Proof Rules for Automated Compositional Verification through Learning

Howard Barringer [*]
Department of Computer Science
University of Manchester
Oxford Road, Manchester, M13 9PL
howard@cs.man.ac.uk

Dimitra Giannakopoulou
RIACS/USRA
NASA Ames Research Center
Moffett Field, CA 94035-1000, USA
dimitra@email.arc.nasa.gov

Corina S. Păsăreanu
Kestrel Technology LLC
NASA Ames Research Center
Moffett Field, CA 94035-1000, USA
pcorina@email.arc.nasa.gov

ABSTRACT
Compositional proof systems not only enable the stepwise development of concurrent processes but also provide a basis to alleviate the state explosion problem associated with model checking. An assume-guarantee style of specification and reasoning has long been advocated to achieve compositionality. However, this style of reasoning is often non-trivial, typically requiring human input to determine appropriate assumptions. In this paper, we present novel assume-guarantee rules in the setting of finite labelled transition systems with blocking communication. We show how these rules can be applied in an iterative and fully automated fashion within a framework based on learning.

Keywords
Parallel Composition, Automated Verification, Assumption Generation, Learning

1. INTRODUCTION
Our work is motivated by an ongoing project at NASA Ames Research Center on the application of model checking to the verification of autonomous software. Autonomous software involves complex concurrent behaviors for reacting to external stimuli without human intervention. Extensive verification is a pre-requisite for the deployment of missions that involve autonomy.

Given a finite model of a system and of a required property, model checking can be used to determine automatically whether the property is satisfied by the system. The limitation of this approach, commonly referred to as the "state-explosion" problem [7], is that it needs to store the explored system states in memory, which may be prohibitively large for realistic systems.

[*]This author is most grateful to RIACS/USRA and the UK's EPSRC under grant GR/S40435/01 for the partial support provided to conduct this research

Compositional verification presents a promising way of addressing state explosion. It advocates a "divide and conquer" approach where properties of the system are decomposed into properties of its components, so that if each component satisfies its respective property, then so does the entire system. Components are therefore model checked separately. It is often the case, however, that components only satisfy properties in specific contexts (also called environments). This has given rise to the application of assume-guarantee reasoning [16, 21] to model checking [11].

Assume-guarantee[1] reasoning first checks whether a component M guarantees a property P, when it is part of a system that satisfies an assumption A. Intuitively, A characterizes all contexts in which the component is expected to operate correctly. To complete the proof, it must also be shown that the remaining components in the system, i.e., M's environment, satisfy A. Several frameworks have been proposed [16, 21, 6, 14, 24, 15] to support this style of reasoning. However, their practical impact has been limited because they require non-trivial human input in defining assumptions that are strong enough to eliminate false violations, but that also reflect appropriately the remaining system.

In previous work [8], we developed a novel framework to perform assume-guarantee reasoning in an *iterative* and *fully automatic* fashion; the approach uses learning and model-checking. To check that a system made up of components M_1 and M_2 satisfies a property P, our framework automatically learns and refines assumptions for one of the components to satisfy P, which it then tries to discharge on the other component. Our approach is guaranteed to terminate, stating that the property holds for the system, or returning a counterexample if the property is violated.

This work introduces a variety of sound and complete assume-guarantee rules in the setting of Labeled Transition Systems with blocking communication. The rules are motivated by the need for automating assume-guarantee reasoning. How-

[1]The original terminology for this style of reasoning was rely-guarantee or assumption-commitment; it was introduced for enabling top-down development of concurrent systems.

ever, in contrast to our previous work, they are symmetric, meaning that they are based on establishing and discharging assumptions for both components at the same time. The remainder of this paper is organized as follows. We first provide some background in Section 2, followed by some basic compositional proof rules in Section 3. The framework that automates these rules is presented in Section 4. Section 5 introduces rules that optimize and extend the basic rules. Finally, Section 6 presents related work and Section 7 concludes the paper.

2. BACKGROUND

We use Labeled Transition Systems (LTSs) to model the behavior of communicating components in a concurrent system. In this section, we provide background on LTSs and their associated operators, and also present how properties are expressed and checked in our framework. We also summarize the learning algorithm that is used to automate our compositional verification approach.

2.1 Labeled Transition Systems

Let $\mathcal{A}ct$ be the universal set of observable actions and let τ denote a local action *unobservable* to a component's environment. An LTS M is a quadruple $\langle Q, \alpha M, \delta, q0\rangle$ where:

- Q is a non-empty finite set of states
- $\alpha M \subseteq \mathcal{A}ct$ is a finite set of observable actions called the *alphabet* of M
- $\delta \subseteq Q \times \alpha M \cup \{\tau\} \times Q$ is a transition relation
- $q0 \in Q$ is the initial state

An LTS $M = \langle Q, \alpha M, \delta, q0\rangle$ is *non-deterministic* if it contains τ-transitions or if $\exists (q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, M is *deterministic*.

Traces. A *trace* t of an LTS M is a sequence of observable actions that M can perform starting at its initial state. For $\Sigma \subseteq \mathcal{A}ct$, we use $t{\upharpoonright}\Sigma$ to denote the trace obtained by removing from t all occurrences of actions $a \notin \Sigma$. The set of all traces of M is called the *language* of M, denoted $\mathcal{L}(M)$. We will freely use the expression "a word t is accepted by M" to mean that $t \in \mathcal{L}(M)$. Note that the empty word is accepted by any LTS.

Parallel Composition. Let $M = \langle Q, \alpha M, \delta, q0\rangle$ and $M' = \langle Q', \alpha M', \delta', q0'\rangle$. We say that M *transits* into M' with action a, denoted $M \xrightarrow{a} M'$, if and only if $(q0, a, q0') \in \delta$ and $\alpha M = \alpha M'$ and $\delta = \delta'$.

The parallel composition operator $\|$ is a commutative and associative operator that combines the behavior of two components by synchronizing the actions common to their alphabets and interleaving the remaining actions.

Let $M_1 = \langle Q_1, \alpha M_1, \delta_1, q0_1\rangle$ and $M_2 = \langle Q_2, \alpha M_2, \delta_2, q0_2\rangle$ be two LTSs. Then $M_1 \parallel M_2$ is an LTS $M = \langle Q, \alpha M, \delta, q0\rangle$, where $Q = Q_1 \times Q_2$, $q0 = (q0_1, q0_2)$, $\alpha M = \alpha M_1 \cup \alpha M_2$, and δ is defined as follows, where a is either an observable action or τ (note that the symmetric rules are implied by the fact that the operator is commutative):

$$\frac{M_1 \xrightarrow{a} M_1',\ a \notin \alpha M_2}{M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2}$$

$$\frac{M_1 \xrightarrow{a} M_1',\ M_2 \xrightarrow{a} M_2',\ a \neq \tau}{M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2'}$$

Note. $\mathcal{L}(M_1 \parallel M_2) = \{t \mid t{\upharpoonright}\alpha M_1 \in \mathcal{L}(M_1) \wedge t{\upharpoonright}\alpha M_2 \in \mathcal{L}(M_2) \wedge t \in (\alpha M_1 \cup \alpha M_2)^*\}$

Properties and Satisfiability. A property is also defined as an LTS P, whose language $\mathcal{L}(P)$ defines the set of acceptable behaviors over αP. An LTS M satisfies P, denoted as $M \models P$, if and only if $\forall t \in \mathcal{L}(M).t{\upharpoonright}\alpha P \in \mathcal{L}(P)$.

2.2 LTSs and Finite-State Machines

As will be described in section 4, our proof-rules require the use of the "complement" of an LTS. LTSs are not closed under complementation (their languages are prefix-closed), so we need to define here a more general class of finite-state machines (FSMs) and associated operators for our framework.

An FSM M is a five tuple $\langle Q, \alpha M, \delta, q0, F\rangle$ where $Q, \alpha M, \delta$, and $q0$ are defined as for LTSs, and $F \subseteq Q$ is a set of accepting states.

For an FSM M and a word t, we use $\hat{\delta}(q, t)$ to denote the set of states that M can reach after reading t starting at state q. A word t is said to be *accepted* by an FSM $M = \langle Q, \alpha M, \delta, q0, F\rangle$ if $\hat{\delta}(q0, t) \cap F \neq \emptyset$. Note that in the following sections, the term trace is often used to denote a word. The *language accepted by* M, denoted $\mathcal{L}(M)$ is the set $\{t \mid \hat{\delta}(q0, t) \cap F \neq \emptyset\}$.

For an FSM $M = \langle Q, \alpha M, \delta, q0, F\rangle$, we use LTS(M) to denote the LTS $\langle Q, \alpha M, \delta, q0\rangle$ defined by its first four fields. Note that this transformation does not preserve the language of the FSM. On the other hand, an LTS is in fact a special instance of an FSM, since it can be viewed as an FSM for which all states are accepting. From now on, whenever we apply operators between FSMs and LTSs, it is implied that the LTS is treated as its corresponding FSM.

We call an FSM M *deterministic* iff LTS(M) is deterministic.

Parallel Composition. Let $M_1 = \langle Q_1, \alpha M_1, \delta_1, q0_1, F_1\rangle$ and $M_2 = \langle Q_2, \alpha M_2, \delta_2, q0_2, F_2\rangle$ be two FSMs. Then $M_1 \parallel M_2$ is an FSM $M = \langle Q, \alpha M, \delta, q0, F\rangle$, where:

- $\langle Q, \alpha M, \delta, q0\rangle = LTS(M_1) \parallel LTS(M_2)$, and
- $F = \{(s_1, s_2) \in Q_1 \times Q_2 \mid s_1 \in F_1 \wedge s_2 \in F_2\}$.

Note. $\mathcal{L}(M_1 \parallel M_2) = \{t \mid t{\upharpoonright}\alpha M_1 \in \mathcal{L}(M_1) \wedge t{\upharpoonright}\alpha M_2 \in \mathcal{L}(M_2) \wedge t \in (\alpha M_1 \cup \alpha M_2)^*\}$

Satisfiability. For FSMs M and P where $\alpha P \subseteq \alpha M$, $M \models P$ if and only if $\forall t \in \mathcal{L}(M).t{\upharpoonright}\alpha P \in \mathcal{L}(P)$.

Complementation. The complement of an FSM (or an LTS) M, denoted *coM*, is an FSM that accepts the complement of M's language. It is constructed by first making

M deterministic, subsequently completing it with respect to αM, and finally turning all accepting states into non-accepting ones, and vice-versa. An automaton is complete with respect to some alphabet if every state has an outgoing transition for each action in the alphabet. Completion typically introduces a non-accepting state and appropriate transitions to that state.

2.3 The L* Algorithm

In Section 4, we present a framework that automates compositional reasoning using a learning algorithm.

The learning algorithm (L*) used by our approach was developed by Angluin [2] and later improved by Rivest and Schapire [22]. L* learns an unknown regular language (U over an alphabet Σ) and produces a deterministic FSM C such that $\mathcal{L}(C) = U$. L* works by incrementally producing a sequence of candidate deterministic FSMs C_1, C_2, ... converging to C. In order to learn U, L* needs a *Teacher* to answer two type of questions. The first type is a *membership query*, consisting of a string $\sigma \in \Sigma^*$; the answer is *true* if $\sigma \in U$, and *false* otherwise. The second type of question is a *conjecture*, i.e. a candidate deterministic FSM C whose language the algorithm believes to be identical to U. The answer is *true* if $\mathcal{L}(C) = U$. Otherwise the Teacher returns a counterexample, which is a string σ in the symmetric difference of $\mathcal{L}(C)$ and U.

At a higher level, L* creates a table where it incrementally records whether strings in Σ^* belong to U. It does this by making membership queries to the Teacher. At various stages L* decides to make a conjecture. It constructs a candidate automaton C based on the information contained in the table and asks the Teacher whether the conjecture is correct. If it is, the algorithm terminates. Otherwise, L* uses the counterexample returned by the Teacher to extend the table with strings that witness differences between $\mathcal{L}(C)$ and U.

L* is guaranteed to terminate with a minimal automaton C for the unknown language U. Moreover, each candidate deterministic FSM C_i that L* constructs is smallest, in the sense that any other deterministic FSM consistent with the table has at least as many states as C_i. The candidates conjectured by L* strictly increase in size; each candidate is smaller than the next one, and all incorrect candidates are smaller than C. Therefore, if C has n states, L* makes at most $n-1$ incorrect conjectures.

3. COMPOSITIONAL PROOF RULES

3.1 Motivation

In our previous work on assumption generation and learning [12, 8], we used the following basic rule for establishing that a property P holds for a (closed) parallel composition of two software components M_1 and M_2.

Rule 0.

$$\begin{array}{rl} 1: & M_1 \parallel A_{M_1} \models P \\ 2: & M_2 \models A_{M_1} \\ \hline & M_1 \parallel M_2 \models P \end{array}$$

A_{M_1} denotes an assumption about the environment in which M_1 is placed.

In [12], we present an approach to synthesizing the assumption that a component needs to make about its environment for a given property to be satisfied. The assumption produced is the *weakest*, that is, it restricts the environment no more and no less than is necessary for the component to satisfy the property. The automatic generation of weakest assumptions has direct application to the assume-guarantee proof. More specifically, it removes the burden of specifying assumptions manually thus automating this type of reasoning.

The algorithm presented in [12] does not compute partial results, meaning no assumption is obtained if the computation runs out of memory, which may happen if the state-space of the component is too large. We address this problem in [8], where we present a novel framework for performing assume-guarantee reasoning using the above rule in an incremental and fully automatic fashion. The framework iterates a process based on gradually *learning* assumptions. The learning process is based on queries to component M_1 and on counterexamples obtained by model checking M_1 and its environment, i.e. component M_2, alternately. Each iteration may conclude that the required property is satisfied or violated in the system analyzed. This process is guaranteed to terminate; in fact, it converges to an assumption that is necessary and sufficient for the property to hold in the specific system.

Although sound and complete, Rule 0 is unsatisfactory from an automation point of view [2] since it is not symmetric. We thus considered whether some form of "circular", assume-guarantee like, rule could be developed. For our framework the obvious rule for the parallel composition of two processes, where the assumption of each process is discharged by the commitment (or guarantee) of the other, however, is unsound. Indeed, we demonstrate the unsoundness of the following rule.

Rule 0m.

$$\begin{array}{rl} 1: & M_1 \parallel A_{M_1} \models P \\ 2: & M_2 \parallel A_{M_2} \models P \\ 3: & P \models A_{M_1} \\ 4: & P \models A_{M_2} \\ \hline & M_1 \parallel M_2 \models P \end{array}$$

Take M_1 and M_2 each to be the same process M and the property P as illustrated in Figure 1.

Now take as assumption A_{M_1} the behaviour defined by P, similarly for A_{M_2}. Clearly, premises 3 and 4 hold. And premises 1 and 2 also hold; the parallel composition of M_1 with the assumption A_{M_1} constrains its behaviour to be just that of P, similarly for premise 2. But unfortunately the conclusion doesn't hold since, in our framework, M_1 composed in parallel with M_2 is the behaviour M again; M clearly violates property P since it allows b to occur

[2]It is also unsatisfactory from a formal development point of view!

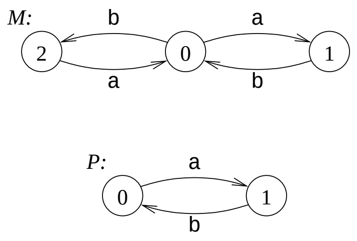

Figure 1: Example of process M and property P to demonstrate unsoundness of *Rule 0m*

first, rather than ensuring a does. The circular reasoning to discharge the assumptions in this case was unsound. The above rule fails for our framework essentially because the two components may have common erroneous behaviour(as far as the property is concerned) which is (mis-)ruled out by assumptions that are overly presumptuous for the particular composition.

3.2 Basic Proof Rule

In the following we give a symmetric parallel composition rule and establish its soundness and completeness for our framework. In Section 4 we then outline how the rule can be used for automated compositional verification along similar lines to the approach given in [8].

Rule 1.

$$\begin{array}{rl} 1: & M_1 \parallel A_{M_1} \models P \\ 2: & M_2 \parallel A_{M_2} \models P \\ 3: & \mathcal{L}\left(coA_{M_1} \parallel coA_{M_2}\right) = \emptyset \\ \hline & M_1 \parallel M_2 \models P \end{array}$$

M_1, M_2, A_{M_1}, A_{M_2} and P are LTSs[3] as defined in the previous section; we require $\alpha P \subseteq \alpha M_1 \cup \alpha M_2$, $\alpha A_{M_1} \subseteq (\alpha M_1 \cap \alpha M_2) \cup \alpha P$ and $\alpha A_{M_2} \subseteq (\alpha M_1 \cap \alpha M_2) \cup \alpha P$. Informally, however, the A_{M_i} are postulated environment assumptions for the components M_i to achieve, respectively, property P. coA_{M_1} denotes the co-assumption for M_1, which is the complement of A_{M_1}. Similarly for coA_{M_2}.

The intuition behind premise 3 stems directly from an understanding of the failure of Rule 0m; premise 3 ensures that the assumptions do not both rule out possible, common, violating behaviour from the components. For example, Rule 0m failed in our example above, because both assumptions ruled out common behaviour $(ba)^*$ of M_1 and M_2, which violates property P. Premise 3 in Rule 1 is a remedy for this problem.

THEOREM 1. *Rule 1 is sound and complete.*

PROOF. To establish soundness, we show that the premises together with the negated conclusion leads to a contradiction. Consider a word t for which the conclusion fails, i.e. t is a trace of $M_1 \parallel M_2$ that violates property P, in other words t is not accepted by P. Clearly, by definition of parallel composition, $t \upharpoonright \alpha M_1$ is accepted by M_1. Hence, by premise 1, the trace $t \upharpoonright \alpha A_{M_1}$ can not be accepted by A_{M_1}, i.e. $t \upharpoonright \alpha A_{M_1}$ is accepted by coA_{M_1}. Similarly, by premise 2, the trace $t \upharpoonright \alpha A_{M_2}$ is accepted by coA_{M_2}. By the definition of parallel composition and the fact that an FSM and its complement have the same alphabet, $t \upharpoonright (\alpha A_{M_1} \cup A_{M_2})$ will be accepted by $coA_{M_1} \parallel coA_{M_2}$. But premise 3 states that there are no common words in the co-sets. Hence we have a contradiction.

Our argument for the completeness of Rule 1 relies upon the use of weakest environment assumptions that are constructed in a similar way to [12]. Let $WA(M, P)$ denote the weakest environment for M that will achieve property P. $WA(M, P)$ is such that, for any environment A, $M \parallel A \models P$ iff $A \models WA(M, P)$.

LEMMA 1. *$coWA(M, P)$ is the set of all traces over the alphabet of $WA(M, P)$ in the context of which M violates property P. In other words, this defines the most general violating environment for (M, P). A violating environment for (M, P) is one that causes M to violate property P in all circumstances.*

To establish completeness, we assume the conclusion of the rule and show that we can construct assumptions that will satisfy the premises of the rule. In fact, we construct the weakest assumptions WA_{M_1}[4], resp. WA_{M_2}, for M_1, resp. M_2, to achieve P, and substitute them for A_{M_1} and A_{M_2}. Clearly premises 1 and 2 are satisfied. It remains to show that premise 3 holds. Again we proceed by proof by contradiction. Suppose there is a word t in $\mathcal{L}\left(coWA_{M_1} \parallel coWA_{M_2}\right)$. By definition of parallel composition, t is accepted by both $coWA_{M_1}$ and $coWA_{M_2}$. By Lemma 1, $t \upharpoonright \alpha P$ violates property P. Furthermore, there will exist $t_1 \in \mathcal{L}\left(M_1 \parallel coP\right)$ such that $t_1 \upharpoonright \alpha t = t$, where αt is the alphabet of the assumptions. Similarly for $t_2 \in \mathcal{L}\left(M_2 \parallel coP\right)$. t_1 and t_2 can then be combined to be a trace t_3 of $M_1 \parallel M_2$ such that $t_3 \upharpoonright \alpha t = t$. But if that is so, this contradicts the assumed conclusion that $M_1 \parallel M_2 \models P$, since t violates P. Therefore, there can not be such a common word t and premise 3 holds. □

4. AUTOMATED REASONING

4.1 Framework

For the use of *Rule 1* to be justified, the assumptions A_{M_1} and A_{M_2} must be more abstract than the components that they represent, i.e. M_2 and M_1 respectively, but also strong enough for the three steps of the rule to be satisfied. Developing such assumptions is a non-trivial process. We propose an iterative approach to automate the application of Rule 1. The approach extends the framework of counterexample-based learning presented in [8]. As in our previous work and as supported by the LTSA model checking tool [19], we assume that both properties and assumptions are described by deterministic FSMs; this is not a serious limitation since any non-deterministic FSM can be transformed to a deterministic one via the subset construction.

[3]except for when A_{M_1}, A_{M_2} and P are false, in which case they are represented as FSMs

[4]Since the context is clear we abbreviate $WA(M, P)$ as WA_M.

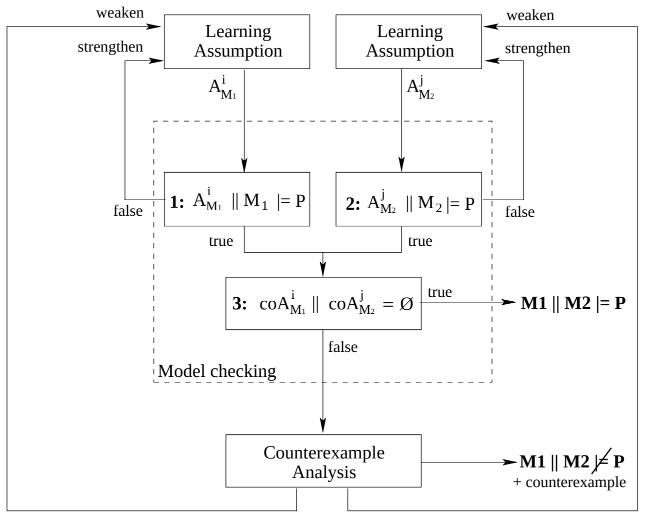

Figure 2: Incremental compositional verification

To obtain appropriate assumptions, our framework applies the compositional rule in an iterative fashion as illustrated in Fig. 2. We use a learning algorithm to generate incrementally an assumption for each component, each of which is strong enough to establish the property P, i.e. to discharge premises 1 and 2 of Rule 1.

We have seen in the previous section that Rule 1 is guaranteed to return conclusive results with the *weakest assumptions* WA_{M_1}, resp. WA_{M_2}, for M_1, resp. M_2, to achieve P. We therefore use L* to iteratively learn the traces of WA_{M_1}, resp. WA_{M_2}. Conjectures are intermediate assumptions $A^i_{M_1}$, resp. $A^j_{M_2}$. As in [8], we use model checking to implement the Teacher needed by L*.

At each iteration, L* is used to build approximate assumptions $A^i_{M_1}$ and $A^j_{M_2}$, based on *querying* the system and on the results of the previous iteration. The first two premises of the compositional rule are then checked. Premise 1 is checked to determine whether M_1 guarantees P in environments that satisfy $A^i_{M_1}$. If the result is false, it means that this assumption is too *weak*, i.e. $A^i_{M_1}$ does not restrict the environment enough for P to be satisfied. The assumption therefore needs to be *strengthened*, which corresponds to removing behaviours from it, with the help of the counterexample produced by checking premise 1. In the context of the next assumption $A^{i+1}_{M_1}$, component M_1 should at least not exhibit the violating behaviour reflected by this counterexample. Premise 2 is checked in a similar fashion, to obtain an assumption $A^j_{M_2}$ such that component M_2 guarantees P in environments that satisfy $A^j_{M_2}$.

If both premise 1 and premise 2 hold, it means that $A^i_{M_1}$ and $A^j_{M_2}$ are strong enough for the property to be satisfied. To complete the proof, premise 3 must be discharged. If premise 3 holds, then the compositional rule guarantees that P holds in $M_1 \parallel M_2$. If it doesn't hold, further analysis is required to identify whether P is indeed violated in $M_1 \parallel M_2$ or whether either $A^i_{M_1}$ or $A^j_{M_2}$ are stronger than necessary. Such analysis is based on the counterexample returned by checking premise 3 and is described in more detail below. If an assumption is too strong it must be *weakened*, i.e. behaviors must be added, in the next iteration. The result of such weakening will be that at least the behavior that the counterexample represents will be allowed by the respective assumption produced at the next iteration. The new assumption may of course be too weak, and therefore the entire process must be repeated.

4.2 Counterexample analysis

If premise 3 fails, then we can obtain a counterexample in the form of a trace t. Similar to [8], we analyse the trace in order to determine how to proceed. We need to determine whether the trace t indeed corresponds to a violation in $M_1||M_2$. This is checked by simulating t on $M_i||coP$, for $i = 1, 2$. The following cases arise. *(1)* If t is a violating trace of both M_1 and M_2, then M_1 and M_2 do indeed have a common bad trace and therefore do not compose to achieve P. *(2)* If t is not a violating trace of M_1 or M_2 then we use t to weaken the corresponding assumption(s).

4.3 Discussion

A characteristic of L* that makes it particularly attractive for our framework is its monotonicity. This means that the intermediate candidate assumptions that are generated increase in size; each assumption is smaller than the next one, i.e. $|A^i_{M_1}| \leq |A^{i+1}_{M_1}| \leq |WA_{M_1}|$ and $|A^j_{M_2}| \leq |A^{j+1}_{M_2}| \leq |WA_{M_2}|$. However, we should note that there is no monotonicity at the semantic level, i.e. it is not necessarily the case that $\mathcal{L}(A^i_{M_1}) \subseteq \mathcal{L}(A^{i+1}_{M_1})$ or $\mathcal{L}(A^j_{M_2}) \subseteq \mathcal{L}(A^{j+1}_{M_2})$ hold.

The iterative process performed by our framework terminates for the following reason. At any iteration, our algorithm returns true or false and terminates, or continues by providing a counterexample to L*. By the correctness of L*, we are guaranteed that if it keeps receiving counterexamples, it will eventually, produce WA_{M_1} and WA_{M_2} respectively.

During this last iteration, premises 1 and 2 will hold by definition of the weakest assumptions. The Teacher will therefore check premise 3, which will return either true and terminate, or a counterexample. Since the weakest assumptions are used, by the completeness proof of *Rule 1*, we know that the counterexample analysis will reveal a true error, and hence the process will terminate.

It is interesting to note that our algorithm may terminate before the weakest assumptions are constructed via the iterative learning and refinement process. It terminates as soon as two assumptions have been constructed that are strong enough to discharge the first two premises but weak enough for the third premise to produce conclusive results, i.e. to prove the property or produce a real counterexample; these assumptions are smaller (in size) than the weakest assumptions.

5. VARIATIONS

In Section 3 we established that Rule 1 is sound and complete for our framework and in Section 4 we showed its applicability for the automated learning approach to compositional verification. However, we need to explore and understand its effectiveness in our automated compositional verification approach. In this section we introduce some straightforward modifications to the rule, maintaining soundness and completeness of course, that may remove unnecessary assumption refinement steps and therefore result in a probable overall improvement in performance.

5.1 First Modification

Our first variation, Rule 1a given below, relaxes the third premise by requiring that any common "bad" trace, as far as the assumptions are concerned, satisfies the property P. The intuition behind this is that the assumptions may well have been overly restrictive and therefore there may be common behaviours of M_1 and M_2, ruled out by the assumptions, that do indeed satisfy the property P.

Rule 1a.

$$\begin{array}{rl} 1: & M_1 \parallel A_{M_1} \models P \\ 2: & M_2 \parallel A_{M_2} \models P \\ 3: & \mathcal{L}\left(coA_{M_1} \parallel coA_{M_2}\right) \subseteq \mathcal{L}\left(P\right) \\ \hline & M_1 \parallel M_2 \models P \end{array}$$

THEOREM 2. *Rule 1a is sound and complete.*

PROOF. Follows easily from the soundness and completeness proofs for Rule 1. □

Rule 1b.

$$\begin{array}{rl} 1: & M_1 \parallel A_{M_1} \models P \\ 2: & M_2 \parallel A_{M_2} \models P \\ 3: & M_1 \parallel coA_{M_1} \models A_{M_2} \textbf{ or } M_2 \parallel coA_{M_2} \models A_{M_1} \\ \hline & M_1 \parallel M_2 \models P \end{array}$$

In essence, in this variation, premise 3 effectively now checks whether any trace in the intersection of the co-assumptions is an illegal behaviour of either component, rather than it just satisfying the property. Notice that the disjunct $M_1 \parallel coA_{M_1} \models A_{M_2}$ is equivalent to $\mathcal{L}\left(coA_{M_1} \parallel coA_{M_2}\right) \subseteq \overline{\mathcal{L}\left(M_1\right)}$, similarly for the other disjunct. We've used this particular form for the disjuncts because of similarity with assumption discharge.

THEOREM 3. *Rule 1b is sound and complete.*

PROOF. Similar to proofs of Theorems 1 and 2. □

Incorporation of Rules 1a and 1b.

Rule 1a can easily be incorporated into our incremental compositional verification framework. Step 3 of Fig. 2 is followed by an extra step, Step 4, for the case when the intersection of the co-assumptions is not empty. Step 4 checks whether the intersection satisfies the given property: if it returns true then we terminate, otherwise continue with counter-example analysis and assumption refinement. In order to incorporate Rule 1b, we simply include a further check to discharge one of the disjuncts of the rule's third premise.

Clearly these "optimisation"s may result in the verification process terminating after fewer learning iterations. On the other hand there will be some increased overhead in performing the extra checks on each weakening iteration. These issues will be analysed more fully in our future implementation of this incremental approach.

5.2 Further Variation

Suppose we are now given components, M_1 and M_2, with associated properties, P_1 and P_2. The following composition rule can be used to establish that property$P_1 \parallel P_2$ holds for $M_1 \parallel M_2$.

Rule 2.

$$\begin{array}{rl} 1: & M_1 \parallel A_{M_1} \models P_1 \\ 2: & M_2 \parallel A_{M_2} \models P_2 \\ 3: & M_1 \parallel A_{M_1} \models A_{M_2} \\ 4: & M_2 \parallel A_{M_2} \models A_{M_1} \\ 5: & \mathcal{L}\left(coA_{M_1} \parallel coA_{M_2}\right) = \emptyset \\ \hline & M_1 \parallel M_2 \models P_1 \parallel P_2 \end{array}$$

where we require $\alpha P_1 \subseteq \alpha M_1$, $\alpha P_2 \subseteq \alpha M_2$, $\alpha A_{M_1} \subseteq \alpha M_1 \cap \alpha M_2$ and $\alpha A_{M_2} \subseteq \alpha M_1 \cap \alpha M_2$.

THEOREM 4. *Rule 2 is sound and complete.*

PROOF. Soundness is established by contradiction, in a similar way to the soundness results for Rules 1, 1a and 1b. We outline the steps. We also abuse and simplify notation by omitting the projections of traces onto the appropriate alphabets.

We assume the properties P_1 and P_2 are not contradictory, i.e. $\mathcal{L}(P_1 \parallel P_2)$ is not empty, or all behaviours are not erroneous. Further, assume the conclusion does not hold, i.e. $M_1 \parallel M_2 \not\models P_1 \parallel P_2$. There then exists a trace t of $M_1 \parallel M_2$ s.t. t is in not accepted by $P_1 \parallel P_2$. There are three subcases to consider.

1. t not in P_1 and t not in P_2
2. t not in P_1 and t in P_2
3. t in P_1 and t not in P_2

The first case contradicts premise 5. By premise 1, t not in P_1 means t is not a trace of $M_1 \parallel A_{M_1}$. But since t is a trace of $M_1 \parallel M_2$ and hence of M_1, then t must be accepted by coA_{M_1}. Similarly, by premise 2, t must be accepted by coA_{M_2}. But this now contradicts premise 5.

For the second case, and similarly for the third case, we will show a contradiction of premise 4, resp. premise 3. As for the first case, by premise 1 if t is not in P_1 and t in M_1 then t must be accepted by coA_{M_1}. As t in P_2, t is accepted by $M_2 \parallel A_{M_2}$. Hence, by premise 4, t is in A_{M_1}. But t can't be both in A_{M_1} and in coA_{M_1}. The mirror argument follows for the third case.

Observe that if premises 3 and 4 were not present, as in the case of rule 1, then soundness is not obtained.

Completeness follows by constructing the weakest assumptions WA_{M_1}, resp. WA_{M_2}, for M_1, resp. M_2, to achieve P_1, resp. P_2, and substituting them for A_{M_1} and A_{M_2}. We can then show that if the rule's conclusion holds, then so do the premises. $\square$

It is interesting to note that if premises 3 and 4 of Rule 2 are modified to be in the more usual form of guarantee discharging assumption, i.e. $P_1 \models A_{M_2}$ and $P_2 \models A_{M_1}$, then the rule is not complete.

As was the case with Rule 1, we can weaken premise 5 of Rule 2 to obtain similar rules to Rule 1a and Rule 1b.

6. HISTORICAL PERSPECTIVE

Over two decades ago, the quest for obtaining sound and complete compositional program proof systems, in various frameworks, remained open. The foundational work on proof systems for concurrent programs, for example [3, 20, 18], whilst not achieving compositional rules, introduced key notions of meta-level co-operation proofs and non-interference proofs. These meta-level proofs were carried out using program code and intermediate assertions from the proofs of the sequential processes. Assumption-commitment, or rely-guarantee, style specifications, in addition to pre- and post-conditions, were then introduced to capture the essence of the meta-level co-operation and non-interference proofs, lifting the assumptions that were implicitly made in the sequential proof outlines to be an explicit part of the specification. Program proof systems, built over such extended specifications, were then developed to support the stepwise, or hierarchical, development of concurrent, or distributed, programs, see for example [16, 25, 4, 23]. The development of such compositional proof systems continues to this day and the interested reader should consult [10] for an extensive and detailed coverage.

In recent years, there has been a resurgence of interest in formal techniques, and in particular assume-guarantee reasoning, for supporting component-based design: see for example [9]. Even though various sound and often complete proof systems have been developed for this style of reasoning, more often than not it is a mental challenge to obtain the most appropriate assumptions [15]. It is even more of a challenge to find automated techniques to support this style of reasoning. The thread modular reasoning underlying the Calvin tool [11] is one start in this direction. One way of addressing both the design and verification of large systems is to use their natural decomposition into components. Formal techniques for support of component-based design are gaining prominence, see for example [9]. In order to reason formally about components in isolation, some form of assumption (either implicit or explicit) about the interaction with, or interference from, the environment has to be made. Even though we have sound and complete reasoning systems for assume-guarantee reasoning, see for example [16, 21, 6, 14], it is always a mental challenge to obtain the most appropriate assumption [15].

It is even more of a challenge to find automated techniques to support this style of reasoning. The thread modular reasoning underlying the Calvin tool [11] is one start in this direction. The Mocha toolkit [1] provides support for modular verification of components.

The problem of generating an assumption for a component is similar to the problem of generating component interfaces to deal with intermediate state explosion in CRA. Several approaches have been defined for automatically abstracting a component's environment to obtain interfaces [5, 17]. These approaches do not address the incremental refinement of interfaces.

Learning in the context of model checking has also been investigated in [13], but with a different goal. In that work, the L* Algorithm is used to generate a model of a software system which can then be fed to a model checker. A conformance checker determines if the model accurately describes the system.

7. CONCLUSIONS AND FUTURE WORK

Although theoretical frameworks for sound and complete assumption-commitment reasoning have existed for many years, their practical impact has been limited because they involve non-trivial human interaction. In this paper, we have presented a new set of sound and complete proof rules for parallel composition that support a fully automated verification approach based upon such a reasoning style. The automation approach extends and improves upon our previous work that introduced a learning algorithm to generate and refine assumptions based on queries and counterexamples, in an iterative process. The process is guaranteed to terminate, and return true if a property holds in a system, and a counterexample otherwise. If memory is insufficient to reach termination, intermediate assumptions are generated, which may be useful in approximating the requirements that a component places on its environment to satisfy certain properties.

One advantage of our approach is its generality. It relies on standard features of model checkers, and could therefore easily be introduced in any such tool. For example, we are currently in the process of implementing it in the LTSA. The architecture of our framework is modular, so its components can easily be substituted by more efficient ones.

We have implemented our framework within the LTSA tool and over the coming months we will conduct a number of experiments to establish the practical effectiveness of our new composition rule and its variations. We need to understand better the various trade-offs between the increased overhead of additional premise testing and the computational savings from earlier termination of the overall process. In addition, we need to investigate known variants of our rules for N-process compositions, again considering various practical tradeoffs in implementation terms. Of course, an interesting challenge will also be to extend the types of properties that our framework can handle to include liveness, fairness, and timed properties.

REFERENCES

[1] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proc. of the Tenth Int. Conf. on*

Comp.-Aided Verification (CAV), pages 521–525, June 28–July 2, 1998.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, Nov. 1987.

[3] K. R. Apt, N. Francez, and W.-P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.

[4] H. Barringer and R. Kuiper. Hierarchical development of concurrent systems in a framework. In S. B. et al, editor, *Seminar in Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 35–61, 1985.

[5] S. C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Trans. on Soft. Eng. and Methodology*, 5(4):334–377, Oct. 1996.

[6] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proc. of the Fourth Symp. on Logic in Comp. Sci.*, pages 353–362, June 1989.

[7] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[8] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *9th International Conference for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *LNCS*, Warsaw, Poland, 2003. Springer.

[9] L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *Proc. of the First Int. Workshop on Embedded Soft.*, pages 148–165, Oct. 2001.

[10] W.-P. de Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Non-compositional Methods.* Cambridge University Press, 2001.

[11] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proc. of the Eleventh European Symp. on Prog.*, pages 262–277, Apr. 2002.

[12] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *Proc. of the Seventeenth IEEE Int. Conf. on Auto. Soft. Eng.*, Sept. 2002.

[13] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Proc. of the Eighth Int. Conf. on Tools and Alg. for the Construction and Analysis of Sys.*, pages 357–370, Apr. 2002.

[14] O. Grumberg and D. E. Long. Model checking and modular verification. In *Proc. of the Second Int. Conf. on Concurrency Theory*, pages 250–265, Aug. 1991.

[15] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc. of the Tenth Int. Conf. on Comp.-Aided Verification (CAV)*, pages 440–451, June 28–July 2, 1998.

[16] C. B. Jones. Specification and design of (parallel) programs. In R. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332. IFIP: North Holland, 1983.

[17] J.-P. Krimm and L. Mounier. Compositional state space generation from Lotos programs. In *Proc. of the Third Int. Workshop on Tools and Alg. for the Construction and Analysis of Sys.*, pages 239–258, Apr. 1997.

[18] G. Levin and D. Gries. A proof technique for communicating sequential processe s. *Acta Informatica*, 15(3):281–302, 1981.

[19] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs.* John Wiley & Sons, 1999.

[20] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[21] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logic and Models of Concurrent Systems*, volume 13, pages 123–144, New York, 1984. Springer-Verlag.

[22] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, Apr. 1993.

[23] E. W. Stark. A proof technique for rely/guarantee properties. In *Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Theoretical Computer Science*, pages 369–391. Springer-Verlag, Dec. 1985.

[24] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[25] J. Zwiers, W.-P. de Roever, and P. van Emde Boas. Compositionality and concurrent networks: Soundness and completeness of a proof system. In *Proceedings of ICALP '85, Springer LNCS 194*, pages 509–519. Springer-Verlag, 1985.