Assume, Guarantee or Repair*

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

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

Abstract. We present Assume-Guarantee-Repair (AGR) – a novel framework which not only verifies that a program satisfies a set of properties, but also *repairs* the program, in case the verification fails. We consider *communicating programs* – these are simple C-like programs, extended with synchronous communication actions over communication channels.
Our method, which consists of a learning-based approach to assume-guarantee reasoning, preforms the two tasks simultaneously: in every iteration of the procedure, we either make another step towards proving that the (current) system satisfies the specification, or alter the system in a way that brings it closer to satisfying the specification. We manage handling infinite-state systems by using a finite abstract representation, and reduce the semantic problems in hand – satisfying complex specifications that also contain first-order constraints – to syntactic ones, namely membership and equivalence queries for regular languages. We implemented our algorithm and evaluated it on various examples. Our experiments present compact proofs of correctness and quick repairs.

1 Introduction

Verification of large-scale systems is a main challenge in the field of formal verification. Often, the verification process of such a system does not scale well. *Compositional verification* aims to verify small components of a system separately, and from the correctness of the individual components, to conclude the correctness of 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 [20,24] suggests a solution to this problem. The simplest AG rule checks if a system composed of components 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. Several frameworks have been proposed to support this style of reasoning. Finding a suitable assumption A is then a common challenge in such frameworks.

In this work, we present *Assume-Guarantee-Repair* (AGR) – a fully automated framework which applies the Assume-Guarantee rule, and while seeking a suitable assumption A, incrementally repairs the given program in case the verification fails. Our framework is inspired by [22], which presented a learning-based method to finding an assumption A, using the L^* [3] algorithm for learning regular languages.

Our AGR framework handles *communicating programs*. These are infinite-state C-like programs, extended with the ability to synchronously read and write messages over communication channels. We model such programs as finite-state automata over an *action alphabet*,

* This research was partially supported by the Technion Hiroshi Fujiwara Cyber Security Research Center, the Israel National Cyber Directorate and the Israel Science Foundation (ISF)

which reflects the program statements. The accepting states in these automata model points of interest in the program that the specification can relate to. 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 L^*.

The composition of the two program components, M_1 and M_2, denoted $M_1||M_2$, synchronizes on read-write actions on the same channel. Between two synchronized actions, the individual actions of both systems interleave.

```
1: while(true)
2:      x := 0;
3       while(x ≤ 999)
4:          password?x;
5:      x := x · 10;
6:      enter!x;
```

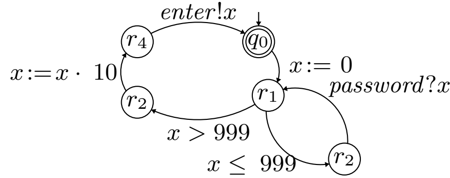

Fig. 1: Modeling a communicating program as an automaton

Figure 1 presents the code of a communicating program (left) and its corresponding automaton (right). The automaton alphabet consists of constraints (e.g. $x \leq 999$), assignment actions (e.g. $x := x \cdot 10$), and communication actions (e.g. $password?x$ reads a value to variable x over channel $password$, and $enter!x$ writes the value of x on channel $enter$).

The specification P is also 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 programs M_1, M_2, and the specification P seen in Figure 2. M_1 reads a bound on the number of times an action must be performed in M_2 (this action can be, say, a $push$ action on a stack). The variable act in M_2 counts the number of times the action has been performed. M_2 performs a sequence of actions, and then reads a value – the bound of M_1 – from M_1 through the channel C. If the number of actions it has performed matches the bound, M_2 finishes the current iteration successfully. The property P makes sure that in the parallel run of the programs the number of actions never exceeds the bound, and that this number eventually reaches the bound in every iteration. The $sync$ actions here denote communication actions on which the components synchronize, and are used for the clarity of the description. Notice that P expresses temporal requirements that contain first order constraints.

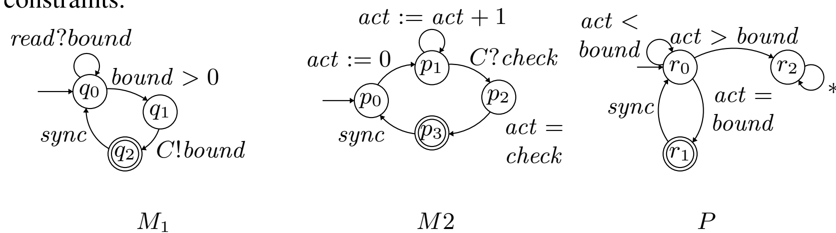

Fig. 2: The programs M_1, M_2, and the specification P

The 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 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 in parallel with 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 is that the components in this procedure – M_1,M_2, P and A_w – are all regular.

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 just by the syntax of the language, and hence there is more here to check than standard language containment. Moreover, A_w above may no longer be regular, see Example 3. However, our method manages to overcome this problem in a way that still guarantees termination in case the verification succeeds, and progress, otherwise.

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 $M_1||M_2 \models P$ ($M_1||M_2$ satisfies P), or return a real, non-spurious counterexample of a computation of $M_1||M_2$ that violates P. In our case, instead of returning the counterexample, we repair M_2 in a way that eliminates this counterexample. We do so by using *abduction* [23] to infer a new constraint which makes the counterexample infeasible. Consider again the program M_2 of Figure 2. Suppose that the transition labeled $C?check$ from p_1 would go directly to p_3 instead of passing through p_2. Then $M_1||M_2 \not\models P$, since without the constraint $act = check$, the number of actions may be different (say, larger) from the bound when reaching p_3. Our algorithm will find a bad trace during the AG stage which will capture this bad behavior, and the abduction in the repair stage will add a constraint that eliminates it, say $act \leq check$, and add it after the $C?check$ action in 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 a following iteration AGR will then find another trace – one in which there are too few actions – and add another constraint, say $act \geq check$, which results in a desired repaired program M_2'.

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 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.

We have implemented a tool for AGR and evaluated it on examples of various sizes and of various types of errors. Our experiments show that for most examples, AGR converges and find repair after 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 to overcome 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.

Related Work Assume-guarantee style compositional verification [20,24] has been extensively studied. The assumptions necessary for compositional verification were first produced manually, limiting the practicality of the method.

More recent works [7,14,12,4] proposed techniques for automatic assumption generation using learning and abstraction refinement techniques, making assume-guarantee verification more appealing. In [22,4] 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 in our framework as well.

Other learning-based approaches for automating assumption generation have been described in [5,15,6]. All these works addresses non-circular rules and are limited to finite state systems. Automatic assumption generation for circular rules is presented in [10,11], using compositional rules similar to the ones studied in [19,21].

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 [17] addresses L^*-based compositional verification and synthesis but it only targets finite state systems.

Also related is the work in [16], which 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 [1], where abduction is used for automatically generating a program environment. Our computation of abduction is similar to that of [1]. However, we require our constraints to be over a predefined set of variables, while they look for a minimal set.

The approach presented in [25] 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 [25]. Furthermore, we do not restrict the components to be deterministic and, mainly, we also address the system repair (in case of dissatisfaction).

2 Communicating Programs

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 graph. Its advantage, however, is in the ability to exploit an automata-learning algorithm such as L^* for its verification.

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 D be a (finite or infinite) data domain. For simplicity, we assume that all variables are defined over D. The elements of D are also used as constants in arithmetic expressions and constraints.

Definition 1. *An* action alphabet *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*, and* $g!x$ *is a* write *action of the value of* x *on channel* g*. We use the notation* $g * x$ *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 *(to be defined later).*
2. $\mathcal{E} \subseteq \{\ x := e \mid e \in E, x \in X\}$ *is a finite set of* assignment statements*, where* E *is a set of arithmetic expressions over* $X \cup D$*.*
3. $\mathcal{C}$ *is a finite set of constraints over* $X \cup D$*.*

Definition 2. *A* communicating program *(or, a program) is* $M = \langle Q, X, \alpha, \delta, q_0, F\rangle$*, 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* D*.*
3. $\alpha = \mathcal{G} \cup \mathcal{E} \cup \mathcal{C}$ *is the action alphabet of* M*.*
4. $\delta \subseteq Q \times \alpha \times Q$ *is the transition relation.*
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 runs* of the program, which are formed by concrete assignments to the program variables in a way that conforms with the actions along the word.

We now formally define these notions.

Definition 3. *A* path *in a program* M *is a finite sequence of states and actions* $p = (q_0, a_1, q_1, \ldots, a_n, q_n)$*, starting with the initial state* q_0*, such that* $\forall 0 \leq i < n$ *we have* $(q_i, a_{i+1}, q_{i+1}) \in \delta$*. The* induced trace *of* p *is the sequence* $t = (a_1, \ldots, a_n)$ *of the actions in* p*. 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 path. We turn to define the concrete runs of the program.

Definition 4. *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)*[4]. *Then a sequence $r = (\beta_0, a_1, \beta_1, a_2, \ldots, a_n, \beta_n)$ is a* run *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.*
5. *If a_i does not involve a read or an assignment, then $\beta_i = \beta_{i-1}$.*
6. *Finally, if $a_i \in \mathcal{C}$ (that is, a_i is a constraint) then $\beta_i(\bar{x}) \vDash a_i$.*

We say that t is feasible *if there exists a run of t.*

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

Example 1.
- The trace $(x := 2 \cdot y,\ g?x,\ y := y + 1,\ g!y)$ is feasible, as it has a run $(x = 1, y = 3), (x = 6, y = 3), (x = 20, y = 3), (x = 20, y = 4), (x = 20, y = 4)$.
- 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.

2.1 Parallel Composition

We now describe and define the parallel run 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 $X_1 \cap X_2 = \emptyset$.

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, $\alpha I = \{\, g?x,\ g!x \mid g \in G_1 \cap G_2,\ x \in X_1 \cup X_2 \}$.

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.

Formally, 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. $X = X_1 \cup X_2$.

[4] 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.

3. $\alpha = \{ (g?x_1, g!x_2), (g!x_1, g?x_2) \,|\, g*x_1 \in (\alpha_1 \cap \alpha I) \text{ and } g*x_2 \in (\alpha_2 \cap \alpha I)\} \cup ((\alpha_1 \cup \alpha_2) \setminus \alpha I)$. That is, the alphabet includes pairs of read-write communication actions on channels 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 $(g * x_1, g * x_2) \in \alpha$:
      i. $\delta((q_1, q_2), (g * x_1, g * x_2)) = (q_1', q_2')$.
      ii. $\delta((q_1', q_2'), x_1 = x_2) = (\delta_1(q_1, g * x_1), \delta_2(q_2, g * x_2))$.

   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 with the constraint $x_1 = x_2$ that immediately follows the transition of the synchronous communication.

   (b) For $a \in \alpha_1 \setminus \alpha I$ we define $\delta((q_1, q_2), a) = (\delta_1(q_1, a), q_2)$.
   (c) For $a \in \alpha_2 \setminus \alpha I$ we define $\delta((q_1, q_2), a) = (q_1, \delta_2(q_2, a))$.

   That is, on actions that are not in the interface alphabet, the two components interleave.
5. $F = F_1 \times F_2$

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.

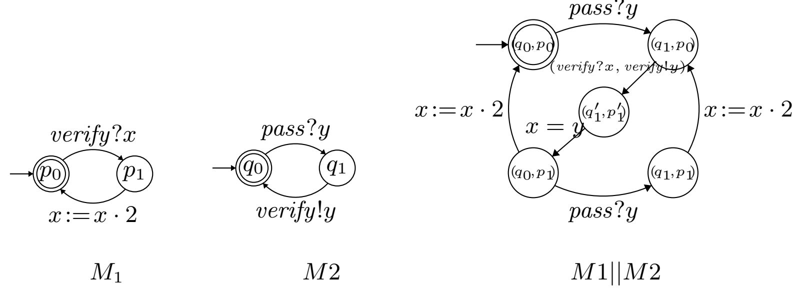

Fig. 3: Components M_1 and M_2 and their parallel composition $M_1||M_2$.

3 Regular Properties and Their Satisfaction

In this section we define the syntax and semantics of the properties that we consider. We consider properties that can be represented as a finite automaton, hence the name *regular*. However, the language of such an automaton consists of words over an alphabet that includes communication actions as well as first-order constraints over program variables. Thus, the language is suitable for specifying the desired and undesired behaviors of a communicating program 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 run has exactly one trace which induces it.

Definition 5. *A program over alphabet α is* deterministic and complete *if for every state q and for every action $a \in \alpha$ the following hold:*

1. *There is exactly one state q' such that (q, a, q') is in δ.*[5]
2. *If (q, c_1, q') and (q, c_2, q'') are in δ for constraints $c_1, c_2 \in \mathcal{C}$ and $q' \neq q''$, then $c_1 \wedge c_2 \equiv false$.*
3. *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.

A trace is accepted by a property P if the unique run on this trace reaches a state in F, the set of accepting states of P. Otherwise, it reaches a state in B, the set of rejecting states of P, 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 runs induced by accepted traces of M reach an accepting state in 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 is defined as $\alpha I = \mathcal{G} \cap \alpha P$. Thus, the interface consistes of the communication actions that occur in 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 the 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 6. *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 $X_M = X_P$. 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. *$X = X_M = X_P$.*
3. *$\alpha = \{ g!x, g?x, (g?x, g!y), (g!x, g?y) \mid g{*}x, (g{*}x, g{*}y) \in \alpha I \} \cup ((\alpha M \cup \alpha P) \setminus \alpha I))$*[6]*.* *That is, the alphabet includes communication actions on channels common to M and P. It also includes individual actions of M and P.*
4. *δ is defined as follows.*
   (a) For $a = (g{*}x, g{*}y) \in \alpha I$, or $a = g{*}x \in \alpha I$: $\delta((q_1, q_2), a) = (\delta_M(q_1, a), \delta_P(q_2, a))$.
   (b) For $a \in \alpha M \setminus \alpha I$: $\delta((q_1, q_2), a) = (\delta_M(q_1, a), q_2)$.
   (c) For $a \in \alpha P \setminus \alpha I$: $\delta((q_1, q_2), a) = (q_1, \delta_P(q_2, a))$.
   That is, on actions that are not common communication actions to M and P, the two components interleave.
5. *$F = F_M \times B_P$.*

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 runs are called *error runs*. Intuitively, an error run is a run along M which violates the properties modeled by P. Such a run 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 runs are manifested in

[5] in our examples we sometimes omit the actions that lead to a rejecting sink for the sake of clarity.
[6] Note that communication actions of the form $(g * x, g * y)$ can only appear if M is a parallel composition of two programs.

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 7. *For a program M and a property P, we define $M \vDash P$ iff $M \times P$ contains no feasible accepted traces.*

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

Example 2. Consider the program M, the property P and a partial construction of $M \times P$ presented in Figure 4. P requires every verified password y to be of length at least 4. It is easy to see that $M \nvDash P$, since the trace $t = (password?y, y > 0, verify!y, y < 1000)$ is a feasible error trace in $M \times P$. Note that from state (q_0, r_1) it is possible to continue to (q_1, r_1) with a transition of M in which a new password y is entered. In that case, if $y < 1000$, it is not an error trace since the new password y has not been verified yet.

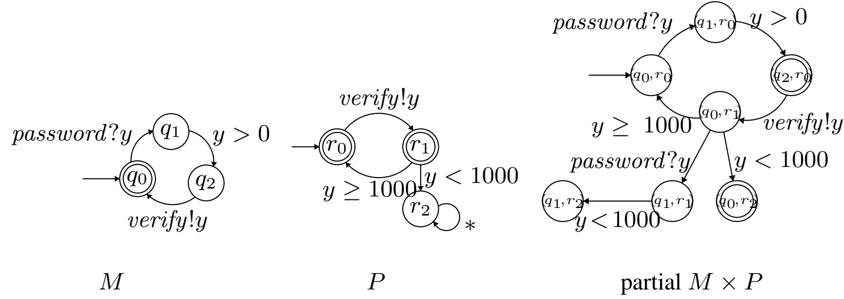

Fig. 4: Partial conjunctive composition of M and P.

4 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 AG_{L^*}, and a REPAIR procedure, which are tightly joined.

Let M_1 and M_2 be two programs, and let P be a property. The classical Assume-Guarantee (AG) proof rule [24] assures that if we find an assumption A (in our case, a communicating program) such that $M_1||A \vDash P$ and $M_2 \vDash A$ both hold, then $M_1||M_2 \vDash P$ holds as well. For LTSs [7], the AG-rule is guaranteed to either prove correctness or return a real (non-spurious) counterexample. The work in [7] relies on the L^* algorithm [3] for learning an assumption A for the AG-rule. In particular, L^* aims at learning A_w, the weakest assumption for which $M_1||A_w \vDash P$. A crucial point of this method is the fact that A_w is *regular* [13], thus can be learned by L^*.

Lemma 1. *For infinite-state communicating programs, the weakest assumption A_w is not always regular.*

Example 3. Consider the programs M_1, M_2 and the property P of Figure 5. The weakest assumption with which M_1 satisfies P should contain exactly all traces (over the alphabet of M_2) that contain equally many actions of the form $x := x + 1$ and $y := y + 1$. This set of traces is not regular, and therefore cannot be learned by L^*.

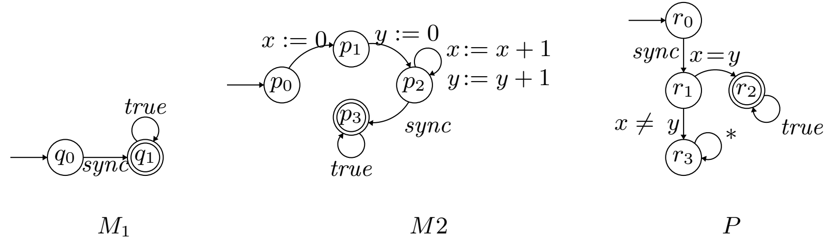

Fig. 5: A system for which the weakest assumption is not regular.

To cope with this difficulty, in our AG_{L^*} algorithm we change the goal to learning M_2 which is regular, as it is an automaton. Note that in case that $M_1||M_2 \models P$, repair is never needed, and M_2 is a valid assumption. In the worst case, the procedure halts once it has learned M_2. If $M_1||M_2 \not\models P$ then there does not exist a matching assumption, and attempting to learn M_2 will reveal this. Therefore, using $\mathcal{T}(M_2)$ as a learning goal matches the AG rule. The nature of AG_{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 [7], AG_{L^*} often terminates with an assumption A that is much smaller than M_2. Indeed, our tool often produces very small assumptions (see Section 5).

As mentioned before, not only that 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 AG_{L^*} algorithm returns an error trace t as a witness for the violation. In this case, we initiate the REPAIR procedure, which eliminates t from M_2. Moreover, REPAIR applies abduction in order to learn a new constraint which, when added to t, creates an infeasible trace. The new constraint enriches the alphabet in a way which may make similar traces infeasible as well. We elaborate on our use of abduction in Section 4.2. The removal of t and the addition of the new constraint result in a new goal M_2' for AG_{L^*} to learn. We now return to AG_{L^*} 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 membership queries previously asked for M_2, since the answer for them has not been changed. See Appendix for a proof that the difference between the languages of M_2 and M_2' lies in words (traces) whose membership has not yet been queried on M_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, since we are only guaranteed to remove one (bad) trace and add one (infeasible) trace in every such iteration (although in practice, every iteration may remove a larger set of traces. See Section 4.3). Thus, we may never converge to a regular 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 AG_{L^*} part of our AGR algorithm deviates from the AG-rule of [7] in two important ways. First, since the goal of our learning is M_2 rather than 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. The resulting process is thus more efficient. Indeed,

in our experiments we encountered several cases in which all repairs were invoked from the membership phase. Then, AGR got to the equivalence query only when the repaired component M_2' already guaranteed $M_1||M_2' \models P$ and terminated successfully.

4.1 The Assume-Guarantee-Repair (AGR) Algorithm

We now describe our AGR algorithm in more detail (see Algorithm 1). Figure 6 describes the flow of the algorithm. AGR comprises two main parts, namely AG_{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 (we can bound the number of iterations, as we discuss in Section 4.4).

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 2, and ends either when AG_{L^*} successfully terminates (line 16) or when procedure REPAIR is called (lines 7 and 24). When a new system M_2^i is constructed, AG_{L^*} does not start from scratch. The information that has been used 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 (lines 9,11,20, 27).

AG_{L^*} consists of two phases: membership, and equivalence.

The membership phase (lines 2-11) consists of a loop in which the learner constructs the next assumption A_j^i according to answers it gets from the teacher on a sequence of membership queries on various traces. These queries are answered with accordance to traces we allow in A_j^i: traces in M_2^i that in parallel with M_1 satisfy P. If a trace t in M_2^i in parallel with M_1 does not satisfy P, then t is a bad behavior of M_2. Therefore, if such a t is found during the membership phase, REPAIR is invoked.

Once the learner reaches a stable assumption A_j^i, it passes it to the equivalence phase (lines 12-27). 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 successfully 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 in A_j^i is also in M_2^i, then t_2 is a bad behavior of M_2^i, and here too the REPAIR phase is invoked. Otherwise, AGR returns to the membership phase with t_2 as a counter example for A_j^i, and continues to learn it.

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 subtrace t_2 from M_2^i should be removed from M_2^i. In this case, REPAIR returns to AG_{L^*} with a new goal $M_2^{i+1} \subseteq M_2^i \setminus \{t_2\}$ to be learned, along with the answer "no" to the membership query on t_2. In 4.3 we discuss different methods for removing t_2 from M_2^i.

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. We deduce c using abduction, see Section 4.2. As before, REPAIR returns to AG_{L^*} with a new goal to be learned, but now also with an extended alphabet. The membership phase is then provided with two new answers to the membership query: t_2 that should *not* be included in the new assumption, and $(t_2 \cdot c)$ that should be included.

Incremental learning One of the advantages of AGR is that it is *incremental*, in the sense that membership answers from previous iterations remain unchanged for the repaired sys-

tem. Indeed, since this is the first time that AG_{L^*} queries t_2, we can return to AG_{L^*} with the answer $t_2 \notin \mathcal{T}(M_2^{i+1})$, without contradicting any previous queries. In addition, t_2' obtained by abduction is a new word (over a new alphabet), which was also not queried earlier. Therefore, we can incrementally add t_2 and t_2' as answers from the teacher, and continue to use answers from previous queries on all other traces.

Algorithm 1 AGR

```
1:  function AG_{L*}
2:      //Membership Queries
3:      Let t_2 ∈ (αM_2^i)*.
4:      if t_2 ∈ T(M_2^i) then
5:          if M_1||t_2 ⊭ P then
6:              Let t ∈ (M_1||t_2) × P be an error trace.        ▷ t is a cex proving M_1||M_2^i ⊭ P
7:              REPAIR(M_2^i, t)
8:          else                                                  ▷ M_1||t_2 ⊨ P
9:              Return to AG_{L*} in Line 2 with t_2 ∈ T(A_j^i).
10:     else                                                      ▷ t_2 ∉ T(M_2^i)
11:         Return to AG_{L*} in Line 2 with t_2 ∉ T(A_j^i).
12:     //Equivalence Queries
13:     Let A_j^i be the candidate assumption generated by the learner.
14:     if M_1||A_j^i ⊨ P then
15:         if T(M_2^i) ⊆ T(A_j^i) then
16:             Terminate and return M_1||M_2^i ⊨ P.
17:         else
18:             Let t_2 ∈ T(M_2^i) \ T(A_j^i).
19:             Set j := j + 1
20:             Return to AG_{L*} in Line 2 with t_2 ∈ T(A_j^i).
21:     else                                                      ▷ M_1||A_j^i ⊭ P
22:         let t ∈ (M_1||A_j^i) × P be an error trace, and denote t = (t_1||t_A) × t_P.
23:         if t_A ∈ T(M_2^i) then
24:             REPAIR(M_2^i, t_A)                               ▷ t_A is a cex proving M_1||M_2^i ⊭ P
25:         else
26:             Set j := j + 1.
27:             Return to AG_{L*} in Line 2 with t_A ∉ T(A_j^i).
28: function REPAIR(M_2^i, t)
29:     Let t_1 ∈ M_1, t_2 ∈ M_2^i, t_p ∈ P such that t = (t_1||t_2) × t_p.
30:     if t does not contain constraints then
31:         Return to AG_{L*} in Line 2 with M_2^{i+1} = T(M_2^i) \ {t_2} and t_2 ∉ T(A_0^{i+1}).
32:     else                                                      ▷ t contains constraints
33:         Use abduction to eliminate t.
34:         Let c be the new constraint learned during abduction.
35:         Update αM_2^{i+1} = αM_2^i ∪ {c}.
36:         Let t_2' = t_2 · c  be the output of the abduction
37:         Return to AG_{L*} in Line 2 with M_2^{i+1} = (T(M_2^i) \ {t_2}) ∪ {t_2'},
38:         and t_2 ∉ T(A_0^{i+1}), t_2' ∈ T(A_0^{i+1})
```

4.2 Repair by Abduction

We now describe the repair we apply to M_2^i, in case the error trace t contains constraints (see Algorithm 1, line 32). Error traces with no constraints are removed from M_2^i syntactically (line 31), while in abduction we *semantically* eliminate t by making it infeasible. The

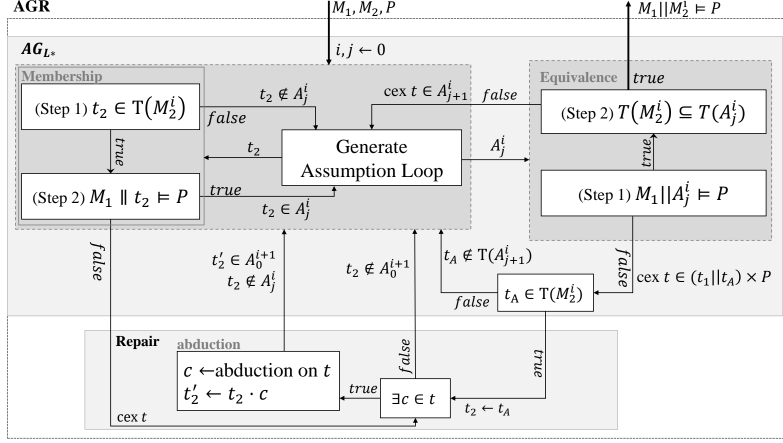

Fig. 6: The flow of AGR

new constraints are then added to the alphabet of M_2^i in a way that may eliminate additional erroronous traces. Note that the constraints added by abduction can only restrict the behavior of M_2, making more traces infeasible. As a result, counterexamples are never added to M_2 in this process.

The process of inferring new constraints from known facts about the program is called *abduction* [9]. We now describe how we apply it. Given a trace t, let φ_t be the first-order formula (a conjunction of constraints), which constitute the SSA representation of t [2]. In order to make t infeasible, we look for a formula ψ such that $\psi \wedge \varphi_t \rightarrow \mathit{false}$[7].

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 in X_2 only.

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 X_2. We now use quantifier elimination [26] to produce a quantifier-free formula over X_2. Computing ψ is similar to the abduction suggested in [9], but the focus here is on finding a formula over X_2 rather than over any minimal set of variables. We use Z3 [8] to apply quantifier elimination and to generate the new constraint. After generating $\psi(X_2)$, we add it to the alphabet of M_2^i (line 35 of Algorithm 1). 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.

Lemma 2. *Let $t = (t_1||t_2) \times t_P$. Then if t_2 is infeasible, t is infeasible as well.*

This is due to the fact that t_p can only restrict the behaviors t_1 and t_2, thus if t_2 is infeasible, t cannot be made feasible. See Appendix for a formal proof. Therefore, by making t_2 infeasible, we eliminate the error trace t.

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, q', and add a transition labeled $\psi(X_2)$ from q to q', where only q' is now accepting. Thus, we eliminated a violating trace from $M_1||M_2^i$. AGR now returns to AG_{L^*} in order to learn an assumption for the repaired component M_2^{i+1}, which now includes t_2' but not t_2.

[7] 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.

4.3 Removal of Error Traces

Recall that the goal of REPAIR is to remove a bad trace t from M_2 once it is found by AG_{L^*}. If t contain constraints, we remove it by using abduction (see Section 4.2). If t does not contains constraints, we can remove it by computing a system whose language is $\mathcal{T}(M_2) \setminus \{t\}$. However, removing a single trace at a time may lead to slow convergence, and to an exponential blow-up in 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 quicker convergence, we have implemented two additional heuristics for removing error traces that do not contain constraints. 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, this does not affect the correctness of the repair, 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 removing a set of error traces in a single step.

On top of the *exact* method we have described above, we have implemented two additional methods: *approximate* and *aggressive*. We briefly survey these three methods.

- *Exact.* To eliminate only t from M_2, we construct a program (an automaton) for all traces except for t and intersect it with M_2.
- *Approximate.* Similarly to our repair via abduction in Section 4.2, we prevent the last transition that t takes from reaching an accepting state. To do so, we duplicate q, the state that t reaches, to create a new accepting state q', to which all in-going transitions to q are diverted, except for the last transition on t. We then remove q from the set of accepting states. This way, some traces that lead to q are preserved by reaching q' instead, and some traces that share the last transition of t are eliminated along with t. As we have argued, these transitions may also be erroneous.
- *Aggressive.* In this simple method, we remove the state q that t reaches from the set of accepting states. This way we eliminate t along with all other traces that lead to q. In case that every accepting state is covered by some error trace, this repair might result in an empty language, creating a trivial repair. However, our experiments show that while this may happen, in most cases this method quickly leads to a non-trivial repair.

As we have explained, since all three methods only remove traces from M_2 without adding new ones, the resulting repair is guaranteed to be valid.

4.4 Correctness and Termination

As we have discussed in the beginning of Section 4, AGR is not guaranteed to terminate, and there are cases where the REPAIR stage may be called infinitely often. Since communicating programs use first-order constraints, the problem is undecidable, and so one cannot hope for a sound and complete algorithm. 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 AG_{L^*} is to syntactically learn M_2^i, which is regular, this stage will terminate at the latest when AG_{L^*} learns exactly M_2^i, and check for satisfaction (it may terminate sooner if a richer satisfying assumption is found). Notice 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 is found in M_2^i, in which case a new system M_2^{i+1}, that does not include t, is produced by REPAIR. If $M_1||M_2^i \not\models P$, then an error trace is guaranteed to be found by AG_{L^*} either in the membership or equivalence

phase. Therefore, also in case of dissatisfaction, the iteration is guaranteed to terminate. To conclude, we have the following.

Theorem 1. – *An iteration i of AGR ends with an error trace t iff $M_1||M_2^i \nvDash P$, where M_2^i is the repaired system at iteration i.*

– *If, after finitely many iterations, a repaired program M_2' is such that $M_1||M_2' \vDash P$, then AGR terminates with a correct answer.*

We have shown that every iteration of AGR by itself is guaranteed to terminate with a correct answer. The detailed correctness proofs are in the Appendix.

Moreover, AGR is *incremental*, in the sense that every repaired system M_2' that is produced by REPAIR is guaranteed to contain less bad traces than the previous system in the previous iteration.

Notice that if there are finitely many error traces in M_2, then AGR is guaranteed to terminate with a correct answer. Indeed, every iteration of AGR finds and removes an error trace t, and no new erroneous traces are introduced in the system post REPAIR. Therefore, finitely many error traces lead to the termination of AGR with a correctly repaired system.

Nevertheless, there are cases in which there are infinitely many error traces, causing our algorithm not to terminate. To avoid divergence in such cases, we specify a bound k and stop AGR after k iterations. If the last iteration resulted in A_j^k such that $M_1||A_j^k \vDash P$, then we can return a program whose traces are $\mathcal{T}(M_2) \cap \mathcal{T}(A_j^k)$, which is a valid repair.

5 Experimental Results

We implemented our AGR framework in Java, integrating L^* implementation from the LTSA tool [18]. We used Z3 [8] for verification and abduction. For each of the examples needed repair, we tested the three repair methods: aggressive, approximate and exact. Figure 7 presents comparisons between the three methods in terms of run-time and the size of the repair. Table 1 consists results of AGR on various examples. Column *Iterations* stands for the number of iterations of the verify-repair loop, until a repaired M_2 is achieved.

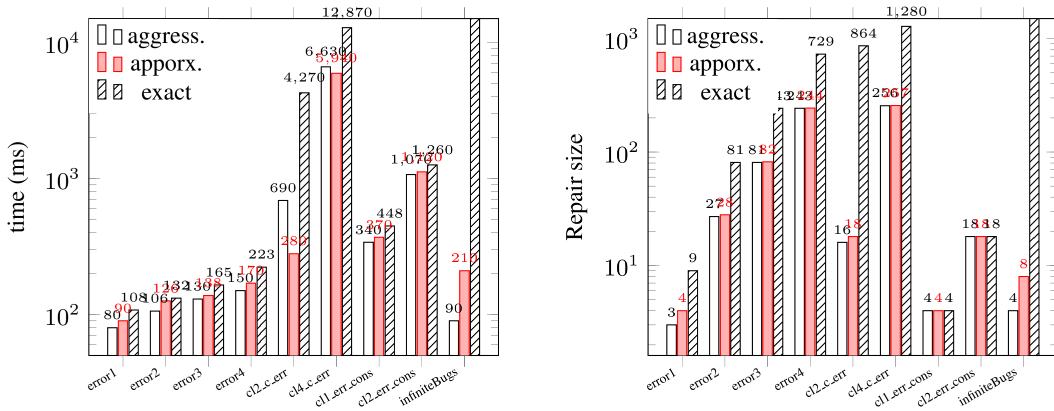

Fig. 7: Comparing repair methods: time and repair size. Logarithmic scale

As shown in the table, for most examples our tool generates assumptions that are significantly smaller (in some cases, a factor of 20) than the repaired and the original M_2. In addition, for examples needed repair, in most cases our tool needed 2-5 iterations of repair in order to successfully construct a repaired component. Example *infiniteBugs* models

a simple structure in which due to a loop in M_2, the same alphabet sequence can generate infinitely many error traces. Therefore, the *exact* repair method timed out, since it attempted to remove one error trace at a time. On the other hand, the *aggressive* repair method removes all accepting states, creating an empty program, a trivial (yet valid) repair. Examples with prefix *client* or *protocol* describe correct behaviour of a server communicating with several clients, while examples contain *client_error* describe erroneous protocols that require both semantic and syntactic repairs, which our tool finds and repairs.

Example	M_1 Size	M_2 Size	P Size	Time (sec.)	A size	Repair Size	Repair Method	Iterations
constraintsComplex	4	4	3	0.2	3	verification		
constraintsComplex2	16	16	3	1.8	4	verification		
constraintsComplex3	32	32	3	11.1	6	verification		
constraintsComplex4	64	64	3	95	7	verification		
error1	2	3	2	0.08	3	3	aggress.	2
				0.09	4	4	approx.	2
				0.108	6	9	exact	2
error2	2	27	2	0.106	5	27	aggress.	2
				0.126	6	28	approx.	2
				0.132	8	81	exact	2
error3	2	81	2	0.13	6	81	aggress.	2
				0.138	7	82	approx.	2
				0.165	9	243	exact	2
error4	2	243	2	0.15	8	243	aggress.	2
				0.17	8	244	approx.	2
				0.223	10	729	exact	2
client1	2	4	3	0.093	3	verification		
client2	3	16	4	0.29	13	verification		
client4	5	256	6	4.88	92	verification		
client1_c	2	4	3	0.08	3	verification		
client2_c	3	16	4	0.22	10	verification		
client4_c	5	256	6	4.44	109	verification		
client2_c_error	3	16	5	0.69	12	16	aggress.	5
				0.28	13	18	approx.	3
				4.27	44	864	exact	5
client4_c_error	4	256	8	6.63	113	256	aggress.	2
				5.94	113	257	approx.	2
				12.87	155	1280	exact	2
client1_c_constraints	2	3	4	0.075	3	verification		
client1_error_cons	2	3	4	0.34	5	4	aggress.	2
				0.37	5	4	approx.	2
				0.488	5	4	exact	2
client2_error_cons	3	16	5	1.07	18	18	aggress.	3
				1.12	18	18	approx.	3
				1.26	18	18	exact	3
Protocol1	9	6	15	0.1	6	verification		
Protocol2	11	13	17	0.18	11	verification		
infiniteBugs	2	4	2	0.09	1	4 (trivial)	aggress.	4
				0.21	6	8	approx.	5
				timeout			exact	timeout

Table 1: AGR algorithm results on various examples

References

1. A. Albarghouthi, I. Dillig, and A. Gurfinkel. Maximal specification synthesis. In *POPL*, 2016.
2. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL*, 1988.
3. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2).
4. S. Chaki and O. Strichman. Optimized L*-based assume-guarantee reasoning, TACAS, 2007.
5. Y.-F. Chen, E. M. Clarke, A. Farzan, M.-H. Tsai, Y.-K. Tsay, and B.-Y. Wang. Automated assume-guarantee reasoning through implicit learning. In *CAV*, 2010.
6. Y.-F. Chen, A. Farzan, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Learning minimal separating DFA's for compositional verification. In *TACAS*, 2009.
7. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS 2003*.
8. L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.
9. I. Dillig and T. Dillig. Explain: A tool for performing abductive inference. In *CAV*, 2013.
10. K. A. Elkader, O. Grumberg, C. S. Pasareanu, and S. Shoham. Automated circular assume-guarantee reasoning. In *FM*, 2015.
11. K. A. Elkader, O. Grumberg, C. S. Pasareanu, and S. Shoham. Automated circular assume-guarantee reasoning with n-way decomposition and alphabet refinement. In *CAV*, 2016.
12. M. Gheorghiu, D. Giannakopoulou, and C. S. Pasareanu. Refining interface alphabets for compositional verification. In *TACAS*, 2007.
13. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK*, pages 3–12. IEEE Computer Society, 2002.
14. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Component verification with automatically generated assumptions. *Autom. Softw. Eng.*, 12(3):297–320, 2005.
15. A. Gupta, K. L. McMillan, and Z. Fu. Automated assumption generation for compositional verification. *Formal Methods in System Design*, 32(3):285–301, 2008.
16. B. Li, I. Dillig, T. Dillig, K. L. McMillan, and M. Sagiv. Synthesis of circular compositional program proofs via abduction. In *TACAS 2013*.
17. S. Lin and P. Hsiung. Compositional synthesis of concurrent systems through causal model checking and learning. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 416–431, 2014.
18. J. Magee and J. Kramer. *Concurrency - state models and Java programs*. Wiley, 1999.
19. K. L. McMillan. Circular compositional reasoning about liveness. In *CHARME*, 1999.
20. J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.
21. K. S. Namjoshi and R. J. Trefler. On the competeness of compositional reasoning. In *CAV*, 2000.
22. C. S. Pasareanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 2008.
23. C. Peirce and C. Hartshorne. *Collected Papers of Charles Sanders Peirce*. Belknap Press, 1932.
24. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems, NATO ASI Series*, 1985.
25. R. Singh, D. Giannakopoulou, and C. S. Pasareanu. Learning component interfaces with may and must abstractions. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 527–542, 2010.
26. V. Weispfenning. Quantifier elimination and decision procedures for valued fields. *Models and Sets.Lecture Notes in Mathematics (LNM)*, 1103:419—472, 1984.

Appendix

A Proofs: Traces and Feasible Traces

Definition 8. *Let t be a trace over alphabet α, and let $\alpha' \subseteq \alpha$. We denote by $t \downarrow_{\alpha'}$ the trace obtained from t by omitting all letters in t that are not in α'.*

Example 4. Let $\alpha = \{g!x, x := x+1, x < 10\}$, $\alpha' = \{g!x, x := x+1\}$ and let $t = (x := x+1, x := x+1, x < 10, g!x)$. Then $t\downarrow_{\alpha'} = (x := x+1, x := x+1, g!x)$.

Lemma 3. *Let M be a 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.*

Proof. Let $M = (Q_M, X_M, \alpha M, \delta_M, q_0^M, F_M)$ and $M = (Q_P, X_P, \alpha P, \delta_P, q_0^P, F_P)$, and denote $M \times P = (Q, X_M, \alpha, \delta, q_0, F)$

Let $p = (q_0, c_1, \cdots, c_m, q_m)$ be the path in $M \times P$ such that t is induced from p. Denote by $t_M = t \downarrow_{\alpha M} = (c_{i_1}, \cdots, c_{i_n})$.

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 true since from the definition of δ, if a_i is not in αM, then $\delta((q'_M, q'_P), a_i) = (q'_M, \delta_P(q'_P))$. Informally, it means that δ only advances in the P component and not in the M component.

We now inductively prove that $\forall j \geq 1 : (c_{i_1}, \cdots, c_{i_j})$ is a trace of M. Let $j := 1$ and denote $k := i_1$. Then $c_1, \ldots, c_{k-1} \notin \alpha M$. Thus, $\forall 1 \leq i < k : \exists q_P^i : \delta((q_0^M, q_{i-1}^P), c_i) = (q_0^M, q_i^P)$. For $c_{i_1} = c_k \in \alpha M$, from the definition of δ, we have $\delta((q_0^M, q_{k-1}^P) = (\delta_M(q_0^M, c_{i_1}), q')$ from some $q' \in Q_P$. Then indeed, $(q_0^M, c_{i_1}, \delta_M(q_0^M, c_{i_1}))$ is a path in M, making (c_{i_1}) a trace of M.

Let $j > 1$, and assume $t_{j-1} = (c_{i_1}, \cdots, c_{i_{j-1}})$ is a trace of M. Let $(q_0, c_{i_1}, \cdots, c_{i_{j-1}}, q_{j-1})$ a path 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 $\forall k < l < k+m : \exists q_P^l : \delta(q_{j-1}, q_P^{l-1}), c_l) = (q_{j-1}, q_P^l)$. For c_{i_j} it holds that $\delta(q_{j-1}, q_P^{k+m-1}), c_{i_j}) = (\delta(q_{j-1}, c_{i_j}), q'))$ for some $q' \in Q_P$. Thus $(c_{i_1}, \cdots, c_{i_j})$ is a trace of M, as needed. □

Lemma 4. *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.*

The proof is almost the same as the proof of Lemma 3.

Lemma 5. *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 a run u on t. Denote $u = (\beta_0, b_1, \beta_1, \cdots, b_n, \beta_n)$ and $t = (b_1, \cdots, b_n)$. We inductively build a run r on $t \downarrow_{\alpha M}$. The existence of such a run r proves that $t \downarrow_{\alpha M}$ is feasible.

Let $t \downarrow_{\alpha M} = (c_1, \cdots, c_k)$. We define $r = (\gamma_0, c_1, \gamma_1, \cdots, c_k, \gamma_k)$ as follows.

- Set $j := 0, i := 0$.
- Define $\gamma_0 := \beta_0$ and set $j := j + 1$.
- Repeat until $j = k$: Let $i' > i$ be the minimal index such that $b_{i'} = c_j$. Then, 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. This is true since if b_l is not a constraint, then $b_l \in \alpha M$. But in that case, b_l has to 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 a run, 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, it holds that $\beta_{i'-1} = \beta_i = \gamma_{j-1}$. Now, since $b_{i'} = c_j$, we can assign γ_j to be the same as $b_{i'}$ and result in a valid assignment. Thus, r is a valid run on $t \downarrow_{\alpha M}$, making $t \downarrow_{\alpha M}$ feasible as needed. □

Lemma 6. *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 6 is different from the proof of Lemma 5, since here we cannot 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 variables of $M \times P$ are the same as the variables of M, and the two runs only differ on the constraints that are added to the trace of $M \times P$. Here M_1 and M_2 are defined over two different sets of variables, with empty intersection between them. Nevertheless, The proof is similar to the proof of Lemma 5.

Proof. Denote by X_i the set of variables of M_i for $i \in \{1, 2\}$. Let $t \in M_1||M_2$ be a feasible trace. Then, there exists a run u on t. Denote $u = (\beta_0, b_1, \beta_1, \cdots, b_n, \beta_n)$ and $t = (b_1, \cdots, b_n)$. We build a run r on $t \downarrow_{\alpha M_1}$ as follows (in the same way, we can build a run on $t \downarrow_{\alpha M_2}$). Let $t \downarrow_{\alpha M_1} = (c_1, \cdots, c_k)$. We define $r = (\gamma_0, c_1, \gamma_1, \cdots, c_k, \gamma_k)$ as follows.

- Set $j := 0, i := 0$.
- Define $\gamma_0 := \beta_0(X_1)$ and set $j := j + 1$.
- Repeat until $j = k$: Let $i' > i$ be the minimal index such that $b_{i'} = c_j$ or $b_{i'} = (g * x, g * y)$ and $c_j = g * x$. Then, define $\gamma_j := \beta_{i'}(X_1)$ and set $j := j + 1, i := i' + 1$.

Note that for each $i < l < i'$ is holds that $b_l \notin \alpha M_1$ and there are no $x \in X_1, y \in X_2$ such that $b_l = (g * x, g * y)$ for $g * x \in \alpha M_1$. Otherwise, b_l is either a synchronization between M_1 and M_2, or b_l is a letter of M_1 that must synchronize with $t \downarrow_{\alpha M_1}$. Both are contradiction to the fact that i' is the minimal index for which $b_{i'}$, c_j synchronize. Thus, since u is a run, and for all $i < l < i'$, b_l does not contain variables of X_1, by the definition of a run, it holds that $\beta_i(X_1) = \beta_l(X_1)$, since an assignment to a variable may only change if the variable is involved in the action alphabet. In particular, it holds that $\beta_{i'-1}(X_1) = \beta_i(X_1) = \gamma_{j-1}(X_1)$. Now, we can assign γ_j to be the same as $b_{i'}(X_1)$ and result in a valid assignment, as needed. □

B Soundness and Completeness of AG Rule for Communicating Programs

Theorem 2. *For communicating programs, the Assume-guarantee rule is sound and complete. That is:*

- *Soundness: 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. Completeness. If $M_1||M_2 \models P$, then we can choose $A = M_2$, and then it holds that $M_1||A \models P$ and $\mathcal{T}(M_2) \subseteq \mathcal{T}(A)$.

Soundness. Assume by a 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 $t \in (M_1||M_2) \times P$. From Lemma 3 and Lemma 4, it holds that $t_2 = t \downarrow_{\alpha M_2} \in \mathcal{T}(M_2)$ and is feasible. Since $\mathcal{T}(M_2) \subseteq \mathcal{T}(A)$, it holds that t is an error trace in $(M_1||A) \times P$, contradicting $M_1||A \models P$. □

C Incremental Learning

An important feature of our AGR algorithm is its *incrementality*. We thus prove the following.

Theorem 3. *Let $\mathcal{T}(M_2^i)$ be the language learned by phase i of the AGR algorithm. Assume that phase i ended with a counterexample t and initiated a call to REPAIR(M_2^i, t). Then:*

1. *For every trace t_2 that was queried before, the answer remains the same for $\mathcal{T}(M_2^{i+1})$.*
2. *AGR did not query $t \downarrow_{\alpha M_2^i}$ before, thus removing it from $\mathcal{T}(M_2^{i+1})$ is consistent with all previous queries.*
3. *If t_2' is a trace learned using abduction, then AGR did not query it before.*

Proof. Item 1. Consider the two cases. If t_2 was answered "$t_2 \in \mathcal{T}(A^j)$" for some previous iteration j , then in particular it holds that $M_1||t_2 \models P$ (line 9 of AGR algorithm). Since M_1 and P are remained unchanged, then in all next iterations the same holds for t_2. Since We only syntactically remove a trace t from $\mathcal{T}(M_2^j)$ if is it an error trace, we conclude that for every j, t_2 is never removed from $\mathcal{T}(M_2^j)$. Thus, line 9 always holds for t_2.

If t_2 was queried before and the answer was "$t_2 \notin \mathcal{T}(A^j)$", then one of the following holds; $M_1||t_2 \not\models P$ and as in the previous case, this remains true for all future iterations; Or, $t_2 \notin \mathcal{T}(M_2^j)$ for some previous iteration j. Since we only remove traces, it holds that for every $j < j', \mathcal{T}(M_2^j) \subseteq \mathcal{T}(M_2^{j'})$. Thus, $t_2 \notin \mathcal{T}(M_2^{i+1})$ as needed.

Item 2. Assume by a way of contradiction that AGR queried $t \downarrow_{\alpha M_2^i}$ before. Since AGR called REPAIR, it holds that t is an error trace in $(M_1||t \downarrow_{\alpha M_2^i}) \times P$. Thus, it cannot be the case the AGR answered "$t \downarrow_{\alpha M_2^i} \in \mathcal{T}(A^j)$" on $t \downarrow_{\alpha M_2^i}$ previously (from Item 1). Moreover, if $t \downarrow_{\alpha M_2^i}$ was queried in a previous iteration and was answered "no", i.e., "$t \downarrow_{\alpha M_2^i} \notin \mathcal{T}(A^j)$", then due to the nature of L^*, $t \downarrow_{\alpha M_2^i}$ cannot be queried again.

Item 3 Since t_2' contains a new alphabet letter, it for sure was not queried in any previous iteration.

□

D Correctness of AGR Algorithm

We now prove the Correctness of AGR algorithm. In particular, the proof of Theorem 4 follows.

Theorem 4. 1. *If $M_1||M_2 \models P$ then AGR terminates with a correct answer.*
2. *If an iteration i of AGR ends with an error trace t, then $M_1||M_2^i \not\models P$, where M_2^i is the updated system at iteration i.*
3. *If $M_1||M_2 \not\models P$ then M_2', the system post REPAIR, contains less bad traces than M_2.*
4. *If, after finitely many iterations, a repaired program M_2' is such that $M_1||M_2' \models P$, then AGR terminates with a correct answer. (Note that this is a generalization of the first part of the theorem).*

Theorem 5. *Every iteration of the AGR algorithm terminates and is consistent with $\mathcal{T}(M_2^i)$. That is, whenever AGR returns "$t \in \mathcal{T}(A_j^i)$" to L^* (lines 9, 20) then indeed $t \in \mathcal{T}(M_2^i)$, and whenever AGR returns "$t \notin \mathcal{T}(A_j^i)$" to L^* (lines 11, 27) then indeed $t \notin \mathcal{T}(M_2^i)$.*

Proof. For one iteration of AGR, we show that both membership queries and equivalence queries are consistent with $\mathcal{T}(M_2^i)$. Since L^* is an algorithm for learning a regular language, and since $\mathcal{T}(M_2^i)$ is a regular language, by the termination of L^* we conclude that each iteration terminates.

To show consistency with $\mathcal{T}(M_2^i)$, note that for $t_2 \in (\alpha M_2^i)^*$, membership queries are of the form"is $t_2 \in \mathcal{T}(M_2^i)$?". The only case in which the algorithm does not return the same answer as the teacher does, is when $t_2 \in \mathcal{T}(M_2^i)$ and $M_1||t_2 \nvDash P$. In this case we conclude that $M_1||M_2^i \nvDash P$ and thus REPAIR is called and the iteration terminate immediately. Therefore membership queries are consistent with $\mathcal{T}(M_2^i)$.

As for equivalence queries, the teacher returns counterexamples to L^* in lines 20 and 27. In line 20 we find a trace $t_2 \in \mathcal{T}(M_2^i)$, therefore the answers on t_2 should be "yes", as we indeed return. In line 27, we found $t_2 \notin \mathcal{T}(M_2^i)$, thus the answer should be "no", as we indeed return. Therefore, the L^* algorithm constantly converges towards $\mathcal{T}(M_2^i)$. □

Theorem 6. *If $M_1||M_2^i \vDash P$, the AGR algorithm terminates. Otherwise, AGR finds a counterexample, proving that $M_1||M_2^i \nvDash P$ (and continues to repairing M_2^i).*

Proof. Termination follows directly from Theorem 5. Since all answers are syntactic according to $\mathcal{T}(M_2^i)$, which is a regular language, from the correctness of L^* algorithm we conclude that the algorithm will eventually learn $\mathcal{T}(M_2^i)$. Note that a phase of the algorithm terminates in lines 5, 16 or 24. We are left to prove that if the algorithm learns exactly $\mathcal{T}(M_2^i)$, then one of the three cases holds.

If $M_1||M_2^i \vDash P$, then since the learned assumption A is exactly $\mathcal{T}(M_2^i)$, it holds that $M_1||A \vDash P$ and $\mathcal{T}(M_2^i) \subseteq \mathcal{T}(A)$, thus the algorithm terminates in line 16.

If $M_1||M_2^i \nvDash P$, then there exists an error trace $t \in (M_1||M_2) \times P$. From Lemmas 5, 6 it holds that $t \downarrow_{\alpha M_2^i}$ is feasible in M_2. In particular, it holds that t is an error trace of $(M_1||t \downarrow_{\alpha M_2^i}) \times P$. Thus, $M_1||t \downarrow_{\alpha M_2^i} \nvDash P$. Since we learned exactly $A = \mathcal{T}(M_2^i)$, it holds that $t_2 \in \mathcal{T}(A)$. Thus, either t_2 shows up as a membership query, and then line 5 holds, proving $M_1||M_2^i \nvDash P$ and call REPAIR, terminating the iteration; or AGR continues to the equivalence query part in which it holds that $M_1||A \nvDash P$ and every trace t_A we find in this case is such that $t_A \in \mathcal{T}(M_2^i)$, resulting in termination of the iteration in line 24, again, by calling REPAIR. □

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

Theorem 7. *L^* terminates and returns the correct answer. That is:*

1. *If L^* outputs an assumption A, then there exists i such that $M_1||A \vDash P$ and $\mathcal{T}(M_2^i) \subseteq \mathcal{T}(A)$, thus we can conclude $M_1||M_2^i \vDash P$.*
2. *If a phase i of AGR ends with an error trace t, then $M_1||M_2^i \nvDash P$.*

Proof. Assume AGR return an assumption A. We can then conclude that the following holds for A: there exists i such that $\mathcal{T}(M_2^i) \subseteq \mathcal{T}(A)$ and $M_1||A \vDash P$. From the soundness of our AG rule (Theorem 2) it holds that $M_1||M_2^i \vDash P$.

Assume now that a phase i of AGR ends with an error trace t. We prove that $M_1||M_2^i \nvDash P$. First note that AGR may output such a trace both while making a membership query and while making an equivalence query. If t was found during a membership query(line 5), then there exists $t_2 \in T(M_2^i)$ such that $M_1||t_2 \nvDash P$, 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 $M_1||M_2^i \nvDash P$.

If t was found during an equivalence query(line 24), 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 $M_1||M_2^i \nvDash P$. This finishes the proof.

□