A Calculus for Orchestration of Web Services[☆]

Rosario Pugliese[a], Francesco Tiezzi[b,*]

[a] *Università degli Studi di Firenze, Viale Morgagni, 65 - 50134 Firenze, Italy*
[b] *IMT Institute for Advanced Studies Lucca, Piazza S. Ponziano. 6 - 55100 Lucca, Italy*

Abstract

Service-oriented computing, an emerging paradigm for distributed computing based on the use of services, is calling for the development of tools and techniques to build safe and trustworthy systems, and to analyse their behaviour. Therefore, many researchers have proposed to use process calculi, a cornerstone of current foundational research on specification and analysis of concurrent, reactive, and distributed systems. In this paper, we follow this approach and introduce C⏲WS, a process calculus expressly designed for specifying and combining service-oriented applications, while modelling their dynamic behaviour. We show that C⏲WS can model all the phases of the life cycle of service-oriented applications, such as publication, discovery, negotiation, orchestration, deployment, reconfiguration and execution. We illustrate the specification style that C⏲WS supports by means of a large case study from the automotive domain and a number of more specific examples drawn from it.

Keywords: Service-oriented computing, Formal methods, Process calculi

1. Introduction

Recently, the increasing success of e-business, e-learning, e-government, and other similar emerging models, has led the World Wide Web, initially thought of as a system for human use, to evolve towards an architecture for *Service-Oriented Computing* (SOC) supporting automated use. This emerging paradigm finds its origin in object-oriented and component-based software development, and aims at enabling developers to build networks of interoperable and collaborative applications, regardless of the platform where the applications run and of the programming language used to develop them, through the use of independent computational units, called *services*. Services are loosely coupled reusable components, that are built with little or no knowledge about clients and other services involved in their operating environment. SOC systems thus deliver application functionalities as services to either end-user applications or other services.

There are by now some successful and well-developed instantiations of the general SOC paradigm, like e.g. Web Services and Grid Computing, that exploit the pervasiveness of Internet

[☆] This work has been partially sponsored by the EU project ASCENS (257414) and by MIUR (PRIN 2009 DISCO).
[*] Corresponding author
Email addresses: rosario.pugliese@unifi.it (Rosario Pugliese), francesco.tiezzi@imtlucca.it (Francesco Tiezzi)
URL: http://www.dsi.unifi.it/~pugliese/ (Rosario Pugliese), http://www.imtlucca.it/francesco.tiezzi (Francesco Tiezzi)

and related standards. However, current software engineering technologies for SOC remain at the descriptive level and lack rigorous formal foundations. In the design of SOC systems we are still experiencing a gap between practice (programming) and theory (formal methods and analysis techniques). The challenges come from the necessity of dealing at once with such issues as asynchronous interactions, concurrent activities, workflow coordination, business transactions, failures, resource usage, and security, in a setting where demands and guarantees can be very different for the many different components. Many researchers have hence put forward the idea of using *process calculi*, a cornerstone of current foundational research on specification and analysis of concurrent, reactive and distributed systems through mathematical — mainly algebraic and logical — tools. Due to their algebraic nature, process calculi provide intuitive and concise notations, and convey in a distilled form the compositional programming style of SOC. Services are built in a compositional way by using the operators provided by the calculus and are syntactically finite, even when the corresponding semantic model is not.

Process calculi enjoy a rich repertoire of elegant meta-theories, proof techniques and analytical tools. SOC could benefit from this large body of knowledge and from the experience gained in the specification and analysis of concurrent, reactive and distributed systems during the last few decades. In fact, it has been already argued that type systems, modal and temporal logics, and observational equivalences provide adequate tools to address topics relevant to SOC (see e.g. [1, 2]). This 'proof technology' can eventually pave the way for the development of automatic property validation tools. Therefore, process calculi might play a central role in laying rigorous methodological foundations for specification and validation of SOC applications. Many process calculi for SOC have hence been proposed either by enriching well-established process calculi with specific constructs (e.g. the variants of π-calculus with transactions [3, 4, 5] and of CSP with compensation [6]) or by designing completely new formalisms (e.g. [7, 8, 9, 10, 11, 12, 13, 14]).

The work presented in this paper falls within the above line of research, since it introduces a process calculus, called C$\clock$WS (*Calculus for Orchestration of Web Services*), that aims at capturing the basic aspects of SOC systems and supporting their analysis. In designing C$\clock$WS, the main principles underlying the OASIS standard for orchestration of web services WS-BPEL [15] have been considered as first-class aspects. This permits a direct representation of the mechanisms underlying the SOC paradigm and is, then, an important step towards their investigation and comprehension. In fact, C$\clock$WS supports service instances with shared states, allows a process to play more than one partner role, permits programming stateful sessions by correlating different service interactions, and enables management of long-running transactions. However, C$\clock$WS intends to be a foundational model not specifically tight to web services' current technology. Thus, some WS-BPEL constructs, such as flow graphs and fault and compensation handlers, do not have a precise counterpart in C$\clock$WS, rather they are expressed in terms of more primitive operators. Of course, C$\clock$WS has also taken advantage of previous work on process calculi. Indeed, it combines in an original way constructs and features borrowed from well-known process calculi, e.g. non-binding input activities, asynchronous communication, polyadic synchronization, pattern matching, protection, delimited receiving and killing activities, while however resulting different from any of them.

We illustrate syntax, operational semantics and pragmatics of C$\clock$WS by means of a large case study from the automotive domain and a number of more specific examples drawn from it. We also present a C$\clock$WS's dialect that smoothly incorporates constraints and operations on them, thus permitting to model Quality of Service requirement specifications and Service Level Agreement achievements. This dialect is obtained by specialising a few syntactic objects (e.g., the set of expressions that can occur within terms of the calculus) and semantic mechanisms

of C⏲WS's definition. By means of our case study, we show that the formalism thus obtained can model all the phases of the life cycle of service-oriented applications, such as publication, discovery, negotiation, orchestration, deployment, reconfiguration and execution. This, on the one hand, provides evidence of the quality of the C⏲WS's design, on the other hand, may enable the application of a wide range of techniques for the analysis of services (see, e.g., [9, 16, 17, 18, 19, 20, 21]).

Summary of the rest of the paper. In Section 2, we provide an overview of SOC and an informal presentation of the case study that will be used throughout the paper for illustration purposes. In Section 3, to gradually introduce C⏲WS's technicalities and distinctive features, we present its syntax and operational semantics in four steps: for each of the four calculi we show many simple clarifying examples. In Section 4, we present the formal specification of the case study, informally described in Section 2, in the calculus corresponding to the untimed fragment of C⏲WS and provide a glimpse of the properties that can be verified over this specification. Then, in Section 5, we introduce the C⏲WS's dialect that permits modelling dynamic service publication, discovery and negotiation; we further elaborate the case study for illustrating both the additional aspects and the ones related to time. In Section 6, we review some strictly related work. Finally, in Section 7, we conclude with some final remarks and touch upon directions for future work.

This work is an extended and revisited version of our former developments introduced in [8, 22, 23]. The novel contribution is a comprehensive, uniform, more detailed and neater presentation of the process calculus C⏲WS and of how it can be effectively used to model the basic aspects of SOC systems. More specifically, Sections 3.1, 3.2 and 3.3 are a revised version of [8], although here we adopt a more detailed step-by-step presentation in order to gradually introduce the C⏲WS's features and discuss, for each of them, the underlying motivations. Moreover, the newer version uses many notations, conventions, definitions and examples that make the presentation of the operational semantics of the calculus simpler and clearer (in the preliminary version, e.g., the definitions of the predicates for checking the presence of receive conflicts and enabled kill activities resort to the notion of 'active context'). Section 3.4 is drawn from [22], while the dialect of C⏲WS presented in Section 5.1 comes from [23]; they have been properly integrated in this uniform presentation. All C⏲WS's features are illustrated by means of a large case study from the automotive domain and a number of more specific examples drawn from it. To sum up, this paper aims at providing the interested reader with a novel presentation of the calculus, where both design motivations and technical details about primitives and mechanisms are taken into account. From a more general perspective, the paper illustrates how SOC systems can be modelled by using an approach based on process calculi.

2. Background notions

In this introductory section, we set the scene of the whole paper by providing the background notions from Service-Oriented Computing that we aim at modelling and by informally presenting a case study used throughout the paper for describing how such notions are rendered in C⏲WS.

2.1. Service-Oriented Computing

Service-Oriented Computing (SOC) is emerging as an evolutionary paradigm for distributed and e-business computing that finds its origin in object-oriented and component-based software development. Early examples of technologies that are at least partly service-oriented are

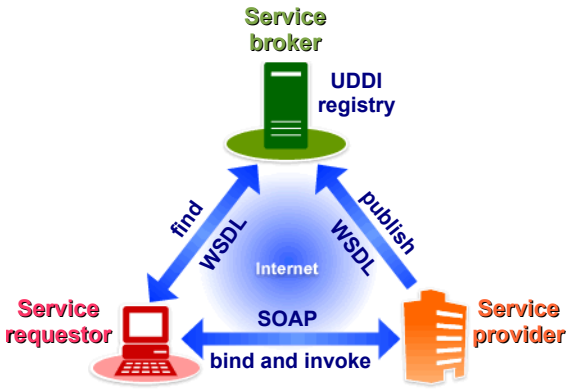

Figure 1: Service-Oriented Architecture

CORBA, DCOM, J2EE or .NET. A more recent successful instantiation of the SOC paradigm are *web services*. These are sets of operations (i.e. functionalities) that can be published, located and invoked through the Web via XML messages complying with given standard formats. To support the web service approach, several new languages and technologies have been designed and many international companies, like IBM, Microsoft and Oracle, have invested a lot of efforts.

There is a common way to view the web service architecture. It focuses on three major roles:

- *Service provider*: The software entity that implements a service specification and makes it available on the Internet. Providers publish machine-readable service descriptions on registries to enable automated discovery and invocation.
- *Service requestor* (or *client*): The software entity that invokes a service provider. A service requestor can be an end-user application or another service.
- *Service broker*: A specific kind of service provider that allows automated publication and discovery of services by relying on a registry.

Figure 1 shows the three service roles and how they interact with each other. This architecture, and the context of services use, imposes a series of constraints. Here are some key characteristics for effective use of services (see, e.g., [24]):

- *Coarse-grain*: Operations on services are frequently implemented to encompass more functionalities and operate on larger data sets, compared to those of fine-grained components as well as object-oriented interfaces.
- *Interface-based design*: Services implement separately defined interfaces. The set of interfaces implemented by a service is called *service description*. In addition to the functions that the service performs, service descriptions should also include non-functional properties (e.g. response time, availability, reliability, security, performance) that jointly represent the *quality of the service* (QoS). In this case, they are also called *service contracts*.
- *Discoverability*: Services need to be found at both design time and run time by service requestors. Moreover, since services are often developed and run by different organizations, a key issue of the discovery process is to define a flexible *negotiation* mechanism

that allows two or more parties to reach a joint agreement about cost and quality of a service, prior to service execution. The outcome of the negotiation phase is a *Service Level Agreement* (SLA), i.e. a contract among the involved parties that sets out both type and bounds on various performance metrics of the service to be provided.

- *Loosely coupling*: Services are connected to other services and clients using standard, dependency-reducing, decoupled message-based methods, as XML document exchanges.

- *Asynchrony*: In general, services use an asynchronous message passing approach, but this is not necessarily required.

Some of these criteria, such as interface-based design and discoverability, are also used in component-based development; however, it is the sum of these attributes that differentiates a service-based application from a component-based one. It is beneficial, for example, to make web services asynchronous to reduce the time a requestor spends waiting for responses. In fact, by making a service call asynchronous, with a separate return message, the requestor will be able to continue execution while the provider has a chance to respond. This is not to say that synchronous service behavior is wrong, just that experience has demonstrated that asynchronous service behavior is desirable, especially where communication costs are high or network latency is unpredictable, and provides the developer with a simpler scalability model [24].

To support the web service approach, many new languages, most of which based on XML, have been designed. The technologies that form the foundations of web services are SOAP, WSDL, and UDDI. Simple Object Access Protocol (SOAP, [25]) is responsible for encoding messages in a common XML format so that they can be understood at either end by all communicating services. Currently, SOAP is the principal XML-based standard for exchanging information between applications within a distributed environment. Web Service Description Language (WSDL, [26]) is responsible for describing the public interface of a specific web service. Through a WSDL description, that is an XML document, a client application can determine the location of the remote web service, the functions it implements, as well as how to access and use each function. After parsing a WSDL description, a client application can appropriately format a SOAP request and dispatch it to the location of the web service. In this setting, Universal Description, Discovery, and Integration (UDDI [27]) is responsible for centralizing services into a common registry and providing easy *publish* and *find* functionalities. The relationships between SOAP, WSDL, and UDDI are depicted in Figure 1.

To move beyond the basic framework *describe-publish-interact* and to better appreciate the real value of web services, mechanisms for service composition are required. Several specifications have been proposed in these areas, among which we would like to mention the composition language Web Services Business Process Execution Language (WS-BPEL, [15]), the OASIS standard for orchestration of web services. In the web services literature [28], the term *orchestration* is used to indicate composition of web services and, in particular, it describes how a collection of web services can interact with each other at the message level, including the business logic and the execution order of the interactions. These interactions may span applications and/or organizations, and result in a long-lived, transactional, multi-step process model.

A service orchestration combines services following a certain composition pattern to achieve a business goal or provide new service functions in general. For example, handling a purchase order is the summation of processes that calculate the final price for the order, select a shipper, and schedule the production and shipment for the order. It is worth emphasizing that service orchestrations may themselves become services, making composition a recursive operation. In

the example above, handling a purchase order may become a service that is instantiated to serve each received purchase order separately from other similar requests. This is necessary because a client might be carrying on many simultaneous purchase order interactions with the same service.

Service descriptions are thus used as templates for creating service instances that deliver application functionality to either end-user applications or other instances. The technology supporting tightly coupled communication frameworks typically establishes an active connection between interacting entities that persists for the duration of a given business activity (or even longer). Because the connection remains active, context is inherently present, and correlation between individual transmissions of data is intrinsically managed by the technology protocol itself. Instead, the loosely coupled nature of SOC implies that a same service should be identifiable by means of different logic names and the connection between communicating instances cannot be assumed to persist for the duration of a whole business activity. Therefore, there is no intrinsic mechanism for associating messages exchanged under a common context or as part of a common activity. Even the execution of a simple request-response message exchange pattern provides no built-in means of automatically associating the response message with the original request. It is up to each single message to provide a form of context thus enabling services to associate the message with others. This is achieved by embedding values in the message which, once located, can be used to correlate the message with others logically forming the same stateful interaction 'session' (also called 'conversation'). A key observation is that *message correlation* is an essential part of messaging within SOC as it enables the persistence of activities' context and state across multiple message exchanges while preserving service statelessness and autonomy, and the loosely coupled nature of service-oriented systems.

A further key feature of languages for service composition is the recovery mechanism for long-running business transactions. In SOC environments, the ordinary assumptions about primitive operations in traditional databases (Atomicity, Consistency, Isolation and Durability, ACID) are not applicable in general because local locks and isolation cannot be maintained for long periods (see [15], Section 12.3). Therefore, many languages for service composition rely on the concept of *compensation*, i.e. activities that attempt to reverse the effects of a previous activity that was carried out as part of a larger unit of work that is being abandoned.

All aspects of SOC we have just described are at the basis of the C◷WS's design. This because we believe that having them as first-class aspects would permit a more direct representation and a deeper comprehension of the mechanisms underlying the SOC paradigm. This is witnessed by the several examples described in the paper.

2.2. An automotive case study

We introduce here a significant case study [29] in the area of automotive systems defined within the EU project SENSORIA [30]. We consider a scenario where vehicles are equipped with a multitude of sensors and actuators that provide the driver with services that assist in conducting the vehicle more safely. Driver assistance systems become automatically operative when the vehicle context renders it necessary. Due to the advances in mobile technology, automotive software installed in the vehicles can contact relevant specific services to deal with driver's necessities.

Specifically, let us consider the case in which, while a driver is on the road with her/his car, the vehicle's *sensors monitor* reports a severe failure, which results in the car being no longer driveable. The car's *discovery* system then identifies garages, car rentals and towing truck services in the car's vicinity. At this point, the car's *reasoner* system chooses a set of adequate services taking into account personalised policies and preferences of the driver, e.g. balancing

cost and delay, and tries to order them. To be authorised to order services, the car's system has to deposit on behalf of the car owner a security payment, which will be given back if ordering the services fails. Other components of the in-vehicle service platform involved in this assistance activity are a *GPS* system, providing the car's current location, and an *orchestrator*, coordinating all the described services.

An UML-like activity diagram of the orchestration of services using UML4SOA, an UML Profile for service-oriented systems [31], is shown in Figure 2. The orchestrator is triggered by a signal from the sensors monitor (concerning, e.g., an engine failure) and consequently contacts the other components to locate and compose the various services to reach its goal. The process starts with a request from the orchestrator to the *bank* to charge the car owner's credit card with the security deposit payment. This is modelled by the UML action CardCharge for charging the credit card whose number is provided as an output parameter of the action call. In parallel to the interaction with the bank, the orchestrator requests the current location of the car from the car's internal GPS system. The current location is modelled as an input to the RequestLocation action and subsequently used by the FindServices interaction which retrieves a list of services. If no service can be found, an action to compensate the credit card charge will be launched. For the selection of services, the orchestrator synchronises with the reasoner service to obtain the most appropriate services.

Service ordering is modelled by the UML actions OrderGarage, OrderTowTruck and RentCar. When the orchestrator makes an appointment with the garage, the diagnostic data are automatically transferred to the garage, which could then be able, e.g., to identify the spare parts needed to perform the repair. Then, the orchestrator makes an appointment with the towing service, providing the GPS data of the stranded vehicle and of the garage, to tow the vehicle to the garage. Concurrently, the orchestrator makes an appointment with the rental service, by indicating the location (i.e. the GPS coordinates either of the stranded vehicle or of the garage) where the car will be handed over to the driver.

The workflow described in Figure 2 models the overall behaviour of the system. Besides interactions among services, it also includes activities using concepts developed for long running business transactions (in e.g. [32, 15]). These activities entail fault and compensation handling, kind of specific activities attempting to reverse the effects of previously committed activities, that are an important aspect of SOC applications. According to UML4SOA Profile, the installation of a compensation handler is modelled by an edge stereotyped ≪compensationEdge≫, and its activation by an activity stereotyped ≪compensate≫. Since each compensation handler is associated to a single UML activity, we omit drawing the enclosing 'scope' construct. Moreover, we use dashed boxes to represent compensation handlers. Specifically, in the considered scenario:

- the security deposit payment charged to the car owner's credit card must be revoked if either the discovery phase does not succeed or ordering the services fails, i.e. both garage/tow truck and car rental services reject the requests;
- if ordering a tow truck fails, the garage appointment has to be cancelled;
- if ordering a garage fails or a garage order cancellation is requested, the rental car delivery has to be redirected to the stranded car's actual location;
- instead, if ordering the car rental fails, it should not affect the tow truck and garage orders.

These requirements motivate the fact that ordering garage/tow truck and renting a car are modelled as activities running in parallel.

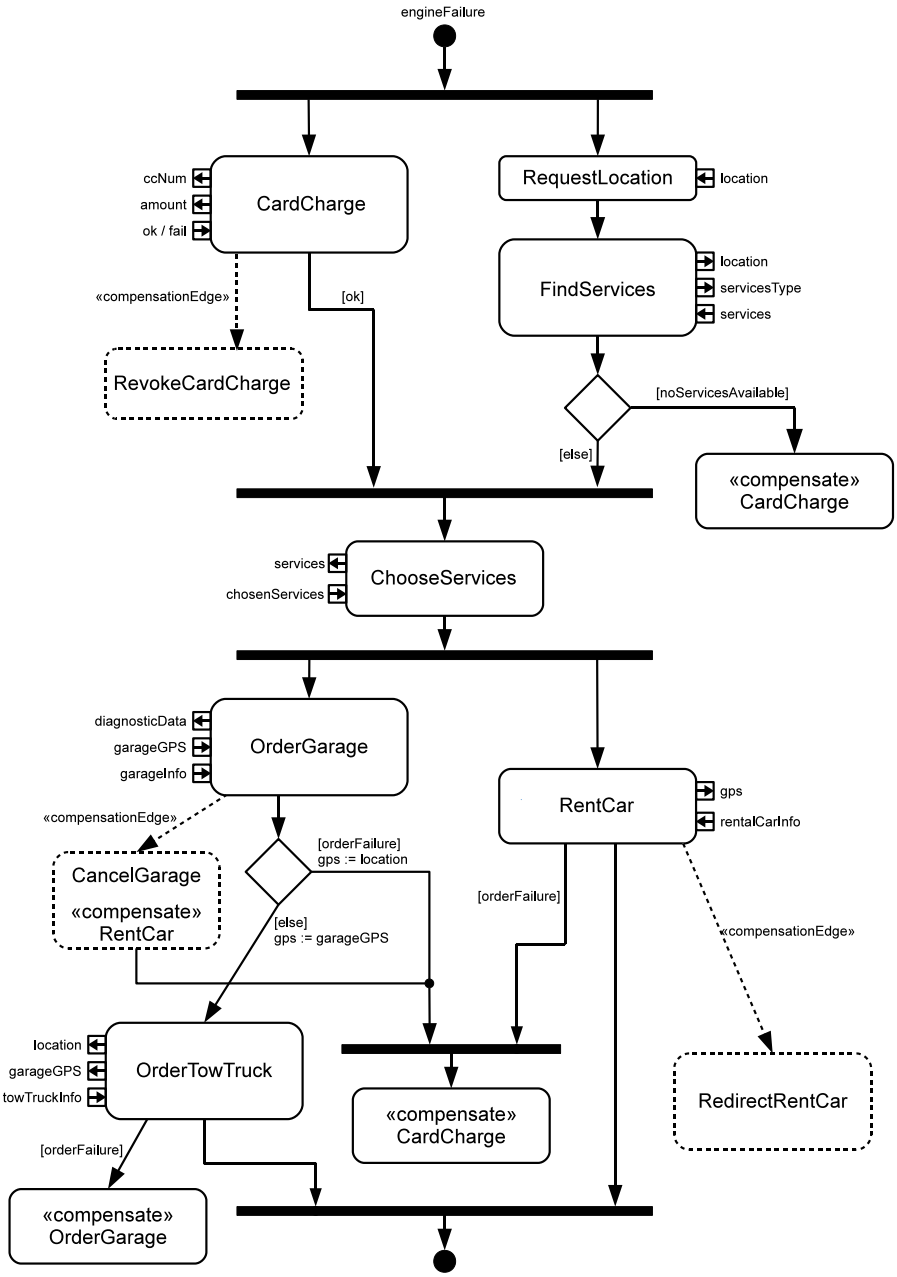

Figure 2: Orchestration in the automotive scenario

3. The language C◷WS

To gradually introduce the technicalities and distinctive features of C◷WS, we present its syntax and operational semantics in four steps. More specifically, in Section 3.1 we consider

μCOWS^m (μCOWS *minus priority*), the fragment of C$\oplus$WS without priority, primitives dealing with termination and timed activities. It retains all the other C$\oplus$WS's features, like e.g. global scope and pattern matching. In Section 3.2 we move on μCOWS (*micro* COWS), the calculus obtained by enriching μCOWS^m with priority. In Section 3.3 we consider COWS, which extends μCOWS with primitives dealing with termination. Finally, in Section 3.4 we study the full calculus, C$\oplus$WS, which incorporates timed orchestration constructs, thus permitting to express, e.g., choices among alternative activities constrained by expiration times. For each of the four calculi we show some accurate clarifying examples.

3.1. μCOWS^m : the priority-, protection-, kill- and time-free fragment of C$\oplus$WS

The fragment of C$\oplus$WS introduced in this section, namely μCOWS^m, dispenses with priority, primitives dealing with termination, and timed activities.

3.1.1. Syntax

The syntax of μCOWS^m is presented in Table 1. We use two countable disjoint sets: the set of *values* (ranged over by v, v', ...) and the set of 'write once' *variables* (ranged over by x, y, ...). The set of values is left unspecified; however, we assume that it includes the set of *names* (ranged over by n, m, p, o, ...) mainly used to represent partners and operations. We also use a set of *expressions* (ranged over by ϵ), whose exact syntax is deliberately omitted; we just assume that expressions contain, at least, values and variables.

Services are structured activities built from basic activities, i.e. the empty activity $\mathbf{0}$, the invoke activity $\_\bullet\_!\_$ and the receive activity $\_\bullet\_?\_$, by means of prefixing $\_.\_$, choice $\_+\_$, parallel composition $\_\mid\_$, delimitation $[\_]\_$ and replication $*\_$. The *empty* activity does nothing. *Invoke* and *receive* are the communication activities, which permit invoking an operation offered by a service and waiting for an invocation to arrive, respectively. *Prefixing* permits starting the execution of some service activities after the execution of a given basic activity is concluded. *Choice* permits selecting one between two alternative activities for execution, while *parallel composition* permits interleaving executions and enables communication between parallel services. *Delimitation* is used, according to its first argument, for two different purposes: to regulate the range of application of substitutions and to generate fresh names. Finally, *replication* permits implementing recursive behaviours and persistent services. We adopt the following conventions about the operators precedence: monadic operators bind more tightly than parallel composition, and prefixing more tightly than choice.

In the sequel, w ranges over values and variables and u ranges over names and variables. Notation $\bar{\cdot}$ stands for tuples, e.g. $\bar{x}$ means $\langle x_1, \ldots, x_n \rangle$ (with $n \geq 0$) where variables in the same tuple are pairwise distinct. We write $a, \bar{b}$ to denote the tuple obtained by concatenating the element a to the tuple $\bar{b}$. All notations shall extend to tuples component-wise. $\mathtt{n}$ ranges over communication endpoints that do not contain variables (e.g. $p \bullet o$), while $\mathtt{u}$ ranges over communication endpoints that may contain variables (e.g. $u \bullet u'$). Sometimes, we will use notation $\mathtt{n}$ and $\mathtt{u}$ for the tuples $\langle p, o \rangle$ and $\langle u, u' \rangle$, respectively, and rely on the context to resolve any ambiguity. When convenient, we shall regard a tuple (hence, also an endpoint) simply as a set, writing e.g. $x \in \bar{y}$ to mean that x is an element of $\bar{y}$. We will omit trailing occurrences of $\mathbf{0}$, writing e.g. $p \bullet o?\bar{w}$ instead of $p \bullet o?\bar{w}.\mathbf{0}$, and write $[\langle u_1, \ldots, u_n \rangle]\, s$ in place of $[u_1] \ldots [u_n]\, s$. We will write $I \triangleq s$ to assign a name I to the term s.

The only *binding* construct is delimitation: $[u]\, s$ binds u in the scope s. In fact, to enable concurrent threads within each service instance to share (part of) the state, receive activities in

Expressions: $\epsilon, \epsilon', \ldots$	*Variables/Names:* $u, u', \ldots$
Variables: $x, y, \ldots$	*Variables/Values:* $w, w', \ldots$
Values: $v, v', \ldots$	
Names: $n, m, \ldots$	*Endpoints:*
Partners: $p, p', \ldots$	without variables: $p \bullet o, \mathtt{n}, \ldots$
Operations: $o, o', \ldots$	may contain variables: $u \bullet u', \mathtt{u}, \ldots$

Services:

s ::=		
	$u \bullet u'!\bar{\epsilon}$	(invoke)
$\mid$	g	(receive-guarded choice)
$\mid$	$s \mid s$	(parallel composition)
$\mid$	$[u]\, s$	(delimitation)
$\mid$	$* s$	(replication)

Receive-guarded choice:

g ::=		
	$\mathbf{0}$	(nil)
$\mid$	$p \bullet o?\bar{w}.s$	(request processing)
$\mid$	$g + g$	(choice)

Table 1: μCOWS^m syntax

μCOWS^m bind neither names nor variables. This is different from most process calculi and somewhat similar to update [33] and fusion [34] calculi. In μCOWS^m, however, inter-service communication give rise to substitutions of variables with values (alike [33]), rather than to fusions of names (as in [34]). The range of application of the substitutions generated by a communication is regulated by the delimitation operator, that additionally permits to generate fresh names (as the restriction operator of π-calculus). Thus, the occurrence of a name/variable is *free* if it is not under the scope of a delimitation for it. Bound and free names are also called *private* and *public* names, respectively. We denote by fu(t) the set of free names/variables that occur free in t. Two terms are *α-equivalent* if one can be obtained from the other by consistently renaming bound names/variables. As usual, we identify terms up to α-equivalence.

Partner names and operation names can be combined to designate *endpoints*, written $p \bullet o$. In fact, alike channels in [35], an endpoint is not atomic but results from the composition of a partner name p and of an operation name o, which can also be interpreted as a specific implementation of o provided by p. This results in a very flexible naming mechanism that allows a service to be identified by means of different logic names (i.e. to play more than one partner role as in WS-BPEL). For example, the following service

$$p_{slow} \bullet o?\bar{w}.s_{slow} + p_{fast} \bullet o?\bar{w}.s_{fast}$$

accepts requests for the same operation o through different partners with distinct access modalities: process s_{slow} implements a slower service provided when the request is processed through the partner p_{slow}, while s_{fast} implements a faster service provided when the request arrives through p_{fast}. Additionally, the names composing an endpoint can be dealt with separately, as in a request-response interaction, where usually the service provider knows the name of the response operation, but not the partner name of the service it has to reply to. For example, the ping service $p \bullet o_{req}?\langle x \rangle.x \bullet o_{res}!\langle \text{“}\mathtt{I\ live}\text{”} \rangle$ will know at run-time the partner name for the reply activity. This mechanisms is also sufficiently expressive to support implementation of explicit locations: a located service can be represented by using a same partner for all its receiving endpoints. Partner and operation names can be exchanged in communication, thus enabling many different interaction patterns among service instances. However, dynamically received names can only be used for service invocation (as in *localised π-calculus* [36]). Indeed, endpoints of receive activities

$*\mathbf{0}$	$\equiv$	$\mathbf{0}$	$*s$	$\equiv$	$s \mid *s$			
$s \mid \mathbf{0}$	$\equiv$	s	$s_1 \mid s_2$	$\equiv$	$s_2 \mid s_1$	$(s_1 \mid s_2) \mid s_3$	$\equiv$	$s_1 \mid (s_2 \mid s_3)$
$g + \mathbf{0}$	$\equiv$	g	$g_1 + g_2$	$\equiv$	$g_2 + g_1$	$(g_1 + g_2) + g_3$	$\equiv$	$g_1 + (g_2 + g_3)$
$[u]\,\mathbf{0}$	$\equiv$	$\mathbf{0}$	$[u_1]\,[u_2]\,s$	$\equiv$	$[u_2]\,[u_1]\,s$	$s_1 \mid [u]\,s_2$	$\equiv$	$[u]\,(s_1 \mid s_2)$ if $u \notin \mathrm{fu}(s_1)$

Table 2: μCOWS^m structural congruence

are identified statically because their syntax only allows using names and not variables.

Remark 3.1 (Localised receive activities). As in localised π-calculus and differently from the standard π-calculus, C⊙WS disallows passing of 'input capability', i.e. the ability of services to receive a name and subsequently accept inputs along an endpoint containing such name. This choice is motivated, on the one hand, by the fact that the design of C⊙WS has been influenced by the current (web) service technologies where endpoints of receive activities are statically determined[1] (recall that service endpoints are not π-calculus channels) and, on the other hand, by the will to support an easier implementation of the calculus. However, the former is the major motivation. In fact, implementation problems due to input capability could be solved by relying on the theory of linear forwarders [37] as in `PiDuce` [38].

To model asynchronous communication, invoke activities cannot be used as prefixes and choice can only be guarded by receive activities (as in *asynchronous π-calculus* [39]). Indeed, in service-oriented systems, communication paradigms are usually asynchronous (as we pointed out in Section 2.1), in the sense that there may be an arbitrary delay between the sending and the receiving of a message, the ordering in which messages are received may differ from that in which they were sent, and a sender cannot determine if and when a sent message will be received.

3.1.2. Operational semantics

The operational semantics of μCOWS^m is defined only for *closed* services, i.e. services without free variables. By following an approach commonly used for process calculi, the semantics is formally given in terms of a structural congruence and of a labelled transition relation. The *structural congruence*, written $\equiv$, identifies syntactically different services that intuitively represent the same service. It is defined as the least congruence relation induced by the equational laws shown in Table 2. All the laws are straightforward. In particular, commutativity of consecutive delimitations implies that the order among the u_i in $[\langle u_1, \ldots, u_n \rangle]\,s$ is irrelevant, thus in the sequel we may use the simpler notation $[u_1, \ldots, u_n]\,s$. The last law permits to extend the scope of names (as in the π-calculus) and variables, thus enabling possible communication (see the examples "Communication" and "Communication of private names" in Section 3.1.3).

The definition of the labelled transition relation is parameterized by two auxiliary functions; we present here their basic definitions and show in Section 5.1 how they can be specialised to obtain a dialect of the language. Firstly, we use the function $[\![\_]\!]$ for evaluating *closed* expressions (i.e. expressions without variables): it takes a closed expression and returns a value. It is not explicitly defined since the exact syntax of expressions is deliberately not specified. Secondly, we use the partial function $\mathcal{M}(\_, \_)$ for performing *pattern-matching* on semi-structured data and,

[1] Indeed, if a WS-BPEL process receives an operation name, it cannot make this operation available to other services and then receive messages through it. In fact, this would require the process to be able to modify at runtime its WSDL interface to add the definition of the new operation, but WS-BPEL provides no construct allowing this dynamic change.

$$\mathcal{M}(x, v) = \{x \mapsto v\} \qquad \mathcal{M}(v, v) = \emptyset \qquad \mathcal{M}(\langle\rangle, \langle\rangle) = \emptyset \qquad \frac{\mathcal{M}(w_1, v_1) = \sigma_1 \quad \mathcal{M}(\bar{w}_2, \bar{v}_2) = \sigma_2}{\mathcal{M}((w_1, \bar{w}_2), (v_1, \bar{v}_2)) = \sigma_1 \uplus \sigma_2}$$

Table 3: Matching rules

$$\frac{[\![\bar{\epsilon}]\!] = \bar{v}}{\mathtt{n}!\bar{\epsilon} \xrightarrow{\mathtt{n} \lhd \bar{v}} \mathbf{0}}\ (inv) \qquad \mathtt{n}?\bar{w}.s \xrightarrow{\mathtt{n} \rhd \bar{w}} s\ (rec) \qquad \frac{g \xrightarrow{\alpha} s}{g + g' \xrightarrow{\alpha} s}\ (choice)$$

$$\frac{s_1 \xrightarrow{\mathtt{n} \rhd \bar{w}} s_1' \qquad s_2 \xrightarrow{\mathtt{n} \lhd \bar{v}} s_2' \qquad \mathcal{M}(\bar{w}, \bar{v}) = \sigma}{s_1 \mid s_2 \xrightarrow{\sigma} s_1' \mid s_2'}\ (com) \qquad \frac{s_1 \xrightarrow{\alpha} s_1'}{s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2}\ (par)$$

$$\frac{s \xrightarrow{\sigma \uplus \{x \mapsto v\}} s'}{[x]\, s \xrightarrow{\sigma} s' \cdot \{x \mapsto v\}}\ (del_{com}) \qquad \frac{s \xrightarrow{\alpha} s' \quad u \notin \mathrm{u}(\alpha)}{[u]\, s \xrightarrow{\alpha} [u]\, s'}\ (del) \qquad \frac{s \equiv \xrightarrow{\alpha} \equiv s'}{s \xrightarrow{\alpha} s'}\ (str)$$

Table 4: μCOWS^n operational semantics

thus, determining if a receive and an invoke over the same endpoint can synchronise. The rules defining $\mathcal{M}(\_, \_)$ are shown in Table 3. They state that two tuples match if they have the same number of fields and corresponding fields have matching values/variables. Variables match any value, and two values match only if they are identical. When tuples $\bar{w}$ and $\bar{v}$ do match, $\mathcal{M}(\bar{w}, \bar{v})$ returns a substitution for the variables in $\bar{w}$; otherwise, it is undefined. *Substitutions* (ranged over by σ) are functions mapping variables to values and are written as collections of pairs of the form $x \mapsto v$. Application of substitution σ to s, written $s \cdot \sigma$, has the effect of replacing every free occurrence of x in s with v, for each $x \mapsto v \in \sigma$, by possibly using α-conversion for avoiding v to be captured by name delimitations within s. We use $\emptyset$ to denote the empty substitution, $|\sigma|$ to denote the number of pairs in σ, and $\sigma_1 \uplus \sigma_2$ to denote the union of σ_1 and σ_2 when they have disjoint domains.

The *labelled transition relation* $\xrightarrow{\alpha}$ is the least relation over services induced by the rules in Table 4, where label α is generated by the following grammar:

$$\alpha \quad ::= \quad \mathtt{n} \lhd \bar{v} \quad \mid \quad \mathtt{n} \rhd \bar{w} \quad \mid \quad \sigma$$

The meaning of labels is as follows: $\mathtt{n} \lhd \bar{v}$ and $\mathtt{n} \rhd \bar{w}$ denote execution of invoke and receive activities over the endpoint $\mathtt{n}$ with arguments $\bar{v}$ and $\bar{w}$, respectively; σ denotes execution of a communication with generated substitution σ to be still applied. The empty substitution $\emptyset$ denotes a *computational step* corresponding to taking place of communication without pending substitutions. In the sequel, we will use $\mathrm{u}(\alpha)$ to denote the set of names and variables occurring in α, where $\mathrm{u}(\{x \mapsto v\}) = \{x\} \cup \mathrm{fu}(v)$ and $\mathrm{u}(\sigma_1 \uplus \sigma_2) = \mathrm{u}(\sigma_1) \cup \mathrm{u}(\sigma_2)$.

Let us now comment on the operational rules. A service invocation can proceed only if the expressions in the argument can be evaluated (rule *(inv)*). This means, for example, that if it contains a variable x (in its endpoint or argument) it is stuck until x is not replaced by a value because of execution of a receive assigning a value to x. A receive activity offers an invocable operation along a given partner name (rule *(rec)*), and execution of a receive permits to take a decision between alternative behaviours (rule *(choice)*). Communication can take place when two parallel services perform matching receive and invoke activities (rule *(com)*). Communication generates

a substitution that is recorded in the transition label (for subsequent application), rather than a silent transition as in most process calculi. Execution of parallel services is interleaved (rule *(par)*). When the delimitation of a variable x argument of a receive involved in a communication is encountered, i.e. the whole scope of the variable is determined, the delimitation is removed and the substitution for x is applied to the term (rule *(del$_{com}$)*). Variable x disappears from the term and cannot be reassigned a value (for this reason we say that μCOWS^m's variables are 'write once'). Notably, since in closed services all variables are delimited, the taking place of a communication within such kind of services always corresponds to a computational step and leads to services that are closed too. $[u]\, s$ behaves like s (rule *(del)*), except when the transition label α contains u. Rule *(str)* is standard and states that structurally congruent services have the same transitions.

3.1.3. Examples

We report here a few examples aimed at clarifying the peculiarities of μCOWS^m. For the sake of presentation, the examples focus on a part of the automotive case study described in Section 2.2 that involves the interactions with a service of the car owner's bank. This service allows its clients to charge a credit card for a specified amount by sending charge requests via the endpoint $p_{bank} \cdot o_{charge}$. A client, besides his credit card number, the amount to be charged and the timestamp (i.e. date and time) of the transaction, is required to provide the partner name that he will use to receive a response.

Communication. Communication can exploit scope extension (last law of Table 2) to allow receive and invoke activities to interact. In fact, they can synchronise only if both are in the scope of the delimitations that bind the variables argument of the receive. Thus, we must possibly extend the scopes of some variables, as in the following example, where a client with partner name p_c invokes the bank service for charging his credit card 1234 with 100 euros at time t:

$$
\begin{array}{l}
p_{bank} \cdot o_{charge}!\langle p_c, 1234, 100, t\rangle \\
\quad \mid\ [x_{cust}, x_{cc}, x_{amount}, x_{ts}]\,(p_{bank} \cdot o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \ \mid\ s') \qquad \equiv \\
\\
[x_{cust}, x_{cc}, x_{amount}, x_{ts}]\,(p_{bank} \cdot o_{charge}!\langle p_c, 1234, 100, t\rangle \\
\qquad\qquad \mid\ p_{bank} \cdot o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \ \mid\ s') \quad \xrightarrow{\emptyset} \\
\\
(s \ \mid\ s') \cdot \{x_{cust} \mapsto p_c\,, x_{cc} \mapsto 1234\,, x_{amount} \mapsto 100\,, x_{ts} \mapsto t\}
\end{array}
$$

Notice that, as shown by the inference of the above transition reported in Table 5, the substitution $\{x_{cust} \mapsto p_c\,, x_{cc} \mapsto 1234\,, x_{amount} \mapsto 100\,, x_{ts} \mapsto t\}$ is applied to all terms delimited by $[x_{cust}, x_{cc}, x_{amount}, x_{ts}]$, not only to the continuation s of the service performing the receive. This is different from most process calculi and accounts for the global scope of variables. This very feature permits, e.g., to easily model the *delayed input* of fusion calculus [34], which is instead difficult to express in π-calculus.

Communication of private names. Communication of private names is standard and exploits scope extension as in π-calculus. To enable communication of private names, besides their scopes, we must possibly extend the scopes of some variables. Consider to modify the previous example by restricting the scope of the partner name p_c to the invoke activity, with p_c fresh in s and s'. Now, the communication can take place as follow:

$$p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle \xrightarrow{p_{bank} \bullet o_{charge} \lhd \langle p_c, 1234, 100, t\rangle} \mathbf{0} \quad (inv)$$

$$p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \xrightarrow{p_{bank} \bullet o_{charge} \rhd \langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle} s \quad (rec)$$

$$\frac{\mathcal{M}(\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle, \langle p_c, 1234, 100, t\rangle) = \{x_{cust} \mapsto p_c\ , x_{cc} \mapsto 1234\ , x_{amount} \mapsto 100\ , x_{ts} \mapsto t\}}{p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle \mid p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \xrightarrow{\{x_{cust} \mapsto p_c\ , x_{cc} \mapsto 1234\ , x_{amount} \mapsto 100\ , x_{ts} \mapsto t\}} s} \ (com)$$

$$\frac{p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle \mid p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \xrightarrow{\{x_{cust} \mapsto p_c\ , x_{cc} \mapsto 1234\ , x_{amount} \mapsto 100\ , x_{ts} \mapsto t\}} s}{p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle \mid p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \mid s' \xrightarrow{\{x_{cust} \mapsto p_c\ , x_{cc} \mapsto 1234\ , x_{amount} \mapsto 100\ , x_{ts} \mapsto t\}} s \mid s'} \ (par)$$

$$\frac{p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle \mid p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \mid s' \xrightarrow{\{x_{cust} \mapsto p_c\ , x_{cc} \mapsto 1234\ , x_{amount} \mapsto 100\ , x_{ts} \mapsto t\}} s \mid s'}{[x_{ts}]\,(p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle \mid p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \mid s') \xrightarrow{\{x_{cust} \mapsto p_c\ , x_{cc} \mapsto 1234\ , x_{amount} \mapsto 100\}} (s \mid s') \cdot \{x_{ts} \mapsto t\}} \ (del_{com})$$

$$\frac{[x_{ts}]\,(p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle \mid p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \mid s') \xrightarrow{\{x_{cust} \mapsto p_c\ , x_{cc} \mapsto 1234\ , x_{amount} \mapsto 100\}} (s \mid s') \cdot \{x_{ts} \mapsto t\}}{[x_{amount}, x_{ts}]\,(p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle \mid p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \mid s') \xrightarrow{\{x_{cust} \mapsto p_c\ , x_{cc} \mapsto 1234\}} (s \mid s') \cdot \{x_{amount} \mapsto 100\ , x_{ts} \mapsto t\}} \ (del_{com})$$

$$\frac{[x_{amount}, x_{ts}]\,(p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle \mid p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \mid s') \xrightarrow{\{x_{cust} \mapsto p_c\ , x_{cc} \mapsto 1234\}} (s \mid s') \cdot \{x_{amount} \mapsto 100\ , x_{ts} \mapsto t\}}{[x_{cc}, x_{amount}, x_{ts}]\,(p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle \mid p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \mid s') \xrightarrow{\{x_{cust} \mapsto p_c\}} (s \mid s') \cdot \{x_{cc} \mapsto 1234\ , x_{amount} \mapsto 100\ , x_{ts} \mapsto t\}} \ (del_{com})$$

$$\frac{[x_{cc}, x_{amount}, x_{ts}]\,(p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle \mid p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \mid s') \xrightarrow{\{x_{cust} \mapsto p_c\}} (s \mid s') \cdot \{x_{cc} \mapsto 1234\ , x_{amount} \mapsto 100\ , x_{ts} \mapsto t\}}{[x_{cust}, x_{cc}, x_{amount}, x_{ts}]\,(p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle \mid p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \mid s') \xrightarrow{\emptyset} (s \mid s') \cdot \{x_{cust} \mapsto p_c\ , x_{cc} \mapsto 1234\ , x_{amount} \mapsto 100\ , x_{ts} \mapsto t\}} \ (del_{com})$$

Table 5: Inference of a computational step

$$[p_c]\,(p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle) \mid [x_{cust}, x_{cc}, x_{amount}, x_{ts}]\,(p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \mid s') \quad \equiv$$

$$[p_c]\,(p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle \mid [x_{cust}, x_{cc}, x_{amount}, x_{ts}]\,(p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \mid s')) \quad \equiv$$

$$[p_c, x_{cust}, x_{cc}, x_{amount}, x_{ts}]\,(p_{bank} \bullet o_{charge}!\langle p_c, 1234, 100, t\rangle \mid p_{bank} \bullet o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ s \mid s') \xrightarrow{\emptyset}$$

$$[p_c]\,(s \mid s') \cdot \{x_{cust} \mapsto p_c\ , x_{cc} \mapsto 1234\ , x_{amount} \mapsto 100\ , x_{ts} \mapsto t\}$$

Persistent services. The replication operator, which spawns in parallel as many copies of its argument term as necessary (law $* s \equiv s \mid * s$ of Table 2), permits specifying *persistent* services, i.e. services capable of creating multiple instances to serve several requests simultaneously[2].

[2]It is worth noticing that this is the standard behaviour of web services and, in particular, this is always the case for services resulting from WS-BPEL orchestrations [15, Section 5.5].

Thus, the bank service previously introduced can be made persistent by simply applying the replication operator to the μCOWS^m term as shown in the following example, where the (persistent) service definition runs in parallel with two clients:

$$
\begin{array}{l}
(p_{bank} \cdot o_{charge}!\langle p_{cA}, 1234, 100, t_A\rangle \mid [x]\, p_{cA} \cdot o_{resp}?\langle x, t_A\rangle.s_A) \\
\mid\ (p_{bank} \cdot o_{charge}!\langle p_{cB}, 5678, 200, t_B\rangle \mid [y]\, p_{cB} \cdot o_{resp}?\langle y, t_B\rangle.s_B) \\
\mid\ *\,[x_{cust}, x_{cc}, x_{amount}, x_{ts}]\, p_{bank} \cdot o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ x_{cust} \cdot o_{resp}!\langle check(x_{cc}, x_{amount}), x_{ts}\rangle
\end{array}
$$

For each client request, the bank service creates an instance that replies to the corresponding client with a message, containing the result of the transaction and the timestamp, along either the endpoint $p_{cA} \cdot o_{resp}$ or $p_{cB} \cdot o_{resp}$. Here, for the sake of simplicity, the acceptance or rejection of a charge request is the result of the evaluation of a function $check(\_, \_)$, which is left unspecified, that takes as arguments a credit card number and an amount. Symmetrically, the client A (resp. B) invokes the bank service and, once a response along $p_{cA} \cdot o_{resp}$ (resp. $p_{cB} \cdot o_{resp}$) is received, proceeds as s_A (resp. s_B).

After a computational step, due to the interaction between the service definition and the client A, a new instance (highlighted by a gray background) runs in parallel with the other terms:

$$
\begin{array}{l}
[x]\, p_{cA} \cdot o_{resp}?\langle x, t_A\rangle.s_A \\
\mid\ (p_{bank} \cdot o_{charge}!\langle p_{cB}, 5678, 200, t_B\rangle \mid [y]\, p_{cB} \cdot o_{resp}?\langle y, t_B\rangle.s_B) \\
\mid\ *\,[x_{cust}, x_{cc}, x_{amount}, x_{ts}]\, p_{bank} \cdot o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ x_{cust} \cdot o_{resp}!\langle check(x_{cc}, x_{amount}), x_{ts}\rangle \\
\quad \mid\ p_{cA} \cdot o_{resp}!\langle check(1234, 100), t_A\rangle
\end{array}
$$

If, similarly, the client B invokes the service, a second instance (highlighted by a dark gray background) is created:

$$
\begin{array}{l}
[x]\, p_{cA} \cdot o_{resp}?\langle x, t_A\rangle.s_A \\
\mid\ [y]\, p_{cB} \cdot o_{resp}?\langle y, t_B\rangle.s_B \\
\mid\ *\,[x_{cust}, x_{cc}, x_{amount}, x_{ts}]\, p_{bank} \cdot o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ x_{cust} \cdot o_{resp}!\langle check(x_{cc}, x_{amount}), x_{ts}\rangle \\
\quad \mid\ p_{cA} \cdot o_{resp}!\langle check(1234, 100), t_A\rangle \\
\quad \mid\ p_{cB} \cdot o_{resp}!\langle check(5678, 200), t_B\rangle
\end{array}
$$

Now, the two instances can reply to the corresponding clients by invoking the operation o_{resp} through the two different client partner names p_{cA} and p_{cB}. Thus, assuming that the *check* function returns *ok* for the A's request and *fail* for the B's one, after two computational steps the system becomes

$$
\begin{array}{l}
s_A \cdot \{x \mapsto ok\} \\
\mid\ s_B \cdot \{y \mapsto fail\} \\
\mid\ *\,[x_{cust}, x_{cc}, x_{amount}, x_{ts}]\, p_{bank} \cdot o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle.\ x_{cust} \cdot o_{resp}!\langle check(x_{cc}, x_{amount}), x_{ts}\rangle
\end{array}
$$

Services' execution modalities. In μCOWS^m, a service can be modelled by a term of the form $*\,[\bar{u}\,]\,s$, where tuple $\bar{u}$ contains all the free variables of s. The use of replication enables providing as many concurrent instances as needed, while that of delimitation permits modelling the state (by restricting the scope of variables). This means that the previous term corresponds to a service whose instances do *not share a state*. For instance, consider the following service definition:

$$*\,[x_1, \ldots, x_n]\, p \cdot o?\langle x_1\rangle.s$$

If we put it in parallel with the invocation $p \cdot o!\langle v_1 \rangle$, the resulting system can evolve as follows:

$$\begin{array}{l} *\,[x_1, \ldots, x_n]\, p \cdot o?\langle x_1 \rangle.s \;\mid\; p \cdot o!\langle v_1 \rangle \xrightarrow{\emptyset} \\ *\,[x_1, \ldots, x_n]\, p \cdot o?\langle x_1 \rangle.s \;\mid\; [x_2, \ldots, x_n]\, s \cdot \{x_1 \mapsto v_1\} \end{array}$$

Each time an invocation is processed, a new service instance with private variables $x_2, \ldots, x_n$ is activated. For example, if we have two concurrent invocations, we get

$$\begin{array}{l} *\,[x_1, \ldots, x_n]\, p \cdot o?\langle x_1 \rangle.s \;\mid\; p \cdot o!\langle v_1 \rangle \;\mid\; p \cdot o!\langle v_2 \rangle \xrightarrow{\emptyset} \xrightarrow{\emptyset} \\ *\,[x_1, \ldots, x_n]\, p \cdot o?\langle x_1 \rangle.s \;\mid\; [x_2, \ldots, x_n]\, s \cdot \{x_1 \mapsto v_1\} \;\mid\; [x_2, \ldots, x_n]\, s \cdot \{x_1 \mapsto v_2\} \end{array}$$

The resulting system is composed of the service definition and of two different instances, each with its own state.

To allow instances of a same service to *share (part of) the state*, we move the delimitations of the variables to be shared outside the scope of replication. Thus, if $x_1, \ldots, x_k$ are shared and $x_{k+1}, \ldots, x_n$ are not, the previous example can be modified as follows:

$$[x_1, \ldots, x_k] \;*\; [x_{k+1}, \ldots, x_n]\, p \cdot o?\langle x_1 \rangle.s$$

After a parallel request $p \cdot o!\langle v_1 \rangle$ has been processed, we have:

$$[x_2, \ldots, x_k]\,(\,*\,[x_{k+1}, \ldots, x_n]\, p \cdot o?\langle v_1 \rangle.s \cdot \{x_1 \mapsto v_1\} \;\mid\; [x_{k+1}, \ldots, x_n]\, s \cdot \{x_1 \mapsto v_1\}\,)$$

In this case, since x_1 is shared both by the service definition and by its instances, new instances can be created only if the service definition receives requests along $p \cdot o$ with the same value (i.e. v_1) as the first invocation. In general, however, instantiation variables, such as x_1, are not shared, in order to allow service invocations with different arguments to trigger instance creation. To model this behaviour, we can simply leave instantiation variables within the scope of replication. Consider for example the term:

$$[x_2] \;*\; [x_1, x_3]\, p \cdot o?\langle x_1 \rangle.s$$

If requests $p \cdot o!\langle v_1 \rangle$ and $p \cdot o!\langle v_2 \rangle$ are put in parallel, the resulting system can evolve as follows:

$$\begin{array}{l} [x_2] \;*\; [x_1, x_3]\, p \cdot o?\langle x_1 \rangle.s \mid p \cdot o!\langle v_1 \rangle \mid p \cdot o!\langle v_2 \rangle \xrightarrow{\emptyset} \xrightarrow{\emptyset} \\ [x_2]\,(\,*\,[x_1, x_3]\, p \cdot o?\langle x_1 \rangle.s \mid [x_3]\, s \cdot \{x_1 \mapsto v_1\} \mid [x_3]\, s \cdot \{x_1 \mapsto v_2\}\,) \end{array}$$

After two computational steps, two instances, each with a local state (i.e. the variable x_3) and sharing variable x_2, are activated.

Message correlation. The loosely coupled nature of SOC implies that the connection between communicating instances should not be assumed to persist for the duration of a whole business activity. Therefore, it is up to each single message to provide a form of context that enables services to associate the message with others. This is achieved by embedding values, called *correlation data*, in the content of the message itself. Pattern-matching is the mechanism for locating such data important to identify service instances for the delivering of messages.

To explain how message correlation is realized in μCOWS^m, let us consider a variant of the bank service composed of two persistent subservices: *BankInterface*, that is publicly invocable by customers, and *CreditRating*, that instead is an 'internal' service that can only interact with

BankInterface (indeed, all the operations used by *CreditRating*, i.e. o_{check}, $o_{checkOk}$ and $o_{checkFail}$, are restricted and this prevents them to be invoked from the outside). Specifically, *Bank* is the μCOWS^m term

$$[o_{check}, o_{checkOk}, o_{checkFail}]\,(\,*\,\mathit{BankInterface} \mid *\,\mathit{CreditRating}\,)$$

where *BankInterface* and *CreditRating* are defined as follows:

$$\begin{aligned}
\mathit{BankInterface} \triangleq\ & [x_{cust}, x_{cc}, x_{amount}, x_{ts}] \\
& p_{bank} \cdot o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle. \\
& (\ p_{bank} \cdot o_{check}!\langle x_{ts}, x_{cc}, x_{amount}\rangle \\
& \quad \mid [x_{info}]\,(p_{bank} \cdot o_{checkFail}?\langle x_{ts}, x_{cc}, x_{info}\rangle.\ x_{cust} \cdot o_{resp}!\langle \mathit{fail}, x_{ts}, x_{info}\rangle \\
& \qquad\qquad + \ p_{bank} \cdot o_{checkOk}?\langle x_{ts}, x_{cc}, x_{info}\rangle.\ x_{cust} \cdot o_{resp}!\langle \mathit{ok}, x_{ts}, x_{info}\rangle\ )
\end{aligned}$$

$$\begin{aligned}
\mathit{CreditRating} \triangleq\ & [x_{ts}, x_{cc}, x_a] \\
& p_{bank} \cdot o_{check}?\langle x_{ts}, x_{cc}, x_a\rangle. \\
& [p, o]\,(\ p \cdot o!\langle\rangle \mid p \cdot o?\langle\rangle.\ p_{bank} \cdot o_{checkOk}!\langle x_{ts}, x_{cc}, \mathit{ratingInfo}(x_{cc}, x_a)\rangle \\
& \qquad\qquad + p \cdot o?\langle\rangle.\ p_{bank} \cdot o_{checkFail}!\langle x_{ts}, x_{cc}, \mathit{ratingInfo}(x_{cc}, x_a)\rangle\ )
\end{aligned}$$

Whenever prompted by a client request, *BankInterface* creates an instance to serve that specific request and is immediately ready to concurrently serve other requests. Each instance forwards the request to *CreditRating*, by invoking the internal operation o_{check} through the invoke activity $p_{bank} \cdot o_{check}!\langle x_{ts}, x_{cc}, x_{amount}\rangle$, then waits for a reply on one of the other two internal operations $o_{checkFail}$ and $o_{checkOk}$, by exploiting the receive-guarded choice operator, and finally sends the reply back to the client by means of a final invoke activity using the partner name of the client stored in the variable x_{cust}. Service *CreditRating* takes care of checking clients' requests and decides if they can be authorised or not. For the sake of simplicity, the choice between approving or not a request is left here completely non-deterministic, and rating information are calculated by an (unspecified) function $\mathit{ratingInfo}(\_, \_)$.

Consider now the above 'compound' bank service running in parallel with two clients:

$$\begin{aligned}
& (p_{bank} \cdot o_{charge}!\langle p_{cA}, 1234, 100, t_A\rangle \mid [x, x_i]\ p_{cA} \cdot o_{resp}?\langle x, t_A, x_i\rangle.s_A) \\
& \mid\ (p_{bank} \cdot o_{charge}!\langle p_{cB}, 5678, 200, t_B\rangle \mid [y, y_i]\ p_{cB} \cdot o_{resp}?\langle y, t_B, y_i\rangle.s_B) \\
& \mid\ [o_{check}, o_{checkOk}, o_{checkFail}] \\
& \quad (\,*\,\mathit{BankInterface} \mid *\,\mathit{CreditRating}\,)
\end{aligned}$$

After a certain number of computational steps have taken place, two instances of *BankInterface* (highlighted by a gray background) and two of *CreditRating* (highlighted by a dark gray background) would have been created and the system would be:

$$\begin{aligned}
& [x, x_i]\ p_{cA} \cdot o_{resp}?\langle x, t_A, x_i\rangle.s_A \\
& \mid\ [y, y_i]\ p_{cB} \cdot o_{resp}?\langle y, t_B, y_i\rangle.s_B \\
& \mid\ [o_{check}, o_{checkOk}, o_{checkFail}] \\
& \quad (\,*\,\mathit{BankInterface} \mid *\,\mathit{CreditRating} \\
& \qquad \mid\ [x_{info}]\,(p_{bank} \cdot o_{checkFail}?\langle t_A, 1234, x_{info}\rangle.\ p_{cA} \cdot o_{resp}!\langle \mathit{fail}, t_A, x_{info}\rangle \\
& \qquad\qquad + p_{bank} \cdot o_{checkOk}?\langle t_A, 1234, x_{info}\rangle.\ p_{cA} \cdot o_{resp}!\langle \mathit{ok}, t_A, x_{info}\rangle) \\
& \qquad \mid\ p_{bank} \cdot o_{checkOk}!\langle t_A, 1234, \mathit{ratingInfo}(1234, 100)\rangle
\end{aligned}$$

$$
\begin{array}{l}
\mid\ [x_{info}]\,(p_{bank} \bullet o_{checkFail}?\langle t_B, 5678, x_{info}\rangle.\ p_{cB} \bullet o_{resp}!\langle fail, t_B, x_{info}\rangle \\
\qquad\qquad + p_{bank} \bullet o_{checkOk}?\langle t_B, 5678, x_{info}\rangle.\ p_{cB} \bullet o_{resp}!\langle ok, t_B, x_{info}\rangle) \\
\mid\ p_{bank} \bullet o_{checkFail}!\langle t_B, 5678, ratingInfo(5678, 200)\rangle\ )
\end{array}
$$

Notably, the *BankInterface*'s instance created to serve the client A (resp. B) is identified by the client data t_A and 1234 (resp. t_B and 5678) that are exploited as correlation values. In fact, we assume that, from the point of view of the bank service, each client request is uniquely identified by the timestamp of the transaction and the client's credit card. Instead, if we consider the point of view of the client and suppose that he has only one credit card and has sent more charge requests for it, thus the timestamp would be enough to correlate a bank service response to a client instance. Recall that it is the responsibility of the service programmer to individuate the proper correlation data in a given conversation.

Now, if the invocation along the endpoint $p_{bank} \bullet o_{checkOk}$ is performed (we assume $ratingInfo(1234, 100) = info$), since the sent message contains the correlation data t_A and 1234, the interaction takes place with the instance created to serve the client A (indeed, $\mathcal{M}(\langle t_B, 5678, x_{info}\rangle, \langle t_A, 1234, info\rangle)$ does not hold):

$$
\begin{array}{l}
[x, x_i]\ p_{cA} \bullet o_{resp}?\langle x, t_A, x_i\rangle.s_A \\
\mid\ [y, y_i]\ p_{cB} \bullet o_{resp}?\langle y, t_B, y_i\rangle.s_B \\
\mid\ [o_{check}, o_{checkOk}, o_{checkFail}] \\
\quad (*\,BankInterface \mid *\,CreditRating \\
\qquad \mid\ p_{cA} \bullet o_{resp}!\langle ok, t_A, info\rangle \\
\qquad \mid\ [x_{info}]\,(p_{bank} \bullet o_{checkFail}?\langle t_B, 5678, x_{info}\rangle.\ p_{cB} \bullet o_{resp}!\langle fail, t_B, x_{info}\rangle \\
\qquad\qquad + p_{bank} \bullet o_{checkOk}?\langle t_B, 5678, x_{info}\rangle.\ p_{cB} \bullet o_{resp}!\langle ok, t_B, x_{info}\rangle) \\
\qquad \mid\ p_{bank} \bullet o_{checkFail}!\langle t_B, 5678, ratingInfo(5678, 200)\rangle\ )
\end{array}
$$

Therefore, although two *BankInterface*'s instances waiting for a message along the endpoint $p_{bank} \bullet o_{checkOk}$ were available when the service is invoked, the message sent by the *CreditRating*'s instance has been delivered to the correct instance.

It is worth noticing that, as witnessed by the above example, this correlation mechanism is flexible enough for allowing a single message to participate in *multiparty conversations* (indeed, the above conversation involves one provider service and two clients).

Notice also that, differently from other correlation-based formal languages for SOC, such as WS-CALCULUS [7], SOCK [12] and B*lite* [40], correlation variables in C⏲WS are not syntactically distinguished by other data variables. In fact, correlation variables can be recognized by their use (as the variables x_{ts} and x_{cc} of the example above). This is due to the fact that C⏲WS intends to be a foundational formalism, with a small number of simple primitives.

*3.2. μ*COWS *: the protection-, kill- and time-free fragment of* C⏲WS

The fragment of C⏲WS presented in this section, namely μCOWS, extends μCOWS^m with priority among concurrent activities.

3.2.1. Syntax and operational semantics

The syntax of μCOWS and the set of laws defining the structural congruence coincide with those of μCOWS^m, shown in Tables 1 and 2, respectively. Instead, the labelled transition relation

$$\frac{s_1 \xrightarrow{\mathtt{n} \rhd \bar{w}} s_1' \quad s_2 \xrightarrow{\mathtt{n} \lhd \bar{v}} s_2' \quad \mathcal{M}(\bar{w}, \bar{v}) = \sigma \quad \mathrm{noConf}(s_1 \mid s_2, \mathtt{n}, \bar{v}, |\sigma|)}{s_1 \mid s_2 \xrightarrow{\mathtt{n}\sigma|\sigma|\bar{v}} s_1' \mid s_2'} \; (com_2)$$

$$\frac{s_1 \xrightarrow{\alpha} s_1' \quad \alpha \neq \mathtt{n}\sigma\ell\bar{v}}{s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2} \; (par_2) \qquad \frac{s_1 \xrightarrow{\mathtt{n}\sigma\ell\bar{v}} s_1' \quad \mathrm{noConf}(s_2, \mathtt{n}, \bar{v}, \ell)}{s_1 \mid s_2 \xrightarrow{\mathtt{n}\sigma\ell\bar{v}} s_1' \mid s_2} \; (par_{com})$$

$$\frac{s \xrightarrow{\mathtt{n}\sigma \uplus \{x \mapsto v\}\,\ell\,\bar{v}} s'}{[x]\, s \xrightarrow{\mathtt{n}\sigma\ell\bar{v}} s' \cdot \{x \mapsto v\}} \; (del_{com\,2})$$

Table 6: μCOWS operational semantics (additional rules)

$$\mathrm{noConf}(\mathtt{u}!\bar{\epsilon}, \mathtt{n}, \bar{v}, \ell) \;=\; \mathrm{noConf}(\mathbf{0}, \mathtt{n}, \bar{v}, \ell) \;=\; \mathbf{true}$$

$$\mathrm{noConf}(\mathtt{n}'?\bar{w}.s, \mathtt{n}, \bar{v}, \ell) \;=\; \begin{cases} \mathbf{false} & \text{if } \mathtt{n}' = \mathtt{n} \wedge \; |\mathcal{M}(\bar{w}, \bar{v})| < \ell \\ \mathbf{true} & \text{otherwise} \end{cases}$$

$$\mathrm{noConf}(g + g', \mathtt{n}, \bar{v}, \ell) \;=\; \mathrm{noConf}(g, \mathtt{n}, \bar{v}, \ell) \;\wedge\; \mathrm{noConf}(g', \mathtt{n}, \bar{v}, \ell)$$

$$\mathrm{noConf}(s \mid s', \mathtt{n}, \bar{v}, \ell) \;=\; \mathrm{noConf}(s, \mathtt{n}, \bar{v}, \ell) \;\wedge\; \mathrm{noConf}(s', \mathtt{n}, \bar{v}, \ell)$$

$$\mathrm{noConf}([u]\, s, \mathtt{n}, \bar{v}, \ell) \;=\; \begin{cases} \mathrm{noConf}(s, \mathtt{n}, \bar{v}, \ell) & \text{if } u \notin \mathtt{n} \\ \mathbf{true} & \text{otherwise} \end{cases}$$

$$\mathrm{noConf}(* \, s, \mathtt{n}, \bar{v}, \ell) \;=\; \mathrm{noConf}(s, \mathtt{n}, \bar{v}, \ell)$$

Table 7: There are not conflicting receives along $\mathtt{n}$ matching $\bar{v}$

$\xrightarrow{\alpha}$ is the least relation over μCOWS services induced by the rules in Tables 4 and 6, where rules *(com$_2$)*, *(par$_2$)* and *(del$_{com\,2}$)* replace *(com)*, *(par)* and *(del$_{com}$)*, respectively. Labels are now generated by the following grammar:

$$\alpha \quad ::= \quad \mathtt{n} \lhd \bar{v} \;\; | \;\; \mathtt{n} \rhd \bar{w} \;\; | \;\; \mathtt{n}\sigma\ell\bar{v}$$

The new label $\mathtt{n}\sigma\ell\bar{v}$ enriches the previous communication label σ with information about the communication that has taken place, i.e. the endpoint, the transmitted values, and the length of the generated substitution. These information are carried during the inference of a computational step to establish a priority-based execution in the presence of conflicting receives. Specifically, $\mathtt{n}\sigma\ell\bar{v}$ (with ℓ natural number) denotes execution of a communication over $\mathtt{n}$ with matching values $\bar{v}$, originally generated substitution having ℓ pairs, and substitution σ to be still applied. Now, *computational steps* are denoted by labels of the form $\mathtt{n}\emptyset\ell\bar{v}$. Notation $\mathrm{u}(\alpha)$, indicating the set of names and variables occurring in α, is extended by letting $\mathrm{u}(\mathtt{n}\sigma\ell\bar{v}) = \mathrm{u}(\sigma)$.

The definition of the labelled transition relation exploits an auxiliary *no conflict* predicate $\mathrm{noConf}(s, \mathtt{n}, \bar{v}, \ell)$. The predicate, defined inductively by the clauses in Table 7, holds true if s cannot immediately perform a receive over the endpoint $\mathtt{n}$ matching $\bar{v}$ and generating a substitution σ with $|\sigma| < \ell$. Notably, in the clauses for the choice and parallel operators the predicate holds true if and only if all arguments of the operators do not contain conflicting receives.

We comment on the new rules. In μCOWS, as mentioned above, the communication label $\mathtt{n}\,\sigma\,\ell\,\bar{v}$, produced by rule *(com$_2$)*, carries information used to check the presence of conflicting receives in parallel components. Indeed, if more than one matching is possible, the receive that needs fewer substitutions is selected to progress (rules *(com$_2$)* and *(par$_{com}$)*). This mechanism permits to correlate different service communications thus implicitly creating interaction sessions and can be exploited to model the precedence of a service instance over the corresponding service specification when both can process the same request (see Section 3.2.2 for some examples). Rule *(del$_{com\,2}$)* is similar to *(del$_{com}$)* (shown in Table 4) but deals with labels generated by communications subject to priority. Notably, during the inference of a transition labelled by $\mathtt{n}\,\sigma\,\ell\,\bar{v}$, the length of the substitution to be applied decreases, while the length ℓ of the initial substitution does never change, which makes it suitable to check, in any moment, existence of a better matching, i.e. of parallel receives with greater priority. Execution of parallel services is interleaved (rule *(par$_2$)*), but when a communication is performed. In such case, the progress of the receive activity with greater priority must be ensured.

3.2.2. Examples

We present now some examples and observations that point out the peculiarities of μCOWS.

Multiple start activities. Services could be able of receiving multiple messages in a statically unpredictable order and in such a way that the first incoming message triggers the creation of a service instance which subsequent messages are routed to. This would require all those receive activities that can be immediately executed (according to [15], Section 16.3, there are *multiple start activities*) to share a non-empty set of variables (the so-called *correlation set*).

Consider, for example, a variant of the bank service that deals with joint accounts. Now, to charge a credit card associated to a joint account, the service requires each co-holder of the account to send a charge request, thus making sure that the transaction is authorized by all co-holders. An excerpt of such service running in parallel with two co-holder clients, willing to charge their card 1234 with 100 euros, is as follows:

$$
\begin{array}{l}
(p_{bank} \bullet o_{charge1}!\langle p_{cA}, 1234, 100, t_A\rangle \mid s_A) \\
\mid\ (p_{bank} \bullet o_{charge2}!\langle p_{cB}, 1234, 100, t_B\rangle \mid s_B) \\
\mid\ *[x_{cust1}, x_{cust2}, x_{cc}, x_{amount}, x_{ts1}, x_{ts2}]\,(p_{bank} \bullet o_{charge1}?\langle x_{cust1}, x_{cc}, x_{amount}, x_{ts1}\rangle.s_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid p_{bank} \bullet o_{charge2}?\langle x_{cust2}, x_{cc}, x_{amount}, x_{ts2}\rangle.s_2)
\end{array}
$$

After an interaction with the client B, an instance running in parallel with the service definition is created:

$$
\begin{array}{l}
(p_{bank} \bullet o_{charge1}!\langle p_{cA}, 1234, 100, t_A\rangle \mid s_A) \\
\mid\ s_B \\
\mid\ *[x_{cust1}, x_{cust2}, x_{cc}, x_{amount}, x_{ts1}, x_{ts2}]\,(\,p_{bank} \bullet o_{charge1}?\langle x_{cust1}, x_{cc}, x_{amount}, x_{ts1}\rangle\,.s_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid p_{bank} \bullet o_{charge2}?\langle x_{cust2}, x_{cc}, x_{amount}, x_{ts2}\rangle.s_2) \\
\mid\ [x_{cust1}, x_{ts1}]\,(\,p_{bank} \bullet o_{charge1}?\langle x_{cust1}, 1234, 100, x_{ts1}\rangle\,.s_1 \ \mid\ s_2) \cdot \sigma
\end{array}
$$

where σ is $\{x_{cust2} \mapsto p_{cB}, x_{cc} \mapsto 1234, x_{amount} \mapsto 100, x_{ts2} \mapsto t_B\}$. Now, the service definition and the created instance, being both able to receive the same tuple $\langle p_{cA}, 1234, 100, t_A\rangle$ along the endpoint $p_{bank} \bullet o_{charge1}$, compete for the request $p_{bank} \bullet o_{charge1}!\langle p_{cA}, 1234, 100, t_A\rangle$, i.e. in WS-BPEL jargon, two *conflicting* receive activities (in the term above, highlighted by a gray background)

are enabled. However, μCOWS's (prioritized) semantics, in particular rule *(com$_2$)* in combination with rule *(par$_{com}$)*, allows only the existing instance to evolve. Indeed, suppose to try to infer the transition corresponding to the interaction between client A and the service definition. Then, the generated substitution would have length 4 and, hence, let s_{inst} be the term representing the created instance, the predicate noConf$(s_{inst}, p_{bank} \bullet o_{charge1}, \langle p_{cA}, 1234, 100, t_A \rangle, 4)$ would not hold. In fact, the instance can perform a receive matching the same message and producing a substitution with fewer pairs (it has length 2). This way, the creation of a new instance is prevented and the only feasible computation leads to the following term:

$$\begin{array}{l} s_A \\ \mid\ s_B \\ \mid\ *\,[x_{cust1}, x_{cust2}, x_{cc}, x_{amount}, x_{ts1}, x_{ts2}]\,(p_{bank} \bullet o_{charge1}?\langle x_{cust1}, x_{cc}, x_{amount}, x_{ts1}\rangle.s_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid\ p_{bank} \bullet o_{charge2}?\langle x_{cust2}, x_{cc}, x_{amount}, x_{ts2}\rangle.s_2) \\ \mid\ (s_1 \ \mid\ s_2) \cdot \sigma \uplus \sigma' \end{array}$$

where σ' is $\{x_{cust1} \mapsto p_{cA}, x_{ts1} \mapsto t_A\}$.

It is worth noticing that the above considerations still hold if we use choice rather than parallel to compose the start activities of the bank service, as shown below:

$$\begin{array}{l} *\,[x_{cust1}, x_{cust2}, x_{cc}, x_{amount}, x_{ts1}, x_{ts2}] \\ \quad (p_{bank} \bullet o_{charge1}?\langle x_{cust1}, x_{cc}, x_{amount}, x_{ts1}\rangle.\, p_{bank} \bullet o_{charge2}?\langle x_{cust2}, x_{cc}, x_{amount}, x_{ts2}\rangle.\ \ldots) \\ \quad\ + p_{bank} \bullet o_{charge2}?\langle x_{cust2}, x_{cc}, x_{amount}, x_{ts2}\rangle.\, p_{bank} \bullet o_{charge1}?\langle x_{cust1}, x_{cc}, x_{amount}, x_{ts1}\rangle.\ \ldots) \end{array}$$

noConf *predicate.* Rules *(com$_2$)* and *(par$_{com}$)* use the predicate noConf$(_, \mathtt{n}, \bar{v}, \ell)$ for checking the presence of concurrent conflicting receives. When these rules must be used to infer a transition, a preventive α-conversion may be necessary. Indeed, condition noConf$(\mathtt{n}?\bar{w}.s, \mathtt{n}, \bar{v}, \ell)$ might single out patterns that could not really match the transmitted values. These 'false alarms' would block the inference (but allow us to stay on the 'safe' side).

For instance, consider the following term:

$$\mathtt{n}!\langle m\rangle \mid [x]\,\mathtt{n}?\langle x\rangle \mid [m]\,\mathtt{n}?\langle m\rangle \tag{1}$$

Apparently, both receive activities match the invoke activity, but only $\mathtt{n}?\langle x\rangle$ can synchronise with $\mathtt{n}!\langle m\rangle$, because the argument of $\mathtt{n}?\langle m\rangle$ is a restricted name, thus it is certainly different from the name transmitted by the invoke. However, if we try to naively infer the transition corresponding to the synchronisation between $\mathtt{n}!\langle m\rangle$ and $\mathtt{n}?\langle x\rangle$, we fail due to rules *(com$_2$)* or *(par$_{com}$)*. In fact, noConf$([m]\,\mathtt{n}?\langle m\rangle, \mathtt{n}, \langle m\rangle, 1)$ does not hold because $\mathcal{M}(m, m)$ produces the substitution $\emptyset$, that is smaller than $\{x \mapsto m\}$, that is produced by $\mathcal{M}(x, m)$.

However, the wanted transition can be inferred by first applying α-conversion. In fact, (1) can be re-written as follows:

$$\mathtt{n}!\langle m\rangle \mid [x]\,\mathtt{n}?\langle x\rangle \mid [m']\,\mathtt{n}?\langle m'\rangle$$

Now, it is clear that $\mathtt{n}?\langle m'\rangle$ is not a conflicting receive, because $\mathcal{M}(m', m)$ is undefined.

The same observations hold for the term:

$$[m]\,(\,\mathtt{n}!\langle m\rangle \mid [x]\,\mathtt{n}?\langle x\rangle\,) \mid \mathtt{n}?\langle m\rangle$$

Again, α-conversion is necessary for inferring the correct transitions. Instead, if in (1) we replace delimitation of m with that of $\mathtt{n}$, the correct transition can be directly inferred because noConf$([\mathtt{n}]\,\mathtt{n}?\langle m'\rangle, \mathtt{n}, \bar{v}, \ell)$ holds **true**.

Default behaviour. The previous examples show that the μCOWS's priority mechanism can be used for orchestration purposes, i.e. to properly coordinate interactions among services. However, this priority mechanism can be also exploited to coordinate activities (i.e. to mange their interdependencies) within the same service. For example, in the variant of the service *CreditRating* reported below

$$
\begin{aligned}
[x_{ts}, x_{cc}, x_a]\,(\; & p_{bank} \bullet o_{check}?\langle x_{ts}, 4321, x_a\rangle.\; p_{bank} \bullet o_{checkFail}!\langle x_{ts}, 4321, \mathit{ratingInfo}(4321, x_a)\rangle \\
& + p_{bank} \bullet o_{check}?\langle x_{ts}, 5432, x_a\rangle.\; p_{bank} \bullet o_{checkFail}!\langle x_{ts}, 5432, \mathit{ratingInfo}(5432, x_a)\rangle \\
& + p_{bank} \bullet o_{check}?\langle x_{ts}, 6543, x_a\rangle.\; p_{bank} \bullet o_{checkFail}!\langle x_{ts}, 6543, \mathit{ratingInfo}(6543, x_a)\rangle \\
& + p_{bank} \bullet o_{check}?\langle x_{ts}, x_{cc}, x_a\rangle. \\
& \qquad [p, o]\,(\; p \bullet o!\langle\rangle \mid p \bullet o?\langle\rangle.\; p_{bank} \bullet o_{checkOk}!\langle x_{ts}, x_{cc}, \mathit{ratingInfo}(x_{cc}, x_a)\rangle \\
& \qquad\qquad + p \bullet o?\langle\rangle.\; p_{bank} \bullet o_{checkFail}!\langle x_{ts}, x_{cc}, \mathit{ratingInfo}(x_{cc}, x_a)\rangle\;)
\end{aligned}
$$

the priority mechanism enables implementing a sort of 'default' behaviour. Indeed, when the service is invoked along the endpoint $p_{bank} \bullet o_{check}$ with a *black-listed* credit card number (e.g. numbers 4321, 5432, 6543) a negative response is returned; instead, if the credit card number is not in the black list, the service by default behaves in a non-deterministic way. For example, if *CreditRating* is invoked by $p_{bank} \bullet o_{check}!\langle t, 4321, 100\rangle$, although the invocation and the receive $p_{bank} \bullet o_{check}?\langle x_{ts}, x_{cc}, x_a\rangle$ do match, the priority mechanism ensures that the service replies with $p_{bank} \bullet o_{checkFail}!\langle t, 4321, \mathit{ratingInfo}(4321, 100)\rangle$.

3.3. COWS *: the time-free fragment of* C⏲WS

COWS, which is basically the untimed fragment of C⏲WS, is obtained by enriching μCOWS with two primitives permitting to express transactional behaviours of services and scenarios with fault and compensation handling.

3.3.1. Syntax

The syntax of COWS is given in Table 8. Besides the sets of values and variables, we also use a countable set of *(killer) labels* (ranged over by $k, k', \ldots$). Services syntax is extended with the kill activity **kill**(_) and the protection operator $\{\!|\_|\!\}$, while now the delimitation $[\_]\_$ accepts as first argument also killer labels (the new constructs are highlighted in Table 8 by a gray background). The *kill* activity forces the immediate termination of concurrent activities which are not enclosed within the *protection* operator. The *delimitation* of a killer label is then used to confine the killing effect. Notably, expressions do not include killer labels that, hence, are *non-communicable* values. This way the scope of killer labels cannot be dynamically extended and the activities whose termination would be forced by execution of a kill can be statically determined.

We still use w to range over values and variables, u to range over names and variables, while we use e to range over *elements*, namely killer labels, names and variables. Delimitation now is a binder also for killer labels. fe(t) denotes the set of free elements in t, and fk(t) denotes the set of free killer labels in t. A closed service is a COWS term without free variables and killer labels.

3.3.2. Operational semantics

The structural congruence $\equiv$ for COWS, besides the laws in Table 2, additionally includes the laws in Table 9. Notably, the last law of Table 9 prevents extending the scope of a killer label k when it is free in s_1 or s_2 (this avoids involving s_1 in the effect of a kill activity inside s_2 and is essential to statically determine which activities can be terminated by a kill). Thus, this law can be used to garbage-collect killer labels, e.g. $[k]\,\mathrm{n}!\bar{\epsilon} \equiv [k]\,(\mathrm{n}!\bar{\epsilon} \mid \mathbf{0}) \equiv \mathrm{n}!\bar{\epsilon} \mid [k]\,\mathbf{0} \equiv \mathrm{n}!\bar{\epsilon} \mid \mathbf{0} \equiv \mathrm{n}!\bar{\epsilon}$.

Killer labels: $k, k', \ldots$	*Elements (Labels/Vars/Names):* $e, e', \ldots$
Expressions: $\epsilon, \epsilon', \ldots$	*Variables/Names:* $u, u', \ldots$
Variables: $x, y, \ldots$	*Variables/Values:* $w, w', \ldots$
Values: $v, v', \ldots$	
Names: $n, m, \ldots$	*Endpoints:*
Partners: $p, p', \ldots$	without variables: $p \bullet o, \mathtt{n}, \ldots$
Operations: $o, o', \ldots$	may contain variables: $u \bullet u', \mathtt{u}, \ldots$

Services:

s ::=

	$\mathbf{kill}(k)$	(kill)
$\mid$	$u \bullet u'!\bar{\epsilon}$	(invoke)
$\mid$	g	(receive-guarded choice)
$\mid$	$s \mid s$	(parallel composition)
$\mid$	$\{\!\{s\}\!\}$	(protection)
$\mid$	$[e]\, s$	(delimitation)
$\mid$	$* s$	(replication)

Receive-guarded choice:

g ::=

	$\mathbf{0}$	(nil)
$\mid$	$p \bullet o?\bar{w}.s$	(request processing)
$\mid$	$g + g$	(choice)

Table 8: COWS syntax

$$\{\!\{\mathbf{0}\}\!\} \equiv \mathbf{0} \qquad [k]\,\mathbf{0} \equiv \mathbf{0}$$

$$\{\!\{\{\!\{s\}\!\}\}\!\} \equiv \{\!\{s\}\!\} \qquad [e_1]\,[e_2]\,s \equiv [e_2]\,[e_1]\,s$$

$$\{\!\{[e]\,s\}\!\} \equiv [e]\,\{\!\{s\}\!\} \qquad s_1 \mid [k]\,s_2 \equiv [k]\,(s_1 \mid s_2) \quad \text{if } k \notin \mathrm{fk}(s_1) \cup \mathrm{fk}(s_2)$$

Table 9: COWS structural congruence (additional laws)

To define the labelled transition relation, we need two new auxiliary functions. The function $halt(\_)$ takes a service s as an argument and returns the service obtained by only retaining the protected activities inside s. $halt(\_)$ is defined inductively on the syntax of services. The most significant case is $halt(\{\!\{s\}\!\}) = \{\!\{s\}\!\}$. In the other cases, $halt(\_)$ returns $\mathbf{0}$, except for parallel composition, delimitation and replication operators, for which it acts as an homomorphism.

$$halt(\mathbf{kill}(k)) = halt(\mathtt{u}!\bar{\epsilon}) = halt(g) = \mathbf{0} \qquad halt(\{\!\{s\}\!\}) = \{\!\{s\}\!\}$$

$$halt(s_1 \mid s_2) = halt(s_1) \mid halt(s_2) \qquad halt([e]\,s) = [e]\,halt(s) \qquad halt(* s) = * halt(s)$$

Then, in Table 10, we inductively define the predicate noKill(s, e), that holds true if either e is not a killer label or $e = k$ and s cannot immediately perform a free kill activity $\mathbf{kill}(k)$. Moreover, the predicate noConf$(s, \mathtt{n}, \bar{v}, \ell)$, defined for μCOWS by the rules in Table 7, is extended to COWS by adding the following rules:

$$\mathrm{noConf}(\mathbf{kill}(k), \mathtt{n}, \bar{v}, \ell) = \mathbf{true} \qquad \mathrm{noConf}(\{\!\{s\}\!\}, \mathtt{n}, \bar{v}, \ell) = \mathrm{noConf}(s, \mathtt{n}, \bar{v}, \ell)$$

$$\mathrm{noConf}([e]\,s, \mathtt{n}, \bar{v}, \ell) = \begin{cases} \mathrm{noConf}(s, \mathtt{n}, \bar{v}, \ell) & \text{if } e \notin \mathtt{n} \\ \mathbf{true} & \text{otherwise} \end{cases}$$

The labelled transition relation $\xrightarrow{\alpha}$ is the least relation over services induced by the rules in Tables 4, 6 and 11, where *(com$_2$)*, *(del$_2$)* and *(del$_{com2}$)* replace *(com)*, *(del)* and *(del$_{com}$)*, respectively,

$\mathrm{noKill}(s, e) = \mathbf{true} \quad \text{if } \mathrm{fk}(e) = \emptyset$	$\mathrm{noKill}(s \mid s', k) = \mathrm{noKill}(s, k) \wedge \mathrm{noKill}(s', k)$
$\mathrm{noKill}(\mathbf{kill}(k), k) = \mathbf{false}$	$\mathrm{noKill}([e]\, s, k) = \mathrm{noKill}(s, k) \quad \text{if } e \neq k$
$\mathrm{noKill}(\mathbf{kill}(k'), k) = \mathbf{true} \quad \text{if } k \neq k'$	$\mathrm{noKill}([k]\, s, k) = \mathbf{true}$
$\mathrm{noKill}(\mathrm{u}!\bar{\varepsilon}, k) = \mathrm{noKill}(g, k) = \mathbf{true}$	$\mathrm{noKill}(\{\!|s|\!\}, k) = \mathrm{noKill}(* s, k) = \mathrm{noKill}(s, k)$

Table 10: There are no active **kill**(k)

$$\mathbf{kill}(k) \xrightarrow{k} \mathbf{0} \quad \textit{(kill)} \qquad \frac{s \xrightarrow{\alpha} s'}{\{\!|s|\!\} \xrightarrow{\alpha} \{\!|s'|\!\}} \quad \textit{(prot)}$$

$$\frac{s_1 \xrightarrow{\alpha} s_1' \quad \alpha \neq k, \mathrm{n}\,\sigma\,\ell\,\bar{v}}{s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2} \quad \textit{(par}_3\textit{)} \qquad \frac{s_1 \xrightarrow{k} s_1'}{s_1 \mid s_2 \xrightarrow{k} s_1' \mid \mathit{halt}(s_2)} \quad \textit{(par}_{\mathit{kill}}\textit{)}$$

$$\frac{s \xrightarrow{k} s'}{[k]\, s \xrightarrow{\dagger} [k]\, s'} \quad \textit{(del}_{\mathit{kill}\,1}\textit{)} \qquad \frac{s \xrightarrow{k} s' \quad k \neq e}{[e]\, s \xrightarrow{k} [e]\, s'} \quad \textit{(del}_{\mathit{kill}\,2}\textit{)}$$

$$\frac{s \xrightarrow{\dagger} s'}{[e]\, s \xrightarrow{\dagger} [e]\, s'} \quad \textit{(del}_{\mathit{kill}\,3}\textit{)} \qquad \frac{s \xrightarrow{\alpha} s' \quad e \notin \mathrm{e}(\alpha) \quad \alpha \neq k, \dagger \quad \mathrm{noKill}(s, e)}{[e]\, s \xrightarrow{\alpha} [e]\, s'} \quad \textit{(del}_2\textit{)}$$

Table 11: COWS operational semantics (additional rules)

and *(par*$_3$*)* replaces rules *(par)* and *(par*$_2$*)*. Labels are now generated by the following grammar:

$$\alpha \quad ::= \quad \mathrm{n} \lhd \bar{v} \quad | \quad \mathrm{n} \rhd \bar{w} \quad | \quad \mathrm{n}\,\sigma\,\ell\,\bar{v} \quad | \quad k \quad | \quad \dagger$$

The meaning of the new labels is as follows: k denotes execution of a request for terminating a term from within the delimitation $[k]$, and $\dagger$ denotes a computational step corresponding to taking place of forced termination. In the sequel, we use $\mathrm{e}(\alpha)$ to denote the set of elements occurring in α (it is defined similarly to $\mathrm{u}(\alpha)$, Section 3.1.2, page 12, and Section 3.2.1, page 19).

Let us now comment on the added rules. Activity **kill**(k) forces termination of all unprotected parallel activities (rules *(kill)* and *(par*$_{\mathit{kill}}$*)*) inside an enclosing $[k]$, that stops the killing effect by turning the transition label k into $\dagger$ (rule *(del*$_{\mathit{kill}\,1}$*)*). Such delimitation, whose existence is ensured by the assumption that the semantics is only defined for closed services, prevents a single service to be capable to stop all the other parallel services, which would be unreasonable in a service-oriented setting (as services are loosely coupled and organized in different administrative domains). Critical activities can be protected from killing by putting them into a protection $\{\!|\_|\!\}$; this way, $\{\!|s|\!\}$ behaves like s (rule *(prot)*). Similarly, $[e]\, s$ behaves like s (rule *(del*$_2$*)*), except when the transition label α contains e, in which case α must correspond either to a communication assigning a value to e (rule *(del*$_{\mathit{com}\,2}$*)*) or to a kill activity for e (rule *(del*$_{\mathit{kill}\,1}$*)*), or when a free kill activity for e is active in s, in which case only actions corresponding to kill activities can be executed (rules *(del*$_{\mathit{kill}\,2}$*)* and *(del*$_{\mathit{kill}\,3}$*)*). This means that kill activities are executed *eagerly* with respect to the activities enclosed within the delimitation of the corresponding killer label. Execution of parallel services is interleaved (rule *(par*$_3$*)*), but when a kill activity or a communication is performed. Indeed, the former must trigger termination of all parallel services (according to

rule *(par$_{kill}$)*), while the latter must ensure that the receive activity with greater priority progresses (rules *(com$_2$)* and *(par$_{com}$)*).

3.3.3. Examples

We present here some examples aimed at clarifying the peculiar features of COWS. We will show in Section 4 how the COWS activities dealing with termination, i.e. kill and protection, can be used for implementing fault and compensation handling.

Protected kill activity. The following simple example illustrates the effect of executing a kill activity within a protection block:

$$[k]\,(\{\!|s_1 \mid \{\!|s_2|\!\} \mid \mathbf{kill}(k)|\!\} \mid s_3) \mid s_4 \xrightarrow{\dagger} [k]\,\{\!|\{\!|s_2|\!\}|\!\} \mid s_4$$

where, for simplicity, we assume that $halt(s_1) = halt(s_3) = \mathbf{0}$. In essence, $\mathbf{kill}(k)$ terminates all parallel services inside delimitation $[k]$ (i.e. s_1 and s_3), except those that are protected at the same nesting level of the kill activity (i.e. s_2).

Interplay between communication and kill activity. Kill activities can break communication, as the following example shows:

$$\mathtt{n}!\langle v\rangle \mid [k]\,([x]\,\mathtt{n}?\langle x\rangle.s \mid \mathbf{kill}(k)) \xrightarrow{\dagger} \mathtt{n}!\langle v\rangle \mid [k]\,[x]\,\mathbf{0}$$

In fact, due to the priority of the kill activity over communication, this is the only feasible computational step of the above term. Communication can however be guaranteed by protecting the receive activity, as follows

$$\begin{array}{ll}
\mathtt{n}!\langle v\rangle \mid [k]\,([x]\,\{\!|\mathtt{n}?\langle x\rangle.s|\!\} \mid \mathbf{kill}(k)) & \xrightarrow{\dagger} \\
\mathtt{n}!\langle v\rangle \mid [k]\,[x]\,\{\!|\mathtt{n}?\langle x\rangle.s|\!\} & \equiv \\
[x]\,(\mathtt{n}!\langle v\rangle \mid [k]\,\{\!|\mathtt{n}?\langle x\rangle.s|\!\}) & \xrightarrow{\mathtt{n}\,\emptyset\,1\,\langle v\rangle} \\
[k]\,\{\!|s \cdot \{x \mapsto v\}|\!\} &
\end{array}$$

Notably, priority of kill activities over communication acts only with respect to the activities enclosed within the delimitation of the corresponding killer labels (i.e. priority is *local* to killer label scopes). For instance, if we re-write the above example as follows:

$$[y]\,\mathtt{n}?\langle y\rangle.s' \mid \mathtt{n}!\langle v\rangle \mid [k]\,([x]\,\mathtt{n}?\langle x\rangle.s \mid \mathbf{kill}(k))$$

communication between $\mathtt{n}!\langle v\rangle$ and $\mathtt{n}?\langle x\rangle$ is still preempted by $\mathbf{kill}(k)$, while communication with $\mathtt{n}?\langle y\rangle$ can take place and lead to

$$s' \cdot \{y \mapsto v\} \mid [k]\,([x]\,\mathtt{n}?\langle x\rangle.s \mid \mathbf{kill}(k))$$

Non-communicability of killer labels. We require killer labels not to be communicable to avoid a service be capable to indiscriminately stop the execution of other services' activities. However, when desired, this behaviour can be modelled in COWS. Consider, for example, the following term where two parallel services share the private name *stop*:

$$[stop]\,(\,s_1 \mid s_2\,) \mid s_3$$

where $s_1 \triangleq [k]\,(\mathrm{n}?\langle stop\rangle.\,\mathbf{kill}(k) \mid s'_1)$ and $s_2 \triangleq \mathrm{n}!\langle stop\rangle \mid s'_2$. In s_1, the activity $\mathbf{kill}(k)$ is prefixed by the receive $\mathrm{n}?\langle stop\rangle$ that does not allow forced termination to take place until the 'termination signal' *stop* is received. In fact, if a communication between s_1 and s_2 takes place along the endpoint n, the term evolves to

$$[stop]\,(\,[k]\,(\mathbf{kill}(k) \mid s'_1)\;\mid\; s'_2\,)\;\mid\; s_3$$

Now, due to the priority of the kill activity over communication, the term $[k]\,(\mathbf{kill}(k) \mid s'_1)$ can only perform a kill activity and evolve, e.g., to $[k]\,halt(s'_1)$.

3.4. C⏲WS

The full calculus, C⏲WS, is obtained by enriching COWS with an analogous of WS-BPEL's *wait* activity [15, Section 10.7] which causes execution of the invoking service to be suspended until the time interval specified as an argument has elapsed[3]. The extension of COWS with specific activities dealing with time is motivated by the fact that it is still unknown to what extent timed computation can be reduced to untimed forms of computation [41].

3.4.1. Syntax

We assume that the set of values now includes a set of positive numbers (ranged over by δ, δ', ...), used to represent *time intervals*. The syntax of COWS is extended as follows (the new construct is highlighted by a gray background):

$$g \;::=\; \mathbf{0} \;\mid\; p \bullet o?\bar{w}.s \;\mid\; g + g \;\mid\; \colorbox{gray!10}{$\textcircled{\scriptsize t}_{\epsilon}.s$}$$

Basically, guards are extended with the *wait activity* $\textcircled{\scriptsize t}_{\epsilon}$, that specifies the time interval, whose value is given by evaluation of ϵ, the executing service has to wait for. Consequently, the choice construct can now be guarded both by message reception and timeout expiration, like WS-BPEL *pick* activity [15, Section 11.5]. We assume that evaluation of expressions and execution of basic activities, except for $\textcircled{\scriptsize t}_{\epsilon}$, are instantaneous (i.e. do not consume time units) and that time elapses between them.

3.4.2. Operational semantics

The operational semantics of C⏲WS is defined in terms of the labelled transition relation $\xrightarrow{\hat{\alpha}}$, where $\hat{\alpha}$ stands for α or δ (that models time elapsing), obtained by adding the rules shown in Table 12 to those defining the semantics of COWS (see Section 3.3.2 and Tables 4, 6 and 11). Let us briefly comment on the new rules. Time can elapse while waiting on receive/invoke activities, rules *(rec$_{elaps}$)* and *(inv$_{elaps}$)*. When time elapses, but the timeout is still not expired, the argument of wait activities is updated (rule *(wait$_{elaps}$)*). Time elapsing cannot make a choice within a choice activity (rule *(choice$_2$)*), while the occurrence of a timeout can. Indeed, this is signalled by label †, thus it is a computational step, generated by rule *(wait$_{tout}$)* and used by rule *(choice)* (in Table 4) to discard the alternative branches. Time elapses synchronously for all services running in parallel: this is modelled by rule *(par$_{sync}$)* and by the remaining rules for empty activity (rule *(nil$_{elaps}$)*), replication (rule *(rep$_{elaps}$)*), wait activity (rule *(wait$_{err}$)*), protection

[3]For the sake of simplicity, we do not consider here the 'until' variant of the wait activity, which causes suspension of the invoking service until the *absolute* time reaches the value specified as an argument, and refer the interested reader to [21] for an account of this variant.

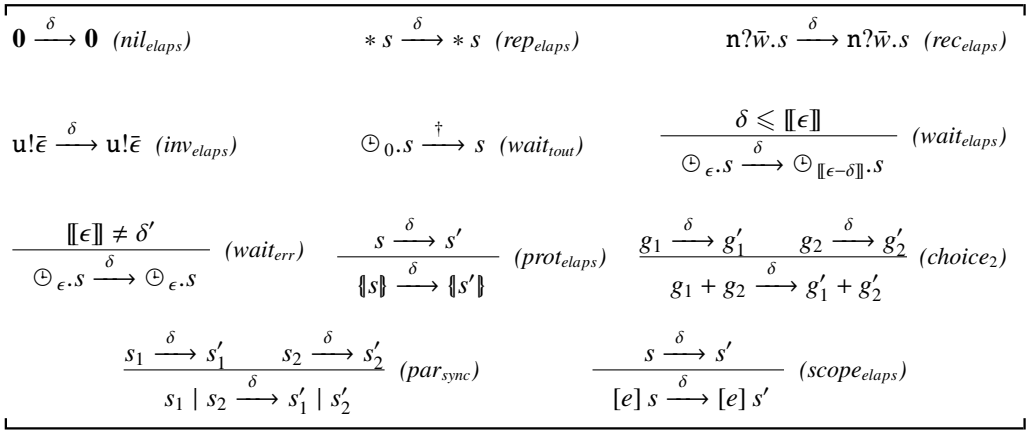

Table 12: C⏲WS operational semantics (additional rules)

(rule $(prot_{elaps})$) and delimitation (rule $(scope_{elaps})$). In particular, rule $(wait_{err})$ enables time passing for the wait activity also when the expression ϵ used as an argument does not return a positive number; in this case the argument of the wait is left unchanged. Note that, in agreement with its eager semantics, the kill activity does not allow time to pass. In C⏲WS, computational steps also include transitions labelled by δ corresponding to time elapsing.

Since time elapses synchronously for all services in parallel, we can think of as all services run on a same service *engine* and share the same clock. By further extending the language syntax, as shown in [21], we can make explicit the notion of service engine and of deployment of services on engines. This way, we can model time so that it progresses synchronously for services located within the same engine and asynchronously among services deployed onto different engines.

3.4.3. Examples

We end this section with some examples of application of the the timed constructs provided by the full language C⏲WS. Such constructs are also exploited in Section 5.2 to model a variant of the automotive case study presented in Section 2.2.

WS-BPEL pick activity. Consider again the bank service scenario used in other previous examples in Sections 3.1.3 and 3.2.2, where now clients, after having sent requests for charging their credit cards, wait for a response for a given amount of time. By using the wait activity and the choice operator, we can define in C⏲WS a client service implementing a pick activity *à la* WS-BPEL as follows:

$$p_{bank} \bullet o_{charge}!\langle p_{cA}, 1234, 100, t_A\rangle \;\mid\; [x, x_i]\,(\, p_{cA} \bullet o_{resp}?\langle x, t_A, x_i\rangle.s_A) \\ + \textcircled{\scriptsize 0}_{15}\,.\, s_{chargeTimeoutExpired}\,)$$

If the *Bank* service does not reply in the given amount of time units (e.g. 15 minutes), the client service will discard the client activities $p_{cA} \bullet o_{resp}?\langle x, t_A, x_i\rangle.s_A$ (hence, the activity s_A dealing with the *Bank* response will never be carried out) and execute the activity $s_{chargeTimeoutExpired}$ handling the non-response event. This latter activity can, e.g., ask the driver to provide the data of another credit card, or simply show an error message inviting the driver to contact the assistance services by herself/himself; in any case $s_{chargeTimeoutExpired}$ may contact or not the *Bank* service to inform it that the response to the sent request is not waited any longer. Of course, if a response from the *Bank* is received before the timeout expiration, the timeout is disabled and s_A is executed.

Time-bound search. Consider a registry service storing information about on road services and providing searching functionalities to its clients (see, e.g., Section 5.2). A search that continues to query the data stored in the registry until a given timeout expires can be rendered in C⏲WS as a term of the following form:

$$[k]\,(\,s_{search}\;\mid\;\oplus_{\delta}.\,(\,\mathbf{kill}(k)\mid\{\!|\,s_{searchComplete}\,|\!\}\,)\,)$$

where s_{search} performs the search and $s_{searchComplete}$ sends the search result to the client. After δ time units, the search is stopped by means of the kill activity and then the result is communicated.

4. The automotive case study: specification and analysis in COWS

We present in this section the most relevant parts of the specification in COWS of the automotive case study introduced in Section 2.2 (the complete specification is reported in [42]) and provide a brief description of a few properties that it satisfies. Notably, to specify the case study we use COWS rather than C⏲WS, because the verification methods and tools currently available only apply to the former language. We further refine the case study and its specification, in order to illustrate an application of the C⏲WS constructs for managing time and constraints, later on in Section 5.2.

The COWS term modelling the overall scenario is:

$$[p_{car}]\,(\,\mathit{SensorsMonitor}\;\mid\;\mathit{GpsSystem}\;\mid\;\mathit{Discovery}\;\mid\;\mathit{Reasoner}\;\mid\;\mathit{Orchestrator}\,)\;\mid\;\mathit{Bank}\;\mid\;\mathit{OnRoadRepairServices}$$

All services of the in-vehicle platform share a private partner name p_{car}, that is used for intra-vehicle communication and is passed to external services (e.g. the bank service) for receiving data from them.

When an engine failure occurs, a signal (raised by *SensorsMonitor*) triggers the execution of the *Orchestrator* and activates the corresponding 'recovery' service. *Orchestrator*, the most important component of the in-vehicle platform, is

$$[x_{carData}, x_{ts}]\,(p_{car}\bullet o_{engineFailure}?\langle x_{ts}, x_{carData}\rangle.s_{engfail} + p_{car}\bullet o_{lowOilFailure}?\langle x_{ts}, x_{carData}\rangle.s_{lowoil} + \ldots)$$

This term uses the choice operator $\_+\_$ to pick one of those alternative recovery behaviours whose execution can start immediately. Notice that, while executing a recovery behaviour, *Orchestrator* does not accept other recovery requests. We are also assuming, for the sake of simplicity, that it is reinstalled at the end of the recovery task.

The recovery behaviour $s_{engfail}$ executed when an engine failure occurs is

$$[p_{end}, o_{end}, x_{info}, x_{loc}, x_{list}, o_{undo}]\;(\,[k]\,(\,\mathit{CardCharge}\mid\mathit{FindServices}\,)\;\mid\;p_{end}\bullet o_{end}?\langle\rangle.\,p_{end}\bullet o_{end}?\langle\rangle.\,\mathit{ChooseAndOrder}\,)$$

$p_{end}\bullet o_{end}$ is a scoped endpoint along which successful termination signals (i.e. communications that carry no data) are exchanged to orchestrate execution of the different components. *CardCharge* corresponds to the homonymous UML action of Figure 2, while *FindServices* corresponds to the sequential composition of the UML actions RequestLocation and FindServices. The two terms are defined as follows:

$$
\begin{aligned}
\mathit{CardCharge} \triangleq\ & p_{bank} \cdot o_{charge}!\langle p_{car}, ccNum, amount, x_{ts}\rangle \\
& \mid \{\!| p_{car} \cdot o_{resp}?\langle fail, x_{ts}, x_{info}\rangle.\ \mathbf{kill}(k) \\
& \quad + p_{car} \cdot o_{resp}?\langle ok, x_{ts}, x_{info}\rangle. \\
& \quad\ (\ p_{end} \cdot o_{end}!\langle\rangle \\
& \quad\quad \mid p_{car} \cdot o_{undo}?\langle cc\rangle.\ p_{car} \cdot o_{undo}?\langle cc\rangle.\ p_{bank} \cdot o_{revoke}!\langle x_{ts}, ccNum\rangle\ )\ |\!\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{FindServices} \triangleq\ & p_{car} \cdot o_{reqLoc}!\langle\rangle \\
& \mid p_{car} \cdot o_{respLoc}?\langle x_{loc}\rangle. \\
& \quad (\ p_{car} \cdot o_{findServ}!\langle x_{loc}, servicesType\rangle \\
& \quad\ \mid p_{car} \cdot o_{found}?\langle x_{list}\rangle.\ p_{end} \cdot o_{end}!\langle\rangle \\
& \quad\ + p_{car} \cdot o_{notFound}?\langle\rangle. \\
& \quad\quad (\ \{\!| p_{car} \cdot o_{undo}!\langle cc\rangle \mid p_{car} \cdot o_{undo}!\langle cc\rangle |\!\} \mid \mathbf{kill}(k)\ )\ )
\end{aligned}
$$

Therefore, the recovery service concurrently contacts service *Bank*, to charge the car owner's credit card with a security amount, and services *GpsSystem* and *Discovery*, to get the car's location (stored in x_{loc}) and a list of on road services (stored in x_{list}). When both activities terminate (the fresh endpoint $p_{end} \cdot o_{end}$ is used to appropriately synchronise their successful terminations), the recovery service forwards the obtained list to service *Reasoner*, that will choose the most convenient services (see definition of *ChooseAndOrder*). Whenever services finding fails, *FindServices* terminates the whole recovery behaviour (by means of the kill activity $\mathbf{kill}(k)$) and sends two signals *cc* (abbreviation of 'card charge') along the endpoint $p_{car} \cdot o_{undo}$. Similarly, if charging the credit card fails, then *CardCharge* terminates the whole recovery behaviour. Otherwise, it installs a compensation handler that takes care of revoking the credit card charge. Activation of this compensation activity requires two signals *cc* along $p_{car} \cdot o_{undo}$ and, thus, takes place either whenever *FindService* fails or, as we will see soon, whenever both garage and car rental orders fail.

ChooseAndOrder tries to order the selected services by contacting a car rental and, concurrently, a garage and a tow truck. It is defined as follows:

$$
\begin{aligned}
& [x_{gps}]\,(\ p_{car} \cdot o_{choose}!\langle x_{list}\rangle \\
& \qquad \mid [x_{garage}, x_{towTruck}, x_{rentalCar}]\ p_{car} \cdot o_{chosen}?\langle x_{garage}, x_{towTruck}, x_{rentalCar}\rangle. \\
& \qquad\qquad (\ \mathit{OrderGarageAndTowTruck} \mid \mathit{RentCar}\ )\ )
\end{aligned}
$$

$$
\begin{aligned}
\mathit{OrderGarageAndTowTruck} \triangleq\ & [x_{garageInfo}] \\
& (\ x_{garage} \cdot o_{orderGar}!\langle p_{car}, x_{carData}\rangle \\
& \ \mid p_{car} \cdot o_{garageFail}?\langle\rangle. \\
& \quad\ (\ p_{car} \cdot o_{undo}!\langle cc\rangle \mid [p, o]\,(p \cdot o!\langle x_{loc}\rangle \mid p \cdot o?\langle x_{gps}\rangle)\ ) \\
& \quad + p_{car} \cdot o_{garageOk}?\langle x_{gps}, x_{garageInfo}\rangle. \\
& \quad\ (\ \mathit{OrderTowTruck} \\
& \quad\quad \mid p_{car} \cdot o_{undo}?\langle gar\rangle. \\
& \quad\quad\ (\ x_{garage} \cdot o_{cancel}!\langle p_{car}\rangle \\
& \quad\quad\quad \mid p_{car} \cdot o_{undo}!\langle cc\rangle \mid p_{car} \cdot o_{undo}!\langle rc\rangle\ )\ )\ )
\end{aligned}
$$

$$
\begin{aligned}
\mathit{OrderTowTruck} \triangleq\ & [x_{towInfo}] \\
& (\ x_{towTruck} \cdot o_{orderTow}!\langle p_{car}, x_{loc}, x_{gps}\rangle \\
& \ \mid p_{car} \cdot o_{towTruckFail}?\langle\rangle.\ p_{car} \cdot o_{undo}!\langle gar\rangle \\
& \quad + p_{car} \cdot o_{towTruckOK}?\langle x_{towInfo}\rangle\ )
\end{aligned}
$$

$$
\begin{aligned}
\mathit{RentCar} \triangleq\ & [x_{rcInfo}] \\
& (\, x_{rentalCar} \cdot o_{orderRC}!\langle p_{car}, x_{gps}\rangle \\
& \quad \mid\ p_{car} \cdot o_{rentalCarFail}?\langle\rangle.\ p_{car} \cdot o_{undo}!\langle cc\rangle \\
& \quad\ + p_{car} \cdot o_{rentalCarOK}?\langle x_{rcInfo}\rangle. \\
& \qquad p_{car} \cdot o_{undo}?\langle rc\rangle.\ x_{rentalCar} \cdot o_{redirect}!\langle p_{car}, x_{loc}\rangle\,)
\end{aligned}
$$

If ordering a garage fails, the compensation of the credit card charge is invoked by sending a signal cc along the endpoint $p_{car} \cdot o_{undo}$, and the car's location (stored in x_{loc}) is assigned to variable x_{gps} (whose value will be passed to the rental car service). This assignment is rendered as a communication along the private endpoint $p \cdot o$. Otherwise, the tow truck ordering starts and the garage's location is assigned to variable x_{gps}. Moreover, a compensation handler is installed; it will be activated whenever tow truck ordering fails and, in that case, attempts to cancel the garage order (by invoking operation o_{cancel}) and to compensate the credit card charge and the rental car order (by sending signal cc and rc along $p_{car} \cdot o_{undo}$). Renting a car proceeds concurrently and, in case of successful completion, the compensation handler for the redirection of the rented car is installed; otherwise, the compensation of the credit card charge is invoked.

For the sake of presentation, we relegate the specification of the remaining components of the in-vehicle platform, i.e. *SensorsMonitor*, *GpsSystem*, *Discovery* and *Reasoner*, to [42].

The COWS specification of the service *Bank* is given by the compound term introduced in Section 3.1.3 (paragraph "Message correlation" at page 16) where the subservice *BankInterface* is extended with compensation activities (highlighted below by a gray background) for revoking credit card charges:

$$
\begin{aligned}
& \mathit{BankInterface} \triangleq \\
& [x_{cust}, x_{cc}, x_{amount}, x_{ts}] \\
& p_{bank} \cdot o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}, x_{ts}\rangle. \\
& (\, p_{bank} \cdot o_{check}!\langle x_{ts}, x_{cc}, x_{amount}\rangle \\
& \quad \mid [x_{info}]\,(p_{bank} \cdot o_{checkFail}?\langle x_{ts}, x_{cc}, x_{info}\rangle.\ x_{cust} \cdot o_{resp}!\langle \mathit{fail}, x_{ts}, x_{info}\rangle \\
& \qquad\ + \ p_{bank} \cdot o_{checkOk}?\langle x_{ts}, x_{cc}, x_{info}\rangle. \\
& \qquad\quad [k']\,(\, x_{cust} \cdot o_{resp}!\langle ok, x_{ts}, x_{info}\rangle \mid p_{bank} \cdot o_{revoke}?\langle x_{ts}, x_{cc}\rangle.\ \mathbf{kill}(k')\,)\ )
\end{aligned}
$$

In case of a positive answer, the possibility of revoking the request through invocation of operation o_{revoke} is enabled (in fact, should the discovery phase or ordering the services fail, the customer charge operation should be cancelled in order to implement the wanted transactional behaviour). Revocation causes deletion of the reply to the client, if this has still to be performed.

OnRoadRepairServices is actually a composition of various on road services, i.e. it is

$$
\mathit{Garage}_1 \mid \mathit{Garage}_2 \mid \mathit{TowTruck}_1 \mid \mathit{TowTruck}_2 \mid \mathit{RentalCar}_1 \mid \mathit{RentalCar}_2 \mid \ldots
$$

Such concurrent on road services are all modelled in a similar way, e.g.

$$
\begin{aligned}
\mathit{Garage}_i \triangleq\ * & [x_{cust}, x_{sensorsData}, o_{checkOK}, o_{checkFail}] \\
& p_{garage\_i} \cdot o_{orderGar}?\langle x_{cust}, x_{sensorsData}\rangle. \\
& \quad (\, p_{garage\_i} \cdot o_{checkOK}!\langle\rangle \mid p_{garage\_i} \cdot o_{checkFail}!\langle\rangle \\
& \qquad \mid\ p_{garage\_i} \cdot o_{checkFail}?\langle\rangle.\ x_{cust} \cdot o_{garageFail}!\langle\rangle \\
& \qquad\ + p_{garage\_i} \cdot o_{checkOK}?\langle\rangle. \\
& \qquad\quad [k]\,(\, x_{cust} \cdot o_{garageOK}!\langle \mathit{garageGPS}_i, \mathit{garageInfo}_i\rangle \\
& \qquad\qquad \mid\ p_{garage\_i} \cdot o_{cancel}?\langle x_{cust}\rangle.\ \mathbf{kill}(k)\,)\,)
\end{aligned}
$$

For simplicity, success or failure of garage orders are modelled by means of non-deterministic choice by exploiting internal operations $o_{checkOK}$ and $o_{checkFail}$.

To give a flavour of which kind of analyses COWS's specifications can be subject to, we end this section by illustrating some properties of the automotive case study that can be verified by using two of the techniques devised so far.

The type system introduced in [17] uses types to express and enforce policies for regulating the exchange of data among services. Over the specification of the automotive scenario, this approach enables the verification of such confidentiality properties as, e.g., "*information about the credit card and location of a driver in trouble cannot become available to unauthorized users*" and "*critical data sent by on-road services to the in-vehicle services, e.g. cost and quality of the service supplied, are not disclosed to competitors*".

The logical verification methodology presented in [18] permits describing service properties by means of a branching-time temporal logic, specifically designed to express in a convenient way distinctive aspects of services, and verifying them over COWS specifications by exploiting an on-the-fly model checker. Over the specification of the automotive scenario, this methodology enables the specification and verification of such functional properties as, e.g., "*once the service Orchestrator is requested, it always provides at least one response about the status of the garage/tow truck ordering and at least one response about the status of the car renting*", "*it will never happen that, after the driver's credit card has been charged and some service ordered, the credit card charge is revoked*", and "*after the garage has been booked, if the tow truck service is not available then the garage is revoked*".

5. Service publication, discovery and negotiation with C◷WS

In the previous sections, we showed that C◷WS is particularly suitable for modelling different and typical aspects of SOC. We now present a dialect of C◷WS (Section 5.1) equipped with mechanisms of *concurrent constraint programming*, which permits modelling the phases of dynamic service publication, discovery and negotiation. This way, we obtain a linguistic formalism capable of modelling all the phases of the life cycle of SOC applications (as we show in Section 5.2).

5.1. A C◷WS*'s dialect for concurrent constraint programming*

We describe here how we can define a dialect of C◷WS exploiting the concurrent constraint programming paradigm to model Service Level Agreement (SLA) achievements. Technically, we take advantage of the fact that C◷WS syntax and operational semantics are parametrically defined with respect to the set of *values*, the syntax of *expressions* that operate on values and, therefore, the definition of the *pattern-matching* function. We follow the approach put forward in cc-pi [43], a language that combines basic features of name-passing calculi with concurrent constraint programming [44]. Specifically, we show that constraints and operations on them can be smoothly incorporated in C◷WS, and propose a disciplined way to model and manipulate multisets of constraints. This way, SLA requirements are expressed as constraints that can be dynamically generated and composed, and that can be used by the involved parties both for service publication and discovery (on the Web), and for the SLA negotiation process. Consistency of the set of constraints resulting from negotiation means that the agreement has been reached.

Intuitively, a *constraint* is a relation among a specified set of variables which gives some information on the set of possible values that these variables may assume. Such information is

usually not complete as a constraint may be satisfied by several assignments of values to the variables. For example, we can employ constraints such as

$$\mathsf{cost} \geqslant 350 \qquad \mathsf{cost} = \mathsf{bw} \cdot 0.05 \qquad \mathsf{z} = 1 \,/\, (\, 1 + |\mathsf{x} - \mathsf{y}|\,)$$

In practice, we do not take a definite standing on which of the many kind of constraints to use. From time to time, the appropriate kind of constraints to work with should be chosen depending on what one intends to model.

Formally a constraint c is represented as a function $c : (V \to D) \to \{\mathbf{true}, \mathbf{false}\}$, where V is the set of constraint variables (that, as explained in the sequel, is included in the set of C⏲WS names), and D is the domain of interpretation of V, i.e. the domain of values that the variables may assume. If we let $\eta : V \to D$ be an assignment of domain elements to variables, then a constraint is a function that, given an assignment η, returns a truth value indicating if the constraint is satisfied by η. For instance, the assignment $\{\mathsf{cost} \mapsto 500\}$ satisfies the first constraint, while $\{\mathsf{cost} \mapsto 500, \mathsf{bw} \mapsto 8000\}$ does not satisfy the second constraint, that is, instead, satisfied by $\{\mathsf{cost} \mapsto 400, \mathsf{bw} \mapsto 8000\}$. An assignment that satisfies a constraint is called a *solution*.

The constraints we have presented are called *crisp* in the literature, because they can only be satisfied or violated. In fact, we can also use more general constraints called *soft constraints* [45]. These constraints, given an assignment for the variables, return an element of an arbitrary constraint semiring (*c-semiring*, [46]), namely a partially ordered set of 'preference' values equipped with two suitable operations for combination ($\times$) and comparison ($+$) of (tuples of) values and constraints. Formally, a c-semiring is an algebraic structure $\langle A, +, \times, 0, 1\rangle$ such that: A is a set and $0, 1 \in A$; $+$ is a binary operation on A that is commutative, associative, idempotent, 0 is its unit element and 1 is its absorbing element; $\times$ is a binary operation on A that is commutative, associative, distributes over $+$, 1 is its unit element and 0 is its absorbing element. Operation $+$ induces a partial order $\leq$ on A defined by $a \leq b$ iff $a + b = b$, which means that a is more constrained than b. The minimal element is thus 0 and the maximal 1. For example, crisp constraints can be understood as soft constraints on the c-semiring $\langle\{\mathbf{true}, \mathbf{false}\}, \vee, \wedge, \mathbf{false}, \mathbf{true}\rangle$.

The C⏲WS dialect we work with in this section specializes expressions to also include *constraints*, ranged over by c, and *constraint multisets*, ranged over by C, and to be formed by using the following operators.

- *Consistency check:* predicate $isCons(C)$ takes a constraint multiset C and holds true if C is consistent. Formally, $isCons(\{c_1, \ldots, c_n\})$ holds true if there exists an assignment η such that $c_1\eta \wedge \ldots \wedge c_n\eta \neq \mathbf{false}$, i.e. if the combination of all constraints has at least a solution[4]. The predicate $isCons(\_)$ is defined for crisp constraints. However, we can generalize its definition to soft constraints by requiring that it is satisfied if there exists an assignment η such that $c_1\eta \times \ldots \times c_n\eta \neq 0$.

- *Entailment check:* predicate $C \vdash c$ takes a constraint multiset C and a constraint c and holds true if c is entailed by C. Formally, $\{c_1, \ldots, c_n\} \vdash c$ holds true if for all assignments η it holds that $c_1\eta \wedge \ldots \wedge c_n\eta \leq_B c\eta$, where $\leq_B$ is the partial ordering over booleans (i.e. $b_1 \leq_B b_2$ iff $b_1 \vee b_2 = b_2$). Also this predicate can be generalized to soft constraints by requiring that $\{c_1, \ldots, c_n\} \vdash c$ holds true if for all assignments η it holds that $c_1\eta \times \ldots \times c_n\eta \leq c\eta$.

[4]We do not consider here the well-studied problem of solving a constraint system. Among the many techniques exploited to this aim, we mention dynamic programming [47, 48] and branch and bound search [49].

$$\frac{isCons(C \uplus \{c\})}{\mathcal{M}(\langle c, x\rangle, C) = \{x \mapsto C\}} \qquad \frac{C \vdash c}{\mathcal{M}(\langle c^{\vdash}, x\rangle, C) = \{x \mapsto C\}}$$

Table 13: Pattern-matching function (additional rules)

- *Retraction:* operation $C - c$ takes a constraint multiset C and a constraint c and returns the multiset $C \backslash \{c\}$ if $c \in C$, otherwise returns C.
- *Multiset union:* binary operator $\uplus$ is the standard union operator between multisets.

Since constraints and constraint multisets are expressions, they need to be evaluated. The (expression) evaluation function $[\![\_]\!]$ acts on constraints and constraint multisets as the identity, except for constraints containing C⏲WS variables, for which the function is undefined. Therefore, evaluated constraints and constraint multisets are values that can be communicated by means of synchronization of invoke and receive activities and can replace variables by means of application of substitutions to terms.

To efficiently implement the primitives of the concurrent constraint programming paradigm, we tailor the rules in Table 3 (Section 3.1) defining the pattern-matching function $\mathcal{M}(\_, \_)$ to deal with constraints and operations on them, by adding the rules in Table 13. We assume here that tuples can be arbitrarily nested. The original matching rules (reported in Table 3) are still valid and state that variables match any value (thus, e.g., $\mathcal{M}(x, C) = \{x \mapsto C\}$), two values match only if they are identical, and two tuples match if they have the same number of fields and corresponding fields do match. The new rules allow a two-field tuple to match a single value in two specific cases: a tuple $\langle c, x\rangle$ and a multiset of constraints C do match if $C \uplus \{c\}$ is consistent, while a tuple $\langle c^{\vdash}, x\rangle$ and a multiset of constraints C do match if c is entailed by C; in both cases, the substitution $\{x \mapsto C\}$ is returned. Notably, by applying the operator $\_^{\vdash}$ to a constraint one can require an entailment check instead of a consistency check.

The concurrent constraint computing model is based on a shared store of constraints that provides partial information about possible values that variables can assume. In C⏲WS the store of constraints is represented by the following service:

$$store_C \triangleq [\mathtt{n}]\,(\,\mathtt{n}!\langle C\rangle \mid *\,[x]\,\mathtt{n}?\langle x\rangle.\,(\,p_s \cdot o_{get}!\langle x\rangle \mid [y]\,p_s \cdot o_{set}?\langle y\rangle.\,\mathtt{n}!\langle y\rangle\,)\,)$$

where p_s is a distinguished partner, o_{get} and o_{set} are distinguished operations. Other services can interact with the store service in mutual exclusion, by acquiring the lock (and, at the same time, the stored value) with a receive along $p_s \cdot o_{get}$ and by releasing the lock (providing the new stored value) with an invoke along $p_s \cdot o_{set}$. Notably, local stores of constraints can be simply modelled by restricting the scope of the partner name p_s.

The store is composed in parallel with the other services, which can act on it by performing operations for adding/removing constraints to/from the store (tell and retract, respectively), and for checking entailment/consistency of a constraint by/with the store (ask and check, respectively). These four operations can be rendered in C⏲WS as follows:

$$\langle\!\langle \mathsf{tell}\ c.s \rangle\!\rangle = [\mathtt{n}]\,(\,\mathtt{n}!\langle c\rangle \mid [y]\,\mathtt{n}?\langle y\rangle.\,[x]\,p_s \cdot o_{get}?\langle\langle y, x\rangle\rangle.\,(\,\{\!|\,p_s \cdot o_{set}!\langle x \uplus \{y\}\rangle\,|\!\}\mid \langle\!\langle s\rangle\!\rangle\,)\,)$$

$$\langle\!\langle \mathsf{ask}\ c.s \rangle\!\rangle = [\mathtt{n}]\,(\,\mathtt{n}!\langle c^{\vdash}\rangle \mid [y]\,\mathtt{n}?\langle y\rangle.\,[x]\,p_s \cdot o_{get}?\langle\langle y, x\rangle\rangle.(\,\{\!|\,p_s \cdot o_{set}!\langle x\rangle\,|\!\}\mid \langle\!\langle s\rangle\!\rangle\,)\,)$$

$$\langle\!\langle \mathsf{check}\ c.s \rangle\!\rangle = [\mathtt{n}]\,(\,\mathtt{n}!\langle c\rangle \mid [y]\,\mathtt{n}?\langle y\rangle.\,[x]\,p_s \cdot o_{get}?\langle\langle y, x\rangle\rangle.\,(\,\{\!|\,p_s \cdot o_{set}!\langle x\rangle\,|\!\}\mid \langle\!\langle s\rangle\!\rangle\,)\,)$$

$$\langle\!\langle \mathsf{retract}\ c.s \rangle\!\rangle = [\mathtt{n}]\,(\,\mathtt{n}!\langle c\rangle \mid [y]\,\mathtt{n}?\langle y\rangle.\,[x]\,p_s \cdot o_{get}?\langle x\rangle.\,(\,\{\!|\,p_s \cdot o_{set}!\langle x - y\rangle\,|\!\}\mid \langle\!\langle s\rangle\!\rangle\,)\,)$$

where $\mathtt{n}$ is fresh. Essentially, each operation is a term that first takes the store of constraints (thus acquiring the lock so that other services cannot concurrently interact with the store) and then returns the (possibly) modified store (thus releasing the lock). Since the invoke activities $\mathtt{n}!\langle c\rangle$ and $\mathtt{n}!\langle c^{\vdash}\rangle$ can be performed only if $[\![c]\!]$ is defined, i.e. if c does not contain C⏲WS variables, the store can only contain evaluated constraints. Availability of the store is guaranteed by the fact that, once the store and the lock have been acquired, the activities reintroducing the store and releasing the lock are protected from the effect of kill activities. This disciplined use of the store permits to preserve its consistency. Notably, the matching rules in Table 13 are essential for faithfully modelling the semantics of the original operations. Also notice that, in the definition of tell, the expression $x \uplus \{y\}$ is well-defined, since the variable x is replaced by a multiset of constraints while y by a single constraint.

While tell and ask are the classical concurrent constraint programming primitives, operations check and retract are borrowed from [43]. In particular, operation retract is debatable since its adoption prevents the store of constraints to be 'monotonically' refined. In fact, in concurrent constraint programming a computation step does not change the value of a variable, but may rule out certain values that were previously possible; therefore, the set of possible values for the variable is contained in the set of possible values at any prior step. This monotonic evolution of the store during computations permits to define the result of a computation as the least upper bound of all the stores occurring along the computation and provides concurrent constraint languages with a simple denotational semantics in which programs are identified to closure operators on the semi-lattice of constraints [50]. Therefore, if one wants to exploit some of the properties of concurrent constraint programming that require monotonicity, he must consider the fragment of C⏲WS without retract. On the other hand, in the context of dynamic service discovery and negotiation, the use of operation retract enables modelling many frequent situations where it is necessary to remove a constraint from the store for, e.g., weakening a request.

To avoid interference between communication and operations on the store, we do not allow constraints in the store to contain variables, thus they cannot change due to application of substitutions generated by communication. Indeed, suppose constraints in the store may contain variables and consider the following example:

$$[x]\,(\,\mathit{store}_{\emptyset} \mid \mathsf{tell}(x \leq 5).\,(\mathtt{n}!\langle 6\rangle \mid \mathtt{n}?\langle x\rangle)\,)$$

After action tell has added $x \leq 5$ to the store, communication along the endpoint $\mathtt{n}$ can modify the constraint in $6 \leq 5$. This way, the communication can make the store inconsistent. This means that the write-once variables of C⏲WS are not suitable for modelling constraint variables.

Therefore, as we stated before, we do not allow constraints in the store to contain variables. Instead, they can use specific names, that we call *constraint variables* and, for the sake of presentation, write as $\mathsf{x}, \mathsf{y}, \ldots$ (i.e. in the sans serif style). Indeed, names are not affected by expression evaluation (i.e. $[\![\mathsf{x}]\!] = \mathsf{x}$) and by substitution application (i.e. $\mathsf{x} \cdot \sigma = \mathsf{x}$). Moreover, names can be delimited, thus allowing us to model *local constraints*. Notice however that constraints occurring as arguments of operations may contain variables so that we can specify constraints that will be dynamically determined. E.g., we can write tell $(\mathsf{cost} \geqslant x_{min\_cost}).s$; since $[\![\mathsf{cost} \geqslant x_{min\_cost}]\!]$ is undefined, this operation is blocked until variable x_{min_cost} is substituted by a value.

Besides ask, tell, retract and check, inter-service communication can be used to implement many protocols allowing two parties to generate new constraints. For instance, in [43], service synchronization works like two global ask and tell constructs: as a result of the synchronization between the output $\bar{x}\langle y\rangle$ and the input $x\langle y'\rangle$ the new constraint $y = y'$ is added to the store. Therefore, synchronization allows local constraints (i.e. constraints with restricted names) to interact,

thus establishing an SLA between the two parties, and (possibly) to become globally available. Differently, C⏲WS does not allow communication to directly generate new constraints: e.g., an invoke $p \cdot o!\langle \mathsf{x} \rangle$ and a receive $p \cdot o?\langle \mathsf{y} \rangle$ cannot synchronize, because $\mathcal{M}(\mathsf{y}, \mathsf{x})$ does not hold. Thus, to create constraints of the form $\mathsf{x} = \mathsf{y}$, where each of x and y is initially local to only one party, we can use the standard C⏲WS communication mechanism together with operation tell. For example, the following term

$$store_C \mid p \cdot o!\langle \mathsf{x} \rangle \mid [z]\, p \cdot o?\langle z \rangle.\, \mathsf{tell}\,(z = \mathsf{y}).\, s \tag{2}$$

for z fresh in s, adds to the store the constraint $\mathsf{x} = \mathsf{y}$, if it is consistent with C. This protocol is simple and divergence-free, but it may introduce deadlocked states in the terms, because the communication along endpoint $p \cdot o$ takes place before the consistency check (performed by operation tell). For other protocols that permit establishing new constraints by overtaking this problem, we refer the interested reader to [23]. Anyway, since the problem mentioned above does not occur in the specification in Section 5.2, in the sequel we implicitly rely on protocol (2).

5.2. Automatic discovery and negotiation in the automotive case study

We show here how our framework can be used to integrate publication, discovery and negotiation into the automotive case study presented in Section 2.2 and specified in COWS in Section 4.

Initially, each on road service (e.g. garages, tow trucks, ...) has to publish its service description on a service registry. For example, assume that a garage service description consists of: a string identifying the kind of provided service, the provider's partner name, and a constraint that defines the garage location. By assuming that the registry provides the operation o_{pub} through the partner name p_{reg}, a garage service can request the publication of its description as follows:

$$p_{reg} \cdot o_{pub}!\langle \text{“}garage\text{”}, p_{garage}, \mathsf{gps} = (4348.1143N, 1114.7206E) \rangle$$

where gps is a constraint variable.

The service registry is defined as

$$[o_{DB}]\,(\,*\,[x_{type}, x_p, x_c]\, p_{reg} \cdot o_{pub}?\langle x_{type}, x_p, x_c \rangle.p_{reg} \cdot o_{DB}!\langle x_{type}, x_p, x_c \rangle \;\mid\; R^{search}\,)$$

For each publication request received along the endpoint $p_{reg} \cdot o_{pub}$ from a provider service, the registry service outputs a service description along the private endpoint $p_{reg} \cdot o_{DB}$. The parallel composition of all these outputs represents the database of the registry. The subservice R^{search}, serving the searching requests, is defined as

$$\begin{aligned} R^{search} \;\triangleq\; & *\,[x_{type}, x_{client}, x_c, o_{addToList}, o_{askList}] \\ & p_{reg} \cdot o_{search}?\langle x_{type}, x_{client}, x_c \rangle.\, [p_s]\,(\, store_\emptyset \mid \mathsf{tell}\, x_c.\, R' \mid List\,) \end{aligned}$$

$$\begin{aligned} R' \;\triangleq\; & [k]\,(\,*\,[x_p, x_{const}]\, p_{reg} \cdot o_{DB}?\langle x_{type}, x_p, x_{const} \rangle. \\ & \qquad (\, \{\!|\, p_{reg} \cdot o_{pub}!\langle x_{type}, x_p, x_{const} \rangle \,|\!\}\, \mid \mathsf{check}\, x_{const}.\, p_{reg} \cdot o_{addToList}!\langle x_p \rangle\,) \\ & \quad \mid \oplus_\delta.\,(\,\mathbf{kill}(k) \mid \{\!|\, [x_{list}]\, p_{reg} \cdot o_{askList}?\langle x_{list} \rangle.\, x_{client} \cdot o_{resp}!\langle x_{list} \rangle \,|\!\}\,)\,) \end{aligned}$$

When a searching request is received along $p_{reg} \cdot o_{search}$, the registry service initializes a new local store (delimitation $[p_s]$ makes $store_\emptyset$ inaccessible outside of service R^{search}) by adding the constraint within the query message. Then, it cyclically reads a description (whose first field is the string specified by the client) from the internal database, checks if the provider constraints are consistent with the store and, in case of success, adds the provider's partner name to a list (by

exploiting an internal service *List*, that provides operations $o_{addToList}$ and $o_{askList}$). After δ time units from the initialization of the local store, the loop is terminated by executing a kill activity and the current list of providers for service type x_{type} is sent to the client. Notably, reading a description in the database, in this case, consists of an input along $p_{reg} \cdot o_{DB}$ followed by an output along $p_{reg} \cdot o_{pub}$; this way we are guaranteed that, after being consumed, the description is correctly added to the database. It is worth noticing that, for the sake of simplicity, service descriptions are non-deterministically retrieved, thus the same provider can occur in the returned list many times. This behaviour could be avoided by refining the specification, e.g. by tagging each service description with an index (stored in an additional field) that is then exploited to read the descriptions in an ordered way.

After the user's car breaks down and *Orchestrator* is triggered, the service *Discovery* of the in-vehicle platform will receive from *Orchestrator* a request containing the GPS data of the car, that it stores in x_{loc}, and a string identifying the kind of the required services (see the specification in Section 4). By exploiting the latter information, it will know that it has to search a garage, a tow truck and a rental car service. For example, the component taking care of discovering a garage service can be

$$p_{reg} \cdot o_{search}!\langle \text{“}garage\text{”}, p_{car}, dist(x_{loc}, \mathsf{gps}) < 20 \rangle \mid [x_{garageList}]\, p_{car} \cdot o_{resp}?\langle x_{garageList} \rangle$$

where the constraint $dist(x_{loc}, \mathsf{gps}) < 20$ means that the required garages must be less than 20 km far from the stranded car's actual location.

Once the discovery phase terminates and *Reasoner* communicates the best garage service to *Orchestrator*, the latter and the selected garage engage in a negotiation phase in order to sign an SLA. First, *Orchestrator* invokes the operation $o_{orderGar}$ provided by the selected garage (see the term *OrderGarageAndTowTruck* in Section 4); then, it starts the negotiation by performing an operation tell that adds *Orchestrator*'s local constraints (i.e. constraints with restricted constraint variables) to the shared global store; finally, it synchronizes with the garage service, by invoking o_{sync}, for sharing its local constraints with it.

$$\begin{array}{l}
[\mathsf{cost}, \mathsf{duration}] \\
\mathsf{tell}\,((\mathsf{cost} < 1500 \wedge \mathsf{duration} < 48) \vee (\mathsf{cost} < 800 \wedge \mathsf{duration} \geqslant 48)). \\
(\, x_{garage} \cdot o_{sync}!\langle \mathsf{cost}, \mathsf{duration} \rangle \\
\quad \mid\; p_{car} \cdot o_{garageOK}?\langle x_{gps}, x_{garageInfo} \rangle . \cdots + \; p_{car} \cdot o_{garageFail}?\langle\rangle . \cdots)
\end{array}$$

In our example, the constraints state that for a repair in less than two days (i.e. 48 hours) the driver is disposed to spend up to 1500 Euros, otherwise he is ready to spend less than 800 Euros.

After the synchronization with *Orchestrator*, the selected garage service tries to impose its first-rate constraint $c = ((\mathsf{cost'} > 2000 \wedge 6 < \mathsf{duration'} < 24) \vee (\mathsf{cost'} > 1500 \wedge \mathsf{duration'} \geqslant 24))$ and, if it fails to reach an agreement within δ' time units, weakens the requirements and retries with the constraint $c' = ((\mathsf{cost'} > 1700 \wedge 6 < \mathsf{duration'} < 24) \vee (\mathsf{cost'} > 1200 \wedge \mathsf{duration'} \geqslant 24))$. Both constraints are specifically generated by the garage service for the occurred engine failure, by exploiting the transmitted diagnostic data. After δ'' time units, if also the second attempt fails, it gives up the negotiation. This negotiation task is modelled as follows:

$$\begin{array}{l}
[x_{cost}, x_{duration}, \mathsf{cost'}, \mathsf{duration'}] \\
p_{garage} \cdot o_{sync}?\langle x_{cost}, x_{duration} \rangle . \mathsf{tell}\,(x_{cost} = \mathsf{cost'} \wedge x_{duration} = \mathsf{duration'}). \\
\quad (\mathsf{tell}\; c . x_{cust} \cdot o_{garageOK}!\langle garageGPS, garageInfo \rangle \\
\quad\quad + \; \textcircled{\scriptsize t}_{\delta'} . (\mathsf{tell}\; c' . x_{cust} \cdot o_{garageOK}!\langle garageGPS, garageInfo \rangle \\
\quad\quad\quad + \; \textcircled{\scriptsize t}_{\delta''} . x_{cust} \cdot o_{garageFail}!\langle\rangle))
\end{array}$$

Notably, operations tell could not be directly used as guards for the choice operator. Thus, a term like $\mathsf{tell}\ c.\ s\ +\ \oplus_e.\ s'$ should be considered as an abbreviation for

$$[p, q, o]\,(\,\mathsf{check}\ c.\,(p \bullet o!\langle\rangle \mid q \bullet o?\langle\rangle.\,\mathsf{tell}\ c.\ s)\ \mid\ \oplus_e.\ s'\ +\ p \bullet o?\langle\rangle.\,q \bullet o!\langle\rangle\,)$$

Intuitively, if the constraint c is consistent with the store, the timer can be stopped (i.e. communication along $p \bullet o$ makes a choice and removes the wait activity); afterward, the constraint can be added to the store, provided that other interactions that took place in the meantime do not lead to inconsistency (which, anyway, is not the case in our scenario). Otherwise, if the timeout expires, the constraint cannot be added to the store.

6. Related work

We have already pointed out, mainly in Section 3.1.1, main relationships of C⏲WS with other process calculi. By summing up, C⏲WS borrows, e.g., global scoping and non-binding input from update calculus [33] and fusion calculus [34], distinction between variables and values from value-passing CCS [51], Applied π-calculus [52] and Distributed π-calculus [53], pattern-matching from KLAIM [54], prioritised activities from variants of CCS with priority [55, 56, 57], and forced termination and protection from StAC [58].

Many works put forward enrichments of some well-known process calculus with constructs inspired by those of WS-BPEL. Most of them deal with issues of web transactions such as interruptible processes, failure handlers and time. This is, for example, the case of [4, 5, 59, 60] that present timed and untimed extensions of the π-calculus, called $\mathtt{web}\pi$ and $\mathtt{web}\pi_\infty$, tailored to study a simplified version of the scope construct of WS-BPEL. Other proposals on the formalization of flow compensation are [61, 62] that give a more compact and closer description of the Sagas mechanism [32] for dealing with long running transactions, while some other works [4, 60] have concentrated on modelling web transactions and on studying their properties in programming languages based on the π-calculus. In contrast, C⏲WS aims at dealing at once with many different and typical aspects of SOC, thus modelling an expressive subset of WS-BPEL rather than only focussing on a few specific constructs.

The formalism closest to C⏲WS is ws-calculus [7], which has been introduced to formalize the semantics of WS-BPEL. C⏲WS represents a more foundational formalism than ws-calculus in that it does not rely on explicit notions of location and state, it is more manageable (e.g. has a simpler operational semantics) and has, at least, equally expressive power (as the encoding of ws-calculus in COWS [21, Section 5.1.3] shows). Moreover, C⏲WS is equipped with timed constructs while ws-calculus is not.

For modelling time and timeouts, we have drawn again our inspiration from the rich literature on timed process calculi (see, e.g., [63, 64] for a survey). In C⏲WS, basic actions are *duration-less*, i.e. instantaneous, and the passing of time is modelled by using explicit actions, like in TCCS [65]. Moreover, actions execution is *lazy*, i.e. can be delayed arbitrary long in favour of passing of time, like in lTCCS [66].

The correlation mechanism was first exploited in [67], that, however, only considers interaction among different instances of a single business process. Instead, to connect the interaction protocols of clients and of the respective service instances, a large strand of work (among which we mention [68, 69, 70, 14, 71, 72, 73]) relies on the explicit modelling of interaction sessions and their dynamic creation (that exploits the mechanism of private names of π-calculus). Sessions are not explicitly modelled in C⏲WS: they can be identified by tracing all those exchanged

messages that are correlated each other through their same contents (as in [12]). We believe that the mechanism based on correlation sets, that exploits business data and communication protocol headers to correlate different interactions, is more robust and fits the loosely coupled world of Web Services better than that based on explicit session references. It is not a case that also WS-BPEL uses correlation sets.

Another body of work has been devoted to study mechanisms for comparing global descriptions (i.e. choreographies) and local descriptions (i.e. orchestrations) of a same system. Means to check conformance of these different views have been defined in [74, 10] and, by relying on session types, in [75]. C⏲WS, instead, only considers service orchestration and focuses on modelling the dynamic behaviour of services without the limitations possibly introduced by a layer of choreography.

Regarding QoS requirement specifications and SLA achievements, most of the proposals in the literature result from the extension of some well-known process calculus with constructs to describe QoS requirements. This is, for example, the case of cc-pi [43], a calculus that generalises the explicit name 'fusions' of the pi-F calculus [76] to 'named constraints', namely constraints defined on enriched c-semiring structures. cc-pi, as well as C⏲WS, combines basic features of name-passing calculi with those of concurrent constraint programming, firstly introduced in [44], and its *soft* variant [45]. However, rather than on fusions of names, C⏲WS relies on substitutions of variables with values and can thus express also soft constraints by exploiting the simpler notion of c-semiring. Moreover, C⏲WS permits defining local stores of constraints while cc-pi processes necessarily share one global store. [77] introduces another formalism, namely *nmsccp*, for soft concurrent constraint programming that permits the nonmonotonic evolution of the store of constraints. Besides the *retract* operation also used in C⏲WS, *nmsccp* provides an *update* operation, to relax some constraints of the store dealing with certain variables while adding a constraint, and a *nask* operation, to test if a constraint is not entailed by the store. If and how the latter two operations can be rendered in the variant of C⏲WS presented in Section 5.1 is left for future investigation. A similar approach to SLAs negotiation is proposed in [78], although it is based on fuzzy sets instead of constraints and relies on three different languages, one for client requests, one for provider descriptions and one for contracts creation and revocation. SLA compliance has been also the focus of $\mathcal{K}$oS [79] and Kaos [80], two calculi designed for modelling network aware applications with located services and mobility. In both cases, QoS parameters are associated to connections and nodes of nets, and operations have a QoS value; the operational semantics ensures that systems evolve according to SLAs. All the mentioned proposals aim at specifying and concluding SLAs, while C⏲WS permits also modelling other service-oriented aspects, such as e.g. service instances and interactions, fault and compensation handling, and dynamic service publication, discovery and orchestration. Integrations of the concurrent constraint paradigm with process calculi have also been used to define foundational formalisms for computer music languages. This is the case of the π^+-calculus [81], an extension of the (polyadic) π-calculus with agents that can interact with a store of constraints by performing 'tell' and 'ask' actions. Differently from C⏲WS, the store of constraints is not a term of the calculus, indeed the operational semantics of π^+-calculus is defined over configurations consisting of pairs of an agent and a store, and local stores are not supported.

Some other works, differently from C⏲WS, exploit static service discovery mechanisms. For example, [82] introduces an extension of the λ-calculus with primitive constructs for call-by-contract invocation for which a completely static approach for regulating secure service composition has been devised. In particular, an automatic machinery, based on a type system and a model-checking technique, has been defined to construct a viable plan for the execution of

services belonging to a given orchestration. Non-functional aspects are included and enforced by means of a runtime security monitor. In [83], user's requests and compositions of web services are statically modelled via constraints. Finally, the calculi of contracts of [84] represent a more abstract approach for statically checking compliance between the client requirements and the service functionalities. A contract defines the possible flows of interactions of a service, but does not take into account non-functional properties and, thus, cannot be used for specifying and negotiating SLAs.

Up to here, we have discussed the relationship between C⏲WS and other formal languages for specifying SOC applications and their main features. We conclude now this section with a discussion about the relationship between C⏲WS and WS-BPEL, namely the SOC technology that more than any other has influenced C⏲WS's design. On the one hand, C⏲WS distills out of WS-BPEL those features that are, in our opinion, absolutely necessary to formally define the basic elements and mechanisms underlying the SOC paradigm. Indeed, C⏲WS directly borrows from WS-BPEL the notions of partner and operation, the communication primitives (for invoking an operation offered by a service and waiting for an invocation to arrive), the related mechanism for message correlation, and the timed activity (for delaying the execution for some amount of time). C⏲WS also retains the WS-BPEL's constructs *flow*, to execute activities in parallel, and *pick*, to execute activities selectively, which corresponds to the C⏲WS's parallel composition and choice operators, respectively. On the other hand, while the set of WS-BPEL constructs is not intended to be a minimal one, C⏲WS aims at being a foundational model and, thus, at keeping its semantics rigorous but still manageable and not strongly tight to web services' current technology. Therefore, some WS-BPEL constructs do not have a precise counterpart in C⏲WS, rather they are expressed in terms of more primitive operators. For example, fault and compensation handlers are rendered in C⏲WS, as shown in Section 4, by means of the primitives dealing with termination, i.e. kill, protection and delimitation. Indeed, when a fault occurs during the execution of a given activity, the kill primitive permits to immediately interrupt the currently running activities under the scope of the fault (identified by the delimitation operator), while the protection operator is used to avoid involving any fault/compensation handling behaviour in the forced termination. Similarly, service instantiation is rendered in C⏲WS through the replication operator, while shared states among service instances through variable delimitation. Finally, as shown in [8], the standard imperative constructs (assignment, while, if-then-else, etc.) can be easily expressed in C⏲WS, as well as the remaining features of WS-BPEL, like e.g. the synchronisation dependencies within flow activities.

7. Concluding remarks and future work

This paper provides a formal account of the SOC paradigm and related technologies. The introduction of C⏲WS as a formalism specifically devised for modelling service-oriented applications is indeed an important step towards the comprehension of the mechanisms underlying the SOC paradigm. On the one hand, since the design of the calculus has been influenced by the principles underlying WS-BPEL, C⏲WS permits modelling in a natural way different and typical aspects of (web) services technologies, such as multiple start activities, receive conflicts, timed constructs, delivering of correlated messages, service instances and interactions among them. On the other hand, C⏲WS is a foundational formalism, not specifically tight to web services' current technology, and borrows many constructs from well-known process calculi, as e.g. π-calculus, update calculus, StAC$_i$, and Lπ. We have illustrated syntax, operational semantics and pragmatics of the calculus by means of a large case study from the automotive domain and a number of

more specific examples drawn from it. We have also introduced a dialect of the language that turned out to be capable of modelling all the phases of the life cycle of service-oriented applications, such as publication, discovery, negotiation, orchestration, deployment, reconfiguration and execution.

As a further evidence of the quality of the design of our formalism, since its definition a number of methods and tools have been devised to analyse COWS specifications. We want to mention here the stochastic extension and the BPMN-based notation defined in [9, 16] to enable quantitative reasoning on service behaviours, the type system introduced in [17] to check confidentiality properties, the logic and model checker presented in [18] and exploited in [85] to express and check functional properties of services, the bisimulation-based observational semantics defined in [19] to check interchangeability of services and conformance against service specifications, and the symbolic characterisation of the operational semantics of COWS presented in [20] to avoid infinite representations of COWS terms due to the value-passing nature of communication. An overview of most of the tools mentioned above and the classes of properties that can be analysed by using them can be found in [21]. For the time being, the above analysis tools are applicable to the time-free fragment COWS. We do not envisage any major issue in tailoring them to C$\clock$WS, but leave this extension as a future work.

To complete our programme to lay rigorous methodological foundations and provide supporting tools for specification, validation and development of SOC applications, we plan in the near future to develop a prototype implementation of C$\clock$WS possibly enriched with standard linguistic constructs supporting real application development. This would permit to assess C$\clock$WS practical usability and to shorten the gap between theory and practice. The implementation of a language based on a process calculus typically consists of a run-time system (a sort of abstract machine) implemented in a high level language like Java, and of a compiler that, given a program written in the programming language based on the calculus, produces code that uses the run-time system above. In the development of our language, we intend to follow a similar approach. In this regard, the major issue we envisage is the integration of our framework with the current standard technologies supporting web services interaction, such as WSDL and SOAP. In particular, the code generated from C$\clock$WS services should be able to invoke operations provided by available web services and, in its turn, to expose its functionalities as a standard web service. Some implementations of service-oriented calculi that could serve as a guide for our work are as follows: JCaSPiS [86], a Java implementation of the calculus CaSPiS [14] based on a generic framework that provides recurrent mechanisms for network applications; B*lite*C [87], a Java tool that accepts as an input a specification written in B*lite* [40], a formal orchestration language inspired to but simpler than WS-BPEL, and returns the corresponding WS-BPEL program together with the associated WSDL and deployment descriptor files; JOLIE [88], an interpreter written in Java for a programming language designed for web service orchestration and based on SOCK [12]; JSCL [89], a coordination middleware for services based on the event notification paradigm of Signal Calculus [90]; and `PiDuce` [38], a distributed run-time environment devised for experimenting web services technologies that implements a variant of asynchronous π-calculus extended with native XML values, datatypes and patterns.

Acknowledgments. We thank the anonymous reviewers for their useful comments. We also thank Alessandro Lapadula for his fundamental contribution to the definition of C$\clock$WS.

References

[1] L. Meredith, S. Bjorg, Contracts and types, Communications of the ACM 46 (2003) 41–47.

[2] F. van Breugel, M. Koshkina, Models and verification of BPEL, Technical Report, Department of Computer Science and Engineering, York University, 2006. Available at `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf`.

[3] L. Bocchi, C. Laneve, G. Zavattaro, A Calculus for Long-Running Transactions, in: FMOODS, volume 2884 of *LNCS*, Springer, 2003, pp. 124–138.

[4] C. Laneve, G. Zavattaro, Foundations of Web Transactions, in: FoSSaCS, volume 3441 of *LNCS*, Springer, 2005, pp. 282–298.

[5] C. Laneve, G. Zavattaro, web-pi at Work, in: TGC, volume 3705 of *LNCS*, Springer, 2005, pp. 182–194.

[6] M. Butler, C. Hoare, C. Ferreira, A Trace Semantics for Long-Running Transactions, in: 25 Years Communicating Sequential Processes, volume 3525 of *LNCS*, Springer, 2005, pp. 133–150.

[7] A. Lapadula, R. Pugliese, F. Tiezzi, A WSDL-based type system for WS-BPEL, in: COORDINATION, volume 4038 of *LNCS*, Springer, 2006, pp. 145–163.

[8] A. Lapadula, R. Pugliese, F. Tiezzi, A Calculus for Orchestration of Web Services, in: ESOP, volume 4421 of *LNCS*, Springer, 2007, pp. 33–47.

[9] D. Prandi, P. Quaglia, Stochastic COWS, in: ICSOC, volume 4749 of *LNCS*, Springer, 2007, pp. 245–256.

[10] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, G. Zavattaro, Choreography and orchestration conformance for system design, in: COORDINATION, volume 4038 of *LNCS*, Springer, 2006, pp. 63–81.

[11] C. Laneve, L. Padovani, Smooth Orchestrators, in: FoSSaCS, volume 3921 of *LNCS*, Springer, 2006, pp. 32–46.

[12] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, G. Zavattaro, SOCK: A Calculus for Service Oriented Computing, in: ICSOC, volume 4294 of *LNCS*, Springer, 2006, pp. 327–338.

[13] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, G. Zavattaro, SCC: a Service Centered Calculus, in: WS-FM, volume 4184 of *LNCS*, Springer, 2006, pp. 38–57.

[14] M. Boreale, R. Bruni, R. De Nicola, M. Loreti, Sessions and Pipelines for Structured Service Programming, in: FMOODS, volume 5051 of *LNCS*, Springer, 2008, pp. 19–38.

[15] OASIS WSBPEL TC, Web Services Business Process Execution Language Version 2.0, Technical Report, OASIS, 2007. Available at `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[16] D. Prandi, P. Quaglia, N. Zannone, Formal analysis of BPMN via a translation into COWS, in: COORDINATION, volume 5052 of *LNCS*, Springer, 2008, pp. 249–263.

[17] A. Lapadula, R. Pugliese, F. Tiezzi, Regulating data exchange in service oriented applications, in: FSEN, volume 4767 of *LNCS*, Springer, 2007, pp. 223–239.

[18] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, F. Tiezzi, A Logical Verification Methodology for Service-Oriented Computing, ACM Transactions on Software Engineering and Methodology (2011). To appear.

[19] R. Pugliese, F. Tiezzi, N. Yoshida, On observing dynamic prioritised actions in SOC, in: ICALP, volume 5556 of *LNCS*, Springer, 2009, p. 558570.

[20] R. Pugliese, F. Tiezzi, N. Yoshida, A Symbolic Semantics for a Calculus for Service-Oriented Computing, in: PLACES, volume 241 of *ENTCS*, Elsevier, 2009, pp. 135–164.

[21] F. Tiezzi, Specification and Analysis of Service-Oriented Applications, PhD Thesis in Computer Science, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, 2009. Available at `http://rap.dsi.unifi.it/cows`.

[22] A. Lapadula, R. Pugliese, F. Tiezzi, C⏲WS: A timed service-oriented calculus, in: ICTAC, volume 4711 of *LNCS*, Springer, 2007, pp. 275–290.

[23] A. Lapadula, R. Pugliese, F. Tiezzi, Service discovery and negotiation with COWS, in: WWV, volume 200(3) of *ENTCS*, Elsevier, 2008, pp. 133–154.

[24] A. Brown, S. Johnston, K. Kelly, Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications, Technical Report, Rational Software Corporation, 2003.

[25] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, D. Winer, Simple Object Access Protocol (SOAP) 1.2, W3C recommendation, 2003. Available at `http://www.w3.org/TR/SOAP/`.

[26] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web Services Description Language (WSDL) 1.1, Technical Report, W3C, 2001. Available at `http://www.w3.org/TR/wsdl/`.

[27] UDDI Spec TC, UDDI Specification Technical Committee Draft, Technical Report, OASIS, 2004. Available at `http://uddi.org/pubs/uddi_v3.htm/`.

[28] C. Peltz, Web Services Orchestration and Choreography, Computer 36 (2003) 46–52.

[29] N. Koch, Automotive Case Study: UML Specification of On Road Assistance Scenario, Technical Report 1, FAST GmbH, 2007. Available at `http://rap.dsi.unifi.it/sensoriasite/files/FAST_report_1_2007_ACS_UML.pdf`.

[30] SENSORIA, Software engineering for service-oriented overlay computers, 2005-2010. Web site: http://www.sensoria-ist.eu/.
[31] P. Mayer, A. Schroeder, N. Koch, A Model-Driven Approach to Service Orchestration, in: SCC, volume 2, IEEE Computer Society Press, 2008, pp. 533–536.
[32] H. Garcia-Molina, K. Salem, Sagas, in: SIGMOD, ACM Press, 1987, pp. 249–259.
[33] J. Parrow, B. Victor, The update calculus, in: AMAST, volume 1349 of *LNCS*, Springer, 1997, pp. 409–423.
[34] J. Parrow, B. Victor, The fusion calculus: Expressiveness and symmetry in mobile processes, in: LICS, IEEE Computer Society Press, 1998, pp. 176–185.
[35] M. Carbone, S. Maffeis, On the expressive power of polyadic synchronisation in π-calculus, Nordic Journal of Computing 10 (2003) 70–98.
[36] M. Merro, D. Sangiorgi, On asynchrony in name-passing calculi, Mathematical Structures in Computer Science 14 (2004) 715–767.
[37] P. Gardner, C. Laneve, L. Wischik, Linear Forwarders, in: CONCUR, volume 2761 of *LNCS*, Springer, 2003, pp. 408–422.
[38] S. Carpineti, C. Laneve, L. Padovani, PiDuce - a project for experimenting Web services technologies, Science of Computer Programming 74 (2009) 777–811.
[39] R. Amadio, I. Castellani, D. Sangiorgi, On Bisimulations for the Asynchronous pi-Calculus, Theoretical Computer Science 195 (1998) 291–324.
[40] A. Lapadula, R. Pugliese, F. Tiezzi, A formal account of WS-BPEL, in: COORDINATION, volume 5052 of *LNCS*, Springer, 2008, pp. 199–215.
[41] R. van Glabbeek, On Specifying Timeouts, in: APC, volume 162 of *ENTCS*, Elsevier, 2006, pp. 173–175.
[42] R. Pugliese, F. Tiezzi, A COWS Specification of an Automotive Case Study, Technical Report, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, 2011. Available at http://rap.dsi.unifi.it/cows/automotiveCSinCOWSforJAL.pdf.
[43] M. Buscemi, U. Montanari, CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements, in: ESOP, volume 4421 of *LNCS*, Springer, 2007, pp. 18–32.
[44] V. Saraswat, M. Rinard, Concurrent Constraint Programming, in: POPL, ACM Press, 1990, pp. 232–245.
[45] S. Bistarelli, U. Montanari, F. Rossi, Soft concurrent constraint programming, ACM Transactions on Computational Logic 7 (2006) 563–589.
[46] S. Bistarelli, U. Montanari, F. Rossi, Semiring-based constraint satisfaction and optimization, Journal of the ACM 44 (1997) 201–236.
[47] U. Montanari, F. Rossi, Constraint Relaxation may be Perfect, Artificial Intelligence 48 (1991) 143–170.
[48] S. Bistarelli, Semirings for Soft Constraint Solving and Programming, LNCS, Springer, 2004.
[49] M. Wirsing, G. Denker, C. Talcott, A. Poggio, L. Briesemeister, A Rewriting Logic Framework for Soft Constraints, in: WRLA, volume 176(4) of *ENTCS*, Elsevier, 2007, pp. 181–197.
[50] V. A. Saraswat, M. C. Rinard, P. Panangaden, Semantic Foundations of Concurrent Constraint Programming, in: POPL, ACM Press, 1991, pp. 333–352.
[51] R. Milner, Communication and concurrency, Prentice-Hall, 1989.
[52] M. Abadi, C. Fournet, Mobile values, new names, and secure communication, in: POPL, ACM Press, 2001, pp. 104–115.
[53] M. Hennessy, J. Riely, Resource access control in systems of mobile agents, Information and Computation 173 (2002) 82–120.
[54] R. De Nicola, G. Ferrari, R. Pugliese, KLAIM: A Kernel Language for Agents Interaction and Mobility, Transactions on Software Engineering 24 (1998) 315–330.
[55] R. Cleaveland, G. Lüttgen, V. Natarajan, Priorities in process algebra, Handbook of Process Algebra, chapter 12 (2001) 391–424.
[56] J. Camilleri, G. Winskel, CCS with Priority Choice, Information and Computation 116 (1995) 26–37.
[57] I. Phillips, CCS with priority guards, Journal of Logic and Algebraic Programming 75 (2008) 139–165.
[58] M. Butler, C. Ferreira, An Operational Semantics for StAC, a Language for Modelling Long-Running Business Transactions, in: COORDINATION, volume 2949 of *LNCS*, Springer, 2004, pp. 87–104.
[59] M. Mazzara, I. Lanese, Towards a Unifying Theory for Web Services Composition, in: WS-FM, volume 4184 of *LNCS*, Springer, 2006, pp. 257–272.
[60] M. Mazzara, R. Lucchi, A pi-calculus based semantics for WS-BPEL, Journal of Logic and Algebraic Programming 70 (2006) 96–118.
[61] R. Bruni, H. Melgratti, U. Montanari, Theoretical foundations for compensations in flow composition languages, in: POPL, ACM Press, 2005, pp. 209–220.
[62] R. Bruni, M. Butler, C. Ferreira, T. Hoare, H. Melgratti, U. Montanari, Comparing two approaches to compensable flow composition, in: CONCUR, volume 3653 of *LNCS*, Springer, 2005, pp. 383–397.
[63] F. Corradini, D. D'Ortenzio, P. Inverardi, On the Relationships among four Timed Process Algebras, Fundamenta

Informaticae 38 (1999) 377–395.
[64] X. Nicollin, J. Sifakis, An Overview and Synthesis on Timed Process Algebras, in: CAV, volume 575 of *LNCS*, Springer, 1991, pp. 376–398.
[65] F. Moller, C. Tofts, A Temporal Calculus of Communicating Systems, in: CONCUR, volume 458 of *LNCS*, Springer, 1990, pp. 401–415.
[66] F. Moller, C. Tofts, Relating Processes With Respect to Speed, in: CONCUR, volume 527 of *LNCS*, Springer, 1991, pp. 424–438.
[67] M. Viroli, Towards a Formal Foundation to Orchestration Languages, in: WS-FM, volume 105 of *ENTCS*, Elsevier, 2004, pp. 51–71.
[68] K. Honda, V. T. Vasconcelos, M. Kubo, Language Primitives and Type Discipline for Structured Communication-Based Programming, in: ESOP, volume 1381 of *LNCS*, Springer, 1998, pp. 122–138.
[69] M. Carbone, K. Honda, N. Yoshida, Structured Communication-Centred Programming for Web Services, in: ESOP, volume 4421 of *LNCS*, Springer, 2007, pp. 2–17.
[70] I. Lanese, F. Martins, A. Ravara, V. Vasconcelos, Disciplining Orchestration and Conversation in Service-Oriented Computing, in: SEFM, IEEE Computer Society Press, 2007, pp. 305–314.
[71] L. Caires, H. Vieira, Conversation types, Theor. Comput. Sci. 411 (2010) 4399–4440.
[72] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: POPL, ACM Press, 2008, pp. 273–284.
[73] R. Bruni, I. Lanese, H. Melgratti, E. Tuosto, Multiparty Sessions in SOC, in: COORDINATION, volume 5052 of *LNCS*, Springer, 2008, pp. 67–82.
[74] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, G. Zavattaro, Choreography and Orchestration: A Synergic Approach for System Design, in: ICSOC, volume 3826 of *LNCS*, Springer, 2005, pp. 228–240.
[75] M. Carbone, K. Honda, N. Yoshida, A Calculus of Global Interaction based on Session Types, in: DCM, volume 171(3) of *ENTCS*, Elsevier, 2007, pp. 127–151.
[76] L. Wischik, P. Gardner, Explicit fusions, Theoretical Computer Science 340 (2005) 606–630.
[77] S. Bistarelli, F. Santini, A Nonmonotonic Soft Concurrent Constraint Language for SLA Negotiation, in: VODCA, volume 236 of *ENTCS*, Elsevier, 2009, pp. 147–162.
[78] D. Bacciu, A. Botta, H. Melgratti, A fuzzy approach for negotiating quality of services, in: TGC, volume 4661 of *LNCS*, Springer, 2006, pp. 200–217.
[79] R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, E. Tuosto, A Formal Basis for Reasoning on Programmable QoS, in: Verification: Theory and Practice, volume 2772 of *LNCS*, Springer, 2003, pp. 436–479.
[80] R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, E. Tuosto, A Process Calculus for QoS-Aware Applications, in: COORDINATION, volume 3454 of *LNCS*, Springer, 2005, pp. 33–48.
[81] J. Díaz, C. Rueda, F. Valencia, π^+-calculus: A Calculus for Concurrent Processes with Constraints, CLEI electronic journal 1 (1998).
[82] M. Bartoletti, P. Degano, G. Ferrari, Security Issues in Service Composition, in: FMOODS, volume 4037 of *LNCS*, Springer, 2006, pp. 1–16.
[83] A. Lazovik, M. Aiello, R. Gennari, Encoding Requests to Web Service Compositions as Constraints, in: CP, volume 3709 of *LNCS*, Springer, 2005, pp. 782–786.
[84] M. Bravetti, G. Zavattaro, Contract Based Multi-party Service Composition, in: FSEN, volume 4767 of *LNCS*, Springer, 2007, pp. 207–222.
[85] F. Banti, R. Pugliese, F. Tiezzi, An Accessible Verification Environment for UML Models of Services, Journal of Symbolic Computation 46 (2011) 119–149.
[86] L. Bettini, R. De Nicola, M. Lacoste, M. Loreti, Implementing Session Centered Calculi, in: COORDINATION, volume 5052 of *LNCS*, Springer, 2008, pp. 17–32.
[87] L. Cesari, R. Pugliese, F. Tiezzi, A tool for rapid development of WS-BPEL applications, SIGAPP Applied Computing Review 11 (2010) 27–40.
[88] F. Montesi, C. Guidi, R. Lucchi, G. Zavattaro, JOLIE: a Java Orchestration Language Interpreter Engine, in: MTCoord, volume 181 of *ENTCS*, Elsevier, 2007, pp. 19–33.
[89] G. Ferrari, R. Guanciale, D. Strollo, E. Tuosto, Event-Based Service Coordination, in: Concurrency, Graphs and Models, volume 5065 of *LNCS*, Springer, 2008, pp. 312–329.
[90] G. Ferrari, R. Guanciale, D. Strollo, Event based service coordination over dynamic and heterogeneous networks, in: ICSOC, volume 4294 of *LNCS*, Springer, 2006, pp. 453–458.