Interface Decomposition for Service Compositions

Domenico Bianculli*  
Faculty of Informatics  
University of Lugano  
Lugano, Switzerland  
domenico.bianculli@usi.ch

Dimitra Giannakopoulou  
NASA Ames Research Center and  
Carnegie Mellon Silicon Valley  
Moffett Field, CA, USA  
{dimitra.giannakopoulou,corina.s.pasareanu}@nasa.gov

Corina S. Păsăreanu

ABSTRACT

Service-based applications can be realized by composing existing services into new, added-value composite services. The external services with which a service composition interacts are usually known by means of their syntactical interface. However, an interface providing more information, such as a behavioral specification, could be more useful to a service integrator for assessing that a certain external service can contribute to fulfill the functional requirements of the composite application.

Given the requirements specification of a composite service, we present a technique for obtaining the behavioral interfaces — in the form of labeled transition systems — of the external services, by decomposing the global interface specification that characterizes the environment of the service composition. The generated interfaces guarantee that the service composition fulfills its requirements during the execution. Our approach has been implemented in the LTSA tool and has been applied to two case studies.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications; D.2.4 [Software Engineering]: Software/Program Verification; F.3.1 [Logics and meanings of programs]: Specification techniques; H.3.5 [Information storage and retrieval]: Online Information Services—Web-based services

General Terms

Verification

Keywords

Behavioral interface, interface decomposition, services, service compositions

*This work was carried out while the author was an intern at Mission Critical Technologies, Inc., on site at NASA Ames Research Center.

Copyright 2011 ACM. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

ICSE ’11, May 21–28, 2011, Waikiki, Honolulu, HI, USA  
Copyright 2011 ACM 978-1-4503-0445-0/11/05 ...$10.00.

1. INTRODUCTION

Service-oriented computing is a paradigm that promotes the construction of software applications by means of basic components called services, which make available a specific functionality through a set of operations accessible over a network infrastructure. Software engineers use composition mechanisms such as BPEL orchestrations [23] to assemble complex, added-value services, called service compositions, out of existing services, possibly offered by third-party providers and thus called external or partner services.

In previous work, some of the authors emphasized the need for achieving continuous lifelong verification for this new class of software, which belongs to the realm of open-world software [2]. They proposed a methodology [5] providing an integrated approach for design-time and run-time verification with an assume-guarantee flavor, and they showed how to realize continuous lifelong verification by adopting a verification-oriented life cycle [4].

At the basis of these proposals there is the definition of a set of correctness properties that the service composition should manifest. These properties usually depend on a certain behavior or on some quality of service attributes of the external services with which the composite service interacts. The aforementioned work made the working assumption that behavioral descriptions were available for the external services interacting with a composite service. These descriptions were then used as assumptions for performing the assume-guarantee style model checking of the service composition behavior, and as models to synthesize run-time monitors.

However, in the dynamic and evolvable settings that characterize open-world software, it is unrealistic to assume the availability of the interface descriptions of third-party services. In general, service providers make available to service integrators only the syntactical interface of the services they provide. It is then clear that an automated technique for deriving, from the requirements specification of a composite service, the required interface of its partner services, could improve the process followed by service integrators to assemble service compositions.

In this paper, we focus on the automatic generation of the behavioral interfaces of the partner services, by decomposing the requirements specification of a service composition. Our technique generates behavioral interfaces that constitute required specifications for the partner services; these specifications guarantee that the composite service will fulfill its required safety properties at run time, while it interacts with the external services. Since we assume that the behav-
ioral descriptions of external services are not available, our technique is based on the purely syntactical knowledge of their interfaces. In particular, the specific contributions of the paper are: i) the formulation of the interface decomposition problem for service compositions; ii) a sound, heuristic-based technique for decomposing the global interface specification of the environment into the behavioral interfaces — in the form of labeled transition systems (LTSs) — of the individual partner services; iii) the implementation of this technique in the LTSA tool [21] and its application to two case studies.

Once the behavioral specifications of the external services have been inferred, they can serve multiple purposes. For example, they can be used with (semi-)automated composition mechanisms, for selecting the services that fulfill in the best way the functional requirements of the composite service. Moreover, they can be translated into verifiable run-time assertions, which can be monitored while the system is operating, to check if the external services behave as expected, i.e., to check if the service providers meet the obligations they signed in the SLAs.

The rest of this paper is structured as follows. Section 2 provides background material on LTSs. Section 3 introduces the running example used to show how our approach works. Section 4 presents our formal models for service compositions and their interface specification. Section 5 presents the interface decomposition problem, illustrates our technique to solve it, and shows its correctness. Section 6 discusses the application of our approach to two case studies. Section 7 reviews the related work and Section 8 concludes the paper, outlining future research directions.

2. PRELIMINARIES

We use labeled transition systems (LTSs) to model the behavior of service compositions, the global specifications of the environment with which a composite service interacts, and the behavioral interfaces of the individual services. In the rest of this section, we formally define LTSs and the operations that can be performed over them.

Labeled Transition Systems

Let $\text{Act}$ be the universal set of observable actions and let $\tau$ denote an internal action that cannot be observed by the environment of a component. Let $\pi$ denote a special error state, which models safety violations in the associated transition system. A Labeled Transition System $M$ is a 4-tuple $(Q, A, \delta, q_0)$ where $Q$ is a finite non-empty set of states; $A = \alpha M \cup \{\tau\}$, with $\alpha M \subseteq \text{Act}$ is the actions alphabet; $\delta \subseteq Q \times A \times Q$ is a transition relation; $q_0 \in Q$ is the initial state. Moreover, let $\Pi$ denote a special LTS, which models safety violations in the associated $M$. Let $\tau$ contains

$\exists$ the initial state. Moreover, let $\Pi$ denote a special LTS, which models safety violations in the associated

$M$. An LTS $M = (Q, A, \delta, q_0)$ is complete with a sink state and the transitions leading~to it; the resulting LTS is denoted as $\hat{M}$. Formally, given an LTS $M = (Q, \alpha M \cup \{\tau\}, \delta, q_0)$, its complete-by-

construction variant is $\hat{M} = (Q \cup \{\hat{q}\}, \alpha \hat{M} \cup \{\tau\}, \delta, q_0)$, where $\alpha \hat{M} = \alpha M, \delta' = \delta \cup \{(q, a, \hat{q}) \mid a \in \alpha M\} \cup \{(q, a, q) \mid a \in \alpha M \land \neg 3q' \in Q \mid (q, a, q') \in \delta\}$.

For an LTS $M = (Q, A, \delta, q_0)$, there is a path $\sigma$ from state $q$ to state $q''$, with $q, q'' \in Q$, if there exists a set of states $\{q_1, \ldots, q_n\} \subseteq Q$ and a sequence of actions $\sigma_1, \ldots, a_{n-1}$, with each $a_i \in A$, such that $q = q_0 \land q'_i = q_i \land q_i \land 1 \leq i \leq n - 1, (q_i, a_i, q_{i+1}) \in \delta$. The sequence of actions $\sigma_1, \ldots, a_{n-1}$, where the $\tau$-transitions are ignored, is called the trace defined by the path $\sigma$. A trace of an LTS $M$ is a trace defined by a path that originates in the initial state; i.e., it is a finite sequence of observable actions that label the transitions that $M$ can perform starting at its initial state. The set of traces of $M$ is denoted as $\text{Tr}(M)$. For an LTS $M$, $\text{errTr}(M) \subseteq \text{Tr}(M)$ is the set of traces $\{t \in \text{Tr}(M) \mid \exists$ a path $\sigma$ from $q_0$ to $\pi$ and $t$ is defined by $\sigma\}$; $\text{errTr}(M)$ is called the set of error traces of $M$. Furthermore, given a trace $t$ and a set $\mathcal{A} \subseteq \text{Act}$, the expression $(t \upharpoonright \mathcal{A})$ denotes the trace obtained from $t$ by removing all occurrences of actions $a \not\in \mathcal{A}$; "$\upharpoonright$" is the restriction operator for traces.

In some cases, it might be useful to explicitly indicate that an LTS has the error state $\pi$, reachable from the initial state. For an LTS $M = (Q, A, \delta, q_0)$, we use the notation $M_\pi$ iff $\pi \in Q$ and $\text{errTr}(M) \neq \emptyset$. This notation can be combined with the one denoting the completion-by-construct, as in $M_\pi$, to identify an LTS that is complete and that contains the error state (reachable from the initial state).

Operators

Let $M = (Q, A, \delta, q_0)$ and $M' = (Q', A', \delta', q_0')$, with $q_0' \neq \pi$. $M$ transits into $M'$ with action $a$, denoted $M \overset{a}{\rightarrow} M'$, if $(q_0, a, q_0') \in \delta'$, with $Q' = Q', A' = A'$, $\delta'$. Moreover, we say that $M$ transits into $\Pi$ with action $a$, $M \overset{a}{\rightarrow} \Pi$, if $(q_0, a, \pi) \in \delta'$.

The interface operator $\uparrow$ is used to make unobservable some actions of an LTS. Given an LTS $M = (Q, A, \delta, q_0)$ and a set of observable actions $\mathcal{A} \subseteq \text{Act}$, $M \uparrow \mathcal{A}$ is defined as follows. If $M = \Pi$, $M \uparrow \mathcal{A} = \Pi$. For $M \neq \Pi$, $M \uparrow \mathcal{A} = (Q, (\alpha M \cap \mathcal{A}) \cup \{\tau\}, q_0', \delta')$, where $\delta'$ is described by the rules shown in Fig. 1a. The semantics of this operator ensures that $\text{errTr}(M) \neq \emptyset$ iff $\text{errTr}(M \uparrow \mathcal{A}) \neq \emptyset$.

Two LTSs can be combined by means of the parallel composition "$\parallel$" operator, which is commutative and associative. Given two LTSs $M_1 = (Q_1, A_1, \delta_1, q_0)$ and $M_2 = (Q_2, A_2, \delta_2, q_0)$, the parallel composition $M_1 \parallel M_2$ is defined as follows. If either $M_1 = \Pi$ or $M_2 = \Pi$, then $M_1 \parallel M_2 = \Pi$. Otherwise, $M_1 \parallel M_2$ is an LTS $M = (Q, A, \delta, q_0)$, where $Q = Q_1 \times Q_2, q_0 = (q_0', q_0')$. $A = A_1 \cup A_2$ and $\delta$ is described by the rules shown in Fig. 1b. The traces of a parallel composition are defined as follows: $\text{Tr}(M_1 \parallel M_2) = \{t \mid (t \upharpoonright \alpha M_1) \in \text{Tr}(M_1) \land (t \upharpoonright \alpha M_2) \in \text{Tr}(M_2) \land t \in (\alpha M_1 \cup \alpha M_2)^\ast\}$. As for error traces, a parallel composition has an error trace if at least one of its components has an error trace. In symbols: $\text{errTr}(M_1 \parallel M_2) = \{t \in \text{Tr}(M_1 \parallel M_2) \mid (t \upharpoonright \alpha M_1) \in \text{errTr}(M_1) \lor (t \upharpoonright \alpha M_2) \in \text{errTr}(M_2)\}$.

Safety Properties

A safety property can be specified as a deterministic LTS that contains no $\pi$ state. The set of traces $\text{Tr}(P)$ of a property $P$ defines the set of acceptable behaviors over $\alpha P$. An LTS $M$ satisfies $P$, denoted as $M \models P$ iff $\text{Tr}(M \uparrow \alpha P) \subseteq \text{Tr}(P)$. For a property LTS $P$ we can define the
error LTS $P_{err}$ as follows: given $P = (Q, A, P, \delta, q_0)$, $P_{err} = (Q \cup \{\pi\}, A_{P_{err}}, \delta', q_0)$, where $A_{P_{err}} = A \setminus \{q, a, \pi\} \setminus \{(q, a, \tau), (q, a, q')\}$. Note that the error LTS is complete by construction\(^1\).

Let $M$ be an LTS such that $err\text{Tr}(M) = \emptyset$. We detect possible violations of a property $P$ by the component $M$ by computing $M \parallel P_{err}$. As shown in [8], the execution of $M$ leads to a violation of a property $P$ iff $err\text{Tr}(M \parallel P_{err}) \neq \emptyset$, i.e., iff the $\pi$ state is reachable in $M \parallel P_{err}$.

3. EXAMPLE

Our running example is a simplified version of the Car Rental Agency one presented in [5]; we call it Simple Car Rental (SCR). The example illustrates a service composition that runs at a car rental office branch. The composite service interacts with a Car Broker (CB) service, which controls the operations of the branch; with a User Interaction (UI) service, through which customers can make car rental requests; with a Car Information (CI) service, which maintains a database of cars availability and allocates cars to customers; with a Car Parking Sensor (CPS) service, which exposes as a service the sensor that senses cars as they are driven in or out of the parking lot of the branch. The workflow of the composite service is sketched in Fig. 2 using the notation presented in [5]: boxes with a right arrow correspond to onMessage events, while the ones with two opposite arrows indicate an invoke activity.

The SCR service starts when it receives the startRental message from the CB service. It then enters an infinite loop; at each iteration it can receive one of the following messages:

- **findCar**. A customer requests to rent a car; the SCR service checks the availability of a car by invoking the lookupCar operation on the CI service. The lookupCar operation returns its result — which can be either a negative answer or an identifier corresponding to the digital key to access the car — in the result variable, which is then passed as parameter to the findCarCB operation, a callback invoked on the UI service.

- **carEnter** and **carExit**. These two messages are sent out by the CPS service when a car enters (respectively, exits) the parking lot. The process reacts to this information by updating the cars database, invoking, respectively, the markAvailable and markUnavailable operations on the CI service.

- **stopRental**. The CB service stops the operations of the branch, terminating also the composite service.

To keep the example compact, we assume that a single car is available in the branch, and that the CI service is accessed only by the SCR service instance running in the branch.

The correct execution of the SCR service depends on the functionalities provided by the CI and CPS services. Therefore, in the next two sections we show the application of our interface decomposition technique to derive the behavioral interfaces of these two services.

4. SERVICE COMPOSITION AND GLOBAL INTERFACE SPECIFICATION MODELS

In this section we present the formal model of service compositions and describe how we can infer the global interface specification of the environment (i.e., the set of partner services) with which a composite service interacts. We refer the reader to Fig. 3, for mapping symbols onto components.

4.1 Service Composition

A service composition $C$ interacts with a set of external services denoted as $E = \{E_1, \ldots, E_n\}$. Each service $E_i \in E$ makes available a set of operations $O_i = \{o_{i1}, \ldots, o_{in}\}$, which constitute its syntactical interface. We assume that $\forall i,j, 1 \leq i \neq j \leq n, O_i \cap O_j = \emptyset$, since each operation can be unambiguously identified by its name combined with the name of the service it belongs to (e.g., by means of the interface and service elements of a WSDL 2.0 description).

We assume that service compositions are implemented as BPEL processes, which can be formalized in terms of labeled transition systems as shown in [11], with tools such as WS-Engineer [10]. For a service $C$, let $M_C$ be the corresponding LTS.

The safety requirements on the behavior of the composite service $C$, when it interacts with the external services $E$, can be modeled by a property LTS $P$. This LTS can be synthesized, for example, from a specification in a temporal logic formalism such as LTL or Fluent LTL [13]. Note that the property $P$ implicitly defines the unwanted behaviors, by means of the corresponding error LTS $P_{err}$.
Modeling the Running Example

In the example, we are interested in the environment constituted by the services CI and CPS, so we have \( E = \{CI, CPS\}, \) \( O^{CI} = \{\text{markAvailable}, \text{markUnavailable}, \text{lookupCar}\} \) and \( O^{CPS} = \{\text{carEnter}, \text{carExit}\} \).

In the rest of this paper, we use the FSP textual notation [21] to compactly represent LTS models. In FSP, identifiers beginning with a lowercase letter denote actions while identifiers beginning with an uppercase letter denote processes (states in the underlying LTS); the symbol “\( \rightarrow \)” denotes the action prefix operator, while the vertical bar “\( | \)” denotes the choice operator. The following code snippet corresponds to the LTS model of the SCR service:

```plaintext
range KEY = 0..1 // (0 means car not available)
SCR = (startRental -> Main, 
Main = (findCar -> lookupCar[result=KEY] -> Main 
| carExit -> markUnavailable -> Main 
| carEnter -> markAvailable -> Main 
| stopRental -> END).
```

Note that each operation invoked on the SCR service and on its partner services is modeled as an action. Moreover, since the variable `result` ranges over the domain `KEY`, the `lookupCar` action is internally represented as `lookupCar[0]` and `lookupCar[1]`; the same applies to `findCarCB`.

Service Behavior

The expected behavior of the SCR service is expressed by the following requirement: “If the car enters the parking lot, and it does not exit until a customer requests it for renting, then this request should not return a negative answer.” This requirement can be formalized in Fluent LTL as the formula G(CarIn ⇒ ψ), where CarIn is a fluent that changes value when the car is in the parking lot, and it is defined as CarIn = (carEnter, carExit) initially False; \( ψ \) is the auxiliary formula findCar ⇒ (¬findCarCB[0] W findCarCB[1]). Here G and W are, respectively, the LTL temporal operators “globally” and “weak until”. This Fluent LTL formula represents a safety property and thus can be translated automatically [13] into an (error) LTS model, whose textual description is shown below:

```plaintext
Perr = Q0,
Q0 = (findCar, findCarCB[0..1] | Q0 
| carEnter -> Q1),
Q1 = (carExit -> Q0 
| carEnter, findCarCB[0..1] | Q1 
| findCar -> Q2),
Q2 = (findCarCB[0] | ERROR 
| findCarCB[1] | Q1 
| carEnter, findCar | Q2 
| carExit | Q3),
Q3 = (findCarCB[0] | ERROR 
| findCarCB[1] | Q1 
| carEnter | Q2 
| carExit, findCar | Q3).
```

4.2 Global Interface Specification

In this work, we want to characterize the global expectations from E in order for C to fulfill its requirement P; i.e., we want to infer the global interface specification of the environment E with which C interacts. By following the technique introduced in [14] and summarized below, we can determine the global interface specification by computing the LTS \( I_e \), with \( I_e = \text{BUILDINTERFACE}(M_e \parallel P_{err}), O) \), where \( O = \bigcup_{i=1}^n O^i \).

The pseudo-code of function \( \text{BUILDINTERFACE} \) is shown in Fig. 4. The function receives as first parameter an LTS model; the actual parameter that is passed \( (M_e \parallel P_{err}) \) contains all the traces that violate the property \( P \). The actual value of the second parameter actions, is the set of all the operations provided by the external services and is used on line 2 as an operand of the interface operator, to get the LTS named gen_interface. This LTS is further processed with a special determinization step (line 3), provided internally by LTSA. This determinization step performs \( τ \)-elimination and subset construction but, unlike standard automata theory algorithms, it handles in a special way the \( π \) state. Since during the subset construction the states of the deterministic LTS correspond to set of states of the original, non-deterministic LTS, if any of the states in the set is \( π \), then the entire set becomes a \( π \) state in the deterministic LTS. This means that a trace that non-deterministically may or may not lead to the error state has to be considered an error trace. In practical terms, it means that performing a certain sequence of actions on the external services does not guarantee that the service composition will not reach an error state. Subsequently, the LTS gen_interface is completed (line 4) with a sink state and the transitions leading to it, by invoking an auxiliary function. The missing transitions in the original LTS represent behaviors of the external services that are never exercised by the service composition; with the completion, they are made sink behaviors and thus no restriction is imposed on them.

![Figure 3: Notation and general model of the service interface decomposition problem](image)

![Figure 4: Pseudo-code of the \text{BUILDINTERFACE} function](image)
The notation used for the resulting LTS, ̂I, denotes that it contains the error state (deriving from the error LTS ̂Perr) and that it has been completed with a sink state. Hereafter, we use the notation ̂I to refer to the variant of ̂I that does not contain the error state, without the transitions leading to it. In symbols, given ̂I = (Q ∪ {π}, α̂I, δ̂I, q0), ̂I = (Q, αI, δI, q0), where αI = α̂I and δI = δ̂I \{(q, a, π) | a ∈ αI\}.

**Application to the Running Example**

The first parameter passed to the BuildInterface function is (SCR || Perr). As for the second parameter, the list of actions passed to the function is composed by markAvailable, markUnavailable, lookupCar[0] and lookupCar[1] (from CI), and by carEnter and carExit (from CPS). The resulting interface ̂I is defined as follows:

| ̂Ipi = Q0, |
| Q0 = ({lookupCar[0..1]} -> Q0 |
| | {carExit -> Q1 |
| | | {carEnter -> Q2 |
| | | | {markUnavailable, markAvailable} -> SINK), |
| Q1 = (markUnavailable -> Q0 |
| | | {carExit, carEnter, markAvailable, lookupCar[KEY]}) -> SINK), |
| Q2 = (markAvailable -> Q3 |
| | | {carExit, carEnter, markUnavailable, lookupCar[KEY]}) -> SINK), |
| Q3 = (lookupCar[0] -> ERROR |
| | | {carExit -> Q1 |
| | | | {carEnter -> Q2 |
| | | | | {lookUpCar[1]} -> Q3 |
| | | | | {markUnavailable, markAvailable} -> SINK), |
| SINK = {{carExit, carEnter, markUnavailable, markAvailable, lookupCar[KEY]}} -> SINK). |

### 5. Decomposing Interface Specifications

The method described in Section 4.2 computes the global interface specification of a service composition, i.e., the behavior that its partner services, considered as a whole, should manifest in order for the composite service to fulfill its requirements specification. However, this “centralized” solution is not realistic for the domain of service-based applications, since each service is operated independently by its own provider, and has no knowledge of the other services with which its client service (i.e., a composite service) interacts. Therefore, we argue it is necessary to define a more “distributed” approach, which generates the individual behavioral interfaces for the partner services.

To this end, we define the interface decomposition problem as follows (refer to Fig. 3 for mapping symbols onto components): given a service composition C, which interacts with a set of external services E = {E1, ..., En} whose behavior, as a whole, is represented by I, we decompose I into interface specifications for the individual partner services, denoted as ̂I, 1 ≤ i ≤ |E|.

The individual interface specifications obtained by means of the interface decomposition technique should guarantee that the composite service fulfills its requirement specification. This correctness requirement can be formally stated as (|E| ̂I) \ |= P.

In the rest of this section, we illustrate our technique for decomposing interface specifications and show its application to the running example. We first present a basic approach to the problem and observe that it generates over-constraining interfaces. Subsequently, we propose our heuristic-based technique, which generates less constraining, but still correct behavioral interfaces. We conclude by discussing some approaches for the validation of the decomposition technique, as well as its shortcomings.

#### 5.1 Basic Decomposition Approach

A first approach to the problem of interface decomposition can be based on the intuition that each external service can contribute to the global interface specification only through the operations it provides. Formally, this means the interface specification ̂Ii, of an external service Ei, can be computed as ̂Ii = BuildInterface(Ii, Oi).

Note that ̂Ii contains the error state; as done for the case of the global interface specification, we use the notation ̂I for the variant of ̂I that contains neither the error state nor the transitions leading to it.

However, simple experimentation with this technique reveals that such an approach generates interfaces that are too restrictive. For example, its application to the running example generates the following interface specifications.

For the CI service, we restrict the global interface specification over the alphabet {markUnavailable, markAvailable, lookupCar[0], lookupCar[1]}. The resulting LTS is:

| CI = Q0, |
| Q0 = (lookupCar[0..1], markUnavailable) -> Q0 |
| | {markAvailable -> Q1 |
| | | {lookupCar[KEY]}) -> SINK), |
| Q1 = (lookupCar[0] -> ERROR |
| | {carExit -> Q1 |
| | | {carEnter -> Q2 |
| | | | {lookUpCar[1]} -> Q3 |
| | | | | {markUnavailable, markAvailable} -> SINK), |
| SINK = {{carExit, carEnter, markUnavailable, markAvailable, lookupCar[KEY]}} -> SINK). |

It states that after a markAvailable operation, when the computation is in state Q1, the lookupCar operation will return successfully (i.e., a value different from 0). Essentially, state Q1 denotes the fact that the car is in the parking lot.

As for the CPS service, the global interface specification is restricted over the alphabet {carEnter, carExit}. The resulting LTS is:

| CPS = Q0, |
| Q0 = (carEnter -> ERROR |
| | {carExit -> Q0 |
| | | {lookupCar[1], markAvailable}) -> Q1). |

This interface is too restrictive, since it disallows a car from ever entering the parking lot. Furthermore, considering that according to the definition of the fluent CarIn in Section 3, the car is initially out of the parking, this interface in practice blocks any behavior from the car.

In fact, we can make a stronger observation about the individual interfaces built in this way:

**Proposition 1.** Let ̂Ii = BuildInterface(̂I, Oi), and ̂Ii = BuildInterface(MC || Perr, Oi). Then ̂Ii and ̂Ii are isomorphic.

**Proof.** By construction, function BuildInterface generates a canonical deterministic LTS whose error traces are equal to the error traces of its first argument projected to the alphabet represented by its second argument [14]. Since ̂I = BuildInterface((MC || Perr), O), it follows
that \(\text{errTr}(\hat{I}_s) = \text{errTr}(\{MC \mid \text{Perr} \uparrow O\})\). In a similar way, 
\(\text{errTr}(\hat{I}_s) = \text{errTr}(\{E \mid \text{Perr} \uparrow O\})\). From these two statements, we 
derive that \(\text{errTr}(\hat{I}_s) = \text{errTr}(\{MC \mid \text{Perr} \uparrow O\})\). Since \(O_i \subseteq O\), we conclude that \(\text{errTr}(\hat{I}_s) = \text{errTr}(\{MC \mid \text{Perr} \uparrow O\})\). Additionally, \(\hat{I}_s = \text{BuildInterface}(\{MC \mid \text{Perr} \uparrow O\})\) implies that \(\text{errTr}(\hat{I}_s) = \text{errTr}(\{MC \mid \text{Perr} \uparrow O\})\). Since the error traces of \(\hat{I}_s\) and \(\hat{I}_s\) are equal, we conclude that the canonical representations \(\hat{I}_s\) and \(\hat{I}_s\) generated 
by function \text{BuildInterface}, are isomorphic, and therefore so are \(\hat{I}_s\) and \(\hat{I}_s\).

As a result, each interface that we compute in this fashion is sufficient by itself, to guarantee the global property on 
the system, meaning that \(\forall i, (\hat{I}_i \parallel M_i) \models P\), which implies that 
\(\{i = 1 \mid \hat{I}_s\} \parallel M_i \models P\).

However, imposing such interfaces would be overly constraining. Moreover, a solution that assigns the entire 
responsibility for achieving the global property to every single service is not desirable. Ideally, we would like a solution 
that distributes the responsibility to the partner services in a way that allows as much participation from each service as possible in the behavior of the service composition. To this end, in the next section we propose a heuristic that avoids 
to unnecessarily constrain the interface of partner services that cannot lead to error behaviors of the system.

### 5.2 Heuristic-based Decomposition Technique

The heuristic we propose to use is based on inspecting the actions that label the transitions that lead to the error state in the global interface specification. It may be the case that none of these actions corresponds to one of the operations provided by the partner service (hereafter referred to as \(E_i\)) for which we want to compute the behavioral interface. This means that service \(E_i\) will never cause an error behavior in the system constituted by the composite service and its 
partner services. In this case, the behavioral interface of \(E_i\) can be obtained by decomposing a simplified model of 
the global interface specification, which does not include the error behaviors that are not directly ascribable to \(E_i\).

More formally, for a service \(E_i\) with actions \(O_i\), given a global interface specification \(\hat{I}_s = (\langle Q, \alpha, I, \delta, \delta', q_0 \rangle)\), the 
heuristic builds an auxiliary global interface specification, denoted with \(\hat{I}_{aux}(i)\). This heuristic-based, auxiliary global interface 
specification is computed as \(\hat{I}_{aux}(i) = (\langle Q, \alpha, I, \delta', q_0 \rangle)\), where \(\delta' = \delta \setminus \{(q, a, \pi) \mid a \notin O_i\}\). The definition of \(\delta'\) shows that the 
heuristic removes the transitions to the error state labeled with actions (operations) not provided by \(E_i\). 
Note that as a result of removing such transitions, \(\hat{I}_{aux}(i)\) may not be complete; note also the error state may be removed in case the error states were ascribable only to the other 
services different from \(E_i\). The interface specification of the service \(E_i\), denoted with \(\hat{I}_s\), can then be computed as \(\hat{I}_s = \text{BuildInterface}(\hat{I}_{aux}(i), O_i)\).

**Correctness**

Before showing that this technique is a correct solution of the interface decomposition problem, we introduce and prove some helpful propositions.

**Proposition 2.** Given \(\hat{I}_s\) and \(\hat{I}_s\) defined as above, the relation 
\(\text{errTr}(\{E \mid \hat{I}_s\}) \supseteq \text{errTr}(\hat{I}_s)\) holds.

**Proof.** The proof is by contradiction. Suppose there is a 
trace \(t\), such that \(t \in \text{errTr}(\hat{I}_s)\) and that \(t \notin \text{errTr}(\{E \mid \hat{I}_s\})\). 
Let \(a\) be the last action in \(t\), and \((q, a, q')\) the corresponding 
transition that leads to the error state in \(\hat{I}_s\). Since there must exist a \(k\) such that \(a, E_i\), we know that transition 
\((q, a, q')\) will not be removed from \(I_{aux}(k)\). From the 
semantics of the interface operator, we can then conclude that 
\((t \parallel O_i) \in \text{errTr}(\hat{I}_s)\). For all \(i, \hat{I}_s\) is complete, we also know that \(t \in \text{errTr}(\{E \mid \hat{I}_s\})\). But since \(t\) leads to the 
error state with at least one component of this, we conclude that 
\(t \in \text{errTr}(\{E \mid \hat{I}_s\})\), which is a contradiction.

**Proposition 3.** Given \(I\) and \(\hat{I}\), defined as above, the relation 
\(\text{errTr}(\{E \mid \hat{I}\}) \subseteq \text{errTr}(I)\) holds.

**Proof.** Consider the set \(O\) of all the operations made 
available by the external services; let \(O^*\) represent its Kleene 
closure. Similarly, let \(O'\) be the Kleene closure of the set 
of operations provided by an individual external service \(E_i\). 
By construction, \(\hat{I}\) is obtained from \(I\) by 
removing the error state and the transitions leading to it. 
Since no trace of the \(I\) interface leads to the 
error state, we know that \(\text{errTr}(\hat{I}) = O' \setminus \text{errTr}(\hat{I})\); similarly, \(\forall i, 1 \leq i \leq |E|, \text{errTr}(\hat{I}_i) = O' \setminus \text{errTr}(\hat{I}_i)\). Moreover, we 
know that a composite process has an error trace, if at least 
one of its constituent processes has an error trace. In symbols:

\[
\text{errTr}(\{E \mid \hat{I}_i\}) = \{t \in \text{errTr}(\{E \mid \hat{I}_i\}) \mid (t \parallel O_i) \in \text{errTr}(\hat{I}_i) \}
\]

Hence:

\[
O^* \setminus \text{errTr}(\{E \mid \hat{I}_i\}) = \{t \in \text{errTr}(\{E \mid \hat{I}_i\}) \mid (t \parallel O_i) \notin \text{errTr}(\hat{I}_i)\}
\]

Since \(\text{errTr}(\{E \mid \hat{I}_i\}) \supseteq \text{errTr}(\hat{I}_i)\) holds from Proposition 2, 
\(O^* \setminus \text{errTr}(\{E \mid \hat{I}_i\}) \subseteq O' \setminus \text{errTr}(\hat{I}_i)\) also holds. 
Hence \(\text{errTr}(\{E \mid \hat{I}\}) \subseteq \text{errTr}(\hat{I})\).
Application to the Running Example

By analyzing the global interface specification $I_{pi}$ showed in Section 4.2, we notice that the error state can be reached by executing, in state $Q_3$, the transition labeled with $\text{lookupCar}[0..1]$, which is an operation provided by the CI service. The heuristic described above can then be applied to compute the interface for the CPS service.

We first create a refined model of the global interface, by removing the transitions that lead to the error state and that are not labeled with actions belonging to the alphabet of the CPS service:

| $I_{pi,\text{cps}}$ = $Q_0$, | $Q_0$ = ($\text{lookupCar}[0..1] \rightarrow Q_0$ | ($\text{carExit} \rightarrow Q_1$ | ($\text{carEnter} \rightarrow Q_2$ | ($\text{markUnavailable}$, markAvailable) $\rightarrow$ SINK), |
| $Q_1$ = (markUnavailable $\rightarrow Q_0$ | ($\text{carExit}, \text{carEnter}$, markAvailable, $\text{lookupCar}[\text{KEY}]$) $\rightarrow$ SINK), |
| $Q_2$ = (markAvailable $\rightarrow Q_3$ | ($\text{carExit}, \text{carEnter}$, markUnavailable, $\text{lookupCar}[\text{KEY}]$) $\rightarrow$ SINK), |
| $Q_3$ = ($\text{carExit} \rightarrow Q_1$ | ($\text{carEnter} \rightarrow Q_2$ | ($\text{lookupCar}[1]$) $\rightarrow$ Q3 | ($\text{markUnavailable}$, markAvailable) $\rightarrow$ SINK), |
| SINK = ($\text{carExit}, \text{carEnter}$, markUnavailable, markAvailable, $\text{lookupCar}[\text{KEY}]$) $\rightarrow$ SINK). |

Next, the global interface specification is restricted over the alphabet \{\text{carEnter}, \text{carExit}\}; the resulting LTS is:

| CPS = $Q_0$, | $Q_0$ = \{(\text{carEnter}, \text{carExit}) \rightarrow Q_0\). |

As expected, this new interface, obtained for the CPS service with the application of the heuristic, allows for more behaviors than the one computed with the basic technique. More specifically, in this case the interface represents the universal interface of service CPS, i.e., the interface that allows any of its operations. Since the error behaviors of the system are prevented by the interface of the other service (CI), there is no need to constrain the interface of CPS.

As for the interface specification of service CI, the application of the heuristic does not affect its generation, i.e., it coincides with the one shown in Section 5.1.

5.3 Discussion

Validation of the Generated Interfaces

Although the definition of the interface decomposition problem includes a correctness requirement, which guarantees that the generated interfaces will not lead the system into the error state, this is not enough to characterize the quality of the generated interfaces. Ideally, they should be validated by using some kind of oracle, such as descriptions of good and bad behaviors, usually defined by domain experts or encoded in a certain model.

By decomposing this interface to compute $\hat{I}_1$ and $\hat{I}_2$, we notice that our heuristic blocks $E_1$ completely (no operation can be performed on it), while generates the universal interface for $E_2$.

More generally, our heuristic may block some good behaviors of the individual services, which instead could be safely allowed. This may happen because an operation of a service that directly leads to the error state, which is the one considered by our heuristic, may be actually triggered by an operation of another service. In the example above, the transition $c \rightarrow \text{ERROR}$ is actually performed only after the transition $a \rightarrow S_2$ occurs; another heuristic could then allow $E_1$ to perform $c$, while the interface of $E_2$ could mandate the execution of $b$ and $a$ in this order.

6. Evaluation

The interface specifications decomposition technique has been implemented in the LTSA tool; here we report about the evaluation of our approach on two case studies. Each case study consisted of a service composition in the form of a BPEL process.
of a BPEL process, of the syntactical interfaces (WSDL description) of the partner services of the composition, and of an informal description of the requirements that the composition had to fulfill.

The BPEL processes have been translated into the input format of the LTSA tool by means of WS-Engineer; the requirements have been first formalized in a temporal logic and then translated into an LTS description. The experiments have been executed on a computer running Apple Mac OS X 10.6.4 with a 2.16 GHz Intel Core 2 Duo processor and 2 GiB of memory.

### 6.1 Car Rental (full version)

This case study is the full-fledged version of the example described in Section 3, with which it also shares the same requirements specification. The main difference lies in a fine-grained description of the BPEL process, which leads to more refined, and sometimes verbose, interface descriptions. For example, the two single transitions `lookupCar[0..1]` that in the running example correspond to invoking the `lookupCar` operation of the CI service and receiving, as output parameter, either 0 or 1, are expanded in a sequence of four operations: `(cr_ci_invoke_lookupcar, cr_ci_receive_lookupcar, cr_ckr.condition.read.{false, true} -> Q0, cr_ckr.condition.read.true -> Q3).

If we consider this kind of expansion, we easily conclude that the interfaces generated are equivalent to, but bigger — in term of size of the model — than the ones built obtained for the running example. For example, the interface of the CI service is the following:

- **CIS = Q0**, where:
  - Q0 = (cr_ci_invoke_markcaravailable -> Q0, |cr_ci_invoke_lookupcar -> Q2, |cr_ci_invoke_markcaravailable -> Q3),
  - Q2 = (cr_ci_receive_lookupcar -> Q4),
  - Q3 = (cr_ci_invoke_markcaravailable -> Q0, |cr_ci_invoke_markcaravailable -> Q3, |cr_ci_invoke_lookupcar -> Q5),
  - Q4 = (cr_ckr.condition.read.{false, true} -> Q0),
  - Q5 = (cr_ckr.condition.read.true -> ERROR),
  - Q6 = (cr_ckr.condition.read.false -> Q3).

The interface of the CPS service, as before, remains the universal interface.

In this example, the LTS model of the service composition contains 16 states and 20 transitions; the global interface specification contains 9 states and 22 transitions, and was built in 70 ms; the interface specifications of the services CI and CPS were built, respectively in 90 ms and 75 ms.

### Validation against Original Specifications

The original example definition [5] contained a set of logical specifications of the behavior expected from the external services, manually written by the authors of the paper. We consider these specifications as a possible oracle for evaluating how well our technique performs and thus we compared these specifications with the ones generated by the tool.

The specification of the CI service was “If the car is marked as unavailable until a `lookupCar` operation is invoked, then the `lookupCar` operation should return successfully”. It is clear that this behavior is captured by the interface specification generated for the CI service.

For the CPS service, the specification was “between two events signaling that the car exits the parking lot, an event signaling the entrance for the same car must occur”; basically it states the two events “car enter” and “car exit” should alternate. The interface specification obtained for this service, however, is the universal interface. In our opinion, this result is still correct, even if less useful, because the CPS service cannot be responsible for violations of the expected requirement.

In LTSA we have also implemented the possibility to search for and analyze lost behaviors, by checking \( Tr(I) \subseteq Tr(I_{\text{crit}}) \). This check failed, revealing one lost behavior whose trace is:

- cr_cr_invoke_markcaravailable, cr_cr_invoke_lookupcar, cr_cr.receive_lookupcar, cr_cyr.condition.read.false.

This trace can be interpreted as “if the car is marked as available in the parking lot, then a request for the car will return a negative result”, which is an incorrect behavior. Note that this behavior is disallowed by the structure of the composite service, since `cr_cr_invoke_markcaravailable` will never be executed as the first action. Therefore we can safely state that this behavior has been added to the global interface specification through the completion with the sink state; it will never occur in the real system. This is the reason for which it is also missing from the interfaces derived for the partner services.

### 6.2 Order Booking

This case study has been taken from the sample processes distributed with the Oracle SOA Suite 10gR3. It consists of a process that is started when a customer places an order from a client web application. The process first inserts the order information in a database through the `ERPService`, then it retrieves customer information by invoking the `CustomerService`. The process checks the customer’s credit card by invoking the `CreditService` and then determines if the order requires manual approval by invoking the `DecisionService (DS)`, which applies some business rules that take into account the status (platinum or not) of the customer. For orders that require manual approval, the process invokes the `requiresApproval` operation on the `Manager` Web service. When an order is approved, the process requests, in parallel, quotes from the suppliers, `SelectManufacturer`, and `RapidService`, and then selects the supplier that responded with the lower quote. Afterwards, a shipping method is chosen by checking the amount of the order. After updating the order status on the database through the `ERPService`, the project sends a confirmation email to the customer, by invoking the `EmailService`, and then terminates.

A possible requirement specifications for this composite service is: “if a platinum customer places an order, it must be automatically approved; otherwise it must be approved manually”. This specification indirectly requires a certain behavior of the DS service, which we picked as the service for which to compute the interface specification.

We translated this specification into a proper LTS and then applied the interface decomposition method based on the heuristic, to obtain the interface for the DS service. We omit its textual representation for space reasons but, in essence, it states that “if a platinum customer places an order, then the return value will not be manual approval, and equivalently, if a non-platinum customer places an order, then the return value will not be automatic approval”.

```plaintext
```
This specification matches the one informally described in the documentation of the example.

Since we were not interested in getting an individual interface specification for each of the other partner services, we generated an interface for them when considered as a whole and, as expected, we obtained the universal interface.

The LTS model of the composite service contains 80 states and 93 transitions; the global interface specification contains 29 states and 63 transitions, and was built in 76 ms; the interface specification of the DS service contains 6 states and 12 transitions, and was built in 82 ms. The interface for the rest of the components contains only one state, allowing all possible behaviors (i.e., it encodes the universal environment). A search for lost behaviors reveals two behaviors, which a manual inspection shows to be sink.

7. RELATED WORK

This work is closely related to the problem of synthesizing individual service behaviors from a choreography specification, such as conversation protocols [12], WS-CDL models [24], and collaboration diagrams [25]. These works define a projection operation that derives the implementations of the participating peers by filtering the global specification on the actions alphabet of each peer, which is similar in spirit to the basic decomposition approach described in Section 5.1; additionally, in [25], extra communication actions among the generated peers are added in case some behaviors may not be realizable in a distributed fashion. The difference with our work lies in the point of view adopted: the aforementioned works consider a superset of the possible behaviors and narrow it down to achieve the exact behavior dictated by the choreography specification. In our work, we view the global interface as the maximum behavior that could be allowed for the composition based on a property, and we generate a subset of the possible behaviors. Our process is driven by the error behaviors that have to be blocked; error traces guide us in the heuristic to assign to partner services the responsibility of blocking those behaviors.

Still related to the synthesis problem, reference [19] shows, in the context of verification of choreographies expressed in BPEL4CHOR, how a single participant of a choreography can be synthesized starting from the description of the choreography and from the BPEL models of the other participants. Besides the limitation of synthesizing at most one participant, this work makes the assumption that the BPEL models of the other participants are available; this assumption is unrealistic in the context of open-world services.

The problem of generating the interface of the environment of a system, given a property it should satisfy, has been originally dealt with in [14], in the context of model checking. However, the approach generates only the global interface, not the interfaces of the individual components of the system. Other work [9] describes a compositional reasoning approach for the verification of middleware-based software architecture descriptions. Given a graphical scenario of the architecture of a generic application in terms of Message Sequence Charts (MSCs), the approach tries to verify the global property by verifying local properties of the architectural components. This last step requires to decompose the global property into local properties; the decomposition is based on the analysis of the structure of the MSCs, which is similar to our heuristic that considers the structure of the global interface specification.

The use of a description of the system requirements to generate behavioral models of the system components is also common in the context of behavioral model synthesis. One approach [9] proposes to inductively synthesize the LTS models of each system component from a set of end-users scenarios, both positive and negative, in the form of MSCs. The approach operates at the stage of requirements, where users can interactively refine the scenario-based description by answering questions; in our work, we assume the requirements are fixed and thus rely on the accuracy of the specification to get expressive interfaces. The approach presented in [1] derives operational requirements (in the form of pre- and trigger-conditions) from goal models, using a combination of model checking, inductive learning and manual elaboration of scenarios; however, the approach does not support learning the operational requirements for an individual component of a system. In [17], behavioral models, in the form of Modal Transition Systems, are generated at the component level from a set of scenarios and property specifications. The algorithm assumes that domain variables are used for defining the pre- and post-conditions of component operations; however, for service components, pre- and post-conditions are usually not available. Another technique [26] constructs behavioral models (in the form of Modal Transition Systems) from both safety properties and scenario-based specifications; however, the models generated are at the system level, not at the component level.

While the main motivation behind this work is to decompose a global specification of a system to obtain the individual specifications of the system components, which would otherwise be unknown, other approaches perform decomposition of a global specification either to reduce the size of the model to verify — as in [18, 6], with the application of slicing — or to support compositional verification for systems that are not structured into parallel components [22].

Inferring the specifications of components is also a goal shared with program specification miners, such as Adabu [27] and GK-tail [20]. These approaches usually perform static analysis, code instrumentation and analysis of the execution traces to derive the usage patterns of components and thus need to access the code of the components for which you want to discover the specification. This latter step is not feasible in the domain of service-oriented computing. In the context of Web services, the Strawberry approach [3] derives the behavioral model of a service by analyzing its syntactical interface and applying a combination of graph synthesis, heuristics and testing. However, all these approaches consider the behavior of a single service (component) in isolation, while we are interested in discovering the behavioral interfaces of components that guarantee the requirements of the composite application.

The work in [15] presents a generic theoretical assumption-guarantee framework for adaptable systems that guarantees that adaptation-related changes do not affect the global invariant of the system. It assumes the availability of existing techniques to perform assumptions generation and system and property decomposition. The last one is a direction toward which this work can contribute.

8. CONCLUSION AND FUTURE WORK

The correct behavior of a service composition, with respect to its requirements specification, depends on a certain, expected behavior of its partner services. However, most of
the times the behavioral descriptions of the partner services are unknown. In this paper, we have presented our novel technique to automatically generating the behavioral interfaces of the partner services of a service composition, by decomposing the requirements specification of the composite services. We have formalized this problem, proposed a heuristic-based technique to solve it, implemented this technique in the LTSA tool and applied it to two case studies.

We plan to further develop and improve the technique presented in this paper. First, we will consider alternative heuristics, to address the limitations of the current one. For example, we want to assess precisely to which extent a partner service contributes to fulfill (or not) the global requirements. This is particularly important in the case in which multiple partner services have operations that could possibly lead to the error state. Secondly, we will support the refinement of the generated specifications, by extending the analysis of the counterexamples to filter missing behaviors (for example, by performing behavior realizability analysis as suggested in [25]). Last, we will add support for timed property specifications.

Acknowledgments

This work has been partially supported by the Swiss NSF projects no. 125337-CLAVOS and no. 125604; by the EU under the grant agreement no. EU-FP7-215483-S-Cube and the IDEAS-ERC grant agreement no. 227977-SMScom. The authors wish to thank Howard Foster for his promptly support with the WS-Engineer tool; Ivo Krka for his comments on an earlier version of the paper.

9. REFERENCES

[16] JavaPathFinder.
http://babelfish.arc.nasa.gov/trac/jpf.