International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-022-00669-9

GENERAL

TACAS 2020

Assume, guarantee or repair: a regular framework for non regular properties

Hadar Frenkel[1] · **Orna Grumberg**[2] · **Corina S. Păsăreanu**[3,4] · **Sarai Sheinvald**[5]

Accepted: 29 August 2022

Abstract
We present Assume-Guarantee-Repair (AGR)—a novel framework which verifies that a program satisfies a set of properties and also *repairs* the program in case the verification fails. We consider *communicating programs*—these are simple C-like programs, extended with synchronous actions over communication channels. Our method, which consists of a learning-based approach to assume–guarantee reasoning, performs verification and repair simultaneously: in every iteration, AGR either makes another step towards proving that the (current) system satisfies the required properties, or alters the system in a way that brings it closer to satisfying the properties. To handle infinite-state systems we build finite abstractions, for which we check the satisfaction of complex properties that contain first-order constraints, using both syntactic and semantic-aware methods. We implemented AGR and evaluated it on various communication protocols. Our experiments present compact proofs of correctness and quick repairs.

Keywords Compositional verification · Repair · Automata learning · Assume–guarantee reasoning · Concurrent systems

1 Introduction

Verification of large-scale systems is a main challenge in the field of formal verification. Often, the verification process of such systems does not scale well. *Compositional verification* aims to address this challenge by breaking up the verification of a large system into the verification of its smaller components which can be checked separately. The results of the verification can be composed back to conclude the correctness of Assume-Guarantee-Repair the entire system. This, however, is not always possible, since the correctness of a component often depends on the behavior of its environment.

The Assume–Guarantee (AG) style compositional verification [33,39] suggests a solution to this problem. The simplest AG rule checks if a system composed of M_1 and M_2 satisfies a property P by checking that M_1 under assumption A satisfies P and that any system containing M_2 as a component satisfies A.

In this work, we present *Assume-Guarantee-Repair* (AGR)—a fully automated framework which applies the AG rule, and while seeking a suitable assumption A, incrementally repairs the given program in case the verification fails. Our framework is inspired by [37], which presented a learning-based method to finding a suitable assumption A, using the $\mathbf{L}^*$ [4] algorithm for learning regular languages. However, unlike previous work, AGR not only performs verification but also repair.

Our AGR framework handles *communicating programs*, which are commonly used for modeling concurrent systems. These are infinite-state C-like programs, extended with the ability to synchronously read and write messages over communication channels. We model such programs as finite-state automata over an *action alphabet*, which reflects the program statements. The automata representation is similar in nature

This research was partially supported by the ISRAEL SCIENCE FOUNDATION (ISF) Grant No. 346/17.

✉ Hadar Frenkel
hadar.frenkel@cispa.de

1 CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

2 Department of Computer Science, The Technion, Haifa, Israel

3 Carnegie Mellon University, Pittsburgh, USA

4 NASA Ames Research Center, Mountain View, CA, USA

5 Department of Software Engineering, ORT Braude College, Karmiel, Israel

Published online: 29 September 2022

to that of control-flow graphs. Its advantage, however, is in the ability to exploit an automata-learning algorithm such as $\mathbf{L}^*$. The accepting states in the automaton representation model points of interest in the program, to which the specification can relate.

We have implemented a tool for AGR and evaluated it on examples modeling communication protocols of various sizes and with various types of errors. Our experiments show that for most examples, AGR converges and finds a repair after a few (2–5) iterations of verify-repair. Moreover, our tool generates assumptions that are significantly smaller then the (possibly repaired) M_2, thus constructing a compact and efficient proof of correctness.

Contributions

To summarize, the main contributions of this paper are:

1. A learning-based Assume–Guarantee algorithm for infinite-state communicating programs, which manages overcoming the difficulties such programs present. In particular, our algorithm overcomes the inherent irregularity of the first-order constraints in these programs, and offers syntactic solutions to the semantic problems they impose.
2. An Assume-Guarantee-Repair algorithm, in which the Assume–Guarantee and the Repair procedures intertwine to produce a repaired program which, due to our construction, maintains many of the "good" behaviors of the original program. Moreover, in case the original program satisfies the property, our algorithm is guaranteed to terminate and return this conclusion.
3. An incremental learning algorithm that uses query results from previous iterations in learning a new language with a richer alphabet.
4. A novel use of abduction to repair communicating programs over first order constraints.
5. An implementation of our algorithm, demonstrating the effectiveness of our framework.

Contribution over conference version

Preliminary results of this work were published in [17]. This paper extends the results of [17] by the following new contributions.

- A formal definition of the weakest assumption for communicating programs, and a study of special cases for which the weakest assumption is regular (Sect. 7).
- A full description of the implementation of the teacher in the context of communicating programs (Sect. 8.1 and Algorithm 1).
- Convergence of the syntactic repair—characterizing cases in which this repair does not terminate (Sect. 8.4.1).

```
1:  while(true)
2:      password:=readInput;
3       while(password≤ 999)
4:          password:=readInput;
5:      password2:=encrypt(password);
```

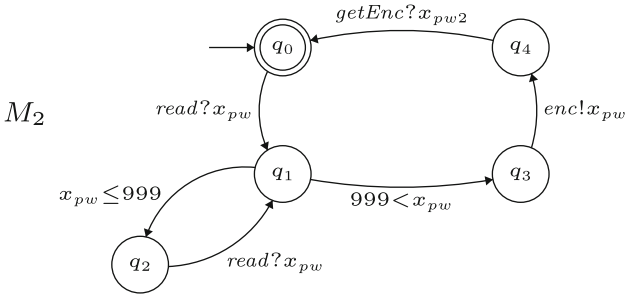

Fig. 1 Modeling a communicating program (M_2) as an automaton

- A formal discussion regarding the soundness of the AG-rule for communicating programs, completeness and termination (Sect. 8.5).
- Full proofs and multiple additional examples throughout the paper.
- A detailed discussion regarding future work and possible extensions (Sect. 10).

Paper organization

The rest of the paper is organized as follows. In the next section, we give a high-level overview of AGR, highlighting the salient features of our approach. Section 3 describes related work. The following three sections set up the background necessary for understanding our approach. Section 4 gives preliminary definitions, Sect. 5 describes communicating programs and their properties (expressed as regular languages), and Sect. 6 proves properties of traces that we use later when proving soundness and completeness of AGR. Section 7 describes and analyzes the assume–guarantee rule used by AGR, while Sect. 8 describes in detail the AGR approach. Finally, Sect. 9 describes our experimental evaluation and Sect. 10 concludes the paper.

2 Overview

We give a high level overview of AGR via an example. Figure 1 presents the code of a communicating program (upper part) and its corresponding automaton M_2 (lower part). The automaton alphabet consists of constraints (e.g. $x_{pw} \leq 999$), assignment actions (e.g. $y_{pw} := 2 \cdot y_{pw}$ in M_1 of Fig. 2), and communication actions (e.g. $enc!x_{pw}$ sends the value of vari-

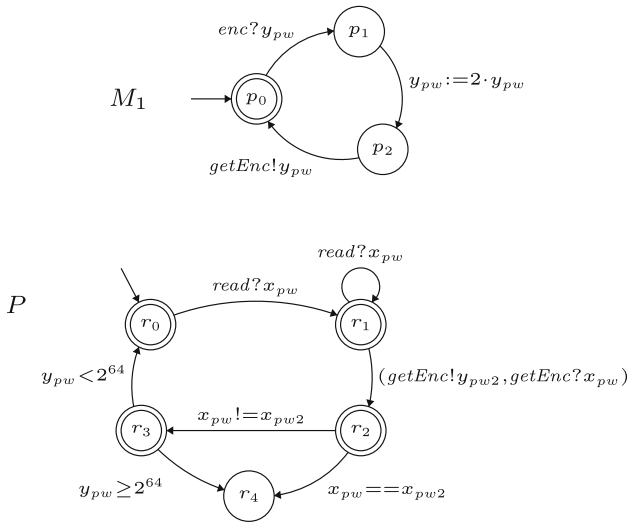

Fig. 2 The program M_1 and the specification P

able x_{pw} over channel *enc*, and $getEnc?x_{pw2}$ reads a value to x_{pw2} on channel $getEnc$).

The specification P[1] is modeled as an automaton that does not contain assignment actions. It may contain communication actions in order to specify behavioral requirements, as well as constraints over the variables of both system components, that express requirements on their values in various points in the runs.

Consider, for example, the program M_1 and the specification P seen in Fig. 2, and the program M_2 of Fig. 1. M_2 reads a password on channel *read* to the variable x_{pw}, and once it is long enough (has at least four digits), it sends the value of x_{pw} to M_1 through channel *enc*. M_1 reads this value to variable y_{pw} and then applies a simple function that changes its value, and sends the changed variable back to M_2. The property P reasons about the parallel run of the two programs. The pair $(getEnc!y_{pw}, getEnc?x_{pw2})$ denotes a synchronization of M_1 and M_2 on channel *getEnc*. P makes sure that the parallel run of M_1 and M_2 always reads a value and then encrypts it—a temporal requirement. In addition, it makes sure that the value after encryption is different from the original value, and that there is no overflow—both are semantic requirements over the program variables. That is, P expresses temporal requirements that contain first order constraints. In case one of the requirements does not hold, P reaches the state r_4 which is an error state. Note that P here is not complete, for simplicity of presentation (see Definition 8 for a formal definition of a complete program).

The $\mathbf{L}^*$ algorithm aims at learning a 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 $\mathbf{L}^*$ algorithm for learning an assumption A for the AG-rule, membership queries are answered according to the satisfaction of the specification P: If $M_1||t$ satisfies P, then the trace t in hand should be in 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 $M_1||A$ satisfies P, and whether the language of M_2 is contained in the language of A. According to the results, the process either continues or halts with an answer to the verification problem. The learning procedure aims at learning the weakest assumption A_w, which contains all the traces that in parallel with M_1 satisfy P . The key observation that guarantees termination in [37] is that the components in this procedure—M_1,M_2, P and A_w are all regular.

Our setting

Our setting is more complicated, since the traces in the components—both the programs and the specification contain constraints, which are to be checked semantically and not syntactically. These constraints may cause some traces to become infeasible. For example, if a trace contains an assignment $x := 3$ followed by a constraint $x \geq 4$ (modeling an "if" statement), then this trace does not contribute any concrete runs, and therefore does not affect the system behavior. Thus, we must add feasibility checks to the process.

Constraints in the specification also pose a difficulty, as satisfiability of a specification is determined by the semantics of the constraints and not only by the language syntax, and so there is more here to check than standard language containment. Moreover, in our setting A_w above may no longer be regular, see Lemma 5. However, our method manages overcoming this problem in a way that still guarantees termination in case the verification succeeds, and progress, otherwise. In addition, we characterize special cases in which the weakest assumption is in fact regular.

As we have described above, not only do we construct a learning-based method for the AG-rule for communicating programs, but we also repair the programs in case the verification fails. An AG-rule can either conclude that $M_1||M_2$ satisfies P, or return a real, non-spurious counterexample of a computation t of $M_1||M_2$ that violates P. In our case, instead of returning t, we repair M_2 in a way that eliminates the counterexample t. Our repair is both syntactic and semantic, where for semantic repair we use *abduction* [38] to infer a new constraint which makes the counterexample t infeasible.

Consider again M_1 and P of Fig. 2 and M_2 of Fig. 1. The composition $M_1||M_2$ does not satisfy P. For example,

[1] Throughout the paper we use *property* and *specification* interchangeably.

if the initial value of x_{pw} is 2^{63}, then after encryption the value of y_{pw} is 2^{64}, violating P. Our algorithm finds a bad trace during the AG stage which captures this bad behavior, and the abduction in the repair stage finds a constraint that eliminates it: $x_{pw} < 2^{63}$, and inserts this constraint to M_2.

Following this step we now have an updated M_2, and we continue with applying the AG-rule again, using information we have gathered in the previous steps. In addition to removing the error trace, we update the alphabet of M_2 with the new constraint. Continuing our example, in the following iteration AGR will verify that the repaired M_2 together with M_1 satisfy P, and terminate.

Thus, AGR operates in a verify-repair loop, where each iteration runs a learning-based process to determine whether the (current) system satisfies P, and if not, eliminates bad behaviors from M_2 while enriching the set of constraints derived from these bad behaviors, which often leads to a quicker convergence. In case the current system does satisfy P, we return the repaired M_2 together with an assumption A that abstracts M_2 and acts as a smaller proof for the correctness of the system. The original motivation for using the AG rule for verification is to find small proofs for the correctness of $M_1||M_2 \vDash P$, without the need to compute the whole composition of $M_1||M_2$, but using the smaller assumption A. In our case, we use the same reasoning, but we do not only prove correctness or provide a counterexample, we also repair M_2. Thus, we keep learning abstractions to the repaired M_2, and, as we later show, rely on information from previous iterations in order to learn an abstraction for the next iteration. Our algorithm produces both a repaired system M_2' such that $M_1||M_2' \vDash P$; and an assumption A that serves as an abstraction for M_2' and allows to prove the correctness of the repaired system with respect to P, without the need to compute the whole composition of $M_1||M_2'$.

The assumption A can later be used for the verification of different components other than M_2 (or the repaired M_2'). If some M_2'' is such that $M_2'' \subseteq A$ (that is, A is an abstraction of M_2''), then from $M_1||A \vDash P$ we conclude that $M_1||M_2'' \vDash P$. That is, given M_2'' and the small A, all we need to check is that A is indeed an abstraction of M_2'', instead of computing the composition of the whole system and specification.

3 Related work

Assume–guarantee style compositional verification [33,39] has been extensively studied. The assumptions necessary for compositional verification were first produced manually, limiting the practicality of the method. More recent works [7, 10,19,21] proposed techniques for automatic assumption generation using learning and abstraction refinement techniques, making assume–guarantee verification more appealing. In [7,37] alphabet refinement has been suggested as an optimization, to reduce the alphabet of the generated assumptions, and consequently their sizes. This optimization can easily be incorporated into our framework as well. Other learning-based approaches for automating assumption generation have been described in [8,9,23]. All these works address non-circular rules and are limited to finite state systems. Automatic assumption generation for circular rules is presented in [13,14], using compositional rules similar to the ones studied in [30,34]. Our approach is based on a non-circular rule but it targets complex, infinite-state concurrent systems, and addresses not only verification but also repair. The compositional framework presented in [27] addresses $\mathbf{L}^*$-based compositional verification and synthesis but it only targets finite state systems.

Several works use abduction. The work in [26] addresses automatic synthesis of circular compositional proofs based on logical abduction; however the focus of that work is sequential programs, while here we target concurrent programs. A sequential setting is also considered in [2], where abduction is used for automatically generating a program environment. Our computation of abduction is similar to that of [2]. However, we require our constraints to be over a predefined set of variables, while they look for a minimal set. Moreover, our goal is to repair a faulty program.

The approach presented in [45] aims to compute the *interface* of an infinite-state component. Similar to our work, the approach works with both over- and under-approximations but it only analyzes one component at a time. Furthermore, the component is restricted to be deterministic (necessary for the permissiveness check). In contrast we use both components of a system to compute the necessary assumptions, and as a result they can be much smaller than in [45]. Furthermore, we do not restrict the components to be deterministic and, more importantly, we also address the system repair in case of dissatisfaction.

Many works study the learnability of symbolic automata, e.g. [5,15,29,44]. The work of [24] studies alphabet refinement for learning symbolic automata. However, all of these are restricted to the abstraction of single transitions, where valuations of variables (states) and their update via program statements are not considered. We, on the other hand, model program statements and program states.

There is extensive research on automated program repair. Examples for testing-based approaches are [22,31,32,35,40], whereas in this work we repair with respect to a formal specification rather than a test-suit. In the context of formal repair with respect to a specification, [41] lays foundations of diagnosis of errors in multi-component systems, with respect to logical specifications. Their setting relies on the logical representation of the system and the specification, and is concerned with finding all reasons to the violation of the property. In our work we are indeed looking for error traces that violate the specification, but our specifications allow us to reason

also about the temporal behavior of the program due to its representation as an automaton. [25] presents an algorithm to repair finite-state programs with respect to a temporal specification, where we are concerned with infinite-state programs that admit some finite representation. [42,43] present automated repair algorithms using SAT based techniques for specifications given as assertions. Note that while we are aiming at repairing the system, our work is not restricted only to repair, but also for finding a small proof of correctness, in the form of the assumption A, that abstracts (the repaired) M_2 and allows a smaller and more efficient composition of the multiple components.

4 Preliminaries

4.1 Regular languages

In this section we define the notions of finite automata and regular languages. We then give a high-level description of the automata learning algorithm $\mathbf{L}^*$.

Definition 1 A *Finite Automaton* is $M = \langle Q, \alpha, \delta, q_0, F \rangle$, where all sets are finite and nonempty, and:

1. Q is the set of states and $q_0 \in Q$ is the initial state.
2. α is the set of alphabet of M.
3. $\delta \subseteq Q \times \alpha \times Q$ is the transition relation. We sometimes also refer to it as a function $\delta : Q \times \alpha \to 2^Q$.
4. $F \subseteq Q$ is the set of accepting states.

A Finite automaton is *deterministic* (DFA) if for every $q \in Q$ and every $a \in \alpha$, $|\delta(q, a)| \leq 1$. It is also *complete* if for every $q \in Q$ and every $a \in \alpha$, $|\delta(q, a)| = 1$.

Let $w = a_1, \ldots, a_k$ be a word over alphabet α. We say that M accepts w if there exists $r_0, \ldots, r_k \in Q$ such that $r_0 = q_0$, $r_k \in F$ and for every i, $0 \leq i \leq k-1$, $r_{i+1} \in \delta(r_i, a_{i+1})$. The language of M, $L(M)$, is the set of all words accepted by M. A set of words W over α is called *regular* if there exists a finite automaton M such that $W = L(M)$.

4.1.1 Learning regular languages

In *active automata learning* we are interested in learning a DFA $\mathcal{A}$ for some regular language U, by issuing queries to some knowledgeable teacher. $\mathbf{L}^*$ [4] is an algorithm for active automata learning, using membership and equivalence queries. Its entities consist of a *teacher*—an oracle who answers *membership queries* (MQs) "is the word w in U?", and *equivalence queries* (EQs) "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.

The learner presents its queries to the teacher following the instructions of the $\mathbf{L}^*$ algorithm. In particular, membership queries present words in increasing length and in alphabetical order. Based on its queries, the learner maintains an *observation table* T, which accumulates all words learned so far, together with an indication of whether each word belongs to U or not. If w is in T and is indicated to be in U we call w a *positive example*, denoted by $+w$, and if it is indicated to not be in U we call it a *negative example*, denoted by w.

When the observation table is "stable", the learner generalizes the table and constructs a conjectured automaton $\mathcal{A}'$, which is sent via equivalence query to the teacher. The teacher either determines that $L(\mathcal{A}') = \mathcal{U}$, in which case $\mathbf{L}^*$ terminates; or returns a counterexample in the form of a word which is in the symmetric difference between $L(\mathcal{A}')$ and U. The learner updates its observation table accordingly and continues with its construction.

If the teacher answers according to a regular language U, then the $\mathbf{L}^*$ algorithm is guaranteed to terminate and learn a DFA for U, in polynomial time. In the rest of the paper we rely on the correctness of $\mathbf{L}^*$ and the fact that the learner queries words in increasing order.

For more details regarding the $\mathbf{L}^*$ algorithm, we refer the reader to [4].

4.2 Assume–guarantee reasoning

For large systems, composed of two components (or more), Assume–Guarantee (AG) proof rule [39] is highly effective. Given a system composed of components M_1 and M_2 and a property P, the AG-rule concludes that $M_1||M_2 \models P$ provided that an assumption A is found, such that $M_1||A \models P$ and the behaviors of M_2 are contained in the behaviors of A. For M_1, M_2, P and A that are all Label Transition Systems (LTSs), an automated AG rule has been suggested in [10]. There, the assumption A is learned using the $\mathbf{L}^*$ algorithm. Often, the learned A is much smaller (in terms of states and transitions) than M_2. Thus, an efficient composition proof rule is obtained, for proving that $M_1||M_2 \models P$.

In using the $\mathbf{L}^*$ algorithm for learning an assumption A for the AG-rule, membership queries are answered according to the satisfaction of the specification P: If $M_1||t$ satisfies P, then the trace t in hand should be in 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 $M_1||A$ satisfies P, and whether the language of M_2 is contained in the language of A. According to the results, the process either continues or halts with an answer to the verification problem. The learning procedure aims at learning the weakest assumption A_w, which contains all the traces that in parallel with M_1 satisfy P . The key observation that guarantees termination

in [37] is that the components in this procedure—M_1,M_2, P and A_w—are all regular.

It worth noticing that often A, learned by $\mathbf{L}^*$, is much smaller than the weakest assumption and than M_2.

5 Communicating programs and regular properties

In this section we present the notion of *communicating programs*. These are C-like programs, extended with the ability to synchronously read and write messages over communication channels. We model such programs as automata over an *action alphabet* that reflects the program statements. The alphabet includes *constraints*, which are quantifier-free first-order formulas, representing the conditions in *if* and *while* statements. It also includes *assignment statements* and $read$ and $write$ *communication actions*. The automata representation is similar in nature to that of control-flow graphs. Its advantage, however, is in the ability to exploit an automata-learning algorithm such as $\mathbf{L}^*$ for its verification [4].

We first formally define the alphabet over which communicating programs are defined. Let G be a finite set of communication channels. Let X be a finite set of variables (whose ordered vector is $\bar{x}$) and $\mathbb{D}$ be a (possibly infinite) data domain. For simplicity, we assume that all variables are defined over $\mathbb{D}$. The elements of $\mathbb{D}$ are also used as constants in arithmetic expressions and constraints.

Definition 2 An *action alphabet* over G and X is $\alpha = \mathcal{G} \cup \mathcal{E} \cup \mathcal{C}$ where:

1. $\mathcal{G} \subseteq \{ g?x_1, g!x_1, (g?x_1, g!x_2), (g!x_1, g?x_2) : g \in G, x_1, x_2 \in X\}$ is a finite set of *communication actions*.
   - $g?x$ is a *read* action of a value to the variable x through channel g.
   - $g!x$ is a *write* action of the value of x on channel g. We use $g * x$ to indicate some action, either *read* or *write*, through g.
   - The pairs $(g?x_1, g!x_2)$ and $(g!x_1, g?x_2)$ represent a synchronization of two programs on read-write actions over channel g (defined later).
2. $\mathcal{E} \subseteq \{ x := e : e \in E, x \in X\}$ is a finite set of *assignment statements*, where E is a set of expressions over $X \cup \mathbb{D}$. For an expression e, we denote by $e[\bar{x} \leftarrow \bar{d}]$ the expression over $\mathbb{D}$ in which the variables $\bar{x} \subseteq X$ are assigned with the values of $\bar{d} \subseteq \mathbb{D}$.
3. $\mathcal{C}$ is a finite set of constraints over $X \cup \mathbb{D}$.

Definition 3 A *communicating program* (or, a program) is $M = \langle Q, X, \alpha M, \delta, q_0, F \rangle$, where:

1. Q is a finite set of states and $q_0 \in Q$ is the initial state.
2. X is a finite set of variables that range over $\mathbb{D}$.
3. $\alpha M = \mathcal{G} \cup \mathcal{E} \cup \mathcal{C}$ is the action alphabet of M.
4. $\delta \subseteq Q \times \alpha \times Q$ is the transition relation. We sometimes also refer to it as a function $\delta : Q \times \alpha \to 2^Q$.
5. $F \subseteq Q$ is the set of accepting states.

The words that are read along a communicating program are a *symbolic representation* of the program behaviors. We refer to such a word as a *trace*. Each such trace induces *concrete executions* of the program, which are formed by concrete assignments to the program variables in a way that conforms with the actions along the word.

Although communicating programs are an extension of finite automata, we investigate them from a different perspective. Usually, an automaton takes as input a word w and checks whether w is in the language of the automaton. In this work we like to think of the automaton as the generator of the behavior, as it describes the program. Therefore, we begin with a run of the program, and induce traces from the run, and not the other way around. We now formally define these notions.

Definition 4 A run in a program automaton M is a finite sequence of states and actions $r = \langle q_0, a_1, q_1 \rangle \ldots \langle q_{n-1}, a_n, q_n \rangle$, starting with the initial state q_0, such that $\forall 0 \leq i < n$ we have $\langle q_i, a_{i+1}, q_{i+1} \rangle \in \delta$. The *induced trace* of r is the sequence $t = (a_1, \ldots, a_n)$ of the actions in r. If q_n is accepting, then t is an *accepted trace* of M.

From now on we assume that every trace we discuss is induced by some run. We turn to define the concrete executions of the program.

Definition 5 Let $t = (a_1, \ldots, a_n)$ be a trace and let $(\beta_0, \ldots, \beta_n)$ be a sequence of valuations (i.e., assignments to the program variables).[2] Then a sequence $\pi = (\beta_0, a_1, \beta_1, a_2, \ldots, a_n, \beta_n)$ is an *execution* of t if the following holds.

1. β_0 is an arbitrary valuation.
2. If $a_i = g?x$, then $\beta_i(y) = \beta_{i-1}(y)$ for every $y \neq x$. Intuitively, x is arbitrarily assigned by the read action, and the rest of the variables are unchanged.
3. If a_i is an assignment $x := e$, then $\beta_i(x) = e[\bar{x} \leftarrow \beta_{i-1}(\bar{x})]$ and $\beta_i(y) = \beta_{i-1}(y)$ for every $y \neq x$.
4. If $a_i = (g?x, g!y)$ then $\beta_i(x) = \beta_{i-1}(y)$ and $\beta_i(z) = \beta_{i-1}(z)$ for every $z \neq x$. That is, the effect of a synchronous communication on a channel is that of an assignment.

[2] Such valuations are usually referred to as states. We do not use this terminology here in order not to confuse them with the states of the automaton.

5. If a_i does not involve a read or an assignment, then $\beta_i = \beta_{i-1}$.
6. Finally, if a_i is a constraint in $\mathcal{C}$, then $\beta_i(\bar{x}) \vDash a_i$ (and since a_i does not change the variable assignments, then $\beta_{i-1}(\bar{x}) \vDash a_i$ holds as well).

We say that t is *feasible* if there exists an execution of t.

The *symbolic language* of M, denoted $\mathcal{T}(M)$, is the set of all *accepted* traces induced by runs of M. The *concrete language* of M, denoted $\mathcal{L}(M)$, is the set of all executions of accepted traces in $\mathcal{T}(M)$. We will mostly be interested in feasible traces, which represent (concrete) executions of the program.

Example 1 – The trace $(x := 2 \cdot y,\ g?x,\ y := y + 1,\ g!y)$ is feasible, as it has an execution $(x = 1, y = 3), (x = 6, y = 3), (x = 20, y = 3), (x = 20, y = 4), (x = 20, y = 4)$.
– The trace $(g?x,\ x := x^2, x < 0)$ is not feasible since no β can satisfy the constraint $x < 0$ if $x := x^2$ is executed beforehand.

5.1 Parallel composition

We now describe and define the parallel composition of two communicating programs, and the way in which they communicate.

Let M_1 and M_2 be two programs, where $M_i = \langle Q_i, X_i, \alpha_i, \delta_i, {q_0}^i, F_i\rangle$ for $i = 1, 2$. Let G_1, G_2 be the sets of communication channels occurring in actions of M_1, M_2, respectively. We assume that $X_1 \cap X_2 = \emptyset$.

The *interface alphabet* αI of M_1 and M_2 consists of all communication actions on channels that are common to both components. That is, $\alpha I = \{\, g?x,\ g!x\ :\ g \in G_1 \cap G_2,\ x \in X_1 \cup X_2\}$.

In *parallel composition*, the two components synchronize on their communication interface only when one component writes data through a channel, and the other reads it through the same channel. The two components cannot synchronize if both are trying to read or both are trying to write. We distinguish between communication of the two components with each other (on their common channels), and their communication with their environment. In the former case, the components must "wait" for each other in order to progress together. In the latter case, the communication actions of the two components interleave asynchronously.

Definition 6 The *parallel composition* of M_1 and M_2, denoted $M_1||M_2$, is the program $M = \langle Q, x, \alpha, \delta, q_0, F\rangle$, defined as follows.

1. $Q = (Q_1 \times Q_2) \cup (Q'_1 \times Q'_2)$, where Q'_1 and Q'_2 are new copies of Q_1 and Q_2, respectively. The initial state is $q_0 = (q_0^1, q_0^2)$.
2. $X = X_1 \cup X_2$.
3. $\alpha = \{\, (g?x_1, g!x_2), (g!x_1, g?x_2)\ :\ g * x_1 \in (\alpha_1 \cap \alpha I)$ and $g * x_2 \in (\alpha_2 \cap \alpha I)\} \cup ((\alpha_1 \cup \alpha_2) \setminus \alpha I)$.
That is, the alphabet includes pairs of read-write communication actions on channels that are common to M_1 and M_2. It also includes individual actions of M_1 and M_2, which are not communications on common channels.
4. δ is defined as follows.
   (a) For $(g * x_1, g * x_2) \in \alpha$[3]:
      i. $\delta((q_1, q_2), (g * x_1, g * x_2)) = \{(q'_1, q'_2)_g\}$.
      ii. $\delta((q'_1, q'_2)_g, x_1 = x_2) = \{(p_1, p_2) \mid p_1 \in \delta_1(q_1, g * x_1), p_2 \in \delta_2(q_2, g * x_2)\}$.
   That is, when a communication is performed synchronously in both components, the data is transformed through the channel from the writing component to the reading component. As a result, the values of x_1 and x_2 equalize. This is enforced in M by adding a transition labeled by the constraint $x_1 = x_2$ that immediately follows the synchronous communication.[4]
   (b) For $a \in \alpha_1 \setminus \alpha I$ we set $\delta((q_1, q_2), a) = \{(p_1, q_2) \mid p_1 \in \delta_1(q_1, a)\})$.
   (c) For $a \in \alpha_2 \setminus \alpha I$ we set $\delta((q_1, q_2), a) = \{(q_1, p_2) \mid p_2 \in \delta_2(q_2, a)\}$.
   That is, on actions that are not in the interface alphabet, the two components interleave.
5. $F = F_1 \times F_2$

Definition 7 For a trace t, we define M_t to be the communicating program which only follows the trace t, and has no other transitions. Then, we define $M||t$ to be the composition $M||M_t$ (and similarly for $t||M$).

Figure 3 demonstrates the parallel composition of components M_1 and M_2. The program $M = M_1||M_2$ reads a password from the environment through channel *pass*. The two components synchronize on channel *verify*. Assignments to x are interleaved with reading the value of y from the environment.

5.2 Regular properties and their satisfaction

The specifications we consider are also given as some variation of communicating programs. We now define the syntax and semantics of the properties that we consider as specifications. These are properties that can be represented as

[3] Note that according to item 3, one of the actions must be a read action and the other one is a write action.

[4] Note that, equality is implied by Definition 5 item 4. Here it is included syntactically to emphasize this fact. In-fact, when implementing feasibility checks this equality has to be included explicitly.

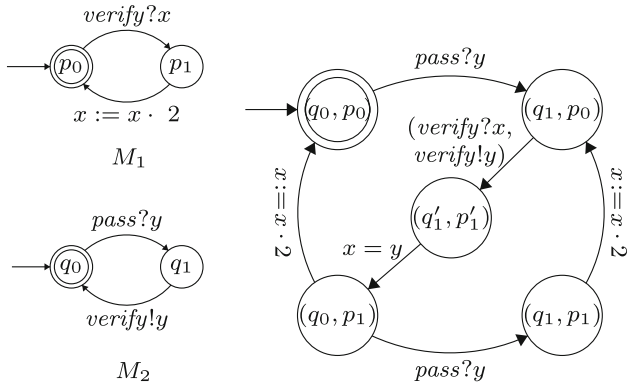

Fig. 3 Components M_1 and M_2 and their parallel composition $M_1||M_2$

finite automata, hence the name *regular*. However, the alphabet of such automata includes communication actions and first-order constraints over program variables. Thus, such automata are suitable for specifying the desired and undesired behaviors of communicating programs over time.

In order to define our properties, we first need the notion of a *deterministic and complete* program. The definition is somewhat different from the standard definition for finite automata, since it takes the semantic meaning of constraints into account. Intuitively, in a deterministic and complete program, every concrete execution has exactly one trace that induces it.

Definition 8 A communicating program over alphabet α is *deterministic and complete* if for every state q and for every action $a \in \alpha$ the following hold:

1. *Syntactic determinism and completeness.* There is exactly one state q' such that $\langle q, a, q' \rangle$ is in δ.[5]
2. *Semantic determinism.* If $\langle q, c_1, q' \rangle$ and $\langle q, c_2, q'' \rangle$ are in δ for constraints $c_1, c_2 \in \mathcal{C}$ such that $c_1 \neq c_2$ and $q' \neq q''$, then $c_1 \wedge c_2 \equiv false$.
3. *Semantic completeness.* Let C_q be the set of all constraints on transitions leaving q. Then $(\bigvee_{c \in C_q} c) \equiv true$.

A *property* is a deterministic and complete program with no assignment actions, whose language defines the set of desired and undesired behaviors over the alphabet αP.

A trace is accepted by a property P if it reaches a state in F, the set of accepting states of P. Otherwise, it reaches a state in $Q \setminus F$, and is rejected by P.

Next, we define the satisfaction relation $\models$ between a program and a property. Intuitively, a program M satisfies a property P (denoted $M \models P$) if all executions induced by accepted traces of M reach an accepting state in P. Thus, the accepted behaviors of M are also accepted by P.

A property P specifies the behavior of a program M by referring to communication actions of M and imposing constraints over the variables of M. Thus, the set of variables of P is identical to that of M. Let $\mathcal{G}$ be the set of communication actions of M. Then, αP includes a subset of $\mathcal{G}$ as well as constraints over the variables of M. The *interface* of M and P, which consists of the communication actions that occur in P, is defined as $\alpha I = \mathcal{G} \cap \alpha P$.

In order to capture the satisfaction relation between M and P, we define a *conjunctive composition* between M and P, denoted $M \times P$. In conjunctive composition, the two components synchronize on their common communication actions when both read or both write through the same communication channel. They interleave on constraints and on actions of αM that are not in αP.

Definition 9 Let $M = \langle Q_M, X_M, \alpha M, \delta_M, q_0^M, F_M \rangle$ be a program and $P = \langle Q_P, X_P, \alpha P, \delta_P, q_0^P, F_P \rangle$ be a property, where $X_M \supseteq X_P$. The *conjunctive composition* of M and P is $M \times P = \langle Q, X, \alpha, \delta, q_0, F \rangle$, where:

1. $Q = Q_M \times Q_P$. The initial state is $q_0 = (q_0^M, q_0^P)$.
2. $X = X_M$.
3. $\alpha = \{ g * x,\ g * y, ,\ (g?x, g!y),\ (g!x, g?y) : g * x, g * y,\ (g * x, g * y) \in \alpha I \} \cup ((\alpha M \cup \alpha P) \setminus \alpha I))$.
   That is, the alphabet includes communication actions on channels common to M and P.
   It also includes individual actions of M and P.
   Note that communication actions of the form $(g * x, g * y)$ can only appear if M is itself a parallel composition of two programs.
4. δ is defined as follows.
   (a) For $a = (g * x, g * y)$ in αI, or $a = g * x$ in αI, we define $\delta((q_1, q_2), a) = \{(q_M, q_P) \mid q_M \in \delta_M(q_1, a), q_P = \delta_P(q_2, a)\}$.[6]
   (b) For $a \in \alpha M \setminus \alpha I$ we define $\delta((q_1, q_2), a) = \{(q_M, q_2) \mid q_M \in \delta_M(q_1, a)\}$.
   (c) For $a \in \alpha P \setminus \alpha I$ we define $\delta((q_1, q_2), a) = \{(q_1, \delta_P(q_2, a))\}$.

   That is, on actions that are not common communication actions to M and P, the two components interleave.
5. $F = F_M \times B_P$, where $B_P = Q_P \setminus F_P$.

Note that accepted traces in $M \times P$ are those that are accepted in M and rejected in P. Such traces are called *error traces* and their corresponding executions are called *error executions*. Intuitively, an error execution is an execution along M

[5] In our examples we sometimes omit the actions that lead to a rejecting sink for the sake of clarity.

[6] Recall that P is deterministic thus its transition relation only corresponds to one state, for each letter.

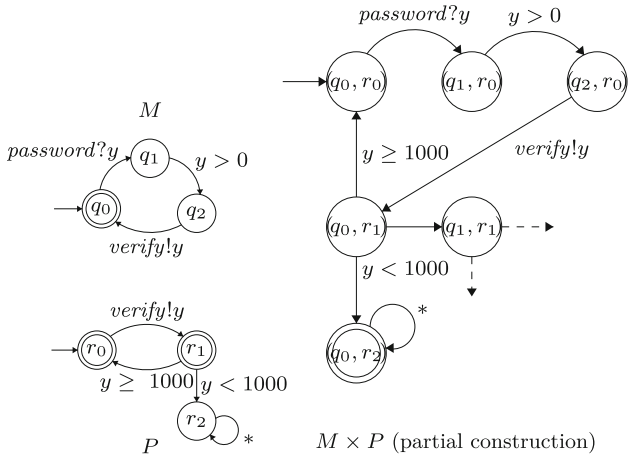

Fig. 4 Partial conjunctive composition of M and P

which violates the properties modeled by P. Such an execution either fails to synchronize on the communication actions, or reaches a point in the computation in which its assignments violate some constraint described by P. These executions are manifested in the traces that are accepted in M but are composed with matching traces that are rejected in P. We can now formally define when a program satisfies a property.

Definition 10 For a program M and a property P, we define $M \models P$ iff $M \times P$ contains no feasible accepted traces.

Thus, a feasible error trace in $M \times P$ is an evidence to $M \not\models P$, since it indicates the existence of an execution that violates P.

Example 2 Consider the program M, the property P and the partial construction of $M \times P$ presented in Fig. 4. The property P requires every verified password y to be of length at least 4. It is easy to see that $M \not\models P$, since the trace $t = (password?y,\ y > 0, verify!y, y < 1000)$ is a feasible error trace in $M \times P$.

6 Traces in the composed system

Before we discuss our framework for compositional verification and repair of communicating systems, we prove some properties of traces in the composed system. We later use these properties in order to prove that our framework is sound and complete (Sect. 7.1), and to prove the correctness and termination of our algorithm (Sects. 8.3 and 8.5).

Definition 11 Let t be a trace over alphabet α, and let $\alpha' \subseteq \alpha$. We denote by $t \downarrow_{\alpha'}$ the *restriction of t on α'*, which is the trace obtained from t by omitting all letters in t that are not in α'. If α contains a communication action $a = (g*x, g*y)$ and we have $g * x \in \alpha'$ then the restriction $t \downarrow_{\alpha'}$ includes the corresponding communication, $g * x$, and similarly for $g * y \in \alpha'$.

Example 3 Let $\alpha = \{g_1!x, x := x+1, x < 10, (g_2?x, g_2!y)\}$, and $\alpha' = \{g_1!x, x := x + 1, g_2?x\}$. Then $(g_2?x, g_2!y) \downarrow_{\alpha'} = g_2?x$, and for

$$t = ((g_2?x, g_2!y), x := x + 1, x := x + 1, x < 10, g_1!x)$$

we have

$$t \downarrow_{\alpha'} = (g_2?x, x := x + 1, x := x + 1, g_1!x).$$

For traces in the conjunctive and parallel compositions, we have the following two lemmas.

Lemma 1 *Let M be a communicating program and P be a property, and let t be a trace of $M \times P$. Then $t \downarrow_{\alpha M}$ is a trace of M.*

Lemma 2 *Let M_1, M_2 be two programs, and let t be a trace of $M_1 || M_2$. Then $t \downarrow_{\alpha M_1}$ is a trace of M_1 and $t \downarrow_{\alpha M_2}$ is a trace of M_2.*

We prove Lemma 1. The proof of Lemma 2 is similar, with the special care of communications actions, and is provided in the full version of this paper [18].

Lemma 1 Let $M = \langle Q_M, X_M, \alpha M, \delta_M, q_0^M, F_M\rangle$ and $P = \langle Q_P, X_P, \alpha P, \delta_P, q_0^P, F_P\rangle$, and denote $M \times P = \langle Q, X_M, \alpha, \delta, q_0, F\rangle$.[7]

Let $r = \langle q_0, c_1, q_1\rangle \ldots \langle q_{m-1}, c_m, q_m\rangle$ be the run in $M \times P$ such that t is induced from r. Denote by $t_M = t \downarrow_{\alpha M}$ the trace $t_M = (c_{i_1}, \cdots, c_{i_n})$.

We first observe the following. If $(a_1, \cdots, a_k)$ is a trace of $M \times P$ such that $\forall i : a_i \notin \alpha M$, and $q = (q_M, q_P^0)$ is the state in $M \times P$ before reading a_1, then $\forall i \geq 1 : \exists q_P^i : \delta((q_M, q_P^{i-1}), a_i) = (q_M, q_P^i)$, that is, when reading a trace that does not contain letters from αM, the program $M \times P$ only advances on the P component. This is true since by the definition of δ, if a_i is not in αM, then $\delta((q_M, q_P), a_i) = (q_M, \delta_P(q_P, a_i))$.

We now inductively prove that $(c_{i_1}, \cdots, c_{i_j})$ is a trace of M for every $1 \leq j \leq n$. In particular, for $j = n$ this means that t_M is a trace of M.

Let $j := 1$ and denote $k := i_1$. Then $c_1, \ldots, c_{k-1} \notin \alpha M$ since k is the first index of t for which $c_k \in \alpha M$. Thus, $\forall 1 < i < k : \exists q_i^P : \delta((q_0^M, q_{i-1}^P), c_i) = (q_0^M, q_i^P)$. For $c_{i_1} = c_k \in \alpha M$, by the definition of δ, we have

$$\delta((q_0^M, q_{k-1}^P), c_{i_1}) = (\delta_M(q_0^M, c_{i_1}), q')$$

[7] Recall that the set of variables X_P is a subset of X_M.

for some $q' \in Q_P$. Then indeed, $\langle q_0^M, c_{i_1}, \delta_M(q_0^M, c_{i_1})\rangle$ is a run in M, making (c_{i_1}) a trace of M.

Let $1 < j \leq n$, and assume $t_{j-1} = (c_{i_1}, \cdots, c_{i_{j-1}})$ is a trace of M. Let $\langle q_0, c_{i_1}, q_1\rangle \ldots \langle q_{j-2}, c_{i_{j-1}}, q_{j-1}\rangle$ be a run that induces t_{j-1}. Denote $i_{j-1} = k, i_j = k + m$ for some $m > 0$. Then, as before, $c_{k+1}, \ldots, c_{k+m-1} \notin \alpha M$, thus $\forall k < l < k + m : \exists q_l^P : \delta(q_{j-1}, q_{l-1}^P), c_l) = (q_{j-1}, q_l^P)$. For c_{i_j} it holds that $\delta((q_{j-1}, q_{k+m-1}^P), c_{i_j}) = (\delta_M(q_{j-1}, c_{i_j}), q'))$ for some $q' \in Q_P$. Thus $(c_{i_1}, \cdots, c_{i_j})$ is a trace of M, as needed. □

We now discuss the *feasibility* of traces in the composed system.

Lemma 3 *Let M be a program and P be a property, and let t be a **feasible** trace of $M \times P$. Then $t \downarrow_{\alpha M}$ is a **feasible** trace of M.*

Proof Let $t \in \mathcal{T}(M \times P)$ be a feasible trace. Then, there exists an execution u on t. Denote $t = (b_1, \cdots, b_n)$ and $u = (\beta_0, b_1, \beta_1, \cdots, b_n, \beta_n)$. We inductively construct an execution e on $t \downarrow_{\alpha M}$. The existence of such an execution e proves that $t \downarrow_{\alpha M}$ is feasible.

Let $t \downarrow_{\alpha M} = (c_1, \cdots, c_k)$. We set $e = (\gamma_0, c_1, \gamma_1, \cdots, c_k, \gamma_k)$ where $\gamma_0, \ldots, \gamma_k$ are defined as follows.

1. Set $j := 0, i := 0$.
2. Define $\gamma_0 := \beta_0$ and set $j := j + 1$.
3. Repeat until $j = k$:
   - Let $i' > i$ be the minimal index such that $b_{i'} = c_j$.
   - Define $\gamma_j := \beta_{i'}$ and set $j := j + 1, i := i' + 1$.

Note that for each $i < l < i'$ is holds that b_l is a constraint. Indeed, by the definition of conjunctive composition (Definition 9), if b_l is not a constraint, then $b_l \in \alpha M$. But in that case, b_l must synchronize with some alphabet letter in $t \downarrow_{\alpha M}$, contradicting the fact that i' is the minimal index for which $b_{i'} = c_j$. Thus, since u is an execution, and for all $i < l < i' : b_l$ is a constraint, it holds that $\forall i \leq l < i' : \beta_i = \beta_l$. In particular, $\beta_{i'-1} = \beta_i = \gamma_{j-1}$. Now, since $b_{i'} = c_j$, we can assign γ_j to be the same as $\beta_{i'}$ and result in a valid assignment. Thus, e is a valid execution on $t \downarrow_{\alpha M}$, making $t \downarrow_{\alpha M}$ feasible, as needed. □

Lemma 4 *Let M_1, M_2 be two programs, and let t be a **feasible** trace of $M_1||M_2$. Then $t \downarrow_{\alpha M_i}$ is a **feasible** trace of M_i for $i \in 1, 2$.*

The proof of Lemma 4 is different from the proof of Lemma 3, since here we can no longer use the exact same assignments as the ones of the run on $M_1||M_2$. In the case of $M \times P$, the set of variables of $M \times P$ is equal to that of M, and the two runs only differ on the constraints that are added to the trace of $M \times P$. In $M_1||M_2$, on the other hand, M_1 and M_2 are defined over two disjoint sets of variables. Nevertheless, The proof is similar to the proof of Lemma 3, and is provided in the full version [18].

7 The assume–guarantee rule for communicating systems

Let M_1 and M_2 be two programs, and let P be a property. The classical Assume–Guarantee (AG) proof rule [39] assures that if we find an assumption A (in our case, a communicating program) such that $M_1||A \models P$ and $M_2 \models A$ both hold, then $M_1||M_2 \models P$ holds as well. For labeled transition systems over a finite alphabet (LTSs) [10], the AG-rule is guaranteed to either prove correctness or return a real (non-spurious) counterexample. The work in [10] relies on the $\mathbf{L}^*$ Algorithm [4] for learning an assumption A for the AG-rule. In particular, $\mathbf{L}^*$ aims at learning A_w, the weakest assumption for which $M_1||A_w \models P$. A crucial point of this method is the fact that A_w is *regular* [20], and thus can be learned by $\mathbf{L}^*$. For communicating programs, this is not the case, as we show in Lemma 5.

Definition 12 *(Weakest Assumption)* Let P be a property and let M_1 and M_2 be two programs. The weakest assumption A_w with respect to M_1, M_2 and P has the language $\mathcal{L}(A_w) = \{w \in (\alpha M_2)^* : M_1||w \models P\}$. That is, A_w is the set of all words over the alphabet of M_2 that together with M_1 satisfy P.

Lemma 5 *For infinite-state communicating programs, the weakest assumption A_w is not always regular.*

Proof Consider the programs M_1 and M_2, and the property P of Fig. 5. Let $\alpha M_2 = \{x := 0,\ y := 0,\ x := x + 1,\ y := y + 1,\ sync\}$. Note that in order to satisfy P, after the *sync* action, a trace t must pass the test $x = y$. Also note that the weakest assumption A_w does not depend on the behavior of M_2, but only on its alphabet. Assume by way of contradiction that $\mathcal{L}(A_w)$ is a regular language, and consider the language

$$L = \{x := 0\} \cdot \{y := 0\} \cdot \{x := x + 1, y := y + 1\}^* \cdot \{sync\}$$

By closure properties of regular languages, it holds that L is a regular language, and thus following our assumption, we have that $L \cap \mathcal{L}(A_w)$ is regular as an intersection of two regular languages. However $L \cap \mathcal{L}(A_w)$ is the set of all words that after the initialization $\{x := 0\}\{y := 0\}$, contain equally many actions of the form $x := x + 1$ and $y := y + 1$. That is

$$L \cap \mathcal{L}(A_w) = \{x := 0\} \cdot \{y := 0\} \cdot L_{eq} \cdot \{sync\}$$

where

$$L_{eq} = \{u \in \{x := x + 1, y := y + 1\}^* :$$

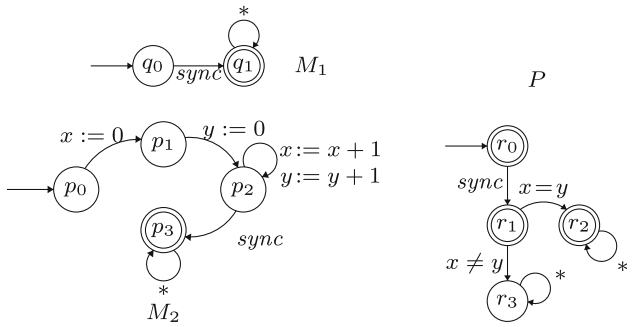

Fig. 5 A system for which the weakest assumption is not regular

num of $x := x + 1$ in u is equal to num of $y := y + 1$ in $u\}$

L_{eq} is not regular since the pumping lemma does not hold for it. For the same reason $L \cap \mathcal{L}(A_w)$ is also not regular, contradicting our assumption that $\mathcal{L}(A_w)$ is regular. □

To cope with this difficulty, we change the focus of learning. Instead of learning the (possibly) non-regular language of A_w, we learn $\mathcal{T}(M_2)$, the set of accepted traces of M_2. This language is guaranteed to be regular, as it is represented by the automaton M_2.

7.1 Soundness and completeness of the assume–guarantee rule for communicating systems

Since we have changed the goal of learning, we first show that in the setting of communicating systems, the assume–guarantee rule is sound and complete.

Theorem 1 *For communicating programs, the Assume-Guarantee rule is sound and complete. That is,*

- *Soundness: for every communicating program A such that $\alpha A \subseteq \alpha M_2$, if $M_1||A \models P$ and $\mathcal{T}(M_2) \subseteq \mathcal{T}(A)$ then $M_1||M_2 \models P$.*
- *Completeness: If $M_1||M_2 \models P$ then there exists an assumption A such that $M_1||A \models P$ and $\mathcal{T}(M_2) \subseteq \mathcal{T}(A)$.*

Proof *Soundness*. Assume by way of contradiction that there exists an assumption A such that $M_1||A \models P$ and $\mathcal{T}(M_2) \subseteq \mathcal{T}(A)$, but $M_1||M_2 \not\models P$. Therefore, there exists an error trace $t \in (M_1||M_2) \times P$. By Lemma 1 and Lemma 2, it holds that $t_2 = t \downarrow_{\alpha M_2} \in \mathcal{T}(M_2)$ and by Lemma 3 and Lemma 4 it holds that t_2 is feasible. Since $\mathcal{T}(M_2) \subseteq \mathcal{T}(A)$, it holds that $t_2 \in \mathcal{T}(A)$ and thus t is an error trace in $(M_1||A) \times P$, contradicting $M_1||A \models P$.

Completeness. If $M_1||M_2 \models P$, then for $A = M_2$ it holds that $M_1||A \models P$ and $\mathcal{T}(M_2) \subseteq \mathcal{T}(A)$. □

7.2 Weakest assumption: special cases

As we have proven in Lemma 5, in the context of communicating systems, the weakest assumption is not always regular, and so we changed our learning goal to the language of M_2, that is, to $\mathcal{T}(M_2)$. We now consider special cases for which the weakest assumption is guaranteed to be regular, and can therefore be used as a target for the learning process. This may result in the generation of a more general assumption.

We first show that if all components are constraints-free, then the weakest assumption is guaranteed to be regular. Intuitively, this is since when there are no constraints in the composed system (that includes also the specification), all traces are feasible. Thus we can reduce the problem of finding a weakest assumption for communicating programs to finding the weakest assumption for finite state automata.

Lemma 6 *Let P be a property and M_1 and M_2 be communicating programs such that αP, αM_1 and αM_2 do not contain constraints. Then, the weakest assumption A_w with respect to M_1, M_2 and P, is regular.*

Proof We construct a communicating system A over αM_2, based on $M_1 \times P$, as follows. A state in A is accepting if its M_1 component is rejecting, or its P component is accepting. We add self loops to all states, labeled by the alphabet of M_2 that does not synchronize with M_1 and P. We replace the alphabet of M_1 and P that is not also in M_2 with ε-transitions (and leave the rest unchanged). Finally, we determinize the result. We therefore have that t is an accepting trace of A iff all the states that A reaches when reading t are either not accepting in M_1, or accepting (hence, not rejecting) in P. For the complete details, see the full version [18]. Let $\mathcal{L}(A)$ be the set of all traces of A.

Claim 1 $\mathcal{L}(A) = \mathcal{L}(A_w)$.

Note that in case that there are no constraints, all traces are feasible, that is, $\mathcal{L}(A) = \mathcal{T}(A)$. To show the correctness of Claim 1, we state the following claim, which can be proved by induction on the trace t. See [18] for the full proof.

Claim 2 For a trace $t \in (\alpha M_2)^*$, the program A reaches the same set of states S when reading t, as the set of states that $M_1 \times P$ reaches given t.

Claim 1 then follows from the definition of the set of accepting states of A. Then, to conclude the proof, we have that $\mathcal{L}(A_w) = \mathcal{T}(A)$, and therefore is regular. □

In [20] the authors prove that the weakest assumption is regular for the setting of LTSs, which are prefix-closed finite-state automata. Their proof relies on the fact that LTSs are prefix closed, and they construct the weakest assumption accordingly. Our proof holds for general communicating programs, and not only for prefix-closed ones. We can therefore extend the result of [20] to the case of general finite-state automata. Corollary 1 follows from the fact that we can refer

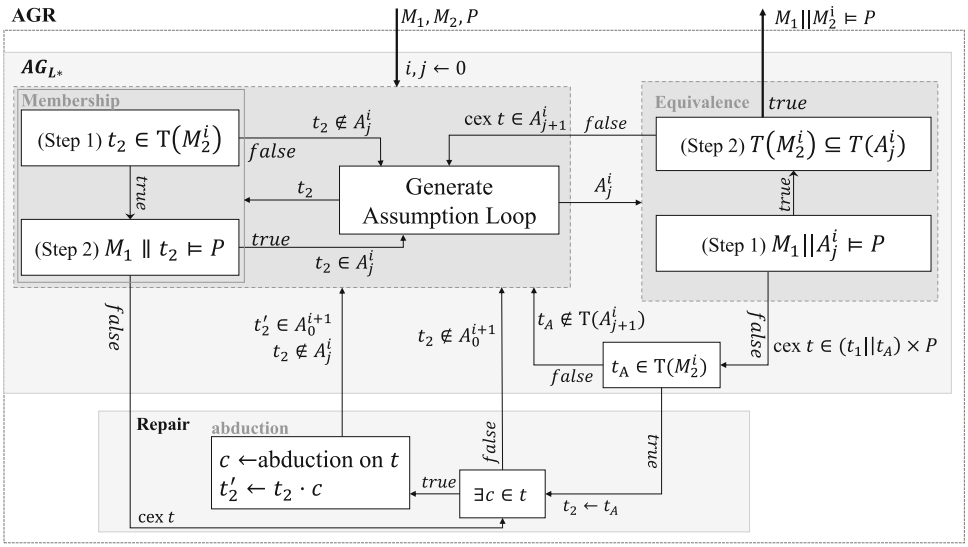

Fig. 6 The flow of AGR

to finite-state automata as communicating programs without constraints; when applying the composition operators $||$ and $\times$, we relax the requirements of synchronization on read-write channels, to requiring synchronization on mutual alphabet letters.

Corollary 1 *Let $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_p$ be finite-state automata such that $\alpha\mathcal{A}_p \subseteq (\alpha\mathcal{A}_1 \cup \alpha\mathcal{A}_2)$. Then, the weakest assumption A_w for $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_p$ is regular.*

We now consider a more general case for which we can find regular weakest assumptions. To this end, we define a refined notion of the weakest assumption, in which we do not care what the behavior of A_w is, for traces that are not feasible. We call it the *semantic weakest assumption*. In the following, for a trace $t \in (\alpha M_2)^*$ we denote $S_t := \mathcal{T}((M_1||t) \times P)$.

Definition 13 *(Semantic weakest assumption)* Let P be a property and let M_1 and M_2 be two communicating programs. A semantic weakest assumption A_s with respect to M_1, M_2 and P has the following property: $\forall t \in (\alpha M_2)^*$ *such that S_t contains feasible traces, it holds that $t \in \mathcal{T}(A_s)$ iff $M_1||t \vDash P$.*

That is, a semantic weakest assumption A_s contains all words over the alphabet of M_2 that together with M_1 satisfy P non-vacuously. Note that there can be more than one such assumption, as we do not define its behavior for sets S_t that include no infeasible traces.

Lemma 7 *For a property P and communicating programs M_1 and M_2 we have the following: If for every $t \in (\alpha M_2)^*$ it holds that either all traces in S_t are feasible, or all of them are infeasible, then there exists a regular semantic weakest assumption.*

Proof We show that the assumption A from the proof of Lemma 6 is such a semantic weakest assumption. We prove that under the terms of Lemma 7, for every trace $t \in (\alpha M_2)^*$ such that S_t contains feasible traces, it holds that $t \in \mathcal{T}(A)$ iff $M_1||t \vDash P$.

Let $t \in \mathcal{T}(A)$. Then, according to the construction of A, and similar to the proof of Lemma 6, all paths in $(M_1||t) \times P$ reach an accepting state, and thus $M_1||t \vDash P$.

For the other direction, let $t \in (\alpha M_2)^*$ such that $M_1||t \vDash P$. According to the terms of the lemma, it holds that either all traces in S_t are not feasible, or that all of them are feasible. If the former holds, then we have no requirement on t.

If the latter holds, then, since $M_1||t \vDash P$, then all paths in $(M_1||t) \times P$ reach an accepting state, and due to the construction of A, it holds that $t \in \mathcal{T}(A)$. □

Note that we restrict the traces in S_t, since if S_t contains both feasible and infeasible traces, then we can no longer guarantee that all paths of $(M_1||t) \times P$ reach an accepting state, even if $M_1||t \vDash P$. For general communicating programs, for which this restriction does not hold, we can use the proof as in Lemma 5 to show that there are cases in which there is no regular semantic weakest assumption, that is, every semantic weakest assumption is not regular.

As a special case of Lemma 7 we have the following.

Corollary 2 *In case the behavior of $(M_1||M_2) \times P$ is deterministic in the sense that for every trace $t \in (\alpha M_2)^*$ there is only one corresponding trace $t' \in (M_1||M_2) \times P$ such that $t' \downarrow_{\alpha M_2} = t$, then there exists a regular semantic weakest assumption.*

8 The assume-guarantee-repair (AGR) framework

In this section we discuss our Assume-Guarantee-Repair (AGR) framework for communicating programs. The framework consists of a learning-based Assume–Guarantee algorithm, called $\text{AG}_{\mathbf{L}^*}$, and a REPAIR procedure, which are tightly joined.

Recall that the goal of $\mathbf{L}^*$ in our case is to learn $\mathcal{T}(M_2)$, though we might terminate earlier if we find a suitable assumption for the AG rule: The nature of $\text{AG}_{\mathbf{L}^*}$ is such that the assumptions it learns before it reaches M_2 may contain the traces of M_2 and more, but still be represented by a smaller automaton. Therefore, similarly to [10], $\text{AG}_{\mathbf{L}^*}$ often terminates with an assumption A that is much smaller than M_2. Indeed, our tool often produces very small assumptions (see Sect. 9).

As mentioned before, not only do we determine whether $M_1||M_2 \models P$, but we also repair the program in case it violates the specification. When $M_1||M_2 \not\models P$, the $\text{AG}_{\mathbf{L}^*}$ algorithm finds an error trace t as a witness for the violation. In this case, we initiate the REPAIR procedure, which eliminates t from M_2. REPAIR applies abduction in order to learn a new constraint which, when added to t, makes the counterexample infeasible. [8] The new constraint enriches the alphabet in a way which may eliminate additional counterexamples from M_2, by making them infeasible. We elaborate on our use of abduction in Sect. 8.3. The removal of t and the addition of the new constraint result in a new goal M_2' for $\text{AG}_{\mathbf{L}^*}$ to learn. Then, $\text{AG}_{\mathbf{L}^*}$ continues to search for a new assumption A' that allows to verify $M_1||M_2' \models P$.

An important feature of our AGR algorithm is its *incrementality*. When learning an assumption A' for M_2' we can use the information gathered in previous MQs and EQs, since the answer for them has not been changed (see Theorem 2 in Sect. 8.2). This allows the learning of M_2' to start from the point where the previous learning has left off, resulting in a more efficient algorithm.

As opposed to the case where $M_1||M_2 \models P$, we cannot guarantee the termination of the repair process in case $M_1||M_2 \not\models P$. This is because we are only guaranteed to remove one (bad) trace and add one (infeasible) trace in every iteration (although in practice, every iteration may remove a larger set of traces). Thus, we may never converge to a repaired system. Nevertheless, in case of property violation, our algorithm always finds an error trace, thus a progress towards a "less erroneous" program is guaranteed.

It should be noted that the $\text{AG}_{\mathbf{L}^*}$ part of our AGR algorithm deviates from the AG-rule of [10] in two important ways. First, since the goal of our learning is $\mathcal{T}(M_2)$ rather than $\mathcal{L}(A_w)$, our membership queries are different in type and order. Second, in order to identify real error traces and send them to REPAIR as early as possible, we add additional queries to the membership phase that reveal such traces. We then send them to REPAIR without ever passing through equivalence queries, which improves the overall efficiency. Indeed, our experiments include several cases in which all repairs were invoked from the membership phase. In these cases, AGR ran an equivalence query only when it has already successfully repaired M_2, and terminated.

[8] There are also cases in which we do not use abduction, as discussed in Sect. 8.4.

8.1 MQs & EQs and the implementation of the teacher

As our algorithm heavily relies on the $\mathbf{L}^*$algorithm, we begin with a description of membership and equivalence queries, both from the side of the learner and the teacher. When using $\mathbf{L}^*$ in verification, the implementation of the teacher plays a major role. We therefore describe the queries issued by the learner, and for each query, we elaborate on how exactly the query is answered by the teacher. The pseudo-code for the teacher is given in Algorithm 1.

8.1.1 Membership queries

As a MQ, the learner asks whether a given trace t is in the language of some suitable assumption A. The order of MQs is as in the $\mathbf{L}^*$ algorithm, that is, we traverse traces in alphabetical order and increasing length. The teacher answers the MQ as follows.

- If $t \notin \mathcal{T}(M_2)$, answer `no`.
- If $t \in \mathcal{T}(M_2)$, check if t is an error trace, and if so, turn directly to repair. That is
  - If $M_1||t \not\models P$, pause learning and turn to repair.
  - If $M_1||t \models P$, answer the MQ with `yes`.

Note that not every membership query is answered immediately, since the learning process may pause for repairing the system. If a repair was issued on a trace t, then after repair, the teacher answers `no` on the MQ on t, but it may provide additional information as we describe in Sect. 8.2.

8.1.2 Equivalence queries

As an EQ, the learner asks, given assumption A, whether A is a suitable assumption to verify the correctness of the system. An EQ is issued once the learner was able to construct an automaton that is consistent with all previous MQs, as done in $\mathbf{L}^*$algorithm. The teacher answers an EQ as follows.

– If $M_1||A \models P$, then A is a suitable candidate according to the AG-rule. Then, we check if the second condition of the AG-rule holds, that is, if $\mathcal{T}(M_2) \subseteq \mathcal{T}(A)$, and answer accordingly.
– If $M_1||A \not\models P$, check if the error trace t that violates P is also a trace of M_2. If so, pause learning and turn to repair. If t is not a real error trace, return `no` to the EQ , along with a trace to be eliminated from A.

Note that, as with MQs, not every EQ is immediately answered. In case of a violation, the algorithm first repairs the system, and only then the teacher returns `no` to the EQ (as the queried A contained a real error trace). Also as in MQs, the repair stage might provide the learner with additional information (Sect. 8.2).

Algorithm 1 below presents the pseudo-code of the teacher when answering the two types of queries. While M_1 and P remain unchanged, M_2 is iteratively repaired (if needed). The teacher therefore considers at iteration i, the repaired program M_2^i.

Algorithm 1 The $\text{AG}_{\mathbf{L}^*}$ Teacher

```
1: function MQ- ORACLE (trace t2)
2:   if t2 ∈ T(M2^i) then
3:     if M1||t2 ⊭ P then
4:       let t ∈ (M1||t2) × P be an error trace          ▷ t is a cex
5:                                                ▷ proving M1||M2^i ⊭ P
6:       return repair - t
7:     else return yes                                  ▷ M1||t2 ⊨ P
8:   else return no                                     ▷ t2 ∉ T(M2^i)
9: function EQ- ORACLE (candidate assumption Aj^i)
10:   if M1||Aj^i ⊨ P then
11:     if T(M2^i) ⊆ T(Aj^i) then return yes
12:     else
13:       let t2 ∈ T(M2^i) \ T(Aj^i)
14:       return no + t2
15:   else                                              ▷ M1||Aj^i ⊭ P
16:     let t ∈ (M1||Aj^i) × P be an error trace
17:     denote t = (t1||tA) × tP
18:     if tA ∈ T(M2^i) then return repair - t
19:     else return no - tA
```

8.2 The assume-guarantee-repair (AGR) algorithm

We now describe our AGR algorithm in more detail (see Algorithm 2). Figure 6 describes the flow of the algorithm. AGR comprises two main parts, namely $\text{AG}_{\mathbf{L}^*}$ and REPAIR.

The input to AGR are the components M_1 and M_2, and the property P. While M_1 and P stay unchanged during AGR, M_2 keeps being updated as long as the algorithm recognizes that it needs repair.

The algorithm works in iterations, where in every iteration the next updated M_2^i is calculated, starting with iteration $i = 0$, where $M_2^0 = M_2$. An iteration starts with the membership phase in line 4 of Algorithm 2, and ends either when $\text{AG}_{\mathbf{L}^*}$ successfully terminates (Algorithm 2 line 12) or when procedure REPAIR is called (Algorithm 2 lines 7 and 15). When a new system M_2^i is constructed, $\text{AG}_{\mathbf{L}^*}$ does not start from scratch. The information that has been learned in previous iterations is still valid for M_2^i. The new iteration is given additional new trace(s) that have been added or removed from the previous M_2^i (Algorithm 2 lines 8, 16).

$\text{AG}_{\mathbf{L}^*}$ consists of two phases: membership, and equivalence. In the membership phase (lines 4–9 of Algorithm 2), the algorithm issues MQs as calls to the function MQ-ORACLE of Algorithm 1. If, during the membership phase, we encounter a trace $t_2 \in M_2^i$ that in parallel with M_1 does not satisfy P, then t_2 is a bad behavior of M_2, and REPAIR is invoked. To this end, we enhance the answers of the teacher not only to `yes` and `no`, but also to `repair`. This holds also for EQs.

Once the learner reaches a candidate assumption A_j^i, it issues an EQ (Algorithm 2 lines 10–20). A_j^i is a suitable assumption if both $M_1||A_j^i \models P$ and $\mathcal{T}(M_2^i) \subseteq \mathcal{T}(A_j^i)$ hold. In this case, AGR terminates and returns M_2^i as a successful repair of M_2. If $M_1||A_j^i \not\models P$, then a counterexample t is returned, that is composed of bad traces in M_1, A_j^i, and P. If the bad trace t_2, the restriction of t to the alphabet of A_j^i, is also in M_2^i, then t_2 is a bad behavior of M_2^i, and here too REPAIR is invoked. Otherwise, AGR updated the $\mathbf{L}^*$ table, returns to the membership phase, and continues to learn A_j^i.

As we have described, REPAIR is called when a bad trace t is found in $(M_1||M_2^i) \times P$ and should be removed. If t contains no constraints then its sequence of actions is illegal and its restriction $t_2 \in \mathcal{T}(M_2^i)$ should be removed from M_2^i. In this case, REPAIR returns to $\text{AG}_{\mathbf{L}^*}$ and updates the learning goal to be $\mathcal{T}(M_2^{i+1}) := \mathcal{T}(M_2^i) \setminus \{t_2\}$, along with the answer "- t_2" that indicates that t_2 should not be a part of the learned assumption. In Sect. 8.4 we discuss different methods for removing t_2 from M_2^i.[9]

The more interesting case is when t contains constraints. In this case, we not only remove the matching t_2 from M_2^i, but also add a new constraint c to the alphabet, which causes t_2 to be infeasible. This way we eliminate t_2, and may also eliminate a family of bad traces that violate the property in the same manner— adding a new constraint can only cause the removal of additional error traces, and cannot add traces to the system. We deduce c using abduction, as we describe in Sect. 8.3. As before, REPAIR returns to $\text{AG}_{\mathbf{L}^*}$ with a new goal to be learned, but now also with an extended alphabet. In addition, we are provided with information about two traces:

[9] For the different methods for removing t_2, we actually end up with a learning goal $\mathcal{T}(M_2^{i+1}) \subseteq \mathcal{T}(M_2^i) \setminus \{t_2\}$ and not necessarily $\mathcal{T}(M_2^i) \setminus \{t_2\}$ itself. We discuss this further in Sect. 8.4.

Algorithm 2 AGR

Input: M_1, M_2, P
Output: A repaired M_2^i (if needed) and an assumption A^i that proofs that $M_1||M_2^i \vDash P$

1: set $i = 0$, $j = 0$, $M_2^i = M_2$
2: **function** $\text{AG}_{\mathbf{L}^*}$
3: **while** true **do** ▷ continue running until repairing M_2, might not terminate
4: **while** $\mathbf{L}^*$learner did not converge to a candidate assumption **do** ▷ issue MQs
5: let $t_2 \in (\alpha M_2^i)^*$ chosen according to the $\mathbf{L}^*$learner
6: **if** MQ- ORACLE (t_2) = `repair - t` **then**
7: $t_2, t_2' :=$ REPAIR(M_2^i, t)
8: add $-t_2$ to the $\mathbf{L}^*$ table and in case of abduction, add $+t_2'$ to the $\mathbf{L}^*$ table
9: **else** add t_2 to the $\mathbf{L}^*$ table according to the answer of the MQ
10: let A_j^i be the candidate assumption generated by the $\mathbf{L}^*$ learner ▷ issue EQ
11: **if** EQ- ORACLE (A_j^i) = `yes` **then**
12: **return** $M_1||M_2^i \vDash P$ together with the assumption A_j^i
13: **else**
14: **if** EQ- ORACLE (A_j^i) = `repair - t` **then**
15: $t_2, t_2' :=$ REPAIR(M_2^i, t)
16: add $-t_2$ to the $\mathbf{L}^*$ table and in case of abduction, add $+t_2'$ to the $\mathbf{L}^*$ table
17: **else** ▷ EQ- ORACLE (A_j^i) = `no` $\pm$ t
18: let t_2 be the cex from the EQ
19: add t_2 to the $\mathbf{L}^*$table according to the answer of the EQ ▷ after issuing an EQ, the algorithm returns to the MQs phase
20: set $j := j + 1$
21: **function** REPAIR(M_2^i, t)
22: let $t_1 \in M_1, t_2 \in M_2^i, t_p \in P$ such that $t = (t_1||t_2) \times t_p$
23: **if** t does not contain constraints **then**
24: set $M_2^{i+1} := \mathcal{T}(M_2^i) \setminus \{t_2\}$
25: set $i := i + 1$, $j := 0$ ▷ i is set to 0, start a new iteration of learning M_2^{i+1}
26: **return** $-t_2$
27: **else** ▷ t contains constraints, use abduction to eliminate t
28: let c be the new constraint learned during abduction and let $t_2' = t_2 \cdot c$
29: update $\alpha M_2^{i+1} := \alpha M_2^i \cup \{c\}$
30: set $M_2^{i+1} := (\mathcal{T}(M_2^i) \setminus \{t_2\}) \cup \{t_2'\}$
31: set $i := i + 1$, $j := 0$ ▷ i is set to 0, start a new iteration of learning M_2^{i+1}
32: **return** $-t_2, +t_2'$

t_2 that should *not* be included in the new assumption, and $(t_2 \cdot c)$ that should be included.

8.2.1 Incremental learning

One of the advantages of AGR is that it is *incremental*, in the sense that answers to membership queries from previous iterations remain unchanged for the repaired system. Formally, we have the following.

Theorem 2 *Assume that T^i is the $\mathbf{L}^*$ table (see Sect. 4.1.1) at iteration i, and let M_2^{i+1} be the repaired component after that iteration. Then, T^i is consistent with M_2^{i+1}.*

From Theorem 2 it follows that, in particular, T^i is consistent with (1) traces that are removed between M_2^i and M_2^{i+1}; and (2) traces that are learned using abduction and added to M_2^{i+1}.

Proof Traces are added to the $\mathbf{L}^*$ table in three scenarios: during MQs; as counterexamples to EQs; and while repairing the system, removing error traces and adding traces learned by abduction. The difference between M_2^i and M_2^{i+1} is only due to the repair part, that is, in error traces that are removed and traces that were added due to abduction. The two components agree on all other traces, and so T^i is consistent with all traces in M_2^{i+1} that are not part of REPAIR.

Let $t_2 \in \mathcal{T}(M_2^i) \setminus \mathcal{T}(M_2^{i+1})$ be an error trace that was removed during REPAIR. We consider the two cases—in which REPAIR is invoked during a MQ, or during an EQ.

In the former case, this is the first time the $\mathbf{L}^*$ learner queries t_2, as the $\mathbf{L}^*$ learner does not issue MQs on traces that are already in the table. Therefore, t_2 is not part of T^i yet, and there is no inconsistency.

If t_2 is an error trace that is found during an EQ, then $M_1||t_2 \nvDash P$ (Algorithm 1, line 16). Since whether $M_1||t_2 \nvDash P$ only depends on M_1, P and the specific trace t_2, and not on M_2^i, and since M_1 and P remain constant, it holds that t_2 was an error trace also in previous iterations. Therefore, t_2 cannot appear in T^i as a positive trace. This concludes the case of traces that are removed from M_2^i during repair.

Now, let t_2' be a positive trace added to M_2^{i+1} using abduction, that is $t_2' = t_2 \cdot c$ for t_2 that was removed by REPAIR. If the constraint c is a new alphabet letter, then in particular t_2' is not over the alphabet of M_2^i and cannot appear in T^i.

We now consider the case in which the added constraint c is already a part of αM_2^i. If the error trace t_2 was found during a MQ, then, since the $\mathbf{L}^*$ learner issues queries in an increasing-length order, it holds that t_2' was not queried in a MQ in previous iterations as $|t_2| < |t_2'|$. Moreover, it cannot be the case that t_2' was a negative counterexample to an EQ since these are only derived from error traces (Algorithm 1, lines 16–19), and t_2' is not an error trace. The same argument holds also if t_2 was found during an EQ.

We are left to show that if t_2 was found during an EQ, it cannot be the case that t_2' was previously queried during a MQ. In this case it holds that the $\mathbf{L}^*$ learner did not issue a MQ on t_2. Since the $\mathbf{L}^*$ learner issues queries in increasing length, and since $|t_2| < |t_2'|$, it holds that t_2' was not previously queried. □

8.3 Semantic repair by abduction

We now describe our repair of M_2^i, in case the error trace t contains constraints (Algorithm 2 line 27). Error traces with no constraints are removed from M_2^i syntactically (Algorithm 2 lines 24–26), while in abduction we *semantically* eliminate t by making it infeasible. The new constraint is then added to the alphabet of M_2^i and may eliminate additional error traces. Note that the constraints added by abduction can only restrict the behavior of M_2, making more traces infeasible. Therefore, we do not add counterexamples to M_2.

The process of inferring new constraints from known facts about the program is called *abduction* [12]. We now describe how we apply it. Given a trace t, let φ_t be the first-order formula (a conjunction of constraints), which constitutes the SSA representation of t [3]. In order to make t infeasible, we look for a formula ψ such that $\psi \wedge \varphi_t \rightarrow \mathit{false}$.[10]

Example 4 Consider the component M_2 of Fig. 1 and the component M_1 and specification P of Fig. 2 from Sect. 1. The following t is an error trace in $(M_1||M_2) \times P$:

$$t = (\mathit{read}?x_{pw}, 999 < x_{pw}, (\mathit{enc}?y_{pw}, \mathit{enc}!x_{pw}), y_{pw} = 2 \cdot y_{pw}, (\mathit{getEnc}!y_{pw}, \mathit{getEnc}?x_{pw2}), x_{pw} \neq x_{pw2}, y_{pw} \geq 2^{64})$$

due to the execution

$$(\mathit{read}?2^{63}, 999 < 2^{63}, (\mathit{enc}?2^{63}, \mathit{enc}!2^{63}), y_{pw} = 2 \cdot 2^{63}, (\mathit{getEnc}!2^{64}, \mathit{getEnc}?2^{64}), 2^{63} \neq 2^{64}, 2^{64} \geq 2^{64}).$$

We then look for a constraint ψ that will make the sequence t, and in particular the violation of the last constraint $y_{pw} \geq 2^{64}$, infeasible.

Note that $t \in \mathcal{T}(M_1||M_2^i) \times P$, and so it includes variables both from X_1, the set of variables of M_1, and from X_2, the set of variables of M_2^i. Since we wish to repair M_2^i, the learned ψ is over the variables of X_2 only. In Example 4, this corresponds to learning a formula over x_{pw}.

The formula $\psi \wedge \varphi_t \rightarrow \mathit{false}$ is equivalent to $\psi \rightarrow (\varphi_t \rightarrow \mathit{false})$. Then, $\psi = \forall x \in X_1 : (\varphi_t \rightarrow \mathit{false}) = \forall x \in X_1(\neg\varphi_t)$, is such a desired constraint: ψ makes t infeasible and is defined only over the variables of X_2. We now use quantifier elimination [46] to produce a quantifier-free formula over X_2. Computing ψ is similar to the abduction suggested in [12], but the focus here is on finding a formula over X_2 rather than over any minimal set of variables as in [12]; in addition, in [12] they look for ψ such that $\varphi_t \wedge \psi$ is not a contradiction, while we specifically look for ψ that blocks φ_t. We use Z3 [11] to apply quantifier elimination and to generate the new constraint.

Example 5 For t of Example 4, the process described above results in the constraint $\psi = x_{pw} < 2^{63}$. Note that while ψ blocks the erroneous behavior of t, it allows all executions of t in which x_{pw} is assigned with smaller values than 2^{63}. In addition, it does not only block the one execution in which $x_{pw} = 2^{63}$, but the set of all erroneous executions in $(M_1||M_2) \times P$ of the example.

After generating $\psi(X_2)$, we add it to the alphabet of M_2^i (line 29 of Algorithm 2). In addition, we produce a new trace $t_2' = t_2 \cdot \psi(X_2)$. The trace t_2' is returned as the output of the abduction.

We now turn to prove that by making t_2 infeasible, we eliminate the error trace t.

Lemma 8 *Let $t = (t_1||t_2) \times t_P$. If t_2 is infeasible, then t is infeasible as well.*

Proof This is due to the fact that t_P can only restrict the behaviors of t_1 and t_2, thus if t_2 is infeasible, t cannot be made feasible. Formally, Lemma 8 follows from Lemma 3 and Lemma 4 given in Sect. 6. By Lemma 3, if $t = (t_1||t_2) \times t_P$ is feasible, then $t_1||t_2$ is a feasible trace of $M_1||M_2$. By Lemma 4, if $t_1||t_2$ is feasible, then t_2 is feasible as well. Therefore, if t_2 is infeasible, then t is infeasible, proving Lemma 8. □

In order to add $t_2 \cdot \psi(X_2)$ to M_2^i while removing t_2, we split the state q that t_2 reaches in M_2^i into two states q and q', and add a transition labeled $\psi(X_2)$ from q to q', where only q' is now accepting (see Fig. 7). Thus, we eliminate the

[10] Usually, in abduction, we look for ψ such that $\psi \wedge \varphi_t$ is not a contradiction. In our case, however, since φ_t is a violation of the specification, we want to infer a formula that makes φ_t unsatisfiable.

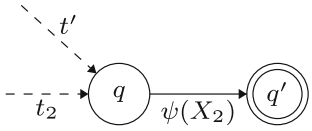

Fig. 7 Adding the constraint $\psi(X_2)$ to block the error trace t_2. Note that ψ is also added to traces other than t_2, for example to t'. Then, $\psi(X_2)$ blocks assignments of t' that violate P in the same way as t_2, but it allows for other assignments of t' to hold

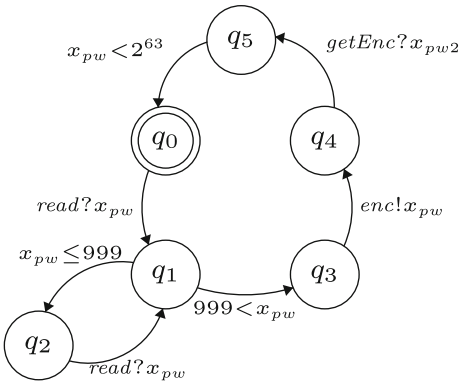

Fig. 8 The repaired component M_2^1

violating trace from $M_1||M_2^i$. REPAIR now returns to $\text{AG}_{\mathbf{L}^*}$ with the negative example—t_2 and the positive example $+t_2'$ to add to the $\mathbf{L}^*$ table T^{i+1} of the next iteration, in order to learn an assumption for the repaired component M_2^{i+1} (which includes t_2' but not t_2).

Example 6 Figure 8 presents the repaired component M_2^1 we generate given M_2 of Fig. 1 and M_1 and P of Fig. 2. As there is only one error trace (that induces many erroneous executions), the repaired component is achieved after one iteration. The new constraint, $\psi = x_{pw} < 2^{63}$, is added at the end of the trace $t_2 = t \downarrow_{\alpha M_2^i}$ for t of Example 4. Intuitively, this constraint is equivalent to adding an *assume* statement in the program.

8.4 Syntactic removal of error traces

Recall that the goal of REPAIR is to remove a bad trace t_2 from M_2 once it is found by $\text{AG}_{\mathbf{L}^*}$. If t_2 contains constraints, we remove it by using abduction as described in Sect. 8.3. Otherwise, we can remove t_2 by constructing a system whose language is $\mathcal{T}(M_2) \setminus \{t_2\}$. We call this the *exact* method for repair. However, removing a single trace at a time may lead to slow convergence, and to an exponential blow-up in the sizes of the repaired systems. Moreover, as we have discussed, in some cases there are infinitely many such traces, in which case AGR may never terminate.

For faster convergence, we have implemented two additional heuristics, namely *approximate* and *aggressive*. These heuristics may remove more than a single trace at a time, while keeping the size of the systems small. While "good" traces may be removed as well, the correctness of the repair is maintained, since no bad traces are added. Moreover, an error trace is likely to be in an erroneous part of the system, and in these cases our heuristics manage to remove a set of error traces in a single step.

We survey the three methods.

- *Exact.* To eliminate only t_2 from M_2, we construct the program M_{t_2} that accepts only t_2, and complement it to construct $\bar{M}_{t_2}$ that accepts all traces except for t_2. Finally, we intersect $\bar{M}_{t_2}$ with M_2. This way we only eliminate t_2, and not other (possibly good) traces. On the other hand, this method converges slowly in case there are many error traces, or does not converge at all if there are infinitely many error traces.
- *Approximate* Similarly to our repair via abduction in Sect. 8.3, we prevent the last transition that t_2 takes from reaching an accepting state. Let q be the state that M_2 reaches when reading t_2. We mark q as a non-accepting state, and add an accepting state q', to which all in-going transitions to q are diverted, except for the last transition on t_2. This way, some traces that lead to q are preserved by reaching q' instead, and the traces that share the last transition of t_2 are eliminated along with t_2. As we have argued, these transitions may also be erroneous.
- *Aggressive* In this simple method, we remove q, the state that M_2 reaches when reading t_2, from the set of accepting states. This way we eliminate t_2 along with all other traces that lead to q. In case that every accepting state is reached by some error trace, this repair might result in an empty language, creating a trivial repair. However, our experiments show that in most cases, this method quickly leads to a non-trivial repair.

8.4.1 Towards convergence of syntactic repair

As discussed above, the exact repair may not terminate in case of infinitely many traces that introduce the same error. Indeed, we now claim that when provided with long-enough counterexamples, we can conclude that the exact repair will not converge, and justifiably turn to the other repair types.

In the following, we claim that once a long-enough error trace is found, it is induced from some run in the underlying automaton, that contains a cycle. Then, all similar traces that follow the same sequence of states are also error traces, and the cycle induces infinitely many error traces that cannot be removed together using the exact repair method. We now formalise this intuition.

We make use of the pumping lemma for regular languages, as stated in the following claim.

Claim Let L be a regular language and let $\mathcal{A}$ be a finite state automaton for L, with n states. Let $z \in L$ such that $|z| > n$. Then, we can write $z = uvw$ such that for every $i \geq 0$, it holds that $uv^i w \in L$.

Lemma 9 *Let $t \in \mathcal{T}((M_1||M_2) \times P)$ be an error trace without constraints. Let N be the number of states in $(M_1||M_2) \times P$. Then, if $|t| > N$, it induces an error trace $t_2 \in \mathcal{T}(M_2)$, such that we can write $t = uvw$ for $|v| > 0$ and for every i it holds that $t_{2^i} := uv^i w$ also corresponds to an error trace.*

Proof Denote $M = (M_1||M_2) \times P$. Let $t \in \mathcal{T}(M)$ be an error trace such that $|t| > N$. Then, the run of M on t contains a cycle. Since M is the composition of three components, there is a state (p, q, r) of M that appears more than once on the run of M on t. Let us denote

$$t = t[1]t[2] \cdots t[j] \cdots t[k] \cdots t[m]$$

such that the run of M on t visits the state (p, q, r) when reading $t[j]$ and $t[k]$. Consider the partition $t = u \cdot v \cdot w$ where

$$u = t[1] \cdots t[j], v = t[j+1] \cdots t[k], w = t[k+1] \cdots t[m]$$

Then, using the argument of the pumping lemma, it holds that $t_i := uv^i w$ is a trace of M, that reaches the same state as t, for every $i \geq 0$. Now, note that for every trace t' that does not contain constraints, and for every system M', it holds that t' is a trace of M' iff t' is a *feasible trace* of M'. In particular, since t does not contain constraints, then t_i is a feasible trace of M for every i, and therefore is an error trace for every i. In particular, it holds that $t_{2^i} = t_i \downarrow_{\alpha M_2}$ is an error trace in M_2 and should be eliminated, for every i. □

Lemma 9 proves that once a long-enough error trace t is detected, in case that t does not contain constraints, then it induces infinitely many error traces. Thus, the exact repair process will never terminate. Note that in this scenario, the approximate repair can fix the system, as it diverts all traces of the same nature to a non-accepting state.

We remark that the same reasoning cannot be applied to traces with constraints.

Example 7 Consider the programs M_1 and M_2 and the property P of Fig. 9, and consider the trace

$$t = (x := 0, (x := x + 1)^{90}, sync, x < 100)$$

where $(x := x+1)^{90}$ means repeating the letter $(x := x+1)$ for 90 times.

The trace t is of length 93, and it is an error trace in the system $(M_1||M_2) \times P$, which is of size at most 9.[11] Then, even-though $|t| > 9$, we cannot decompose it as stated in the claim. This, since after applying $x := x + 1$ more than 100 times, it will no longer be an error trace of the system.

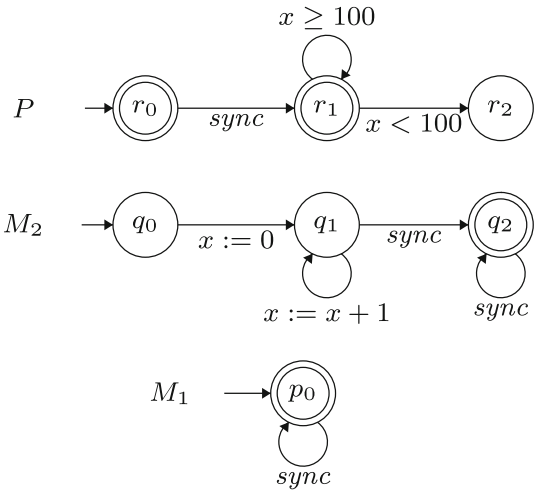

Fig. 9 Programs and a property for Example 7.

Following the discussion in Sect. 7.2, note that if all the three components—M_1, M_2, and P do not contain constraints, then only syntactic queries are needed, and only syntactic repair is applied.

8.5 Correctness and termination

For this discussion, we assume a sound and complete teacher who can answer the membership and equivalence queries in $\text{AG}_{\mathbf{L}^*}$, which require verifying communicating programs and properties with first-order constraints. Our implementation uses Z3 [11] in order to answer satisfiability queries issued in the learning process. The soundness and completeness of Z3 depend on the underlying theory (induced by the program statements we allow). For first-order linear arithmetic over the reals, as we consider in this work, this is indeed the case. However, our method can be applied to all theories for which there exists a sound solver.[12]

As we have discussed earlier, AGR is not guaranteed to terminate, due to its repair part. There are indeed cases for which the REPAIR stage may be called infinitely many times. However, in case that no repair is needed, or if a repaired system is obtained after finitely many calls to REPAIR , then AGR is guaranteed to terminate with a correct answer.

To see why, consider a repaired system M_2^i for which $M_1||M_2^i \models P$. Since the goal of $\text{AG}_{\mathbf{L}^*}$ is to learn $\mathcal{T}(M_2^i)$, which is (syntactically) regular, this stage will terminate at the latest when $\text{AG}_{\mathbf{L}^*}$ learns exactly $\mathcal{T}(M_2^i)$ (it may terminate sooner if a smaller appropriate assumption is found). Notice

[11] In fact, when composing the systems, the resulting system is of size 5.

[12] In case we use an incomplete solver, then termination of $\mathbf{L}^*$ iterations is not guaranteed.

that, in particular, if $M_1||M_2 \models P$, then AGR terminates with a correct answer in the first iteration of the verify-repair loop.

REPAIR is only invoked when a (real) error trace t_2 is found in M_2^i, in which case a new system M_2^{i+1}, that does not include t_2, is produced by REPAIR. If $M_1||M_2^i \not\models P$, then an error trace is guaranteed to be found by $\text{AG}_{\mathbf{L}^*}$ either in the membership or equivalence phase. Therefore, also in case that $M_1||M_2^i$ violates P, the iteration is guaranteed to terminate.

In particular, since every iteration of AGR finds and removes an error trace t_2, and no new erroneous traces are introduced in the updated system, then in case that M_2 has finitely many error traces, AGR is guaranteed to terminate with a repaired system, which is correct with respect to P.[13]

To conclude the above discussion, Theorem 3 formally states the correctness and termination of the AGR algorithm. Recall that in Algorithm 2 we set $M_2^0 := M_2$ and that M_2^i is the repaired component after i iterations of repair.

Theorem 3 *1. If $M_1||M_2 \models P$ then AGR terminates with the correct answer. That is, the output of AGR is an assumption A_0 such that $M_1||A_0 \models P$ and $M_2 \subseteq A_0$.*
2. *If, after i iterations, a repaired program M_2^i is such that $M_1||M_2^i \models P$, then AGR terminates with the correct answer. That is, AGR outputs an assumption A^i for the AG rule (this is a generalization of item 1).*
3. *If an iteration i of AGR ends with an error trace t, then $M_1||M_2^i \not\models P$.*
4. *If $M_1||M_2^i \not\models P$ then AGR finds an error trace. In addition, M_2^{i+1}, the system post* REPAIR*, contains fewer error traces than M_2^i.*

The proof of Theorem 3 follows from Lemmas 10, 11, and 12, given below.

Lemma 10 *Every iteration i of the AGR algorithm terminates. In addition, unless* REPAIR *is invoked, answers to* MQs *and* EQs *are consistent with $\mathcal{T}(M_2^i)$.*[14] *That is, whenever the AG_{L^*} teacher (Algorithm 1) returns* `yes` *for a* MQ *on t_2 or $+t_2$ as a counterexample for an* EQ*, then indeed $t_2 \in \mathcal{T}(M_2^i)$; and whenever the AG_{L^*} teacher returns* `no` *for a* MQ *or t_2 as a counterexample for an* EQ*, then indeed $t_2 \notin \mathcal{T}(M_2^i)$.*

Proof Consistency of MQs with M_2^i is straight forward as the MQ- ORACLE (Algorithm 1) answers MQs according to membership in $\mathcal{T}(M_2^i)$, or invokes REPAIR. The same holds for EQs – in case REPAIR is not invoked, EQ- ORACLE returns the counterexample $+t_2$ (Algorithm 1 line 14) if it is in M_2^i but not part of the assumption, and it returns t_2 (Algorithm 1 line 19) if it is in the assumption but not in M_2^i.

If REPAIR was not invoked, we have that since both types of queries are consistent with $\mathcal{T}(M_2^i)$, which is a regular language, the current iteration terminates due to the correctness of $\mathbf{L}^*$. If REPAIR was invoked, then obviously the iteration terminates, by calling REPAIR. Also note that REPAIR is only called when a real error is found, that is, a trace $t_2 \in \mathcal{T}(M_2^i)$ such that $M_1||t_2 \not\models P$. □

Lemma 11 *If $M_1||M_2^i \models P$, the AGR algorithm terminates with an assumption A^i for the AG rule. If $M_1||M_2^i \not\models P$, AGR finds a counterexample witnessing the violation (and continues to repair M_2^i).*

Proof Assume that $M_1||M_2^i \models P$. By Lemma 10, the answers to MQs and EQs are consistent with $\mathcal{T}(M_2^i)$, and from the correctness of $\mathbf{L}^*$ algorithm we conclude that the algorithm will eventually learn $\mathcal{T}(M_2^i)$. Note that in case that $M_1||M_2 \models P$ and that AGR learned $\mathcal{T}(M_2^i)$, that is $\mathcal{T}(A^i) = \mathcal{T}(M_2^i)$, then the the EQ- ORACLE returns `yes` (Algorithm 1 line 11) and the algorithm terminates with the assumption A_i as a proof of correctness (Algorithm 2 line 12).

Assume that $M_1||M_2^i \not\models P$. Then there exists an error trace $t \in (M_1||M_2^i) \times P$. From Lemmas 3, 4 it holds that $t_2 = t \downarrow_{\alpha M_2^i}$ is feasible in M_2. In particular, it holds that t is an error trace of $(M_1||t_2) \times P$. Thus, $M_1||t_2 \not\models P$. Since AGR converges towards $\mathcal{T}(M_2^i)$ (by Lemma 10), either t_2 shows up as an MQ, and the MQ- ORACLE indicates that repair is needed (Algorithm 1 lines 2–6); Or AGR comes up with a candidate assumption and issues an EQ on it. There, again, t_2 (or some other error trace) will come up as an error trace $t_2 \in \mathcal{T}(M_2^i)$, and the EQ- ORACLE will indicate that repair is needed (Algorithm 1 lines 15–18). □

Note that although each phase converges towards $\mathcal{T}(M_2^i)$, it may terminate earlier. We show that in case that the algorithm terminates before finding $\mathcal{T}(M_2^i)$, it returns the correct answer.

Lemma 12 1. *If AGR outputs an assumption A, then $M_1||A \models P$ and there exists i such that $\mathcal{T}(M_2^i) \subseteq \mathcal{T}(A)$, thus we can conclude $M_1||M_2^i \models P$.*
2. *If a phase i of AGR ends with finding an error trace t, then $M_1||M_2^i \not\models P$.*

Proof *Item 1.* Assume AGR returns an assumption A. Then there exists i such that $\mathcal{T}(M_2^i) \subseteq \mathcal{T}(A)$ and $M_1||A \models P$, since this is the only scenario in which an assumption A is returned (Algorithm 1 lines 10–11 and Algorithm 2 lines 11–12). From the soundness of the AG rule for communicating systems (Theorem 1) it holds that $M_1||M_2^i \models P$.

Item 2. Assume now that a phase i of AGR ends with finding an error trace t (and a call to REPAIR). We prove that

[13] Note that finitely many error traces might induce infinitely many erroneous executions, that are all eliminated together when we eliminate t_2.

[14] If REPAIR is invoked then, as we remove the trace from M_2^i, the answer will not be consistent with M_2^i, but it will be consistent with M_2^{i+1}.

$M_1||M_2^i \not\models P$. First note that AGR may output such a trace both while making a MQ and while making an EQ. If t was found during a MQ (Algorithm 1 lines 3-6), then there exists $t_2 \in T(M_2^i)$ such that $M_1||t_2 \not\models P$, and $t \in (M_1||t_2) \times P$. Since $t_2 \in \mathcal{T}(M_2^i)$, it holds that t is also an error trace of $(M_1||M_2^i) \times P$, proving $M_1||M_2^i \not\models P$.

If t was found during an EQ (Algorithm 1 lines 15-18), then t is an error trace in $(M_1||A_j^i) \times P$. Moreover, $t \downarrow_{\alpha A_j^i} \in \mathcal{T}(M_2^i)$. This makes t an error trace of $(M_1||M_2^i) \times P$ as well, thus $M_1||M_2^i \not\models P$. This concludes the proof. □

The proof of Theorem 3 follows almost directly from the lemmas above.

Proof of Theorem 3 Lemma 11 states that if $M_1||M_2^i \models P$ then AGR terminates with the correct answer. This implies item 1 and item 2.

In addition, Lemma 11 states that if $M_1||M_2^i \not\models P$ then AGR finds an error trace witnessing the violation. Once such an error trace is found, REPAIR is invoked. Since REPAIR eliminates at least one error trace, the system post REPAIR contains fewer error traces, and item 4 follows.

Lemma 12 states that if an iteration i of AGR ends with an error trace, then $M_1||M_2^i \not\models P$. This implies item 3. □

9 Experimental results

We implemented our AGR framework in Java, integrating the $\mathbf{L}^*$ learner implementation from the LTSA tool [28]. We used Z3 [11] to implement calls to the teacher while answering the satisfaction queries in Algorithm 1, and for abduction in REPAIR.

Table 1 displays results of running AGR on various examples, varying in their sizes, types of errors—semantic and syntactic, and the number of errors. Additional results are available in [16]. The examples are available on [1]. The *iterations* column indicates the number of iterations of the verify-repair loop, until a repaired M_2 is achieved. Examples with no errors were verified in the first iteration, and are indicated by *verification*. We experimented with the three repair methods described in Sect. 8.4. Figure 10 presents comparisons between the three methods in terms of run-time and the size of the repair and assumptions. The graphs are given in logarithmic scale.

Most of our examples model multi-client-server communication protocols, with varying sizes. Our tool managed to repair all those examples that were flawed.

As can be seen in Table 1, our tool successfully generates assumptions that are significantly smaller than the repaired and the original M_2.

For the examples that needed repair, in most cases our tool needed 2-5 iterations of verify-repair in order to successfully construct a repaired component. Interestingly, in

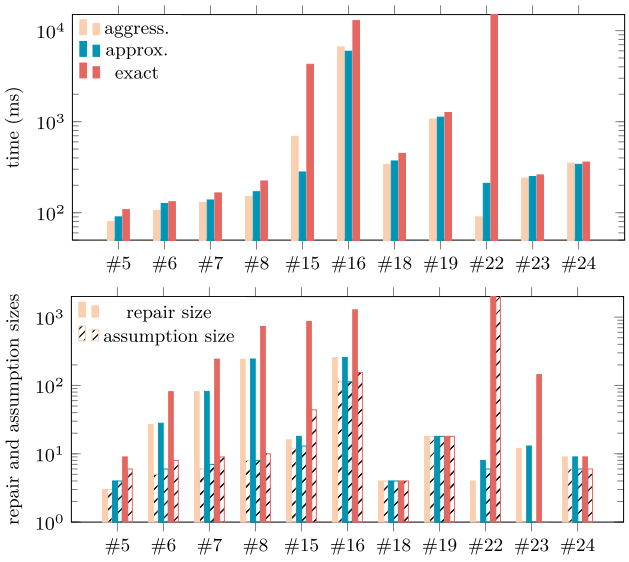

Fig. 10 Comparing repair methods: time and repair size (logarithmic scale)

example #15 the *aggressive* method converged slower than the *approximate* method. This is due to the structure of M_2, in which different error traces lead to different states. Marking these states as non-accepting removed each trace separately. However, some of these traces have a common transition, and preventing this transition from reaching an accepting state, as done in the *approximate* method, managed removing several error traces in a single repair.

Example #22 models a simple structure in which, due to a loop in M_2, the same alphabet sequence can generate infinitely many error traces. The *exact* repair method timed out, since it attempted removing one error trace at a time. On the other hand, the *aggressive* method removed all accepting states, creating an empty program—a trivial (yet valid) repair. In contrast, the *approximate* method created a valid, non-trivial repair.

Example 8 As long as the system needs repair, no assumption can be learned. When we reach a correct M_2, we are usually able to find a smaller assumption that proves the correctness of $M_1||M_2$ with respect to P. Our tool preforms the best on examples in the spirit of M_1 and P of Fig. 11 and M_2 of Fig. 12. There, the specification P allows all traces in which first M_2 acts on one of its channels (G_1, G_2 or G_3), and then M_1 acts on its channel (C). The program M_2 in Fig. 12 is more restrictive than P requires—once the variable x is read through some channel, M_2 continues to use only this channel. $M_1||M_2 \models P$ due to the restriction on their synchronization using $sync_1$ and $sync_2$. We are then able to learn the assumption A of Fig. 13, which is much smaller than M_2, and allows proving the correctness of $M_1||M_2$ with respect to P.

Table 1 AGR algorithm results on various examples

Example	M_1 Size	M_2 Size	P Size	Time (sec.)	A size	Repair size	Repair method	#Iterations
#4	64	64	3	95	7	verification		
#6	2	27	2	0.106	5	27	aggress.	2
				0.126	6	28	approx.	2
				0.132	8	81	exact	2
#7	2	81	2	0.13	6	81	aggress.	2
				0.138	7	82	approx.	2
				0.165	9	243	exact	2
#8	2	243	2	0.15	8	243	aggress.	2
				0.17	8	244	approx.	2
				0.223	10	729	exact	2
#11	5	256	6	4.88	92	verification		
#14	5	256	6	4.44	109	verification		
#15	3	16	5	0.69	12	16	aggress.	5
				0.28	13	18	approx.	3
				4.27	44	864	exact	5
#16	4	256	8	6.63	113	256	aggress.	2
				5.94	113	257	approx.	2
				12.87	155	1280	exact	2
#19	3	16	5	1.07	18	18	aggress.	3
				1.12	18	18	approx.	3
				1.26	18	18	exact	3
#22	2	4	2	0.09	1	4 (trivial)	aggress.	4
				0.21	6	8	approx.	5
				timeout			exact	timeout

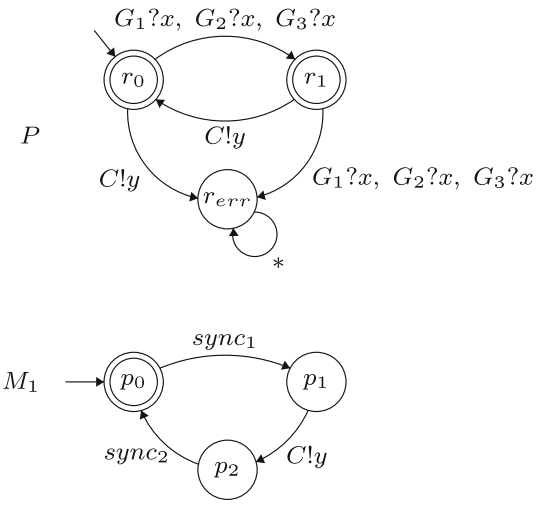

Fig. 11 Program M_1 and specification P of Example 8

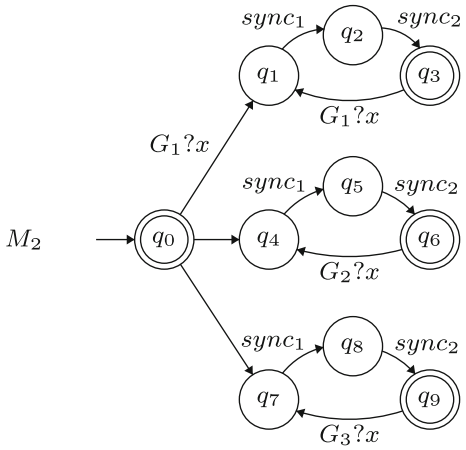

Fig. 12 Program M_2 of Example 8

This example demonstrates a similar behavior to that of examples #4, #6, #7 and #8 of Table 1.

Example 9 Consider now M_1 and P of Fig. 2 and the repaired M_2^1 of Fig. 8. In this case, we learn an assumption with 5

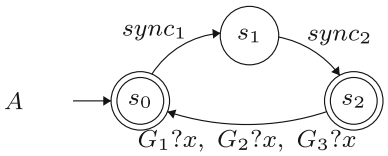

Fig. 13 Assumption A of Example 8

states, that is the result of merging states q_1 and q_2 in M_2^1. This is since we answer queries according to M_2^1, which has a more unique structure than the structure of M_2 of Fig. 12. This demonstrates the behavior of example #19 in Table 1.

10 Conclusion and future work

We have presented the model of communicating programs, which is able to capture program behavior and synchronization between the system components, while exploiting a finite automata representation in order to apply automata learning. We then presented AGR, which offers a new take on the learning-based approach to assume–guarantee verification, by managing to cope with complex properties and by also repairing infinite-state programs.

Our experimental results show that AGR can produce very succinct proofs, and can repair flawed communicating programs efficiently. AGR leverages the finite automata-like representation of the systems in order to apply the $\mathbf{L}^*$ algorithm and to learn small proofs of correctness.

We prove that in general, the weakest assumption that is often used for compositional verification, is not regular for the case of communicating programs, and we come up with a new goal for the learning process. In addition, we find types of communicating programs for which the weakest assumption is regular. We leave finding the full characterization of programs for which the weakest assumption is regular for future work.

In this work, we repair the system by eliminating error traces, and locate new constraints learned using abduction, at the end of the error trace, in order to make it infeasible. A possible extension of this process is to wisely locate constraints over the error trace. Intuitively, we would like the constraint to be as "close" as possible to the error location. However, this is not a trivial task, as the error can be the result of multiple actions of the two communicating programs. Another extension to the repair process is *changing* the program behavior, rather than blocking it. Examples for such a mutation-based approach to program repair are [6,36,42].

For syntactic repair, we characterize cases in which the repair process does not converge. This may happen also in the case of semantic repair, in which infinitely many new constraints are learned and the repair process does not terminate. As future work, we intend to incorporate invariant generation, according to reoccurring error traces, in order to help the convergence of the semantic repair process.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. URL: https://github.com/hadarlh/AGR
2. Albarghouthi, A., Dillig, I., Gurfinkel, A.: Maximal specification synthesis. In: POPL (2016). https://doi.org/10.1145/2837614.2837628
3. Alpern, B., Wegman, M.N., Kenneth Zadeck, F.: Detecting equality of variables in programs. In: POPL (1988). https://doi.org/10.1145/73560.73561
4. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
5. Argyros, G., D'Antoni, L.: The learnability of symbolic automata. In: CAV (2018). https://doi.org/10.1007/978-3-319-96145-3_23
6. Bloem, R., Drechsler, R., Fey, G., Finder, A., Hofferek, G., Könighofer, R., Raik, J., Repinski, U., Sülflow, A.: Forensic- an automatic debugging environment for C programs. In: HVC (2012). https://doi.org/10.1007/978-3-642-39611-3_24
7. Chaki, S., Strichman, O.: Optimized L*-based assume-guarantee reasoning. In: TACAS, (2007). https://doi.org/10.1007/978-3-540-71209-1_22
8. Chen, Y.-F., Clarke, E.M., Farzan, A., Tsai, M.-H., Tsay, Y.-K., Wang, B.-Y.: Automated assume-guarantee reasoning through implicit learning. In: CAV (2010). https://doi.org/10.1007/978-3-642-14295-6_44
9. Chen, Y.-F., Farzan, A., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Learning minimal separating DFA's for compositional verification. In: TACAS (2009). https://doi.org/10.1007/978-3-642-00768-2_3
10. Cobleigh, J. M., Giannakopoulou, D., Pasareanu, C. S.: Learning assumptions for compositional verification. In: TACAS. (2003). https://doi.org/10.1007/3-540-36577-X_24
11. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS (2008). https://doi.org/10.1007/978-3-540-78800-3_24
12. Dillig, I., Dillig, T.: Explain: a tool for performing abductive inference. In: CAV (2013). https://doi.org/10.1007/978-3-642-39799-8_46
13. Elkader, K. A., Grumberg, O., Pasareanu, C. S., Shoham, S.: Automated circular assume-guarantee reasoning. In: FM (2015). https://doi.org/10.1007/978-3-319-19249-9_3
14. Elkader, K. A., Grumberg, O., Pasareanu, C. S., Shoham, S.: Automated circular assume-guarantee reasoning with n-way decomposition and alphabet refinement. In: CAV, (2016). https://doi.org/10.1007/978-3-319-41528-4_18
15. Fisman, Dana, Frenkel, Hadar, Zilles, Sandra: Inferring symbolic automata. In: CSL (2022). URL: https://doi.org/10.4230/LIPIcs.CSL.2022.21
16. Frenkel, H.: Automata over infinite data domains: learnability and applications in program verification and repair. PhD thesis (2021). https://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi/2021/PHD/PHD-2021-09
17. Frenkel, H., Grumberg, O., Pasareanu, C.S., Sheinvald, S.: Assume, guarantee or repair. In: TACAS (2020). https://doi.org/10.1007/978-3-030-45190-5_12

18. Frenkel, H., Grumberg, O., Pasareanu, C. S., Sheinvald, S.: Assume, guarantee or repair: a regular framework for non regular properties (full version). CoRR, (2022). https://doi.org/10.48550/ARXIV.2207.10534
19. Gheorghiu, M., Giannakopoulou, D., Pasareanu, C.S.: Refining interface alphabets for compositional verification. In: TACAS (2007). https://doi.org/10.1007/978-3-540-71209-1_23
20. Giannakopoulou, D., Pasareanu, C. S., Barringer, H.: Assumption generation for software component verification. In: ASE IEEE Computer Society (2002). https://doi.org/10.1109/ASE.2002.1114984
21. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Component verification with automatically generated assumptions. Autom. Softw. Eng. **12**(3), 297–320 (2005). https://doi.org/10.1007/s10515-005-2641-y
22. Goues, C.L., Nguyen, T., Forrest, S., Weimer, W.: Genprog generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2012). https://doi.org/10.1109/TSE.2011.104
23. Gupta, A., McMillan, K.L., Zhaohui, F.: Automated assumption generation for compositional verification. Formal Methods Syst. Des. **32**(3), 285–301 (2008). https://doi.org/10.1007/s10703-008-0050-0
24. Howar, F., Steffen, B., Merten, M.: Automata learning with automated alphabet abstraction refinement. In: VMCAI, (2011). https://doi.org/10.1007/978-3-642-18275-4_19
25. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: CAV, (2005). https://doi.org/10.1007/11513988_23
26. Li, B., Dillig, I., Dillig, T., McMillan, K. L., Sagiv, M.: Synthesis of circular compositional program proofs via abduction. In: TACAS (2013). https://doi.org/10.1007/978-3-642-36742-7_26
27. Lin, Shang-Wei, Hsiung, Pao-Ann: Compositional synthesis of concurrent systems through causal model checking and learning. In: FM, (2014). https://doi.org/10.1007/978-3-319-06410-9_29
28. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. Wiley, Hoboken (1999)
29. Maler, O., Mens, I.-E.: Learning regular languages over large alphabets. In: TACAS, (2014). https://doi.org/10.1007/978-3-642-54862-8_41
30. McMillan, K.L.: Circular compositional reasoning about liveness. In: CHARME (1999). https://doi.org/10.1007/3-540-48153-2_30
31. Mechtaev, S., Yi, J., Roychoudhury, A.: Directfix: looking for simple program repairs. In: ICSE, (2015). https://doi.org/10.1109/ICSE.2015.63
32. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: scalable multiline program patch synthesis via symbolic analysis. In: ICSE, (2016). https://doi.org/10.1145/2884781.2884807
33. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE Trans. Softw. Eng. **7**(4), 417–426 (1981). https://doi.org/10.1109/TSE.1981.230844
34. Namjoshi, K.S., Trefler, R.J.: On the competeness of compositional reasoning. In: CAV (2000). https://doi.org/10.1007/10722167_14
35. Duong T.N., Hoang, Q., Dawei, R., Abhik, C., Satish: Semfix: program repair via semantic analysis. In: ICSE (2013). https://doi.org/10.1109/ICSE.2013.6606623
36. Nguyen, T.-T., Ta, Q.-T., Chin, W.-N.: Automatic program repair using formal verification and expression templates. In: VMCAI 2019. https://doi.org/10.1007/978-3-030-11245-5_4
37. Pasareanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. Formal Methods Syst. Des. (2008). https://doi.org/10.1007/s10703-008-0049-6
38. Peirce, C.S., Hartshorne, C.: Collected Papers of Charles Sanders Peirce. Belknap Press, Cambridge (1932)
39. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Logics and models of concurrent systems, NATO ASI Series, (1985). https://doi.org/10.1007/978-3-642-82453-1_5
40. Qi, Y., Mao, X., Lei, Y.: Efficient automated program repair through fault-recorded testing prioritization. In: 2013 IEEE International Conference on Software Maintenance, (2013). https://doi.org/10.1109/ICSM.2013.29
41. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987). https://doi.org/10.1016/0004-3702(87)90062-2
42. Rothenberg, B.-C., Grumberg, O.: Sound and complete mutation-based program repair. In: FM 2016. https://doi.org/10.1007/978-3-319-48989-6_36
43. Rothenberg, B.-C., Grumberg, O.: Must fault localization for program repair. In: CAV. (2020). https://doi.org/10.1007/978-3-030-53291-8_33
44. Sheinvald, S.: Learning deterministic variable automata over infinite alphabets. In: FM. (2019). https://doi.org/10.1007/978-3-030-30942-8_37
45. Singh, R., Giannakopoulou, D., Pasareanu, C.S.: Learning component interfaces with may and must abstractions. In: CAV (2010). https://doi.org/10.1007/978-3-642-14295-6_45
46. Weispfenning, V.: Quantifier elimination and decision procedures for valued fields. In: Models and Sets. Lecture Notes in Mathematics (LNM). **1103**, 419–472 (1984)