

Context Synthesis

Dimitra Giannakopoulou² and Corina S. Păsăreanu¹

¹ Carnegie Mellon Silicon Valley/

² NASA Ames Research Center,
Moffett Field, CA 94035, USA

{`dimitra.giannakopoulou,corina.s.pasareanu`}@nasa.gov

Abstract. With the advent of component-based and distributed software development, service-oriented computing, and other such concepts, components are no longer viewed as parts of specific systems, but rather as open systems that can be reused, or connected dynamically, in a variety of environments to form larger systems. Reasoning about components as open systems is different from reasoning about closed systems, since property satisfaction may depend on the context in which a component may be introduced.

Component interfaces are an important feature of open systems, since interfaces summarize the expectations that a component has from the contexts in which it gets introduced. Traditionally, component interfaces have been of a purely syntactic form, including 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 aspects of a component. For example, interfaces may characterize legal sequences of invocations to component services.

Generating compact and yet useful component interfaces is a challenging task to perform manually. Over the last decade, several approaches have been developed for performing *context synthesis*, i.e., generating component interfaces automatically. This tutorial mostly reviews such techniques developed by the authors, but also discusses alternative techniques for context synthesis.

1 Introduction

With the advent of component-based and distributed software development, service-oriented computing, and other such concepts, components are no longer viewed as parts of specific systems, but rather as open systems that can be reused, or connected dynamically, in a variety of environments to form larger systems. Reasoning about components as open systems is different from reasoning about closed systems, since property satisfaction may depend on the context in which a component may be introduced. As a result, a component satisfies or violates a property only when the property is satisfied or violated by the component irrespective of context. For all other cases, meaningful analysis would consist of synthesizing a characterization of all contexts in which the component

satisfies the desired property. We refer to such analysis as “context synthesis”, and the result of the synthesis a “component interface”.

Component interfaces therefore summarize the expectations that a component has from the contexts in which it gets introduced. Traditionally, component interfaces have been of a purely syntactic form, including 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 aspects of a component. For example, “temporal” interfaces [3], which are the focus of this tutorial, describe legal sequences of service invocations or method calls to a component. The purpose is to document (for clients of a component) what sequences of calls could lead to undesirable component states and should therefore be avoided.

Generating compact and yet useful component interfaces is a challenging task to perform manually. Over the last decade, several approaches have been developed for generating component interfaces automatically. This tutorial reviews techniques for interface generation of components with respect to safety properties. We discuss in depth some techniques developed by the authors, but also present and discuss alternative techniques, and provide references to some new trends in this research area.

Context synthesis is closely related to compositional reasoning methods for model checking. Compositional verification presents a “divide and conquer” approach to the state-explosion problem [9] associated with model checking. It decomposes the properties of the system into local properties of the system components. Each component is checked separately against its local properties; the combination of these simpler checks guarantees the correctness of the global property on the entire system.

Analysis of components in isolation for compositional verification will often return results that are not meaningful. The reason is again that components usually rely on some features of the environments in which they are introduced. One therefore needs to incorporate some knowledge of the contexts in which the components are expected to operate correctly. Assume-guarantee reasoning [19, 24] addresses this issue by making explicit use of assumptions in component verification. Assumptions are akin to interfaces and they document expectations of a component from its environment in order to fulfill its guarantees. Assume-guarantee rules are then used to merge the results of individual component verification steps for verification of system-level properties.

The fundamental difference between an interface and an assumption in the context of reasoning about safety properties, is that an interface summarizes the component with respect to *all* the possible environments in which the component may be introduced. On the other hand, an assumption serves as a potentially imprecise interface that only needs to reflect interactions with a *specific* environment, representing the rest of the components in the analyzed system. We note that in the context of compositional verification, all the components that participate in the verification problem are known and available. As a result,

context synthesis takes the form of assumption generation in compositional verification, and can be performed more efficiently than interface generation due to the availability of an actual component environment.

The rest of this paper is organized as follows. We provide background for our work in Section 2, followed by a characterization of precise (safe and permissive) component interfaces in the context of safety property checking for finite state systems in Section 3. In Section 4, we present several algorithms for automated interface generation. A construction of what we call the “weakest assumption”, corresponding to a precise component interface, is presented first. We then present an alternative approach that uses the L* learning algorithm to compute component interfaces in an iterative manner. In Section 5 we address the problem of generating interfaces for infinite state components. Context synthesis is subsequently discussed in the context of compositional verification in Section 6. Finally, we discuss implementation and applicability of these techniques and present some open research topics in this domain in Section 7.

2 Background

In this section we introduce labeled transition systems (LTSs) [20], the formalism that we use to model components. We present the definition of traces and parallel composition for LTSs and also present how safety properties are checked. We then introduce assume-guarantee reasoning and the L* algorithm that we use to automatically synthesize interfaces and assumptions.

2.1 Labeled Transition Systems (LTSs)

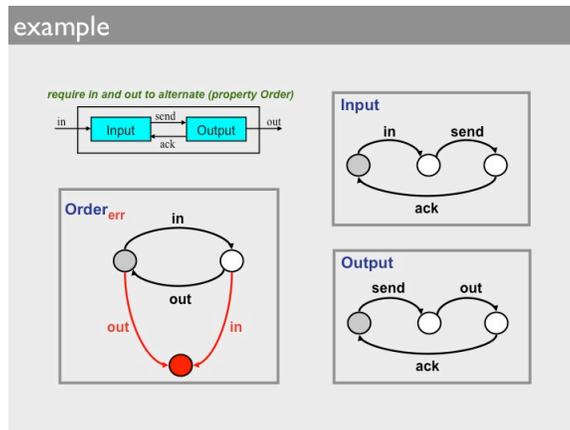


Fig. 1. Example

Let \mathcal{Act} be the universal set of observable actions and let τ denote a local *unobservable* action. Let π denote a special *error state*, which models safety violations; the error state has no outgoing transitions. Formally, an LTS M is a four-tuple $\langle Q, \alpha M, \delta, q_0 \rangle$ where:

- Q is a finite non-empty set of states
- $\alpha M \subseteq \mathcal{Act}$ 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

Let Π denote the LTS $\langle \{\pi\}, \mathcal{Act}, \emptyset, \pi \rangle$. An LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$ is *non-deterministic* if it contains τ -transitions or if there exists $(q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, M is *deterministic*.

2.2 Traces

A *trace* t of LTS M is a finite sequence of observable actions that label the transitions that M can perform starting from its initial state (ignoring the unobservable τ -transitions). We sometimes denote by t both a trace and its *trace LTS*. For a trace t of length n , its trace LTS $lts(t)$ consists of $n + 1$ states, such that there is a transition between states i and $i + 1$ on the i^{th} action in trace t , for $1 \leq i \leq n$. The set of all traces of an LTS M is the language of M , denoted $\mathcal{L}(M)$; $errTr(M)$ denotes the set of traces that lead to π , which are called the *error traces* of M .

For $\Sigma \subseteq \mathcal{Act}$, we use $t \uparrow \Sigma$ to denote the trace obtained by removing from t all occurrences of actions $a \notin \Sigma$. Similarly, $M \uparrow \Sigma$ is defined to be an LTS over alphabet Σ which is obtained from M by renaming to τ all the transitions labeled with actions that are not in Σ .

2.3 Parallel Composition

Let $M = \langle Q, \alpha M, \delta, q_0 \rangle$ and $M' = \langle Q', \alpha M', \delta', q'_0 \rangle$. M *transits* into M' with action a , written $M \xrightarrow{a} M'$, if and only if $(q_0, a, q'_0) \in \delta$ and either $Q = Q'$, $\alpha M = \alpha M'$, and $\delta = \delta'$ for $q'_0 \neq \pi$, or, in the special case where $q'_0 = \pi$, $M' = \Pi$.

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

Formally, let $M_1 = \langle Q^1, \alpha M_1, \delta^1, q_0^1 \rangle$ and $M_2 = \langle Q^2, \alpha M_2, \delta^2, q_0^2 \rangle$ be two LTSs. If $M_1 = \Pi$ or $M_2 = \Pi$, then $M_1 \parallel M_2 = \Pi$. Otherwise, $M_1 \parallel M_2$ is an LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$, 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 δ is defined as follows, where a is either an observable action or τ :

$$\frac{M_1 \xrightarrow{a} M'_1, a \notin \alpha M_2}{M_1 \parallel M_2 \xrightarrow{a} M'_1 \parallel M_2} \quad \frac{M_2 \xrightarrow{a} M'_2, a \notin \alpha M_1}{M_1 \parallel M_2 \xrightarrow{a} M_1 \parallel M'_2}$$

$$\frac{M_1 \xrightarrow{a} M'_1, M_2 \xrightarrow{a} M'_2, a \neq \tau}{M_1 \parallel M_2 \xrightarrow{a} M'_1 \parallel M'_2}$$

2.4 Safety Properties

In our context, properties are modeled as *safety LTS*s. A *safety LTS* is a deterministic LTS that contains no π states. A *safety property* is specified as a safety LTS P , whose language $\mathcal{L}(P)$ defines the set of acceptable behaviors over αP .

For an LTS M and a safety LTS P such that $\alpha P \subseteq \alpha M$, we say that M satisfies P , denoted $M \models P$, if and only if $\forall t \in \mathcal{L}(M) : (t \uparrow \alpha P) \in \mathcal{L}(P)$.

When checking a property P , an *error LTS* denoted P_{err} is created, which traps possible violations with the π state. Formally, the error LTS of a property $P = \langle Q, \alpha P, \delta, q_0 \rangle$ is $P_{err} = \langle Q \cup \{\pi\}, \alpha P_{err}, \delta', q_0 \rangle$, where $\alpha P_{err} = \alpha P$ and

$$\delta' = \delta \cup \{(q, a, \pi) \mid q \in Q, a \in \alpha P, \text{and } \nexists q' \in Q : (q, a, q') \in \delta\}$$

Note that the error LTS is *complete* (except for π), meaning each state other than the error state has outgoing transitions for every action in its alphabet. Also note that the error traces of P_{err} define the language of P 's complement.

As an example, consider the communication channel in Figure 1. It consists of an **Input** and an **Output** component; grey states are initial states. Actions **send** and **ack** are common to the alphabets of the two components and will be synchronized when the LTSs are composed. Property **Order** states that **inputs** and **outputs** come in matched pairs with **input** always preceding **output**. The error state is colored red.

2.5 Assume-Guarantee Reasoning

Concurrent software is inherently difficult to analyze. The problem of reaching a specific global state in a system with N finite-state components can be shown to be PSPACE-complete in N . What this means in practice is that the number of states in a concurrent system may in the worst case be exponential in the size of the components of the system.

One approach to dealing with this state explosion problem is compositional or local reasoning. In essence, the goal of compositional reasoning is to replace the analysis over the global state space with localized analyses, which consider each component by itself, together with a small abstraction of the environment of that component. The intuition behind this principle is that many systems can be viewed as “loosely coupled” collections of components; that is, the proportion of the behavior of one component behavior which influences that of another is small. Hence, there should be an advantage to doing localized reasoning.

Assume-guarantee reasoning [19, 24] is a compositional verification technique that uses assume-guarantee rules for verification. Assume-guarantee rules are proof rules that show how, by performing local verification steps on individual components of the system, one can safely deduce properties that refer to the entire system. The local verification steps usually involve some abstraction of the environment of each component, named *assumption*.

In the assume-guarantee reasoning paradigm, formulas are triples of the type $\langle A \rangle M \langle P \rangle$, 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 $E \parallel M \models P$. For LTS M and safety LTSs A and P , checking $\langle A \rangle M \langle P \rangle$ reduces to checking if state π is reachable in $A \parallel M \parallel P_{err}$. Note that when $\alpha P \subseteq \alpha A \cup \alpha M$, this is equivalent to $A \parallel M \models P$. Also note that we assume that M contains no π states.

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

$$\frac{\begin{array}{l} 1 : \langle A \rangle M_1 \langle P \rangle \\ 2 : \langle true \rangle M_2 \langle A \rangle \end{array}}{\langle true \rangle M_1 \parallel M_2 \langle P \rangle}$$

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 circularity. Despite its simplicity, experience with applying compositional verification has shown this rule to be most useful in the context of checking safety properties. This rule can be extended for multiple components. Several other rules have also been defined in the literature. We have experimented with several rules in practice [23].

Weakest Assumption. For a given LTS component M and safety property P there is a natural notion of the *weakest assumption* A_w , such that $\langle A_w \rangle M \langle P \rangle$ holds. A_w characterizes all the possible environments E under which the property holds, i.e. $\forall E : M \parallel E \models P$ if and only if $E \models A_w$.

2.6 The L* Algorithm

L* is a learning algorithm that was developed by Angluin [2] and later improved by Rivest and Schapire [25]. L* learns an unknown regular language and produces a DFA that accepts it. Let U be an unknown regular language over some alphabet Σ . In order to learn U , L* needs to interact with a *Minimally Adequate Teacher*, from now on called *Teacher*. A Teacher must be able to correctly answer two types of questions from L*. 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 DFA 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 .

Characteristics of L^ .* L^* depends on the correctness of the Teacher in order to provide a number of guarantees. More specifically, L^* is guaranteed to terminate with a minimal automaton for the unknown language U . Moreover, each candidate DFA C that L^* constructs is smallest, in the sense that any other DFA consistent with the information provided to L^* has at least as many states as C . This characteristic of L^* makes it particularly attractive in the context of learning interfaces or assumptions, since in the frameworks that we describe, the candidates provided by L^* are combined with component models in model checking steps. Smaller state machines typically result in easier model checking problems. The conjectures made by L^* strictly increase in size; each conjecture is smaller than the next one, and all incorrect conjectures are smaller than the minimal automaton for language U . Therefore, if that minimal automaton has n states, L^* makes at most $n - 1$ incorrect conjectures. The number of membership queries made by L^* is $\mathcal{O}(kn^2 + n \log m)$, where k is the size of the alphabet of U , n is the number of states in the minimal DFA for U , and m is the length of the longest counterexample returned when a conjecture is made.

3 Component Interfaces

An interface characterizes the expectations that a component has from its environment. As discussed in the introduction, there is a recognized need for extending component interfaces beyond their traditional, purely syntactic form. In this tutorial paper, we focus on interfaces that describe legal sequences of service invocations or method calls to a component. 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. In other words, it should include all the good interactions, and exclude all problematic interactions. We describe this formally in the following.

Let Σ be the set of interaction points of an LTS M , where $\Sigma \subseteq \alpha M$. A word over Σ is considered *legal* if its execution cannot lead M to an error state, and is considered legal otherwise. Accordingly, we define two additional language sets for any LTS M . We use the term $\mathcal{L}_{illegal}(M) = errTr(M)$ to refer to the set of illegal executions of LTS M . The set of legal executions is defined as $\mathcal{L}_{legal}(M) = \Sigma^* \setminus \mathcal{L}_{illegal}(M)$. Note that we slightly abuse the term “executions” in this context, since the set of legal executions may contain words that cannot execute to completion in M , meaning that they do not correspond to traces in $\mathcal{L}(M)$. The reason why it is desirable to consider such words in the set of legal executions is that such words should never be disallowed in the behavior of M ’s environment since they can never be executed in the context of M , and could therefore never lead M to an error state.

Let us now assume that a component is represented as an LTS M , with Σ being the set of its interaction points with the environment. Assume also that an interface A is represented as an LTS over Σ . Then A is a precise interface for M if it satisfies two conditions:

1. **Safe.** Interface A is safe for M iff $\mathcal{L}_{legal}(A) \cap \mathcal{L}_{illegal}(M \uparrow \Sigma) = \emptyset$. Informally, this definition says that any legal word w in A can only trigger legal executions in M .
2. **Permissive.** Interface A is permissive for M iff $\mathcal{L}_{legal}(M \uparrow \Sigma) \subseteq \mathcal{L}_{legal}(A)$. Informally, every legal word in M should be represented by some legal word in A .

Note that safety is concerned with blocking behaviors while permissiveness is concerned with including behaviors. These two concepts are complementary in achieving an exact characterization of correct component usage. When dealing with component interfaces, it is therefore important to be able to determine whether a given interface is safe and permissive.

Let $M = \langle Q_M, \alpha_M, \delta_M, q_{0M} \rangle$ be the LTS description of a component, and let $A = \langle Q_A, \alpha_A, \delta_A, q_{0A} \rangle$ be an interface provided for M .

Checking for safety. Interface A is safe for M if and only if illegal states of M are not reachable in $A \parallel M$. Interface safety can therefore be performed by a reachability check, as supported by any standard model checker. Counterexamples correspond to illegal executions of M that are not blocked by A .

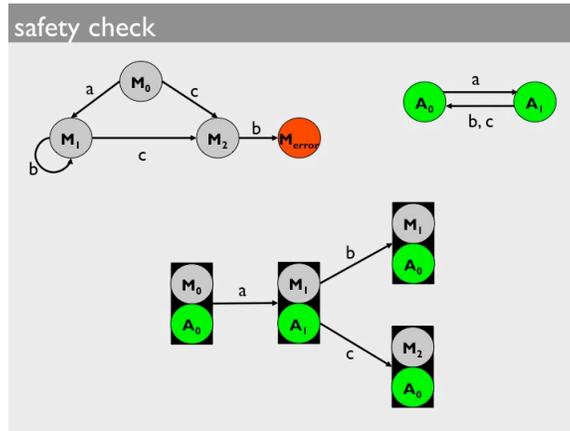


Fig. 2. Safety Check

Figure 2 is used to illustrate the safety check. States M_0 , M_1 , M_2 and M_{error} belong to component M while states A_0 and A_1 belong to interface A . The error state is no longer reachable in the composition.

Checking for permissiveness. To check permissiveness, we need to complete M with a sink state to obtain M_c , and A with an error state to obtain A_{err} . In $M_c \parallel A_{err}$, we then check for reachability of states that correspond to an error state in A_{err} and a non error state in M_c . A path leading to such states could

identify a legal word in M_c that is not accepted by A , reflecting the fact that A is not permissive. However, this check is not sufficient to determine permissiveness of an interface. This is illustrated by an example below.

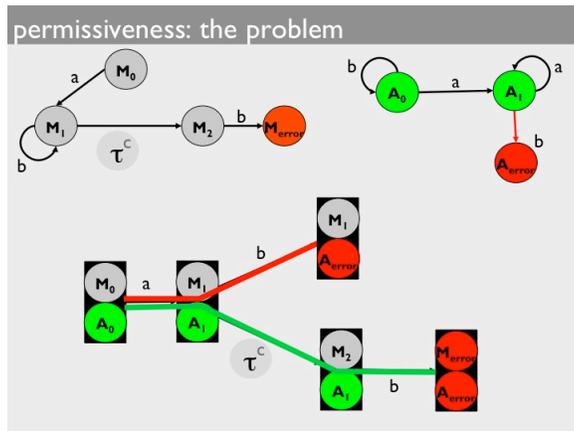


Fig. 3. Checking for Permissiveness

The example in Figure 3 shows the problem with the permissiveness check above. As before, states M_0, M_1, M_2, M_{error} belong to M and states A_0, A_1 and A_{error} belong to interface A . According to the above check, trace a, b leading to state $[M_1, A_{error}]$ in the composition could be an indication that A is not permissive enough. However this is not true, since the same path leads to $[M_{error}, A_{error}]$. This happens because the alphabet of the assumption is $\{a, b\}$, meaning that action c in M is considered as a τ from the point of view of A . In the figure, this is illustrated as a τ 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 practice.

Several approaches have been proposed to deal with this problem. Unless determinization is a viable solution for a targeted component M [14, 3], heuristic approaches are often used to determine whether a counterexample is spurious [1, 13]. Also, if non determinism is introduced through abstraction of a deterministic concrete component, this problem can sometimes be avoided, using a combination of over- and under- approximating abstractions [26].

In the next sections we discuss some of these solutions. We first present an approach that creates a safe and permissive interface by construction, and which involves determinization of the component. Subsequently, we describe an iterative learning-based approach that is based on safety and permissiveness

checks and which uses heuristics to avoid determinization of the component. We then discuss interface generation in the context of infinite-state components and abstraction.

4 Automated Interface Generation

Precise characterization of component interfaces is a difficult task to perform manually. Given the need for automated modular or compositional verification techniques warranted by the size of modern software and hardware systems, automated interface and assumption generation have been thoroughly investigated in the last decade. Our first attempt to automated interface generation consists of a construction that systematically builds finite-state machine interfaces for finite-state components and safety properties expressed as LTSs [14]. The built interfaces are safe and permissive by construction. Learning-based approaches to interface generation are subsequently discussed. These frameworks are based on the use of the L* algorithm for providing and gradually refining guesses of the desired interface.

4.1 Computing the Weakest Assumption

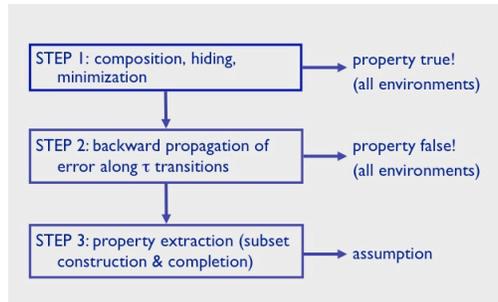


Fig. 4. Model Checking with Assumption Generation

We describe here an approach to building the weakest assumption for a component with respect to a safety property. The approach addresses the more general problem of model checking for *open* systems, i.e. components that interact with their environments. When model checking a component against a property, our algorithm returns one of the following three results: (i) the component satisfies the property for any environment; (ii) the component violates the property for any environment; or finally, (iii) the “weakest assumption” – an automatically generated assumption that characterizes exactly those environments in which the component satisfies the property.

The traditional approach to verifying a property of an open system is to check it for all the possible environments. The result of verification is either **true**, if the property holds for *all* the possible environments, or **false**, if there exists *some* environment that can lead the component to falsify the property. However this approach may be overly pessimistic and we advocate an optimistic view, which assumes a *helpful* environment. The reason is that software components are often required to satisfy properties only in specific environments, so it is natural to accept a component if there are *some* environments in which the component does not violate the property.

In our approach, the result of component verification is **true**, if the property holds for *all* environments, similar to the traditional approach. However, the result is **false** only if the property is falsified in *all* environments. If there exist *some* environments in which the component satisfies the property, the result of verification is not false, as in the traditional approach, but rather **true** in environments that satisfy the *weakest assumption*.

Figure 4 illustrates our approach together with the steps we follow to build the weakest assumption (that are described below).

Step 1: Composition and Minimization

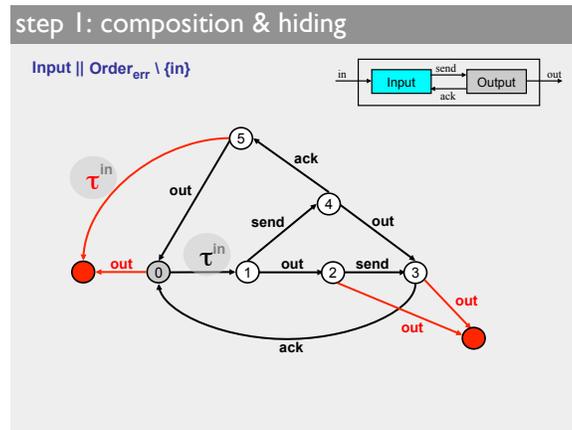


Fig. 5. Step 1: Composition and Hiding

Given an *open* system (described as an LTS) and a *property* LTS that may relate the behavior of the system with the behavior of the environment, the first step is to build the *composition* of the system with the *error* LTS of the property and to hide (i.e., turn into τ) the internal actions of the system. The resulting LTS can be minimized with respect to any equivalence that preserves (error) traces.

As an example, let us consider the communication channel from Section 2. We show here the computation of the weakest assumption for component `Input` with respect to property `Order`. Figure 5 depicts the result of composing `Input` with the error automaton for the property. The internal actions of the system, i.e. the transitions labeled `in`, were abstracted to τ . This is illustrated in Figure 5 by covering action `in` with action τ .

If the error state is not reachable in this composition, the property is **true** in any environment, and this is reported to the user. Otherwise, we determine whether there exist environments that can help the system avoid the error; this is achieved through the following steps.

Step 2: Backward Error Propagation

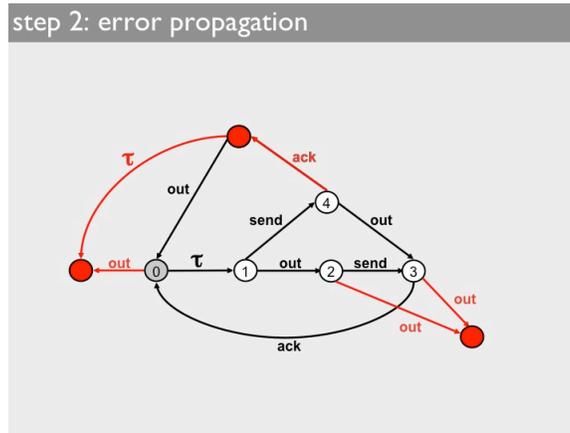


Fig. 6. Step 2: Error Propagation

This step first performs *backward propagation* of the error state over τ transitions, thus pruning the states where the environment cannot prevent the error state from being entered via one or more τ steps. We are interested only in the error traces, and therefore we also eliminate the states that are not backward reachable from the error state. If, as a result of this transformation, the initial state becomes an error state, it means that no environment can prevent the system from reaching the error state, so the property is **false** (for all environments) and this is reported to the user.

Consider again the composite system in Figure 5. As a result of backward propagation, we identify state 5 with the error state; the result is shown in Figure 6. The intuition here is that, if the component is in a state from which it

can violate the property by some number of internal moves, then no environment can prevent the violation from occurring.

Step 3: Property Extraction

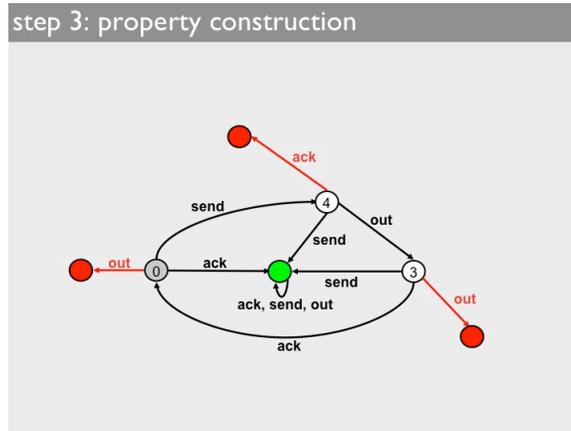


Fig. 7. Step 3: Property Extraction

This step builds the *property LTS* that is our assumption. It performs this in two stages; first it builds the *error LTS* for the assumption, from which it extracts the corresponding property LTS. Note that the LTS resulting from Step 2 might not be an error LTS (i.e. it might not be deterministic or complete), although it contains an error state. Recall from the background section that the error LTS is both deterministic and complete.

In order to get an error LTS we make the LTS obtained from step 2 deterministic by applying to it τ elimination and the subset construction [18], but by taking special care of the π state as follows. During subset construction, the states of the deterministic LTS that is being generated are *sets of states* in the original non-deterministic LTS. If any of these sets contains π , the entire set becomes π . Intuitively, a trace that non-deterministically may or may not lead to an error has to be considered as an error trace. Such non-determinism reflects the fact that, by performing a particular sequence of actions, the environment cannot guarantee that the component will avoid error states.

The resulting LTS is then completed. *Completion* is performed by adding a new “sink” state to the LTS, and adding a transition to this state for each missing transition in the “incomplete” LTS. The missing transitions in the incomplete LTS represent behavior of the environment that is never exercised by the open system under analysis. As a result, no assumptions need to be made about these behaviors. The sink state reflects exactly this fact, since it poses no implementation restrictions to the environment.

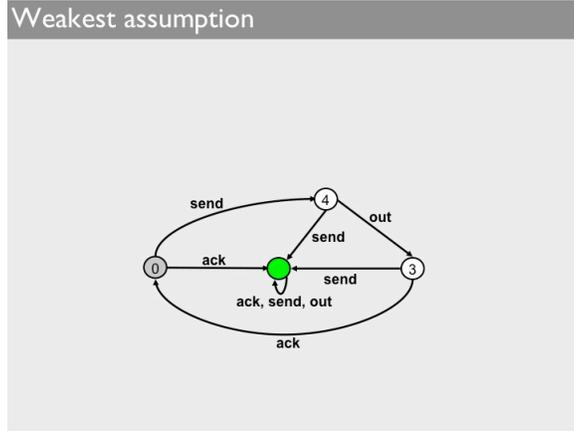


Fig. 8. Computed Assumption

The result of subset construction and completion for our running example is shown in Figure 8. The sink state is colored green.

Once we have the error LTS, we obtain the assumption by deleting the error state and the transitions that lead to it.

The assumption for the running example is depicted in Figure 8. The assumption expresses the fact that actions `send`, `out`, `ack` should happen in this order (and this is in fact the encoding of component `Output`); in addition, the assumption allows extra behaviors (the ones that lead to the sink state). It can be shown that indeed this assumption is the “weakest”.

4.2 Learning Component Interfaces

Let $M = \langle Q_M, \alpha M, \delta_M, q_{0M} \rangle$ be a component, and $\Sigma \subseteq \alpha M$ denote the communication alphabet of component M , i.e., the set of actions through which M communicates with its environment. Our goal is to compute M 's precise interface as a finite state automaton A over Σ , in other words an interface A that is both *safe* and *permissive*, as defined in Section 3.

Since A represents a regular language, we can use the learning algorithm L^* to learn it. To this aim, we need to provide L^* with a teacher that represents the language of A . As discussed in the following, the teacher can be implemented using model checking, since all questions asked by L^* can be reduced to reachability problems (see Figure 9).

Queries L^* is first used to repeatedly *query* M to check whether, in the context of strings s , M reaches an error state. If it does, then s corresponds to an illegal execution of M and should be excluded from A and the query returns *false*. Otherwise, s should be included in A , and the query returns *true*. If error states are introduced by some property P , then the query corresponds to checking the

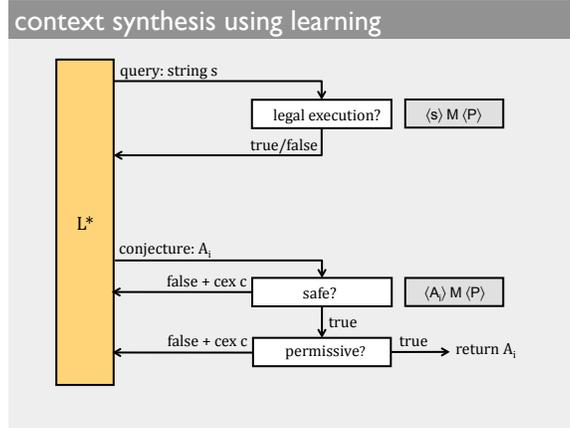


Fig. 9. Learning for Context Synthesis

triple $\langle s \rangle M \langle P \rangle$ as illustrated in Figure 9 (we abuse notation here, and use s to represent $ts(s)$).

Conjectures The conjectured automaton A is checked for correctness, which in this context means checking whether the interface that it represents is safe and permissive. We therefore break down answering conjectures into two parts:

Oracle 1 checks if A is *safe*, using the model checking procedure described in Section 3. Again, if error states are introduced by some property P , the safety checks corresponds to checking the triple $\langle A \rangle M \langle P \rangle$, as illustrated in Figure 9. If A is safe, then the teacher proceeds to Oracle 2. If it is unsafe, the model checker returns a counterexample t . The resulting counterexample t , projected on the interface alphabet Σ , is returned to L^* to refine its conjecture (see Figure 9, where $c = t \upharpoonright \Sigma$). The projection is necessary because L^* needs counterexamples in terms of the alphabet over which it is learning.

Oracle 2 checks if safe interface A is also permissive, using the model checking procedure described in Section 3. If the interface is permissive, then the framework terminates with A as a safe, permissive and minimal interface for M (minimality is guaranteed by the characteristics of the L^* algorithm). If, on the other hand, a counterexample t is returned, then this may be because the interface needs to be refined, or it could be because the permissiveness procedure is not precise in the presence of non-determinism. As discussed above, one could determinize component M for performing this check. Other, more light-weight approaches propose heuristics.

For example, one such heuristic consists of making a *query* on $c = t \uparrow \Sigma$ (with the same mechanism as L^* queries are answered). If the query returns true, then it means the interface is not permissive, and therefore c is returned to L^* for refinement, and the learning process continues with more queries and eventually with a new conjecture (see Figure 9).

If the query returns false, then c does not correspond to a real counterexample. Model checking therefore ignores this state. Several approaches have been proposed at this point. One approach applies an additional heuristic step [1], whereas another backtracks after the spurious counterexample and continues the state space exploration [13]. The latter approach is illustrated in Figure 10. This heuristic is non-trivial to implement within a model checker. The reason is that the permissiveness check consists of a reachability check within which a query is invoked to potentially invalidate a discovered counterexample, in which case the reachability check backtracks and continues the search. In essence, querying within a reachability check would naively mean that a model checker is invoked within a model checker, which is clearly inefficient.

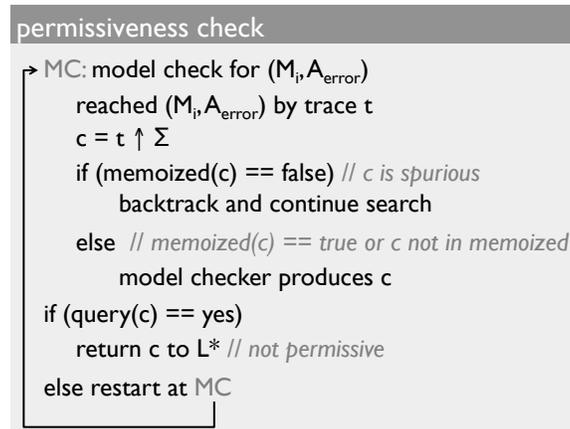


Fig. 10. Permissiveness heuristic

For this reason, all query results are stored in a memoized table which is consulted during reachability analysis. If a potential counterexample discovered is stored in the memoized table as a spurious one, then the algorithm backtracks and tries a different path. If it is a real counterexample, it is returned to L^* . If it is not stored in the table, the reachability check terminates. A query then follows as an independent step, and the reachability check is started from scratch. Since the result of the query is stored in the memoized table, the reachability check is guaranteed to return a different counterexample in the next round.

Non-determinism. In summary, unless component M is deterministic with respect to the alphabet Σ of the assumption, precise interfaces can only be computed through determinization of M , which may result in a component exponentially larger than M . Heuristics can be quite efficient at getting a precise interface, but cannot provide guarantees. Of course, if the permissiveness check does not encounter spurious counterexamples, then we know that the resulting interface is precise, despite heuristics.

There are two cases where determinization can be avoided, as will be described in the following sections. First, when the potential source for non-determinism is abstraction of a deterministic infinite state component, then we may use a combination of over- and under- approximations to precisely compute a component interface while avoiding determinization in the permissiveness step. Second, when the environment of a component is available, determinization can be avoided during compositional reasoning. Since in this context, rather than a precise interface for the component irrespective of environment, we just need the interface to act as an assumption for an assume-guarantee rule, the environment can be used to selectively increase the interface permissiveness. We discuss these cases in the following sections.

5 Interface Generation and Abstraction

The learning frameworks that we discussed in the previous section only apply to finite state components since they rely on teachers that exhaustively explore the component state space. However, most realistic components are infinite state for all practical purposes. A typical approach for dealing with large components in model checking is by using abstraction techniques. In this section, we discuss a framework that computes interfaces of potentially infinite-state components by combining abstraction and learning approaches.

There are two types of abstractions that one may build of a component. An over-approximation (“may” abstraction) is an abstraction that contains a superset of the behaviors of the component. The advantage of over-approximations is that, when used for checking properties, if the property is satisfied for the overapproximation, then it is also satisfied for the concrete component. The disadvantage is that when a counterexample is obtained, it may be a spurious one, since it may correspond to a behavior that is not really feasible in the concrete component. These characteristics are reversed for under-approximations. An under-approximation (“must” abstraction) contains only a subset of the behaviors of the concrete component. As such, it will only return real counterexamples. In the absence of errors, however, there is not guarantee that the concrete component is also error-free since there may be concrete behaviors that are not accounted for in the under-approximation.

The may and must abstractions that we use [26] are obtained using predicate abstraction. Predicate abstraction is a technique that substitutes component variables on large or infinite domains with a finite set of predicates over these variables. Concrete states of the component are then substituted with abstract

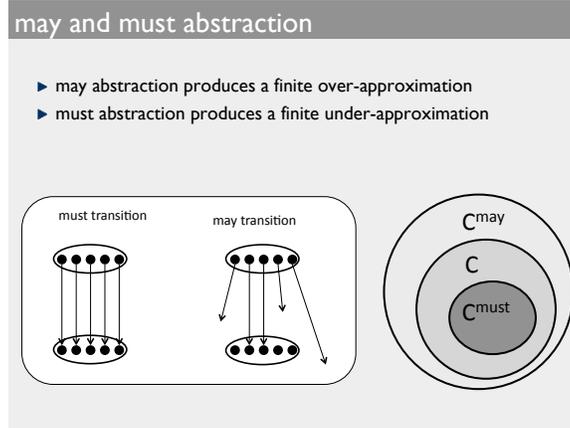


Fig. 11. May and Must Abstraction

states representing valuations of the selected predicates. In a may abstraction, an abstract transition links two abstract states if there exists a concrete transition between concrete states represented by the two abstract states. May transitions between abstract states may or may not correspond to actual transitions in the concrete system. A may abstraction is an over-approximation (see Figure 11). On the other hand, in a must abstraction, abstract transitions link two abstract states only if all the concrete states represented by the two abstract states are linked by concrete transitions. Must transitions are guaranteed to represent transitions in the concrete system, but do not necessarily cover all concrete transitions. A must abstraction is therefore an under-approximation of the concrete component (see Figure 11).

Let C be a component corresponding to a potentially infinite-state transition system S_C . From now on, for simplicity, we will use C to represent the component and its transition system. We have proposed interface-generation algorithms that operate by analyzing *finite-state* abstractions of C [26]. The essence of our approach lies in the following observation:

Theorem 1. *Assume a component C , a may abstraction C^{may} and a must abstraction C^{must} for C . If an interface A for C is permissive with respect to C^{must} and safe with respect to C^{may} , then A is safe and permissive with respect to C .*

To provide an intuition for this theorem, let us analyze the relationships between the languages corresponding to may and must abstractions. For any component C , since C^{may} has more behaviors than C , it follows that $\mathcal{L}_{illegal}(C) \subseteq \mathcal{L}_{illegal}(C^{may})$, and consequently, $\mathcal{L}_{legal}(C) \supseteq \mathcal{L}_{legal}(C^{may})$. On the other hand, $\mathcal{L}_{illegal}(C) \supseteq \mathcal{L}_{illegal}(C^{must})$, and consequently, $\mathcal{L}_{legal}(C) \subseteq \mathcal{L}_{legal}(C^{must})$. If an interface A is safe with respect to C^{may} , it means that its legal executions are

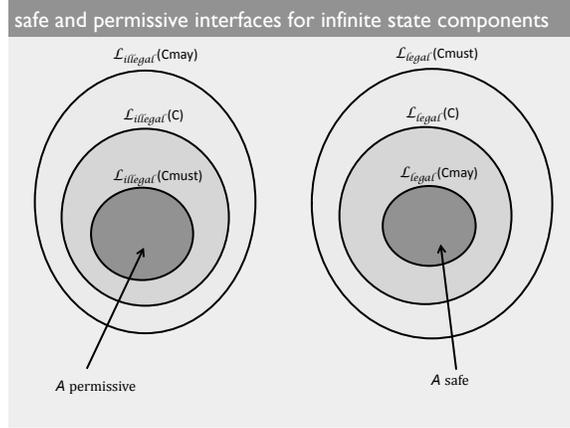


Fig. 12. Relationship between languages of abstractions of a concrete component and the component itself

a subset of the legal executions of C^{may} . Similarly, if A is permissive, its illegal executions are a subset of the illegal executions of C^{must} . These relationships are illustrated in Figure 12. Given the complementary nature of the legal and illegal execution sets of any component, an interface can only have both properties if $\mathcal{L}_{illegal}(C^{must}) = \mathcal{L}_{illegal}(C) = \mathcal{L}_{illegal}(C^{may}) = \mathcal{L}_{illegal}(A)$.

Our approach for interface generation is therefore based on constructing may and must abstractions for a concrete component C (C^{may} and C^{must} , respectively). Its novelty with respect to previous work is that it uses C^{may} to check whether an interface is safe, and C^{must} to check whether an interface is permissive. The advantage of the approach is that, if the concrete component is deterministic, then so is C^{must} , since it under-approximates the concrete behavior. By using C^{must} for the permissiveness check, we therefore avoid determinizing the abstractions that are constructed, while still providing guarantees for safety and permissiveness of the computed interface.

5.1 Learning interfaces using abstractions

In order to compute an interface for component C , we use the learning framework presented in the previous section. The teacher is very similar, except that it sometimes needs to trigger a refinement of the abstraction, in order to provide answers to the learner. Refinement consists of adding predicates, and it is performed on demand, within the teacher's mechanism for answering the L^* questions. More specifically, the teacher operates as follows.

Queries. The procedure for queries is illustrated in Figure 13. It first checks whether the word σ triggers an illegal execution in C^{must} . If it does, σ should not

```
Query( $\sigma$ , C)

1. if checkSafe( $\sigma$ , Cmust) != null
2.   return "false"
3. cex = checkSafe( $\sigma$ , Cmay)
4. if cex == null
5.   return "true"
6. Preds = Preds U Refine(cex)
7. Query( $\sigma$ , C)
```

Fig. 13. Answering queries

```
conjecture : Oracle 1

1. cex = checkSafe(A, Cmay)
2. if cex == null
3.   invoke Oracle2
4. If Query(cex, C) == "false"
5.   return cex to L*
6. else
7.   goto 1
```

Fig. 14. Answering conjectures: Oracle 1

```

conjecture : Oracle 2

1. cex = checkPermissive(A, Cmust)
2. if cex == null
3.     return A
4. If Query(cex, C) == "true"
5.     return cex to L*
6. else
7.     goto 1

```

Fig. 15. Answering Conjectures: Oracle 2

belong to A because it must also trigger an illegal execution in C . So the query returns *false*. Otherwise, σ is checked against C^{may} . If it is safe for C^{may} , then σ must belong to A so the query returns *true*. Otherwise, we have a situation where σ is safe for C^{must} and unsafe for C^{may} . In other words, σ demonstrates that the illegal languages of C^{may} and C^{must} are not equal. As discussed earlier in the section, we are able to compute an interface when the illegal languages of C^{may} and C^{must} become equal. We therefore need to refine the abstraction, and check the query again. L^* is not involved in the refinement or restarted after it; it just awaits for the teacher to come up with a response to the query. The response is always consistent with the concrete component C .

Conjectures. We use Theorem 1 to answer the conjectures using two oracles, as illustrated in Figure 14 and Figure 15.

Oracle 1 is invoked first. If it finds that A is safe with respect to C^{may} , Oracle 2 gets invoked. If Oracle 2 finds that A is also permissive with respect to C^{must} , we conclude from Theorem 1 that A is a safe and permissive interface for C . All remaining cases require either the refinement of A by L^* , or the refinement of the component abstractions. We use queries to help us determine what needs to be refined. Our approach is described in detail below.

Oracle 1: If A is not safe with respect to C^{may} , we obtain a counterexample cex , which is allowed by A but leads to error in C^{may} . We subsequently query cex in order to determine whether it is indeed a counterexample to the safety of A . Note that the querying procedure may involve refinement of the abstraction. If the query returns no, then it means that cex should indeed not be in the language of A , so cex is returned to L^* for A to be refined. Otherwise, we invoke Oracle

1 again, knowing that Predshave been updated because abstraction refinement must have occurred.

Oracle 2: If A is not permissive with respect to C^{must} , we obtain a counterexample cex , which corresponds to a word that is not allowed by A . We subsequently query cex in order to determine whether it is indeed a counterexample to the permissiveness of A . If the query response is positive, then cex should belong to A , so cex is returned to L^* for refining the assumption. Note again that querying may involve refinement. If the response is negative, then the permissiveness check is invoked again, because we know there must have been abstraction refinement involved.

More details and explanations are provided in [26].

5.2 Applicability and Related Approaches

The learning scheme presented in this section for computing interfaces of infinite state components generates deterministic finite state automata. As such, its applicability is restricted to interfaces that can be represented in this fashion. The framework that we have developed may not always terminate, which is always a possibility in abstraction refinement schemes. However, if the concrete component C has a finite bisimulation quotient, then our framework is guaranteed to terminate and produce a minimal safe and permissive interface for C [26].

Other related approaches to interface generation for infinite components have been presented in [1, 17]. Both approaches construct only over-approximations of the component behavior, which may be non-deterministic. As mentioned, checking permissiveness when (abstracted) components are non-deterministic requires a potentially expensive determinization step. Alur et al. [1] avoid this step by using heuristics, and therefore cannot guarantee permissiveness of the generated interfaces. On the other hand, Henzinger et al. [17] build “abstract regions”, which is equivalent to performing a determinization step. Furthermore, the abstraction mechanisms in [17] cannot guarantee minimal interfaces. Even if these interfaces were to be minimized, this approach would suffer from potentially large intermediate interfaces that subsequently get compacted. This latter problem is more pronounced in the presence of the determinization step, which is exponential, in the worst case. In contrast, L^* -based approaches like ours and [1] directly generate minimal interfaces. Note however that the technique by [1] does not provide criteria to automatically detect the need for abstraction refinement. Their refinements are based on inspection of the generated interfaces, and are performed manually. In contrast, refinement in our work [26] is performed automatically.

6 Assumption Generation for Compositional Verification

As discussed in Section 2, assume guarantee reasoning provides solutions to the problem of decomposing the verification of a large system into local verification

steps of the system components. The most challenging part of applying assume-guarantee reasoning, however, is coming up with appropriate assumptions to use in the application of the assume-guarantee rules. In this section, we discuss work on generating assumptions for automated assume-guarantee verification.

We will restrict ourselves to the simple rule presented in Section 2, and will then discuss how one can expand to other rules.

As discussed earlier in this paper, the weakest assumption captures precisely all restrictions that a component needs to make on its environment in order to satisfy some safety property(ies). The weakest assumption can safely be used for assume-guarantee reasoning; in fact, with the weakest assumption, the rule also becomes complete since, the second premise holds ($\langle true \rangle M_2 \langle A \rangle$) if and only if the conclusion of the rule holds ($\langle true \rangle M_1 \parallel M_2 \langle P \rangle$).

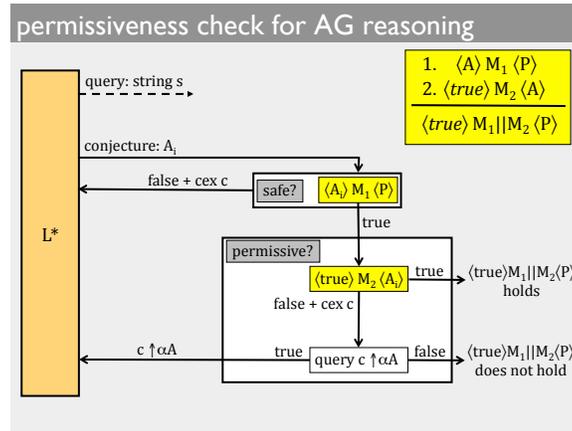


Fig. 16. Learning Assumptions for Assume-Guarantee Reasoning

The weakest assumption corresponds to a safe and permissive interface for component M_1 . We could therefore use L^* to learn this assumption while automatically verifying a property on some system in an assume guarantee style. The framework of Figure 16 demonstrates the steps involved in performing automated assume-guarantee reasoning while learning the weakest assumption. Queries are answered in the exact same fashion as in the interface generation framework of Figure 9. The first Oracle, checking whether the conjecture corresponds to a safe interface for M_1 , is answered identically to the interface generation framework. Note that checking for safety corresponds to checking the first premise of the assume-guarantee Rule ASYM.

In terms of the permissiveness check, we can now take advantage of the fact that M_2 is available, to avoid determinization of M_1 . Remember that our main target in this framework is to prove or disprove a property on the system

using assume-guarantee reasoning. Since Oracle 1 checks that premise 1 of Rule ASYM holds, it remains to check premise 2 ($\langle true \rangle M_2 \langle A \rangle$). Premise 2 therefore substitutes Oracle 2 of the original interface generation framework. If this check passes, then we know that both premises of Rule ASYM hold, and therefore the property holds for $M_1 \parallel M_2$.

If the check fails, the Teacher performs some analysis to determine the underlying reason (see Figure 16). The Teacher performs a query (of the L^* type) in order to determine whether the returned counterexample cex , projected to the alphabet of the assumption, should belong to the conjectured assumption A . If the answer is true, meaning that $c \uparrow \alpha A$ should be included in A , then it means that A is not the weakest assumption since it does not include a safe word, and $c \uparrow \alpha A$ is returned to L^* for refinement of A . 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 .

Notice that the answers that this modified Teacher provides to L^* are always with respect to the weakest assumption. However, the framework uses M_2 to filter which missing words to include in the language of the assumption, as opposed to adding all of them. The reason is that we restrict our reasoning to a specific context, rather than accounting for all possible contexts. As a result, we no longer require determinization of component M_1 .

Note also that we do not always obtain the weakest assumption from this framework; in other words, the obtained assumption is not the most permissive. Our 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 assumptions. At that point, we may, or may not have reached the weakest assumption. We will however have reached an assumption that completes our verification; this assumption is smaller than or equal to the weakest assumption, as guaranteed by the characteristics of L^* . For our running example, the assumption generated is smaller than the weakest assumption, as illustrated in Figure 17. The second conjecture, A_2 , generated by L^* , passes both Oracles and the learning framework terminates reporting that the property holds; notice that A_2 has only two states as compared to the weakest assumption that has four states (see Figure 8).

Given the fact that our Teacher only comes back to L^* for refinement with counterexamples for the weakest assumption, the framework will eventually converge to the weakest assumption unless it terminates earlier. We have shown [23] that with the weakest assumption, the rule becomes sound and complete, and therefore our framework will return a conclusive answer at that iteration. As a result, the framework always terminates.

To summarize, we presented a framework that computes an assumption for automated assume-guarantee reasoning. We cannot tell if the framework computes the weakest assumption, but we know that it will do so if necessary, and thus guarantees termination.

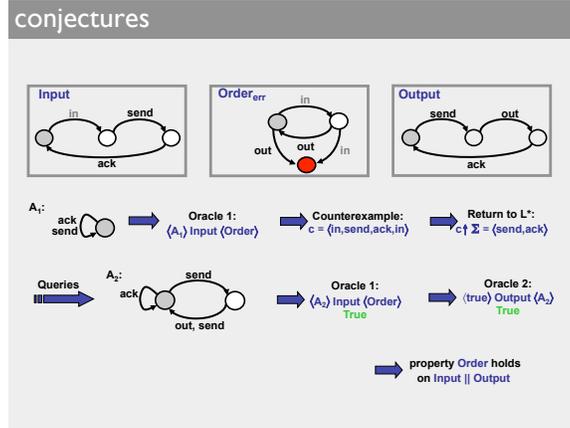


Fig. 17. Assumption for AG reasoning

6.1 Related Approaches

We have showed how to guide our learning of assumptions for compositional verification towards the weakest assumption. Other researchers focused on the more computationally expensive problem of learning a *minimal* assumption [16, 8] for compositional verification. In other words, computing an assumption A_{min} such that any other assumption A that can check satisfaction or violation of P will have a greater than or equal number of states, i.e., $|A| \geq |A_{min}|$.

The alphabet of the assumptions we learn for compositional verification is fixed to $(\alpha M_1 \cup \alpha P) \cap \alpha M_2$. Other researchers and ourselves observed that it may sometimes be possible to verify a problem with a smaller alphabet, and therefore potentially smaller assumptions [23, 6]. Learning assumptions can be extended for other assume-guarantee rules that are symmetric and may involve circularity, and may involve multiple components [23]. Rule ASYM itself can be extended to multiple components through recursive invocation. Learning has also been applied in the context of symbolic and implicit model checking [21, 7], and of assume-guarantee reasoning for liveness properties [11].

7 Discussion and Conclusions

In this tutorial paper, we reviewed several approaches for context synthesis. Our context synthesis techniques rely on standard model checking features, such as reachability analysis and counterexample generation. Over the years, we have implemented our techniques on top of several well known model-checkers, such as LTSA [14], SPIN [22], Java PathFinder [13], and ARMC [26]. We have experimented with compositional verification and interface generation techniques in

the context of several applications, mostly involving NASA systems. Our NASA case studies include a Rover Executive [10, 15, 4], autonomous rendez vous and docking [5], a resource arbiter for the Mars Exploration Rover [12], and models of the flight phases of a spacecraft [13, 26]. We have also experimented with existing benchmarks for compositional verification [12] and for interface generation [26].

In our experience, learning-based interface generation and compositional verification were most successful when a system has a well-designed component-based structure, where component interfaces are small. Beyer, Henzinger and Singh make a similar observation [3]. Moreover, even though abstraction can be introduced in order to deal with large component implementations, it is still much harder to generate interfaces at the level of source code. Interface are ideally generated at design time, and are then used in several ways throughout the life cycle of a component: for compositional verification, concrete component and system integration testing, runtime verification, and incremental verification in the presence of component upgrades or substitutions.

Our learning based algorithms for context synthesis are part of the open-source Java PathFinder tool-set and they are available from the following website: <http://babelfish.arc.nasa.gov/trac/jpf/>, the `jpf-cv` project.

Our work on interface generation needs to be extended and matured in order to make it applicable in practice. There are several interesting future research directions. Some of them involve the generation of interfaces that go beyond purely functional properties such as safety and liveness, but potentially timed or probabilistic properties. Moreover, it would be interesting to try and identify design decisions that facilitate the generation of component interfaces. Finally, one could investigate interfaces in different domains such as service-oriented systems, aerospace systems, and others.

References

1. ALUR, R., CERNÝ, P., MADHUSUDAN, P., AND NAM, W. Synthesis of interface specifications for Java classes. In *POPL* (2005), J. Palsberg and M. Abadi, Eds., ACM, pp. 98–109.
2. ANGLUIN, D. Learning regular sets from queries and counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106.
3. BEYER, D., HENZINGER, T. A., AND SINGH, V. Algorithms for interface synthesis. In *CAV* (2007), W. Damm and H. Hermanns, Eds., vol. 4590 of *Lecture Notes in Computer Science*, Springer, pp. 4–19.
4. BLUNDELL, C., GIANNAKOPOULOU, D., AND PASAREANU, C. S. Assume-guarantee testing. *ACM SIGSOFT Software Engineering Notes* 31, 2 (2006).
5. BRAT, G., DENNEY, E., GIANNAKOPOULOU, D., AND JONSSON, A. Verification of autonomous systems for space applications. In *IEEE Aerospace Conference* (2006).
6. CHAKI, S., AND STRICHMAN, O. Three optimizations for assume-guarantee reasoning with L*. *Formal Methods in System Design* 32, 3 (2008), 267–284.
7. CHEN, Y.-F., CLARKE, E. M., FARZAN, A., TSAI, M.-H., TSAY, Y.-K., AND WANG, B.-Y. Automated assume-guarantee reasoning through implicit learning. In *CAV* (2010), pp. 511–526.

8. CHEN, Y.-F., FARZAN, A., CLARKE, E. M., TSAY, Y.-K., AND WANG, B.-Y. Learning minimal separating DFAs for compositional verification. In *TACAS* (2009), pp. 31–45.
9. CLARKE, E., GRUMBERG, O., AND PELED, D. A. *Model Checking*. The MIT Press, 1999.
10. COBLEIGH, J. M., GIANNAKOPOULOU, D., AND PASAREANU, C. S. Learning assumptions for compositional verification. In *TACAS* (2003), pp. 331–346.
11. FARZAN, A., CHEN, Y.-F., CLARKE, E. M., TSAY, Y.-K., AND WANG, B.-Y. Extending automated compositional verification to the full class of omega-regular languages. In *TACAS* (2008), C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*, Springer, pp. 2–17.
12. GHEORGHIU, M., GIANNAKOPOULOU, D., AND PASAREANU, C. S. Refining interface alphabets for compositional verification. In *TACAS* (2007), pp. 292–307.
13. GIANNAKOPOULOU, D., AND PASAREANU, C. S. Interface generation and compositional verification in JavaPathfinder. In *FASE* (2009), M. Chechik and M. Wirsing, Eds., vol. 5503 of *Lecture Notes in Computer Science*, Springer, pp. 94–108.
14. GIANNAKOPOULOU, D., PASAREANU, C. S., AND BARRINGER, H. Component verification with automatically generated assumptions. *Autom. Softw. Eng.* 12, 3 (2005), 297–320.
15. GIANNAKOPOULOU, D., PASAREANU, C. S., AND COBLEIGH, J. M. Assume-guarantee verification of source code with design-level assumptions. In *ICSE* (2004), pp. 211–220.
16. GUPTA, A., McMILLAN, K. L., AND FU, Z. Automated assumption generation for compositional verification. *Formal Methods in System Design* 32, 3 (2008), 285–301.
17. HENZINGER, T. A., JHALA, R., AND MAJUMDAR, R. Permissive interfaces. *SIGSOFT Softw. Eng. Notes* 30 (September 2005), 31–40.
18. HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1979.
19. JONES, C. B. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 596–619.
20. MAGEE, J., AND KRAMER, J. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.
21. NAM, W., MADHUSUDAN, P., AND ALUR, R. Automatic symbolic compositional verification by learning assumptions. *Formal Methods in System Design* 32, 3 (2008), 207–234.
22. PASAREANU, C. S., AND GIANNAKOPOULOU, D. Towards a compositional SPIN. In *SPIN* (2006), A. Valmari, Ed., vol. 3925 of *Lecture Notes in Computer Science*, Springer, pp. 234–251.
23. PASAREANU, C. S., GIANNAKOPOULOU, D., BOBARU, M. G., COBLEIGH, J. M., AND BARRINGER, H. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design* 32, 3 (2008), 175–205.
24. PNUELI, A. *In transition from global to modular temporal reasoning about programs*. Springer-Verlag New York, Inc., New York, NY, USA, 1985, pp. 123–144.
25. RIVEST, R. L., AND SCHAPIRE, R. E. Inference of finite automata using homing sequences. *Inf. Comput.* 103, 2 (1993), 299–347.
26. SINGH, R., GIANNAKOPOULOU, D., AND PASAREANU, C. S. Learning component interfaces with may and must abstractions. In *CAV* (2010), pp. 527–542.