An Interface Formalism for Web Services

Dirk Beyer   Arindam Chakrabarti   Thomas A. Henzinger
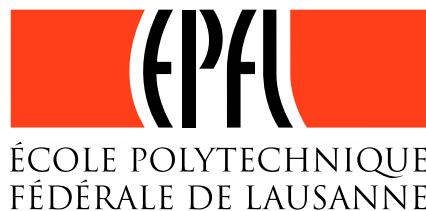
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

An Interface Formalism for Web Services [*] [†]

Dirk Beyer
EPFL, Lausanne, Switzerland

Arindam Chakrabarti
University of California, Berkeley, U.S.A.

Thomas A. Henzinger
EPFL, Lausanne, Switzerland

Abstract

Web application development using distributed components and web services presents new software integration challenges, because solutions often cross vendor, administrative, and other boundaries across which neither binary nor source code can be shared. We present a methodology that addresses this problem through a formalism for specifying and manipulating behavioral interfaces of multithreaded open software components that communicate with each other through method calls. An interface constrains both the implementation and the user of a web service to fulfill certain assumptions that are specified by the interface. Our methodology consists of three increasingly expressive classes of interfaces. Signature interfaces specify the methods that can be invoked by the user, together with parameters. Consistency interfaces add propositional constraints, enhancing signature interfaces with the ability to specify choice and causality. Protocol interfaces specify, in addition, temporal ordering constraints on method invocations. We provide approaches to check if two or more interfaces are compatible; if a web service can be safely substituted for another one; and if a web service satisfies a specification that represents a desired behavioral property.

Keywords: Web services, Web service interfaces, Web service compatibility, Web service substitutivity, Formal specification, Formal verification

1. Introduction

Component-based design for complex software systems has been an area of active interest for some time. Building web applications using distributed components or web services introduces special challenges. Conventional development of a software product is often done by a single vendor; and each developer has access to the entire source code or can use debugging tools on an executable built from all the software that his own contribution needs to interact with. In contrast, a web application often uses services offered by a number of different service providers, most of which do not disclose even their executable binaries, leave alone the source code; and the web application developer using these services has to rely solely on the disclosed documentation, which is usually informal, ambiguous, and often self-contradictory. In such a situation, *interface formalisms* provide a means for unambiguously describing and manipulating constraints under which independently developed software components can work properly together.

Static type systems used in programming languages constitute a simple interface formalism to avoid composition errors: a function's signature ensures, for example, that the function is only called with the correct number of parameters and that the parameters are of the correct type. Richer interface formalisms for software components have been proposed for communication protocols [3], timing requirements [4], and resource usage [2]. In the spirit of these interface theories, we present a formalism for *web service interfaces* which supports a two-player game view of multithreaded open software components. This view makes the formalism applicable to scenarios where the details of the concrete implementation of a service, as well as the details of the environment of the service, are not known at design and analysis time. However, an interface constrains both the implementation and the environment of the service with assumptions that are made by the designer of the service. This enables our approach to be used early in the web software design cycle, and by service developers who do not have access to the source or binary code of partner services that form part of their environment. A preliminary version of our interfaces [1] did not support the two-player view, permitting analysis only for closed systems, where the environment is known.

In contrast to the formal verification of web service *implementations* [6, 7, 5, 8], we explicitly propose to specify and verify *interfaces* of web services, which has a much better chance of succeeding in practice, because interfaces are usually less complex than the corresponding implementations. Indeed, good interface design requires that an interface exposes all information needed to use the service properly, but no more. In this spirit, we present three interface description languages of successively increasing expressiveness. Using these languages, we can automatically check if two or more interfaces are *compatible* (i.e., if they satisfy each other's assumptions), and if one interface can be safely *substituted* for another one in every design. In addition to this, we introduce *specifications* to describe desired propositional and temporal properties of web services, and we provide means for checking whether an interface satisfies them.

The first formalism, called *signature interfaces*, exposes only the names and types of web methods provided by the service, and the names and types of web methods that the interface expects to be provided by the environment. For example, a signature interface may offer the web method `ProcPay`, with the two possible return values `OK` and `FAIL`, and it may itself rely on certain other methods and return values. The second formalism, called *consistency interfaces*, adds propositional constraints representing choice and causality to signature interfaces; for example, it may prohibit having both an invocation of `ProcPay` with return value `FAIL` and invocation of `ShipItem` with return value `OK` in the same conversation. The third and richest formalism, called *protocol interfaces*, adds temporal ordering constraints to consistency interfaces; for example, it may disallow conversations where `ShipItem` is invoked with return value `OK` before `ProcPay` is invoked with return value `OK`.

Example 1. (Supply chain management application) In the following sections, we use a simple example to illustrate the introduced interfaces. The supply chain management application consists of five web services: `Shop`, `Store`, `Bank`, `Transport`, and `Supplier`. Figure 1 shows an architectural overview of the application. Labeled arrows from one service to another indicate web method calls from caller to callee. `Shop` supports the web method `SellItem` called by `Client` to start the selling process, and `ChkAvail` which checks availability of items to be sold and is called by `Shop` itself. `Shop` requires the web method `ChkStore` implemented by `Store` to check whether desired items are in stock. It also requires `ShipItem` implemented by `Transport` to ship items to the customer, and `ProcPay` implemented by `Bank` to process credit card payments. `Store` requires `GetOffer` and `Order` implemented by `Supplier` to get an offer for and order new items respectively. □
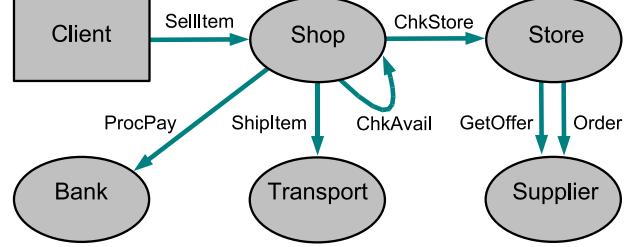

Figure 1. Supply chain management application

2. Signature Interfaces

Let $\mathcal{M}$ and $\mathcal{I}$ be finite sets of *web methods* and *instances*, respectively. Instances are associated with calls to web methods, and encode parameters passed to the web method, return values from the web method, and other behavioral differences between various calls to the web method; for instance, if the invocation was synchronous or asynchronous, or in the latter case, if it will lead to a callback. A *namespace* is a set $\mathcal{N} \subseteq \mathcal{M}$. Let $\mathcal{A} \subseteq \mathcal{M} \times \mathcal{I}$ be the set of *actions*. The web method associated with an action a is denoted as $[a]$. Given $A \subseteq \mathcal{A}$, $[A]$ denotes $\{[a] \mid a \in A\}$.

A *signature* $\mathcal{S} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$ consists of a namespace $\mathcal{N}$, a set $\mathcal{J} \subseteq \mathcal{A}$ of actions that are *supported by* $\mathcal{S}$ such that $[\mathcal{J}] \subseteq \mathcal{N}$, a set $\mathcal{K} \subseteq \mathcal{A}$ of *external* actions that are *required by* $\mathcal{S}$ such that $[\mathcal{K}] \cap \mathcal{N} = \emptyset$, and a partial function $\mathcal{D} : \mathcal{J} \to 2^{\mathcal{A}}$ which assigns to a supported action a a set of actions that can be (directly) invoked by a. A signature $\mathcal{S}$ *supports* a web method $m \in \mathcal{M}$ if $\mathcal{S}$ supports an action a such that $[a] = m$. An action a *requires* an action a' in $\mathcal{S}$ if $a' \in \mathcal{D}(a)$. A signature $\mathcal{S}$ *requires* an action a' if some action a requires a' in $\mathcal{S}$. A signature $\mathcal{S} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$ is a *signature interface* if $\mathcal{D}(a) \subseteq (\mathcal{N} \times \mathcal{I}) \cup \mathcal{K}$ for all $a \in \mathcal{J}$.

Intuitively, an element $(\langle m, i \rangle, D)$ of $\mathcal{D}$ says that when the web method m is called and the caller assumes the instance i, the signature $\mathcal{S} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$ pledges to support this action, and itself relies on that the assumptions carried by the actions $a' \in D$ are fulfilled (by either this signature, or by the environment, or by a refinement of this signature). Thus, a signature interface relates the *"guarantees"* made (actions supported) by the interface to the *"assumptions"* (actions assumed to be supported, either by the interface itself or by the environment) under which they are made.

Example 2. (Signature interface) The signature interface for `Shop` uses the following sets of web methods, instances, and actions:

$$\mathcal{M} = \{\, \text{SellItem}, \text{ChkAvail}, \text{ChkStore}, \text{ProcPay},$$
$$\text{ShipItem}, \text{GetOffer}, \text{Order}\,\}$$
$$\mathcal{I} = \{\, \text{SOLD}, \text{NOTFOUND}, \text{OK}, \text{FAIL}, \text{REC}\,\}$$
$$\mathcal{A} = \{\, \langle \text{SellItem}, \text{SOLD}\rangle, \langle \text{SellItem}, \text{FAIL}\rangle,$$
$$\langle \text{SellItem}, \text{NOTFOUND}\rangle,$$
$$\langle \text{ChkAvail}, \text{OK}\rangle, \langle \text{ChkAvail}, \text{FAIL}\rangle,$$
$$\langle \text{ChkStore}, \text{OK}\rangle, \langle \text{ChkStore}, \text{FAIL}\rangle,$$
$$\langle \text{ProcPay}, \text{OK}\rangle, \langle \text{ProcPay}, \text{FAIL}\rangle,$$
$$\langle \text{ShipItem}, \text{OK}\rangle, \langle \text{ShipItem}, \text{FAIL}\rangle,$$
$$\langle \text{GetOffer}, \text{REC}\rangle, \langle \text{Order}, \text{OK}\rangle\,\}$$

Now we can define a signature $\mathcal{S}_{\text{Shop}}$ consisting of the following components:

$$\mathcal{N}_{\text{Shop}} = \{\, \text{SellItem}, \text{CheckAvail}\,\}$$
$$\mathcal{J}_{\text{Shop}} = \{\, \langle \text{SellItem}, \text{SOLD}\rangle, \langle \text{SellItem}, \text{FAIL}\rangle,$$
$$\langle \text{ChkAvail}, \text{OK}\rangle, \langle \text{ChkAvail}, \text{FAIL}\rangle\,\}$$
$$\mathcal{K}_{\text{Shop}} = \{\, \langle \text{ChkStore}, \text{OK}\rangle, \langle \text{ChkStore}, \text{FAIL}\rangle,$$
$$\langle \text{ProcPay}, \text{OK}\rangle, \langle \text{ProcPay}, \text{FAIL}\rangle,$$
$$\langle \text{ShipItem}, \text{OK}\rangle, \langle \text{ShipItem}, \text{FAIL}\rangle\,\}$$

$$\mathcal{D}_{\text{Shop}} = \{$$
$$\langle \text{SellItem}, \text{SOLD}\rangle \mapsto \{\langle \text{ChkAvail}, \text{OK}\rangle, \langle \text{ProcPay}, \text{OK}\rangle,$$
$$\langle \text{ShipItem}, \text{OK}\rangle\}$$
$$\langle \text{SellItem}, \text{FAIL}\rangle \mapsto \{\langle \text{ChkAvail}, \text{OK}\rangle, \langle \text{ChkAvail}, \text{FAIL}\rangle,$$
$$\langle \text{ProcPay}, \text{OK}\rangle, \langle \text{ProcPay}, \text{FAIL}\rangle,$$
$$\langle \text{ShipItem}, \text{FAIL}\rangle\}$$
$$\langle \text{ChkAvail}, \text{OK}\rangle \mapsto \{\langle \text{ChkStore}, \text{OK}\rangle\}$$
$$\langle \text{ChkAvail}, \text{FAIL}\rangle \mapsto \{\langle \text{ChkStore}, \text{FAIL}\rangle\}$$
$$\}$$

For instance, action $\langle \text{SellItem}, \text{SOLD}\rangle$ is supported by $\mathcal{S}_{\text{Shop}}$, and actions $\langle \text{ChkAvail}, \text{OK}\rangle$, $\langle \text{ProcPay}, \text{OK}\rangle$ and $\langle \text{ShipItem}, \text{OK}\rangle$ are assumed to be supported by the environment. The action $\langle \text{SellItem}, \text{NOTFOUND}\rangle$ is not supported, but belongs to the namespace of the signature — it could be supported in a refinement of this signature.

Signature $\mathcal{S}_{\text{Shop}}$ is a signature interface, because all actions it uses are from the local namespace $\mathcal{N}$ or from the set of environment actions $\mathcal{K}$. $\qquad\square$

2.1. Compatibility and Composition

Given two signature interfaces $\mathcal{S}_1 = (\mathcal{N}_1, \mathcal{J}_1, \mathcal{K}_1, \mathcal{D}_1)$ and $\mathcal{S}_2 = (\mathcal{N}_2, \mathcal{J}_2, \mathcal{K}_2, \mathcal{D}_2)$, if $\mathcal{N}_1 \cap \mathcal{N}_2 = \emptyset$, then $\mathcal{S}_1$ and $\mathcal{S}_2$ are *compatible* (denoted by $comp(\mathcal{S}_1, \mathcal{S}_2)$), and their *composition* (denoted by $(\mathcal{S}_1 \parallel \mathcal{S}_2)$) is $\mathcal{S}_c = (\mathcal{N}_c, \mathcal{J}_c, \mathcal{K}_c, \mathcal{D}_c)$, where $\mathcal{N}_c = \mathcal{N}_1 \cup \mathcal{N}_2$, and $\mathcal{J}_c = \mathcal{J}_1 \cup \mathcal{J}_2$, and $\mathcal{K}_c = (\mathcal{K}_1 \cup \mathcal{K}_2) \setminus (\mathcal{N}_c \times \mathcal{I})$, and $\mathcal{D}_c = \mathcal{D}_1 \cup \mathcal{D}_2$. The composition operation is commutative and associative. Compatibility and composition of signature interfaces can be computed in $O(n \cdot \log n)$ time, where $n = \max(|\mathcal{N}_1|, |\mathcal{N}_2|)$.

A signature interface $\mathcal{S} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$ is *closed* if it requires only actions a with $[a] \in \mathcal{N}$. A signature interface is *open* if it is not closed. A signature interface $\mathcal{S} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$ is *concrete* if it supports all actions a with $[a] \in \mathcal{N}$. A signature is *abstract* if it is not concrete. Given a signature interface $\mathcal{S} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$, an *environment* for $\mathcal{S}$ is a concrete signature interface $\mathcal{E}$ that is compatible with $\mathcal{S}$ such that the composition $\mathcal{S} \parallel \mathcal{E}$ is closed. Note that $\mathcal{S} \parallel \mathcal{E}$ is concrete if and only if $\mathcal{S}$ is concrete, and $\mathcal{E}$ is not unique. Intuitively, each $\mathcal{E}$ represents a design context in which $\mathcal{S}$ can be used.

2.2. Refinement

Given two signature interfaces $\mathcal{S} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$ and $\mathcal{S}' = (\mathcal{N}', \mathcal{J}', \mathcal{K}', \mathcal{D}')$, we say $\mathcal{S}'$ *refines* $\mathcal{S}$ (written $\mathcal{S}' \preccurlyeq \mathcal{S}$) if (i) $\mathcal{N}' \subseteq \mathcal{N}$, (ii) $\mathcal{J}' \supseteq \mathcal{J}$, (iii) $\mathcal{K}' \subseteq \mathcal{K}$, and (iv) for every $a \in \mathcal{J}$, if a requires a' in $\mathcal{S}'$, then a requires a' in $\mathcal{S}$.

The first condition ensures that the refined signature interface does not try to reserve additional names for itself. The second condition ensures that the refined signature interface *guarantees* to support every action that is supported by the abstract one. The other two conditions ensure that the refined signature interface does not *assume* additional actions to be supported by the environment. Given two signature interfaces $\mathcal{S}$ and $\mathcal{S}'$, the question if $\mathcal{S}' \preccurlyeq \mathcal{S}$ can be decided in $O(n \cdot \log n)$ time, where $n = \max(|\mathcal{N}|, |\mathcal{N}'|)$.

Note that the above definition leads to *substitutivity of refinements:* for signature interfaces $\mathcal{S}_1$, $\mathcal{S}'_1$ and $\mathcal{S}_2$, if $comp(\mathcal{S}_1, \mathcal{S}_2)$ and $\mathcal{S}'_1 \preccurlyeq \mathcal{S}_1$, then $comp(\mathcal{S}'_1, \mathcal{S}_2)$ and $\mathcal{S}'_1 \parallel \mathcal{S}_2 \preccurlyeq \mathcal{S}_1 \parallel \mathcal{S}_2$. Intuitively, this means that in an overall design, an abstract placeholder component can be safely replaced with a refined version of it, and the overall design would not exhibit any incorrect behavior if it did not do so before.

3. Consistency Interfaces

A *consistency interface* $\mathcal{C} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{F})$ consists of a namespace $\mathcal{N}$, a set $\mathcal{J} \subseteq \mathcal{A}$ of actions that are *supported* by $\mathcal{C}$ such that $[\mathcal{J}] \subseteq \mathcal{N}$, a set $\mathcal{K} \subseteq \mathcal{A}$ of *external* actions that are *required* by $\mathcal{C}$ such that $[\mathcal{K}] \cap \mathcal{N} = \emptyset$, and a partial function $\mathcal{F} : \mathcal{A} \to \mathcal{B}(\mathcal{A})$, which assigns to a supported action a an expression from $\mathcal{B}(\mathcal{A})$, the set of expressions over the set of actions $\mathcal{A}$ using the binary operators $\sqcap$ and $\sqcup$, and the constant ϵ.

Given a consistency interface $\mathcal{C} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{F})$, the *underlying signature* of $\mathcal{C}$ (denoted by $sig(\mathcal{C})$) is $(\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$, where $\mathcal{D} : \mathcal{A} \to 2^{\mathcal{A}}$ is defined as follows: for all $a \in \mathcal{J}$, $\mathcal{D}(a) = \{a' \mid a' \text{ occurs in expression } \mathcal{F}(a)\}$. We require that the underlying signature of a consistency interface is a signature interface.

Example 3. (Consistency interface) Now we model Shop as a consistency interface $\mathcal{C}_{\text{Shop}} = (\mathcal{N}_{\text{Shop}}, \mathcal{J}_{\text{Shop}}, \mathcal{K}_{\text{Shop}}, \mathcal{F}_{\text{Shop}})$ where $\mathcal{N}_{\text{Shop}}$, and $\mathcal{J}_{\text{Shop}}$, and $\mathcal{K}_{\text{Shop}}$ are as in Example 2, and $\mathcal{F}_{\text{Shop}}$ is as follows:

$\mathcal{F}_{\text{Shop}} = \{$
$\langle \texttt{SellItem}, \texttt{SOLD} \rangle \mapsto \langle \texttt{ChkAvail}, \texttt{OK} \rangle \sqcap \langle \texttt{ProcPay}, \texttt{OK} \rangle \sqcap$
$\qquad\qquad\qquad \langle \texttt{ShipItem}, \texttt{OK} \rangle$
$\langle \texttt{SellItem}, \texttt{FAIL} \rangle \mapsto \langle \texttt{ChkAvail}, \texttt{FAIL} \rangle \sqcup$
$\qquad\qquad (\langle \texttt{ChkAvail}, \texttt{OK} \rangle \sqcap (\langle \texttt{ProcPay}, \texttt{FAIL} \rangle \sqcup$
$\qquad\qquad (\langle \texttt{ProcPay}, \texttt{OK} \rangle \sqcap \langle \texttt{ShipItem}, \texttt{FAIL} \rangle)))$
$\langle \texttt{ChkAvail}, \texttt{OK} \rangle \mapsto \langle \texttt{ChkStore}, \texttt{OK} \rangle$
$\langle \texttt{ChkAvail}, \texttt{FAIL} \rangle \mapsto \langle \texttt{ChkStore}, \texttt{FAIL} \rangle$
$\}$

This consistency interface is a natural extension of the signature interface $\mathcal{S}_{\text{Shop}}$; it keeps different choices in the conversations separate.

For action $\langle \texttt{SellItem}, \texttt{SOLD} \rangle$, all three actions in the expression on the right hand side occur together. For action $\langle \texttt{SellItem}, \texttt{FAIL} \rangle$, action $\langle \texttt{ChkAvail}, \texttt{FAIL} \rangle$ occurs alone, or action $\langle \texttt{ChkAvail}, \texttt{OK} \rangle$ occurs together with either action $\langle \texttt{ProcPay}, \texttt{FAIL} \rangle$ or both, actions $\langle \texttt{ProcPay}, \texttt{OK} \rangle$ and $\langle \texttt{ShipItem}, \texttt{FAIL} \rangle$. Notice that nothing is said about the order of their occurrence. The actions for method ChkAvail result in calls to the method ChkStore in Store.

The underlying signature of $\mathcal{C}_{\text{Shop}}$ is $\mathcal{S}_{\text{Shop}}$ from the example in the last section. Our $\mathcal{C}_{\text{Shop}}$ is a consistency interface because its underlying signature is a signature interface. $\qquad\square$

3.1. Compatibility and Composition

Given two consistency interfaces $\mathcal{C}_1 = (\mathcal{N}_1, \mathcal{J}_1, \mathcal{K}_1, \mathcal{F}_1)$ and $\mathcal{C}_2 = (\mathcal{N}_2, \mathcal{J}_2, \mathcal{K}_2, \mathcal{F}_2)$, if the underlying signatures $sig(\mathcal{C}_1)$ and $sig(\mathcal{C}_2)$ are compatible, then $\mathcal{C}_1$ and $\mathcal{C}_2$ are *compatible* (denoted by $comp(\mathcal{C}_1, \mathcal{C}_2)$), and their *composition* (denoted $\mathcal{C}_1 \parallel \mathcal{C}_2$) is $\mathcal{C}_c = (\mathcal{N}_c, \mathcal{J}_c, \mathcal{K}_c, \mathcal{F}_c)$ where $\mathcal{N}_c = \mathcal{N}_1 \cup \mathcal{N}_2$, and $\mathcal{J}_c = \mathcal{J}_1 \cup \mathcal{J}_2$, and $\mathcal{K}_c = (\mathcal{K}_1 \cup \mathcal{K}_2) \setminus (\mathcal{N}_c \times \mathcal{I})$, and $\mathcal{F}_c = \mathcal{F}_1 \cup \mathcal{F}_2$. The composition operation is commutative and associative. Compatibility and composition of consistency interfaces can be computed in $O(n \cdot \log n)$ time, where $n = \max(|\mathcal{N}_1|, |\mathcal{N}_2|)$. Note that the operators sig and $\parallel$ commute: for all consistency interfaces $\mathcal{C}_1$ and $\mathcal{C}_2$, we have $sig(\mathcal{C}_1 \parallel \mathcal{C}_2) = sig(\mathcal{C}_1) \parallel sig(\mathcal{C}_2)$.

A consistency interface $\mathcal{C}$ is *closed* (*concrete*) if $sig(\mathcal{C})$ is closed (concrete). Given a consistency interface $\mathcal{C}$, an *environment* for $\mathcal{C}$ is a concrete consistency interface $\mathcal{E}$ that is compatible with $\mathcal{C}$, such that the composition $\mathcal{C} \parallel \mathcal{E}$ is closed.

3.2. Refinement

A *conversation* of a consistency interface is a set of actions that are exhibited together in one execution of the system. Given a consistency interface $\mathcal{C} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{F})$, the set of conversations represented by an expression from $\mathcal{B}(\mathcal{A})$ is defined by the function $\llbracket . \rrbracket : \mathcal{B}(\mathcal{A}) \to 2^{2^{\mathcal{A}}}$, which is defined as the least solution of the following system of equations, where $a \in \mathcal{A}$ and $\varphi_1, \varphi_2 \in \mathcal{B}(\mathcal{A})$:

$$
\begin{aligned}
\llbracket \text{T} \rrbracket &= \{\{\}\} \\
\llbracket a \rrbracket &= \{\{a\} \cup y \mid y \in \llbracket \mathcal{F}(a) \rrbracket\} && \text{if } a \in \mathcal{J} \\
\llbracket a \rrbracket &= \{\{a\} \cup y \mid y \subseteq (\mathcal{N} \times \mathcal{I}) \cup \mathcal{K}\} \\
&&& \text{if } a \notin \mathcal{J} \text{ but } [a] \in \mathcal{N} \\
\llbracket a \rrbracket &= \{\{a\}\} && \text{if } [a] \notin \mathcal{N} \\
\llbracket \varphi_1 \sqcup \varphi_2 \rrbracket &= \llbracket \varphi_1 \rrbracket \cup \llbracket \varphi_2 \rrbracket \\
\llbracket \varphi_1 \sqcap \varphi_2 \rrbracket &= \{x \cup y \mid x \in \llbracket \varphi_1 \rrbracket, y \in \llbracket \varphi_2 \rrbracket\}
\end{aligned}
$$

Given two consistency interfaces $\mathcal{C} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{F})$ and $\mathcal{C}' = (\mathcal{N}', \mathcal{J}', \mathcal{K}', \mathcal{F}')$, we say $\mathcal{C}'$ *refines* $\mathcal{C}$ (written $\mathcal{C}' \preccurlyeq \mathcal{C}$) if

i) $sig(\mathcal{C}') \preccurlyeq sig(\mathcal{C})$, and

ii) for every $a \in \mathcal{J}$, for all conversations $x' \in \llbracket \mathcal{F}'(a) \rrbracket$, there exists a conversation $x \in \llbracket \mathcal{F}(a) \rrbracket$ such that $x' \subseteq x$.

The definition above allows the refinement $\mathcal{C}'$ to drop conversations, or actions from a conversation, for actions supported by $\mathcal{C}$. The refinement $\mathcal{C}'$ is allowed to support additional actions that $\mathcal{C}$ does not support, but it is not allowed to require additional actions, and it is not allowed to introduce a new conversation x' in $\mathcal{C}'$ for an action supported by $\mathcal{C}$, such that x' is not a fragment (subset) of a conversation x of the same action in $\mathcal{C}$. The refinement-checking problem for consistency interfaces is in NP.

Theorem 3.1 *Let $\mathcal{C}_1$, $\mathcal{C}_1'$ and $\mathcal{C}_2$ be three consistency interfaces such that $\mathcal{C}_1' \preccurlyeq \mathcal{C}_1$ and $comp(\mathcal{C}_1, \mathcal{C}_2)$. Then $comp(\mathcal{C}_1', \mathcal{C}_2)$ and $\mathcal{C}_1' \parallel \mathcal{C}_2 \preccurlyeq \mathcal{C}_1 \parallel \mathcal{C}_2$.*

3.3. Specifications

A *specification* ψ for a consistency interface is a formula $a \not\rightarrow S$ where $a \in \mathcal{A}$ and $S \subseteq \mathcal{A}$. Intuitively, a specification $a \not\rightarrow S$ states that the invocation of action a must not lead to a conversation in which the actions in set S are all exhibited together. Formally, a specification $\psi = a \not\rightarrow S$ is *satisfied by* a consistency interface $\mathcal{C} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{F})$ (denoted $\mathcal{C} \models \psi$) if $S \not\subseteq x$ for all $x \in \llbracket \mathcal{F}(a) \rrbracket$. The specification satisfaction problem for consistency interfaces is in co-NP. Given two compatible consistency interfaces $\mathcal{C}_1$ and $\mathcal{C}_2$, and a specification ψ, the following holds: $(\mathcal{C}_1 \parallel \mathcal{C}_2) \models \psi$ implies $\mathcal{C}_1 \models \psi$ and $\mathcal{C}_2 \models \psi$. The converse is not true.

Theorem 3.2 *Given a consistency interface $\mathcal{C}$ and a specification ψ:*

1. *If $\mathcal{C} \models \psi$, then for all $\mathcal{C}'$ such that $\mathcal{C}' \preccurlyeq \mathcal{C}$, we have $\mathcal{C}' \models \psi$.*

2. *$\mathcal{C} \models \psi$ if and only if there exists an environment $\mathcal{E}$ of $\mathcal{C}$ such that $\mathcal{C} \parallel \mathcal{E} \models \psi$.*

Corollary 3.3 *Let $\mathcal{C}_1$, $\mathcal{C}_1'$, and $\mathcal{C}_2$ be three consistency interfaces such that $\mathcal{C}_1' \preccurlyeq \mathcal{C}_1$ and $comp(\mathcal{C}_1, \mathcal{C}_2)$. Let ψ be a specification such that $\mathcal{C}_1 \parallel \mathcal{C}_2 \models \psi$. Then $\mathcal{C}_1' \parallel \mathcal{C}_2 \models \psi$.*

4. Protocol Interfaces

Let $Terms$ be a set such that elements $term \in Terms$ are given by the following grammar ($a \in \mathcal{A}$ and $A \subseteq \mathcal{A}$, $|A| \geq 2$):

$$term \;\; ::\;\; \epsilon \mid a \mid \sqcap A \mid \boxplus A$$

A *protocol automaton* $\mathcal{H} = (Q, \delta)$ consists of a finite set of *locations* Q and a finite (nondeterministic) *transition relation* $\delta \subseteq Q \times Terms \times Q$, consisting of tuples $(q, term, q')$ of a *source* location q, a term $term$, and a *successor* location q'. A location that does not occur as a source location in any transition is a *return location*.

A *protocol interface* $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$ consists of a namespace $\mathcal{N}$, a set $\mathcal{J} \subseteq \mathcal{A}$ of actions that are *supported by* $\mathcal{P}$ such that $[\mathcal{J}] \subseteq \mathcal{N}$, a set $\mathcal{K} \subseteq \mathcal{A}$ of *external* actions that are *required by* $\mathcal{P}$ such that $[\mathcal{K}] \cap \mathcal{N} = \emptyset$, a *protocol automaton* $\mathcal{H}$, and a partial function $\mathcal{R} : \mathcal{A} \to Q$ which assigns to a supported action a a location of $\mathcal{H}$.

The execution semantics of a protocol interface can intuitively be understood as follows. The interface maintains a set of current locations. When execution starts due to the invocation of a supported action a, this set contains exactly one location $\mathcal{R}(a)$, and execution ends when the set is empty. If a current location is q, the interface chooses a transition of $\mathcal{H}$ with q as source location and executes the term of the transition. Executing a term means: 1) for an ϵ-term, to proceed with the successor; 2) for a term of the form a, to execute the action a, and, after the execution of a is completed, proceed with the successor; 3) for a term of the form $\sqcap A$, to execute all actions from the set A (in parallel), and, after the execution of all actions is completed, proceed with the successor; and 4) for a term of the form $\boxplus A$, to execute all actions from set A (in parallel), and, after the execution of at least one of the actions is completed, proceed with the successor. Executing an action a means adding the location $\mathcal{R}(a)$ to the set of current locations. Proceeding with the successor means the following: if the successor location of the transition is a return location, the current execution is completed; otherwise the successor location of the transition is added to the current locations.

Given a protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$, the *underlying signature* of $\mathcal{P}$ (denoted $sig(\mathcal{P})$) is $(\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$, where $\mathcal{D}$ is a partial function $\mathcal{D} : \mathcal{A} \to 2^{\mathcal{A}}$ such that for all $a \in \mathcal{J}$, $\mathcal{D}(a) = sigl(\mathcal{R}(a))$. The function $sigl : Q \to 2^{\mathcal{A}}$ that assigns a set of actions to every location q, is defined as follows: $sigl(q) = \bigcup_{i=0,1,\ldots,k} g(term_i) \cup \bigcup_{i=0,1,\ldots,k} sigl(q_i)$ for $(q, term_0, q_0), (q, term_1, q_1), \ldots, (q, term_k, q_k) \in \delta$. The function $g : Terms \to 2^{\mathcal{A}}$ is defined as $g(\epsilon) = \emptyset$, and $g(a) = \{a\}$, and $g(\circ A) = A$ with $\circ \in \{\sqcap, \boxplus\}$, and $a \in \mathcal{A}$, and $A \subseteq \mathcal{A}$. We require that the underlying signature of a protocol interface is a signature interface, and that starting with any location of the automaton as current location, the execution of the protocol interface can be completed.

Example 4. (Protocol interface) Shop is represented by a protocol interface $\mathcal{P}_{\text{Shop}} = (\mathcal{N}_{\text{Shop}}, \mathcal{J}_{\text{Shop}}, \mathcal{K}_{\text{Shop}}, \mathcal{H}_{\text{Shop}}, \mathcal{R}_{\text{Shop}})$ where $\mathcal{N}_{\text{Shop}}$, and $\mathcal{J}_{\text{Shop}}$, and $\mathcal{K}_{\text{Shop}}$ are as in Example 2, and $\mathcal{H}_{\text{Shop}}$ and $\mathcal{R}_{\text{Shop}}$ are defined as follows. For readability, we define $\mathcal{H}_{\text{Shop}}$ and $\mathcal{R}_{\text{Shop}}$ by giving the transition relation δ of $\mathcal{H}_{\text{Shop}}$ as a sequence of tuples $(q, term, q')$, and indicating the partial function $\mathcal{R}$ by writing an action a in front of a transition with $\mathcal{R}(a)$ as source location (location q_0 is a return location):

$\mathcal{H}_{\text{Shop}}$ and $\mathcal{R}_{\text{Shop}}$:

$\langle\texttt{SellItem}, \texttt{SOLD}\rangle \mapsto$	$(q_1, \langle\texttt{ChkAvail}, \texttt{OK}\rangle, q_2),$
	$(q_2, \langle\texttt{ProcPay}, \texttt{OK}\rangle, q_3),$
	$(q_3, \langle\texttt{ShipItem}, \texttt{OK}\rangle, q_0),$
$\langle\texttt{SellItem}, \texttt{FAIL}\rangle \mapsto$	$(q_4, \langle\texttt{ChkAvail}, \texttt{FAIL}\rangle, q_0),$
	$(q_4, \langle\texttt{ChkAvail}, \texttt{OK}\rangle, q_5),$
	$(q_5, \langle\texttt{ProcPay}, \texttt{FAIL}\rangle, q_0),$
	$(q_5, \langle\texttt{ProcPay}, \texttt{OK}\rangle, q_6),$
	$(q_6, \langle\texttt{ShipItem}, \texttt{FAIL}\rangle, q_0),$
$\langle\texttt{ChkAvail}, \texttt{OK}\rangle \mapsto$	$(q_7, \langle\texttt{ChkStore}, \texttt{OK}\rangle, q_0),$
$\langle\texttt{ChkAvail}, \texttt{FAIL}\rangle \mapsto$	$(q_8, \langle\texttt{ChkStore}, \texttt{FAIL}\rangle, q_0)$

This protocol interface is a natural extension of the consistency interface $\mathcal{C}_{\text{Shop}}$; in addition to what $\mathcal{C}_{\text{Shop}}$ does, $\mathcal{P}_{\text{Shop}}$ maintains the temporal order of actions.

It models that for action $\langle\texttt{SellItem}, \texttt{SOLD}\rangle$ the three actions occur in the given sequence in a conversation. When the action $\langle\texttt{SellItem}, \texttt{FAIL}\rangle$ is invoked, Shop checks the availability of the item, nondeterministically assuming the outcome to be either FAIL or OK. In the former case, the next location is q_0, i.e., the sequence induced by action $\langle\texttt{SellItem}, \texttt{FAIL}\rangle$ ends here. In the second case, ($\langle\texttt{ChkAvail}, \texttt{OK}\rangle$), it proceeds with payment processing in location q_5, again nondeterministically assuming the outcome to be FAIL or OK. In the former case the next location is q_0, and in the latter case it tries to ship the item (in location q_6), with the expectation of failure.

The underlying signature of $\mathcal{P}_{\text{Shop}}$ is $\mathcal{S}_{\text{Shop}}$ from Example 2. Our $\mathcal{P}_{\text{Shop}}$ is a protocol interface because its underly-

ing signature is a signature interface and all its executions can be terminated (by reaching a return location). $\square$

Example 5. (Concurrency) The following describes the protocol interface for the Store web service:

$$
\begin{aligned}
\langle \texttt{ChkStore}, \texttt{OK} \rangle \mapsto \quad & (q_1, \epsilon, q_0), \\
& (q_1, \langle \texttt{ChkStore}, \texttt{FAIL} \rangle, q_0), \\
\langle \texttt{ChkStore}, \texttt{FAIL} \rangle \mapsto \quad & (q_2, \langle \texttt{Supp1.GetOffer}, \texttt{REC} \rangle \sqcap \\
& \qquad \langle \texttt{Supp2.GetOffer}, \texttt{REC} \rangle, q_3), \\
& (q_3, \langle \texttt{Supp1.Order}, \texttt{OK} \rangle, q_0), \\
& (q_3, \langle \texttt{Supp2.Order}, \texttt{OK} \rangle, q_0)
\end{aligned}
$$

Let us first consider action $\langle \texttt{ChkStore}, \texttt{FAIL} \rangle$. The interface models that two different supplier web services are simultaneously asked to provide an offer. In this case, the protocol interface expresses not only sequence, but also concurrency. After both offers are received, the automaton switches to location q_3, which models that the Store web service orders the missing item from one of the two suppliers. After this, the automaton switches to the return location q_0, i.e., the conversation induced by action $\langle \texttt{ChkStore}, \texttt{FAIL} \rangle$ ends here. The conversation represented by action $\langle \texttt{ChkStore}, \texttt{OK} \rangle$ is either empty, or it contains the actions for ordering new items (for the case the stock is below a certain threshold). This is modeled by making use of the nondeterministic transition relation. $\square$

4.1. Compatibility and Composition

Given two protocol interfaces $\mathcal{P}_1 = (\mathcal{N}_1, \mathcal{J}_1, \mathcal{K}_1, \mathcal{H}_1, \mathcal{R}_1)$ and $\mathcal{P}_2 = (\mathcal{N}_2, \mathcal{J}_2, \mathcal{K}_2, \mathcal{H}_2, \mathcal{R}_2)$, if the underlying signatures $sig(\mathcal{P}_1)$ and $sig(\mathcal{P}_2)$ are compatible, and $Q_1 \cap Q_2 = \emptyset$, then $\mathcal{P}_1$ and $\mathcal{P}_2$ are *compatible* (denoted $comp(\mathcal{P}_1, \mathcal{P}_2)$), and their *composition* (denoted $\mathcal{P}_1 \parallel \mathcal{P}_2$) is $\mathcal{P}_c = (\mathcal{N}_c, \mathcal{J}_c, \mathcal{K}_c, \mathcal{H}_c, \mathcal{R}_c)$, where $\mathcal{N}_c = \mathcal{N}_1 \cup \mathcal{N}_2$, and $\mathcal{J}_c = \mathcal{J}_1 \cup \mathcal{J}_2$, and $\mathcal{K}_c = (\mathcal{K}_1 \cup \mathcal{K}_2) \setminus (\mathcal{N}_c \times \mathcal{I})$, and $\mathcal{H}_c = (Q_1 \cup Q_2, \delta_1 \cup \delta_2)$, and $\mathcal{R}_c = \mathcal{R}_1 \cup \mathcal{R}_2$, where Q_i and δ_i are the set of locations and the transition relation of the automaton $\mathcal{H}_i$ for $i \in \{1, 2\}$. The composition operation is commutative and associative. Compatibility and composition of protocol interfaces can be computed in $O(n^2 \cdot k^2)$ time, where $n = |Q_1| + |Q_2|$ and $k = \max(k_1, k_2)$, and $k_i = \max_q\{|S| : S = \{(q, term, q') : (q, term, q') \in \delta_i\}\}$ are the corresponding maximal nondeterministic branching factors.

A protocol interface $\mathcal{P}$ is *closed* (*concrete*) if $sig(\mathcal{P})$ is closed (concrete). Given a protocol interface $\mathcal{P}$, an *environment* for $\mathcal{P}$ is a concrete protocol interface $\mathcal{E}$ that is compatible with $\mathcal{P}$, such that the composition $\mathcal{P} \parallel \mathcal{E}$ is closed.

4.2. Refinement

4.2.1. Underlying Game Graph Informally, a protocol interface represents a game being played between two players P_1 and P_2. The interface, together with un-implemented actions in its namespace, collectively constitute the *system*, which plays against the *environment* by making *moves* that change the state of the system and the environment. A *scheduler* (introduced below) arbitrates when both the system and the environment have available moves. Player P_1 plays for the system and the scheduler, and player P_2 plays for the environment. The game is played over a state space consisting of an infinite set of *trees* defined formally as follows.

Given a finite set of *tree symbols* T, a *tree* t over T is a partial function $t : \mathbb{N}^* \to T$, where $\mathbb{N}^*$ denotes the set of finite words over the set of natural numbers $\mathbb{N}$, and the domain $\text{dom}(t) = \{p \in \mathbb{N}^* \mid \exists (p, l) \in t\}$ is prefix-closed. Each element from $\text{dom}(t)$ represents a *node* of tree t: the empty word ρ represents the root of the tree; and the set of child nodes of node p in tree t is $ch(t, p) = \{p' \mid \exists n \in \mathbb{N} : p' = p \cdot n \wedge p' \in \text{dom}(t)\}$, where $\cdot$ is the concatenation operator for strings over $\mathbb{N}^*$. Each node p of the tree is named with the symbol $t(p)$. The set of leaf nodes of a tree t is $leaf(t) = \{p \in \text{dom}(t) \mid ch(t, p) = \emptyset\}$. The set of all trees over a finite set T is denoted $\mathcal{T}(T)$. The initial state of the game, and the winning condition will be defined later. The transitions of the game graph are defined as follows.

Intuitively, locations in Q *belong to* the system, and represent control held by an action supported by the interface. Four fresh locations $q^\forall, q^\exists, q_1,$ and q_2 are introduced. Location $q^\forall$ belonging to the system represents control held by an un-implemented action that will be supported by a refinement of the service being analyzed; the location $q^\exists$ belonging to the environment represents control held by an action required by the interface but outside its namespace that will be supported by the environment; locations q_1 and q_2 are return locations.

Informally, the set of transitions $\to$ of the game graph allows players P_1 and P_2 to change the current state of the game. Players P_1 and P_2 are allowed to move from a given current state s only if s is a pair (t, r) where the tree t has at least one leaf labeled with a location belonging to the system and the environment respectively, and the second component r fulfils the required conditions as described below. Note that if t is a tree with several leaves, not all leaves may necessarily be labeled with locations belonging exclusively to either the system or the environment; then a *scheduler* arbitrates which of them should be allowed to move. If only one of the system or the environment actually has a leaf belonging to it in the tree t, then the scheduler must choose the corresponding player to move next. Otherwise, from a

game state (t, ∇), the scheduler chooses either the former or the latter to be allowed to make the next move by choosing a move to $(t, \forall)$ or $(t, \exists)$ respectively. Note that the scheduler is not allowed to modify the tree configuration. The second component r of a game state (t, r) reflects the decision of the scheduler on whether the system or the environment is allowed to move next: the system can move only from game states of the form $(t, \forall)$, and the enironment can move only from game states of the form $(t, \exists)$. Intuitively, the environment needs to win the game against all possible schedulers. Thus, we allow the scheduler and the system to conspire with each other against the environment, which must play against the coalition. Thus, moves in $\rightarrow$ corresponding to transitions originating from a state (t, r) for $r \in \{\forall, \nabla\}$ belong to player P_1; and those corresponding to transitions originating from a state $(t, \exists)$ belong to player P_2.

Formally, given a protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$, the *underlying game graph* of $\mathcal{P}$ is a labeled *two-player game graph* $\mathcal{G} = (S_1, S_2, L, \rightarrow)$ (denoted by $ugs(\mathcal{P})$) where $S_1 = (\mathcal{T}(Q^\bullet) \times \{\forall, \nabla\})$ is the set of *player-1 states* at which player P_1 chooses the outgoing transition to the next state, and $S_2 = (\mathcal{T}(Q^\bullet) \times \{\exists\})$ is the set of *player-2 states* where player P_2 chooses the nest state, and $L = 2^{\mathcal{A} \cup \{ret\}}$ is the set of *transition labels*, and $\rightarrow \subseteq S \times L \times S$ is the *game transition relation*; where the set of *game states* $S = S_1 \cup S_2$ is the set of pairs (t, r) with r being an element of the set $R = \{\exists, \forall, \nabla\}$ and t being a tree over the set of tree symbols $Q^\bullet = \{(q, \bullet) \mid q \in Q \cup \{q^\forall, q^\exists, q_1, q_2\}, \bullet \in \{\boxplus, \circ\}\}$, with Q is the set of locations of protocol automaton $\mathcal{H}$ and $q^\forall, q^\exists, q_1, q_2$ are fresh symbols not in Q, and the transitions between states are labeled with sets of elements from $\mathcal{A} \cup \{ret\}$. We write $(t, r) \xrightarrow{A} (t', r')$ for $((t, r), A, (t', r')) \in \rightarrow$. Given a protocol interface $\mathcal{P}$ the corresponding game transition relation is defined as follows. In the following, for an action a the symbol q_a is defined as follows: $q_a = \mathcal{R}(a)$ if a is supported ($a \in \mathcal{J}$); $q_a = q^\forall$ if $a \notin \mathcal{J}$ but $[a] \in \mathcal{N}$; and $q_a = q^\exists$ if $[a] \notin \mathcal{N}$. The relation $\delta^\bullet$ used below is defined as $\delta^\bullet = \delta \cup \delta^\forall \cup \delta^\exists$, where δ is the transition relation of $\mathcal{H}$, and

$$
\begin{aligned}
\delta^\forall \;=\; & \{(q^\forall, \epsilon, q_1)\} \\
& \cup \{(q^\forall, a, q^\forall) \mid a \in \mathcal{A}, \text{ s.t. } [a] \in \mathcal{N} \text{ or } a \in \mathcal{K}\} \\
& \cup \{(q^\forall, \circ A, q^\forall) \mid \circ \in \{\sqcap, \boxplus\}, A \subseteq \mathcal{A}, \\
& \qquad \text{s.t. for all } a \in A, [a] \in \mathcal{N} \text{ or } a \in \mathcal{K}\}
\end{aligned}
$$

$$
\begin{aligned}
\delta^\exists \;=\; & \{(q^\exists, \epsilon, q_2)\} \\
& \cup \{(q^\exists, a, q^\exists) \mid a \in \mathcal{A}\} \\
& \cup \{(q^\exists, \circ A, q^\exists) \mid \circ \in \{\sqcap, \boxplus\}, A \subseteq \mathcal{A}\}.
\end{aligned}
$$

Informally, the relation $\delta^\forall$ encodes the ability of an unimplemented action in the interface's namespace to invoke any action under the restrictions imposed on it by the interface; and the relation $\delta^\exists$ encodes the ability of the environment to invoke any action. The relation $\rightarrow$ is defined as follows:

- **Next-Mover:** $(t, \nabla) \xrightarrow{\epsilon} (t, r)$ where $r = \forall$ (or $\exists$) if there exists a node p such that $p \in leaf(t)$, and $t(p) = (q, \circ)$, and $q \in Q \cup \{q^\forall, q_1\}$ (or $q \in \{q^\exists, q_2\}$).

- **Epsilon:** $(t, r) \xrightarrow{\epsilon} (t', \nabla)$ if there exists a node p such that $p \in leaf(t)$, and $t(p) = (q, \circ)$, and $r = \forall$ (or $\exists$) if $q \in Q \cup \{q^\forall, q_1\}$ (or $q \in \{q^\exists, q_2\}$), and $(q, \epsilon, q') \in \delta^\bullet$, and $t' = (t \setminus \{(p, (q, \circ))\}) \cup \{(p, (q', \circ))\}$.

- **Call:** $(t, r) \xrightarrow{\{a\}} (t', \nabla)$ if there exists a node p such that $p \in leaf(t)$, and $t(p) = (q, \circ)$, and $r = \forall$ (or $\exists$) if $q \in Q \cup \{q^\forall, q_1\}$ (or $q \in \{q^\exists, q_2\}$), and $(q, a, q') \in \delta^\bullet$, and $t' = (t \setminus \{(p, (q, \circ))\}) \cup \{(p, (q', \circ)), (p \cdot 0, (q_a, \circ))\}$.

- **Fork:** $(t, r) \xrightarrow{A} (t', \nabla)$ if there exists a node p such that $p \in leaf(t)$, and $t(p) = (q, \circ)$, and $r = \forall$ (or $\exists$) if $q \in Q \cup \{q^\forall, q_1\}$ (or $q \in \{q^\exists, q_2\}$), and $(q, \sqcap A, q') \in \delta^\bullet$, and $t' = (t \setminus \{(p, (q, \circ))\}) \cup \{(p, (q', \circ)), (p \cdot 0, (q_{a_0}, \circ)), \ldots, (p \cdot k, (q_{a_k}, \circ))\}$, where $A = \{a_0, \ldots, a_k\}$.

- **Fork-Choice:** $(t, r) \xrightarrow{A} (t', \nabla)$ if there exists a node p such that $p \in leaf(t)$, and $t(p) = (q, \circ)$, and $r = \forall$ (or $\exists$) if $q \in Q \cup \{q^\forall, q_1\}$ (or $q \in \{q^\exists, q_2\}$), and $(q, \boxplus A, q') \in \delta^\bullet$, and $t' = (t \setminus \{(p, (q, \circ))\}) \cup \{(p, (q', \boxplus)), (p \cdot 0, (q_{a_0}, \circ)), \ldots, (p \cdot k, (q_{a_k}, \circ))\}$, where $A = \{a_0, \ldots, a_k\}$.

- **Return:** $(t, r) \xrightarrow{\{ret\}} (t', \nabla)$ if there exists a node $p \cdot n$, $n \in \mathbb{N}$, such that $p \cdot n \in leaf(t)$, and $t(p \cdot n) = (q, \circ)$ and $r = \forall$ (or $\exists$) if $q \in Q \cup \{q^\forall, q_1\}$ (or $q \in \{q^\exists, q_2\}$), such that q is a return location, $t(p) = (q', \circ)$, and $t' = t \setminus \{(p \cdot n, (q, \circ))\}$.

- **Return & Remove Sibling Tree:** $(t, r) \xrightarrow{\{ret\}} (t', \nabla)$ if there exists a node $p \cdot n$, $n \in \mathbb{N}$, such that $p \cdot n \in leaf(t)$, and $t(p \cdot n) = (q, \circ)$ and $r = \forall$ (or $\exists$) if $q \in Q \cup \{q^\forall, q_1\}$ (or $q \in \{q^\exists, q_2\}$), such that q is a return location, $t(p) = (q'', \boxplus)$, and $t' = (t \setminus \{(p \cdot p', (q', \bullet)) \mid p' \in \mathbb{N}^* \wedge (q', \bullet) \in Q^\bullet\}) \cup \{(p, (q'', \circ))\}$.

A *run* of the game structure is an alternating sequence of game states and sets of actions $s_0, A_1, s_1, A_2, s_2, \ldots$, with $\forall i \in \{1, \ldots n\} : s_{i-1} \xrightarrow{A_i} s_i$. A *trace* is the projection of a run to its action sets; for a run $s_0, A_1, s_1, A_2, s_2, \ldots$, the corresponding trace is $A_1, A_2, \ldots$. The set of *moves* belonging to player P_i at a game state $s_j \in S_i$ is $\{s \mid s_j \xrightarrow{A} s, A \subseteq \mathcal{A} \cup \{ret\}\}$, where $i \in \{1, 2\}$. Move s of player P_i for $i \in \{1, 2\}$ at game state $s_j \in S_i$ *is labeled with* A if $s_j \xrightarrow{A} s$. A *strategy* σ_i of a player P_i for $i \in \{1, 2\}$ is a function that maps every finite run $s_0, A_1, s_1, A_2, s_2, \ldots, s_k$ such that

$s_k \in S_i$ to a move available to P_i at s_k. The set of strategies of player P_i for $i \in \{1, 2\}$ is denoted Ψ_i. For a location q, a *q-run* is a run $s_0, A_1, s_1, \ldots$ with $s_0 = (\{(\rho, (q, \circ))\}, \nabla)$, i.e., a run starting from a game state where the sole thread of control rests at location q of the protocol automaton; a *q-trace* is a trace corresponding to a *q-run*. For a given pair of strategies (σ_1, σ_2) and a location q, the *outcome* is a *q-run* $s_0, A_1, s_1, A_2, s_2, \ldots$, such that for every prefix $r_i = s_0, A_1, s_1, A_2, \ldots, s_i$ of the run, if $s_i \in S_j$, we have $\sigma_j(r_i) = s_{i+1}$, for $i \in \mathbb{N}$ and $j \in \{1, 2\}$. Given a protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$ and an action $a \in \mathcal{J}$, the *initial state of the game representing the invocation of a on $\mathcal{P}$* (denoted $init(\mathcal{P}, a)$) is $(\{\rho \mapsto (\mathcal{R}(a), \circ)\}, \nabla)$.

4.2.2. Alternating Simulation Given two two-player game graphs $\mathcal{G}' = (S_1', S_2', L, \rightarrow')$ and $\mathcal{G} = (S_1, S_2, L, \rightarrow)$ with state spaces $S' = S_1' \cup S_2'$ and $S = S_1 \cup S_2$ respectively, we say $\mathcal{G}'$ *is in alternating simulation with* $\mathcal{G}$ if there exists a relation $\precsim \subseteq S' \times S$ such that:

  i) for every $s_1 \in S_1$, and $s_1' \in S_1'$, if $s_1' \precsim s_1$, then for every player P_1 move s_2' labeled with A and available at s_1' there exists a player P_1 move s_2 labeled with A and available at s_1, such that $s_2' \precsim s_2$, and

  ii) for every $s_2 \in S_2$, and $s_2' \in S_2'$, if $s_2' \precsim s_2$, then for every player P_2 move s_1 labeled with A and available at s_2, there exists a player P_2 move s_1' labeled with A and available at s_2', such that $s_1' \precsim s_1$.

Given two protocol interfaces $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$ and $\mathcal{P}' = (\mathcal{N}', \mathcal{J}', \mathcal{K}', \mathcal{H}', \mathcal{R}')$, we say $\mathcal{P}'$ *refines* $\mathcal{P}$ (written $\mathcal{P}' \preccurlyeq \mathcal{P}$), if:

  i) $sig(\mathcal{P}') \preccurlyeq sig(\mathcal{P})$, and

  ii) for every action $a \in \mathcal{A}$, if $\mathcal{P}$ supports a, then the two two-player game graphs $\mathcal{G}' = ugs(\mathcal{P}')$ and $\mathcal{G} = ugs(\mathcal{P})$ are such that there exists an alternating simulation relation $\precsim$ with $init(\mathcal{P}', a) \precsim init(\mathcal{P}, a)$.

Theorem 4.1 *Let $\mathcal{P}_1$, $\mathcal{P}_1'$ and $\mathcal{P}_2$ be three protocol interfaces such that $\mathcal{P}_1' \preccurlyeq \mathcal{P}_1$ and $comp(\mathcal{P}_1, \mathcal{P}_2)$. Then $comp(\mathcal{P}_1', \mathcal{P}_2)$ and $\mathcal{P}_1' \parallel \mathcal{P}_2 \preccurlyeq \mathcal{P}_1 \parallel \mathcal{P}_2$.*

4.3. Specifications

A *conversation* of a protocol interface is a set of *sequences* of objects A, where each A is a set of *actions*. A *specification* ψ for a protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$ is a formula $a \not\rightsquigarrow \varphi$ where $a \in \mathcal{J}$, and φ is a temporal formula of the form $(\neg C) \, \mathcal{U} \, B$ ("not C until B"), with $C, B \subseteq \mathcal{A}$. Intuitively, a specification $a \not\rightsquigarrow (\neg C) \, \mathcal{U} \, B$ means that the invocation of action a must not lead to a conversation in which an action from the set B occurs before any action from the set C has occurred. A specification ψ for a protocol interface $\mathcal{P}$ is interpreted over

traces generated by the *underlying game graph* of $\mathcal{P}$. Essentially, the specification is taken as the winning condition for the game.

Given a protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$, a location q of $\mathcal{H}$ *satisfies* a temporal formula $(\neg C) \, \mathcal{U} \, B$ (written $q \models (\neg C) \, \mathcal{U} \, B$) if for all strategies $\sigma_2 \in \Psi_2$ of player P_2, there exists a strategy $\sigma_1 \in \Psi_1$ of player P_1, such that the outcome is a run corresponding to a *q-trace* $A_1, A_2, \ldots$ of the underlying game graph $ugs(\mathcal{P})$ such that there exists a $j \in \mathbb{N}$ with $A_j \cap B \neq \emptyset$, and for all $i < j$ we have $A_i \cap C = \emptyset$. An action a *satisfies* a temporal formula $(\neg C) \, \mathcal{U} \, B$ in interface $\mathcal{P}$ if and only if the location $\mathcal{R}(a)$ satisfies the same formula, i.e., $\mathcal{R}(a) \models (\neg C) \, \mathcal{U} \, B$. A protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$ *satisfies* a specification $\psi = a \not\rightsquigarrow \varphi$ (written $\mathcal{P} \models \psi$) if we have $a \not\models \varphi$. To concisely represent our algorithmic solution to this problem, we define a fresh temporal operator $\mathcal{W}$ ("waiting for"), with the following semantics: given a protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$, a location q of $\mathcal{H}$ *satisfies* a temporal formula $(\neg C) \, \mathcal{W} \, B$ (written $q \models (\neg C) \, \mathcal{W} \, B$) if for all strategies $\sigma_2 \in \Psi_2$ of player P_2, there exists a strategy $\sigma_1 \in \Psi_1$ of player P_1, such that the outcome is a run corresponding to a *q-trace* $A_1, A_2, \ldots$ of the underlying game graph $ugs(\mathcal{P})$ such that either there exists no $i \in \mathbb{N}$ such that $A_i \cap C \neq \emptyset$, or there is a $j \in \mathbb{N}$ with $A_j \cap B \neq \emptyset$, and for all $k < j$ we have $A_k \cap C = \emptyset$.

Theorem 4.2 *Given a protocol interface $\mathcal{P}$ and a specification ψ:*

  1. *If $\mathcal{P} \models \psi$, then for all $\mathcal{P}'$ such that $\mathcal{P}' \preccurlyeq \mathcal{P}$, we have $\mathcal{P}' \models \psi$.*

  2. *$\mathcal{P} \models \psi$ if and only if there exists an environment $\mathcal{E}$ of $\mathcal{P}$ such that $\mathcal{P} \parallel \mathcal{E} \models \psi$.*

Corollary 4.3 *Let $\mathcal{P}_1$, $\mathcal{P}_1'$, and $\mathcal{P}_2$ be three protocol interfaces such that $\mathcal{P}_1' \preccurlyeq \mathcal{P}_1$ and $comp(\mathcal{P}_1, \mathcal{P}_2)$. Let ψ be a specification such that $\mathcal{P}_1 \parallel \mathcal{P}_2 \models \psi$. Then $\mathcal{P}_1' \parallel \mathcal{P}_2 \models \psi$.*

While our previous formalism allowed specification-checking only for closed protocol interfaces [1], we now allow to check specifications for open protocol interfaces as well. In the Appendix we present Algorithm 1 which solves the specification-checking problem using the proof rules in Figures 2 and 3.

References

[1] D. Beyer, A. Chakrabarti, and T. Henzinger. Web Service Interfaces. In *Proc. WWW*, pages 148–159. ACM, 2005.

[2] A. Chakrabarti, L. de Alfaro, T. Henzinger, and M. Stoelinga. Resource Interfaces. In *Proc. EMSOFT*, LNCS 2855, pages 117–133. Springer, 2003.

[3] L. de Alfaro and T. Henzinger. Interface Automata. In *Proc. FSE*, pages 109–120. ACM, 2001.

[4] L. de Alfaro, T. Henzinger, and M. Stoelinga. Timed Interfaces. In *Proc. EMSOFT*, LNCS 2491, pages 108–122. Springer, 2002.

[5] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatibility Verification for Web Service Choreography. In *Proc. ICWS*, pages 738–741. IEEE, 2004.

[6] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. WWW*, pages 621–630. ACM, 2004.

[7] X. Fu, T. Bultan, and J. Su. WSAT: A Tool for Formal Analysis of Web Services. In *Proc. CAV*, LNCS 3114, pages 510–514. Springer, 2004.

[8] S. Narayanan and S. A. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. WWW*, pages 77–88. ACM, 2002.

5. Appendix

Proposition 5.1 (Correctness of specification checking)
For a given protocol interface $\mathcal{P}$ and a specification $\psi = a \not\rightarrow (\neg C)\ \mathcal{U}\ B$ for $\mathcal{P}$, procedure $\mathsf{CheckSpec}(\mathcal{P}, a, B, C)$ (Algorithm 1) terminates and returns YES *if $\mathcal{P}$ satisfies ψ, and* NO *otherwise.*

Algorithm 1 $\mathsf{CheckSpec}(\mathcal{P}, a, B, C)$

Input: Protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$,
      Action sets $B, C \subseteq \mathcal{A}$, and action $a \in \mathcal{A}$
Output: YES if $\mathcal{P}$ satisfies $a \not\rightarrow (\neg C)\ \mathcal{U}\ B$, NO otherwise
Variables: Set of judgements S, boolean `done`

1:   `done := F`
2:   **while** ($\neg$ `done`) **do**
3:     `done := T`
4:     **for each** location q of automaton $\mathcal{H}$ **do**
5:       // Try to prove $q \models (\neg C)\ \mathcal{W}\ B$.
6:       **if** all premises of a rule for $q \models (\neg C)\ \mathcal{W}\ B$ are in S **then**
7:         $S := S \cup \{q \models (\neg C)\ \mathcal{W}\ B\}$
8:         `done := F`
9:       // Try to prove $q \models (\neg C)\ \mathcal{U}\ B$.
10:      **if** all premises of a rule for $q \models (\neg C)\ \mathcal{U}\ B$ are in S **then**
11:        $S := S \cup \{q \models (\neg C)\ \mathcal{U}\ B\}$
12:        `done := F`
     end
13:    **if** $q_a \models (\neg C)\ \mathcal{U}\ B\ \in\ S$ **then**
14:      **return** NO
   end
15: **return** YES

Note that we check specifications for open protocol interfaces, which contain calls to unsupported actions, either in its own name space, corresponding to actions that are going to be supported in refined versions, or in the environment's namespace, corresponding to actions that will be supported only when the interface is composed with the environment and additional functionality becomes available. In our case, the game graph allows a variety of choices for behavior for unsupported environment or refinement actions, and the actual behavior of the environment and of the refinement is found by solving the game. In our framework, the environment is friendly, and its goal while playing the game is to help the interface satisfy the specification. The behavior of the system is constrained by the given code, and it has behavioral freedom only to the extent allowed by nondeterminism in the interface, which it uses to attempt to violate the specification. The unsupported actions in the interface namespace, corresponding to actions that will be supported in refinement, play the game attempting to make the interface violate the specification. If the environment has a strat-

egy to win the game, we say the interface satisfies the specification. This means that, irrespective of behavioral nondeterminism, and how the unimplemented actions in the interface namespace are later implemented, the interface can always be used in a way that will ensure the desired property is satisfied. For instance, if vendor A is using a service S with an interface I provided by vendor B, vendor A will receive a guarantee that as long as it respects a protocol (given by the environment's winning strategy for the property desired by A on interface I), no matter what internal refinements are made to the implementation of S by vendor B, the property desired by A will continue to hold true.

The strategies for the players P_1 and P_2 are encoded in the proof rules presented in Figures 2 and 3, where q_a for an action $a \in \mathcal{A}$ is defined as in Section 4.2.1. The first group of rules encodes the strategies of the two players for the formula $(\neg C)\ \mathcal{W}\ B$. Player P_2 tries to make the formula false, her strategy is to call (at least once) an action from $C \setminus B$; for example, from the location $q^{\exists}$, taking an action from $C \setminus B$ and returning immediately afterwards. This always succeeds, except when $C \subseteq B$. Player P_1 tries to make the formula true, so her strategy is to avoid calling any action from the set C. For example, from the location $q^{\forall}$, player P_1 can always immediately return.

The second group of rules encodes the strategies of the two players for formula $(\neg C)\ \mathcal{U}\ B$. Player P_2 tries to make the formula false, her strategy is to call an action from C before an action from B, or none from B; for example, from the location $q^{\exists}$ she can immediately return by taking the ϵ-transition to the return location. There is no proof rule for a judgement of the form $q^{\exists} \models (\neg C)\ \mathcal{U}\ B$, because such transitions can always be chosen by player P_2 to violate the formula. Player P_1 tries to make the formula true, her strategy is to call an action from B before an action from C; for example, from location $q^{\forall}$, she can immediately call an action from B and return immediately afterwards.

$$\frac{}{q \models (\neg C)\ \mathcal{W}\ B} \qquad q \text{ is a return location.} \hspace{3cm} \text{(Return } \mathcal{W})$$

$$\frac{}{q^{\exists} \models (\neg C)\ \mathcal{W}\ B} \qquad C \subseteq B. \hspace{3cm} \text{($\exists$-Return } \mathcal{W})$$

$$\frac{q' \models (\neg C)\ \mathcal{W}\ B}{q \models (\neg C)\ \mathcal{W}\ B} \qquad (q, \epsilon, q') \in \delta \cup \delta^{\forall}. \hspace{2cm} \text{(Epsilon } \mathcal{W})$$

$$\frac{q_a \models (\neg C)\ \mathcal{W}\ B \quad q' \models (\neg C)\ \mathcal{W}\ B}{q \models (\neg C)\ \mathcal{W}\ B} \qquad (q, a, q') \in \delta,\ a \notin C. \hspace{1cm} \text{(Call } \mathcal{W})$$

$$\frac{q_{a_0} \models (\neg C)\ \mathcal{W}\ B\ \dots\ q_{a_k} \models (\neg C)\ \mathcal{W}\ B \quad q' \models (\neg C)\ \mathcal{W}\ B}{q \models (\neg C)\ \mathcal{W}\ B} \qquad \begin{array}{l}(q, \sqcap A, q') \in \delta, \\ A \cap C = \emptyset, \\ A = \{a_0, \dots, a_k\}.\end{array} \hspace{0.5cm} \text{(Fork } \mathcal{W})$$

$$\frac{q_a \models (\neg C)\ \mathcal{W}\ B \quad q' \models (\neg C)\ \mathcal{W}\ B}{q \models (\neg C)\ \mathcal{W}\ B} \qquad \begin{array}{l}(q, \boxplus A, q') \in \delta, \\ A \cap C = \emptyset,\ a \in A.\end{array} \hspace{0.5cm} \text{(Fork-Choice } \mathcal{W})$$

Figure 2. Proof rules for the $\mathcal{W}$ operator

$$\frac{q' \models (\neg C)\ \mathcal{U}\ B}{q \models (\neg C)\ \mathcal{U}\ B} \qquad (q, \epsilon, q') \in \delta. \hspace{3cm} \text{(Epsilon } \mathcal{U})$$

$$\frac{}{q \models (\neg C)\ \mathcal{U}\ B} \qquad \begin{array}{l}(q, a, q') \in \delta \cup \delta^{\forall}\ \vee\ ((q, \circ A, q') \in \delta,\ a \in A,\ \circ \in \{\sqcap, \boxplus\}), \\ a \in B.\end{array} \hspace{0.5cm} \text{(Reached } \mathcal{U}^0)$$

$$\frac{q_a \models (\neg C)\ \mathcal{U}\ B}{q \models (\neg C)\ \mathcal{U}\ B} \qquad \begin{array}{l}((q, a, q') \in \delta, a \notin C)\ \vee\ ((q, \circ A, q') \in \delta,\ \circ \in \{\sqcap, \boxplus\}, \\ A \cap C = \emptyset, a \in A)\end{array} \hspace{0.5cm} \text{(Reached } \mathcal{U}^+)$$

$$\frac{q_a \models (\neg C)\ \mathcal{W}\ B \quad q' \models (\neg C)\ \mathcal{U}\ B}{q \models (\neg C)\ \mathcal{U}\ B} \qquad (q, a, q') \in \delta,\ a \notin C. \hspace{1cm} \text{(Call } \mathcal{U})$$

$$\frac{q_{a_0} \models (\neg C)\ \mathcal{W}\ B\ \dots\ q_{a_k} \models (\neg C)\ \mathcal{W}\ B \quad q' \models (\neg C)\ \mathcal{U}\ B}{q \models (\neg C)\ \mathcal{U}\ B} \qquad \begin{array}{l}(q, \sqcap A, q') \in \delta, \\ A \cap C = \emptyset, \\ A = \{a_0, \dots, a_k\}.\end{array} \hspace{0.5cm} \text{(Fork } \mathcal{U})$$

$$\frac{q_a \models (\neg C)\ \mathcal{W}\ B \quad q' \models (\neg C)\ \mathcal{U}\ B}{q \models (\neg C)\ \mathcal{U}\ B} \qquad \begin{array}{l}(q, \boxplus A, q') \in \delta, \\ A \cap C = \emptyset, a \in A.\end{array} \hspace{0.5cm} \text{(Fork-Choice } \mathcal{U})$$

Figure 3. Proof rules for the $\mathcal{U}$ operator