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.

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 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-

1The 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 Act be the universal set of observable actions and let \( \tau \) denote a local action unobservable to a component's environment. An LTS \( M \) is a quadruple \((Q, aM, \delta, q0)\) where:

- \( Q \) is a non-empty finite set of states
- \( aM \subseteq Act \) is a finite set of observable actions called the alphabet of \( M \)
- \( \delta \subseteq Q \times aM \cup \{\tau\} \times Q \) is a transition relation
- \( q0 \in Q \) is the initial state

An LTS \( M = (Q, aM, \delta, q0) \) is non-deterministic if it contains \( \tau \)-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 \( \Sigma \subseteq Act \), we use \( t|\Sigma \) 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 \( L(M) \). We will freely use the expression "a word t is accepted by \( M \)" to mean that \( t \in L(M) \). Note that the empty word is accepted by any LTS.

Parallel Composition. Let \( M = (Q, aM, \delta, q0) \) and \( M' = (Q', aM', \delta', q0') \). 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 \( aM = aM' \) 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 \( M_1 = (Q_1, aM_1, \delta_1, q0_1) \) and \( M_2 = (Q_2, aM_2, \delta_2, q0_2) \) be two LTSs. Then \( M_1 || M_2 \) is an LTS \( M = (Q, aM, \delta, q0) \), where \( Q = Q_1 \times Q_2 \), \( q0 = (q0_1, q0_2) \), \( aM = aM_1 \cup aM_2 \), and \( \delta \) is defined as follows, where \( a \) is either an observable action or \( \tau \) (note that the symmetric rules are implied by the fact that the operator is commutative):

\[
\frac{M_1 \xrightarrow{a} M_1', a \notin aM_2}{M_1 || M_2 \xrightarrow{a} M_1' || M_2} \\
\frac{M_1 \xrightarrow{a} M_1', M_2 \xrightarrow{a} M_2', a \notin \tau}{M_1 || M_2 \xrightarrow{a} M_1' || M_2'}
\]

Note. \( L(M_1 || M_2) = \{t \mid t | aM_1 \in L(M_1) \wedge t | aM_2 \in L(M_2) \wedge t \in (aM_1 \cup aM_2)^* \} \)

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

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 \((Q, aM, \delta, q0, F)\) where \( Q, aM, \delta, \) 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 \( \delta(q, t) \) 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 \( M = (Q, aM, \delta, q0, F) \) if \( \delta(q0, t) \cap F \neq \emptyset \). Note that in the following sections, the term trace is often used to denote a word. The language accepted by \( M \), denoted \( L(M) \), is the set \( \{t \mid \delta(q0, t) \cap F \neq \emptyset \} \).

For an FSM \( M = (Q, aM, \delta, q0, F) \), we use LTS(M) to denote the LTS \((Q, aM, \delta, q0)\) 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 \( M_1 = (Q_1, aM_1, \delta_1, q0_1, F_1) \) and \( M_2 = (Q_2, aM_2, \delta_2, q0_2, F_2) \) be two FSMs. Then \( M_1 || M_2 \) is an FSM \( M = (Q, aM, \delta, q0, F) \), where:

- \( (Q, aM, \delta, q0) = LTS(M_1) || LTS(M_2) \), 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. \( L(M_1 || M_2) = \{t \mid t | aM_1 \in L(M_1) \wedge t | aM_2 \in L(M_2) \wedge t \in (aM_1 \cup aM_2)^* \} \)

Satisfiability. For FSMs \( M \) and \( P \) where \( aP \subseteq aM \), \( M \models P \) if and only if \( \forall t \in L(M). t \alpha P \in L(P) \).

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
Rule 0m.

\[ 1: \quad M_1 \parallel A_{M_1} \models P \]
\[ 2: \quad M_2 \models A_{M_2} \]
\[ 3: \quad P \models A_{M_1} \]
\[ 4: \quad P \models A_{M_2} \]

\[ M_1 \parallel M_2 \models P \]

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.

\[\text{It is also unsatisfactory from a formal development point of view!}\]
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.

\[
\begin{align*}
1: & \quad M_1 \parallel A_{M_1} \models \neg P \\
2: & \quad M_2 \parallel A_{M_2} \models \neg P \\
3: & \quad L(\alpha A_{M_1} \cup \alpha A_{M_2}) = \emptyset
\end{align*}
\]

\(M_1, M_2, A_{M_1}, A_{M_2}, \text{and } P\) are LTSs\(^3\) as defined in the previ-
ous section; we require \(\alpha P \subseteq \alpha A_{M_1} \cup \alpha A_{M_2}, \alpha A_{M_1} \subseteq (\alpha A_{M_1} \cap
\alpha A_{M_2}) \cup \alpha P\) and \(\alpha A_{M_2} \subseteq (\alpha A_{M_1} \cap \alpha A_{M_2}) \cup \alpha P\). Informally,
however, the \(A_{M_1}\) are postulated environment assumptions
for the components \(M_1\) to achieve, respectively, property \(P\).
\(\alpha A_{M_1}\) denotes the co-assumption for \(M_1\), which is the com-
plement of \(A_{M_1}\). Similarly for \(\alpha A_{M_2}\).

The intuition behind premise 3 stems directly from an un-
derstanding of the failure of Rule 0m; premise 3 ensures that
the assumptions do not both rule out possible, common, vi-
olating 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 contradic-
tion. 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
\(t\) is not accepted by \(P\). Clearly, by definition of par-
allel composition, \(t|aM_1\) is accepted by \(M_1\). Hence, by
premise 1, the trace \(t|aA_{M_1}\) can not be accepted by \(A_{M_1}\),
i.e. \(t|aA_{M_1}\) is accepted by \(coA_{M_1}\). Similarly, by premise 2,
the trace \(t|aA_{M_2}\) 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, \(t|\alpha A_{M_1} \cup \alpha A_{M_2}\) will
be accepted by \(coA_{M_1} \cup coA_{M_2}\). 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 con-
structed 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 \(WA_{M_1}\), 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 con-
diction. Suppose there is a word \(t\) in \(L(coWA_{M_1} \cup coWA_{M_2})\).
By definition of parallel composition, \(t\) is accepted by both
\(coWA_{M_1}\) and \(coWA_{M_2}\). By Lemma 1, \(t|\alpha P\) violates prop-
erty \(P\). Furthermore, there will exist \(t_1 \in L(M_1 \parallel coP)\) such
that \(t_1|aT = t\), where \(aT\) is the alphabet of the assumptions.
Similarly for \(t_2 \in L(M_2 \parallel coP)\). \(t_1\) and \(t_2\) can then be com-
bined to be a trace \(t_3\) of \(M_1 \parallel M_2\) such that \(t_3|\alpha T = t\).
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. \(M_1\) and \(M_2\) respectively, but also strong
enough for the three steps of the rule to be satisfied. Devel-
oping 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 deter-
ministic one via the subset construction.

\(^3\)Since the context is clear we abbreviate \(WA(M, P)\) as \(WA_M\).
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 \( M_i \| M_j \). This is checked by simulating \( t \) on \( M_i \| M_j \) 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 stronger than the next one, i.e. \( |A_{M_i}^1| \leq |A_{M_i}^2| \leq |WAM_{M_i}| \) and \( |A_{M_i}^3| \leq |WAM_{M_i}| \). However, we should note that there is no monotonicity at the semantic level, i.e. it is not necessarily the case that \( L(A_{M_1}^1) \subseteq L(A_{M_1}^2) \) or \( L(A_{M_2}^1) \subseteq L(A_{M_2}^2) \).

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 \( WAM_{M_1} \) and \( WAM_{M_2} \) respectively.

During this last iteration, premises 1 and 2 will hold by definiton 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 verifi-
cation approach. In this section we introduce some straight-
forward modifications to the rule, maintaining soundness
and completeness of course, that may remove unnecessary
assumption refinement steps and therefore result in a prob-
able overall improvement in performance.

### 5.1 First Modification

Our first variation, Rule la 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 com-
mon behaviours of $M_1$ and $M_2$, ruled out by the assump-
tions, that do indeed satisfy the property $P$.

**Rule la.**

\[
\begin{align*}
1: & \quad M_1 \parallel A_{M_1} \models P \\
2: & \quad M_2 \parallel A_{M_2} \models P \\
3: & \quad \mathcal{L}(\text{co}_A M_1 \parallel \text{co}_A M_2) \subseteq \mathcal{L}(P) \\
\hline
M_1 \parallel M_2 \models P
\end{align*}
\]

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

**PROOF.** Follows easily from the soundness and complete-
ness proofs for Rule 1. □

**Rule 1b.**

\[
\begin{align*}
1: & \quad M_1 \parallel A_{M_1} \models P \\
2: & \quad M_2 \parallel A_{M_2} \models P \\
3: & \quad M_1 \parallel \text{co}_A M_1 \models A_{M_2} \text{ or } M_2 \parallel \text{co}_A M_2 \models A_{M_1} \\
\hline
M_1 \parallel M_2 \models P
\end{align*}
\]

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
just satisfying the property. Notice that the disjunct
$M_1 \parallel \text{co}_A M_1 \models A_{M_2}$ is equivalent to
$\mathcal{L}(\text{co}_A M_1 \parallel \text{co}_A M_2) \subseteq \mathcal{L}(M_1)$, 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 com-
positional 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 per-
forming the extra checks on each weakening iteration. These
issues will be analysed more fully in our future implementa-
tion 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.**

\[
\begin{align*}
1: & \quad M_1 \parallel A_{M_1} \models P_1 \\
2: & \quad M_2 \parallel A_{M_2} \models P_2 \\
3: & \quad M_1 \parallel A_{M_1} \models A_{M_2} \\
4: & \quad M_2 \parallel A_{M_2} \models A_{M_1} \\
5: & \quad \mathcal{L}(\text{co}_A M_1 \parallel \text{co}_A M_2) = \emptyset \\
\hline
M_1 \parallel M_2 \models P_1 \parallel P_2
\end{align*}
\]

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

**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 er-
roneous. Further, assume the conclusion does not hold, i.e.
$M_1 \parallel M_2 \not\models P_1 \parallel P_2$. Then there exists a trace $t$ of $M_1 \parallel M_2$
s.t. $t$ is in $P_1$ and $t$ not in $P_2$. There are three sub-
cases 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 $\text{co}_A M_1$. Similarly, by premise 2, $t$ must be accepted by
$\text{co}_A M_2$. 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 $\text{co}_A M_1$. As $t$ in $P_2$, $t$ is accepted by
$M_2 \parallel A_{M_2}$. Hence, by premise 4, $t$ is in $A_{M_2}$. But $t$ can’t be
both in $A_{M_2}$ and in $\text{co}_A M_1$. 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 $W_{A_1}$, resp. $W_{M_1}$ 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.

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


