Consistency of Service Composition

José Luiz Fiadeiro[1] and Antónia Lopes[2]

[1]Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
jose@mcs.le.ac.uk
[2]Faculty of Sciences, University of Lisbon
Campo Grande, 1749–016 Lisboa, Portugal
mal@di.fc.ul.pt

Abstract. We address the problem of ensuring that, when an application executing a service binds to a service that matches required functional properties, both the application and the service can work together, i.e., their composition is consistent. Our approach is based on a component algebra for service-oriented computing in which the configurations of applications and of services are modelled as asynchronous relational nets typed with logical interfaces. The techniques that we propose allow for the consistency of composition to be guaranteed based on properties of service orchestrations (implementations) and interfaces that can be checked at design time, which is essential for supporting the levels of dynamicity required by run-time service binding.

1 Introduction

In recent years, several proposals have been made to characterise the fundamental structures that support service-oriented computing (SOC) independently of the specific languages or platforms that may be adopted to develop or deploy Web services. In this paper, we contribute to this effort by investigating the problem of ensuring that, when an application executing a service binds to a service that it requested, the result is consistent, i.e., both the executing service and the service to which it binds can operate together in the sense that there is a trace that represents an execution of both. In particular, we show how consistency can be checked based on properties of service orchestrations (implementations) and interfaces that can be established at design time. Checking for consistency at discovery time would not be credible because, in SOC, there is no time for the traditional design-time integration and validation activities as the SOA middleware brokers need to discover and bind services at run time.

In order to formulate a notion of consistency and the conditions under which it can be ensured in a way that is as general as possible, i.e., independently of any particular orchestration model (automata, Petri-nets, and so on), we adopt a fairly generic model of behaviour based on traces of observable actions as executed by implementations of services in what are often called 'global computers' — computational infrastructures that are available globally and support the distributed execution of business applications. More precisely, we build on the asynchronous, message-oriented model of interaction that we developed in [10] over which interfaces are defined as temporal logic specifications. That is, instead of a process-oriented notion of interface (which prevails in

most approaches to service orchestration and choreography), we adopt a declarative one that follows in the tradition of logic-oriented approaches to concurrent and distributed system design (as also adopted in [8] for component-based design). One advantage of this approach is that we are able to distinguish between what can be checked at design time to ensure consistency of binding (based on implementations) and what needs to be checked at discovery (run) time to ensure compatibility (based on interfaces).

Having this in mind, in Section 2, we introduce some basic definitions around trace-based models of behaviour and revisit and reformulate, in a more general setting, the notion of asynchronous relational net (ARN) proposed in [10]. In Section 3, we define consistency and prove a sufficient condition for the composition of two consistent ARNs to be consistent, which is based on the notion of safety property. Finally, in Section 4, we discuss which logics support interfaces for ARNs that implement safety properties and propose one such logic that is sufficiently expressive for SOC.

Related work. Most formal approaches that have been proposed for either service choreography or orchestration are process-oriented, for example through automata, labelled transition systems or Petri-Nets. In this context, several notions of compatibility have been studied aimed at ensuring that services are composable. Compatibility in this context may have several different meanings. For example, [16] addresses the problem of ensuring that, at service-discovery time, requirements placed by a requester service are matched by the discovered services — the requirements of the requester are formulated in terms of a graph-based model of a protocol that needs to be simulated by the BPEL orchestration of any provided service that can be discovered. That is, compatibility is checked over implementations. However, one has to assume that the requester has formulated its requirements in such a way that, once bound to a discovered service that meets the requirements, its implementation will effectively work together with that of the provided service in a consistent way — a problem not addressed in that paper.

A different approach is proposed in [6] where compatibility is tested over the interfaces of services (not their implementations), which is simpler and more likely to be effective because a good interface should hide (complex) information that is not relevant for compatibility. A limitation of this approach is that it is based on a (synchronous) method-invocation model of interaction: as argued in [13], web-service composition languages such BPEL (the Business Process Execution Language [20]) rely on an (asynchronous) message-passing model, which is more adequate for interactions that need to run in a loosely-coupled operating environment. An example of an asynchronous framework is the class of automata-based models proposed in [5,7,11], which is used for addressing a number of questions that arise in *choreography*, namely the realisability of conversation protocols among a fixed number of peers in terms of the local behaviour generated by implementations of the peers. Our interest is instead in how dependencies on external services that need to be discovered can be reflected in the interface of a peer and in determining properties of such interfaces that can guarantee that the *orchestration* of the peer can bind to that of a discovered service in a way that ensures consistency of the joint behaviour.

In this respect, the notions of interface that are proposed in [6] do not clearly separate between interfaces for clients of the service and interfaces for providers of required external services, i.e., the approach is not formulated in the context of run-time ser-

vice discovery and binding. Furthermore, [6] does not propose a model of composition of implementations (what is called a component algebra in [8]) so one has to assume that implementations of services with compatible interfaces, when composed, are 'consistent'. The interface and component algebra that we proposed in [10] makes a clear distinction between interfaces for services provided and services requested. Our model, which extends the framework proposed by de Alfaro and Henzinger for component-based systems [8], is based on an asynchronous version of relational nets adapted to SCA (the Service Component Architecture [17]) and defines a component algebra that is is compositional in relation to the binding of required with provided service interfaces. The purpose of this paper is precisely to formulate a notion of consistency at the level of the component algebra through which one can ensure, at design time, that matching required with provided services at the interface level leads to a consistent implementation of the composite service when binding the implementations of the requester and the provider services.

2 Asynchronous relational nets

2.1 Trace-based models of behaviour

The processes that execute in SOC are typically reactive and interactive. Their behaviour can be observed in terms of the actions that they perform. For simplicity, we use a linear model, i.e., we observe streams of actions (which we call segments). In order not to constrain the environment in which processes execute and communicate, we take traces that capture complete behaviours to be infinite and we allow several actions to occur 'simultaneously', i.e. the granularity of observations may not be so fine that we can always tell which of two actions occurred first. Observing an empty set of actions in a trace reflects an execution step during which a process is idle, i.e., a step performed by the environment without the involvement of the process.

More precisely, given a set A (of actions), a *trace* λ over A is an element of $(2^A)^\omega$, i.e., an infinite sequence of sets of actions. We denote by $\lambda(i)$ the i-th element of λ, by λ_i the prefix of λ that ends at $\lambda(i)$, and by λ^i the suffix of λ that starts at $\lambda(i)$. A *segment* over A is an element of $(2^A)^*$, i.e., a finite sequence of sets of actions. We use $\pi \prec \lambda$ to mean that the segment π is a prefix of λ. Given $A' \subseteq A$, we denote by $(\pi \cdot A')$ the segment obtained by extending π with A'.

Definition 1 (Property and Closure) *Let A be an alphabet.*

- *A* property *Λ over A is a subset of $(2^A)^\omega$.*
- *Given $\Lambda \subseteq (2^A)^\omega$, we define $\Lambda^f = \{\pi \in (2^A)^* : \exists \lambda \in \Lambda (\pi \prec \lambda)\}$ — the set of prefixes of traces in Λ, also called the* downward closure *of Λ.*
- *Given $\Lambda \subseteq (2^A)^\omega$, we define $\bar{\Lambda} = \{\lambda \in (2^A)^\omega : \forall \pi \prec \lambda (\pi \in \Lambda^f)\}$ — the set of traces whose prefixes are in Λ, also called the* closure *of Λ.*
- *A property Λ is said to be* closed *iff $\Lambda \supseteq \bar{\Lambda}$.*

The closure operator is defined according to the Cantor topology on $(2^A)^\omega$ used in [1] for characterising safety and liveness properties (see also [4]). In that topology, the closed sets are the safety properties (and the dense ones are the liveness properties).

Functions between sets of actions, which we call alphabet maps, are useful for defining relationships between individual processes and the networks in which they operate. Alphabet maps induce translations that preserve and reflect closed properties:

Proposition and Definition 2 (Translation) *Let $\sigma{:}A{\to}B$ be a function (alphabet map).*

- *For every $\lambda'{\in}(2^B)^\omega$, we define $\lambda'|_\sigma{\in}(2^A)^\omega$ pointwise as $\lambda'|_\sigma(i){=}\sigma^{-1}(\lambda'(i))$.*
- *For every set $\Lambda{\subseteq}(2^A)^\omega$, we define $\sigma(\Lambda) = {\_|_\sigma}^{-1}(\Lambda) = \{\lambda'{\in}(2^B)^\omega : \lambda'|_\sigma{\in}\Lambda\}$.*
- *For every closed property Λ over A, $\sigma(\Lambda)$ is a closed property over B.*
- *For every closed property Λ' over B, $\Lambda'|_\sigma$ is a closed property over A.*

Notice that every alphabet map σ defines a contravariant translation $\_|_\sigma$ between traces by taking the inverse image of the set of actions performed at each step.

2.2 Asynchronous Relational Nets

In this section, we revisit the component algebra proposed in [10] based on the notion of asynchronous relational net (ARN). The main difference is that, where in [10] we formalised ARNs in terms of logical specifications, we are now interested in behaviours (model-theoretic properties) so that we can define and analyse consistency in logic-independent terms. We revisit specifications in the context of interfaces in Sec. 4.

In an asynchronous communication model, interactions are based on the exchange of messages that are transmitted through channels. We organise messages in sets that we call ports: a *port* is a finite set (of messages). Ports are communication abstractions that are convenient for organising networks of processes as formalised below.

Every message belonging to a port has an associated *polarity*: $^-$ if it is an outgoing message (published at the port) and $^+$ if it is incoming (delivered at the port). Therefore, every port M has a partition $M^- \cup M^+$. The actions of sending (publishing) or receiving (being delivered) a message m are denoted by $m!$ and $m¡$, respectively. In the literature, one typically finds $m?$ for the latter. In our model, we use $m?$ for the action of processing the message and $m¿$ for the action of discarding the message: as discussed later, processes cannot refuse the delivery of messages but they should be able to discard them, for example if they arrive outside the protocol expected by the process.

More specifically, if M is a port:

- Given $m{\in}M^-$, the set of actions associated with m is $A_m = \{m!\}$.
- Given $m{\in}M^+$, $A_m = \{m¡, m?, m¿\}$
- The set of actions associated with M is $A_M = \bigcup_{m\in M} A_m$.

A *process* consists of a finite set γ of mutually disjoint ports — i.e., each message that a process can exchange belongs to exactly one of its ports — and a non-empty property Λ over $A_\gamma = \bigcup_{M\in\gamma} A_M$ defining the behaviour of the process.

Interactions in ARNs are established through channels. A *channel* consists of a set M of messages and a non-empty property Λ over the alphabet $A_M{=}\{m!, m¡ : m{\in}M\}$. Channels connect processes through their ports. Given ports M_1 and M_2 and a channel $\langle M, \Lambda\rangle$, a *connection* between M_1 and M_2 via $\langle M, \Lambda\rangle$ consists of a pair of injective maps $\mu_i{:}M{\to}M_i$ such that $\mu_i^{-1}(M_i^+) = \mu_j^{-1}(M_j^-)$, $\{i, j\}{=}\{1, 2\}$ — i.e., a connection

establishes a correspondence between the two ports such that any two messages that are connected have opposite polarities. Each injection μ_i is called the *attachment* of M to M_i. We denote the connection by the triple $\langle M_1 \overset{\mu_1}{\leftarrow} M \overset{\mu_2}{\rightarrow} M_2, \Lambda\rangle$.

Definition 3 (Asynchronous relational net) *An asynchronous relational net (ARN) α consists of:*

- *A simple finite graph $\langle P, C\rangle$ where P is a set of nodes and C is a set of edges. Note that each edge is an unordered pair $\{p, q\}$ of nodes.*
- *A labelling function that assigns a process $\langle \gamma_p, \Lambda_p\rangle$ to every node p and a connection $\langle \gamma_c, \Lambda_c\rangle$ to every edge c such that:*
  - *If $c{=}\{p, q\}$ then γ_c is a pair of attachments $\langle M_p \overset{\mu_p}{\leftarrow} M_c \overset{\mu_q}{\rightarrow} M_q\rangle$ for some $M_p{\in}\gamma_p$ and $M_q{\in}\gamma_q$.*
  - *If $\gamma_{\{p,q\}}{=}\langle M_p \overset{\mu_p}{\leftarrow} M_{\{p,q\}} \overset{\mu_q}{\rightarrow} M_q\rangle$ and $\gamma_{\{p,q'\}}{=}\langle M'_p \overset{\mu'_p}{\leftarrow} M_{\{p,q'\}} \overset{\mu'_{q'}}{\rightarrow} M'_{q'}\rangle$ with $q \neq q'$, then $M_p \neq M'_p$.*

We also define the following sets:

- *$A_p = p.A_{\gamma_p}$ is the language associated with the node p.*
- *$A_\alpha = \bigcup_{p\in P} A_p$ is the language associated with α.*
- *$A_c = \langle p._{-}{\circ}\mu_p, q._{-}{\circ}\mu_q\rangle(A_{M_c})$ is the language associated with $\gamma_c{:}\langle M_p \overset{\mu_p}{\leftarrow} M_c \overset{\mu_q}{\rightarrow} M_q\rangle$.*
- *$\Lambda_\alpha = \{\lambda{\in}(2^{A_\alpha})^\omega{:}\ \forall p{\in}P(\lambda|_p{\in}\Lambda_p) \wedge \forall c{\in}C(\lambda|_c{\in}\Lambda_c)\}$.*

We often refer to the ARN through the quadruple $\langle P, C, \gamma, \Lambda\rangle$ where γ returns the set of ports of the processes that label the nodes and the pair of attachments of the connections that label the edges, and Λ returns the corresponding properties. The fact that the graph is simple — undirected, without self-loops or multiple edges — means that all interactions between two given processes are supported by a single channel and that no process can interact with itself. The graph is undirected because, as already mentioned, channels are bidirectional. Furthermore, different channels cannot share ports.

We take the set Λ_α to define the set of possible traces observed on α — those traces over the alphabet of the ARN that are projected to traces of all its processes and channels. The alphabet of A_α is itself the union of the alphabets of the processes involved translated by prefixing all actions with the node from which they originate.

Notice that nodes and edges denote *instances* of processes and channels, respectively. Different nodes (resp. edges) can be labelled with the same process (resp. channel), i.e., processes and channels act as *types*. This is why it is essential that, in the ARN, it is possible to trace actions to the instances of processes where they originate (all the actions of channels are mapped to actions of processes through the attachments so it is enough to label actions with nodes).

In general, not every port of every process (instance) of an ARN is necessarily connected to a port of another process. Such ports provide the points through which the ARN can interact with other ARNs. An *interaction-point* of an ARN $\alpha = \langle P, C, \gamma, \Lambda\rangle$ is a pair $\langle p, M\rangle$ such that $p{\in}P$, $M{\in}\gamma_p$ and there is no edge $\{p, q\}{\in}C$ labelled with a connection that involves M. We denote by I_α the collection of interaction-points of α.

Interaction-points are used in the notion of composition of ARNs [10]:

Proposition and Definition 4 (Composition of ARNs) *Let* $\alpha_1 = \langle P_1, C_1, \gamma_1, \Lambda_1 \rangle$ *and* $\alpha_2 = \langle P_2, C_2, \gamma_2, \Lambda_2 \rangle$ *be ARNs such that* P_1 *and* P_2 *are disjoint, and a family* $w^i = \langle M_1^i \overset{\mu_1^i}{\leftarrow} M \overset{\mu_2^i}{\rightarrow} M_2^i, \Psi^i \rangle$ *(*$i = 1 \ldots n$*) of connections for interaction-points* $\langle p_1^i, M_1^i \rangle$ *of* α_1 *and* $\langle p_2^i, M_2^i \rangle$ *of* α_2 *such that* $p_1^i \neq p_1^j$ *if* $i \neq j$ *and* $p_2^i \neq p_2^j$ *if* $i \neq j$*. The composition*

$$\alpha_1 \; \Big|\Big|^{i=1\ldots n}_{\langle p_1^i, M_1^i \rangle, w^i, \langle p_2^i, M_2^i \rangle} \; \alpha_2$$

is the ARN whose graph is $\langle P_1 \cup P_2, C_1 \cup C_2 \cup \bigcup_{i=1\ldots n} \{p_1^i, p_2^i\} \rangle$ *and whose labelling function coincides with that of* α_1 *and* α_2 *on the corresponding subgraphs, and assigns to the new edges* $\{p_1^i, p_2^i\}$ *the label* w^i*.*

In order to illustrate the notions introduced in the paper, we consider a simplified bank portal that mediates the interactions between clients and the bank in the context of different business operations such as the request of a credit. Fig. 1 depicts an ARN with two interconnected processes that implement this business operation. Process *Clerk* is responsible for the interaction with the environment and for making decisions on credit requests, for which it relies on an external process $RiskEvaluator$ that is able to evaluate the risk of the transaction. The graph of this ARN consists of two nodes $c{:}Clerk$ and $e{:}RiskEvaluator$ and an edge $\{c, e\}{:}w_{ce}$ where:

- *Clerk* is a process with two ports: L_c and R_c. In port L_c, the process receives messages $creditReq$ and $accept$ and sends $approved$, $denied$ and $transferDate$. Port R_c has outgoing message $getRisk$ and incoming message $riskValue$. The *Clerk*'s behaviour is as follows: immediately after the delivery of the first $creditReq$ message on port L_c, it publishes $getRisk$ on R_c; then it waits five time units for the delivery of $riskValue$, upon which it either publishes $denied$ or $approved$ (we abstract from the criteria that it uses for deciding on the credit); if $riskValue$ does not arrive by the deadline, *Clerk* publishes $denied$ on L_c; after sending $approved$ (if ever), *Clerk* waits twenty time units for the delivery of $accept$, upon which it sends $transferDate$; all other deliveries of $creditReq$ and $accept$ are discarded. The property that corresponds to this behaviour is denoted by Λ_c in Fig. 1.
- *RiskEvaluator* is a process with a single port (L_e) with incoming message $request$ and outgoing message $result$. Its behaviour is quite simple: every time $request$ is delivered, it takes no more than three time units to publish $result$. The property that corresponds to this behaviour is denoted by Λ_e in Fig. 1.
- The port R_c of *Clerk* is connected with the port L_e of *RiskEvaluator* through $w_{ce}{:}\langle R_c \overset{\mu_e}{\leftarrow} \{m, n\} \overset{\mu_c}{\rightarrow} L_e, \Lambda_w \rangle$, with $\mu_c{=}\{m \mapsto getRisk, n \mapsto riskValue\}$, $\mu_e{=}\{m \mapsto request, n \mapsto result\}$. The corresponding channel is reliable: it ensures to delivering $getRisk$, which *RiskEvaluator* receives as $request$, and it ensures to delivering $result$, which *Clerk* receives as $riskValue$, both without any delay. The property that corresponds to this behaviour is denoted by Λ_w in Fig. 1.

3 Consistency

An important property of ARNs, and the one that justifies this paper, is consistency:

Definition 5 (Consistent ARN) *An ARN* α *is said to be* consistent *if* Λ_α *is not empty.*

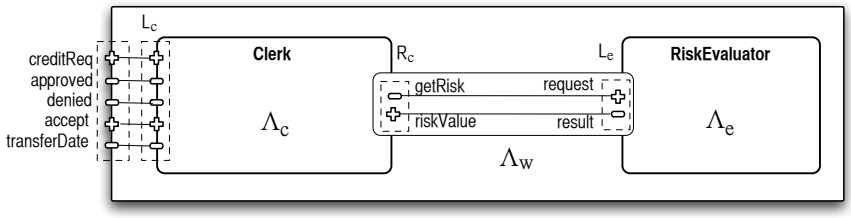

Fig. 1. An example of an ARN with two processes connected through a channel.

Consistency means that the processes, interconnected through the channels, can cooperate and generate at least a joint trace. Naturally, one cannot expect every ARN to be consistent as the interference established through the connections may make it impossible for the processes involved to make progress together. Therefore, some important questions, which this paper attempts to answer, are: *How can one check that an ARN α is consistent without calculating the set Λ_α? How can one guarantee that the composition of two consistent ARNs is consistent based on properties of the ARNs and the interconnections that can be checked at design time?*

In order to answer these questions, we are going to discuss a related property: the ability to make (finite) progress no matter the segment that the ARN has executed, which we call progress-enabledness. We show that, for certain classes of ARNs, progress-enabledness implies consistency. We also provide sufficient conditions for the composition of two progress-enabled ARNs to be progress-enabled that can be checked at design time.

3.1 Progress-enabled ARNs

Consistency is about infinite behaviours, i.e., it concerns the ability of all processes and channels to generate a full joint trace. However, it does not guarantee that, having engaged in a joint partial trace (finite segment), the processes can proceed: it may happen that a joint partial trace is not a prefix of a joint (full) trace, which would be undesirable as it is not possible for individual processes to anticipate what other processes will do — as discussed in Sec. 4, interconnections in the context of SOC are established at run time based on interfaces that capture *what* processes do, not *how* they do it. This is why, in [10], we introduced another useful property of ARNs: that, after any joint partial trace, a joint step can be performed.

Definition 6 (Progress-enabled ARN) *For every ARN α, let*

$$\Pi_\alpha = \{\pi \in 2^{A_\alpha{}^*} : \forall p \in P(\pi|_p \in \Lambda_p^f) \wedge \forall c \in C(\pi|_c \in \Lambda_c^f)\}$$

We say that α is progress-enabled iff $\forall \pi \in \Pi_\alpha . \exists A \subseteq A_\alpha (\pi \cdot A) \in \Pi_\alpha$.

The set Π_α consists of all the partial traces that the processes and channels can jointly engage in. Notice that, as long as the processes and channels involved in α are consistent, Π_α is not empty: it contains at least the empty trace!

Therefore, by itself, being progress-enabled does not guarantee that an ARN is consistent: moving from finite to infinite behaviours requires the analysis of what happens

'at the limit'. A progress-enabled but inconsistent ARN guarantees that all the processes and channels will happily make joint progress but at least one will be prevented from achieving a successful full trace at the limit. Therefore, it seems justifiable that we look for a class of ARNs for which being progress-enabled implies consistency, which we do in the next subsection. However, in relation to the points that we raised at the beginning of this section, we still need to show that, by investigating a stronger property (being progress-enabled and consistent), we have not made the questions harder to answer.

In [10], we also identified properties of ARNs and channels that guarantee that the composition of two progress-enabled ARNs is progress-enabled: that processes are able to buffer incoming messages, i.e., to be 'delivery-enabled', and that channels are able to buffer published messages, i.e., to be 'publication-enabled'.

Definition 7 (Delivery-enabled) *Let $\alpha{=}\langle P, C, \gamma, \Lambda\rangle$ be an ARN, $\langle p, M\rangle{\in}I_\alpha$ one of its interaction-points, and $D_{\langle p,M\rangle}{=}\{p.m¿ : m{\in}M^+\}$. We say that α is delivery-enabled in relation to $\langle p, M\rangle$ if, for every $(\pi{\cdot}A){\in}\Pi_\alpha$ and $B{\subseteq}D_{\langle p,M\rangle}$, $(\pi{\cdot}B \cup (A{\setminus}D_{\langle p,M\rangle})){\in}\Pi_\alpha$.*

That is, being delivery-enabled at an interaction point requires that any joint prefix of the ARN can be extended by any set of messages delivered at that interaction-point. Note that this does not interfere with the decision of the process to publish messages: $B{\cup}(A{\setminus}D_{\langle p,M\rangle}))$ retains all the publications present in A. Also notice that accepting the delivery of a message does not mean that a process will act on it; this is why we distinguish between executing a delivered message $(m?)$ and discarding it $(m¿)$. For example, the processes $Clerk$ and $RiskEvaluator$ informally described in Sec. 2.2 define, individually, atomic ARNs that are delivery-enabled: they put no restrictions on the delivery of messages.

Definition 8 (Publication-enabled) *Let $h{=}\langle M,\Lambda\rangle$ be a channel and $E_h = \{m! : m \in M\}$. We say that h is publication-enabled iff, for every $(\pi{\cdot}A){\in}\Lambda^f$ and $B{\subseteq}E_h$, we have $\pi{\cdot}(B{\cup}(A{\setminus}E_h)){\in}\Lambda^f$.*

The requirement here is that any prefix can be extended by the publication of any set of messages, i.e., the channel should not prevent processes from publishing messages. Notice that this does not interfere with the decision of the channel to deliver messages: $(B{\cup}(A{\setminus}E_h))$ retains all the deliveries present in A. An example is the channel used in Fig. 1, which we informally described in Sec. 2.2.

These two properties allow us to prove that the composition of two progress-enabled ARNs is progress-enabled [10]:

Theorem 9 *Let $\alpha = (\alpha_1 \;||^{i=1\ldots n}_{\langle p_1^i, M_1^i\rangle, w^i, \langle p_2^i, M_2^i\rangle}\; \alpha_2)$ be a composition of progress-enabled ARNs where, for each $i = 1 \ldots n$, $w^i = \langle M_1^i \xleftarrow{\mu_1^i} M \xrightarrow{\mu_2^i} M_2^i, \Lambda^i\rangle$. If, for each $i{=}1\ldots n$, α_1 is delivery-enabled in relation to $\langle p_1^i{,}M_1^i\rangle$, α_2 is delivery-enabled in relation to $\langle p_2^i{,}M_2^i\rangle$ and $h^i{=}\langle M^i{,}\Lambda^i\rangle$ is publication-enabled, then α is progress-enabled.*

3.2 Safe ARNs

The class of ARNs for which we can guarantee consistency are those that involve only closed (safety) properties (cf. Def. 1). As discussed above, progress-enabledness guarantees that all the processes and channels can progress by making joint steps but does

not guarantee that successful full traces will be obtained at the limit. Choosing to work with safety properties essentially means that 'success' does not need to be measured at the limit, i.e., checking the ability to make 'good' progress is enough.

From a methodological point of view, restricting ARNs to safety properties is justified by the fact that, within SOC, we are interested in processes whose liveness properties are bounded (bounded liveness being itself a safety property). This is because, in typical business applications, one is interested only in services that respond within a fixed (probably negotiated) delay. In SOC, one does not offer as a service the kind of systems that, like operating systems, are not meant to terminate

Definition 10 (Safe processes, channels and ARNs) *A process $\langle \gamma, \Lambda \rangle$ (resp. channel $\langle M, \Lambda \rangle$) is said to be* safe *if Λ is closed. A* safe ARN *is one that is labelled with safe processes and channels.*

Proposition 11 *For every safe ARN α, Λ_α is a closed (safety) property.*

Proof. *Λ_α is the intersection of the images of the properties of the processes and channels associated with the nodes and edges of the graph. According to Prop. 2, those images are safety properties. The result follows from the fact that an intersection of closed sets in any topology is itself a closed set.*

Theorem 12 (Consistency) *Any safe progress-enabled ARN is consistent.*

Proof. *Given that the processes and channels in a safe ARN are consistent, Π_α (cf. Def. 6) is not empty (it contains at least the empty segment ϵ). Π_α can be organised as a tree, which is finitely branching because A_α is finite. If the ARN is progress-enabled, the tree is infinite. By Kőnigs lemma, it contains an infinite branch λ.*

We now prove that $\lambda \in \Lambda_\alpha$, i.e., $\lambda|_p \in \Lambda_p$ for all $p \in P$ and $\lambda|_c \in \Lambda_c$ for all $c \in C$. Let $p \in P$ and $\pi \prec \lambda|_p$. We know that π is of the form $\pi'|_p$ where $\pi' \in \Pi_\alpha$. Therefore, $\pi \in \Lambda_p^f$. It follows that $\lambda|_p \in \overline{\Lambda_p}$. Because Λ_p is closed, we can conclude that $\lambda|_p \in \Lambda_p$. The same reasoning applies to all channels.

Note that, in the case of non-safe ARNs, being progress-enabled is a necessary but not sufficient condition to ensure consistency. For example, consider the following two processes: P recurrently sends a given message m and Q is able to receive a message n but only a finite, though arbitrary, number of times. If these processes are interconnected through a reliable channel that ensures to delivering n every time m is published, it is easy to conclude that the resulting ARN is not consistent in spite of being progress-enabled: after having engaged in any joint partial trace, both processes and the channel can proceed (Q will let the channel deliver n once more if necessary); however, they are not able to generate a full joint trace because P will want to send m an infinite number of times and Q will not allow the channel to deliver n infinitely often.

Because the composition of safe ARNs through safe channels is safe, Theo. 9 can be generalised to guarantee consistency of composition:

Corollary 13 (Consistency of composition) *The composition of safe progress-enabled ARNs is both safe and progress-enabled (and, hence, consistent) provided that interconnections are made through safe publication-enabled channels and over interaction-points in relation to which the ARNs are delivery-enabled.*

It remains to determine how ARNs can be proved to be safe, progress-enabled, and delivery-enabled in relation to interaction points, and channels to be safe and publication-enabled. In this respect, another important result (see [10] for details) is that the composition of two ARNs is delivery-enabled in relation to all the interaction-points of the original ARNs that remain disconnected and in relation to which they are delivery-enabled. Therefore, because every process defines an (atomic) progress-enabled ARN (by virtue of being consistent), the proof that an ARN is progress-enabled can be reduced to checking that individual processes are delivery-enabled in relation to their ports and that the channels are publication-enabled. On the other hand, ensuring that processes and channels are safe relates to the way they are specified and implemented.

All these questions are addressed in the next section, where we also discuss how service interfaces should be specified in the context of orchestrations that are safe and progress-enabled. In particular, we show that all the properties that can guarantee consistent composition can be checked at (process/channel) design time, not at (ARN) composition time (which, in SOC, is done at run time).

4 Interface specifications for safe ARNs

4.1 Interfaces and orchestrations

Making the discovery and binding of services to be based on interfaces, not implementations, has the advantage of both simplifying those processes (as interfaces should offer a more abstract view of the behaviour of the services) and decoupling the publication of services in registries from their instantiation when needed. In [10] we proposed an interface theory for ARNs based on linear temporal logic (LTL), which distinguishes between provides- and requires-points:

- A *provides-point* r consists of a port M_r together with a consistent set of sentences Φ_r over A_{M_r} that express what the service offers to any of its clients.
- A *requires-point* r consists of a port M_r and a consistent set of sentences Φ_r over A_{M_r} that express what the service requires from an external service, together with a consistent set of sentences Ψ_r over $\{m!, m¡\colon m{\in}M_r\}$ that express requirements on the channel through which it expects to interact with the external service.
- Matching a requires-point of a service interface with a provides-point of another service interface amounts to checking that the specification of the latter entails that of the former.

In Fig. 2, we present an example of an interface for a credit service using a graphical notation similar to that of SCA. On the left, we have a provides-point $Customer$ and, on the right, a requires-point $IRiskEvaluator$. The set of sentences Φ_c, in the logic discussed in the next subsection, specifies the service offered at $Customer$:

- $(creditReq¡\ \mathcal{R}\ (creditReq¡ \supset \Diamond_{\leq 10}(approved! \vee denied!)))$ — either $approved$ or $denied$ are published within ten time units of the first delivery of $creditReq$.
- $\Box(approved! \supset (accept¡\ \mathcal{R}_{\leq 20}\ (accept¡ \supset \Diamond_{\leq 2} transferDate!)))$ — if $accept$ is received within twenty time units of the publication of $approved$, $transferDate$ will be published within 2 time units.

The specification Φ_r of $IRiskEvaluator$ requires the external service to react to the delivery of every $request$ by publishing $result$ in no more than four time units: $\Box(request_{\text{¡}} \supset \Diamond_{\leq 4} result!)$.

The connection with the external service is required to ensure that messages are transmitted immediately to the recipient.

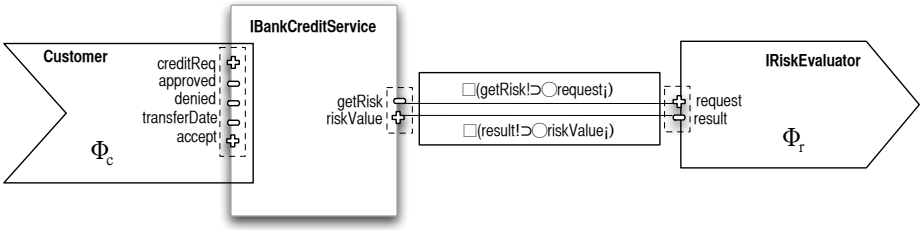

Fig. 2. An example of a service interface.

An ARN orchestrates a service-interface by assigning interaction-points to interface-points in such a way that the behaviour of the ARN validates the specifications of the provides-points on the assumption that it is interconnected to ARNs that validate the specifications of the requires-points through channels that validate the corresponding specifications. Notice that ensuring consistency is essential because an interconnection that leads to an inconsistent composition would vacuously satisfy any specification (there would be no behaviours to check against the specification).

Therefore, in order to check that an ARN α orchestrates a service-interface I:

1. For every requires-point r of I, we consider an ARN α_r defined by a single process $\langle M_r, \Lambda_r \rangle$ where Λ_r is a safety property that validates Φ_r and makes α_r delivery-enabled in relation to r, which is representative of the safe and progress-enabled ARNs that can be interconnected at r, i.e., that provides a 'canonical' orchestration of a service that offers a provides-point that matches r.
2. For every requires-point r of I, we consider a channel c_r=$\langle M_r, \Lambda_r \rangle$ where Λ_r is a safety property that validates Ψ_r and makes the channel publication-enabled, which represents the most general channel that can be used for interconnecting an orchestration with an external service.
3. We consider the composition α^* of α with all the α_r via $\langle M_{p_r} \overset{\theta_r}{\leftarrow} M_r \overset{id}{\rightarrow} M_r, \_ \rangle$ where p_r is the interaction-point of α that corresponds to the requires-point r through the mapping θ_r:$M_r^{op} \rightarrow M_p$ (for every port M, we denote by M^{op} the port defined by M^{op+}=M^- and M^{op-}=M^+).
4. For α to orchestrate the interface I we require that $\Lambda_{\alpha^*}|_{A_{M_r}} \vDash \Phi_r$ for every provides-point r of I. Notice that $\Lambda_{\alpha^*}|_{A_{M_r}}$ is the projection of the traces of the composed ARN on the alphabet of the provides-point r which, by Prop. 2, is a safety property.

The question now is how to choose such canonical processes $\langle M_r, \Lambda_r \rangle$ (and channels). Typically, in logic, the collection Λ_{Φ_r} of all traces that validate Φ_r (Ψ_r in the case of channels) would meet the requirement because any other ARN would give rise

to fewer traces over A_{M_r}. However, if we want to restrict ourselves to processes and channels that are safe, one has to choose interfaces in the class of specifications that denote safety properties, i.e., for which Λ_{Φ_r} is closed. For example, not every specification in LTL is in that class. The same applies to provides-points because, by Prop. 2, $\Lambda_{\alpha^*}|_{A_{M_r}}$ is a safety property. In this case, because the properties offered in a provides-points derive from the ARN that orchestrates the interface, we would need to be able to support the development of safe processes and channels from logical specifications. Therefore, we need to discuss which logics support that class of specifications.

4.2 A logic of safety properties

Several extensions of LTL (e.g., Metric Temporal Logic – MTL [14]) have been proposed in which different forms of bounded liveness can be expressed through eventuality properties of the form $\Diamond_I \phi$ where I is a time interval during which ϕ is required to become true. Another logic of interest is PROMPT-LTL [15] in which, instead of a specific bound for the waiting time, one can simply express that a sentence ϕ will become true within an unspecified bound — $\Diamond_{\mathbf{p}}\phi$. Yet another logic is PLTL [3] in which one can use variables in addition to constants to express bounds on the waiting time and reason about the existence of a bound (or of a minimal bound) for a response time.

The logic we propose to work with, which we call SAFE-LTL, is a 'safety' fragment of LTL — positive formulas with 'release' and 'next' — which corresponds to the fragment of PLTL where intervals are finite and bounded by constants. This logic can also be seen as a restricted version of Safety MTL [18] (a fully decidable fragment of MTL) where, instead of an explicit model of real-time, we adopt an implicit one in which time is measured by the natural numbers (as in PLTL). From a methodological point of view, the adoption of an implicit, discrete time model can be justified by the fact that, in SOC, one deals with 'business' time where delays are measured in discrete time units that are global (i.e., the time model is synchronous even if the interaction model is asynchronous). This is somewhat different from time-critical systems, for which a continuous time model (i.e., with no fixed minimal time unit) is more adequate.

Definition 14 (SAFE-LTL) *Let A be an alphabet.*

- *The language of* SAFE-*LTL over A is defined by (where $a \in A$):*

$$\phi \ ::= \ a \mid \neg a \mid \phi \vee \phi \mid \phi \wedge \phi \mid \bigcirc \phi \mid \phi \,\mathcal{R}\, \phi$$

- *Sentences are interpreted over $\lambda \in (2^A)^\omega$ as follows :*
  $\lambda \vDash a$ *iff* $a \in \lambda(0)$*;* $\lambda \vDash \neg a$ *iff* $a \notin \lambda(0)$
  $\lambda \vDash \phi_1 \wedge \phi_2$ *iff* $\lambda \vDash \phi_1$ *and* $\lambda \vDash \phi_2$*;* $\lambda \vDash \phi_1 \vee \phi_2$ *iff* $\lambda \vDash \phi_1$ *or* $\lambda \vDash \phi_2$
  $\lambda \vDash \bigcirc \phi$ *iff* $\lambda^1 \vDash \phi$
  $\lambda \vDash \phi_1 \,\mathcal{R}\, \phi_2$ *iff, for all j, either $\lambda^j \vDash \phi_2$ or there exists $k<j$ s.t. $\lambda^k \vDash \phi_1$*

Notice that sentences are in positive form: negation is only available for atomic propositions (actions). This allows us to define $(a \supset \phi)$ as an abbreviation for $(\neg a \vee \phi)$ as used in the interface specifications above. We also use $\Box\phi$ as an abbreviation of $(false\,\mathcal{R}\,\phi)$.

The bounded operators used in the interface specifications given in Sec. 4.1 amount to the following abbreviations where $t \in \mathbb{N}$:

– $(\phi_1 \mathcal{R}_{\leq t} \phi_2) \equiv \phi_2 \wedge (\phi_1 \vee \bigcirc\phi_2) \wedge \cdots \wedge (\phi_1 \vee \bigcirc\phi_1 \vee \cdots \vee \bigcirc^{t-1}\phi_1 \vee \bigcirc^t\phi_2)$
– $(\phi_1 \mathcal{U}_{\leq t} \phi_2) \equiv \phi_2 \vee (\phi_1 \wedge \bigcirc\phi_2) \vee \cdots \vee (\phi_1 \wedge \bigcirc\phi_1 \wedge \cdots \wedge \bigcirc^{t-1}\phi_1 \wedge \bigcirc^t\phi_2)$
– $\Box_{\leq t}\phi \equiv false\, \mathcal{R}_{\leq t}\, \phi \equiv \phi \wedge \bigcirc\phi \wedge \cdots \wedge \bigcirc^t\phi$
– $\Diamond_{\leq t}\phi \equiv true\, \mathcal{U}_{\leq t}\, \phi \equiv \phi \vee \bigcirc\phi \vee \cdots \vee \bigcirc^t\phi$

Theorem 15 (Safety) *All the sentences of* SAFE-*LTL express safety properties, i.e., for every sentence ϕ, the set of traces that satisfy it is closed.*

Proof. See [19] for a similar logic that uses 'unless' instead of 'release'.

Corollary 16 (Safe specifications) *It follows from the previous theorem that all specifications over* SAFE-*LTL are safe, i.e., for all sets of sentences Φ, the set Λ_Φ of all traces λ such that $(\lambda \vDash \Phi)$ is a safety property.*

Proof. The results follow from the fact that the intersection of any number of closed properties is closed.

4.3 Ensuring delivery/publication-enabledness

In addition to making sure that specifications generate safety properties, it is important to guarantee that specifications associated with requires-points generate processes that are delivery-enabled in relation to their port and channels that are publication-enabled. Ensuring delivery/publication-enabledness is not the same as proving that an implementation satisfies a specification because those properties are not expressible as sentences whose satisfaction can be checked over individual traces: they need to be checked over the set of all traces that satisfy the specification.

Traces are observations of the behaviours of systems that implement processes. Typical examples of (models of) such systems that are used in association with a logic are finite automata of some kind such that, for every specification $\langle A, \Phi\rangle$, there is a system S_Φ over the alphabet A such that Λ_{S_Φ}=Λ_Φ. The idea is then to check delivery/publication-enabledness directly over S_Φ.

In the case of LTL, systems are *non-deterministic Bücchi automata* (NBAs) [21]. An NBA over an alphabet A is a tuple of the form $\langle Q, \delta, Q_0, Q_\infty\rangle$ where Q is a finite set of states, $Q_0 \subseteq Q$ is the subset of initial states, $Q_\infty \subseteq Q$ is the set of accepting states, and $\delta : Q \times A \to 2^Q$ is the transition relation. The property defined by $\langle Q, \delta, Q_0, Q_\infty\rangle$ is the set of infinite sequences of elements of A that, starting on an initial state, generate a run that visits at least one of the accepting states infinitely often.

In relation to safety properties, there is also a *closure* operator on NBAs [2]: the closure of $\langle Q, \delta, Q_0, Q_\infty\rangle$ is $\langle Q, \delta, Q_0, Q\rangle$, i.e., the NBA obtained by making all states accepting. A reduced NBA (i.e., one in which every state leads to an accepting state) defines a safety property if and only if its closure defines the same property. Furthermore, every NBA is equivalent to a reduced one.

Therefore, given that we are interested in working with safe specifications, we can choose closed reduced NBAs as models of implementations of processes and channels. In this case, it is easy to see that all that needs to be checked for processes (resp. channels) to be delivery (resp. publication) enabled is that, from every state of the automata that implement them, the set of transitions from that state satisfies the corresponding

property, i.e., for every set of deliveries (resp. publications), there is a transition that delivers (resp. publishes) exactly those messages. As a result, the complexity of the checking process is in the order of the product of the size of the automaton and of the sub-language of deliveries/publications.

5 Concluding remarks

In this paper, we discussed the problem of ensuring that the composition of orchestrations of matching service interfaces is consistent, i.e., that the orchestrations of both services can effectively work together when interconnected through the communication channels that bind them. Our findings led us to propose a refinement of the service interface and component algebra presented in [10] in which services are orchestrated by asynchronous relational nets that exhibit only safety properties (i.e., any 'bad' behaviour should be able to detected after a finite number of steps) and are progress-enabled (i.e., always able to make progress, even if by remaining idle). The advantages of working with safe progress-enabled ARNs are that they are consistent (Theo. 12) and closed under composition provided that interconnections are made through channels that are safe and publication-enabled and over interaction-points in relation to which the ARNs are delivery-enabled (Cor. 13).

We also investigated the nature of the logics that should be used for specifying service interfaces and describing the processes and channels through which services are orchestrated. In particular, we exhibited a fragment of LTL in which only safety properties can be specified and argued that this fragment is expressive enough for the typical properties through which service interfaces are specified. In this setting, binding services, through the provides-points of their interfaces, to requires-points of the interfaces of discovered services, leads to a consistent composition of the service orchestrations.

Finally, we showed that, by using a logic such as SAFE-LTL, closed reduced NBAs can be used as models of implementations of safe processes and channels, and that checking processes/channels for delivery/publication enabledness can be done over those automata with a complexity that is in the order of the product of the size of the automata and of the sub-languages of deliveries/publications. Equally importantly, these checks can be made at design time, i.e., when implementations are chosen for orchestrating service interfaces. Therefore, there is no need for any additional checking to be made at discovery/run time to guarantee consistency; the only checking that needs to be made at run time is that the specifications of provides-points entail the specifications of the corresponding requires-points.

One point that we intend to investigate further concerns the interplay between consistency, safety, and the behavioural model. We intend to explore the use of sub-domains of traces that are applicable to SOC and generalise the underlying time model (and associated logic) using the notion of 'safety relative to a given condition' developed in [12]. Choosing a sub-domain can have an impact in the structure of the automata and the complexity of checking that processes and channels satisfy delivery/publication enabledness (and that ARNs orchestrate service interfaces), which are aspects that we did not have space left in the paper to analyse and explain in full.

Acknowledgments

We would like to thank Nir Piterman for many helpful comments and suggestions. This work was partially supported by FCT under contract (PTDC/EIA-EIA/103103/2008) and by the Tracing Networks research programme funded by the Leverhulme Trust.

References

1. B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.
2. B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.
3. R. Alur, K. Etessami, S. L. Torre, and D. Peled. Parametric temporal logic for "model measuring". *ACM Trans. Comput. Log.*, 2(3):388–407, 2001.
4. C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.
5. A. Betin-Can, T. Bultan, and X. Fu. Design for verification for asynchronously communicating web services. In Ellis and Hagino [9], pages 750–759.
6. D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web service interfaces. In Ellis and Hagino [9], pages 148–159.
7. T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW*, pages 403–410, 2003.
8. L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In T. A. Henzinger and C. M. Kirsch, editors, *EMSOFT*, volume 2211 of *LNCS*, pages 148–165. Springer, 2001.
9. A. Ellis and T. Hagino, editors. *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*. ACM, 2005.
10. J. L. Fiadeiro and A. Lopes. An interface theory for service-oriented design. In D. Giannakopoulou and F. Orejas, editors, *FASE*, volume 6603 of *LNCS*, pages 18–33. Springer, 2011.
11. X. Fu, T. Bultan, and J. Su. Conversation protocols: a formalism for specification and verification of reactive electronic services. *Theor. Comput. Sci.*, 328(1-2):19–37, 2004.
12. T. A. Henzinger. Sooner is safer than later. *Inf. Process. Lett.*, 43(3):135–141, 1992.
13. R. Kazhamiakin, M. Pistore, and L. Santuari. Analysis of communication models in web service compositions. In L. Carr, D. D. Roure, A. Iyengar, C. A. Goble, and M. Dahlin, editors, *WWW*, pages 267–276. ACM, 2006.
14. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.
15. O. Kupferman, N. Piterman, and M. Y. Vardi. From liveness to promptness. *Formal Methods in System Design*, 34(2):83–103, 2009.
16. A. Martens. Process oriented discovery of business partners. In C.-S. Chen, J. Filipe, I. Seruca, and J. Cordeiro, editors, *ICEIS (3)*, pages 57–64, 2005.
17. OSOA. Service component architecture: Building systems using a service oriented architecture, 2005. White paper available from www.osoa.org.
18. J. Ouaknine and J. Worrell. Safety metric temporal logic is fully decidable. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920 of *LNCS*, pages 411–425. Springer, 2006.
19. A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Asp. Comput.*, 6(5):495–512, 1994.
20. O. W. TC. Web services business process execution language, 2007. Version 2.0. Technical report, OASIS.
21. M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.