Abstraction and Learning for Infinite-State Compositional Verification

Dimitra Giannakopoulou
NASA Ames
Moffett Field, CA, USA
dimitra.giannakopoulou@nasa.gov

Corina S. Păsăreanu
CMU/NASA Ames
Moffett Field, CA, USA
corina.s.pasareanu@nasa.gov

Despite many advances that enable the application of model checking techniques to the verification of large systems, the state-explosion problem remains the main challenge for scalability. Compositional verification addresses this challenge by decomposing the verification of a large system into the verification of its components. Recent techniques use learning-based approaches to automate compositional verification based on the assume-guarantee style reasoning. However, these techniques are only applicable to finite-state systems. In this work, we propose a new framework that interleaves abstraction and learning to perform automated compositional verification of infinite-state systems. We also discuss the role of learning and abstraction in the related context of interface generation for infinite-state components.

1 Introduction

Despite several breakthroughs that enable the application of model checking to the verification of realistic systems, the essential challenge in model checking remains the well-known state-space explosion problem: the size of a concurrent reactive system to be verified is the product of the sizes of its constituent components. Thus the cost of exhaustive verification as done in model checking grows exponentially in the number of state variables.

Compositional techniques attempt to tame this problem by applying verification to individual components and merging the results without analyzing the whole system. In checking components individually, it is often necessary to incorporate some knowledge of the context in which each component is expected to operate correctly. Assume-guarantee reasoning addresses this issue by using assumptions that capture the expectations that a component makes about its environment. The simplest such 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 \( P \) (Premise 1) and discharging \( A \) on the environment \( M_2 \) (Premise 2). For safety properties, Premise 2 amounts to checking that \( A \) is a conservative abstraction of \( M_2 \), i.e., an abstraction that preserves all of \( M_2 \)’s execution paths. This rule is also represented as follows, where the notation is described in more detail in Section 2.

\[
\begin{align*}
1 : \ & \langle A \rangle M_1 \langle P \rangle \\
2 : \ & \langle \text{true} \rangle M_2 \langle A \rangle \\
\rightarrow \ & \langle \text{true} \rangle M_1 \parallel M_2 \langle P \rangle
\end{align*}
\]

Assumptions have traditionally been developed manually, a limiting factor to the practical impact of assume-guarantee reasoning. In this article, we review learning-based frameworks that automate the application of assume-guarantee reasoning. These frameworks use the L* automata learning algorithm to generate and refine assumptions, based on results obtained from model checking individual components separately. Other related approaches followed (see e.g. articles in journal issue 14).
These prior approaches were performed in the context of finite-state components. In contrast, the focus of the present work is on compositional verification in the context of infinite-state (or very large) components, where the verification of individual components itself is intractable. We introduce component abstractions for this task, and explain how component abstraction refinement interacts with assumption generation. This article contributes a framework for automated assume-guarantee reasoning of infinite-state systems. A related problem that we studied in the past in the context of infinite-state systems is component interface generation [30]. We explain here the differences between the two approaches, in particular the different types of abstractions that are needed in each.

2 Formalisms

2.1 Component Models and Properties

2.1.1 Communicating State Machines

We model software components as (possibly infinite) communicating state machines. Note that typically components have implicit finite representations (e.g. a program) and only their semantics are given as infinite state machines.

Let $\text{Act}$ be the universal set of observable actions and let $\tau$ denote a local action unobservable to a component’s environment.

A CSM $M$ is a four-tuple $\langle Q, \alpha_M, \delta, q_0 \rangle$ where:

- $Q$ is a non-empty set of states.
- $\alpha_M \subseteq \text{Act}$ is a set of observable actions called the alphabet of $M$.
- $\delta \subseteq Q \times (\alpha_M \cup \{\tau\}) \times Q$ is a transition relation
- $q_0 \in Q$ is the initial state

We write $q \xrightarrow{a} q'$ for $(q,a,q') \in \delta$. A trace $t$ of a CSM $M$ is a finite sequence of observable actions that label the transitions that $M$ can perform starting at its initial state (ignoring the $\tau$-transitions). The language of $M$, denoted $L(M)$ is the set of all traces of $M$.

Note that we assume that the state set $Q$ may be infinite but the alphabet $\alpha_M$ is finite; the actions in the alphabet can be seen as modeling e.g. sending and receiving of messages or method calls and returns for software components. Method parameters are not treated here (see [3,16] for approaches that handle parameters).

We sometimes abuse the notation and denote by $t$ both a trace and its trace CSM. For a trace $t$ of length $n$, its trace CSM consists of $n+1$ states, where there is a transition between states $m$ and $m+1$ on the $m^{th}$ action in the trace $t$. For $\Sigma \subseteq \text{Act}$, we use $t \uparrow \Sigma$ to denote the trace obtained by removing from $t$ all occurrences of actions $a \notin \Sigma$. Furthermore, $M \uparrow \Sigma$ is defined to be a CSM over alphabet $\Sigma$ which is obtained from $M$ by renaming to $\tau$ all the transitions labeled with actions that are not in $\Sigma$. Let $t, t'$ be two traces and $\Sigma, \Sigma'$ be the sets of actions occurring in $t, t'$, respectively. By the symmetric difference of $t$ and $t'$ we mean the symmetric difference of the sets $\Sigma$ and $\Sigma'$.

A CSM $M = \langle Q, \alpha_M, \delta, q_0 \rangle$ is non-deterministic if any of these two conditions holds: 1) it contains $\tau$-transitions or 2) if there exists $(q,a,q'), (q,a,q'') \in \delta$ such that $q' \neq q''$. Otherwise, $M$ is deterministic.
2.1.2 Parallel Composition of CSMs

Let \( M_1 = (Q^1, \alpha M_1, \delta^1, q_0^1) \) and \( M_2 = (Q^2, \alpha M_2, \delta^2, q_0^2) \) be two CSMs. The parallel composition operator \( \parallel \) 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. Formally, \( M_1 \parallel M_2 \) is a CSM \( M = (Q, \alpha M, \delta, q_0) \), where \( Q = Q^1 \times Q^2 \), \( q_0 = (q_0^1, q_0^2) \), \( \alpha M = \alpha M_1 \cup \alpha M_2 \), and \( \delta \) is defined as follows, where \( q_1, q_1', q_2, q_2' \in Q^1 \) and \( q_2, q_2' \in Q^2 \):

\[
\begin{align*}
q_1 \xrightarrow{a} q_1', a \notin \alpha M_2 & \quad (q_1, q_2) \xrightarrow{a} (q_1', q_2) \\
q_2 \xrightarrow{a} q_2', a \notin \alpha M_1 & \quad (q_1, q_2) \xrightarrow{a} (q_1, q_2') \\
q_1 \xrightarrow{a} q_1', q_2 \xrightarrow{a} q_2', a \neq \tau & \quad (q_1, q_2) \xrightarrow{a} (q_1', q_2')
\end{align*}
\]

The language of \( M_1 \parallel M_2 \) is \( \mathcal{L}(M_1 \parallel M_2) = \{ t \mid t \uparrow \alpha M_1 \in \mathcal{L}(M_1) \land t \uparrow \alpha M_2 \in \mathcal{L}(M_2) \land t \in (\alpha M_1 \cup \alpha M_2)^* \} \)

2.1.3 Properties

In this paper, we address assume-guarantee reasoning in the context of checking safety properties. For the context of our presentation, a safety property is modeled as a deterministic CSM \( P \), whose language \( \mathcal{L}(P) \) defines the set of acceptable behaviors over \( \alpha P \). For CSMs \( M \) and \( P \) where \( \alpha P \subseteq \alpha M \), \( M \models P \) if and only if

\[
\forall t \in \mathcal{L}(M) : t \uparrow \alpha P \in \mathcal{L}(P).
\]

Checking properties reduces to reachability checks: \( M \models P \) holds iff a special error state is unreachable in \( M \parallel P_{err} \), where \( P_{err} \) is the complement of \( P \) and is obtained by completing \( P \) such that each missing transition on \( \alpha P \) becomes a transition to the error state.

2.2 Assume-guarantee Reasoning

In the assume-guarantee paradigm a formula is a triple \( \langle A \rangle M \langle P \rangle \), where \( M \) is a component, \( P \) is a property, and \( A \) is an assumption about \( M \)'s environment. The formula is true if whenever \( M \) is part of a system satisfying \( A \), then the system must also guarantee \( P \), i.e., \( \forall E, E \parallel M \models A \) implies \( E \parallel M \models P \). Note that when \( \alpha P \subseteq \alpha A \cup \alpha M \), this is equivalent to \( A \parallel M \models P \).

Let \( M \) be a finite-state component with \( \Sigma \) being the set of its interaction points with the environment, i.e. the set of actions which will participate in the composition of \( M \) with another component from the environment. Furthermore, let \( P \) be a safety property. Then there is a natural notion of the weakest assumption \( A_w \) for \( M \) with respect to \( P \), with \( \alpha A_w = \Sigma \). \( A_w \) is characterized by two properties:

- **Safety**: \( \langle A_w \rangle \langle P \rangle \) holds.
- **Permisiveness**: \( A_w \) characterizes all the possible environments \( E \) under which \( P \) holds, i.e. \( \forall E : M \parallel E \models P \Rightarrow E \models A_w \).

These two conditions essentially ensure that \( \forall E : M \parallel E \models P \) iff \( E \models A_w \). It has been shown that, for any finite-state component \( M \), the weakest assumption \( A_w \) exists, and can be constructed algorithmically. The weakest assumption is associated with a similar notion of precision defined in the literature for “temporal” component interfaces [20], i.e., interfaces that capture ordering relationships between invocations.
of component methods. For example, an interface may describe the fact that closing a file before opening it is undesirable because an exception will be thrown. An ideal interface should precisely represent the component in all its intended usages. It should be safe, meaning that it should exclude all problematic interactions, and permissive, in that it should include all the good interactions [20].

Similarly, assumption safety is concerned with restricting behaviors to only those that satisfy $P$. Permissiveness is concerned with including behaviors, making sure that behaviors are restricted only if necessary. Permissiveness is desirable, because $A_w$ is then appropriate for deciding whether an environment $E$ is suitable for $M_1$ (if $E$ does not satisfy $A_w$, then $E \parallel M_1$ does not satisfy $P$).

The simplest assume-guarantee rule is for checking a safety property $P$ on a system with two components $M_1$ and $M_2$.

**Rule ASYM**

$$
\begin{align*}
1 : & \langle A \rangle M_1 \langle P \rangle \\
2 : & \langle true \rangle M_2 \langle A \rangle \\
\hline
\langle true \rangle M_1 \parallel M_2 \langle P \rangle
\end{align*}
$$

In this rule, $A$ denotes an assumption about the environment of $M_1$. Note that the rule is not symmetric in its use of the two components, and does not support circular reasoning. Despite its simplicity, experience has shown it to be quite useful in the context of checking safety properties.

### 2.2.1 Soundness and Completeness

Soundness of an assume-guarantee rule means that whenever its premises hold, its conclusion holds as well. Without soundness, we cannot rely on the correctness of conclusions reached by applications of the rule, making the rule useless for verification. Completeness states that whenever the conclusion of the rule is correct, the rule is applicable, i.e., there exist suitable assumptions such that the premises of the rule hold. Completeness is not needed to ensure correctness, but it is an important measure for the usability of the rule. Rule ASYM is both sound and complete. To show soundness, note that $\langle true \rangle M_2 \langle A \rangle$ implies $\langle true \rangle M_1 \parallel M_2 \langle A \rangle$. Then, since $\langle A \rangle M_1 \langle P \rangle$ also holds, it follows that $\langle true \rangle M_1 \parallel M_2 \langle P \rangle$ holds as well (from the definition of assume-guarantee triples). Completeness holds trivially, by substituting $M_2$ for $A$.

For the use of rule ASYM to be justified, the assumption should be (much) smaller than $M_2$, but still reflect $M_2$’s behavior, i.e. $A$ should be an abstraction of $M_2$, according to premise 2. Additionally, an appropriate assumption for the rule needs to “restrict” $M_1$ enough to satisfy $P$ in premise 1. Coming up with such assumptions manually is highly non-trivial. In the next sections we describe techniques for synthesizing assumptions automatically.

### 2.3 Abstraction

To check properties of infinite-state components, we build may and must abstractions of software components. A popular technique uses predicate abstraction – a special instance of abstract interpretation [10] that maps a potentially infinite state transition system into a finite state transition system via a finite set of predicates $\text{Preds} = \{ p_1, \ldots, p_n \}$ over a program’s variables.

An abstraction function $\alpha$ maps a concrete state $q$ to a set of predicates that hold in $q$: $\alpha(q) = \{ p \in \text{Preds} \mid q \models p \}$. For a concrete transition we define corresponding may and must transitions. Let $q_A, q'_A$ denote abstract states, and $q, q'$ denote concrete states (in the un-abstracted system):
D. Giannakopoulou and C. S. Păsăreanu

From the definition of the weakest assumption \( A \), we can observe that for \( A_w \), the premises of Rule ASYM become necessary, in addition to being sufficient, for the conclusion of the rule to hold. In other words, \( \langle A_w \rangle M_1 \langle P \rangle \) and \( \langle \text{true} \rangle M_2 \langle A \rangle_w \) hold, if and only if \( \langle \text{true} \rangle M_1 \parallel M_2 \langle P \rangle \). This is an advantage for an automated assume-guarantee reasoning framework, since it enables us to also disprove properties of a system, compositionally. The framework illustrated in Figure 1 and first presented in [8], provides a learning-based approach to assume-guarantee reasoning of finite-state components. In this framework, \( L^* \) targets the computation of the weakest assumption \( A_w \), and its application to rule ASYM. The set of communicating actions of
component $M_1$ with its environment is defined as: $(\alpha M_1 \cup \alpha P) \cap \alpha M_2$. We use this set as the alphabet of the weakest assumption in this context, hence the alphabet over which $L^*$ is learning. Note that the framework uses the knowledge of the actual environment of component $M_1$, namely, component $M_2$, to make the reasoning more efficient. More specifically, the framework implements a teacher for $L^*$, meaning that it responds to queries and conjectures, as described in the following.

**Queries.** $L^*$ is first used to repeatedly query $M_1$ to check whether, in the context of strings $s$, $M_1$ violates the property. More formally, the query corresponds to checking the triple $\langle s \rangle M_1 \langle P \rangle$ as illustrated in Figure 1. Checking $\langle s \rangle M_1 \langle P \rangle$ corresponds to simulating string $s$ on $M_1 \parallel P$: if an error is reachable, then the triple is $false$, otherwise it is $true$. The query returns $true/false$ if $\langle s \rangle M_1 \langle P \rangle$ is $true/false$, respectively. This is because, as mentioned, $A_w$ allows all behaviors that satisfy the property, and disallows only violating behaviors.

**Conjectures.** The automaton $A_i$ conjectured during iteration $i$, is checked for correctness, which in this context means checking whether it corresponds to the weakest assumption or not. As discussed earlier, the weakest assumption is safe and permissive. We therefore reduce equivalence queries to two separate checks, for safety, and permissiveness of $A_i$.

The first check is for safety: $\langle A_i \rangle M_1 \langle P \rangle$; it can be performed by a model checker. If $A_i$ is safe, then the teacher proceeds to checking permissiveness. If it is unsafe, the model checker returns a counterexample. The resulting counterexample $c$, projected on the assumption alphabet $\alpha A_i = \alpha A_w$, is returned to $L^*$ to refine its conjecture. The projection is necessary because $L^*$ needs counterexamples in terms of the alphabet over which it is learning.

As discussed, permissiveness is concerned with ensuring that the assumption does not exclude correct behaviors. However, given the fact that the main goal of the framework is to prove or disprove a property on the system using assume-guarantee reasoning, the framework does not need to generate a fully permissive assumption. Rather, it uses $M_2$ to add behaviors to over-restrictive assumptions on demand, and as needed for completion of the verification.

Note that the check for safety coincides with premise 1 of Rule ASYM. It therefore remains to check premise 2 ($\langle true \rangle M_2 \langle A_i \rangle$). We use premise 2 to drive the permissiveness check and potentially complete assume-guarantee reasoning in Figure 1 as follows. If $\langle true \rangle M_2 \langle A_i \rangle$ is true, then we know that both premises of Rule ASYM hold, and therefore that $P$ holds for $M_1 \parallel M_2$. If $\langle true \rangle M_2 \langle A_i \rangle$ is false, the
Teacher performs some analysis to determine the underlying reason (see Figure 1). The Teacher performs a query (of the L* type) in order to determine whether the returned counterexample \( c \), projected to the alphabet of the assumption, belongs to \( A_w \), in which case L* needs to refine the assumption. If the query returns true, then \( A_i \) is not permissive, so \( c \uparrow \alpha A_i \) is returned to L* for refinement of its guess. If, on the other hand, the answer is false, it means that \( c \) is a word that belongs to \( M_2 \), in the context of which \( M_1 \) violates the property \( P \). As a consequence, \( M_1 \parallel M_2 \) does not satisfy the property \( P \).

Each new assumption marks the beginning of the next iteration cycle. Notice that the answers that the framework provides to L* are always precise with respect to the targeted weakest assumption. However, the framework uses \( M_2 \) to select which missing words to include in the language of the assumption. The reason is that we restrict our reasoning to a specific context, rather than accounting for all possible contexts, as required for the computation of \( A_w \). That means, of course, that the assumption obtained from this framework does not necessarily correspond to \( A_w \). On the other hand, we remind the reader that the primary goal is to obtain conclusive results from the assume guarantee rule. As soon as we are able to prove or disprove the property in the system, we stop refining the learned assumption, since we have achieved our goal. The assumption computed with this framework will be smaller than, or in the worst case equal to \( A_w \) in terms of number of states, as guaranteed by the characteristics of L*. In the worst case, where \( A_w \) itself is computed, the framework is guaranteed to terminate, because \( A_w \) is both necessary and sufficient, and therefore the framework will prove or disprove the property during this iteration.

### 3.2 Correctness Arguments.

**Framework correctness argument:** The framework directly uses the assume-guarantee rule Rule ASYM to answer conjectures. Soundness of the rule guarantees correctness of the positive answers by the framework. On the other hand, each counterexample reported is a real counterexample, as discussed above.

**Teacher correctness argument:** Correctness of the teacher corresponds to showing that all the answers returned to L* are consistent with \( A_w \). This was discussed during the presentation of the framework above.

**Termination argument:** Since the Teacher implemented in our framework only comes back to L* for refinement with counterexamples related to \( A_w \), the framework eventually converges to \( A_w \), unless it terminates earlier. As discussed, \( A_w \) makes Rule ASYM sound and complete, and therefore our framework will return a conclusive answer at that iteration.

### 3.3 Extensions

The framework has been extended to reasoning about more than two components (by applying the framework recursively for Premise 2) and to other circular and symmetric rules [28]. It has been demonstrated on checking flight software models, where it achieved significant savings in terms of time and memory, and in some cases it was able to terminate while the monolithic (non-compositional) verification ran out of time and memory resources.

### 4 Compositional Verification for Infinite-State Systems

When reasoning about infinite-state components, abstractions need to be used to further reduce the state spaces of individual components. Figure 2 illustrates a learning-based framework for automating com-
positional verification for infinite-state systems.

In this section, we will first present how assume-guarantee reasoning of infinite-state components can be performed given some assumption $A$. We will subsequently discuss how such steps are introduced in creating automated assume-guarantee frameworks using learning for assumption inference.

\begin{align*}
1 : & \langle A \rangle M_1 \langle P \rangle \\
2 : & \langle \text{true} \rangle M_2 \langle A \rangle \\
      & \text{such that } M_1 \parallel M_2 \langle P \rangle \text{ holds}
\end{align*}

Going back to rule ASYM, let us assume that both $M_1$ and $M_2$ are infinite-state, or too large to perform the steps involved in the two premises by model checking. A standard approach used in such cases is to use abstractions of components $M_1$ and $M_2$.

As discussed in Section 2, finite over-approximations, or may abstractions, of a component, are typically used to guarantee correctness with respect to some property. May abstractions have more behavior than the system that they abstract. As such, counterexamples that are detected using may abstractions may be spurious. On the other hand, finite under-approximations, or must abstractions, are better suited for detecting property violations that are not spurious. Assume-guarantee rules allow us to infer correctness of a system based on local correctness checks of the system components. In this setting, it makes sense to use may abstractions of components $M_1$ and $M_2$, since $\langle A \rangle M_1^{\text{may}} \langle P \rangle =\langle A \rangle M_1 \langle P \rangle$ and $\langle \text{true} \rangle M_2^{\text{may}} \langle A \rangle =\langle \text{true} \rangle M_2 \langle A \rangle$.

We can therefore use the may abstractions to check the two premises of the assume-guarantee rule. If both premises hold for the may abstraction it follows that the premises hold for the original un-abstracted components, and hence $P$ holds on the composition $M_1 \parallel M_2$. Let us now analyze the two premises given $M_1^{\text{may}}$ and $M_2^{\text{may}}$. 
• **Premise 1:** If \( \langle A \rangle M_1^{may} \langle P \rangle \) holds, then we know that \( \langle A \rangle M_1 \langle P \rangle \) holds, which completes this check. If \( \langle A \rangle M_1^{may} \langle P \rangle \) does not hold, then we obtain a counterexample, say \( c \). This counterexample may be spurious either due to abstraction or due to the approximation introduced by assumption inference. Similarly to a CEGAR approach, we can determine whether \( c \) is spurious due to abstraction by checking if it can be simulated to completion on the infinite component \( M_1 \), using an operation \( simulate(c,M_1) \). Since \( c \) is finite, simulation is possible. If \( c \) is not a valid execution of \( M_1 \), it means that it is a spurious counterexample, due to abstraction. If it is a valid execution of \( M_1 \), then it means that premise 1 does not hold, and therefore \( A \) is too approximate to make premise 1 pass. Below we will describe a new learning framework that is able to use this information to automatically refine the abstraction \( M_1^{may} \) or the assumption \( A \), respectively.

• **Premise 2:** If \( \langle true \rangle M_2^{may} \langle A \rangle \) holds, then we know that \( \langle true \rangle M_2 \langle A \rangle \) holds, which completes this check. If \( \langle true \rangle M_2^{may} \langle A \rangle \) does not hold, then we obtain a counterexample, say \( c \), which needs to be analyzed. This time, \( c \) may be spurious due to the abstraction of \( M_1 \) or \( M_2 \); furthermore, \( c \) may be spurious due to the approximation in the assumption. We therefore simulate \( c \) on \( M_2 \), using operation \( simulate(c,M_2) \). If \( c \) is not a valid execution of \( M_2 \), it means that it is a spurious due to abstraction in \( M_2 \), and abstraction-refinement is needed for \( M_2 \). If \( c \) is a valid execution of \( M_2 \), then it means that premise 2 does not hold, and this may be an indication of a real error or it may mean that the counterexample is spurious either due to abstraction of \( M_1 \) or to approximation in \( A \). Below, we describe how the proposed learning framework uses this information to automatically refine the component abstractions or the assumption \( A \), as needed.

### 4.1 Learning for Assumption Inference

Let us now revisit the learning framework for finite-state systems and see how it can be extended for infinite-state. As discussed, \( L^* \) needs a teacher that can answer membership queries and conjectures.

• **Membership queries:** A membership query needs to check if a finite word \( s \) over the alphabet \( \alpha A_w \) should be included in the language of \( A_w \). Similar to the finite-state case (see Figure 1) we are interested in finding out whether in the context of \( s \), \( M_1 \) violates \( P \) or not. We first check \( \langle s \rangle M_1^{may} \langle P \rangle \). If it leads to no error, it means this is true for \( M_1 \) as well (since the abstraction is conservative) and \( true \) is returned to \( L^* \). If an error is detected, the reported counterexample \( c \) is checked to see if it corresponds to a real trace in \( M_1 \). This amounts to symbolically simulating \( c \) on \( M_1 \), denoted by \( simulate(c,M_1) \). Since \( s \) is finite, \( c \) is finite too so the simulation is possible and will terminate. If the result of simulation is that this is a real trace, indicating a real error in \( M_1 \), \( false \) is returned to \( L^* \). However, if this is not a real error, the spurious counterexample \( c \) is used to refine \( M_1^{may} \) and the membership check is repeated. The abstraction-refinement denoted by \( refine(c) \) is described in more detail below.

When \( L^* \) produces an assumption \( A_i \), then the new framework needs to check whether \( A_i \) can be used to complete the assume-guarantee reasoning of the system. In other words, it needs to check Premises 1 and 2. For these checks, the framework uses \( M_1^{may} \) and \( M_2^{may} \), respectively, as described above.

When the safety check associated with premise 1 passes, then the framework proceeds with the permissiveness check involving \( M_2 \). If, however, a counterexample \( c \) is obtained, then \( c \) needs to be analyzed as described below (see Figure 2).

• **Counterexample Analysis 1:** If the result of \( simulate(c,M_1) \) is that \( c \) is a violating execution of \( M_1 \) then \( A_i \) must be refined; \( c \uparrow \alpha A_i \) is returned to \( L^* \), and \( L^* \) will work on creating a new approxima-
tion of the assumption. If \( c \) is not violating in \( M_1 \) then \( M_1^{\text{may}} \) must be refined, using \( \text{refine}(c) \) and the safety check is performed again with the new abstraction.

The permissiveness check consists of applying premise 2 using \( M_2^{\text{may}} \) as described above. If the check passes, we can conclude that \( \langle \text{true} \rangle M_1 \parallel M_2 \langle P \rangle \) holds. If not, we analyze the returned counterexample \( c \) as described below (see Figure 2).

- **Counterexample Analysis 2:** If the result of \( \text{simulate}(c, M_2) \) shows that \( c \) is not a real counterexample, then \( M_2^{\text{may}} \) is refined using \( \text{refine}(c) \), and the permissiveness check is repeated. If the result however indicates a real trace in \( M_2 \), then we must further analyze \( c \) to determine if it uncovers a real violation in the system or not.

Similar to the finite-state case (Figure 1), this further analysis amounts to performing a query for \( c \uparrow A \alpha w \) on \( M_1 \). Note that in the infinite-state case, performing a query might result in further refinements for \( M_1 \) and corresponding subsequent checks for the new abstractions (as described in the query check above). If the result of the analysis is that the assumption needs to be refined, the counterexample \( c \uparrow A \alpha w \) is returned to \( L^* \), which will work on creating a new approximation for \( A \), based on new membership queries and conjecture checks.

Note that this framework has an important characteristic: the information that is communicated to \( L^* \) is always correct with respect to the concrete system. As a result, refinement of abstractions does not require for \( L^* \) to restart learning.

In our proposed framework, abstraction refinement is applied whenever a violating trace \( t \) is discovered that belongs to a \( \text{may} \) abstraction (\( M_1^{\text{may}} \) or \( M_2^{\text{may}} \)) but not to the corresponding un-abstracted components (\( M_1 \) and \( M_2 \) respectively). Consequently \( t \) must contain a \( \text{may} \) transition that has no correspondence in the un-abstracted system. We use a simple strategy based on weakest precondition calculations \([30]\) to compute new abstraction predicates that are guaranteed to eliminate the spurious transitions. Given a counterexample \( t \) as a sequence of transitions \( \{\tau_1, \tau_2, ..., \tau_n\} \) we compute refinement predicates \( \text{wp}(\text{true}, t) \) by using weakest preconditions recursively based on the following definition:

\[
\text{wp}(\phi, t) = \text{wp}(\text{wp}(\phi, \tau_n), \{\tau_1, \tau_2, ..., \tau_{n-1}\}).
\]

### 4.2 Correctness and Termination

We argue now the correctness and the termination for the proposed algorithms. Our framework returns true only if both \( \langle A \rangle M_1^{\text{may}} \langle P \rangle \) and \( \langle \text{true} \rangle M_2^{\text{may}} \langle A \rangle \) hold. Since the assume guarantee rule is sound, it follows that \( \langle \text{true} \rangle M_1^{\text{may}} \parallel M_2^{\text{may}} \langle P \rangle \) holds, and since the \( \text{may} \) abstractions are over-approximations, it follows that \( \langle \text{true} \rangle M_1^{\text{may}} \parallel M_2^{\text{may}} \langle P \rangle \) holds as well. On the other hand, if the framework reports an error, it finds a trace which is both a trace of \( M_2 \) and of \( M_1 \) and it leads to a violation of \( P \), hence it is an error in \( M_1 \parallel M_2 \) as well.

For infinite-state components, the abstraction-refinement algorithm used in our proposed framework may not always terminate. However, from previous work on automatic abstraction refinement \([24]\), we know that if a component \( M \) has a finite bisimulation quotient, then abstraction-refinement (based on weakest preconditions calculations) converges to that finite quotient. It follows that there is a refinement iteration bound \( i \) such that \( M^{\text{may}} \) is bisimilar to \( M \) (in our case the argument applies to \( M_1 \) and \( M_2 \)). Since bisimulation implies trace equivalence it follows that \( \mathcal{L}(M^{\text{may}}) = \mathcal{L}(M) \) at that bound. Once that bound is reached, no abstraction-refinement is performed, and the obtained counter-examples will either be returned to \( L^* \) for assumption-refinement or they will be returned to the user as real errors, in which case the computation will terminate. By the correctness of \( L^* \) we are guaranteed that it will eventually
produce $A_w$ wrt. $M_1^{\text{may}}$ and property $P$. During this step, checking premise 1 will return true (by definition of $A_w$) and checking premise 2 will either return true and terminate, or return a counterexample. This counterexample is a trace in $L(M_2^{\text{may}}) = L(M_2)$ that is not in $L(A_w)$. Since $A_w$ is both safe and permissive, counterexample analysis will return false and the framework will terminate.

The framework may terminate earlier, not necessarily when it reaches bisimulation quotients for $M_1$ and $M_2$, but as soon as the abstractions $M_1^{\text{may}}$ and $M_2^{\text{may}}$ and the inferred assumption $A$ are good enough to show that the two premises hold or to expose a real counterexample.

5 Example: A simple communication protocol

In this section, we illustrate the presented algorithms through a very simple communication protocol. The protocol consists of two components, Sender and Receiver. We analyze two different versions of the protocol, one for the case where its components are finite-state, and one where they are infinite-state.

5.1 Finite State Protocol

The Sender and Receiver for the finite-state case are illustrated in Figure 3. The Sender starts by receiving input from the environment. It subsequently sends a message to the Receiver, and waits for an acknowledgement. The Receiver, upon a message being sent by the Sender produces an output, and subsequently acknowledges receipt of the message. The desired property from the environment’s perspective is that actions in and out alternate, with in occurring first. As illustrated, the compositional verification framework produces two assumptions; the second assumption is suitable for completing the assume-guarantee reasoning for this example. Note that the assumption has 2 states, which is one state less than the Receiver. The example is small and used for illustrative purposes, so the benefits of compositional verification are modest.

5.2 Infinite State Protocol

Now let us modify the example with an infinite-state Sender. The Sender has a variable $x$, initially set to 0, and where $x$ is in the domain of natural numbers. In Figure 4 we represent the component in terms
of pseudo-code, displayed in terms of its control flow. Similarly to the finite-state case, the Sender starts by receiving input from the environment, which prompts it to perform a read, the results of which are stored in x. In other words, this represents the non-deterministic assignment of a natural number value to variable x, modeling in term the communication of a value from the environment. Subsequently, x is set to its previous value modulo 5. The control flow then branches depending on whether x > 5, in which case it communicates an invalid message, and otherwise it sends a valid message, and waits for an acknowledgement from the Receiver. The Receiver may now receive a valid message, in which case it behaves as in the finite-state case, but also an invalid message. In the latter case, it directly transits to its initial state and waits for the next message. The property is identical to the finite-state case.

Assume that one first creates an abstraction of Sender based on predicates x = 0 and x > 0. The abstraction is illustrated in Figure 5 whereas, Figure 6 depicts the learning process for this infinite state system. As illustrated in Figure 6, when the learning algorithm queries the abstraction for "sendInvalid", the answer to the query is negative. The detailed trace at the top of Figure 6 provides the exact trace of the abstracted sender that violates property P. However, if this trace is simulated on the concrete Sender component of Figure 4, we can detect that it is impossible for (x > 5) to hold right after performing operation (x := x%5); therefore, this initial abstraction needs to be refined. If one adds predicates x > 5 and x ≤ 5, then the learning framework is able to show that P is satisfied with assume guarantee reasoning. Note that the assumptions obtained are similar to the infinite-state case, where send is now split in two actions, sendValid and sendInvalid.

6 Interface Generation for Infinite-State Components

Assumptions are closely related to the notion of component interfaces. Intuitively, a component interface summarizes aspects of a component that are relevant to its customers. Traditionally, component interfaces have been of a purely syntactic form, that included information about the services/methods that can be invoked on the component, and their signatures, meaning the numbers and types of arguments and their return values. However, there is a recognized need for richer interfaces that capture additional, behavioral, aspects of a component.

Temporal interfaces, as introduced in Section 2, are richer interfaces that capture ordering relation-
Figure 5: Abstracted sender - version 1

Figure 6: Learning-based AG reasoning for infinite system
ships between invocations of component methods. Ideally, a temporal interface should precisely represent the component in all its intended usages. In other words, it should include all the good interactions, and exclude all problematic interactions.

In such a context, component interfaces have the same flavor as assumptions that relate to safety properties, as studied in the previous section. However typically an interface summarizes the component *irrespective* of the environment in which the component is to be introduced, while we have seen that an assumption (used in compositional reasoning) serves as a potentially imprecise interface that is sufficient for breaking up a targeted verification problem into simpler problems; all components that participate in the verification problem are known and available.

In fact, a precise interface is similar to the weakest component assumption. It is therefore characterized in terms of two properties, safety, and permissiveness. For simplicity, we consider here error states that are not introduced by explicit properties, but are rather assumed to represent undesirable component states (e.g. assertion violations in the component’s code). Interfaces can be learned through frameworks similar to those we developed for assume-guarantee reasoning; queries, and the part of conjectures related to safety, are answered in an identical way.

Permissiveness, however, is more difficult because the environment of the targeted component is not available. The example in Figure 7 shows how a permissiveness check could be performed. Component $M$ has states named $s_0$, $s_1$, $s_2$, $s_{error}$ and interface $A$ has states named $s_0$, $s_1$ and $s_{error}$. A permissiveness check needs to detect sequences that are blocked by the interface but legal in the component. Such sequences identify that the interface is not permissive. This can be performed by checking reachability, in $M \parallel A$, of states of the form $[s_i, s_{error}]$, where $0 \leq i \leq 2$. According to this check, trace $\langle a, b \rangle$ leading to state $[s_1, s_{error}]$ in the composition could be an indication that $A$ is not permissive. However this is not true, since the same sequence of actions leads to $[s_{error}, s_{error}]$ on a different path, due to non-determinism. This happens because the alphabet of the assumption is $\{a, b\}$, meaning that action $c$ in $M$ is considered as a $\tau$ from the point of view of $A$. In the figure, this is illustrated as a $\tau$ action covering action $c$.

This example illustrates the fact that non-determinism in component $M$ may cause spurious counterexamples in the permissiveness reachability check described above. As a consequence, precise characterization of permissiveness requires determinization of component $M$, which can be performed using subset construction. The permissiveness check is therefore NP-hard [1], and can be inefficient in prac-

![Figure 7: Checking for Permissiveness](image-url)
D. Giannakopoulou and C. S. Păsăreanu

practice. Several approaches have been proposed to deal with this problem. Unless determinization is a viable solution for a targeted component $M$ [3], heuristic approaches are often used to determine whether a counterexample is spurious [11, 15].

Let us examine now the case of infinite-state component $M$. Abstraction is again needed to reason about such components. However using may abstractions alone turns out to be insufficient, because the generated interfaces may be overly restricting (due to the spurious error traces present in the abstraction). In previous work [30], we have shown the following result:

Assume a component $M$, a may abstraction $M^{\text{may}}$ and a must abstraction $M^{\text{must}}$ for $M$. If an interface $A$ for $M$ is safe with respect to $M^{\text{may}}$ and permissive with respect to $M^{\text{must}}$, then $A$ is safe and permissive with respect to $M$.

Based on this result, we developed a framework [30] that interleaves abstraction-refinement and L* learning for the automated generation of interfaces for infinite-state components. The framework uses both $M^{\text{may}}$ and $M^{\text{must}}$ to compute an interface for $M$ that is both safe and permissive. The abstractions are refined automatically from counterexamples obtained during the reachability checks performed by the framework.

It is interesting to note that in case component $M$ is observationally deterministic (i.e. deterministic with respect to its interface actions), the must abstraction is deterministic as well. Thus its use enables us to avoid the exponentially expensive determinization step that is required when working with non-deterministic components. The idea of the framework, based on the result above, is to use $M^{\text{may}}$ for the safety check, and $M^{\text{must}}$ for the permissiveness check.

The safety check is similar to the compositional verification framework. Let us therefore analyze how the permissiveness check is performed with $M^{\text{must}}$, when $M$ is observationally deterministic. If in $M^{\text{must}} \parallel A$ it is not possible to reach an error state in $A$ that is an accepting state in $M^{\text{must}}$, then it means that $A$ permissive, in which case the framework produces $A$ as a safe and permissive interface for $M$. If, however, such a combined state is reachable by some counterexample $c$, then $c$ is analyzed as follows. If $c$ leads to an error state in $M$, then $M^{\text{must}}$ needs to be refined to avoid this discrepancy. Otherwise, $c$ represents a real counterexample to the permissiveness of $A$ and it is returned to L* for refining the assumption. If, on the other hand, $M^{\text{must}}$ is observationally non-deterministic, it must be determinized prior to performing the test.

In conclusion, we use both may and must abstractions for interface generation of infinite state components. In contrast, may abstractions are sufficient for compositional verification, because component $M_2$ is known and can be used for selective permissiveness checks.

7 Related Approaches

We briefly describe here some of the related work that combines abstraction and compositional reasoning in the context of infinite system analysis. The Magic tool performs verification of concurrent, message-passing C programs using abstraction-refinement in a compositional way; the work has been extended with a two-level abstraction scheme, but it does not use assume-guarantee style verification. The Blast tool uses predicate abstraction and has been extended to perform assume-guarantee reasoning for checking race conditions in multi-threaded C code [19]. In contrast to our work, it targets shared memory communicating programs, and therefore it uses a different style of assume guarantee rule. Moreover, the approach used by Blast is not based on learning. Bandera [9, 27] was aimed at verifying concurrent Java
programs. Bandera employed data abstraction and modular reasoning with user-supplied assumptions, but not automated assumption generation and CEGAR, as we do here. As already mentioned, algorithms for interface synthesis for infinite state components have been presented in [1, 3, 30]. Recently, L* learning has been combined with symbolic execution to automatically generate interfaces for Java classes that include methods with parameters [16].

8 Conclusion

In this paper, we discussed the different types of abstractions that can be used for applying learning-based assume-guarantee reasoning and interface generation to infinite-state systems. We proposed a new framework for automated compositional verification, and illustrated it with a simple example. At the present time, we are not able to perform extensive experimental evaluation of the framework. For the presented example, we carried out all steps, except for abstraction refinement, automatically. In the future, we plan to implement and evaluate extensively the proposed framework. We are also interested in evaluating the quality of the obtained abstractions and assumptions, and the efficiency of our interleaved approach.

Acknowledgment

Corina Păsăreanu uses this occasion to remember the wonderful times she spent at Kansas State University during her graduate studies. She would like to thank Dave Schmidt and the other professors in the Department of Computing and Information Sciences for having started her interest in abstraction and compositional reasoning, which have been the subject of her thesis and a recurrent research theme in her work since then.

References

[1] Rajeev Alur, Pavol Cerný, P. Madhusudan & Wonhong Nam (2005): Synthesis of interface specifications for Java classes. In Palsberg & Abadi [24], pp. 98–109, doi:10.1145/1040305.1040314. Available at http://dl.acm.org/citation.cfm?id=1040305.
[2] Dana Angluin (1987): Learning Regular Sets from Queries and Counterexamples. Inf. Comput. 75(2), pp. 87–106, doi:10.1016/0890-5401(87)90052-6.
[3] Dirk Beyer, Thomas A. Henzinger & Vasu Singh (2007): Algorithms for Interface Synthesis. In Damm & Hermanns [11], pp. 4–19, doi:10.1007/978-3-540-73368-3_4.
[4] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha & Helmut Veith (2003): Modular Verification of Software Components in C. In: ICSE, pp. 385–395, doi:10.1109/ICSE.2003.1201217.
[5] Sagar Chaki, Joël Ouaknine, Karen Yorav & Edmund M. Clarke (2003): Automated Compositional Abstraction Refinement for Concurrent C Programs: A Two-Level Approach. Electr. Notes Theor. Comput. Sci. 89(3), pp. 417–432, doi:10.1016/S1571-0661(05)80004-0.
[6] Marsha Chechik & Martin Wirsing, editors (2009): Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. Lecture Notes in Computer Science 5503, Springer, doi:10.1007/978-3-642-00593-0.
[7] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu & Helmut Veith (2000): Counterexample-Guided Abstraction Refinement. In Emerson & Sistla [12], pp. 154–169, doi:10.1007/10722167_15.
[8] Jamieson M. Cobleigh, Dimitra Giannakopoulou & Corina S. Păsăreanu (2003): “Learning Assumptions for Compositional Verification.” In Garavel & Hatcliff [13], pp. 331–346, doi:10.1007/3-540-36577-X_24

[9] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby & Hongjun Zheng (2000): “Bandera: extracting finite-state models from Java source code.” In: ICSE, pp. 439–448, doi:10.1145/337180.337234.

[10] Patrick Cousot & Radhia Cousot (1977): “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.” In Graham et al. [17], pp. 238–252, doi:10.1145/512950.512973. Available at http://dl.acm.org/citation.cfm?id=512950.

[11] Werner Damm & Holger Hermanns, editors (2007): “Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings.” Lecture Notes in Computer Science 4590, Springer.

[12] E. Allen Emerson & A. Prasad Sistla, editors (2000): “Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings.” Lecture Notes in Computer Science 1855, Springer.

[13] Hubert Garavel & John Hatcliff, editors (2003): “Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings.” Lecture Notes in Computer Science 2619, Springer.

[14] Dimitra Giannakopoulou & Corina S. Păsăreanu (2008): “Special issue on learning techniques for compositional reasoning.” Formal Methods in System Design 32(3), pp. 173–174, doi:10.1007/s10703-008-0054-9.

[15] Dimitra Giannakopoulou & Corina S. Păsăreanu (2009): “Interface Generation and Compositional Verification in JavaPathfinder.” In Chechik & Wirsing [6], pp. 94–108, doi:10.1007/978-3-642-00593-0_7.

[16] Dimitra Giannakopoulou, Zvonimir Rakamaric & Vishwanath Raman (2012): “Symbolic Learning of Component Interfaces.” In Miné & Schmidt [22], pp. 248–264, doi:10.1007/978-3-642-33125-1_18.

[17] Robert M. Graham, Michael A. Harrison & Ravi Sethi, editors (1977): “Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977.” ACM. Available at http://dl.acm.org/citation.cfm?id=512950.

[18] Michael Hanus, editor (2007): “Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007.” Lecture Notes in Computer Science 4354, Springer.

[19] Thomas A. Henzinger, Ranjit Jhala & Rupak Majumdar (2004): “Race checking by context inference.” In: PLDI, pp. 1–13, doi:10.1145/996841.996844.

[20] Thomas A. Henzinger, Ranjit Jhala & Rupak Majumdar (2005): “Permissive interfaces.” In: Wermelinger & Gall [32], pp. 31–40, doi:10.1145/1081706.1081713.

[21] Cliff B. Jones (1983): “Tentative Steps Toward a Development Method for Interfering Programs.” ACM Trans. Program. Lang. Syst. 5(4), pp. 596–619, doi:10.1145/69575.69577.

[22] Antoine Miné & David Schmidt, editors (2012): “Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012, Proceedings.” Lecture Notes in Computer Science 7460, Springer, doi:10.1007/978-3-642-33125-1.

[23] Kedar S. Namjoshi & Robert P. Kurshan (2000): “Syntactic Program Transformations for Automatic Abstraction.” In Emerson & Sistla [12], pp. 435–449, doi:10.1007/10722167_33.

[24] Jens Palsberg & Martín Abadi, editors (2005): “Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005.” ACM. Available at http://dl.acm.org/citation.cfm?id=1040305.

[25] A. Pnueli (1985): “In transition from global to modular temporal reasoning about programs.” In: POPL 2005, Long Beach, California, USA, January 12-14, 2005. ACM. Available at http://dl.acm.org/citation.cfm?id=101969.101977.
[26] Andreas Podelski & Andrey Rybalchenko (2007): ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In Hanus [13], pp. 245–259, doi:10.1007/978-3-540-69611-7/16

[27] Corina S. Pășăreanu (2001): Abstraction and Modular Reasoning for the Verification of Software. PhD Thesis, Kansas State University.

[28] Corina S. Pășăreanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh & Howard Barringer (2008): Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. Formal Methods in System Design 32(3), pp. 175–205, doi:10.1007/s10703-008-0049-6

[29] Ronald L. Rivest & Robert E. Schapire (1993): Inference of Finite Automata Using Homing Sequences. Inf. Comput. 103(2), pp. 299–347, doi:10.1006/inco.1993.1021

[30] Rishabh Singh, Dimitra Giannakopoulou & Corina S. Pășăreanu (2010): Learning Component Interfaces with May and Must Abstractions. In Touili et al. [31], pp. 527–542, doi:10.1007/978-3-642-14295-6/45

[31] Tayssir Touili, Byron Cook & Paul Jackson, editors (2010): Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings. Lecture Notes in Computer Science 6174, Springer, doi:10.1007/978-3-642-14295-6

[32] Michel Wermelinger & Harald Gall, editors (2005): Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005. ACM.