter case, the search may restart with some new random candidates.

Model checking based fitness was introduced independently by Johnson [7] and by us [8]. In [7], fitness value reflected the number of temporal properties that were satisfied by the checked candidate. We [8] suggested a finer measure of fitness, with more levels per each property: (1) none of the executions satisfy the property, (2) some but not all the executions satisfy the property, (3) each prefix of an execution that does not satisfy the property can be continued into an execution that satisfies it (hence in order to not satisfy the property, infinitely many bad choices must be made) and (4) all the executions satisfy the property. This calls for a deeper model checking algorithm than the standard one [9,12].

In [10] we synthesized solutions for the leader election problem. Since this problem is parametric, properties were checked against all of the problem instances up to a predefined cutoff value, and the ranking depended on whether none, some or all of the instances satisfied the properties. We also used there aggressive partial order reduction to speed up the model checking.

In all cases, a secondary "parsimony" measure was added to the fitness function in order to encourage the generation of shorter and hopefully efficient programs. Correcting programs can follow a similar search as described above, by starting from the bogus version of the code rather than with randomly generated programs.

2.2 The α-Core Protocol

The α-core protocol is developed to schedule multiprocess interaction. It generalizes protocols for handshake communication between pairs of processes. For each multiprocess interaction, there is a dedicated coordinator on a separate process. To appreciate the difficulty of designing such a protocol, recall for example the fact that the language CSP of Hoare [5] included initially an asymmetric construct for synchronous communication; a process could choose between various incoming messages, but had to commit on a particular send. This was important to achieve a simple implementation. Otherwise, one needs to consider the situation in which after communication becomes possible between processes, one of them may already continue to perform an alternative choice. Later Hoare removed this constraint from CSP. The same constraint appears in the asymmetric communication construct of the programming language ADA. The Buckley and Silberschatz protocol [3] solves this problem for the symmetric case in synchronous communication between pairs of processes, where both sends and receives may have choices. Their protocol uses asynchronous message passing between the processes to implement the synchronous message passing construct. The α-core protocol, also based on asynchronous message passing, is more general, and uses coordinator processes to allow synchronization among any number of processes.

The α-core protocol includes the following messages sent from a participant to a coordinator:

PARTICIPATE. A participant is interested in a single particular interaction (hence it can commit on it), and notifies the related coordinator.

OFFER. A participant is interested in one out of several potentially available interactions (a nondeterministic choice).

OK. Sent as a response to a **LOCK** message from a coordinator (described below) to notify that the participant is willing to commit on the interaction.

REFUSE. A participant decides it does not want to commit on an interaction it has previously applied to, and notifies the coordinator. This message can also be sent as a respond to a **LOCK** message from the coordinator.

Messages from coordinators are as follows:

LOCK. A message sent from a coordinator to a participant that has sent an **OFFER.** requesting the participant to commit on the interaction.

UNLOCK. A message sent from a coordinator to a locked participant, indicating that the current interaction is canceled.

START. Notifying a participant that it can start the interaction.

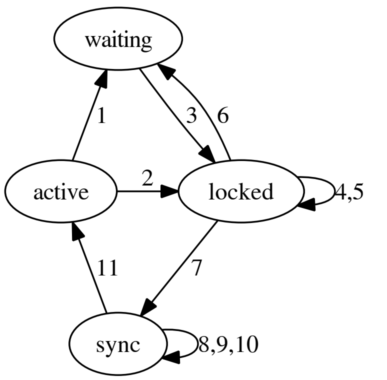
ACKREF. Acknowledging a participant about the receipt of a **REFUSE** message.

Fig. 1(a) describes the extended state machine of a participant. Each participant process keeps some local variables and constants:

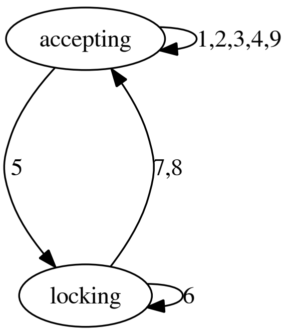
IS	a set of coordinators for the interactions the participant is interested in.
locks	a set of coordinators that have sent a pending **LOCK** message.
unlocks	a set of coordinators from which a pending **UNLOCK** message was received.
locker	the coordinator that is currently considered.
n	the number of **ACKREF** messages required to be received from coordinators until a new coordination can start.
α	the coordinators that were asked for interactions but were subsequently refused.

The actions according to the transitions are written as a pair $en \rightarrow action$, where *en* is the condition to execute the transition, which may include a test of the local variables, a message that arrives, or both of them (then the test should hold *and* the message must arrive). The action is a sequence of statements, executed when the condition holds. in addition, each transition is enabled from some state, and upon execution changes the state according to the related extended finite state machine. The participant's transitions, according to the numbering of Fig. 1(a) are:

1. $|IS > 1| \rightarrow \{$ foreach $p \in IS$ do $p!\mathbf{OFFER}\}$
2. $|IS = 1| \rightarrow \{$ $locker{:=}p$, were $IS = \{p\}$; $locker!\mathbf{PARTICIPATE}$; $locks$, $unlocks{:=}\emptyset\}$
3. $p?\mathbf{LOCK} \rightarrow \{locker{:=}p$; $locks$, $unlocks{:=}\emptyset$; $p!\mathbf{OK}\ \}$
4. $p?\mathbf{LOCK} \rightarrow \{locks{:=}locks \cup \{sender\}\}$



(a) Participant



(b) Coordinator

Fig. 1. State machines

5. $\mathit{locks} \neq \emptyset \wedge p?\textbf{UNLOCK} \rightarrow \{\mathit{locker}{:=}q$ for some $q \in \mathit{locks}$; $q!\textbf{OK}$; $\mathit{locks}{:=}\mathit{locks} \setminus \{q\}$; $\mathit{unlocks}{:=}\mathit{unlocks} \cup \{p\}\}$
6. $\mathit{locks} = \emptyset \wedge p?\textbf{UNLOCK} \rightarrow \{$ foreach $q \in \mathit{unlocks} \cup \{p\}$ do $q!\textbf{OFFER}\}$
7. $p?\textbf{START} \rightarrow \{\alpha{:=}\mathit{IS} \setminus \mathit{unlocks} \setminus \{\mathit{locker}\}$; foreach $q \in \alpha$ do $q!\textbf{REFUSE}$; $n := |\alpha|$; start participating in the joint action managed by $\mathit{locker}\}$
8. $p?\textbf{LOCK} \rightarrow \{\}$
9. $p?\textbf{UNLOCK} \rightarrow \{\}$
10. $p?\textbf{ACKREF} \rightarrow \{n{:=}n-1\}$
11. $n = 0 \rightarrow \{$ Let IS be the new set of interactions required from the current state. $\}$

For a coordinator, whose extended finite state machine appears in Fig. 1(b), we have the variables *waiting*, *locked*, *shared* and α, holding each a set of processes, and n is a counter for the number of processes that indicated their wish to participate in the interaction. The constant C holds the number of processes that need to participate in the interaction (called, the *cardinality* of the interaction), and the variable *current* is the participant the coordinator is trying to lock. The transitions, according to their numbering from Fig. 1(b) are as follows:

1. $n < C \wedge p?\textbf{OFFER} \rightarrow \{\, n{:=}n+1$; $\mathit{shared}{:=}\ \mathit{shared} \cup \{p\}\ \}$
2. $n < C \wedge p?\textbf{PARTICIPATE} \rightarrow \{\, n{:=}n+1$; $\mathit{locked}{:=}\ \mathit{locked} \cup \{p\}\ \}$
3. $p?\textbf{REFUSE} \rightarrow \{$ if $n > 0$ then $n{:=}n-1$; $p!\textbf{ACKREF}$; $\mathit{shared}{:=}\mathit{shared} \setminus \{p\}\}$
4. $n = C \wedge \mathit{shared} = \emptyset \rightarrow \{$ foreach $q \in \mathit{locked}$ do $q!\textbf{START}$; $\mathit{locked}, \mathit{shared}{:=}\emptyset$; $n{:=}0\}$
5. $n = C \wedge \mathit{shared} \neq \emptyset \rightarrow \{\mathit{current}{:=}\ \min(\mathit{shared})$; $\mathit{waiting}{:=}\mathit{shared} \setminus \{\mathit{current}\}$; $\mathit{current}!\textbf{LOCK}\}$
6. $\mathit{waiting} \neq \emptyset \wedge p?\textbf{OK} \rightarrow \{\mathit{locked}{:=}\mathit{locked} \cup \{\mathit{current}\}$; $\mathit{current}{:=}\min(\mathit{waiting})$; $\mathit{waiting}{:=}\mathit{waiting} \setminus \{\mathit{current}\ \}$; $\mathit{current}!\textbf{LOCK}\}$
7. $\mathit{waiting} = \emptyset \wedge p?\textbf{OK} \rightarrow \{\mathit{locked}{:=}\mathit{locked} \cup \{\mathit{current}\}$; foreach q in locked do $q!\textbf{START}$; $\mathit{locked}, \mathit{waiting}, \mathit{shared}{:=}\emptyset$; $n{:=}0\}$
8. $p?\textbf{REFUSE} \rightarrow \{\alpha{:=}(\mathit{locked} \cap \mathit{shared}) \cup \{\mathit{current}, p\}$; foreach $q \in \alpha \setminus \{p\}$ do $q!\textbf{UNLOCK}$; $p!\textbf{ACKREF}$; $\mathit{shared}{:=}\mathit{shared} \setminus \alpha$; $\mathit{locked}{:=}\mathit{locked} \setminus \alpha$; $n{:=}n - |\alpha|\}$
9. $p?\textbf{OK} \rightarrow \{\}$

As with other concurrency coordination constructs, such as semaphores, the irresponsible use of the coordination achieved by the α-core protocol can result in deadlock situation (when processes attempt to get into conflicting coordinations in incompatible order). What the α-core protocol correctness is prescribed to guarantee is that if some processes are all interested in some coordination, then it, or some alternative coordination for at least one of the participant processes will eventually occur. As we will show later, this property does not really hold for this protocol.

3 Evolution of Architectures

The α-core algorithm is parametric, and should work for a family of architectures, where each architecture consists of a set of participants, a set of coordinators, and a particular connectivity between processes of the two kinds. Additional configuration parameters, such as buffer sizes, can be instantiated as well. When verifying the algorithm, we cannot simply perform model checking for all of the architectures up to some size. One reason for that is the large number of possible architectures. Another reason is the high model checking complexity for such a nontrivial protocol, which requires a considerable amount of time and memory, even after some reduction techniques (such as partial order reduction, and coarser atomicity) are used.

Instead, we introduce a new method for the evolution of architectures by genetic programming. The idea is to randomly generate portions of code representing various architectures, each being a basis for a distinct instance of the protocol. Then we gradually evolve these instances and improve them, until a good solution is found. A "good solution" in this context, is an instance of the protocol on which a counterexample for the given algorithm can be found. Thus, our goal at this point is the reverse of the conventional goal of genetic programming, where a good solution is a correct one.

3.1 Architecture Representation

A first step towards our goal is achieving the ability to represent architectures as portions of code. This is done by creating a dedicated initialization process (called *init*), and basing this process on code instructions and building blocks that can let it dynamically generate any relevant architecture. Depending on the problem we try to solve, these building blocks may allow the dynamic creation of processes of various types, and the instantiation of global and local parameters related to the created processes.

Considering our example, we observe that the α-core protocol involves an arbitrary number of processes of two types: participating processes, and coordinating processes. A coordinating process can be responsible for coordinating any given subset of the participating processes, and a participating process may, at each state, interact with any number of coordinators for coordinations it can be involved in. It is even possible that there are several coordinating processes that try to coordinate the same sets of processes. Fig. 2 presents an architecture

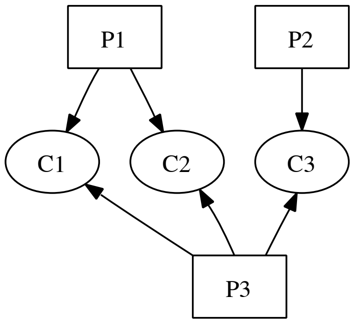


Fig. 2. An architecture with three participants and three coordinators

where there are three participating processes: $P1 - P3$, with three coordinators: $C1 - C3$. An edge appears between a participating process and a process that coordinates it. $P2$ is interested in a single interaction (handled by $C3$), while $P1$ and $P3$ are involved in multiple interactions.

To enable the generation of such architectures, the following building block are provided for the *init* process:

`CreateProc(proctype)` - Dynamically creates a new process of type `proctype`.
`Participant`, `Coordinator` - constants of type `proctype` representing participant and coordinator process types.
`Connect(part_proc_id, coord_proc_id)` - connects a particular participant and a coordinator whose process ids are given in the operands. The connection involves updating *IS* - the set of coordinators that the participant locally stores, and *C* - the cardinality of the coordinator.

A complete program representation includes the code for the *init* process, as well as skeleton code for each other type of process. At runtime, the *init* process is executed first, and according to its instructions, other processes are dynamically created and instantiated. This is similar to the way Promela code is written and executed in the Spin model checker [6], where the *init* process can dynamically create other processes. When searching for the goal architecture, a permanent code is given for each of the process types, where the genetic process is allowed to randomly generate and alter only the code of the *init* process. Using the above building blocks, the code for the architecture of Fig. 2 can look as follows.

```
CreateProc(Participant)      |      Connect(1, 4)
CreateProc(Participant)      |      Connect(1, 5)
CreateProc(Participant)      |      Connect(2, 6)
CreateProc(Coordinator)      |      Connect(3, 4)
CreateProc(Coordinator)      |      Connect(3, 5)
CreateProc(Coordinator)      |      Connect(3, 6)
```

The number of allowed processes, and accordingly, the range for the process ids are bounded. As usual, the fitness function is also based on a secondary parsimony measure which prefers shorter programs. This often leads to the removal of redundant lines from the code, and to the generation of simpler architectures and counterexamples.

3.2 The Fitness Function

The evolution of architectures is based on a fitness function which gives score to each program (consisting of architecture and other processes code). Like in our previous work, here too the fitness values are based on model checking of the given specification. However the function is different. Since the goal is to fail the given program, satisfying the negation of the specification increases the fitness rather than decreases it. This means that it is sufficient to find an architecture which violates *one* of the given properties, and since we are dealing with LTL properties, we only need to find a single execution path that violates a property. Thus (unlike in our previous work) there is no use in considering the amount of satisfied paths, or the need to make a distinction between the level of falsifying the specification (see Section 2.1) when ranking the candidate solutions. Instead, we apply a different method, and try to split properties into smaller building blocks (whenever possible) whose satisfaction may serve as an intermediate step in satisfying an entire property.

As an example, consider the LTL property $\varphi = \Box(P \rightarrow Q)$. We showed in [9] how programs satisfying φ can be evolved. However, we are now interested in progressing towards an architecture *violating* φ. We first negate it, obtaining $\neg\varphi = \Diamond(P \wedge \neg Q)$. Then, we can give intermediate ranking to solutions that contain a path satisfying $\Diamond P$, hoping that they finally evolve to solutions with a path satisfying the entire $\neg\varphi$. This method can be particularly useful in a common case where Q is an assertion about states, and P denotes the location in the code where Q must hold. That is, $P = at(\ell)$ for some label ℓ in the code. Then, when trying to violate the property φ, we can give higher fitness to programs that at least reach the location ℓ, and then give the highest fitness value if the state property Q is violated when reaching ℓ.

During the development of protocols, the developers often look at some intricate possible behaviors of the protocol. The kind of guided search suggested above, by mutating the architecture, can also be used, besides for finding errors in protocols, for finding such scenarios and documenting them. We first demonstrate this by two nontrivial transitions on the code of the α-core participant code, which due to its authors, stem from some intricacies (in Section 5.1 we will show how we used our method for finding a real error in the α-core protocol).

While being in the "sync" state, a participant usually receives **ACKREF** messages (transition 10 in Fig. 1(a)), but it can accidentally receive either a **LOCK** or **UNLOCK** messages (transitions 8 and 9 respectively), which it has to ignore. In order to verify that, we added assertions for the participant's code claiming that the received message under the "sync" state must be **ACKREF**, and then activated the tool in order to find architectures which refute this assertion. Within seconds, the two architectures depicted in Fig. 3 were generated. The architecture on the left is related to an example involving the **LOCK** message, and is simpler than the one presented in [13] (although coordinators with

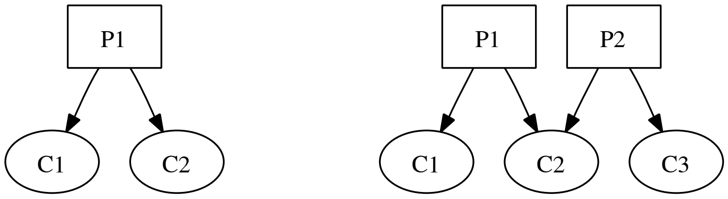


Fig. 3. Architectures found for intricacies with **LOCK** (left) and **UNLOCK** (right)

only one process are not so useful). For the case of the **UNLOCK** message, an identical architecture to the one described in [13] was found.

4 Co-evolution of Programs and Counterexamples

After finding a "wrong" architecture for a program, our next goal is to reverse the genetic programming direction, and try to automatically correct the program, where a "correct" program at this step, is one that has passed model checking against the architecture. Yet, correcting the program for the wrong architecture only, does not guarantee its correctness under different architectures. Therefore, we introduce a new algorithm (see Algorithm 1) which co-evolves both the candidate solution programs, and the architectures that might serve as counterexamples for those programs.

Algorithm 1: Model checking based co-evolution

```
MC-CoEvolution(initialProg, spec, maxArchs)
(1)     prog := initialProg
(2)     archList := ∅
(3)     while |archList| < maxArchs
(4)         arch := EvolveArch(prog, spec)
(5)         if arch = null
(6)             return true // prog stores a "good" program
(7)         else
(8)             add arch to archlist
(9)         prog := EvolveProg(archlist, spec)
(10)        if prog is null
(11)            return false // no "good" program was found
(12)    return false // can't add more architectures
```

The algorithm starts with an initial program *initProg*. This can be the existing program that needs to be corrected, or, in case that we want to synthesize the code from scratch, an initial randomly generated program. It is also given a specification *spec* which the program to be corrected or generated should satisfy. The algorithm then proceeds in two steps. First (lines (4) – (8)), the *EvolveArch* function is called. The goal of this function is to generate an architecture on which the specification *spec* will not hold. If no such architecture is found, the *EvolveArch* procedure returns *null*, and we assume (though we cannot guarantee) that the program is correct, and the algorithm terminates. Otherwise, the found architecture *arch* is added to the architecture list *archList*, and the algorithm proceeds to the second step (lines (9) – (11)).

In this step, the architecture list and the specification are sent to the *Evolve-Prog* function which tries to generate programs which satisfy the specification under *all* of the architectures on the list. If the function fails, then the algorithm terminates without success. Since the above function runs a genetic programming process which is probabilistic, instead of terminating the algorithm, it is always possible to increase the number of iterations, or to re-run the function so a new search is initiated. If a correct program is found, the algorithm returns to the first step at line (4), on which the newly generated program is tested. At each iteration of the *while* loop, a new architecture is added to the list. This method' serves two purposes. First, once a program was suggested, and refuted by a new architecture, it will not be suggested again. Second, architectures which were complex enough to fail programs at previous iterations, are good candidates to do so on future iterations as well. The allowed size of the list is limited in order to bound the running time of the algorithm.

Both *EvolveProg* and *EvolveArch* functions use genetic programming and model checking for the evolution of candidate solutions (each of them is equipped with relevant building blocks and syntactic rules), while the fitness function varies. For the evolution of programs, a combination of the methods proposed in [9,10] is used: for each LTL property, an initial fitness level is obtained by performing a deep model checking analysis. This is repeated for all the architectures in *archList*, which determines the final fitness value. For the evolution of the architectures, the method explained in the previous section is used.

A related approach for automatic bug fixing was suggested in [2] where programs and unit tests were co-evolved. However, that work deals with functional programs, where no model checking is needed. In addition, that work started with a set of simple data structures, representing test cases, which can then be evolved by some search algorithm. In contrast, in our work architectures are represented as variable length programs which allow greater flexibility. Moreover, we start with a single architecture, and dynamically add new ones only when necessary during the evolutionary process. In a recent work [15], locating and repairing bugs on C programs were accomplished by manually defining positive and negative test cases, and using them in the fitness function.

5 Finding and Correcting Errors in α-Core

5.1 Generation of a Violating Architecture

One weakness of the α-core algorithm is that the **REFUSE** message is used both for canceling a previous offer to participate in an interaction, and as a possible response to a **LOCK** message. This may lead to some delicate scenarios, and the authors mention that ideally, it would have been better to add another message type. However, in order to keep the algorithm simple, they refrain from doing so, and instead try to deal with the intricate situations directly. This includes performing the following action when a coordinator receives a **REFUSE** message while being in the "accepting" state according to transition 3:

$$\text{if } n > 0 \text{ then } n{:=}n - 1$$

The variable n serves as a counter for the number of active offers the coordinator currently has. If both the coordinator and one of its participants try to cancel the interaction concurrently, n may be wrongly decreased twice. The comparison to 0 is supposed to avoid the second decrease.

Reading that, we suspected that despite the above check, there may still be situations on which n is decreased twice due to a single participant refusal, thus causing n to no longer represent the correct number of active offers. In order to check that, we added the following assertion to the program of the coordinator just before receiving any message in the "accepting" state:

$$ASSERT(n = |shared| + |locked|)$$

We then applied our tool in order to dynamically search for an architecture that violates the assertion by the method described in section 3. After a short progress between various architectures, the tool found several architectures on which the assertion is indeed violated. The simplest of these architectures is shown at Fig. 4. It includes two participants denoted $P1$ and $P2$, which are both connected to two coordinators denoted $C1$ and $C2$. The message sequence chart at Fig. 5 shows the related counterexample, having the following messages (the comments on the right refer to the values of the counter n of $C2$): At messages (1)-(4) the two participants offers interactions to the two coordinators, which causes $C2$ to set its local counter n to 2. Coordinator $C1$ responses first, and successfully locks both participants (messages (5)-(8)). Coordinator $C2$ then tries too to lock $P1$ (message (9)), and its request remains pending. Then $C1$ asks the participants to start the interaction, which cause them to refuse the offers previously sent to $C2$ (messages (10)-(13)). $C2$ then cancels the interaction by sending messages (14) and (15) (and resetting n), and a new interaction is initiated by $P2$ (messages (16) and (17)), which sets n to 1.

Only then, message (11) with the refusal of $P1$ is received, and since $n > 0$ holds, n is wrongly decreased to 0, although there is an active offer by $P2$. After

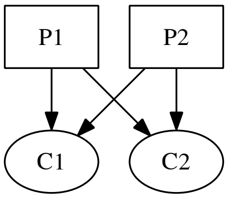


Fig. 4. An architecture violating the assertion

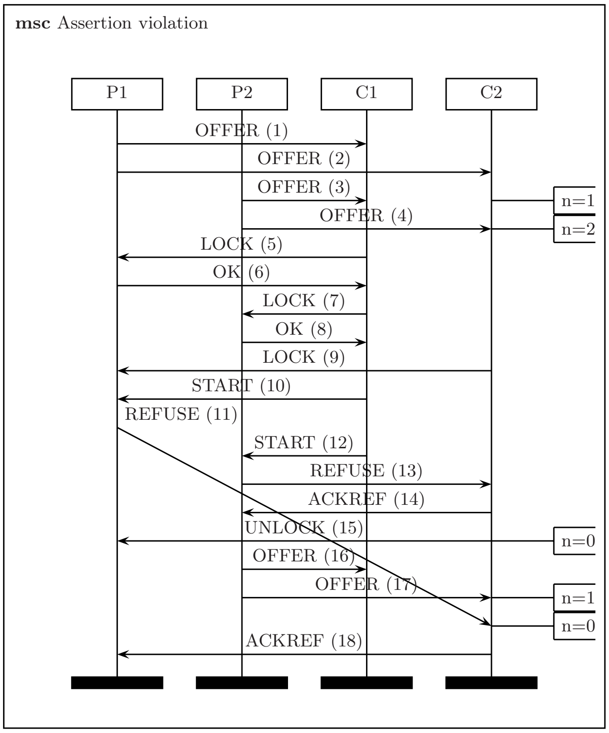


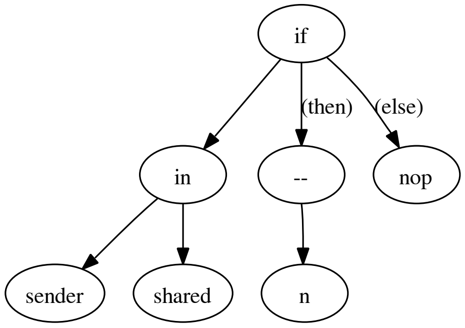
Fig. 5. A Message Sequence Chart showing the counterexample for the α-core protocol

that, if another process (such as $P1$) sends a new offer to $C2$ and no other coordinator tries to lock these participants, $C2$ will never execute the interaction (since n is smaller than its cardinality). This violate the property termed *Progress* in the α-core protocol paper [13], requiring that an enabled interaction (i.e., one in which the participating processes have requested **OFFER** or **PARTICIPATE** and did not subsequently sent a **REFUSE**) will eventually be executed. The result can be a livelock, as some of the processes are waiting for this subsequent coordination, which will not happen, or even deadlock, if this coordination is the only progress that the program is waiting for.

5.2 Generation of a Corrected Algorithm

After finding the error in the algorithm, we set our tool to automatically generate candidate programs correcting the error. The α-core code was divided into dynamic parts which the genetic process can change and evolve, and static parts which are permanent portions of the code, and remain unchanged. We set the code of the participant, and most of the code of the coordinator as static, and set as dynamic only the code that we manually identified as wrong by observing the counterexample we obtained during our search for an error phase. This is the code that deals with the **REFUSE** message. Although we could theoretically allow the dynamic evolution of the *entire* program code, the approach we took has two advantages. First, freely evolving the entire code could lead to a total change in the structure of the original algorithm, while our goal is to handle only some functional aspects of the code. Second, the search space for new code is much smaller, thus allowing a fast progress into correct solutions. Certainly, restricting the search space can make it impossible to reach a perfect solution, but in such cases, it is always possible to set more code portions as dynamic, keeping in mind the trade off between code expressibility and performance.

The tool first found a correction which holds for the architecture of Fig. 4. However, after reversing its search direction and goal, the tool discovered a new architecture on which that correction was not valid. This was followed by an alternating series of code corrections, and generation of new violating architectures (as described in Algorithm 1), until finally a simple correction was generated, without any architecture on which a violation could be found. The syntax tree that was generated for this simple correction, and its resultant code are depicted in Fig. 6.



(a) Program tree

if *sender* $\in$ *shared* then $n{:=}n-1$

(b) Program code

Fig. 6. Final generated correction

This code replaces the original handling of the **REFUSE** message in transition 3 of the coordinator. Instead of the original code comparing n to 0, this code decreases n only if the sender participant belongs to the *shared* list. This indeed seems to solve the previous error, since after the first decrease of n, the sender is removed from the *shared* list, thus avoiding a second redundant decrease.

6 Conclusions

In this work we suggested the use of a methodology and a tool that perform a search among versions of a program by code mutation, guided by model checking results. Code mutation is basically the kernel of genetic programming. Here it is used both for finding an error in a rather complicated protocol, *and* for the correction of this same protocol. Although several methods were suggested for the verification of parametric systems, the problem is undecidable, and in the few methods that promise termination of the verification, quite severe restrictions are required. Although our method does not guarantee termination, neither for finding the error, nor for finding a correct version of the algorithm, it is quite general and can be fine tuned through provided heuristics in a convenient human-assisted process of code correction.

An important strength of the work that is presented here is that it was implemented and applied on a complicated published protocol to find and correct an actual error. The α-core protocol is useful for obtaining multiprocess interaction in a distributed system that permits also alternative (i.e., nondeterministic) choices. To the best of our knowledge, this error in the protocol is not documented. Such a method and tool can be used in an interactive code development process. It is, perhaps, unreasonable to expect in general the automatic generation of distributed code, as it is shown by Pnueli and Rosner [14] to be an undecidable problem. However, it is also quite hard to expect programmers to come up with optimized manual solutions to some existing coordination problems.

References

1. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. 22(6), 307–309 (1986)
2. Arcuri, A., Yao, X.: A novel co-evolutionary approach to automatic software bug fixing. In: IEEE Congress on Evolutionary Computation, pp. 162–168 (2008)
3. Buckley, G.N., Silberschatz, A.: An effective implementation for the generalized input-output construct of csp. ACM Trans. Program. Lang. Syst. 5(2), 223–235 (1983)
4. Emerson, E.A., Kahlon, V.: Parameterized model checking of ring-based message passing systems. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 325–339. Springer, Heidelberg (2004)
5. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM 21(8), 666–677 (1978)
6. Holzmann, G.J.: The SPIN Model Checker. Pearson Education, London (2003)
7. Johnson, C.G.: Genetic programming with fitness based on model checking. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 114–124. Springer, Heidelberg (2007)
8. Katz, G., Peled, D.: Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 33–47. Springer, Heidelberg (2008)
9. Katz, G., Peled, D.: Model checking-based genetic programming with an application to mutual exclusion. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 141–156. Springer, Heidelberg (2008)
10. Katz, G., Peled, D.: Synthesizing solutions to the leader election problem using model checking and genetic programming. In: HVC (2009)
11. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
12. Niebert, P., Peled, D., Pnueli, A.: Discriminative model checking. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 504–516. Springer, Heidelberg (2008)

13. Pérez, J.A., Corchuelo, R., Toro, M.: An order-based algorithm for implementing multiparty synchronization. Concurrency - Practice and Experience 16(12), 1173–1206 (2004)
14. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: FOCS, pp. 746–757 (1990)
15. Weimer, W., Nguyen, T., Goues, C.L., Forrest, S.: Automatically finding patches using genetic programming. In: ICSE, pp. 364–374 (2009)