A full operational semantics for Asynchronous Relational Networks*

Ignacio Vissani[1], Carlos G. Lopez Pombo[1,2],
Ionuţ Ţuţu[3,4], and José Luiz Fiadeiro[3]

[1] Departamento de Computación, Universidad de Buenos Aires
[2] Consejo Nacional de Investigaciones Científicas y Tecnológicas (CONICET)
[3] Department of Computer Science, Royal Holloway University of London
[4] Institute of Mathematics of the Romanian Academy,
Research group of the project ID-3-0439
ivissani@dc.uba.ar clpombo@dc.uba.ar
ittutu@gmail.com Jose.Fiadeiro@rhul.ac.uk

Abstract. Service-oriented computing is a new paradigm where applications run over global computational networks and are formed by services discovered and bound at run-time through the intervention of a middleware. *Asynchronous Relational Nets* (ARNs) were presented by Fiadeiro and Lopes with the aim of formalising the elements of an interface theory for service-oriented software designs. The semantics of ARNs was originally given in terms of sequences of sets of actions corresponding to the behaviour of the service. Later, they were given an institution-based semantics where signatures are ARNs and models are morphisms into ground networks, that have no dependencies on external services.
In this work, we propose a full operational semantics capable of reflecting the dynamic nature of service execution by making explicit the reconfigurations that take place at run-time as the result of the discovery and binding of required services. This provides us a refined view of the execution of ARNs based upon which a specialized variant of linear temporal logic can be used to express, and even to verify through standard model-checking techniques, properties concerning the behaviour of ARNs that are more complex than those considered before.

1 Introduction and Motivation

In the context of global ubiquitous computing, the structure of software systems is becoming more and more dynamic as applications need to be able to respond and adapt to changes in the environment in which they operate. For instance, the new paradigm of *Service-Oriented Computing* (SOC) supports a new generation of software applications that run over globally available computational and network infrastructures where they can procure services on the fly (subject

* This work has been supported by the European Union Seventh Framework Programme under grant agreement no. 295261 (MEALS).

to a negotiation of *Service Level Agreements*, or SLAs for short) and bind to them so that, collectively, they can fulfil given business goals [1]. There is no control as to the nature of the components that an application can bind to. In particular, development no longer takes place in a top-down process in which subsystems are developed and integrated by skilled engineers: in SOC, discovery and binding are performed by middleware.

Asynchronous Relational Networks (ARNs) were presented by Fiadeiro and Lopes in [2] with the aim of formalising the elements of an interface theory for service-oriented software designs. ARNs are a formal orchestration model based on hypergraphs whose hyperedges are interpreted either as processes or as communication channels. The nodes (or points) that are only adjacent to process hyperedges are called *provides-points*, while those adjacent only to communication hyperedges are called *requires-points*. The rationale behind this separation is that a provides-point is the interface through which a service exports its functionality, while a requires-point is the interface through which an activity expects certain service to provide a functionality. The composition of ARNs (i.e., the binding mechanism of services) is obtained by "fusing" provides-points and requires-points, subject to a certain compliance check between the contract associated to them. For example, in [3] the binding is subject to a (semantic) entailment relation between theories over *linear temporal logic* [4], which are attached to the provides- and the requires-points of the considered networks.

Providing semantics to ARNs requires to carefully combine different elements intervening in the rationale behind the formalism and its intended behaviour. In their first definition, ARNs were given semantics in terms of infinite sequences of sets of actions, which capture the behaviour of the service. In this presentation, the behavioural description was given in terms of linear temporal logic theory presentations [2]. A more modern (and also more operational) presentation of the semantics of ARNs, the one on which we rely in this article, resorts to automata on infinite objects whose inputs consist of sequences of sets of actions (see [3]), as defined in the original semantics of ARNs. Under this formalism, both types of hyperedges are labelled with Muller automata; in the case of process hyperedges, the automata formalise the computation carried out by that particular service, while in the case of communication hyperedges, the automata represent the orchestrator that syncs the behaviour of the participants in the communication process. The behaviour of the system is then obtained as the composition of the Muller automata associated to both computation and communication hyperedges. Finally, the reconfiguration of networks (realized through the discovery and binding of services) is defined by considering an institutional framework in which signatures are ARNs and models are morphisms into ground ARNs, which have no dependencies on external services (see, e.g., [3] for a more in depth presentation of this semantics).

Under the above-mentioned consideration, the operational semantics of ARNs (as a set of execution traces) is based on the fact that a network can be reconfigured until all its external dependencies (captured by requires-points) are fulfilled, i.e., the original network admits a morphism to a ground ARN. In our work, we

consider that semantics is assigned modulo a given repository of services, forcing us to drop the assumption that given an ARN it is possible to find a ground network to which the former has a morphism. Regarding previous works, we believe that this approach results in a more realistic executing environment where the potential satisfaction of requirements is limited by the services registered in a repository, and not by the entire universe of possible services.

The aim of this work is to provide a trace-based operational semantics for service-oriented software designs reflecting the true dynamic nature of run-time discovery and binding of services. This is done by making the reconfiguration of an activity an observable event of its behaviour. In SOC, the reconfiguration events are triggered by particular actions associated with a requires-point; at that moment, the middleware has to procure a service that meets the requirements of the activity from an already known repository of services. From this perspective our proposal is to define execution traces where actions can be either

- internal actions of the activity: actions that are not associated with requires-points, thus executable without the need for reconfiguring the activity, or
- reconfiguration actions: actions that are associated with a requires-point, thus triggering the reconfiguration of the system by means of the discovery and binding of a service providing that computation.

Summarising, the main contributions of this paper are: 1) we provide a trace-based operational semantics for ARNs reflecting both internal transitions taking place in any of the services already intervening in the computation and dynamic reconfiguration actions resulting from the process by binding the provides-point of ARNs taken from the repository to its require-points, and 2) we provide support for defining a model-checking technique that can enable the automatic analysis of linear temporal logic properties of activities.

In this way, our work departs from previous approaches to dynamic reconfiguration in the context of service-oriented computing, such as [5], which reasons about functional behaviour and control concerns in a framework based on first-order logic, [6], which relies on typed graph-transformation techniques implemented in Alloy [7] and Maude [8],[9], which makes use of graph grammars as a formal framework for dealing with dynamicity, and [10, 11], which proposes architectural design rewriting as a term-rewriting-based approach to the development and reconfiguration of software architectures. A survey of these general logic-, graph-, or rewriting-based formalisms can be found in [12].

The article is organised as follows. In Sec. 2 we recall the preliminary notions needed for our work. In Sec. 3 we give appropriate definitions for providing operational semantics for ARNs based on a (quasi) automaton *generated* by a repository and on the traces accepted by it. We also provide a variant of Linear Temporal Logic (in Sec. 4) that is suitable for defining and checking properties related to the execution of activities. As a running example, we gradually introduce the details of travel-agent scenario, which we use to illustrate the concepts

presented in the Sec. 3 and 4. Finally in Sec. 5 we draw some conclusions and discuss further lines of research.

2 Preliminary Definitions

In this section we present the preliminary definitions we use throughout this work. We assume the reader has a nodding acquaintance of the basic definitions of category theory. Most of the definitions needed throughout the forthcoming sections can be found in [13–15]. For hypergraph terminology, notation and definitions, the reader is pointed to [16, 17], while for automata on infinite sequences we suggest [18, 19].

Definition 1 (Muller automaton). *The category* $\mathbb{MA}$ *of (action-based) Muller automata (see, e.g. [3]) is defined as follows:*

The objects *of* $\mathbb{MA}$ *are pairs* $\langle A, \Lambda \rangle$ *consisting of a set* A *of* actions *and a* Muller automaton $\Lambda = \langle Q, 2^A, \Delta, I, \mathcal{F} \rangle$ *over the alphabet* 2^A*, where*

- Q *is the set of* states *of* Λ*,*
- $\Delta \subseteq Q \times 2^A \times Q$ *is the* transition relation *of* Λ*, with transitions* $(p, \iota, q) \in \Delta$ *usually denoted by* $p \xrightarrow{\iota} q$*,*
- $I \subseteq Q$ *is the set of* initial states *of* Λ*, and*
- $\mathcal{F} \subseteq 2^Q$ *is the set of* final-state sets *of* Λ*.*

For every pair of Muller automata $\langle A, \Lambda \rangle$ *and* $\langle A', \Lambda' \rangle$*, with* $\Lambda = \langle Q, 2^A, \Delta, I, \mathcal{F} \rangle$ *and* $\Lambda' = \langle Q', 2^{A'}, \Delta', I', \mathcal{F}' \rangle$*, an* $\mathbb{MA}$-morphism $\langle \sigma, h \rangle \colon \langle A, \Lambda \rangle \to \langle A', \Lambda' \rangle$ *consists of functions* $\sigma \colon A \to A'$ *and* $h \colon Q' \to Q$ *such that* $(h(p'), \sigma^{-1}(\iota'), h(q')) \in \Delta$ *whenever* $(p', \iota', q') \in \Delta'$*,* $h(I') \subseteq I$*, and* $h(\mathcal{F}') \subseteq \mathcal{F}$*.*

The composition *of* $\mathbb{MA}$*-morphisms is defined componentwise.*

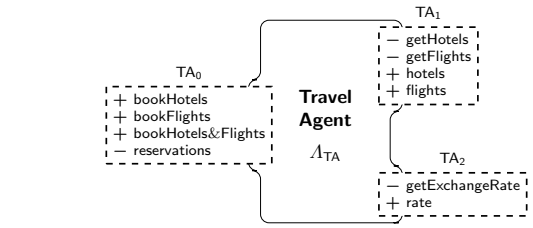
As we mentioned before, in this work we focus on providing semantics to service-oriented software artefacts. To do that, we resort to the formal language of *asynchronous relational nets* (see, e.g., [2]). The intuition behind the definition is that ARNs are hypergraphs where the hyperedges are divided in two classes: computation hyperedges and communication hyperedges. Computation hyperedges represent processes, while communication hyperedges represent communication channels. Hypergraph nodes (also called points) are labelled with *ports*, i.e., with structured sets $M = M^- \cup M^+$ of *publication* (M^-) and *delivery messages* (M^+),[5] along the lines of [20, 21]. At the same time, hyperedges are labelled with Muller automata; thus, both processes and communication channels have an associated behaviour given by their corresponding automata, which interact through (messages defined by) the ports labelling their connected points.

The following definitions formalise the manner in which the computation and communication units are structured to interact with each other.

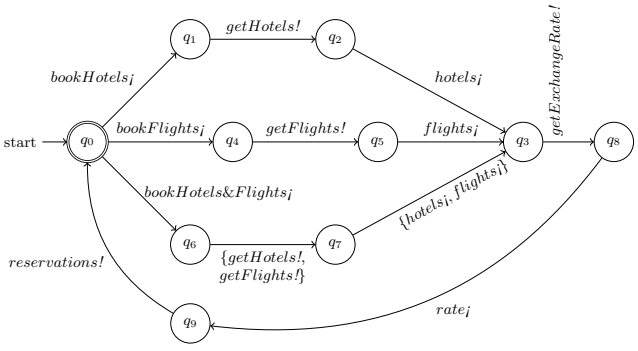
[5] Formally, we can define ports as sets M of messages together with a function $M \to \{-, +\}$ that assigns a polarity to every message.

Definition 2 (Process). *A* process $\langle \gamma, \Lambda \rangle$ *consists of a set* γ *of pairwise disjoint ports and a Muller automaton* Λ *over the set of actions* $A_\gamma = \bigcup_{M \in \gamma} A_M$*, where* $A_M = \{m! \mid m \in M^-\} \cup \{m? \mid m \in M^+\}$.

As an example, Fig. 1 (a) depicts a process $\langle \gamma_{\mathsf{TA}}, \Lambda_{\mathsf{TA}} \rangle$ where $\gamma_{\mathsf{TA}} = \{\mathsf{TA}_0, \mathsf{TA}_1, \mathsf{TA}_2\}$ and Λ_{TA} is the automaton presented in Fig.1 (b). The travel agent is meant to provide hotel and/or flight bookings in the local currency of the customers. Accomplishing this task requires two different interactions to take place: on one hand, the communication with hotel-accommodation providers and with flight-tickets providers, and on the other hand, the communication with a currency-converter provider. In order for the composition of the automata developed along our example to behave well, we need that every automaton is able to stay in any state indefinitely. This behaviour is achieved by forcing every state to have a self-loop labelled with the emptyset. With the purpose of easing the figures we avoid drawing these self-loops. The reader should still understand the descriptions of the automata as if there was a self-loop transition, labelled with the empty set, for every state.



(a)



(b)

Fig. 1. The TravelAgent process together with its automaton Λ_{TA}

Definition 3 (Connection). *Let* γ *be a set of pairwise disjoint ports. A* connection $\langle M, \mu, \Lambda \rangle$ *between the ports of* γ *consists of a set* M *of messages, a*

partial attachment injection $\mu_i \colon M \rightharpoonup M_i$ *for each port* $M_i \in \gamma$*, and a Muller automaton* Λ *over* $A_M = \{m! \mid m \in M\} \cup \{m¡ \mid m \in M\}$ *such that*

$$(a)\ M = \bigcup_{M_i \in \gamma} \mathrm{dom}(\mu_i) \quad \textit{and} \quad (b)\ \mu_i^{-1}(M_i^{\mp}) \subseteq \bigcup_{M_j \in \gamma \setminus \{M_i\}} \mu_j^{-1}(M_j^{\pm}).$$

In Fig. 2 (a) a connection C_0 is shown whose set of messages is the union of the messages of the ports $\mathsf{TA}_1, \mathsf{H}_0, \mathsf{F}_0$ and the family of mappings μ is formed by the trivial identity mapping. In Fig. 2 (b) the automaton Λ_{C_0} that describes the behaviour of the communication channel is shown. This connection just delivers every published message. Nevertheless it imposes some restrictions to the sequences of messages that can be delivered. For example notice that, after the message getHotels of TA_1 is received (and delivered), only the message hotels of H_0 is accepted for delivery.

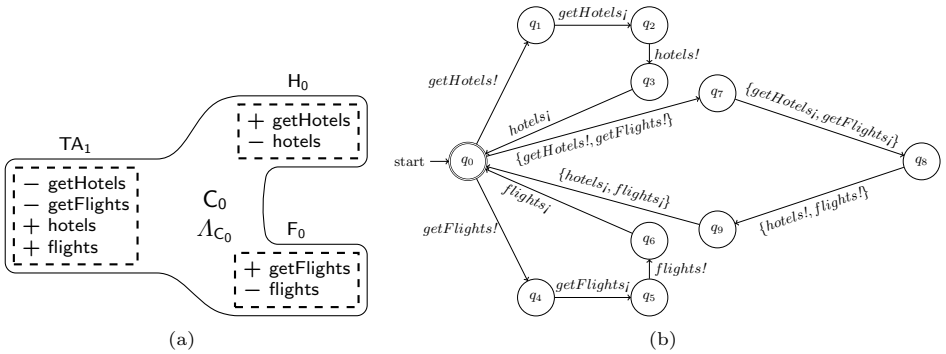


Fig. 2. The C_0 connection

With these elements we can now define asynchronous relational networks.

Definition 4 (Asynchronous Relational Net [3]). *An* asynchronous relational net $\alpha = \langle X, P, C, \gamma, M, \mu, \Lambda \rangle$ *consists of*

- *a hypergraph* $\langle X, E \rangle$*, where* X *is a (finite) set of* points *and* $E = P \cup C$ *is a set of* hyperedges *(non-empty subsets of* X*) partitioned into* computation hyperedges $p \in P$ *and* communication hyperedges $c \in C$ *such that no adjacent hyperedges belong to the same partition, and*
- *three labelling functions that assign* (a) *a port* M_x *to each point* $x \in X$*,* (b) *a process* $\langle \gamma_p, \Lambda_p \rangle$ *to each hyperedge* $p \in P$*, and* (c) *a connection* $\langle M_c, \mu_c, \Lambda_c \rangle$ *to each hyperedge* $c \in C$*.*

Definition 5 (Morphism of ARNs). *A* morphism $\delta \colon \alpha \to \alpha'$ *between two ARNs* $\alpha = \langle X, P, C, \gamma, M, \mu, \Lambda \rangle$ *and* $\alpha' = \langle X', P', C', \gamma', M', \mu', \Lambda' \rangle$ *consists of*

- *an injective map* $\delta \colon X \to X'$ *such that* $\delta(P) \subseteq P'$ *and* $\delta(C) \subseteq C'$*, that is an injective homomorphism between the underlying hypergraphs of* α *and* α' *that preserves the computation and communication hyperedges, and*

– *a family of polarity-preserving injections* $\delta_x^{pt} \colon M_x \to M'_{\delta(x)}$*, for* $x \in X$*,*

such that

– *for every point* $x \in \bigcup P$*,* $\delta_x^{pt} = 1_{M_x}$*,*
– *for every computation hyperedge* $p \in P$*,* $\Lambda_p = \Lambda'_{\delta(p)}$*, and*
– *for every communication hyperedge* $c \in C$*,* $M_c = M'_{\delta(c)}$*,* $\Lambda_c = \Lambda'_{\delta(c)}$ *and, for every point* $x \in \gamma_c$*,* $\mu_{c,x}; \delta_x^{pt} = \mu'_{\delta(c),\delta(x)}$*.*

ARNs together with morphisms of ARNs form a category, denoted $\mathbb{ARN}$, in which the composition is defined component-wise, and left and right identities are given by morphisms whose components are identity functions.

Intuitively, an ARN is a hypergraph for which some of the hyperedges (process hyperedges) formalise computations as Muller automata communicating through ports (identified with nodes of the hypergraph) over a fixed language of actions. Note that the communication between computational units is not established directly but mediated by a communication hyperedge; the other kind of hyperedge which use Muller automata to formalise communication channels.

In order to define service modules, repositories, and activities, we need to distinguish between two types of interaction-points, i.e. of points that are not incident with both computation and communication hyperedges.

Definition 6 (Requires- and provides-point). *A* requires-point *of an ARN is a point that is incident only with a communication hyperedge. Similarly, a* provides-point *is a point incident only with a computation hyperedge.*

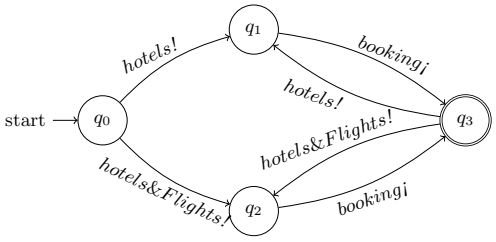
Definition 7 (Service repository). *A* service repository *is just a set* $\mathcal{R}$ *of* service modules*, that is of triples* $\langle P, \alpha, R \rangle$*, also written* $P \underset{\alpha}{\longleftarrow} R$*, where* α *is an ARN,* P *is a provides-point of* α*, and* R *is a finite set of requires-points of* α*.*

Definition 8 (Activity). *An* activity *is a pair* $\langle \alpha, R \rangle$*, also denoted* $\underset{\alpha}{\longmapsto} R$*, such that* α *is an ARN and* R *is a finite set of requires-points of* α*.*

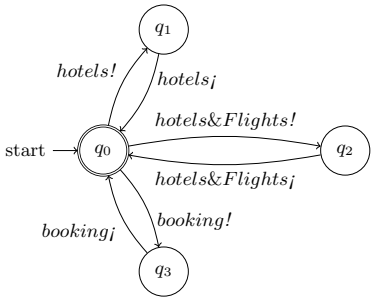
The previous definitions formalise the idea of a service-oriented software artefact as an activity whose computational requirements are modelled by "dangling" connections, and that do not pursue the provision of any service to other computational unit, modelled as the absence of provides points. Fig. 3 depicts a TravelClient activity with a single requires-point through which this activity can ask either for hotels or hotels and flights reservations. As we will show in the forthcoming sections, requires-points act as the ports to which the provides-points of services are bound in order to fulfil these requirements.

Turning a process into a service available for discovery and binding requires, as we mentioned in the previous definitions, the declaration of the communication channels that will be used to connect to other services. In the case of TravelAgent, three services are required to execute, communicating over two different communication channels. On one of them the process interacts with accommodation providers and flight tickets providers, while through the other the process will obtain exchange rates to be able to show the options to the

(a) The TravelClient process.

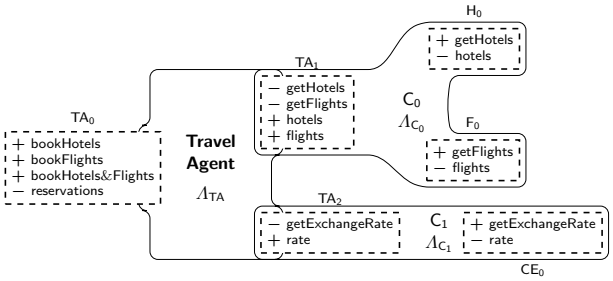


(b) The automaton Λ_{TC}

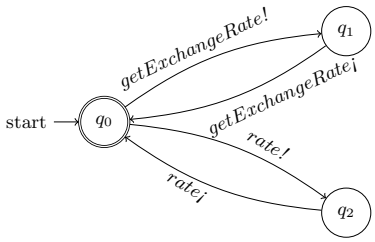


(c) The automaton Λ_{CC}

Fig. 3. The TravelClient activity.



(a) The TravelAgent service.



(b) The automaton Λ_{C_1}.

Fig. 4. The TravelAgent service module.

customer in its local currency. In some sense, TravelAgent provides the ability to coherently combine these three services in order to offer a richer experience. It should then be clear that whenever the TravelAgent is asked for hotels and flights reservations, it will require both services in order to fulfil its task, plus the service for currency exchange conversion. Fig. 4 (a) shows the TravelAgent service obtained by attaching the communication channels to two of the ports defined by the TravelAgent process, resulting in a network with three requires-points. The Fig. 4 (b) shows the automaton for the connection C_1.

3 Operational Semantics for ARNs

In this section we present the main contribution of the paper, being a full operational semantics for activities executing with respect to a given repository. To do this, we introduce two different kinds of transitions for activities: 1) internal transitions, those resulting from the execution of a certain set of actions by the automata that synchronise over them, and 2) reconfiguration transitions, the ones resulting from the need of executing a set of actions on a port of a communication hyperedge. Then, runs (on given traces) are legal infinite sequences of states related by appropriate transitions.

Definition 9 (Alphabet of an ARN). *The* alphabet associated with an ARN α *is the vertex* A_α *of the colimit* $\xi\colon D_\alpha \Rightarrow A_\alpha$ *of the functor* $D_\alpha\colon \mathbb{J}_\alpha \to \mathbb{S}\mathrm{et}$*, where*

- $\mathbb{J}_\alpha$ *is the preordered category whose objects are points* $x \in X$*, hyperedges* $e \in P \cup C$*, or "attachments"* $\langle c, x\rangle$ *of* α*, with* $c \in C$ *and* $x \in \gamma_c$*, and whose arrows are given by* $\{x \to p \mid p \in P, x \in \gamma_p\}$*, for computation hyperedges, and* $\{c \leftarrow \langle c, x\rangle \to x \mid c \in C, x \in \gamma_c\}$*, for communication hyperedges;*
- D_α *defines the sets of actions associated with the ports, processes and channels, together with the appropriate mappings between them.*

Definition 10 (Automaton of an ARN). *Let* $\alpha = \langle X, P, C, \gamma, M, \mu, \Lambda\rangle$ *be an ARN and* $\langle Q_e, 2^{A_{M_e}}, \Delta_e, I_e, \mathcal{F}_e\rangle$ *be the components of* Λ_e*, for each* $e \in P \cup C$*. The automaton* $\Lambda_\alpha = \langle Q_\alpha, 2^{A_\alpha}, \Delta_\alpha, I_\alpha, \mathcal{F}_\alpha\rangle$ associated with α *is defined as follows:*

$$
\begin{aligned}
Q_\alpha &= \textstyle\prod_{e\in P\cup C} Q_e,\\
\Delta_\alpha &= \{(p, \iota, q) \mid (\pi_e(p), \xi_e^{-1}(\iota), \pi_e(q)) \in \Delta_e \text{ for each } e \in P \cup C\},\\
I_\alpha &= \textstyle\prod_{e\in P\cup C} I_e, \text{ and}\\
\mathcal{F}_\alpha &= \{F \subseteq Q_\alpha \mid \pi_e(F) \in \mathcal{F}_e \text{ for all } e \in P \cup C\},
\end{aligned}
$$

where $\pi_e\colon Q_\alpha \to Q_e$ *are the corresponding projections of the product* $\prod_{e\in P\cup C} Q_e$*.*

Fact 1. Under the notations of Def. 10, for every hyperedge e of α, the maps ξ_e and π_e define an $\mathbb{M}\mathbb{A}$-morphism $\langle \xi_e, \pi_e\rangle\colon \langle A_{M_e}, \Lambda_e\rangle \to \langle A_\alpha, \Lambda_\alpha\rangle$.

Intuitively, the automaton of an ARN is the automaton resulting from taking the product of the automata of the several components of the ARN. This product is synchronized over the shared alphabet of the components. Notice that the notion of *shared alphabet* is given by the mappings defined in the connections.

Proposition 1. *For every ARN $\alpha = \langle X, P, C, \gamma, M, \mu, \Lambda\rangle$, the $\mathbb{MA}$-morphisms $\langle \xi_e, \pi_e\rangle \colon \langle A_{M_e}, \Lambda_e\rangle \to \langle A_\alpha, \Lambda_\alpha\rangle$ associated with hyperedges $e \in P \cup C$ form colimit injections for the functor $G_\alpha \colon \mathbb{J}_\alpha \to \mathbb{MA}$ that maps*

- *every computation or communication hyperedge $e \in P \cup C$ to $\langle A_{M_e}, \Lambda_e\rangle$ and*
- *every point $x \in X$ (or attachment $\langle c, x\rangle$) to $\langle A_{M_x}, \Lambda_x\rangle$, where Λ_x is the Muller automaton $\langle \{q\}, 2^{A_{M_x}}, \{(q, \iota, q) \mid \iota \subseteq A_{M_x}\}, \{q\}, \{\{q\}\}\rangle$ with only one state, which is both initial and final, and with all possible transitions.*

Therefore, both the alphabet and the automaton of an ARN α are given by the vertex $\langle A_\alpha, \Lambda_\alpha\rangle$ of a colimiting cocone of the functor $G_\alpha \colon \mathbb{J}_\alpha \to \mathbb{MA}$.

The universality property discussed above of the alphabet and of the automaton of an ARN allows us to extend Def. 10 to morphisms of networks.

Corollary 1. *For every morphism of ARNs $\delta \colon \alpha \to \alpha'$ there exists a unique $\mathbb{MA}$-morphism $\langle A_\delta, \_\restriction_\delta\rangle \colon \langle A_\alpha, \Lambda_\alpha\rangle \to \langle A_{\alpha'}, \Lambda_{\alpha'}\rangle$ such that*

$$(a)\ \ \xi_x ; A_\delta = \xi'_{\delta(x)} \quad \textit{and} \quad (b)\ \ (\_\restriction_\delta) ; \pi_x = \pi'_{\delta(x)}$$

for every point or hyperedge x of α, where $\langle \xi_x, \pi_x\rangle$ and $\langle \xi'_{x'}, \pi'_{x'}\rangle$ are components of the colimiting cocones of the functors $G_\alpha \colon \mathbb{J}_\alpha \to \mathbb{MA}$ and $G_{\alpha'} \colon \mathbb{J}_{\alpha'} \to \mathbb{MA}$.[6]

Operational semantics of ARNs. From a categorical perspective, the uniqueness aspect of Cor. 1 is particularly important in capturing the operational semantics of ARNs in a fully abstract manner: it enables us to describe both automata and morphisms of automata associated with ARNs and morphisms of ARNs through a functor $\mathcal{A} \colon \mathbb{ARN} \to \mathbb{MA}$ that maps every ARN α to $\langle A_\alpha, \Lambda_\alpha\rangle$ and every morphisms of ARNs $\delta \colon \alpha \to \alpha'$ to $\langle A_\delta, \_\restriction_\delta\rangle$.

3.1 Open Executions of ARNs

In order to formalise open executions of ARNs, i.e. of executions in which not only the states of the underlying automata of ARNs can change as a result of the publication or the delivery of various messages, but also the ARNs themselves through discovery and binding to other networks, we rely on the usual automata-theoretic notions of execution, trace, and run, which we consider over a particular (super-)automaton of ARNs and local states of their underlying automata.

Definition 11. *The "flattened" automaton $\mathcal{A}^\sharp = \langle Q^\sharp, A^\sharp, \Delta^\sharp, I^\sharp, \mathcal{F}^\sharp\rangle$ induced by the functor $\mathcal{A} \colon \mathbb{ARN} \to \mathbb{MA}$*[7] *is defined as follows:*

$$Q^\sharp = \{\langle \alpha, q\rangle \mid \alpha \in |\mathbb{ARN}| \textit{ and } q \in Q_\alpha\},$$
$$A^\sharp = \{\langle \delta, \iota\rangle \mid \delta \colon \alpha \to \alpha' \textit{ and } \iota \subseteq A_\alpha\},$$
$$\Delta^\sharp = \{(\langle \alpha, q\rangle, \langle \delta, \iota\rangle, \langle \alpha', q'\rangle) \mid \delta \colon \alpha \to \alpha' \textit{ and } (q, \iota, q'\restriction_\delta) \in \Delta_\alpha\},$$
$$I^\sharp = \{\langle \alpha, q\rangle \mid \alpha \in |\mathbb{ARN}| \textit{ and } q \in I_\alpha\}, \textit{ and}$$
$$\mathcal{F}^\sharp = \{\{\langle \alpha, q\rangle \mid q \in F\} \mid \alpha \in |\mathbb{ARN}| \textit{ and } F \in \mathcal{F}_\alpha\}.$$

[6] The definitions of G_α and $G_{\alpha'}$ follow the presentation given in Prop. 1.

[7] Note that $\Lambda^\sharp$ is in fact a quasi-automaton, because its components are proper classes.

This "flattened" automaton amalgamates in a single structure both the configuration and the state of the system. These two elements are viewed as a pair ⟨ARN, state⟩. Now the transitions in this automaton can represent state changes and structural changes together. In this sense, the "flattened" automaton achieves the goal of giving us a unified view of both aspects of a service oriented system. The construction of this automaton can be seen, from a categorical point of view, as the flattening of the indexed category induced by $\mathcal{A}\colon \mathbb{ARN} \to \mathbb{MA}$.

We recall that a *trace* over a set A of actions is an infinite sequence $\lambda \in (2^A)^\omega$, and that a *run* of a Muller automaton $\Lambda = \langle Q, 2^A, \Delta, I, \mathcal{F}\rangle$ on a trace λ is a sequence of states $\varrho \in Q^\omega$ such that $\varrho(0) \in I$ and $(\varrho(i), \lambda(i), \varrho(i+1)) \in \Delta$ for every $i \in \omega$; together, λ and ϱ form an *execution* of the automaton Λ. An execution $\langle \lambda, \rho\rangle$, or simply the run ϱ, is *successful* if the set of states that occur infinitely often in ϱ, denoted $\mathrm{Inf}(\varrho)$, is a member of $\mathcal{F}$. Furthermore, a trace λ is *accepted* by Λ if and only if there exists a successful run of Λ on λ.

Definition 12 (Open execution of an ARN). *An* open execution *of an ARN α is an execution of $\mathcal{A}^\sharp$ that starts from an initial state of Λ_α, i.e. a sequence*

$$\langle \alpha_0, q_0\rangle \xrightarrow{\delta_0,\iota_0} \langle \alpha_1, q_1\rangle \xrightarrow{\delta_1,\iota_1} \langle \alpha_2, q_2\rangle \xrightarrow{\delta_2,\iota_2} \cdots$$

such that $\alpha_0 = \alpha$, $q_0 \in I_\alpha$ and, for every $i \in \omega$, $\langle \alpha_i, q_i\rangle \xrightarrow{\langle \delta_i,\iota_i\rangle} \langle \alpha_{i+1}, q_{i+1}\rangle$ is a transition in $\Delta^\sharp$. An open execution as above is successful *if it is successful with respect to the automaton $\mathcal{A}^\sharp$, i.e. if there exists $i \in \omega$ such that (a) for all $j \geq i$, $\alpha_j = \alpha_i$, $\delta_j = 1_{\alpha_i}$, and (b) $\{q_j \mid j \geq i\} \in \mathcal{F}_{\alpha_i}$.*

Based on the definition of the transitions of $\mathcal{A}^\sharp$ and on the functoriality of $\mathcal{A}\colon \mathbb{ARN} \to \mathbb{MA}$, it is easy to see that, for every ARN α, every successful open execution of α gives a successful execution of its underlying automaton Λ_α.

Proposition 2. *For every (successful) open execution*

$$\langle \alpha_0, q_0\rangle \xrightarrow{\delta_0,\iota_0} \langle \alpha_1, q_1\rangle \xrightarrow{\delta_1,\iota_1} \langle \alpha_2, q_2\rangle \xrightarrow{\delta_2,\iota_2} \cdots$$

of the quasi-automaton $\mathcal{A}^\sharp$, the infinite sequence

$$q_0 \xrightarrow{\iota_0} q_1\restriction_{\delta_0} \xrightarrow{A_{\delta_0}^{-1}(\iota_1)} q_2\restriction_{\delta_0;\delta_1} \xrightarrow{A_{\delta_0;\delta_1}^{-1}(\iota_2)} \cdots$$

corresponds to a (successful) execution of the automaton Λ_{α_0}.

Note that, since the restrictions imposed to the transitions of $\mathcal{A}^\sharp$ are very weak – more precisely, because there are no constraints on the morphisms of ARN $\delta\colon \alpha \to \alpha'$ that underlie open-transitions $\langle \alpha, q\rangle \xrightarrow{\delta,\iota} \langle \alpha', q'\rangle$ – Prop. 2 cannot be generalised to executions of the automata Λ_{α_i}, for $i > 0$. To address this aspect, we need to take into consideration the fact that, in practice, the reconfigurations of ARNs are actually triggered by certain actions of their alphabet, and that they comply with the general rules of the process of service discovery and binding. Therefore, we need to consider open executions of activities with respect to given service repositories.

3.2 Open Executions of Activities

For the rest of this section we assume that $\mathcal{R}$ is an arbitrary but fixed repository of service modules.

Definition 13. *The* activity (quasi-)automaton $\mathcal{R}^\sharp = \langle Q^\mathcal{R}, A^\mathcal{R}, \Delta^\mathcal{R}, I^\mathcal{R}, \mathcal{F}^\mathcal{R} \rangle$ *generated by the service repository* $\mathcal{R}$ *is defined as follows:*

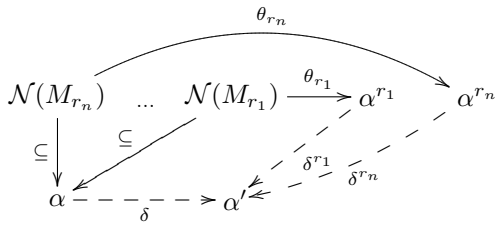
The states *in* $Q^\mathcal{R}$ *are pairs* $\langle \underset{\alpha}{\vdash\!\!\!-} R, q \rangle$*, where* $\underset{\alpha}{\vdash\!\!\!-} R$ *is an activity – i.e.* α *is an ARN and* R *is a finite set of requires-points of* α *– and* q *is a state of* Λ_α*.*

The alphabet $A^\mathcal{R}$ *is given by pairs* $\langle \delta, \iota \rangle$*, where* $\delta \colon \alpha \to \alpha'$ *is a morphism of ARNs and* ι *is a set of* α*-actions; thus,* $A^\mathcal{R}$ *is just the alphabet of* $\mathcal{A}^\sharp$*.*

There exists a transition $\langle \underset{\alpha}{\vdash\!\!\!-} R, q \rangle \xrightarrow{\delta,\iota} \langle \underset{\alpha'}{\vdash\!\!\!-} R', q' \rangle$ *whenever:*

1. $\langle \alpha, q \rangle \xrightarrow{\delta,\iota} \langle \alpha', q' \rangle$ *is a transition of* $\mathcal{A}^\sharp$*;*
2. *for each requires-point* $r \in R$ *such that* $\xi_r(A_{M_r^+}) \cap \iota \neq \emptyset$ *there exists*
   - *a service module* $P^r \underset{\alpha^r}{\vdash\!\!\!-} R^r$ *in* $\mathcal{R}$ *and*
   - *a polarity-preserving injection* $\theta_r : M_r \to M_{P^r}$

   such that the following colimit can be defined in the category of ARNs



   where $\{r_1, \ldots, r_n\}$ *is the biggest subset of* R *such that* $\xi_{r_i}(A_{M_{r_i}^+}) \cap \iota \neq \emptyset$ *for all* $1 \leq i \leq n$ *and* $\mathcal{N}(M_{r_i})$ *is the atomic ARN that consists of only one point, labelled with the port* M_{r_i}*, and no hyperedges;*

3. *there exists a transition* $p' \xrightarrow{\iota'} q'$ *of* $\Lambda_{\alpha'}$ *such that* $p' \restriction_\delta = q$*,* $A_\delta^{-1}(\iota') = \iota$ *and, for each requires-point* $r \in R$ *as above,* $p' \restriction_{\delta^r}$ *is an initial state of* Λ_{α^r}*.*

The states *in* $I^\mathcal{R}$ *are those pairs* $\langle \underset{\alpha}{\vdash\!\!\!-} R, q \rangle$ *for which* $q \in I_\alpha$*.*

The final-state sets *in* $\mathcal{F}^\mathcal{R}$ *are those sets* $\{\langle \underset{\alpha}{\vdash\!\!\!-} R, q \rangle \mid q \in F\}$ *for which* $F \in \mathcal{F}_\alpha$*.*

Note that the definition of the transitions of $\mathcal{R}^\sharp$ integrates both the operational semantics of ARNs given by the functor $\mathcal{A} \colon \mathbb{ARN} \to \mathbb{MA}$ and the logic-programming semantics of service discovery and binding described in [3], albeit in a simplified form, since here we do not take into account the linear temporal sentences that label requires-points. The removal of linear temporal sentences does not limit the applicability of the theory, but rather enables us to give a clearer and more concise presentation of the operational semantics of activities.

Open executions of activities can be defined relative to the automaton $\mathcal{R}^\sharp$ in a similar way to the open executions of ARNs (see Def. 12).

Definition 14 (Open execution of an activity). *An* open execution *of an activity* $\vdash\!\!\!\underset{\alpha}{}\, R$ *with respect to* $\mathcal{R}$ *is an execution of the quasi-automaton* $\mathcal{R}^\sharp$ *that starts from an initial state of* Λ_α*, i.e. a sequence of transitions of* $\mathcal{R}^\sharp$

$$\langle \vdash\!\!\!\underset{\alpha_0}{}\, R_0, q_0\rangle \xrightarrow{\delta_0,\iota_0} \langle \vdash\!\!\!\underset{\alpha_1}{}\, R_1, q_1\rangle \xrightarrow{\delta_1,\iota_1} \langle \vdash\!\!\!\underset{\alpha_2}{}\, R_2, q_2\rangle \xrightarrow{\delta_2,\iota_2} \cdots$$

such that $\alpha_0 = \alpha$*,* $R_0 = R$*, and* $q_0 \in I_\alpha$*. An open execution as above is* successful *if there exists* $i \in \omega$ *such that* (a) *for all* $j \geq i$*,* $\alpha_j = \alpha_i$*,* $\delta_j = 1_{\alpha_i}$*, and* (b) $\{q_j \mid j \geq i\} \in \mathcal{F}_{\alpha_i}$*.*

To illustrate open executions, let's consider a repository $\mathcal{R}$ formed by the service TravelAgent (depicted in Fig. 4) and the very simple services CurrenciesAgent, AccomodationAgent and FlightsAgent described in Fig. 5. Let's also consider the TravelClient activity of Fig. 3. Observing the automata of Fig. 3 (b) and 3 (c), an execution starts with the activity TravelClient performing one of two actions, *hotels*! or *hotels&Flights*!. Let us assume it is *hotels*! without loss of generality. The prefix of the execution after the transition has the following shape:

$$\langle \vdash\!\!\!\underset{\mathsf{TravelClient}}{}\, \{\mathsf{CC}_1\}, q_0\rangle \xrightarrow{id,hotels!} \langle \vdash\!\!\!\underset{\mathsf{TravelClient}}{}\, \{\mathsf{CC}_1\}, q_1\rangle$$

where q_0 and q_1 are the states (q_0, q_0) and (q_1, q_1) of the composed automaton $\Lambda_{TC} \times \Lambda_{CC}$ respectively. After this, the only plausible action in this run is the delivery of the message *hotels* by the communication channel CC. Since $\xi_{\mathsf{TravelClient}}(A_{M^+_{\mathsf{CC}_1}}) \cap \{hotels_i\} = \{hotels_i\}$ this action triggers a reconfiguration of the activity. In our example's repository, $\mathcal{R}$, the only service that can satisfy the requirement CC_1 is TravelAgent. Thus, the action $hotels_i$ leads us to the activity TravelClient′ shown in Fig. 6. The prefix of the execution after this last transition is:

$$\cdots \xrightarrow{id,hotels!} \langle \vdash\!\!\!\underset{\mathsf{TravelClient}}{}\, \{\mathsf{CC}_1\}, q_1\rangle \xrightarrow{\delta,hotels_i} \langle \vdash\!\!\!\underset{\mathsf{TravelClient}'}{}\, \{\mathsf{H}_0, \mathsf{F}_0, \mathsf{CE}_0\}, q_2\rangle$$

where q_2 is the state $(q_1, q_0, q_1, q_0, q_0)$ of the automaton of TravelClient′. To see that the morphism $\delta : \mathsf{TravelClient} \to \mathsf{TravelClient}'$ exists is straightforward.

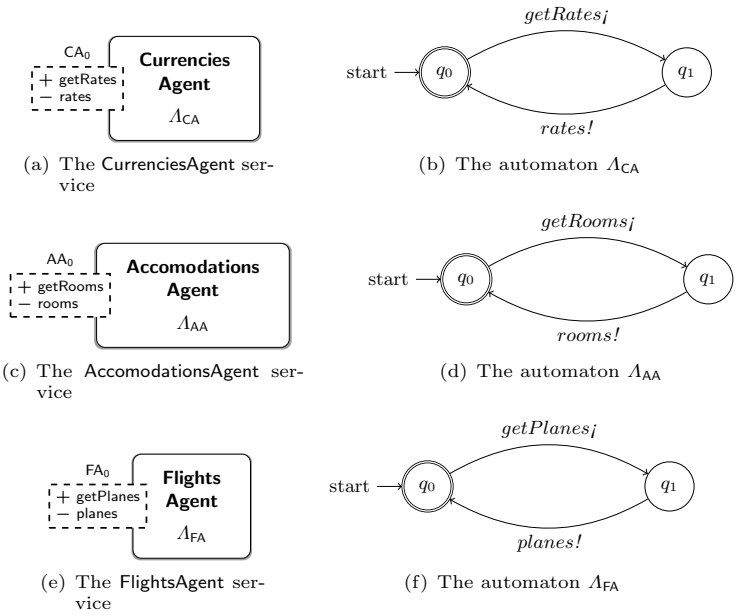
A continuation for this execution is obtained by the automaton Λ_{TA}, associated with TravelAgent, publishing the action *getHotels!* and the mandatory delivery *getHotels!* that comes after. This actions trigger a new reconfiguration of the activity on port H_0 of the communication channel C_0; in this case, and considering once again our repository $\mathcal{R}$, the result of the reconfiguration should be the *attachment* of the service module AccomodationsAgent.

The following fact allows us to easily generalise Prop. 2 from open executions of ARNs to open executions of activities.

Fact 2. There exists a (trivial) forgetful morphism of Muller automata $\mathcal{R}^\sharp \to \mathcal{A}^\sharp$ that maps every state $\langle \vdash\!\!\!\underset{\alpha}{}\, R, q\rangle$ of $\mathcal{R}^\sharp$ to the state $\langle \alpha, q\rangle$ of $\mathcal{A}^\sharp$.

Proposition 3. *For every (successful) execution*

$$\langle \vdash\!\!\!\underset{\alpha_0}{}\, R_0, q_0\rangle \xrightarrow{\delta_0,\iota_0} \langle \vdash\!\!\!\underset{\alpha_1}{}\, R_1, q_1\rangle \xrightarrow{\delta_1,\iota_1} \langle \vdash\!\!\!\underset{\alpha_2}{}\, R_2, q_2\rangle \xrightarrow{\delta_2,\iota_2} \cdots$$



(a) The CurrenciesAgent service

(b) The automaton Λ_{CA}

(c) The AccomodationsAgent service

(d) The automaton Λ_{AA}

(e) The FlightsAgent service

(f) The automaton Λ_{FA}

Fig. 5. Very simple services in $\mathcal{R}$

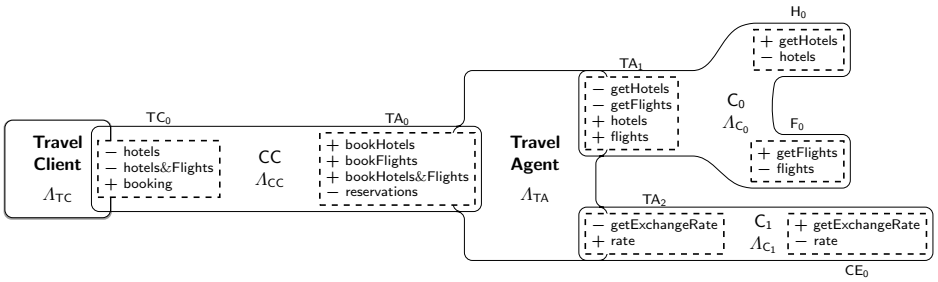


Fig. 6. The TravelClient′ activity

of the activity quasi-automaton $\mathcal{R}^\sharp$, the infinite sequence

$$q_0 \xrightarrow{\iota_0} q_1{\restriction_{\delta_0}} \xrightarrow{A_{\delta_0}^{-1}(\iota_1)} q_2{\restriction_{\delta_0;\delta_1}} \xrightarrow{A_{\delta_0;\delta_1}^{-1}(\iota_2)} \cdots$$

is a (successful) execution of the automaton Λ_{α_0}.

Theorem 1 shows the relation that exists between the traces of an activity with respect to a respository and the automata of each component of the activity. It shows that every (successful) open execution of an activity can be projected to a (successful) execution of each of the automata interveaning. In order to prove that open executions of activities give rise to "local" executions of Λ_{α_i} – for every $i \in \omega$, not only for $i = 0$ – we rely on a consequence of the fact that the functor $\mathcal{A}\colon \mathbb{ARN} \to \mathbb{MA}$ preserves colimits and, in addition, we restrict the automata associated with the underlying ARNs of service modules.

Proposition 4. *The functor $\mathcal{A}\colon \mathbb{ARN} \to \mathbb{MA}$ preserves colimits. In particular, for every transition $\langle \underset{\alpha}{\longmapsto} R, q\rangle \xrightarrow{\delta,\iota} \langle \underset{\alpha'}{\longmapsto} R', q'\rangle$ as in Def. 13, the Muller automaton $\Lambda_{\alpha'}$ is isomorphic with the product*

$$\Lambda_\alpha^\delta \times \prod_{\substack{r\in R\\ \xi_r(A_{M_r^+})\cap\iota\neq\emptyset}} \Lambda_{\alpha^r}^{\delta^r}$$

of the cofree expansions Λ_α^δ and $\Lambda_{\alpha^r}^{\delta^r}$, for $r \in R$ such that $\xi_r(A_{M_r^+}) \cap \iota \neq \emptyset$, of the automata Λ_α and Λ_{α^r} along the alphabet maps A_δ and A_{δ^r}, respectively.[8]

Consequently, a transition $p' \xrightarrow{\iota'} q'$ is defined in the automaton $\Lambda_{\alpha'}$ if and only if $p'{\restriction_\delta} \xrightarrow{A_\delta^{-1}(\iota')} q'{\restriction_\delta}$ is a transition of Λ_α and, for each $r \in R$ such that $\xi_r(A_{M_r^+}) \cap \iota \neq \emptyset$, $p'{\restriction_{\delta^r}} \xrightarrow{A_{\delta^r}^{-1}(\iota')} q'{\restriction_{\delta^r}}$ is a transition of Λ_{α^r}.

Definition 15 (Idle initial states). *An automaton $\Lambda = \langle Q, 2^A, \Delta, I, \mathcal{F}\rangle$ is said to have* idle initial states *if for every initial state $q \in I$ there exists a transition $(p, \emptyset, q) \in \Delta$ such that p is an initial state too.*

The following result can be proved by induction on $i \in \omega$. The base case results directly from Prop. 3, while the induction step relies on condition 3 of Def. 13 and on Prop. 4.

Theorem 1. *If, for every service module $P \underset{\alpha}{\longleftarrow} R$ in $\mathcal{R}$, the automaton Λ_α has idle initial states, then for every (successful) execution*

$$\langle \underset{\alpha_0}{\longmapsto} R_0, q_0\rangle \xrightarrow{\delta_0,\iota_0} \langle \underset{\alpha_1}{\longmapsto} R_1, q_1\rangle \xrightarrow{\delta_1,\iota_1} \langle \underset{\alpha_2}{\longmapsto} R_2, q_2\rangle \xrightarrow{\delta_2,\iota_2} \cdots$$

[8] We recall from [3] that the cofree expansion of an automaton $\Lambda = \langle Q, 2^A, \Delta, I, \mathcal{F}\rangle$ along a map $\sigma\colon A \to A'$ is the automaton $\Lambda' = \langle Q, 2^{A'}, \Delta', I, \mathcal{F}\rangle$ for which $(p, \iota', q) \in \Delta'$ if and only if $(p, \sigma^{-1}(X'), q) \in \Delta$.

of $\mathcal{R}^\sharp$ there exists a (successful) execution of Λ_{α_i}, for $i \in \omega$, of the form

$$q'_0 \xrightarrow{\iota'_0} q'_1 \xrightarrow{\iota'_1} \cdots q'_{i-1} \xrightarrow{\iota'_{i-1}} q_i \xrightarrow{\iota_i} q_{i+1}{\restriction_{\delta_i}} \xrightarrow{A^{-1}_{\delta_i}(\iota_{i+1})} q_{i+2}{\restriction_{\delta_i;\delta_{i+1}}} \xrightarrow{A^{-1}_{\delta_i;\delta_{i+1}}(\iota_{i+2})} \cdots$$

where, for every $j < i$, $q'_j{\restriction_{\delta_j;\cdots;\delta_{i-1}}} = q_j$ and $A^{-1}_{\delta_j;\cdots;\delta_{i-1}}(\iota'_j) = \iota_j$.

The reader should notice that all the automata used as examples in this work have *idle initial states* as a consequence of the hidden self loop, labelled with the empty set, that we assumed to exist in every state.

4 Satisfiability of Linear Temporal Logic Formulae

In this section we show how we can use the trace semantics we presented in the previous section to reason about Linear Temporal Logic (LTL for short) [4, 22] properties of activities. Next we define linear temporal logic by providing its grammar and semantics in terms of sets of traces.

Definition 16. *Let $\mathcal{V}$ be a set of proposition symbols, then the set of LTL formulae on $\mathcal{V}$, denoted as LTLForm($\mathcal{V}$), is the smallest set S such that:*

- *$\mathcal{V} \subseteq S$, and*
- *if $\phi, \psi \in S$, then $\{\neg\phi, \phi \vee \psi, \mathbf{X}\phi, \phi\mathbf{U}\psi\} \subseteq S$.*

We consider the signature of a repository to be the union of all messages of all the service modules in it. This can give rise to an infinite language over which it is possible to express properties refering to any of the services in the repository, even those that are not yet bound (and might never be). To achieve this we require the alphabets of the service modules in a repository $\mathcal{R}$ to be pairwise disjoint.

Definition 17. *Let $\mathcal{R}$ be a repository and $\underset{\alpha}{\vdash\!\!-} R$ an activity. We denote with $A_{\mathcal{R},\alpha}$ the set $\left(\bigcup \{A_{\alpha'}\}_{P' \xleftarrow[\alpha']{} R' \in \mathcal{R}}\right) \cup A_\alpha$*

Defining satisfaction of an LTL formula requires that we first define what is the set of propositions over which we can express the LTL formulae. We consider as the set of propositions all the actions in the signature of the repository or in the activity to which we are providing semantics. Thus, the propositions that hold in a particular state will be the ones that correspond to the actions in the label of the transition that took the system to that state.

In order to define if a run satisfies an LTL formula it is necessary to consider the suffixes of a run, thus let

$$r = \langle \underset{\alpha_0}{\vdash\!\!-} R_0, q_0 \rangle \xrightarrow{\delta_0,\iota_0} \langle \underset{\alpha_1}{\vdash\!\!-} R_1, q_1 \rangle \xrightarrow{\delta_1,\iota_1} \langle \underset{\alpha_2}{\vdash\!\!-} R_2, q_2 \rangle \xrightarrow{\delta_2,\iota_2} \cdots$$

be a succesful open execution of $\underset{\alpha}{\vdash\!\!-} R$ with respect to a repository $\mathcal{R}$ we denote with r_i the i^{th} suffix of r. That is:

$$r_i = \langle \underset{\alpha_i}{\vdash\!\!-} R_i, q_i \rangle \xrightarrow{\delta_i,\iota_i} \langle \underset{\alpha_{i+1}}{\vdash\!\!-} R_{i+1}, q_{i+1} \rangle \xrightarrow{\delta_{i+1},\iota_{i+1}} \langle \underset{\alpha_{i+2}}{\vdash\!\!-} R_{i+2}, q_{i+2} \rangle \xrightarrow{\delta_{i+2},\iota_{i+2}} \cdots$$

The thoughtful reader may notice that while our formulae are described over the union of the alphabet of the repository $\mathcal{R}$ and the alphabet of the activity $\underset{\alpha}{\vdash\!\!-} R$, the labels ι_i in a run belong to the alphabet A_{α_i}, that is the computed co-limit described in Def 9. Therefore, we need to *translate* our formula accordingly with the modifications suffered by the activity during the particular run to be able to check if it holds. In order to define how the translation of the formula is carried out we rely on the result of Cor. 1. The following definition provides the required notation to define these translations:

Definition 18. *Let $\mathcal{R}$ be a repository and $\langle \underset{\alpha}{\vdash\!\!-} R, q\rangle \xrightarrow{\delta,\iota} \langle \underset{\alpha'}{\vdash\!\!-} R', q'\rangle$ a transition of $\mathcal{R}^\sharp$ then we define $A_{\hat{\delta}} : A_{\mathcal{R},\alpha} \to A_{\mathcal{R},\alpha'}$ as*

$$A_{\hat{\delta}}(a) = \begin{cases} A_\delta(a) & a \in A_\alpha \\ A_{\delta^{r_i}}(a) & a \in A_{\alpha^{r_i}} \\ a & \textit{otherwise} \end{cases}$$

Definition 19. *Let $\mathcal{R}$ be a repository and let $\underset{\alpha}{\vdash\!\!-} R$ be an activity. Also let $\mathcal{V} = A_{\mathcal{R},\alpha}$, $\phi, \psi \in LTLForm(\mathcal{V})$, $a \in \mathcal{V}$ and $v \subseteq \mathcal{V}$ then:*

- $\langle r, v, \tau\rangle \models \mathbf{true}$,
- $\langle r, v, \tau\rangle \models a$ *iff* $\tau(a) \in v$,
- $\langle r, v, \tau\rangle \models \neg\phi$ *iff* $\langle r, v, \tau\rangle \not\models \phi$,
- $\langle r, v, \tau\rangle \models \phi \vee \psi$ *if* $\langle r, v, \tau\rangle \models \phi$ *or* $\langle r, v, \tau\rangle \models \psi$,
- $\langle r, v, \tau\rangle \models \mathbf{X}\phi$ *iff* $\langle r_1, \iota_0, \tau; A_{\hat{\delta}_0}\rangle \models \phi$, *and*
- $\langle r, v, \tau\rangle \models \phi\mathbf{U}\psi$ *iff there exists* $0 \leq i$ *such that* $\langle r_i, \iota_{i-1}, \tau; A_{\hat{\delta}_0}; ...; A_{\hat{\delta}_{i-1}}\rangle \models \psi$ *and for all* j, $0 \leq j < i$, $\langle r_j, \iota_{j-1}, \tau; A_{\hat{\delta}_0}; ...; A_{\hat{\delta}_{j-1}}\rangle \models \phi$ *where* $\iota_{-1} = \emptyset$ *and* $A_{\hat{\delta}_{-1}} = 1_{A_{\mathcal{R},\alpha}}$.

If $\mathcal{V}$ is a set of propositions, $\phi, \psi \in LTLForm(\mathcal{V})$, the rest of the boolean constants and operators are defined as usual as: $\mathbf{false} \equiv \neg\mathbf{true}$, $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$, $\phi \implies \psi \equiv \neg\phi \vee \psi$, etc. We define $\Diamond\phi \equiv \mathbf{true}\mathbf{U}\phi$ and $\Box\phi \equiv \neg(\mathbf{true}\mathbf{U}\neg\phi)$.

Definition 20. *Let $\mathcal{R}$ be a repository and let*

$$r = \langle \underset{\alpha_0}{\vdash\!\!-} R_0, q_0\rangle \xrightarrow{\delta_0,\iota_0} \langle \underset{\alpha_1}{\vdash\!\!-} R_1, q_1\rangle \xrightarrow{\delta_1,\iota_1} \langle \underset{\alpha_2}{\vdash\!\!-} R_2, q_2\rangle \xrightarrow{\delta_2,\iota_2} \cdots$$

be a successful open execution of $\mathcal{R}^\sharp$. Then a formula $\phi \in LTLForm(A_{\mathcal{R},\alpha_0})$ is satisfied by r ($r \models \phi$) if and only if $\langle r, \emptyset, 1_{A_{\mathcal{R},\alpha_0}}\rangle \models \phi$.

Following the previous definitions, checking if an activity $\underset{\alpha}{\vdash\!\!-} R$ satisfies a proposition ϕ under a repository $\mathcal{R}$ is equivalent to checking if every successful open execution of $\underset{\alpha}{\vdash\!\!-} R$ with respect to $\mathcal{R}$ satisfies ϕ.

In the following we will show how the satisfaction relation in Def. 20 can be used to reason about properties of activities. We are particularly interested in asserting properties regarding the future execution of an activity with respect to a repository.

In order to exemplify, let us once again consider the activity TravelClient of Fig. 3(a) and the repository $\mathcal{R}$ formed by the services TravelAgent, CurrenciesAgent, AccomodationAgent, and FlightsAgent described in Figs. 4 and 5. We are then interested in the open successful executions of the quasi-automaton $\mathcal{R}^\sharp$. Two examples of statements we could be interested in are the following properties:

1. Every execution of TravelClient requires the execution of CurrenciesAgent:
   For all successful open executions r of $\mathcal{R}^\sharp$, $r \models \Diamond \left(\bigvee_{a \in A_{M_{\mathsf{CurrenciesAgent}}}} a \right)$.
2. There exists an execution of TravelClient that does not require the execution of FlightsAgent:
   There exists a successful open execution r of $\mathcal{R}^\sharp$, $r \models \Box \left(\neg \bigvee_{a \in A_{M_{\mathsf{FlightsAgent}}}} a \right)$.

The first property is true and it can be checked by observing that in the automaton Λ_{TA} no matter what is the choice for a transition made in the initial state ($bookHotels_i$, $bookFlights_i$, or $bookHotels\&Flights_i$), the transition labelled with action $getExchangeRate!$ belongs to every path that returns to the initial state, that is the only accepting state. Therefore, the reconfiguration of the activity on port CE_0 is enforced in every successful execution.

The second one is also true as it states that there is an execution that does not requires the binding of a flights agent. Observing TravelClient, one can consider the trace in which no order on flights is placed never as the client always choose to order just accommodation.

5 Conclusions and Further Work

The approach that we put forward in this paper combines, in an integrated way, the operational semantics of processes and communication channels, and the dynamic reconfiguration of ARNs. As a result, it provides a full operational semantics of ARNs by means of automata on infinite sequences built from the local semantics of processes, together with the semantics of those ARNs that are selected from a given repository by means of service discovery and binding. Another use for this semantics is in identifying the differences between the nondeterministic behaviour of a component, reflected within the execution of an ARN, and the nondeterminism that arises from the discovery and binding to other ARNs.

In comparison with the logic-programming semantics of services described in [3], this gives us a more refined view of the execution of ARNs; in particular, it provides a notion of execution trace that reflects both internal actions taken by services that are already intervening in the execution of an activity, and dynamic reconfiguration events that result from triggering actions associated with a requires-point of the activity. In addition, by defining the semantics of an activity with respect to an arbitrary but fixed repository, it is also possible to describe and reason about the behaviour of those ARNs whose executions may not lead to ground networks, despite the fact that they are still sound and successful executions of the activity.

The proposed operational semantics allows us to use various forms of temporal logic to express properties concerning the behaviour of ARNs that surpasses those considered before. We showed this by defining a variant of the satisfaction relation for linear temporal logic, and exploiting the fact that reconfiguration actions are observable in the execution traces; thus, it is possible to determine whether or not a given service module of a repository is necessarily used, or may be used, during the execution of an activity formalised as an ARN.

Many directions for further research are still to be explored in order to provide an even more realistic execution environment for ARNs. Among them, in the current formalism, services are bound once and forever. In real-life scenarios services are bound only until they finish their computation (assuming that no error occurs); this does not prevent the activity to require the execution of the same action associated to the same requires-point, triggering a new discovery with a potential different outcome on the choice of the service to be bound. Also, our approach does not consider any possible change on the repository during the execution which leads to a naive notion of distributed execution as simple technical problems can make services temporarily unavailable.

References

1. Fiadeiro, J.L., Lopes, A., Bocchi, L.: An abstract model of service discovery and binding. Formal Aspects of Computing **23**(4) (2011) 433–463
2. Fiadeiro, J.L., Lopes, A.: An interface theory for service-oriented design. Theoretical Computer Science **503** (2013) 1–30
3. Ţuţu, I., Fiadeiro, J.L.: Service-oriented logic programming. Logical Methods in Computer Science (in press)
4. Pnueli, A.: The temporal semantics of concurrent programs. Theoretical Computer Science **13**(1) (1981) 45–60
5. Simonot, M., Aponte, V.: A declarative formal approach to dynamic reconfiguration. In: Proceedings of the 1st International Workshop on Open Component Ecosystems. IWOCE '09 (2009) 1–10
6. Bruni, R., Bucchiarone, A., Gnesi, S., Hirsch, D., Lluch-Lafuente, A.: Graph-based design and analysis of dynamic software architectures. In: Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday. Volume 5065 of Lecture Notes in Computer Science., Springer (2008) 37–56
7. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press (2006)
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L., eds.: All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. Volume 4350 of Lecture Notes in Computer Science., Springer (2007)
9. Bruni, R., Bucchiarone, A., Gnesi, S., Melgratti, H.C.: Modelling dynamic software architectures using typed graph grammars. Electr. Notes Theor. Comput. Sci. **213**(1) (2008) 39–53
10. Bruni, R., Lluch-Lafuente, A., Montanari, U., Tuosto, E.: Service oriented architectural design. In: Trustworthy Global Computing, Third Symposium, TGC 2007,

Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers. Volume 4912 of Lecture Notes in Computer Science., Springer (2007) 186–203
11. Gadducci, F.: Graph rewriting for the π-calculus. Mathematical Structures in Computer Science **17**(3) (2007) 407–437
12. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems, WOSS 2004, ACM (2004) 28–33
13. McLane, S.: Categories for working mathematician. Graduate Texts in Mathematics. Springer-Verlag, Berlin, Germany (1971)
14. Barr, M., Wells, C.: Category theory for computer science. Prentice Hall, London (1990)
15. Fiadeiro, J.L.: Categories for software engineering. Springer-Verlag (2005)
16. Cambini, R., Gallo, G., Scutellà, M.G.: Flows on hypergraphs. Mathematical Programming **78** (1997) 195–217
17. Ausiello, G., Franciosa, P.G., Frigioni, D.: Directed hypergraphs: Problems, algorithmic results, and a novel decremental approach. In Restivo, A., Rocca, S.R.D., Roversi, L., eds.: Proceedings of 7th Italian Conference on Theoretical Computer Science. Volume 2202 of Lecture Notes in Computer Science., Springer-Verlag (October 2001) 312–327
18. Thomas, W.: Automata on infinite objects. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics. Elsevier (1990) 133–192
19. Perrin, D., Pin, J.É.: Infinite Words: Automata, Semigroups, Logic and Games. Pure and Applied Mathematics. Elsevier Science (2004)
20. Brand, D., Zafiropulo, P.: On communicating finite-state machines. Journal of the ACM **30**(2) (1983) 323–342
21. Benatallah, B., Casati, F., Toumani, F.: Representing, analysing and managing Web service protocols. Data & Knowledge Engineering **58**(3) (2006) 327–357
22. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems. Springer-Verlag, New York (1995)