Component Verification with Automatically Generated Assumptions *

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

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

Howard Barringer
Department of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, UK (howard@cs.man.ac.uk)

October 13, 2003

Abstract. Model checking is an automated technique that can be used to determine whether a system satisfies certain required properties. The typical approach to verifying properties of software components is to check them for all possible environments. In reality, however, a component is only required to satisfy properties in specific environments. Unless these environments are formally characterized and used during verification (assume-guarantee paradigm), the results returned by verification can be overly pessimistic. This work introduces an approach that brings a new dimension to model checking of software components. When checking a component against a property, our modified model checking algorithms return one of the following three results: the component satisfies a property for any environment; the component violates the property for any environment; or finally, our algorithms generate an assumption that characterizes exactly those environments in which the component satisfies its required property. Our approach has been implemented in the LTSA tool and has been applied to the analysis of two NASA applications.

Keywords: assume-guarantee reasoning, model checking, component verification

1. Introduction

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

Model checking is an automated verification technique that can be used to determine whether a concurrent system satisfies certain properties by exhaustively exploring all its possible executions. Software model checking is typically applied to *components* of a larger system for

* This paper is an expanded version of (Giannakopoulou et al., 2002).

several reasons. For example: a software component may be embedded as is the case for autonomous software; one would typically ignore the details of the operating system in which a component operates; a system may be partially specified; finally, given the fact that the state explosion problem (Clarke et al., 1999) is particularly acute in software systems, one realistically needs to "divide and conquer", that is, to break up the verification task into smaller tasks.

In order to model check a component in isolation one needs to incorporate a model of the environment interacting with the component. By default, this is the "most general environment", an environment that can invoke, in any order, any action of the interface between the two, or that may refuse any service that the component requires. We believe that the above approach to component checking is overly *pessimistic*; the underlying assumption is that the environment is free to behave as it pleases, and that the component will satisfy the required property for any environment. A similar observation is made by de Alfaro and Henzinger in the context of interface compatibility checking (de Alfaro and Henzinger, 2001a; de Alfaro and Henzinger, 2001b).

In the world of model checking, this problem has given rise to the assume-guarantee style of reasoning (Jones, 1983; Pnueli, 1984), where the model of the environment is restricted by assumptions provided by the developer. Assume-guarantee reasoning first checks whether a component guarantees a property, when it is part of a system that satisfies an assumption. Intuitively, the assumption characterizes all contexts in which the component is expected to operate correctly. To complete the proof, it must also be shown that the remaining components in the system, i.e., the environment, satisfy the assumption. This style of reasoning is typically performed in an interactive fashion. Developers first check a component with the most general environment. If a counterexample is returned that is unrealistic for the system under analysis, they make several attempts at defining an assumption that is strong enough to eliminate false violations, but that also reflects appropriately the remaining system.

In this paper we propose and describe a novel framework for model checking of components that provides more useful user feedback than the usual counter-example generation for property violations. When model checking a component against a property, our algorithms return 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) an automatically generated assumption that characterizes exactly those environments in which the component satisfies the property.

Let us illustrate this with a small example. A multi-threaded component uses a mutex to coordinate accesses to a shared variable, which may also be accessed by the environment. The requirement is that race violations should not occur in the system. If some thread within the component performs unprotected accesses to the variable, the requirement may be violated irrespective of the environment. Our approach reports this fact, together with a counterexample illustrating it. Now assume that all accesses to the variable within the component are protected by the mutex. Model checking under the most general environment would return a violation. Our algorithms would return an assumption, reflecting the fact that all accesses to the shared variable by the environment must similarly be protected by the lock.

In fact, our approach generates the *weakest* environment assumption that enables the property to hold. Therefore, in selecting an appropriate environment for a component, one can safely reject any environment that does not satisfy the assumption generated. Assumption generation may also be seen as a way of providing extra automated support for assume-guarantee reasoning. Finally, for systems like the ones we study, the environment is often unpredictable. Some assumptions are typically made about it, but loss of mission must be avoided even if the environment falls outside these assumptions. For such cases, assumptions can be used as runtime monitors of the actual environment (Havelund and Rosu, 2001). Monitors can generate appropriate warnings when the environment falls outside expected behavior and trigger special system behavior, if necessary.

We have implemented our approach in the Labeled Transition Systems Analyzer (LTSA) tool (Magee et al., 1999; Magee and Kramer, 1999), which provides good support for incremental system design and verification. It implements such features as component abstraction and minimization that make the integration of our approach straightforward.

The problem of assumption generation can be associated with such problems as submodule construction, controller synthesis and model matching. To our knowledge, such work has not been directly applied to model checking before; the relation of our approach with these domains is further discussed in Section 6. The remainder of the paper is organized as follows. Section 2 briefly discusses the LTSA tool and the underlying theory that is used by our approach. It is followed by the presentation of our approach in Section 3. Section 4 describes our experience with analyzing the Executive modules of two autonomous systems developed at NASA Ames. We discuss the applicability of our approach in practice and extensions that we are considering in Section

5. Finally, Section 6 presents related work, and Section 7 concludes the paper.

2. Background

In this section, we describe the LTSA framework in which our approach has been introduced. We provide formal definitions for those aspects of the tool that we have used and/or modified.

2.1. The LTSA Tool

The LTSA (Magee et al., 1999; Magee and Kramer, 1999) is an automated tool that supports Compositional Reachability Analysis (CRA) of a software system based on its architecture. In general, the software architecture of a concurrent system has a hierarchical structure (Magee et al., 1994). CRA incrementally computes and abstracts the behavior of composite components based on the behavior of their immediate children in the hierarchy. Abstraction consists of hiding the actions that do not belong to the interface of a component, and minimizing with respect to observational equivalence (Giannakopoulou et al., 1999).

The input language "FSP" of the tool is a process-algebra style notation with Labeled Transition Systems (LTS) semantics. A property is also expressed as an LTS, but with extended semantics, and is treated as an ordinary component during composition. Properties are combined with the components to which they refer. They do not interfere with system behavior, unless they are violated. In the presence of violations, the properties introduced may reduce the state space of the (sub)systems analyzed.

As in our approach, the LTSA framework treats components as open systems that may only satisfy some requirements in specific contexts. By composing components with their properties, it postpones analysis until the system is closed, meaning that all contextual behavior that is applicable has been provided. We extend this framework by performing useful analysis at the component level.

The LTSA tool also features graphical display of LTSs, interactive simulation and graphical animation of behavior models (Magee et al., 2000), all helpful aids in both design and verification of system models.

2.2. Program Model

We use labeled transition systems (LTSs) to model the behavior of communicating components in a concurrent system. Let $\mathcal{A}ct$ be the universal set of observable actions, and $\mathcal{A}ct_\tau = \mathcal{A}ct \cup \{\tau\}$, where τ

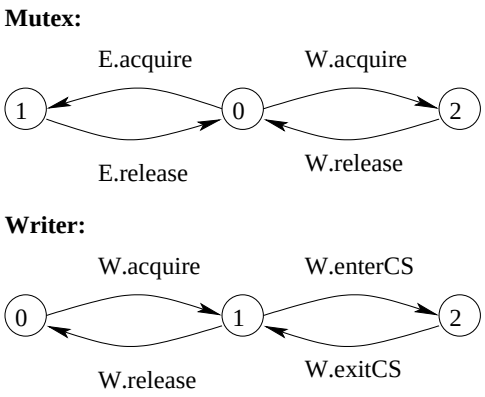

Figure 1. LTSs for a Mutex and a Writer

denotes a local action *unobservable* to a component's environment. A *labeled transition system* T is a quadruple $\langle S, \alpha T, R, s_0 \rangle$, where S is a set of states, $\alpha T \subseteq \mathcal{A}ct$ is a set of actions called the *alphabet* of T, $R \subseteq S \times \alpha T \cup \{\tau\} \times S$ is a transition relation and $s_0 \in S$ is the initial state. We use π to denote a special error state, which models the fact that a safety violation has occurred in the associated system, and then use Π to denote the LTS $\langle \{\pi\}, \mathcal{A}ct, \emptyset, \pi \rangle$.

For example, Figure 1 illustrates LTSs for a *Writer* component and a *Mutex*. In all illustrations of LTSs in this paper, state 0 is the initial state. The *Writer* acquires the mutex (action *W.acquire*), enters and subsequently exits a critical section (*W.enterCS, W.exitCS*) used to model the fact that the *Writer* updates some shared variable, and then releases the mutex *W.release*. The *Mutex* component can be acquired and released by the *Writer* (*W.acquire, W.release*) or its environment (*E.acquire, E.release*), but only a single component can hold it at any one time.

We call an LTS well formed if the error state π has no outgoing transitions: by construction, we only consider well formed LTSs in this work. An LTS $T = \langle S, \alpha T, R, s_0 \rangle$ is *non-deterministic* if $\exists (s, a, s'), (s, a, s'') \in R$ such that $s' \neq s''$ (otherwise T is *deterministic*).

A *trace* σ of an LTS T is a sequence of observable actions that T can perform starting at its initial state. For example, $<$*W.acquire*$>$ and $<$*W.acquire, W.enterCS, W.exitCS*$>$ are both traces of the *Writer* component of Figure 1. The set of traces of T is denoted as $Tr(T)$. For $\mathcal{A} \subseteq \mathcal{A}ct$, we use $\sigma \upharpoonright \mathcal{A}$ to denote the trace obtained by removing from σ all occurrences of actions $a \notin \mathcal{A}$. We denote as $errTr(T)$ the set of traces that may lead to state π, which we call *error traces* of T.

2.2.1. *Operators*
In the following, we provide semantics for the operators defined on LTSs that are used in our work. Although we provide transitional semantics in a typical process algebra style, our aim here is not to define an algebra.

Let $T = \langle S, \alpha T, R, s_0 \rangle$ and $T' = \langle S', \alpha T', R', s'_0 \rangle$. We say that T *transits* into T' with action a, denoted $T \xrightarrow{a} T'$, iff $(s_0, a, s'_0) \in R$ and: either $\alpha T = \alpha T'$ and $R = R'$ for $s'_0 \neq \pi$, or, in the special case where $s'_0 = \pi$, $T' = \Pi$.

The *interface* operator $\uparrow$ is used to make unobservable those actions in the LTS of a component that are not part of its interface. Formally, given an LTS T and a set of observable actions $\mathcal{A} \subseteq \mathcal{A}ct$, $T \uparrow \mathcal{A}$ is defined as follows. For $T = \Pi$, $T \uparrow \mathcal{A} = \Pi$. For $T \neq \Pi$, $T \uparrow \mathcal{A}$ is an LTS with the same set of states and initial state as T. The alphabet of $T \uparrow \mathcal{A}$ is $\alpha T \cap \mathcal{A}$, and its transition relation is described by the following rules:

$$\frac{T \xrightarrow{a} T',\ a \in \mathcal{A}}{T \uparrow \mathcal{A} \xrightarrow{a} T' \uparrow \mathcal{A}} \qquad \frac{T \xrightarrow{a} T',\ a \notin \mathcal{A}}{T \uparrow \mathcal{A} \xrightarrow{\tau} T' \uparrow \mathcal{A}}$$

Parallel composition "||" is a *commutative* and *associative* operator that combines the behavior of two components by synchronization of the actions common to their alphabets and interleaving of the remaining actions. For example, in computing the parallel composition of components *Writer* and *Mutex* of Figure 1, actions *W.acquire* and *W.release* will each be synchronized.

Formally, let $T_1 = \langle S^1, \alpha T_1, R^1, s_0^1 \rangle$ and $T_2 = \langle S^2, \alpha T_2, R^2, s_0^2 \rangle$ be two LTSs. If either $T_1 = \Pi$ or $T_2 = \Pi$, then $T_1 || T_2 = \Pi$. Otherwise, $T_1 || T_2$ is an LTS $T = \langle S, \alpha T, R, s_0 \rangle$, where $S = S^1 \times S^2$, $s_0 = (s_0^1, s_0^2)$, $\alpha T = \alpha T_1 \cup \alpha T_2$, and R is defined as follows, where a is an observable action or τ (the symmetric rules are implied since the operator is commutative):

$$\frac{T_1 \xrightarrow{a} T'_1,\ a \notin \alpha T_2}{T_1 || T_2 \xrightarrow{a} T'_1 || T_2} \qquad \frac{T_1 \xrightarrow{a} T'_1,\ T_2 \xrightarrow{a} T'_2,\ a \neq \tau}{T_1 || T_2 \xrightarrow{a} T'_1 || T'_2}$$

2.2.2. *Properties*
A safety property is specified as a *deterministic* LTS that contains no τ transitions, and no π state. The set of traces $Tr(P)$ of property P defines the set of acceptable behaviors over αP. An LTS T satisfies P, denoted as $T \models P$ iff $Tr(T \uparrow \alpha P) \subseteq Tr(P)$.

The LTSA automatically derives from a property P an *error LTS* denoted P_{err}, which traps possible violations with the π state. Formally, the error LTS of a property $P = \langle S, \alpha P, R, s_0 \rangle$ is $P_{err} = \langle S \cup$

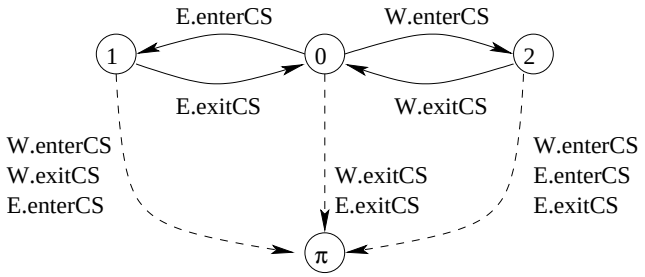

Figure 2. Mutual exclusion property

$\{\pi\}, \alpha P_{err}, R', s_0\rangle$, where $\alpha P_{err} = \alpha P$ and $R' = R \cup \{(s, a, \pi) | a \in \alpha P$ and $\neg \exists s' \in S : (s, a, s') \in R\}$. Note that the error automaton is *complete*, i.e., each state (other than the error state) has outgoing transitions for every action in the alphabet.

For example, Figure 2 illustrates a mutual exclusion property for a system consisting of the LTSs of Figure 1. The property comprises states $0, 1, 2$ and the transitions denoted by solid arrows. It expresses the fact that the component and its environment should never be in their critical sections at the same time. In other words, the intervals defined by their mutual *enterCS* and *exitCS* actions should never overlap. The dashed arrows illustrate the transitions to the error state that are added to the property to obtain its error LTS.

Let T be an LTS that has no error traces. To detect violations of property P by component T, the LTSA computes $T||P_{err}$. It has been proven in (Cheung and Kramer, 1999) that T violates P iff the π state is reachable in $T||P_{err}$, or equivalently, iff $errTr(T||P_{err}) \neq \emptyset$. The error state has special treatment during minimization, so that a violation does not disappear as a result of abstraction. In fact, an error state within a component can only disappear with composition, i.e., if a component with which it interacts blocks the erroneous behavior.

3. Assumption Generation

In this section we describe in detail our extensions to traditional model checking, and their implementation in LTSA. We also provide a formal proof of correctness.

3.1. General Method

The traditional approach to verifying a property of an *open system* (i.e., a software component that interacts with an environment, represented by other components) is to check it for all the possible environments.

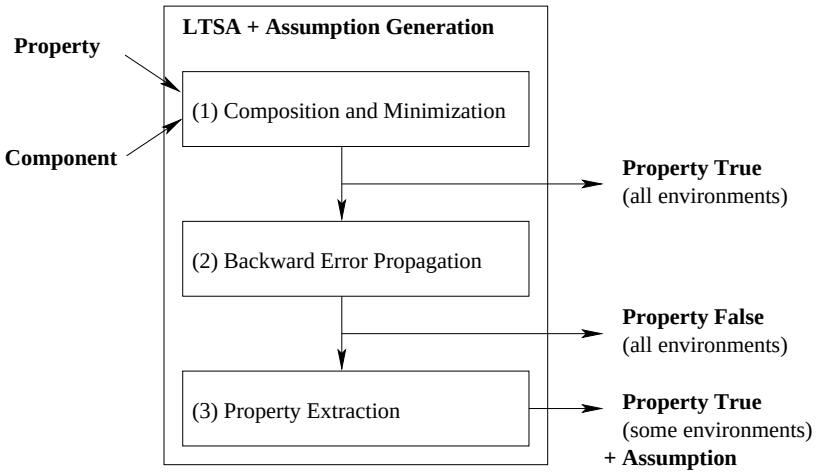

Figure 3. Model Checking with Assumption Generation

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. We believe that this approach is overly pessimistic and only appropriate for the analysis of *closed* systems, where no further interaction with the environment is expected. When analyzing open systems, an optimistic view, which assumes a *helpful* environment, is more appropriate. Usually, software components are required to satisfy properties 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 also **true**, if the property holds for *all* environments. 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 a specific assumption. This assumption, i.e. a *property* LTS, is automatically generated and characterizes exactly those environments. Intuitively, this environment assumption encodes all possible "winning strategies" of the environment in a game between the system, which attempts to get to the error state, and the environment, which attempts to prevent this. Figure 3 illustrates our approach together with the steps we follow to build the assumptions (that are described below).

Step 1: Composition and Minimization

Given an *open* system and a *property* LTS that may relate the behavior of the system with the behavior of the environment, our first step

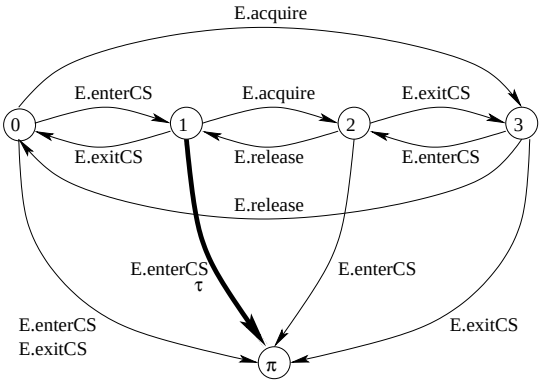

Figure 4. Composite LTS

is to compute all the violating traces of the system for unrestricted environments, and turn into τ all actions in these traces over which the environment has no control, i.e., the internal actions of the system. We perform this step by building the *composition* of the system with the *error* LTS of the property, and subsequently hiding the internal actions of the system. The resulting LTS can be minimized with respect to any equivalence that preserves (error) traces. In our implementation, we use minimization with respect to observational equivalence as defined in the presence of error states by Cheung and Kramer in (Cheung and Kramer, 1999), and as supported by the LTSA tool. For example, Figure 4 depicts the result of composing the components depicted in Figure 1 with the mutual exclusion property of Figure 2, after minimization. The internal actions of the system, i.e. the "W" labeled transitions, were abstracted to 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 in all circumstances; this is achieved through the following steps.

Step 2: Backward 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. Since we are interested only in the error traces, 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 possibly

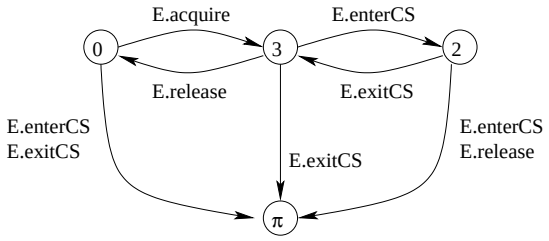

Figure 5. The Result after Backward Error Propagation

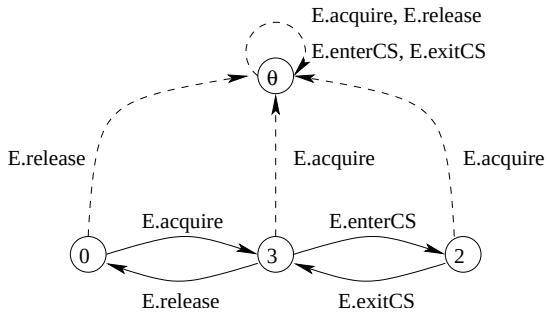

Figure 6. Generated Assumption (after deletion of π state)

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 4. The thicker line marks the only τ transition that remains in the system after minimization. As a result of backward propagation, we identify state 1 with the error state; the result is shown in Figure 5. 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

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, although it contains an error state. Recall from the background section that the error LTS is 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 (Aho et al., 2000), 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 nondeterministic LTS. In our context, if any one of the states in the set is 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.

For example, consider again the composite system in Figure 4. There are two outgoing transitions from the initial state 0 that are labeled by the same environment action *E.enterCS*: one leads to the error state, while the other one leads to state 1. This means that if the environment performs action *E.enterCS*, it can not prevent the system from getting to the error, so we would like to identify state 1 with π. In our example in Figure 4, this was achieved during Step 2, but this may not be the case in general.

What remains to be performed at this stage is to make the resulting LTS complete. *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.

Once we have the error LTS, we obtain the assumption by deleting the error state and the transitions that lead to it. Figure 6 depicts the assumption generated for our example. Since the result from Step 2 is already deterministic, we get the assumption by completing it with the sink state, denoted by θ, and deleting the π state. The assumption expresses the fact that the environment should only access its critical section protected by the mutex. Moreover, as imposed by the mutex, *E.acquire* and *E.release* actions of the environment can only alternate, and therefore any different behavior is inconsequential. Notice for example that from state 0, action *E.release* leads to state θ.

3.2. Implementation in the LTSA

As mentioned, the LTSA provides a framework that facilitates the introduction of the extensions we have presented. For example, we took advantage of its support for composition, abstraction, minimization and determinization. The extra features that our approach required are:

- Special treatment of the error state, π, during determinization. The special semantics of this state were not previously taken into account.

- Backwards reachability and error propagation as required by step 2. We believe that error propagation should be performed during CRA for increased efficiency, irrespective of our approach.

- Completion with the sink state, θ, and property extraction from the error LTS.

3.3. Correctness of Approach

Let T denote an *open* system with alphabet αT and let E denote another system representing an arbitrary environment for T, whose alphabet is αE. Let P denote a *property* LTS with alphabet $\alpha P \subseteq \alpha T \cup \alpha E$ (a property may refer to actions in both T and E).

Let $\mathcal{C} = \alpha T \cap \alpha E$ be the set of *common* actions between T and E, and let $\mathcal{I} = \alpha T - C$ denote the *internal* actions of the system.

Our approach generates the *property* LTS A with alphabet $\alpha A = \mathcal{C} \cup (\alpha P - \mathcal{I})$, representing the weakest assumption characterizing all the environments that, composed with the system, satisfy the property, i.e., $E \models A$ if and only if $E||T \models P$.

The following proposition says that the error traces of A_{err} are obtained from the traces in $T||P_{err}$ that *may* lead to an error state, from which we remove the actions not present in αA.

PROPOSITION 3.1. $errTr(A_{err}) = \{\sigma \in \alpha A^* | \exists \sigma' \in errTr(T||P_{err}) \wedge \sigma = \sigma' \upharpoonright \alpha A\}$.

The following theorem makes precise the claim that A is the weakest assumption about the environment E of T that ensures property P.

THEOREM 3.2. *$\forall E$, $E \models A$ if and only if $E||T \models P$.*

Proof.

- $\forall E$ such that $E \models A$, we have to show that $E||T \models P$. The proof is by contradiction.

  Assume $E||T \not\models P$. Then, there is a trace σ in $E||T||P_{err}$ that leads to the error state (i.e., $\sigma \in errTr(E||T||P_{err})$). We use σ to build a trace $\sigma' \in Tr(E)$ such that $\sigma' \upharpoonright \alpha A \in errTr(A_{err})$, thus contradicting $E \models A$.

  Since σ is an error trace in $E||T||P_{err}$, it follows that $\sigma \upharpoonright \alpha E \in Tr(E)$ and $\sigma \upharpoonright (\alpha T \cup \alpha P) \in errTr(T||P_{err})$. From Proposition 3.1, it follows that $(\sigma \upharpoonright (\alpha T \cup \alpha P)) \upharpoonright \alpha A \in errTr(A_{err})$.

Since $\alpha A \subseteq \alpha E$ and $\alpha A \subseteq \alpha T \cup \alpha P$, we also have that $(\sigma \upharpoonright \alpha E) \upharpoonright \alpha A = (\sigma \upharpoonright (\alpha T \cup \alpha P)) \upharpoonright \alpha A$. Let $\sigma' = \sigma \upharpoonright \alpha E$. We then have that $\sigma' \upharpoonright \alpha A = (\sigma \upharpoonright (\alpha T \cup \alpha P)) \upharpoonright \alpha A \in errTr(A_{err})$, and thus we have a contradiction.

– $\forall E$ such that $E||T \models P$, we have to show that $E \models A$. Again, we prove this by contradiction.

Assume $E \not\models A$. Then, there is a trace $\sigma \in Tr(E)$ such that $\sigma \upharpoonright \alpha A \in errTr(A_{err})$. From Proposition 3.1, it follows that there is a trace $\sigma' \in errTr(T||P_{err})$ such that $\sigma \upharpoonright \alpha A = \sigma' \upharpoonright \alpha A$. We use σ and σ' to build a trace σ'' in $E||T||P_{err}$ such that $\sigma'' \upharpoonright \alpha P \in errTr(P_{err})$, thus reaching the contradiction of $E||T \models P$.

Since σ is a trace of E, σ' is a trace of $T||P_{err}$, $\sigma \upharpoonright \alpha A = \sigma' \upharpoonright \alpha A$ and $\mathcal{C} \subseteq \alpha A$ it follows that σ and σ' may differ only on non-common actions. It follows that there exists a trace σ'' in $E||T||P_{err}$ such that $\sigma'' \upharpoonright \alpha E = \sigma$ and $\sigma'' \upharpoonright (\alpha T \cup \alpha P) = \sigma'$. (we build σ'' by "composing" σ and σ' using the same rules as for parallel composition of systems).

Since $\sigma' \in errTr(T||P_{err})$, it follows that $\sigma' \upharpoonright \alpha P \in errTr(P_{err})$. We also have $\sigma'' \upharpoonright \alpha P = \sigma' \upharpoonright \alpha P$, since σ may introduce in σ'' only actions that are not present in αA or αP. It follows that $\sigma'' \upharpoonright \alpha P \in errTr(P_{err})$, and thus we have a contradiction.

□

3.4. Potential Deadlock Removal

The construction we have presented so far will build the weakest assumption A about the environment for component T to achieve a given property P. The weakest assumption characterizes all environments in the context of which the behavior of the component will be restricted as necessary for P to always hold. However, it is possible that in order to ensure P, the assumption A leaves to T no alternative than to stop interacting with the environment indefinitely. We call such situations introduced by the assumption *potential deadlocks*, or deadlocks, for simplicity.

Consider for example the component and property illustrated in Figure 7. The given property P requires that the component either performs a b followed by as, or just as; the component, however, expects to perform a b after the first a. The weakest environment assumption, as per the above construction, is depicted in Figure 8. Under this environment assumption, the given component would deadlock if an initial

a transition is undertaken since a b transition is not feasible. Although such behaviour is correct according to the semantics of the property, it may be undesirable. In this section, we present a modification to our construction that avoids this particular type of deadlocks, if they are undesirable.

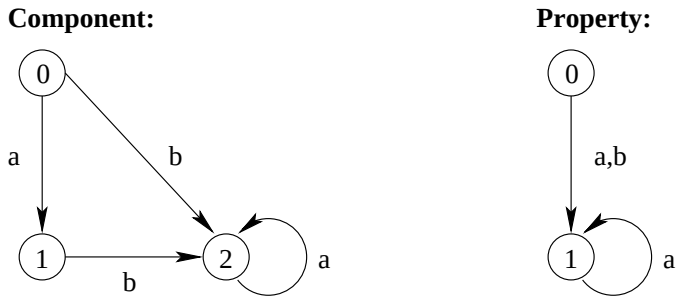

Figure 7. A Potential Deadlock

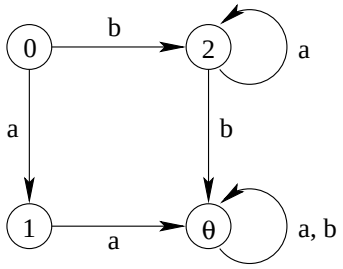

Figure 8. The Weakest Environment Assumption

To make our presentation more precise, we first introduce a few definitions. Given an LTS $T = \langle S, \alpha T, R, s_0 \rangle$ representing the behaviour of a component or parallel composition of components, we say that:

- a state $s \in S$ is *reachable* from state s' iff either s is s' or there is some $(s'', a, s) \in T$ and s'' is reachable from s'; a state $s \in S$ is *reachable* iff s is reachable from s_0.
- a state $s \in S$ is a *terminal state* iff s is reachable and s has no outgoing transitions, i.e. there is no $(s, a, s') \in R$ for any a and s'.
- T is *non-terminating* iff it contains no terminal states.
- T is *potentially finite* iff T has at least one terminal state.

For an LTS T representing a reactive process (meant to continuously interact with its environment) we then say that:

– T has a *deadlock* iff T is potentially finite.

– T is *free of deadlock* iff T is non-terminating.

However, the above definition of deadlock is too strong; whilst it is good for an LTS representing a single component, it does not characterize deadlock in a sub-component of some composition. For example, suppose the property LTS of Figure 7 is modified by adding a new transition, labeled by an action not in the alphabet of the component - say c - from state 1 back to state 1. This results in the weakest environment assumption of Figure 9 which has a c loop on state 1. Under the above definitions, this modified LTS does not introduce a deadlock. However, an environment that first performed an a and then continued with cs would result in the component getting deadlocked at state 1 since no b would ever be forthcoming.

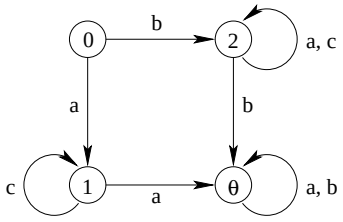

Figure 9. The Weakest Assumption for the Modified Property

To handle such cases, we modify the above definitions for deadlock to be relative to a given (component) alphabet, say $\alpha C \subseteq \alpha T$. The principal change is:

– a state $s \in S$ is *terminal wrt* αC iff s is reachable and no path from s is labelled at any point by an interface action from αC.

Now given a component T, which is free of deadlock under the above definitions, and a desired property, the following modification to our construction will yield a weakest assumption that does not introduce deadlock. Step 2 becomes an iterative process during which 1) deadlock states are identified as errors and 2) backward error propagation is applied, until these steps have no effect on the resulting graph. In this context, deadlock states are states that are terminal if we ignore all their outgoing transitions that lead directly to the error state. These transitions are ignored because they will be removed by our construction of the weakest assumption. Note that, since determinization as applied by step 3 of our algorithm changes the structure of an LTS, further deadlocks may be identified. Therefore, right after determinization and

before completion, step 3 also applies the iterative deadlock removal process described above. Note that the deadlock removal process of step 2 cannot be skipped, because determinization does not preserve deadlocks.

4. Applications

In the context of our project on the verification of autonomous systems, we applied our approach as presented in Section 3 to components of two such applications, the executive subsystems of the K9 Mars Rover and of the Remote Agent.

For the K9 Mars Rover our framework was used to illustrate to the developers the way in which some required properties decompose across components of the system. For the Remote Agent, our approach detected the violation of a property in an assume-guarantee style. The Remote Agent case study also demonstrates a situation where the generated assumption may cause the component to deadlock. This potential deadlock could be removed by applying the modified construction presented in Section 3.4.

4.1. K9 Mars Rover executive

Our first application is the planetary rover controller K9, and in particular its executive subsystem, developed at NASA Ames Research Center. The executive receives flexible plans from a planner, which it executes according to the plan language semantics. A plan is a hierarchical structure of actions that the Rover must perform. Traditionally, plans are deterministic sequences of actions. However, increased Rover autonomy requires added flexibility. The plan language therefore allows for branching based on state or temporal conditions that need to be checked, and also for flexibility with respect to the starting time of an action. The plan language allows the association of each action with a number of state or temporal pre-, maintenance, and post-conditions, which must hold before, during, and on completion of the action execution, respectively.

Description
The executive has been implemented as a multi-threaded system (see Figure 10), made up of a main coordinating component named *Executive*, components for monitoring the state conditions *ExecCondChecker*, and temporal conditions *ExecTimerChecker* - each further decomposed into two threads - and finally an *ActionExecution* thread that is responsible for issuing the commands to the Rover. Synchronization between

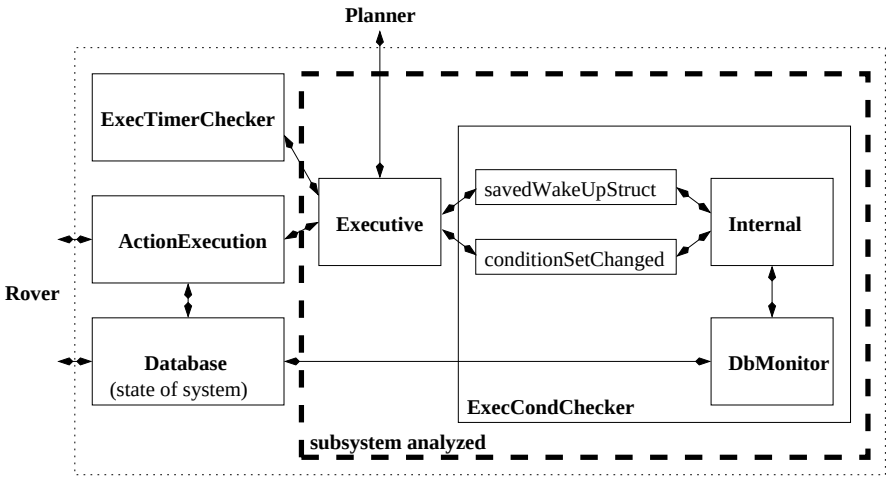

Figure 10. The Executive of the K9 Mars Rover

these threads is performed through mutexes and condition variables. The developers provided some design documents to us, which described the synchronization between these components in an ad-hoc flowchart-style language. They looked very much like LTSs, which allowed us to translate them in a straightforward and systematic, albeit manual, way into FSP for the LTSA.

Model checking

We first checked the occurrence of race conditions for the case of a variable (*conditionSetChanged*) of the *ExecCondChecker* shared with the *Executive*. We checked the property on the *ExecCondChecker* (that consists of threads *Internal* and *DbMonitor*) together with the mutexes it uses, since mutexes constitute the synchronization mechanism in this system. The *ExecCondChecker* with mutexes and the property had 426 states but minimized to 18 states. The propagation of the error state then produced an LTS of just 10 states, and the final assumption generated had 12 states (one being the sink state). We were surprised to see that our approach did not generate the expected assumption, i.e. that accesses to the shared variable by the environment must be protected by the appropriate mutex, as in the example of Section 3. In fact, the assumption obtained was weaker. It reflected the knowledge that, once the environment holds the mutex, the values that the environment reads reflect changes that only the environment may have made. For example, assume that, while holding the mutex, the environment assigns value x to the variable. Then reading any value $x' \neq x$ would lead the environment to the sink state, because this behavior will never actually be exercised in the context of the *ExecCondChecker*.

The second property that we checked in this fashion was one that the developers thought might be violated by the code, but could actually not produce an execution that would demonstrate this fact. For a specific variable (*savedWakeUpStruct*) of the *ExecCondChecker* shared with the Executive, the property stated the following: if the Executive reads the value of the variable, then the *ExecCondChecker* should not read this value until the *Executive* clears it first. Again, we used the *ExecCondChecker* together with mutexes and the property to generate an assumption on the behavior of the *Executive*. The result had 524 states, minimized to 9 states, reduced to 7 states with error propagation, and to 6 states with determinization as applied by step 3 of our construction (see Section 3). The resulting assumption had 7 states (including the sink state). It stated that the environment of the component should read the variable after acquiring a mutex, and should hold on to that mutex until it clears the variable. Note that, again, there were transitions to the sink state, expressing the fact that some behavior of the environment is never exercised. For example, the assumption made clear that the *ExecCondChecker* only updates the variable with values larger than the one it currently holds.

The assumption generated was satisfied by the design level *Executive*. Our result gave confidence to the developers about the correctness of their design and implementation. They also found it very useful to be able to understand how the property decomposes across modules of the system.

4.2. Remote Agent executive

NASA's Remote Agent (RA) is an autonomous spacecraft control architecture that was one of 12 technologies tested on the DEEP-SPACE 1 spacecraft launched in October 1998. It demonstrated for the first time in NASA's history the complete control of a spacecraft by artificial intelligence based software. Similarly to the K9 Rover, the architecture of the RA includes a planner and a plan execution module (executive).

The RA executive (RAX) was developed collaboratively by NASA Ames and the Jet Propulsion Laboratory (JPL) (Pell et al., 1997). In a very successful application of model checking to the RAX *before flight*, the Software Engineering group at NASA Ames discovered three subtle but critical errors in the system, that had not been uncovered through testing (Havelund et al., 2001). This section describes how our assumption generation approach has been subsequently used to detect one of these known errors in an assume-guarantee style. We used an FSP model of the RAX developed for a previous experiment; the FSP

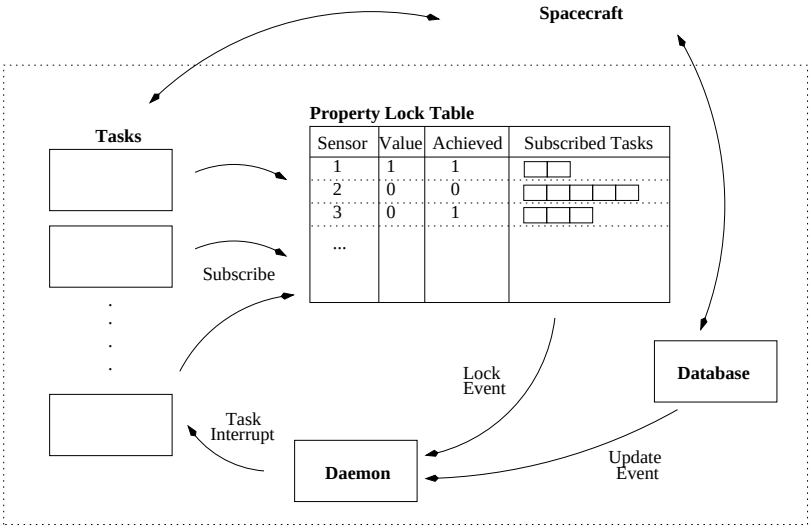

Figure 11. The Executive of the Remote Agent

model was translated from the Promela model (Holzmann, 1991) used in the original experiment.

RAX description

The RAX is designed to support execution of software-controlled *tasks* on board the spacecraft (see Figure 11). A task may be, for example, to run and survey the camera. A task typically requires that some *properties* hold throughout its execution, where a property is a pair representing a specific value for a specific equipment sensor. For example, the camera-surveying task may require the camera to be turned on throughout task execution.

Notice that tasks are similar to actions and properties are similar to conditions in the Rover example. However, in the RAX, tasks may run *concurrently*, unless they are *conflicting*, i.e., they require different values for the same sensor. When tasks are started, they subscribe for the properties upon which they depend. Among the subscribing tasks for a particular property, a single one, named the *owner*, is responsible for *achieving* the property. The remaining tasks go to sleep; a task is then awoken when any one of its subscribed properties is achieved. When all properties on which a task depends are achieved, the task starts performing its main action, otherwise it may go back to sleep. However, a property may be unexpectedly broken due to some fault, in which case executing tasks that depend on the specific property must be interrupted.

To prevent conflicting tasks from executing simultaneously, the RAX provides a locking mechanism, implemented as a *Property Lock Table*.

The table records property locks in terms of 1) the sensor that they refer to, 2) the desired sensor value, 3) whether the desired value has been achieved – recorded in the *achieved bit*, and 4) which (non-conflicting) tasks currently hold the particular lock (see Figure 11). A database is also used to record the actual values of the spacecraft sensors. To ensure that properties are maintained during task execution, a *Daemon* periodically checks for inconsistencies between the lock table and the database. If some achieved value in the lock table disagrees with the corresponding value in the database, the *Daemon* interrupts all tasks currently holding the particular lock.

Model Checking

We applied our framework to check the requirement that: *whenever there exist inconsistencies in the system that affect task execution, affected tasks will be interrupted*. For this property, we decomposed the system into two parts, the *Daemon* on the one side, and the tasks, lock table, and database on the other. We used an instance of the system with two tasks that both require some sensor with id 1 to have value 1.

We then generated an assumption for the *Daemon* to satisfy the requirement, and used the rest of the system to discharge this assumption. The Daemon together with the property consisted of 38 states minimized to 12 states, and the assumption generated consisted of 11 states (one being the sink state). The assumption expressed the fact that, throughout the execution of the main function of a task, the achieved bit for its required properties must be set.

When trying to discharge the assumption on the rest of the system, the following counterexample was obtained (2142 states explored out of 14059), which illustrates a violation of the required property in the system:

`task.1.lock.1.acquire.1`	*(Task 1 acquires lock for sensor 1, value 1)*
`task.1.db.1.read.0`	*(Task 1 reads in DB value 0 for sensor 1)*
`task.1.lock.1.owner_is.1`	*(Task 1 is owner of lock for sensor 1)*
`task.1.db.1.write.1`	*(Task 1 achieves property)*
`task.2.lock.1.acquire.1`	*(Task 2 acquires lock for sensor 1, value 1)*
`task.2.db.1.read.1`	*(Task 2 finds that sensor 1 has value 1)*
`task.2.lock.1.owner_is.1`	*(Task 2 is not the owner)*
`task.2.start_operation.1`	*(Task 2 starts its main operation)*
`db.1.external_write.0`	*(fault sets sensor 1 to value 0)*
`daemon.1.read.0.1.0`	*(Daemon checks values for sensor 1; in DB: 0; in lock table: 1; achieved bit:0)*

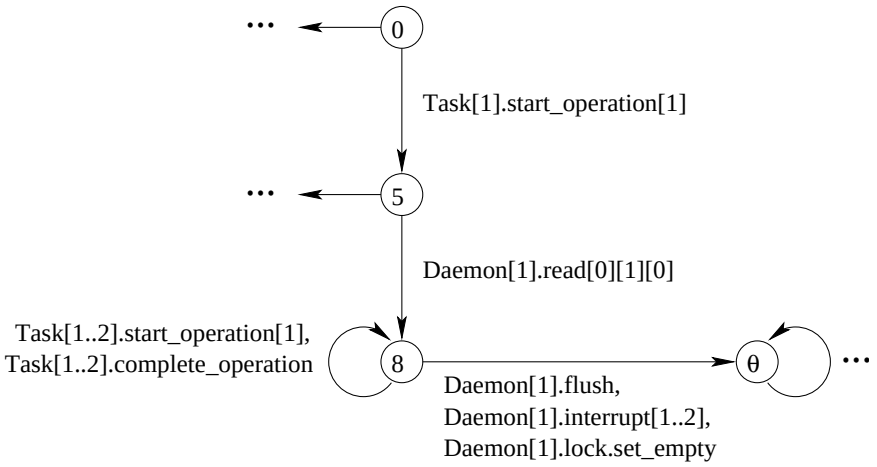

Figure 12. Assumption (excerpts) produced for RAX

This counterexample reflects a problem in the system caused by the fact that a task achieves a property and sets the achieved bit in the lock table in a non-atomic fashion. As a result, a different task may in the meantime be scheduled, find that the value of the sensor is as expected, and start its operation. At this point, if the *Daemon* detects an inconsistency, the achieved bit will not be set, and therefore tasks will not get interrupted. The same problem was detected in (Havelund et al., 2001) and was fixed by the developers of the RAX by introducing a critical section around the code that achieves a property and sets the achieved bit.

The largest state space involved in the application of our approach was explored when discharging the assumption, and consisted of 2142 states (out of 14059 states), at which stage a violation was detected and a counterexample was produced. Performing verification of the RAX directly (i.e., by composing all the RAX components with the property) and obtaining a similar counterexample would require exploring 6240 states out of 37760. This shows the potential benefits of our approach as compared to monolithic, non-compositional, model checking. In this case study, the assumption produced referred to a component that was small as compared to the rest of the system. We expect that when assumptions are computed for larger components (with small interfaces) the benefits of our approach will be more pronounced.

4.3. Component deadlocks

As mentioned earlier, the RAX case study exhibited that, although the assumptions that we generate always prevent a component from

reaching error states, they might also prevent the component from participating in the system behavior. In Section 3.4, we discussed modifications to our construction algorithm to remove such component deadlocks. Figure 12 depicts a portion of the assumption generated for the *Daemon* in the RAX case study. In this assumption, state 8 is reached from the initial state when *Task 1* starts its main operation and the *Daemon* finds that the values of the sensor in the database and the lock table disagree while the achieved bit is 0 (i.e., `daemon.1.read[0][1][0]`). Although state 8 appears to allow the *Daemon* to keep interacting with its environment, one can observe that all such interactions lead to the sink state. As discussed in Section 3, transitions to the sink state reflect legal environment behavior that is never exercised in the context of the component under analysis. Note also that the self-loop transitions on state 8 are labeled with actions that were introduced by the property, and that do not belong to the *Daemon* interface. Therefore, state 8 reflects a component deadlock situation where the *Daemon* will be inactive in the system. Precisely such a deadlock would be removed by our modified procedure.

5. Discussion

The complexity bottleneck of our approach is the determinization step, which, in the worst case, is exponential in the number of the states of the given LTS. There are several reasons that lead us to believe that this may not be the case often in practice. In our experiments such as the Rover study reported in Section 4, non-determinism almost disappears by propagation of the error state. Moreover, in the subset construction of step 3 of our construction, composite states including the error state also become error states and all outgoing transitions (and subsequent behaviors) are pruned. As we only study modules of a larger system, we expect that the state space of these modules will be relatively small. This will be the case in particular when they interact through limited interfaces with their environment, which will allow the minimization step to considerably reduce their behavior. Note also that, if we extend our results to other frameworks, the assumption may not be required to be deterministic. Admittedly, however, deterministic assumptions tend to be clearer to understand.

From our extensive experience with compositional reachability analysis (CRA) techniques, we are only too aware of the potential intermediate state explosion associated with them (Graf et al., 1996). This problem describes the fact that, in lack of a context, a component may exhibit an excessively large state-space. However, this does not

occur in the general case for well-designed software architectures. Moreover, several approaches have been proposed in the literature (Graf et al., 1996; Cheung and Kramer, 1996; Krimm and Mounier, 1997) for addressing the problem.

The assumptions produced by our approach are *weakest*, that is, they restrict the environment no more and no less than is necessary for a component to satisfy a given property. The possibility to generate these assumptions automatically has direct application to assume-guarantee proofs. More specifically, it removes the burden of specifying assumptions manually thus automating this type of reasoning. However, our algorithm does not compute partial results, meaning no assumption is obtained if the computation runs out of memory, which may happen if the state-space of the component is too large. We address this problem in (Cobleigh et al., 2003), where we present a novel framework for performing assume-guarantee reasoning in an incremental and fully automatic fashion. To check a component against a property, assumptions are generated that the environment needs to satisfy for the property to hold. These assumptions are then discharged on the rest of the system. Assumptions are computed by a learning algorithm. They are initially approximate, but become gradually more precise by means of counterexamples obtained by model checking the component and its environment, alternately. This iterative process may at any stage conclude that the property is either true or false in the system. Moreover, even if it runs out of memory before reaching conclusive results, intermediate assumptions may be used to give some indication to the developer of the requirements that the component places on its environment.

Our approach extends the LTSA tool in several useful ways. First of all, it achieves further reduction of component behavior by applying propagation of the error states, a computationally inexpensive but efficient step. Moreover, our approach generates the *weakest* environment assumptions. As such, these assumptions may be used for runtime monitoring, or for component retrieval, capabilities that were not formerly provided by the tool.

As far as component retrieval is concerned, we would like to stress the following observation from our experiments (Section 4). The sink state that our assumptions contain, reflects the fact that some services that a component provides will never be used in the context of a system. Our assumptions allow free implementations for these services, and simply ensure that the used services comply with the requirements.

The ability to generate assumptions also opens up a number of other interesting research directions: we mention a few to give some flavor.

– Our work has been performed with a limited but important set of properties (safety) expressed within a specific framework that facilitates the development of our algorithms. However, we believe our approach has application in other frameworks. In particular, we are investigating the extension of our approach for the case of fairness and/or liveness properties, which requires a more expressive formalism.

– When the behavior of the environment, or part thereof, is provided, we wish to find effective ways of discharging assumptions on the environment. One way would be to use the assumption as a property, and model check in the same fashion components in the environment. This process can be seen as a way of decomposing, automatically, a property across components of a system. Indeed, an assumption reflects those aspects of the property that have not been satisfied by the component and that remain to be satisfied by its environment. Property decomposition is an extremely difficult problem, and our approach may be seen as a helpful step in its facilitation. Of course, such decomposition will not be effective in all cases. It is easy to imagine that there will be cases where assumptions may gradually grow in size during this process, a problem referred to in the literature as "property explosion".

– Our approach to assumption generation can straightforwardly be used for submodule construction, where the submodule is placed as an interacting component in parallel with the given one. Generalization to other forms of composition is a natural step, such as sequential composition, for example.

– Assumptions may be further analyzed. For example, if the generated assumption expects the environment to hold on to a specific lock for ever, this may indicate something inherently wrong with the behavior of the component under analysis.

6. Related Work

For over three decades now, there has been research effort focused on finding tractable approaches to the formal specification, design and development of complex systems. Significant early progress occurred with techniques and tools for sequential, non-interacting or transformational systems. However, the quest for obtaining effective methods and tools for the formal support of compositional and/or modular development

and reasoning for *reactive* systems still remains, in our view, a major challenge. As there is insufficient space to do justice to the work that has been undertaken, we refer the interested reader to the proceedings of (de Roever et al., 1997) - its introductory chapter in particular (de Roever, 1997) - and the recent book (de Roever et al., 2001).

In more recent years with the development and take-up of OO-design technology, formal techniques for support of component-based design is also gaining prominence, see for example (de Alfaro and Henzinger, 2001a; de Alfaro and Henzinger, 2001b), for which *modular*-based reasoning is key. The work of Inverardi and colleagues, (Inverardi et al., 2000), has also been aimed at providing support for the modular checking of certain properties, such as deadlock freedom, but is somewhat limited in the checks performed for compatibility between components.

In order to make progress in any of these areas, some form of assumption (either implicit or explicit) about the interaction with, or interference from, the environment has to be made. Even though we have sound and complete reasoning systems for such rely-guarantee (or assumption-commitment) style of reasoning, see for example (Jones, 1983; Stølen, 1991) and most recently (Xu et al., 1997), it is always a mental challenge to obtain the most appropriate assumption (if there is such). It is even more of a challenge to find automated techniques to support this reasoning style - the thread modular reasoning underlying the Calvin tool (Flanagan et al., 2002) and the assume-guarantee software verification framework presented in (Păsăreanu et al., 1999) are examples in this direction. In the framework of temporal logic, the work on Alternating time Temporal Logic ATL (and transition systems) (Alur et al., 1997) was proposed for the specification and verification of open systems together with automated support via symbolic model checking procedures, albeit of rather high complexity; the Mocha toolkit (Alur et al., 1998) provides support for modular verification of components with requirement specification based on the ATL. It goes without saying that if tool support is lacking, take-up of these techniques will be rather low.

The underlying approach to automated assumption generation that we've adopted and implemented in LTSA has similarity to a number of other problems that have been considered by a number of researchers over the past two decades. Closest to our our work in the software engineering and concurrency theory are the "sub-module construction problem", "scheduler synthesis" and "interface equation solving" problems. In the discrete event community, it appears as the "supervisory control" problem, in control theory there is the "model matching" problem and in the logic synthesis world there is the "interacting FSM synthesis". Of course, the particular frameworks in which these prob-

lems are considered makes all the difference to their solution(s) and as such it would be quite inappropriate to claim they are solving the same problem. However, in very general terms, each can be seen as an instance of the following problem: given a component, C, and a desired behaviour, B, find a context for C, X, such that $X(C) \equiv B$, for some appropriate notion of equivalence.

Merlin and Bochmann (Merlin and Bochmann, 1983) were probably the first to address the above as submodule construction in the world of communication protocol specification and synthesis. In a setting of labelled transition systems, given a module specification M_0 and a submodule specification M_1, they outlined and exemplified a manual approach to construct an interacting submodule M_2 such that M_1 and M_2 together achieve the desired specification of M_0. Their construction has much in common with ours although some significant aspects of the construction were left unspecified. The later work of (Sidhu and Aristizabal, 1988; Haghverdi and Ural, 1999) has revisited the Merlin-Bochmann approach and provided new, detailed, algorithms for the sub-module construction and implemented an automated tool. One recognized limitation of the Merlin-Bochmann is that the notion of correctness, namely just trace equivalence, does not capture a number of behavioural properties, e.g. potential deadlock.

The work of Shields (Shields, 1989), over a decade later, introduces the "Interface Equation" in the setting of the process algebra, CCS (Milner, 1989), under observational equivalence. In order to solve $(C|X)\backslash_L = B$ for the process X, he restricts to cases where B is deterministic, with some minor restrictions on the sorts of C and B, and provides necessary and sufficient conditions for a solution to exist and then in such situations presents an explicit construction. Parrow (Parrow, 1989) also addressed the interface equation and presented a procedure for solving the equations via successive transformation of the CCS equations to simpler ones, generating a solution along the way; his approach is based upon a tableau method. Parrow's method attempts to find a *most general* solution, but even if this solution exists, it is not necessarily appropriate for implementation. Continuing in the process algebra framework, Larsen and Xinxin (Larsen and Xinxin, 1990) consider the more general problem of solving a system of equations $C_i(X) \simeq P_i$, for $0 < i \leq n$, where the C_i are arbitrary contexts, P_i are arbitrary processes and X is the process to be found - the equivalence is taken as bisimulation. They considered the problem in the context of disjunctive modal transition systems, (Larsen and Thomsen, 1988) and implemented an automated tool that would solve the equations (in the finite state case) when a solution exists.

As stated above, there is a further body of work in supervisory control synthesis, discrete event systems, and logic synthesis areas, see for example (Aziz et al., 1995; di Benedetto and Sangiovanni-Vincentelli, 2001; Tronci, 1998; Khatri et al., 1996; Balemi et al., 1993). However, we should stress that whilst these approaches are in general set in a FSM/DFA context, the principal goal is quite different in comparison with ours.

7. Conclusions

We presented an approach to model checking components as open, rather than closed systems. Our approach reports whether there is something inherently wrong with the component behavior, or whether satisfying a requirement is simply a matter of providing the right environment. Moreover, it characterizes exactly all helpful environments.

The possibility of generating assumptions provides increased flexibility in model checking, and opens up a number of interesting research topics. It allows, for example, the discharge of assumptions at run-time for unpredictable environments, the retrieval of components focused on only relevant aspects of their behavior, or the decomposition of properties across components. It remains to further investigate how useful our approach is in practice. Open research issues include optimizations and extensions for fairness/liveness properties and other frameworks. However, our early experiments with real case studies provide strong evidence in favor of this line of research.

Acknowledgements

The authors wish to thank Klaus Havelund for providing descriptions and the Promela model for the Remote Agent case study. The third author is most grateful for the partial support received from RIACS/USRA to undertake this research whilst on leave at NASA Ames Research Center.

References

Aziz, A., F. Balarin, R. K. Brayton, M. D. Dibenedetto, A. Sladanha, and A. L. Sangiovanni- Vincentelli. Supervisory control of finite state machines. In P. Wolper, editor, *7th International Conference On Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 279–292, Liege, Belgium, 1995. Springer Verlag.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2000.

Alur, R., T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In *Proceedings of 10th International Conference on Computer Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525. Springer Verlag, 1998.

Alur, R., T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In de Roever et al. (de Roever et al., 1997), pages 23–60.

Balemi, S., G. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. Franklin. Supervisory control of a rapid thermal multiprocessor. *IEEE Transactions on Automatic Control*, 38(7):1040–1059, July 1993.

Cheung, S. and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 5(4):334–377, 1996.

Cheung, S. and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–78, 1999.

Clarke, E. M., O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

Cobleigh, J. M., D. Giannakopoulou, and C. S. Păsăreanu. Learning Assumptions for Compositional Verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, April 2003.

de Alfaro, L. and T. Henzinger. Interface automata. In *Proc. of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM Press, 2001.

de Alfaro, L. and T. Henzinger. Interface theories for component-based design. In *Proceedings of EMSOFT 01: Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer Verlag, 2001.

de Roever, W.-P. The need for compositional proof systems: A survey. In de Roever et al. (de Roever et al., 1997), pages 1–22.

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

de Roever, W.-P., H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference - An International Symposium, COMPOS'97*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

di Benedetto, M. and A. Sangiovanni-Vincentelli. Model matching for finite-state machines. *IEEE Transactions on Automatic Control*, 46(11):1726–1743, November 2001.

Flanagan, C., S. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proceedings of the European Symposium on Programming*, 2002.

Giannakopoulou, D., J. Kramer, and S. Cheung. Analysing the behaviour of distributed systems using Tracta. *Journal of Automated Software Engineering, special issue on Automated Analysis of Software*, 6(1):7–35, 1999.

Giannakopoulou, D., C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, September 2002.

Graf, S., B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Aspects of Computation*, 8, 1996.

Haghverdi, E. and H. Ural. Submodule construction from concurrent system specifications. *Information and Software Technology*, 41:499–506, 1999.

Havelund, K. and G. Rosu. Monitoring Java programs with Java pathexplorer. In *First Workshop on Runtime Verification (RV'01)*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*, Paris, France, 2001.

Havelund, K., M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. In IEEE Transactions on Software Engineering, volume 27, Number 8, 2001.

Holzmann, G. *The Design and Validation of Computer Protocols.* Prentice Hall, 1991.

Inverardi, P., A. Wolf, and D. Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Transactions on Software Engineering Methods*, 9(3):239–272, July 2000.

Jones, C. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

Khatri, S., A. Narayan, S. Krishnan, K. McMillan, R. Brayton, and A. Sangiovanni-Vincentelli. Engineering change in a non-deterministic FSM setting. In *Proceedings of 33rd IEEE/ACM Design Automation Conference*, 1996.

Krimm, J.-P. and L. Mounier. Compositional state space generation from LOTOS programs. In E. Brinksma, editor, *3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, Enschede, The Netherlands, 1997. Springer.

Larsen, K. and B. Thomsen. A modal process logic. In *Proceedings of the IEEE/ACM Conference on Logic in Computer Science, LICS'88*, 1988.

Larsen, K. and L. Xinxin. Equation solving using modal transition systems. In *Proceedings of the IEEE/ACM Conference on Logic in Computer Science, LICS'90*, 1990.

Magee, J., N. Dulay, and J. Kramer. Regis: A constructive development environment for parallel and distributed programs. *Distributed Systems Engineering Journal, Special Issue on Configurable Distributed Systems*, 1(5):304–312, 1994.

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

Magee, J., J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *1st Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, TX, USA, 1999.

Magee, J., N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical animation of behavior models. In *Proceedings of the 22nd International Conference of Software Engineering (ICSE)*, 2000.

Merlin, P., and G. V. Bochmann. On the construction of submodule specification and communication protocols. *ACM Transactions on Programming Languages and Systems*, 5:1–25, 1983.

Milner, R. *Communication and Concurrency.* Prentice-Hall, 1989.

Parrow, J. Submodule construction as equation solving CCS. *Theoretical Computer Science*, 68:175–202, 1989.

Păsăreanu, C. S., M. B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 168–183. Springer-Verlag, 1999.

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

Pell, B., E. Gat, R. Keesing, N. Muscettola, and B. Smith. Plan execution for autonomous spacecrafts. In *Proc. of the Int. Joint Conf. on Artificial Intelligence*, August 1997.

Shields, M. A note on the simple interface equation. *The Computer Journal*, 32(5):399–412, 1989.

Sidhu, D. P. and J. Aristizabal. Constructing submodule specifications and network protocols. *IEEE Transactions on Software Engineering*, 14(11):1565–1577, November 1988.

Stølen, K. A method for the development of totally correct shared-state parallel programs. In J. Baeten and J. Groote, editors, *Proceedings of Concur'91*, volume 527 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

Tronci, E. Automatic synthesis of controllers from formal specifications. In *Proc. of 2nd IEEE Int. Conf. on Formal Engineering Methods, Brisbane, Australia*, 1998.

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