Specifying and Analysing SOC Applications with COWS*

– Dedicated to Ugo Montanari on the occasion of his 65th birthday –

Alessandro Lapadula, Rosario Pugliese and Francesco Tiezzi

Dipartimento di Sistemi e Informatica Università degli Studi di Firenze

Abstract. COWS is a recently defined process calculus for specifying and combining service-oriented applications, while modelling their dynamic behaviour. Since its introduction, a number of methods and tools have been devised to analyse COWS specifications, like e.g. a type system to check confidentiality properties, a logic and a model checker to express and check functional properties of services. In this paper, by means of a case study in the area of automotive systems, we demonstrate that COWS, with some mild linguistic additions, can model all the phases of the life cycle of service-oriented applications, such as publication, discovery, negotiation, orchestration, deployment, reconfiguration and execution. We also provide a flavour of the properties that can be analysed by using the tools mentioned above.

1 Introduction

In recent years, 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. SOC advocates the use of loosely coupled 'services', to be understood as autonomous, platform-independent, computational entities that can be described, published, discovered, and assembled, as the basic blocks for building interoperable and evolvable systems and applications. While early examples of technologies that are at least partly service-oriented date back to CORBA, DCOM, J2EE and IBM WebSphere, the most successful instantiation of the SOC paradigm are probably the more recent *web services*. These are sets of operations 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 have invested a lot of efforts.

Current software engineering technologies for SOC, however, remain at the descriptive level and lack rigorous formal foundations. We are still experiencing a gap between practice (programming) and theory (formal methods and analysis techniques) in the design of SOC applications. The challenges come from the necessity of dealing at once with issues like communication, co-operation, resource usage, security, failures, etc. 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*

* This work has been supported by the EU project SENSORIA, IST-2005-016004.

calculi, a cornerstone of current foundational research on specification and analysis of concurrent, distributed and mobile systems through mathematical — mainly algebraic and logical — tools. Thus, many process calculi have been designed, addressing one aspect or another of SOC and aiming at assessing the adequacy of diverse sets of primitives w.r.t. modelling, combining and analysing service-oriented applications.

Due to their algebraic nature, process calculi convey in a distilled form the compositional programming style of SOC. Thus, for example, many well-known problems related to services composition (e.g., messages not received, race conditions, deadlocks, incompatible behaviours) could be investigated through an adequate and sufficiently expressive process calculus. A major benefit of using process calculi is that they enjoy a rich repertoire of elegant meta-theories, proof techniques and analytical tools that can be likely tailored to the needs of service-based applications. It has been already argued that type systems, model checking and (bi)simulation analysis provide adequate tools to address topics relevant to the web services technology (see e.g. [20, 24]). 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.

By taking inspiration from well-known process calculi and from the standard language for orchestration of web services WS-BPEL [22], in [15] we have designed COWS (*Calculus for Orchestration of Web Services*), a process calculus for specifying and combining service-oriented applications, while modelling their dynamic behaviour. We have shown that COWS can model different and typical features of web services, such as, e.g., multiple start activities, receive conflicts, routing of correlated messages, service instances and interactions among them. Since its definition, some linguistic extensions have been introduced to model timed activities [17] and dynamic service discovery and negotiation [19], thus obtaining a linguistic formalism capable of modelling all the phases of the life cycle of service-oriented applications. A number of methods and tools have also been devised to analyse COWS specifications, such as the stochastic extension defined in [23] to enable quantitative reasoning on service behaviours, the type system introduced in [18] to check confidentiality properties, and the logic and model checker presented in [9] to express and check functional properties of services. In this paper, by means of the 'on road assistance scenario', a case study in the area of automotive systems defined and analysed within the EU project SENSORIA [2], we provide a flavour of COWS main features and specification style, and illustrate the classes of properties that can be analysed by using some of the tools mentioned above.

The rest of the paper is organized as follows. Section 2 introduces the scenario that will be used throughout the paper for illustration purposes. Section 3 presents syntax and main features of COWS; this is done in a step-by-step fashion while modelling some services within the scenario and their orchestration. Section 4 shows that also service discovery and negotiation can be naturally modelled in COWS by exploiting some mild linguistic additions, i.e. timed activities, constraints and operations on them. Section 5 sums up a type-based approach for expressing and enforcing confidentiality properties. Section 6 illustrates a logical verification framework including the logic SocL for expressing functional properties of services and the on-the-fly model checker CMC for verifying them. Section 7 concludes the paper with some final remarks.

2 On road assistance scenario

The ‘on road assistance scenario’ [13] is one of the scenarios in the area of automotive systems defined and analysed within the EU project SENSORIA [2] and describes some functionalities that will be likely available in the near future. The scenario involves a number of services that are discovered and bound at run-time according to levels of service specified at design time, so as to deliver the best available functionalities at agreed levels of quality. A brief description follows.

> The in-vehicle *diagnostic* system reports a severe failure when the car is no longer drivable. 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 selects 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. Before being enable to order services, the owner of the car has to deposit a security payment, that 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* service, providing the car’s current location, and an *orchestrator*, coordinating all the described services.

An UML-like activity diagram of the orchestration of services is shown in Figure 1. For simplicity, we assume that the orchestration is only triggered either by an ‘engine failure’ or by a ‘low oil level’ sensor signal. The process starts with a request from the orchestrator to the bank to charge the driver’s credit card with the security deposit payment. This is modelled by the UML action requestCardCharge 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 service. 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 (best) services.

Service ordering is modelled by the UML actions orderGarage, orderTowTruck and orderRentalCar. 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 where the car will be handed over to the driver.

The workflow described in Figure 1 models the overall behaviour of the system. Besides interactions among services, it also includes activities using concepts developed for long running business transactions (e.g. in [11, 22]). 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. Specifically, in the considered scenario, the security deposit payment charged to the

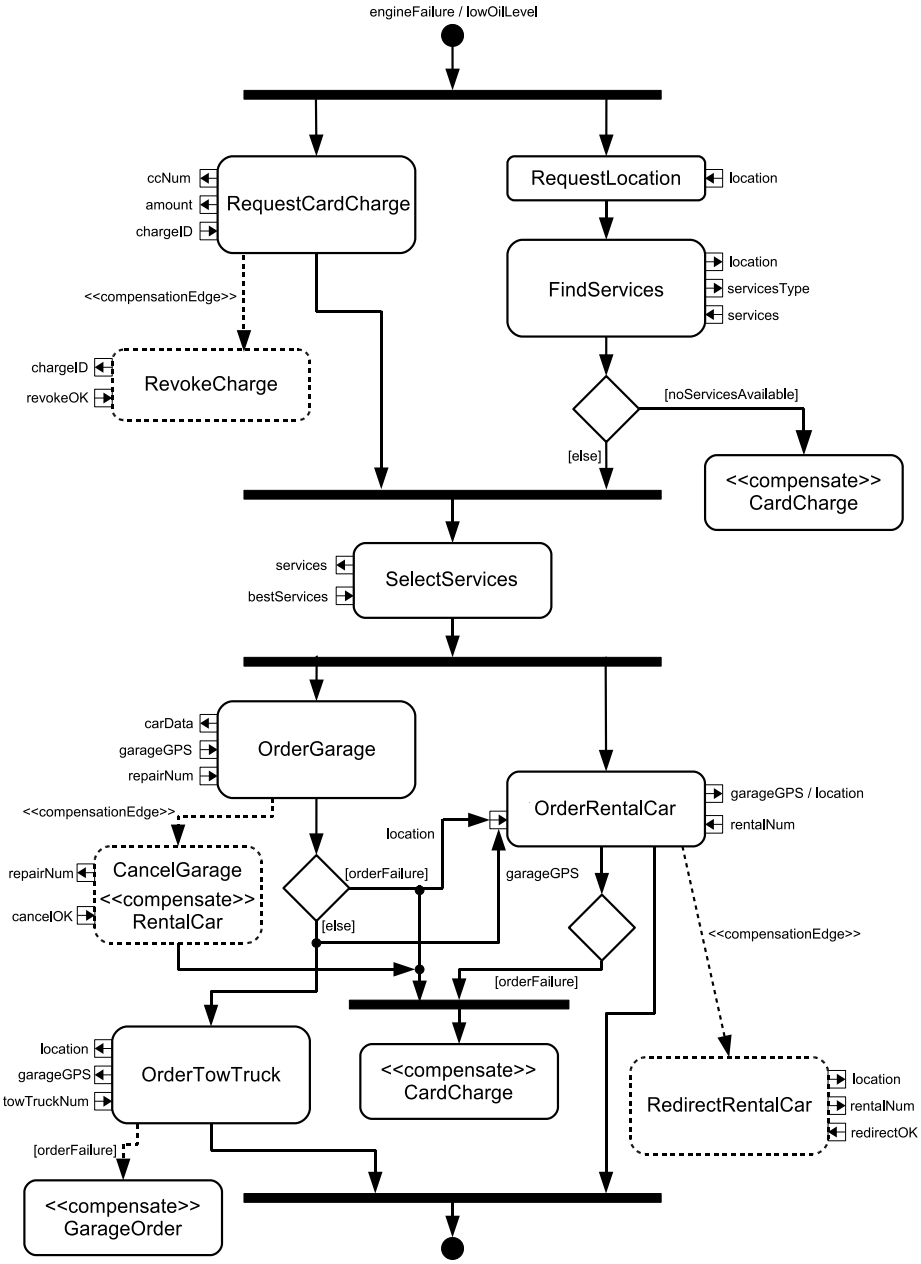

Fig. 1. Orchestration in the on road assistance scenario

$s ::=$	$\mathbf{kill}(k)$	$u \bullet u'!\overline{e}$	(kill, invoke)
	$\sum_{i=0}^{l} p_i \bullet o_i?\overline{w_i}.s_i$	$s \mid s$	(receive-guarded choice, parallel)
	$\{\!\{s\}\!\}$	$[d]\, s$ $\quad *\, s$	(protection, delimitation, replication)

Table 1. COWS syntax

driver'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. Moreover, if ordering a tow truck fails, the garage appointment has to be cancelled and the rental car delivery has to be redirected to the stranded car's actual location. Instead, if ordering the car rental fails, the overall process may not fail, as the activity is enclosed in a sub-transaction.

3 COWS: a Calculus for Orchestration of Web Services

In this section, we report the syntax of COWS and explain the semantics of its primitives in a step-by-step fashion while modelling the on road assistance scenario (the complete specification can be found in [16]). Due to lack of space, here we only provide an informal account of the semantics of COWS and refer the interested reader to [15, 14] for a formal presentation, for examples illustrating peculiarities and expressiveness of the language, and for comparisons with other process-based and orchestration formalisms. To get accustomed to using the language one can also use CMC [1], a tool supporting the automated derivation of all computations originating from a COWS term.

3.1 Syntax

The syntax of COWS, given in Table 1, is parameterized by three countable and pairwise disjoint sets: the set of *(killer) labels* (ranged over by $k, k', \ldots$), 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, but we assume that it includes the set of *names*, ranged over by m, n, o, p, ..., mainly used to represent partners and operations. The language is also parameterized by a set of *expressions*, ranged over by e, whose exact syntax is deliberately omitted. We just assume that expressions contain, at least, values and variables, and do not include killer labels (that, hence, are *not* communicable values). Partner and operation names can be combined to designate communication *endpoints*; e.g. $p \bullet o$ denotes the endpoint composed of the partner p and the operation o. Being values, partner and operation names can be exchanged in communication, but dynamically received names can only be used to designate endpoints for service invocation. Indeed, endpoints of receive activities are identified statically because their syntax only allows using names and not variables.

We use w to range over values and variables, u to range over names and variables, and d to range over killer labels, names and variables. Notation $\overline{\cdot}$ stands for tuples of objects, e.g. $\overline{x}$ is a compact notation for denoting the tuple of variables $\langle x_1, \ldots, x_n \rangle$ (with $n \geq 0$). In the sequel, we shall use $\mathbf{0}$ to denote empty choice and $+$ to abbreviate binary

choice. We will omit trailing occurrences of $\mathbf{0}$, writing e.g. $p \cdot o?\overline{w}$ instead of $p \cdot o?\overline{w}.\mathbf{0}$, and write $[d_1, \ldots, d_n]\, s$ in place of $[d_1] \ldots [d_n]\, s$. We will write $Z \triangleq W$ to assign a symbolic name Z to the term W.

The only *binding* construct is delimitation: $[d]\, s$ binds d in the scope s (the notions of *bound* and *free* occurrences of a name/variable/label are defined accordingly). In fact, differently from most process calculi, receive activities in COWS bind neither names nor variables. This enables e.g. easily modelling and updating the shared state of concurrent threads within each service instance. Delimitation can be used to generate fresh names whose scope can later dynamically change because of taking place of communication. This is exactly as in the π-calculus [21]. However, delimitation is more general than the restriction of the π-calculus since it can be also used to declare variables (thus regulating the range of application of the substitutions generated by communications) and to delimit the field of action of kill activities. Notably, killer labels are dealt with differently from names and variables since, being not communicable values, their scope is statically determined by the corresponding delimitation and can never change.

3.2 Basic operators for service orchestration

The COWS term representing the 'orchestration' in Figure 1 is

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

The services above are composed by using the *parallel composition* operator $\_ \mid \_$ that allows the different components to be concurrently executed and to interact with each other. The *delimitation* operator $[\_]\_$ is used here to declare that p_{car} is a (partner) name known to all services of the in-vehicle platform, i.e. *Orchestrator*, *GPS*, *Discovery*, *Reasoner* and *SensorsMonitor*, and only to them.

Orchestrator, the most important component of the in-vehicle platform, is

$$[x_{carData}]\,(\; p_{car} \cdot o_{engfail}?\langle x_{carData}\rangle.s_{engfail} \;+\; p_{car} \cdot o_{lowoil}?\langle x_{carData}\rangle.s_{lowoil}\;)$$

This term uses the *choice* operator $\_ + \_$ to pick one of those alternative 'recovery' behaviours whose execution can start immediately. The *receive-guarded prefix* operator $p_{car} \cdot o_i?\langle x_{carData}\rangle.\_$ expresses that each recovery behaviour starts with a *receive* activity of the form $p_{car} \cdot o_i?\langle x_{carData}\rangle$ corresponding to reception of a request emitted, when a failure arises, by *SensorsMonitor* (a term representing the behaviour of the 'low level vehicle platform'). *Receives*, together with *invokes*, written as $p \cdot o!\langle e_1, \ldots, e_m\rangle$, are the basic communication activities provided by COWS. Besides input parameters and sent values, they indicate the endpoint $p \cdot o$ through which the communication should occur. $p \cdot o$ can be interpreted as a specific implementation of operation o provided by the service identified by the logic name p. This naming mechanism allows a same service to be identified by means of different logic names, i.e. to play more than one partner role as in WS-BPEL. An inter-service communication takes place when the arguments of a receive and of a concurrent invoke along the same endpoint do match[1],

[1] The pattern-matching mechanism permits correlating messages logically forming a same interaction 'session' by means of their same contents. We refer to [15, 14] for further details.

and causes replacement of the variables arguments of the receive with the corresponding values arguments of the invoke (within the scope of variables declarations). For example, variable $x_{carData}$, declared local to *Orchestrator* by means of the delimitation operator, is initialized by the receive leading the recovery activity with data provided by *SensorsMonitor*. Notice that, while executing a recovery behaviour, *Orchestrator* does not accept other recovery requests. We are also assuming that it is restarted at the end of the recovery task.

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

$$
\begin{array}{l}
[p_e, o_e, x_{loc}, x_{list}] \\
(\ (\ \mathit{RequestCardCharge} \mid \mathit{RequestLocation}.\mathit{FindServices}\ ) \\
\quad \mid\ p_e \cdot o_e?\langle\rangle.\ p_e \cdot o_e?\langle\rangle.\ \mathit{SelectServices}. \\
\qquad\qquad [x_{garageGPS}]\ (\ \mathit{OrderGarage}.\mathit{OrderTowTruck} \mid \mathit{OrderRentalCar}\ )\ )
\end{array}
$$

$p_e \cdot o_e$ 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. Variables x_{loc}, x_{list} and $x_{garageGPS}$ are used to store the value of the car's current location, the list of closer on road services discovered and the garage's GPS location, respectively. To present the specification of $s_{engfail}$ in terms of the UML actions of Figure 1, we have used an auxiliary 'sequence' notation. Thus, e.g., *RequestLocation*.*FindServices* indicates that execution of *RequestLocation* terminates before execution of *FindServices* starts. Indeed, *RequestLocation*.*FindServices* actually stands for the COWS term

$$
\begin{array}{l}
p_{car} \cdot o_{reqLoc}!\langle\rangle \mid p_{car} \cdot o_{respLoc}?\langle x_{loc}\rangle. \\
\qquad (\ p_{car} \cdot o_{find}!\langle x_{loc}, \mathit{ServicesType}\rangle \\
\qquad\quad \mid\ p_{car} \cdot o_{servicesFound}?\langle x_{list}\rangle.\ p_e \cdot o_e!\langle\rangle\ +\ p_{car} \cdot o_{servicesNotFound}?\langle\rangle\ )
\end{array}
$$

where *RequestLocation* and *FindServices* are

$$
\begin{array}{rcl}
\mathit{RequestLocation} & \triangleq & p_{car} \cdot o_{reqLoc}!\langle\rangle \mid p_{car} \cdot o_{respLoc}?\langle x_{loc}\rangle \\
\\
\mathit{FindServices} & \triangleq & p_{car} \cdot o_{find}!\langle x_{loc}, \mathit{ServicesType}\rangle \\
& & \mid\ p_{car} \cdot o_{servicesFound}?\langle x_{list}\rangle.\ p_e \cdot o_e!\langle\rangle
\end{array}
$$

Endpoints of service invocations can also contain variables as, e.g., in the term

$$
\begin{array}{rcl}
\mathit{OrderGarage} & \triangleq & x_{garage} \cdot o_{order}!\langle p_{car}, x_{carData}\rangle\ \mid \\
& & [x_{repairNum}]\ (p_{car} \cdot o_{garageFail}?\langle\rangle\ +\ p_{car} \cdot o_{garageOK}?\langle x_{repairNum}, x_{garageGPS}\rangle)
\end{array}
$$

Here, variable x_{garage} is used to invoke a garage service whose partner name is unknown at design time. This garage will be selected dynamically by activity *SelectService* that, through a communication, replaces x_{garage} with the actual partner name of the garage. Indeed, in COWS dynamic binding of discovered services and service reconfiguration rely on the exchange of partner and operation names in communication.

Bank, the last service we show in this section, can serve multiple requests simultaneously. This behaviour is modelled by exploiting the *replication* operator $*\_$ to spawn

in parallel as many copies of its argument term as necessary. The definition of *Bank* is

$$
\begin{array}{l}
*\,[x_{cust}, x_{cc}, x_{amount}, o_{checkOK}, o_{checkFail}] \\
\quad p_{bank} \cdot o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}\rangle. \\
\quad (\,< \texttt{perform some checks and reply on } o_{checkOK} \texttt{ or } o_{checkFail} > \\
\quad\quad \mid p_{bank} \cdot o_{checkFail}?\langle\rangle.\ x_{cust} \cdot o_{chargeFail}!\langle\rangle \\
\quad\quad\ + \ p_{bank} \cdot o_{checkOK}?\langle\rangle.\ [chargeID]\,(\ x_{cust} \cdot o_{chargeOK}!\langle chargeID\rangle \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid p_{bank} \cdot o_{revoke}?\langle chargeID\rangle. \\
\quad\quad\quad\quad\quad\quad\quad\quad\ < \texttt{revoke } chargeID >.\ x_{cust} \cdot o_{revokeOK}!\langle\rangle\,)\,)
\end{array}
$$

Once prompted by a request, differently from *Orchestrator*, *Bank* creates one specific instance to serve that request and is immediately ready to concurrently serve other requests. Notably, each instance exploits communication on 'internal' operations $o_{checkOK}$ and $o_{checkFail}$ to model a conditional choice, and creates a new 'charge identifier' by means of the delimitation operator (that acts here as the restriction operator of the π-calculus). Thus, if after some invocations the service receives a message along the endpoint $p_{bank} \cdot o_{revoke}$ to revoke a request, a certain number of service instances could be able to accept it. However, the message is routed to the proper instance by exploiting, as a correlation value, a unique identifier (that is named *chargeID* in the term above) characterizing the instance.

3.3 Fault and compensation handling

We now show how to modify the specification described in the previous section for adding the fault and compensation activities depicted in Figure 1. For improving readability, these activities are highlighted by a gray background to distinguish them from 'normal behaviour'. For example, the term modelling the garage ordering is:

$$
\begin{array}{rcl}
\mathit{OrderGarage} & \triangleq & x_{garage} \cdot o_{order}!\langle p_{car}, x_{carData}\rangle \\
& & \mid [x_{repairNum}]\ p_{car} \cdot o_{garageFail}?\langle\rangle. \\
& & \qquad (\ p_{car} \cdot o_{undo}!\langle cc\rangle \\
& & \qquad\ \mid [p, o]\,(p \cdot o!\langle x_{loc}\rangle \mid p \cdot o?\langle x_{garageGPS}\rangle)\,) \\
& & \quad + \ p_{car} \cdot o_{garageOK}?\langle x_{repairNum}, x_{garageGPS}\rangle. \\
& & \qquad p_{car} \cdot o_{undo}?\langle garage\rangle. \\
& & \qquad (\ x_{garage} \cdot o_{cancel}!\langle x_{repairNum}\rangle \\
& & \qquad\ \mid p_{car} \cdot o_{cancelOK}?\langle\rangle \\
& & \qquad\ \mid p_{car} \cdot o_{undo}!\langle cc\rangle \mid p_{car} \cdot o_{undo}!\langle rentalCar\rangle\,)
\end{array}
$$

Thus, if ordering a garage fails, the compensation of the credit card charge is invoked by sending a signal *cc* (abbreviation of 'card charge') along the endpoint $p_{car} \cdot o_{undo}$ and the rental car delivery is redirected by assigning the car's current location x_{loc} to the variable $x_{garageGPS}$ (this assignment is modelled by means of communication along the private endpoint $p \cdot o$). Otherwise, a compensation handler is installed that is invoked whenever tow truck ordering fails and, in that case, attempts to cancel the garage order and to compensate the credit card charge and the rental car order.

To model fault handling and compensation behaviours, the term *OrderGarage* exploits interactions along the endpoint $p_{car} \cdot o_{undo}$. However, to better support the specification of these aspects, COWS provides two further constructs. *Kill* activities of the form **kill**(k), where k is a killer label, can be used to force termination of all unprotected parallel terms inside the enclosing $[k]$, that stops the killing effect. Kill activities are executed *eagerly* with respect to the other parallel activities but critical code, such as e.g. fault/compensation signals and handlers, can be protected from the effect of a forced termination by using the *protection* operator $\{\!|\_|\!\}$. By exploiting these new features, the recovery behaviour $s_{engfail}$ becomes

$$
\begin{array}{l}
[p_e, o_e, x_{loc}, x_{list}, o_{undo}, k\,] \\
(\,(\,\mathit{RequestCardCharge} \mid \mathit{RequestLocation}.\,\mathit{FindServices}\,) \\
\quad \mid\; p_e \cdot o_e?\langle\rangle.\; p_e \cdot o_e?\langle\rangle.\; \mathit{SelectServices}. \\
\qquad\qquad [x_{garageGPS}]\,(\,\mathit{OrderGarage}.\,\mathit{OrderTowTruck} \mid \mathit{OrderRentalCar}\,)\,)
\end{array}
$$

where *RequestCardCharge* and *FindServices* are defined as

$$
\begin{array}{rcl}
\mathit{FindServices} & \triangleq & p_{car} \cdot o_{find}!\langle x_{loc}, \mathit{ServicesType}\rangle \\
& & \mid\; p_{car} \cdot o_{servicesFound}?\langle x_{list}\rangle.\; p_e \cdot o_e!\langle\rangle \\
& & \quad + p_{car} \cdot o_{servicesNotFound}?\langle\rangle. \\
& & \qquad (\,\mathbf{kill}(k) \mid \{\!|\, p_{car} \cdot o_{undo}!\langle cc\rangle \mid p_{car} \cdot o_{undo}!\langle cc\rangle \,|\!\}\,)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{RequestCardCharge} & \triangleq & p_{bank} \cdot o_{charge}!\langle p_{car}, \mathit{ccNum}, \mathit{amount}\rangle \\
& & \mid\; \{\!|\, [x_{chargeID}]\, p_{car} \cdot o_{chargeFail}?\langle\rangle.\; \mathbf{kill}(k) \\
& & \qquad + \; p_{car} \cdot o_{chargeOK}?\langle x_{chargeID}\rangle. \\
& & \qquad\quad (\, p_e \cdot o_e!\langle\rangle \mid p_{car} \cdot o_{undo}?\langle cc\rangle.\, p_{car} \cdot o_{undo}?\langle cc\rangle. \\
& & \qquad\qquad\quad (\, p_{bank} \cdot o_{revoke}!\langle x_{chargeID}\rangle \\
& & \qquad\qquad\quad \mid p_{car} \cdot o_{revokeOK}?\langle\rangle\,)\,)\,|\!\}
\end{array}
$$

Thus, whenever services finding fails, *FindServices* terminates the whole recovery behaviour and sends two signals *cc* along the endpoint $p_{car} \cdot o_{undo}$. Similarly, if charging the credit card fails, then *RequestCardCharge* terminates the whole recovery behaviour $s_{engfail}$. 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 whenever both *OrderGarage* and *OrderRentalCar* (not shown here) fail.

4 Service publication, discovery and negotiation

We have demonstrated so far that COWS can model service specification, orchestration, reconfiguration, and execution. Now, we focus on other important phases of the life cycle of service-oriented applications. In fact, in a service-oriented architecture, services can play essentially three different roles: the provider, the requester and the registry. Providers offer functionalities and publish machine-readable service descriptions on registries to enable automated discovery and invocation by requesters. In addition to the function that the service performs, service descriptions also include non-functional properties, such as e.g., response time, availability, reliability, security, and

performance, that jointly represent the *quality of the service* (QoS). Some of these properties could depend on the current run-time configuration of the system (e.g. the maximum allowed bandwidth might depend on the actual load of the server), thus a *dynamic discovery* process is often needed to find a provider that meets the requesters' requirements. 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, and the remedial actions to be performed if these are not met.

We want now to demonstrate that service publication, discovery and SLA negotiation can be naturally modelled in COWS by exploiting the additions of 'timed' activities and 'constraints'. Timed activities have been introduced in [17], by adding specific rules for modelling time passing to the COWS operational semantics, since it is not known to what extent timed computation can be reduced to untimed forms of computation [25]. Specifically, COWS is extended with a WS-BPEL-like *wait* activity of the form $\oplus_e$, that suspends the execution of the invoking service until the time interval whose duration is specified as an argument has elapsed and can be used as a guard for the choice operator. Constraints have been introduced in [19], by exploiting the fact that COWS' definition is parameterised with respect to a few sets of objects, namely the set of values and that of expressions that operate on them. Notably, we do not take a definite standing on which of the many kinds of constraints one should use. For example, one could use *crisp* constraints, that can only be satisfied or violated, or *soft* constraints, that instead can be satisfied with multiple consistency levels (these are usually expressed by means of *c-semirings* [4] and interpreted as levels of preference or importance). From time to time, the appropriate kind of constraints to work with should be chosen depending on what one intends to model.

Still in [19] we argue that the concurrent constraint computing model can be easily mimicked in COWS. This model of computation is based on a shared store of constraints that provides partial information about possible values that program variables can assume. In COWS the store of constraints is represented by the following service:

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

where C is the multiset of constraints currently in the store, while p_s is a distinguished partner, and 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 $p_s \cdot o_{set}$. The programs running in parallel with the store 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). For example, the service `tell` $c.s$ willing to perform operation `tell` c and then to continue as service s can be rendered in COWS as follows:

$$[p,o]\,(\, p \cdot o!\langle c\rangle \mid [y]\; p \cdot o?\langle y\rangle.[x]\; p_s \cdot o_{get}?\langle\langle y, x\rangle\rangle.(\{\!|\, p_s \cdot o_{set}!\langle x \uplus \{y\}\rangle \,|\!\}\mid s)\,)$$

Due to lack of space, we refer the interested reader to [19] for the implementation of the other operations and further details.

Now, like in cc-pi [5], service descriptions and SLA requirements can be 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, and for the SLA negotiation process. Consistency of the set of constraints resulting from negotiation means that the agreement has been reached. Timed activities can be exploited to allow services not to get stuck forever waiting on a receive.

We use the on road assistance scenario to illustrate all such features and to put the related mechanisms to work. Initially, each on road service 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. Now, by assuming that the registry provides the operation o_{pub} by means of the partner name p_{reg}, a garage service can request the publication of its description as follows:

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

`gps` is what we call a *constraint variable*. In fact, it is a specific name and, hence, is not affected by substitution application. Constraint variables are used to avoid that taking place of communication can make the store inconsistent. Indeed, suppose constraints in the store may contain variables and consider the following example:

$$[x]\,(\, store_{\emptyset} \mid \mathtt{tell}(x \leq 5).\,(p \cdot o!\langle 6\rangle \mid p \cdot o?\langle x\rangle)\,)$$

After action `tell` has added the constraint $x \leq 5$ to the store, communication along the endpoint $p \cdot o$ can modify the constraint in $6 \leq 5$, thus making the store inconsistent. To distinguish constraint variables from COWS (true) variables, the formers are written in the typewriter style (e.g. $\mathtt{x}$, $\mathtt{y}$, ...). The service registry can be 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

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

$$\begin{aligned} R' \;\triangleq\; & [k]\,(\,*\,[x_p, x_{const}]\; p_{reg} \cdot o_{DB}?\langle x_{type}, x_p, x_{const}\rangle. \\ & \qquad (\, \{\!|\, p_{reg} \cdot o_{pub}!\langle x_{type}, x_p, x_{const}\rangle \,|\!\} \mid \mathtt{check}\; x_{const}.\; p_{reg} \cdot o_{addToList}!\langle x_p\rangle\,) \\ & \quad \mid {}^{\textcircled{\scriptsize t}}_{\delta}.\,(\,\mathbf{kill}(k) \mid \{\!|\, [x_{list}]\; p_{reg} \cdot o_{askList}?\langle x_{list}\rangle.\; x_{client} \cdot o_{resp}!\langle x_{list}\rangle \,|\!\}\,)\,) \end{aligned}$$

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 service descriptions are non-deterministically retrieved, thus the same provider can occur in the returned list many times. This 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. Moreover, since our notion of time does not rely on the so-called 'maximal progress assumption', i.e. communication does not prevent the execution of timed transitions, there is no guarantee that any service at all is retrieved.

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 3.2). 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

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

where the constraint $dist(x_{loc}, \texttt{gps}) < 20$ 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_{order} provided by the selected garage (see *OrderGarage* definition at page 8); 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.

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

In our example, the constraints state that for a repair in less than two days 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 $c = ((\texttt{cost}' > 2000 \wedge 6 < \texttt{duration}' < 24) \vee (\texttt{cost}' > 1500 \wedge \texttt{duration}' \geqslant 24))$ and, if it fails to reach an agreement within δ' time units, weakens the requirements and retries with the constraint $c' = ((\texttt{cost}' > 1700 \wedge 6 < \texttt{duration}' < 24) \vee (\texttt{cost}' > 1200 \wedge \texttt{duration}' \geqslant 24))$. 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:

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

Notably, operations `tell` cannot be used as guards for the choice operator. Thus, a term like $\mathtt{tell}\ c.\ s + {}^{\textcircled{\scriptsize t}}{}_{e}.\ s'$ should be considered as an abbreviation for

$$[p, q, o]\,(\ \mathtt{check}\ c.\ (p \cdot o!\langle\rangle \mid q \cdot o?\langle\rangle.\ \mathtt{tell}\ c.\ s)\ \mid {}^{\textcircled{\scriptsize t}}{}_{e}.\ s' + p \cdot o?\langle\rangle.\ q \cdot o!\langle\rangle\ )$$

Intuitively, if the constraint c is consistent with the store, the timer can be stopped (i.e. communication along $p \cdot o$ 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. Otherwise, if the timeout expires, the constraint cannot be added to the store.

5 A type system for checking confidentiality properties

The type system for COWS introduced in [18] permits expressing and forcing policies regulating the exchange of data among interacting services and ensuring that, in that respect, services do not manifest unexpected behaviours. This enables us to check confidentiality properties, e.g., that critical data such as credit card information are shared only with authorized partners. The type system has been obtained by tailoring to COWS the type-based approach for protecting data in distributed systems put forward in [12], in the context of a higher-order functional programming language, and drawn on in [6], in that of languages for global computing.

The types express the policies for data exchange in terms of *regions*, i.e. sets of service partner names attachable to each single datum. Service programmers can thus settle the partners usable to exchange any given datum (and, then, the services that can share it), thus avoiding the datum be accessed (by unwanted services) through unauthorized partners. Then, a type inference system (statically) performs some coherence checks (e.g. the service used in an invocation must belong to the regions of all data occurring in the argument of the invocation) and derives the minimal region annotations for variable declarations that ensure consistency of services initial configuration. COWS operational semantics uses these annotations in very efficient checks (i.e. subset inclusions) to authorise or block transitions, in order to guarantee that computations proceed according to them. This property, called *soundness*, can be stated as follows: a service s is *sound* if, for any datum v in s associated to region r and for all evolutions of s, it holds that v can be exchanged only by using services in r. As a consequence of the type soundness of the language, it follows that well-typed services always comply with the policies regulating the exchange of data among interacting services. In fact, it is also possible to move all dynamic checks to the static phase. This would require a static analysis that gathers information about all the values that each variable can assume at runtime and uses these information to verify the compliance with the specified policies. At the price of a more complex static phase, this approach, on the one hand,

would alleviate the runtime checks but, on the other hand, could discard terms that at runtime would behave soundly since statically they cannot guarantee to comply with their policies. We are currently evaluating and implementing the two approaches.

We illustrate now some relevant properties for the on road assistance scenario. Firstly, a driver in trouble must be assured that information about his credit card and his location cannot become available to unauthorized users. Thus, for example, the credit card identifier *ccNum*, communicated by activity *RequestCardCharge* to service *Bank*, gets annotated with the policy $\{p_{bank}\}$, that allows *Bank* to receive the datum but prevents it from transmitting the datum to other services. Other non-critical data, e.g. *amount*, can be transmitted without an attached policy. The typed version of *RequestCardCharge* (where irrelevant fault/compensation details are omitted) is defined as follows

$$p_{bank} \cdot o_{charge}!\langle p_{car}, \{ccNum\}_{\{p_{bank}\}}, amount\rangle \\ \mid [x_{chargeID}]\, p_{car} \cdot o_{chargeFail}?\langle\rangle + \; p_{car} \cdot o_{chargeOK}?\langle x_{chargeID}\rangle . p_e \cdot o_e!\langle\rangle$$

Notably, the annotations set by programmers are written as a subscript of the datum to which they refer to. Instead, the annotations put by the type inference, to better distinguish them from those put by the programmers, are written as a superscript of the variable declaration to which they refer to. Thus, the syntax of variable delimitation becomes $[\{x\}^r]\, s$, which means that the datum that dynamically will replace x will be used in s at most by the partners belonging to the region r. Hence, once the type inference phase ends, *Bank* gets annotated as follows

$$* [\{x_{cust}\}^{\{p_{bank}\}}, \{x_{cc}\}^{\{p_{bank}\}}, \{x_{amount}\}^{\{p_{bank}\}}, o_{checkOK}, o_{checkFail}] \\ p_{bank} \cdot o_{charge}?\langle x_{cust}, x_{cc}, x_{amount}\rangle . \\ (\, < \texttt{perform some checks and reply on } o_{checkOK} \texttt{ or } o_{checkFail} > \\ \mid p_{bank} \cdot o_{checkFail}?\langle\rangle . \, x_{cust} \cdot o_{chargeFail}!\langle\rangle \\ + \; p_{bank} \cdot o_{checkOK}?\langle\rangle . \\ [chargeID]\, (\, x_{cust} \cdot o_{chargeOK}!\langle chargeID\rangle \\ \mid p_{bank} \cdot o_{revoke}?\langle chargeID\rangle . \\ < \texttt{revoke } chargeID > . \, x_{cust} \cdot o_{revokeOK}!\langle\rangle \,)\,)$$

Indeed, the annotations inferred for variables x_{cust}, x_{cc} and x_{amount} are derived from the use of these variables made by *Bank*. Thus, they are assigned region $\{p_{bank}\}$ because they are only used in the receive along $p_{bank} \cdot o_{charge}$ and, of course, the partner name of the endpoint must belong to the region of the variables.

Suppose instead that service *Bank* (accidentally or maliciously) attempts to reveal the credit card number through some 'internal' operation such as $p_{int} \cdot o!\langle\{x_{cc}\}_r\rangle$, for some region r. For *Bank* to successfully complete the type inference phase, we should have $p_{int} \in r$. Then, as result of the inference, we would get the annotated variable declaration $[\{x_{cc}\}^{r'}]$, for some region r' with $r \subseteq r'$. Now, the interaction between the typed terms *RequestCardCharge* and *Bank* would be blocked by the runtime checks because the datum sent by *RequestCardCharge* would be annotated as $\{ccNum\}_{\{p_{bank}\}}$ while the region r' of the receiving variable x_{cc} is such that $p_{int} \in r \subseteq r' \not\subseteq \{p_{bank}\}$.

When delivering a datum, we can specify different policies according to the invoked service. For example, when sending the car's current location stored in x_{loc} to services *OrderTowTruck* and *OrderRentalCar*, we annotate it with the regions $\{x_{towTruck}\}$ and

$\{x_{rentalCar}\}$, respectively. This means that the corresponding service invocations get annotated as follows:

$$x_{towTruck} \cdot o_{order}!\langle p_{car}, \{x_{loc}\}_{\{x_{towTruck}\}}, x_{garageGPS}\rangle$$

$$x_{rentalCar} \cdot o_{redirect}!\langle x_{rentalNum}, \{x_{loc}\}_{\{x_{rentalCar}\}}\rangle$$

Notably, the used policies are not fixed at design time, but *depend* on the partner variables $x_{towTruck}$ and $x_{rentalCar}$, and, thus, will be determined by the services that these variables will be bound to as computation proceeds. For example, consider a towing truck service annotated as follows:

$$\begin{aligned} \mathit{TowTruck} \triangleq\ & *[\{x_{client}\}^{r_1}, \{x_{carLoc}\}^{r_2}, \{x_{garageLoc}\}^{r_3}, o_{checkOK}, o_{checkFail}] \\ & p_{towTruck} \cdot o_{order}?\langle x_{client}, x_{carLoc}, x_{garageLoc}\rangle. \\ & (< \texttt{perform some checks and reply on } o_{checkOK} \texttt{ or } o_{checkFail} > \\ & \quad | \ p_{towTruck} \cdot o_{checkFail}?\langle\rangle.\ x_{client} \cdot o_{towTruckFail}!\langle\rangle \\ & \quad + \ p_{towTruck} \cdot o_{checkOK}?\langle\rangle. \\ & \qquad [\mathit{towTruckNum}]\ x_{client} \cdot o_{towTruckOK}!\langle \mathit{towTruckNum}\rangle\ ) \end{aligned}$$

Now, the car's current location can be communicated to the towing truck if, and only if, the region of the variable x_{carLoc} that, after communication, will store the datum and the region of x_{loc} do comply, i.e. $r_2 \subseteq \{p_{towTruck}\}$.

As a final example, the on road services could want to guarantee that critical data sent to the in vehicle services, such as cost and quality of the service supplied, are not disclosed to competitors. For example, suppose that the towing truck services, like *TowTruck* before, must send the estimated travel time (ETT) to clients. To prevent this datum from being sent to competitor services, ETT is communicated with an attached policy that only authorizes the client partner to access it, as in the following activity

$$x_{client} \cdot o_{towTruckOK}!\langle \mathit{towTruckNum}, \{ETT\}_{\{x_{client}\}}\rangle$$

6 A logical framework for verifying functional properties

The logical verification framework introduced in [9] permits checking functional properties of services by abstracting away from the computational contexts in which they are operating. Specifically, services are abstractly considered as entities capable of accepting requests, delivering corresponding responses and, on-demand, cancelling requests, over specified interactions. The 'abstract' service actions are the following: $\mathit{request}(i, c)$, $\mathit{response}(i, c)$, $\mathit{cancel}(i, c)$ and $\mathit{fail}(i, c)$, where the name i indicates the interaction to which the corresponding 'concrete' action (i.e. the action occurring in the COWS specification) belongs, and c denotes a tuple of correlation values that identifies a particular invocation of the action. For example, $\mathit{request}(i, c)$ indicates that the corresponding concrete action represents the initial request of the interaction i and its invocation is identified by the correlation tuple c; similarly, $\mathit{response}(i, c)$, $\mathit{cancel}(i, c)$ and $\mathit{fail}(i, c)$ characterise actions that correspond to a response, a cancellation and a failure notification, respectively, of the interaction i. The name of the interaction or the correlation tuple will be omitted whenever they are not relevant. The correspondence between concrete actions used in the specifications and the abstract actions above must be defined from time to time by the user through appropriate abstraction rules.

Our abstract notion of services can be modelled by Doubly Labelled Transition Systems (L^2TSs, [7]) in a very natural way. Thus, to formalize functional properties of services, we have tailored UCTL [3], a branching time temporal logic interpreted over L^2TSs originally introduced to express properties of UML statecharts, to deal with service-oriented aspects. The resulting logic, that we call SocL, combines the action paradigm of ACTL [8] with predicates that are true over states. A key novelty of SocL is the possibility to specify parametric formulae to correlate service requests to the corresponding answers. Technically, correlation tuples in the actions of SocL formulae can use variables. Let *var* be a correlation variable name; we use \$*var* to indicate the binder of the occurrences of %*var*. For example, *request*(*i*, ⟨\$*var*⟩) denotes a request action for the interaction *i* that is uniquely identified through the correlation variable \$*var*. This way, subsequent actions, corresponding e.g. to response to that specific request, can unambiguously refer it through %*var*.

SocL allows us to express several relevant abstract properties for the services within the on road assistance scenario. A few examples follow.

1. $AG\, accepting\_request(engineFailure)$
   This formula means that the service *Orchestrator* is *available*, i.e. it is always capable to accept a request for the interaction *engineFailure*. Indeed, a formula like $AG\,\phi$ holds in a state q of a given L^2TS if, and only if, the formula ϕ holds in q and in all the states reachable from q along each path starting from q. *accepting_request*(*engineFailure*) is an atomic proposition that can hold or not in a state of the L^2TS and means that the service is able to accept a request for the interaction *engineFailure*.
2. $AG\,[request(garage, \langle \$car \rangle)]\, AF_{response(garage,\langle \%car \rangle) \vee fail(garage,\langle \%car \rangle)}\, true$
   This formula means that all garage services contacted by *Orchestrator* are *responsive*, i.e. they always guarantee a response to each received request. Indeed, a formula like $[\gamma]\,\phi$ means that in the next state of any path, reached by an action satisfying the action formula γ, the formula ϕ holds; a formula like $AF_\gamma\,\phi$ holds in a state q if, and only if, ϕ holds in q or in one of the states reachable from q (by a last action satisfying γ) along each path starting from q. Notably, responses (both positive and negative) from the contacted garage service belong to the same interaction *garage* of the garage appointment request and are correlated by the variable *car*.
3. $\neg E(true_{\neg\, response(charge)} U_{request(garage) \vee request(rentalCar)} true)$
   This formula means that a garage or a rental car request can be processed only after the driver's credit card has been successfully charged. Indeed, $\neg$ is the negation operator and $E(\phi\ {}_\chi U_\gamma\ \phi')$ is the *until* operator, that means that there exists a path starting from the current state for which ϕ' holds at the starting state or at a future state (reached by an action satisfying γ), and ϕ has to hold until that state is reached (by executing unobservable actions or actions satisfying χ). Notably, some of the previously used operators can be derived from the until operator: $EF\,\phi$ stands for $E(true\ {}_{tt} U_{tt}\, \phi)$, where *tt* is the action formula always satisfied, $AG\,\phi$ stands for $\neg EF \neg \phi$, $AF_\gamma\, true$ stands for $A(true\ {}_{tt} U_\gamma true)$, and $EF_\gamma\, true$ stands for $E(true\ {}_{tt} U_\gamma true)$.
4. $EF_{response(rentalCar,\langle \$rentalNum \rangle)}\, EF_{fail(towTruck)}\, AF_{cancel(rentalCar,\langle \%rentalNum \rangle)}\, true$
   This formula means that, if renting a car succeeds and finding a tow truck fails,

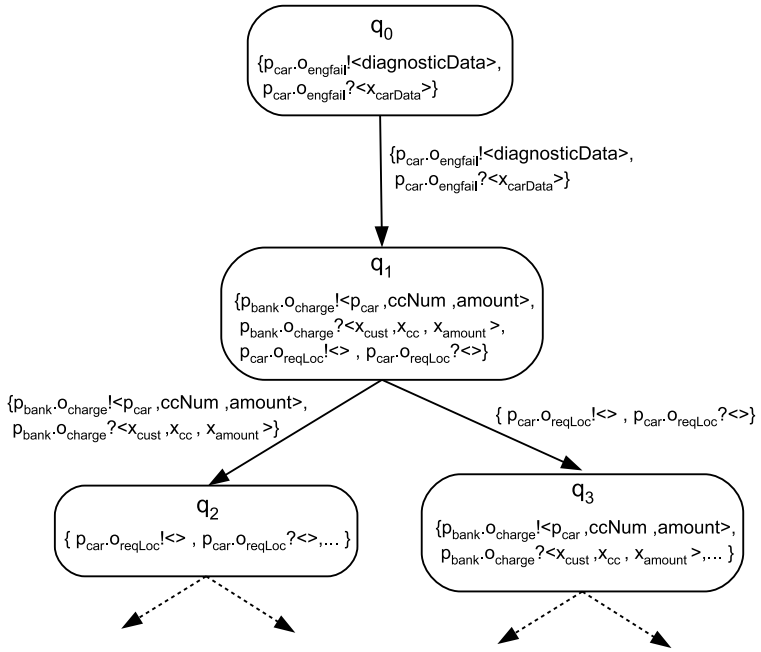

Fig. 2. Excerpt of the concrete L^2TS for the on road assistance scenario

then the rental car order must be cancelled (because the car must be redirected to the driver's current location). Notably, the cancelling request belongs to the same interaction *rentalCar* of the rent confirmation and they are correlated by the variable *rentalNum*.

5. $EF_{fail(rentalCar)}\, EF_{response(towTruck)}\, true$
   This formula means that if renting a car fails, tow truck (and, therefore, garage appointment) can succeed.
6. $AG\,[fail(towTruck)]\, AF_{cancel(garage)}\, true$
   This formula means that if finding a tow truck fails, the garage appointment will be revoked.
7. $\neg E(true_{\neg\, response(garage)} U_{request(towTruck)} true)$
   This formula means that before looking for a tow truck, a garage must be found.

To check if a COWS term enjoys some abstract properties expressed as SocL formulae, the following steps must be performed. Firstly, the LTS defining the semantics of the COWS term (see [15] for a commented presentation of the LTS) is transformed into an L^2TS by labelling each state with the set of actions the COWS term is able to perform immediately from that state. Of course, the transformation preserves the structure of the original COWS LTS. For example, the concrete L^2TS obtained by applying this transformation to the on road assistance scenario is shown in Figure 2. Notably, in our L^2TS arcs are labelled by set of actions, rather than by single actions as it is usual.

Secondly, since we are interested in verifying abstract properties of services, such as those shown before, we need to abstract away from unnecessary details by transforming

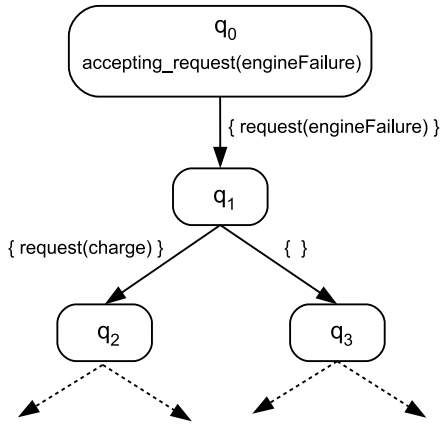

Fig. 3. Excerpt of the abstract L^{2}TS for the on road assistance scenario

concrete actions into abstract ones. This is done by means of suitable *abstraction rules* that replace the concrete labels on the transitions with abstract actions (i.e. *request*(i, c), *response*(i, c), *cancel*(i, c) and *fail*(i, c)) and the concrete labels on the states with atomic propositions (such as, e.g., *accepting_request*(i)). The transformation only involves the concrete actions we want to observe; the concrete actions that are not replaced by their abstract counterparts may not be observed. Thus, the application of the abstraction rules transforms the concrete L^{2}TS into an 'abstract' one. For example, the abstract L^{2}TS of the on road assistance scenario shown in Figure 3, is obtained by applying to the concrete L^{2}TS of Figure 2 the following abstraction rules:

$$
\begin{aligned}
&\textit{Action}: \; o_{engfail} \rightarrow \textit{request}(\textit{engineFailure})\\
&\textit{Action}: \; o_{charge} \rightarrow \textit{request}(\textit{charge})\\
&\textit{Action}: \; o_{chargeOK} \rightarrow \textit{response}(\textit{charge})\\
&\textit{Action}: \; p_{garage\_1} \cdot o_{order}\langle \$1, * \rangle \rightarrow \textit{request}(\textit{garage}, \langle \$1 \rangle)\\
&\textit{Action}: \; p_{garage\_2} \cdot o_{order}\langle \$1, * \rangle \rightarrow \textit{request}(\textit{garage}, \langle \$1 \rangle)\\
&\textit{Action}: \; \$1 \cdot o_{garageOK} \rightarrow \textit{response}(\textit{garage}, \langle \$1 \rangle)\\
&\textit{Action}: \; \$1 \cdot o_{garageFail} \rightarrow \textit{fail}(\textit{garage}, \langle \$1 \rangle)\\
&\textit{Action}: \; o_{cancel} \rightarrow \textit{cancel}(\textit{garage})\\
&\ldots\\
&\textit{Action}: \; o_{rentalCarOK}\langle \$1 \rangle \rightarrow \textit{response}(\textit{rentalCar}, \langle \$1 \rangle)\\
&\textit{Action}: \; o_{redirect}\langle \$1, * \rangle \rightarrow \textit{cancel}(\textit{rentalCar}, \langle \$1 \rangle)\\
&\textit{State}: \; o_{engfail} \rightarrow \textit{accepting\_request}(\textit{engineFailure})
\end{aligned}
$$

Most of the rules are self-explicative, we comment on the remaining ones. Variables "$*n*" (with *n* natural number) are used to define parametric abstraction rules. Also the wildcard " $*$ " is used for increasing flexibility. The fourth and fifth rules prescribe that whenever an action over the endpoints $p_{garage\_1} \cdot o_{order}$ or $p_{garage\_2} \cdot o_{order}$ with sent data $\langle p_{car}, data \rangle$ (that match $\langle \$1, * \rangle$) occurs in the label of a transition, then it is replaced by the abstract action $request(garage, \langle p_{car} \rangle)$. This way, the car partner name p_{car} can be used to correlate responses from the contacted garage service. Similarly, the second-last rule prescribes that whenever an action over the operation $o_{redirect}$ with

sent data $\langle rentalNum, gps \rangle$ occurs in the label of a transition, then it is replaced by $cancel(rentalCar, \langle rentalNum \rangle)$. The last rule works similarly, but it applies to labels of states rather than to labels of transitions.

Finally, the SocL formulae are checked over the abstract L^2TS. To assist the verification process, one can use CMC [1], that is a model checker for SocL formulae over L^2TS, other than an interpreter for COWS terms. One can thus verify that, as expected, all the abstract properties we introduced before do hold for the COWS specification of on road assistance scenario, but the first property, because *Orchestrator* is not a persistent service capable of accepting and serving multiple requests (indeed, as we noted in Section 3.2, it can only perform one recovery task at a time).

7 Concluding remarks

COWS falls within a main line of research that aims at developing process calculi capable of capturing the basic aspects of service-oriented systems and, possibly, of supporting the analysis of qualitative and quantitative properties of services. We have demonstrated that one can use COWS to model all the phases of the life cycle of SOC applications such as publication, discovery, negotiation, orchestration, deployment, reconfiguration and execution. We believe that the methods and tools we have described for expressing and checking properties of services are already an important added value of using COWS as a modelling language.

The fact that several relevant aspects of SOC systems can be suitably addressed and dealt with in an homogeneous and direct way by using a single linguistic low-level formalism somehow suggests that COWS could serve as a common and convenient basis to enable analysis of service-oriented applications by translation from higher level languages. As further steps in this direction, we are currently studying translations from the service orchestration language WS-BPEL [22] and the SENSORIA Reference Modelling Language SRML [10] into COWS. A short-term goal of this activity is to define, via translation in COWS, an operational semantics for these two high level languages. A long-term goal is to enable using proof techniques and analytical tools developed for COWS, such as the type system and the logical verification framework summed up in this paper, and the stochastic extension defined in [23], to analyse service-oriented applications programmed in WS-BPEL or modelled in SRML.

Acknowledgements. We thank Alessandro Fantechi, Stefania Gnesi and Franco Mazzanti for their contribution on the development of SocL and CMC. The anonymous referees provided helpful suggestions for improving the presentation.

References

1. CMC: an on-the-fly model checker and interpreter for COWS.
   Available at `http://fmt.isti.cnr.it/cmc/`.
2. Software Engineering for Service-Oriented Overlay Computers (SENSORIA), 2005.
   Web site: `http://sensoria.fast.de/`.

3. M.H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An action/state-based model-checking approach for the analysis of communication protocols for Service-Oriented Applications. In *FMICS*, LNCS. Springer, 2007. To appear.
4. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.
5. M. Buscemi and U. Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In *ESOP*, volume 4421 of *LNCS*, pages 18–32. Springer, 2007.
6. R. De Nicola, D. Gorla, and R. Pugliese. Confining data and processes in global computing applications. *Science of Computer Programming*, 63:57–87, 2006.
7. R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995.
8. R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In *Proc. Ecole de Printemps on Semantics of Concurrency*, volume 469 of *LNCS*, pages 407–419. Springer, 1990.
9. A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach for verifying COWS specifications. In *FASE*, LNCS. Springer, 2008. To appear.
10. J. Fiadeiro, A. Lopes, and L. Bocchi. A formal approach to service component architecture. In *WS-FM*, volume 4184 of *LNCS*, pages 193–213. Springer, 2006.
11. H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD*, pages 249–259. ACM Press, 1987.
12. Z. D. Kirli. Confined mobile functions. In *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, pages 283–294. IEEE Computer Society, 2001.
13. N. Koch. Automotive case study: UML specification of on road assistance scenario. Technical Report 1, FAST, 2007. Available at `http://rap.dsi.unifi.it/sensoria/files/FAST_report_1_2007_ACS_UML.pdf`.
14. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services (full version). Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze, 2006. Available at `http://rap.dsi.unifi.it/cows`.
15. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *ESOP*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.
16. A. Lapadula, R. Pugliese, and F. Tiezzi. COWS specification of the on road assistance scenario. Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze, 2007. Available at `http://rap.dsi.unifi.it/cows`.
17. A. Lapadula, R. Pugliese, and F. Tiezzi. C$^{\odot}$WS: A timed service-oriented calculus. In *ICTAC*, volume 4711 of *LNCS*, pages 275–290. Springer, 2007.
18. A. Lapadula, R. Pugliese, and F. Tiezzi. Regulating data exchange in service oriented applications. In *FSEN*, volume 4767 of *LNCS*, pages 223–239. Springer, 2007.
19. A. Lapadula, R. Pugliese, and F. Tiezzi. Service discovery and negotiation with COWS. In *WWV*, ENTCS. Elsevier, 2007. To appear.
20. L.G. Meredith and S. Bjorg. Contracts and types. *Commun. ACM*, 46(10):41–47, 2003.
21. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Inf. Comput.*, 100(1):1–40, 41–77, 1992.
22. OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, April 2007. Available at `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.
23. D. Prandi and P. Quaglia. Stochastic COWS. In *ICSOC*, volume 4749 of *LNCS*, pages 245–256. Springer, 2007.
24. F. van Breugel and M. Koshkina. Models and verification of bpel. Technical report, 2006. Available at `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf`.
25. R.J. van Glabbeek. On specifying timeouts. *ENTCS*, 162:173–175, 2006.