

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/224323627>

Service-Oriented Modelling of Automotive Systems

Conference Paper · September 2008

DOI: 10.1109/COMPSAC.2008.228 · Source: IEEE Xplore

CITATIONS

28

READS

387

3 authors:



[Laura Bocchi](#)

University of Kent

52 PUBLICATIONS 1,005 CITATIONS

[SEE PROFILE](#)



[José Luiz Fiadeiro](#)

University of Dundee

301 PUBLICATIONS 4,177 CITATIONS

[SEE PROFILE](#)



[Antónia Lopes](#)

University of Lisbon

130 PUBLICATIONS 2,165 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



Tracing Networks [View project](#)



Time-sensitive protocol design and implementation [View project](#)

Service-Oriented Modelling of Automotive Systems

Laura Bocchi
Department of Computer Science
University of Leicester
Leicester LE1 7RH
United Kingdom

bocchi@mcs.le.ac.uk

José Luiz Fiadeiro
Department of Computer Science
University of Leicester
Leicester LE1 7RH
United Kingdom

jose@mcs.le.ac.uk

Antónia Lopes
Departamento de Informática
Fac. Ciências, Universidade de Lisboa
Campo Grande, 1749-016 Lisboa
Portugal

mal@di.fc.ul.pt

ABSTRACT

We discuss the suitability of service-oriented computing for the automotive domain. We present a formal high-level language in which complex automotive activities can be modelled in terms of core components and services that can either be provided by other components of the on-board software system or procured from external providers (e.g. via the web) through a negotiation process that involves quality of service attributes and constraints. We argue that the ability to re-configure activities, in real-time, through service discovery and dynamic binding takes us one step further from current component-based development techniques: it enhances flexibility and adaptability to changes that occur in the environment in which the system operates (driver, automobile, and external circumstances) and, ultimately, leads to improved levels of satisfaction, safety and reliability.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Techniques and Tools – *modules, interfaces*.

General Terms

Design, Languages, Theory.

Keywords

Services: Composition, Discovery, Selection, Binding.

1. INTRODUCTION

Service-Oriented Computing (SOC) is emerging as a preferred paradigm for the development of systems that can take advantage of the new modes of computation that are being made available based on code and data mobility over wide area networks (what is usually called Global Computing), enabling the flexible interconnection of autonomously developed and operated applications discovered in real-time according to required levels of service. In this paper we discuss the suitability of SOC to the automotive domain, namely the way it can enhance the adaptability of automotive systems to the properties of the environment in which they operate (driver, automobile, and external circumstances) and, ultimately, improve levels of user satisfaction, safety and reliability.

We present and discuss results and experiences gathered during the IST-FET integrated project SENSORIA [15], namely in using the SENSORIA Reference Modelling Language (SRML) in an automotive case study. SRML is a high-level modelling language with a mathematical semantics in which complex services can be modelled using concepts and primitives that are independent of the technologies that provide the middleware infrastructure over which services can be deployed, published and discovered. In this sense, our work is very different from the languages and notations that have been proposed for supporting the description of web

services and their composition – WSDL, BPEL4WS, WSPOLICY, inter alia – which are tailored to the technological infrastructure that is currently enabling web services [1]. In the case of this paper, this is reflected in the fact that we will concentrate on the way the principles of SOC in general (and not just existing web-service technologies) can be used in modelling systems for the automotive domain and the benefits that can ensure for high-level properties of such systems.

This is why, in the rest of this section and throughout the paper, we compare SOC with what is still the prevailing approach to automotive system design: component-based development (CBD). This comparison is made not at the level of the specific languages used in SOC or CBD, but of the methodological and architectural principles that distinguish them. In fact, we will see that SRML offers primitives for building systems based on the use of both components and services, thus getting the best of both worlds.

The paper is organised as follows. Section 2 discusses the way we see SOC to contribute beyond CBD in general, and automotive applications in particular. Section 3 presents the case study that is used in the rest of the paper. Section 4 presents the notion of module, the cornerstone of our modelling approach – SRML. Section 5 presents the language primitives that are used in SRML for modelling the static aspects of service behaviour and composition, and Section 6 addresses the dynamic aspects.

2. SERVICES vs COMPONENTS

The term ‘service’ is being widely used in software engineering with a variety of meanings, which suggests that we make clear from the very beginning what precise aspects we are capturing with SRML. One can find in [6] a comprehensive overview of some of the concepts of service that have been around, and a proposal for a (formal) model of services based on CBD. It so happens that [6] uses an automotive case study to illustrate the proposed approach, which suggests that it is indeed a good choice of a CBD approach for comparing with SRML.

Starting from a universe of (software) components as “structural entities”, CBD (in the sense of [6]) views a service as a way of orchestrating interactions among a subset of components in order to obtain some required functionality – “services coordinate the interplay of components to accomplish specific tasks”. Hence, a car can offer several different services (e.g. unlocking doors, storing driver’s identifier, fetching the tuner’s presets from a database) using a fixed pool of components (a control unit, a key fob for remote and mechanical entry that also stores the driver’s identification, a lock manager, a lighting system, a tuner, a database of presets, and a user interface for operating the tuner). In summary, services are seen as “crosscutting elements of the system under consideration”, describing “partial views on the set of components in the system under consideration” [6].

SOC differs in that there is no “system under consideration”, conceived *a priori*, that services crosscut. If we take one of the accepted meanings of ‘system’ – *a combination of related elements organised into a complex whole* – we can see why it is not directly applicable to SOC: services in the sense of SOC get combined at run time and redefine the way they are orchestrated as they execute; no ‘whole’ is given *a priori* and services do not compute within a fixed configuration of a ‘universe’.

Indeed, the very basic difference between SOC and CBD (which we share with [7]) is in the fact that, in SOC, we are dealing with run-time, not design-time complexity. In CBD, selecting the components that deliver a service is a design time activity: the way the control unit, the key fob and the lock manager together unlock the doors is fixed when the on-board system is designed and installed. If a new lock manager is installed (say to comply with new safety legislation), the software that provides the unlock-doors service needs to be redesigned and implemented. One can also program this selection in the sense that more than one orchestration of the existing components may deliver the service, but the universe in which this selection is made is fixed.

Instead, SOC provides a means of obtaining functionalities by orchestrating interactions among components that are *procured at run time* according to given (functional) types and service level constraints from a universe that is not fixed *a priori*. Discovery and selection do not need to be programmed: these activities are provided by the underlying middleware (SOA) from a set of services (and service providers) that is itself dynamically changing. This means that, when designing a system, we can abstract from the identity of the components that will provide required services and the mechanisms that will be responsible for discovering and binding these services to the running system.

The added flexibility provided through SOC comes at a price – dynamic interactions have the overhead of selecting the co-party at each invocation – which means that the choice between invoking a service and calling a component needs to be carefully justified. This is why SRML makes a provision for both types of interaction as explained in the paper. For instance, in automotive systems, this dynamic aspect of SOC may not (always) make sense: the key fob, the tuner, the database of presets, and so on, are all there in the car and do not need to be discovered as the driver operates the key (drivers are known to lose keys and tuners sometimes get stolen, but such situations are not part of the engineering process).

However, one could well imagine that, as the driver presses the key, the onboard control system detects that the weather conditions are particularly adverse and procures a component for the ABS that can optimise the car’s performance in the new situation. This procurement is performed according to criteria and rules that will have been decided at design time, but the binding of the components that provide the selected service to the onboard system will take place at run time (which does not mean that it takes place while the car is running ;-). There is no *a priori* knowledge of the structure of the discovered service and, hence, of the way the run-time configuration of the onboard system will change as a result of the binding. However, as explained in the paper, a modelling language like SRML can offer formal mechanisms through which one can ascertain that global properties will be enforced regardless of the way the configuration changes.

The increasing availability of sensor-networks helps explain why SOC can enhance adaptability: each time an activity needs to run, it can bind to the services that offer the best properties for the en-

vironment in which it is operating (driver, automobile, and external circumstances). Having said this, one has to recognise that these capabilities of SOC as a paradigm are not always fully exploited by current Web/Grid-based technologies, especially in their ability to cope with the increasing complexity of sensor networks and other aspects of the technological platforms in which services will operate. In particular, one of the aims of SRML (and SENSORIA in general) is to provide a modelling framework that is abstract enough to tame this level of (run-time) complexity. Ultimately, developers of systems that, like in the automotive domain, have safety-critical aspects will need to *trust* service providers. That is, there are aspects that the underlying service-oriented architecture will need to guarantee for SOC to be effectively used. As already mentioned, we do not address such aspects in this paper: SRML is concerned with modelling systems over platforms that are assumed to deliver the required architectural elements (registries, discover mechanisms, and so on) and levels of interconnectivity and trust.

3. THE ON-ROAD REPAIR CASE STUDY

A class of automotive scenarios in which the advantages of SOC are perhaps clearer concerns applications that span over the boundaries of a single vehicle. This may happen when we want a car to interact with external parties, for example another vehicle, a hotel or restaurant at the destination, or an emergency service.

The case study that we consider in this paper involves an activity, embedded in a vehicle, that handles engine failure as triggered by a sensor. When the activity is triggered, the system (1) determines the current location of the car by using a GPS component, (2) searches for the garage that is closest to the current location and can ensure minimal levels of repair as required for the problem detected in the engine and calls a tow truck and (3) contacts a car rental service near the garage.

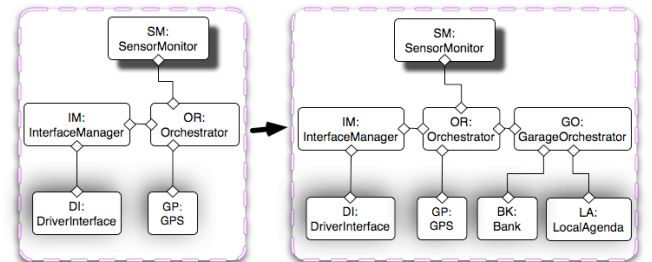


Figure 1: Reconfiguration of the *OnRoadRepair* activity

The activity, in its initial configuration as it is triggered by the sensor, involves the following software entities: *SM* (the sensor that triggers the activity), *GP* (the GPS system), *DI* (the interface to the driver), *OR* (the orchestrator that coordinates the interactions with the external services and *GP*), *IM* (the component that manages the interactions with the driver through *DI*). These entities are interconnected through wires, each of which defines an interaction protocol between two entities. Typically, wires are required for dealing with the heterogeneity of partners involved in the activity by performing data (or, more, generally, semantic) integration, which is useful for instance when a car has to travel across different countries.

In addition, the activity relies on a number of external services that will be discovered if and when required according to given constraints: (1) the service for booking a garage and for calling a tow-truck and (2) the service for booking a car rental. This dependency on external services is made explicit in the module *On-*

RoadRepair that constitutes the type of the activity in its initial configuration. This module is shown in Figure 2. As the components execute as part of the configuration, the discovery and binding of external services identified in the activity module may be triggered, leading to a reconfiguration of the activity. This reconfiguration is depicted in the right-hand side of Figure 1 for the case of the discovery of a service that matches the requires-interface *GA* (garage), e.g., *RepairService* in Figure 3. The new configuration is obtained by adding the components and wires that provide the required service and wires that connect them to components of the old configuration.

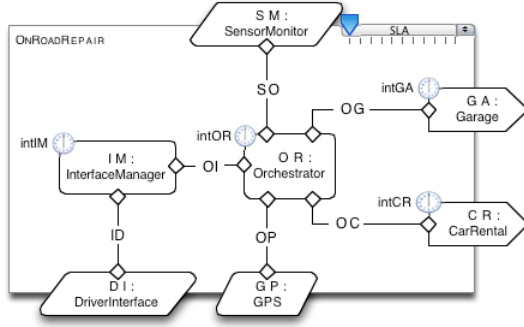


Figure 2: The activity module *OnRoadRepair*

The shadows in Figure 1 represent the fact that, in SRML, components may reside in three different layers. Layers are architectural abstractions that reflect different levels of organisation and change. *SM* belongs to the top layer; it is the software component that triggers the creation and *uses* the activity. The middle layer is the one that evolves as services are discovered and bound. When the *OnRoadRepair* activity is launched, (new instances of) the components *IM* and *OR* are added to the middle layer with the aim of orchestrating the interactions with the services to be discovered (e.g., *GA*) or resources belonging to the bottom layer.

Entities of the bottom layer provide services in the CBD sense. They are persistent as far as the life cycle of the activities is concerned, and can be shared by multiple instances of the same activity. For example, the driver interface *DI* and GPS *GP* components can be used by other activities of the on-board system or other instances of the *OnRoadRepair* activity. The entities in the bottom layer can also be used to ensure that the effects of an activity will persist beyond its lifetime, for instance by updating a database. In the rest of the paper, we present the primitives of SRML through which activities and services can be modelled, and discuss the process of reconfiguration.

4. MODULES AS UNITS OF DESIGN

The unit of design in SRML is the *module*. A module is specified in terms of a number of entities and the way they are interconnected. Figure 2 illustrates the activity module *OnRoadRepair*. SRML also offers a textual notation:

```

MODULE OnRoadRepair(carID:vehicleId)
  carID is the identifier of the
  car in which the activity will
  run, which is passed as a
  parameter when the activity
  is launched.

  SERVES
    SM: SensorMonitor

  COMPONENTS
    OR: Orchestrator(carId)
      intOR ⚙️ init: s=INIT
      intOR ⚙️ term: s=FINAL
      initialization and termination
      conditions for OR
    IM: InterfaceManager

```

USES

GP: GPS
DI: DriverInterface

REQUIRES

GA: Garage
intGA ⚙️ **trigger**: default
CR: CarRental
intCR ⚙️ **trigger**: bookGarage ⚙️!

the discovery of *GA* is triggered by the first interaction with *GA*.

EXTERNAL POLICY

(see Section 6)

the discovery of *CR* is triggered by the request to book the garage.

WIRES

OR Orchestrator	c_3	OG $pORGA$ (carID)	d_3	GA Garage
bookGarage	S		R	acceptBooking
collectPnt	i		i_1	carID
servicePrice	o		i_2	collectPnt
card	c		o	servicePrice
			c	card

...

Every activity module declares interfaces of four possible kinds:

- One (and only one) *serves-interface* that binds the activity to a component in the top layer. This interface is typed by a *layer protocol* that specifies the interactions that the top layer component (the user of the activity) can maintain with the (service-layer) components that orchestrate the behaviour of the activity. In *OnRoadRepair*, the serves-interface binds to the sensor that detects engine failures.
- A number of *uses-interfaces* (possibly none) that bind to components in the bottom layer of the configuration, such as a GPS or a component that provides an interface to an external entity like the driver in *OnRoadRepair*. Uses-interfaces are also typed by layer protocols.
- A number of *component-interfaces* (at least one) that bind to service-layer components that are created when the activity is triggered by the top-layer component. The service-layer components interact with each other and all the other components in the top and bottom layer that are bound to the corresponding serves and uses-interfaces. Bottom-layer components are not created by the module; they already exist when the activity is launched and bind to the corresponding uses-interfaces at that moment. Component-interfaces are typed by *business roles* as discussed in Section 5.
- A number of *requires-interfaces* (possibly none) that bind the activity to services that are procured externally when triggered by component execution. Requires-interfaces are typed by *business protocols* as discussed in Section 5.


The workflow of the activity is defined collectively by the components in its configuration and the wires that connect them. The possibility of defining the business process in terms of different components and wires facilitates modular development, reflecting the structure of the business domain.

In a module, *wire-interfaces* are typed by connectors. A connector consists of an interaction protocol (e.g., the interaction protocol *pORGA* for the wire *OG* presented in Section 5) and two attachments that link the roles of the protocol (c_3 and d_3) to the interfaces that are being interconnected (*OR* and *GA*, respectively).

Finally, every module also defines:

- An internal configuration policy (indicated by the symbol ⚙️) that identifies the triggers of the external service dis-

covery process, and the initialization and termination conditions of the components.

- An external configuration policy (indicated by the symbol ) that consists of the variables and constraints that determine the quality profile of the activity to which the discovered services need to adhere.

The configuration policies (both internal and external) are discussed in Section 6.

Activities reconfigure their workflows at run time by triggering and binding to services after a discovery, ranking and selection process. Services are modelled in SRML through service modules, which are like activity modules except that they include a ‘provides-interface’ through which customers can connect to the service. Like requires-interfaces, provides-interfaces are typed by business protocols that describe the properties that a customer can expect from the interactions with that service. On the other hand, service modules do not include a serves-interface for the top layer: interactions with the top layer are performed exclusively by activities; services compose only horizontally. Uses-interfaces can be included in service modules in the same way as for activities.

Figure 3 presents *RepairService*, one of the possible service modules that could bind with the interface *GT* of *OnRoadRepair*. The module relies on the bottom layer entities *BK*, which binds to the bank through which the garage handles payments, and *LA* which binds to the agenda of the repair service. When according to this agenda there are no locally available tow trucks, the service may decide to invoke an external service for hiring a tow truck, which is represented by the interface *TT*.

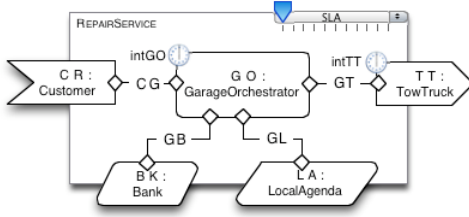


Figure 3: The service module *RepairService* through which a garage and a tow truck can be booked

5. MODULE SPECIFICATION

All interfaces are formal specifications that involve a *signature* declaring the set of supported interactions, which can be of several different types:

- r&s**— The co-party initiates the interaction and expects a reply. It does not block while waiting for the reply.
- s&r**— The party initiates the interaction and expects a reply. It does not block while waiting for the reply.
- rcv**— The co-party initiates the interaction and does not expect a reply.
- snd**— The party initiates the interaction and does not expect a reply.
- ask**— The party synchronises with the co-party to obtain data.
- rpl**— The co-party synchronises with the party to send data.
- tll**— The party requests the co-party to perform an action and blocks until the action is done.
- prf**— The party performs an action and frees the co-party.

Interactions of type *r&s* or *s&r* are durative/conversational. We distinguish several events that can occur during such interactions:

- $\text{interaction} \triangleleft$ is the event that initiates *interaction*.
- $\text{interaction} \boxtimes$ is the reply-event of *interaction*.
- $\text{interaction} \checkmark$ is the commit-event of *interaction*.
- $\text{interaction} \times$ is the cancel-event of *interaction*.
- $\text{interaction} \dagger$ is the revoke-event of *interaction*.

All interactions can have parameters for transmitting data when they are initiated, declared as \triangleleft . Conversational interactions can additionally have parameters for carrying a reply, declared as \boxtimes and for carrying data if there is a commit, a cancel or a revoke, declared as \checkmark , \times and \dagger respectively. A mathematical semantics of these operators can be found in [1].

Each specification – *business roles*, *business protocols*, *layer protocols* and *interaction protocols* – provides a slightly different style of behavioural description. A business role defines an execution pattern involving the interactions that it declares in its signature, what we call an orchestration. The orchestration is defined in terms of local variables that store data and model the abstract state of the component, and a number of guarded transitions. Every transition may cause a change in the local state, specified as *effects*, and trigger a number of interaction events, specified as *sends*. The orchestration of the service provided by the module is the composition of the orchestrations defined within its components and the way they are wired together.

The following is an excerpt of the business role *Orchestrator*:

BUSINESS ROLE *Orchestrator*(carID:vehicleId) **is**

INTERACTIONS

```

tll handleEngineFailure
ask currentLocation:location
s&r askUsrDetails
   $\boxtimes$  cust:customerId
    card:paydata
    workRelated:boolean
    destination:location
    apointmentTime:time
s&r bookGarage
   $\triangleleft$  collectPnt:location
   $\boxtimes$  servicePrice:moneyVal
   $\checkmark$  card:paydata
rcv confirmation
s&r rentACar
   $\triangleleft$  cust:customerId
    card:paydata

```

ORCHESTRATION

```

local s: [ INIT, FAILURE_DETECTED, CONTEXT_RECEIVED, CAR,
  GARAGE&TT, CAR+GARAGE&TT, FINAL ], l:location

```

transition StartProcess

```

triggeredBy engineFailure
guardedBy s=INIT
effects s'=FAILURE_DETECTED
   $\wedge$  l'=currentLocation
sends askUsrDetails $\triangleleft$ 

```

When a failure is detected in the initial state, the current location is recorded and the user is asked for details.

transition getContextData

```

triggeredBy askUsrDetails $\boxtimes$ 
guardedBy s=FAILURE_DETECTED
effects s'=CONTEXT_RECEIVED
sends bookGarage $\triangleleft$ 
   $\wedge$  bookGarage.collectPnt=l

```

When a reply to a request for user details is received after a failure has been detected, a garage is booked and the current location is transmitted.

Requires and provides-interfaces are typed by business protocols. The difference between business protocols and roles is that, instead of an orchestration, they provide a set of properties that describe the behaviour that can be expected of the service (in case of provides-interface) or specify the behaviour that is expected (in

the case of requires-interface) of the external party. For instance, the type of the requires-interface *GA* is the business protocol *Ga*–*rage* specified as follows:

BUSINESS PROTOCOL Garage is	
INTERACTIONS	
r&s acceptBooking	The acceptance of a booking is enabled from the beginning of the session until it occurs.
🔔 carId:vehicleId	
collectPnt:location	
✉ servicePrice:moneyVal	
✓ card:paydata	A confirmation will be sent after a commitment is received for the booking.
snd confirmation	
BEHAVIOUR	
initiallyEnabled acceptBooking🔔?	
acceptBooking✓? ensures confirmation!🔔	

Typically, uses and servers interfaces involve synchronous interactions as in:

LAYER PROTOCOL GPS is	
INTERACTIONS	
tll where:location	
BEHAVIOUR	

In this case, we do not make any special requirement on the GPS component that we wish *OnRoadRepair* to bind to.

As already mentioned, wires are typed by connectors defined in terms of business protocols and attachments. Because interaction protocols establish a relationship between two parties, the interactions in which they are involved are divided in two subsets called *roles* – *A* and *B*. The “semantics” of the protocol is provided through a collection of properties that establish how the interactions are coordinated, which may include routing events or transforming sent data to the format expected by the receiver. This is precisely what happens in the wire *OG* connecting the orchestrator and the garage as presented in Section 4. The garage expects a car identifier, which the interaction protocol sets to the value of the parameter *carId* of the module.

INTERACTION PROTOCOL pORGA(id:vehicleId) is			
ROLE A		ROLE B	
snd S		rcv R	
Ⓐ i:location		Ⓐ i ₁ :vehicleId	
ⓧ o:moneyVal		i ₂ :location	
✓ c:paydata		ⓧ o:moneyVal	
		✓ c:paydata	
COORDINATION		all interaction parameters are shared except for <i>i₁</i> : <i>vehicleId</i> which is set to the protocol parameter id.	
R = S			
R.i ₁ =id			
R.i ₂ =S.i			
R.o=S.o			
R.c=S.c			

6. CONFIGURATION POLICIES

SRML offers primitives for modelling the dynamic aspects concerned with session management and service level agreement, which together we call *configuration policies*.

The *internal configuration policy* defines:

- For each component-interface, a (initialisation) condition $\textcircled{\text{init}}$ that holds when the session is initiated and a (termination) condition $\textcircled{\text{term}}$ that determines when the corresponding component will no longer have to be available for interactions (and may thus be removed from the configuration). For example, the initialisation of *OR* sets the state variable *s* to the value *INIT* and termination occurs when the final state *s*=*FINAL* is reached.

- For each requires-interface, a (trigger) condition $\textcircled{\text{trigger}}$, specifies the condition that launches the discovery process for that requires-interface. The default trigger (as in *GA*) launches the discovery when the first interaction declared in the interface is required. The trigger of *CR* launches the discovery on the occurrence of the first interaction of *OR* with *GA* – *bookGarage*Ⓐ?

The *external configuration policy* concerns the constraints that the process of discovery, negotiation and binding must satisfy to establish service level agreements (SLA) with service providers. In SRML, we use the algebraic approach developed in [4] for constraint satisfaction and optimization.

In the case of *OnRoadRepair*, we use a constraint system where the degree of satisfaction has fuzzy values, i.e. it takes value in the interval $[0,1]$. In order to define the constraints that we wish to apply to the module, we use the following SLA variables: *GA.LOCATION* – the location of the garage, *CR.LOCATION* – the location of the car rental, and *CR.COST* – the cost of the car rental service. We consider three constraints:

- CheapRent* applies to the cost of the car rental service. The best degree of satisfaction is when the trip is work-related (i.e., the driver will be reimbursed) otherwise it is inversely proportional to the cost. Information about whether the trip is work-related is provided by the value of the interaction parameter *OR.askUsrDetails.workRelated*.
- Closeness₁* and *Closeness₂* minimize the distance between garage and destination, and between car rental and garage, respectively. The destination and location are provided by *OR.askUsrDetails.destination*, *OR.bookGarage.collectPnt*. The best degrees of satisfaction are achieved when the distances between garage-destination and garage-car rental are less than 20. Otherwise they are inversely proportional to the distance. The function *distance* returns the distance between two locations.

EXTERNAL CONFIGURATION POLICY

SLA VARIABLES

GA.LOCATION, CR.LOCATION, CR.COST

CONSTRAINTS

CheapRent: {CR.COST}

$$def_1(c) = \begin{cases} 1 & \text{if } OR.askUsrDetails.workRelated \\ 1000/c & \text{otherwise} \end{cases}$$

Closeness₁: {GA.LOCATION}

$$def_2(g) = \begin{cases} 1 & \text{if } \Delta(g) < 20 \\ 100/\Delta(g) & \text{otherwise} \end{cases}$$

$$\Delta(g) = distance(g, OR.askUsrDetails.destination)$$

Closeness₂: {CR.LOCATION, GR.LOCATION}

$$def_3(c, g) = \begin{cases} 1 & \text{if } distance(c, d) < 20 \\ 100/distance(c, d) & \text{otherwise} \end{cases}$$

For each potential garage and car rental partner, the set of constraints has to be solved. The solution assigns a degree of satisfaction to each possible tuple of values for the SLA variables. Negotiation in our framework consists in finding an assignment that maximizes the degree of satisfaction. Hence, the outcome of the negotiation between *OnRoadRepair* and the potential partner is any tuple that maximizes the degree of satisfaction. Selection then picks a partner with a service level agreement that offers the best degree of satisfaction.

7. CONCLUDING REMARKS

The main goal of this paper was to put forward the case for service-oriented computing to be used in the automotive domain as a means of enhancing flexibility and adaptability of systems to changes that occur in the domain in which they operate (driver, automobile, and external circumstances). More specifically, whereas in component-based development services are defined statically at design time (e.g. as in [6]), we claim that SOC lets services to be procured and bound at run time so as to optimise the level of satisfaction of a constraint system that captures quality of service properties.

For this purpose, we used a language – SRML – that we are defining in the context of the SENSORIA project for service-oriented modelling. SRML is inspired by recent work on the Service Component Architectures (SCA) [14] and Web Services [3], but is more abstract than the typical languages that one finds for web services – e.g. WSDL and BPEL – in the sense that it abstracts away the behaviour of services from the way the components that provide them are programmed and interact with each other. For instance, a specific mapping has now been defined from BPEL to SRML [5].

A distinctive feature of SRML is that it supports, in an integrated way, both CBD and SOC-based service provision. Indeed, the proposed notion of module and the separation of components in three architectural layers have the advantage that they clearly separate the service-oriented (horizontal) layer from the component-based (vertical) one while working on a uniform modelling language and environment. In this respect, there are clear advantages in using declarative languages such as SRML over formalisms based on, say Petri-Nets [13], in the sense that SRML offers a higher-level, more domain-level support for modelling.

SRML also offers the advantage of having a mathematical semantics covering the static aspects of composition [9], its dynamic aspects [10], as well the computational aspects that relate to the way sessions are executed [1]. The latter are being mapped into a logic adapted from $\mu UCTL$, a formalism being developed within SENSORIA for supporting qualitative analysis [11]. The aim is to provide (1) a notion of correctness for service modules (i.e., the properties of a provides-interface are entailed by the body of a module, assuming the properties described in the requires/uses-interfaces), (2) a way of formalising the matching of provides/requires-interfaces, and (3) a means for validation of activity and service design. Together with the usage of the c-semiring approach to constraint optimisation [4], SRML offers the means through which one can improve levels of user satisfaction, safety and reliability to automotive system while enhancing adaptability.

8. ACKNOWLEDGMENTS

We would like to thank our colleagues in the SENSORIA project (IST-2005-16004) for many useful discussions on the topics covered in this paper.

9. REFERENCES

- [1] Abreu, J. and Fiadeiro, J. *A Coordination Model for Service-Oriented Interactions*. In *Coordination Languages and Models*, LNCS, Springer, to appear, 2008.
- [2] Alonso, G., Casati, F., Kuno, H. and Machiraju, V. *Web Services*. Springer, 2004.
- [3] Baina, K., Benatallah, B., Casati, F. and Toumani, F. Model-Driven Web Service Development. In *Proceedings of CAiSE 2004 (Riga, Latvia, June 2004)*.
- [4] Bistarelli, S., Montanari, U. and Rossi, F. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2): 201-236, 1997.
- [5] Bocchi, L., Hong, Y., Lopes, A. and Fiadeiro, J. From BPEL to SRML: a formal transformational approach. In *Web Services and Formal Methods*. LNCS 4937, pp 92-107, 2007
- [6] Broy, M., Krüger, I. and Meisinger M. A formal model of services. *ACM TOSEM* 16(1): 1-40, 2007.
- [7] Elfatraty, A. Dealing with change: components versus services. *Communications of the ACM* 50(8): 35-39, 2007.
- [8] Fiadeiro, J.L., Lopes, A. and Bocchi, L. A formal approach to service-oriented architecture. In *Web Services and Formal Methods*, LNCS 4184, pp 193-213, 2006.
- [9] Fiadeiro, J. L., Lopes, A. and Bocchi, L. Algebraic semantics of service component modules. In: J. L. Fiadeiro and P. Y. Schobbens (eds) *Algebraic Development Techniques*. LNCS 4409, Springer, pp 37-55, 2006.
- [10] Fiadeiro, J. L., Lopes, A. and Bocchi, L. Semantics of Service-Oriented System Configuration. University of Leicester, submitted, 2008.
- [11] Gnesi, S. and Mazzanti, F. A model checking verification environment for UML Statecharts. In: *Proceedings of XLIII Congresso Annuale AICA "Comunita' Virtuale dalla Ricerca all'Impresa dalla Formazione al Cittadino"*, University of Udine – AICA, 2005. (paper available from fint.isti.cnr.it).
- [12] OASIS. Web Services Business Process Execution Language Version 2.0. OASIS standard (2007) <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf>
- [13] Reisig, W. Modeling- and analysis techniques for web services and business processes. In *Proceedings of FMOODS 2005*, LNCS 3535, Springer, pp 243-258, 2005.
- [14] SCA Consortium. *Building Systems using a Service Oriented Architecture*. Whitepaper. [www-128.ibm.com/ developer-works/library/specification/ws-sca](http://www-128.ibm.com/developer-works/library/specification/ws-sca) (version 0.9), 2005.
- [15] SENSORIA consortium (2007) White paper. <http://www.sensoria-ist.eu/files/whitePaper.pdf>.
- [16] Web Service Description Language (WSDL). <http://www.w3.org/TR/wsdL>.