7 Sensoria: Engineering for Service-Oriented Overlay Computers

Martin Wirsing, Laura Bocchi, Allan Clark, José Luiz Fiadeiro, Stephen Gilmore, Matthias Hölzl, Nora Koch, Philip Mayer, Rosario Pugliese, and Andreas Schroeder

7.1 Introduction

The last decades have shown tremendous advances in the field of information technology (IT). In fact, with online access to vast amounts of information, states offering their services for citizens on the World Wide Web, and software in control of critical areas such as flight control systems, information systems now lie at the very foundations of our society. But it is the business sector where IT has had the most impact. Driven by the need to stay competitive in an environment of rapid change in global markets and business models, as well as local regulations and requirements, organizations now strongly depend on a functional and efficient IT infrastructure which is flexible enough to deal with unexpected changes while still offering stability for business processes and connections to customers, partners, and suppliers.

With the emerging trend of automating complex and distributed business processes as a whole, many organizations face the problem of integrating their existing, already vastly complicated, systems to reach yet another layer of sophistication in the interconnection of business value, business processes, and information technology. Many IT systems are difficult to adapt and work with; progress in the world of businesses is often impeded by the difficulty of changing the existing IT infrastructure. Service-oriented computing (SOC) is a new approach to software development and integration that addresses these challenges. SOC provides the opportunity for organizations to manage their heterogeneous infrastructures in a coherent way, thus gaining new levels of interoperability and collaboration within and across the boundaries of an organization. In addition, SOC promises new flexibility in linking people, processes, information, and computing platforms.

The IST-FET Integrated Project Sensoria[1] is a European Community-funded project that develops methodologies and tools for dealing with service-oriented computing. It addresses major problems found in current approaches to SOC and provides mathematically founded and sound methodologies and tools for dealing with the amount of flexibility and interoperability needed in these next-generation infrastructures. Sensoria aims to support a more systematic and scientifically well-founded approach to engineering of software systems for service-oriented overlay computers. At the core of our research is a concept of service that generalizes current approaches such as Web Services and Grid Computing. Sensoria provides the scientific foundations for a development process where developers can concentrate on modeling the high-level behavior of the system and use transformations for deployment and analysis. To this end we investigate programming primitives supported by a mathematical semantics; analysis and verification techniques for qualitative and quantitative properties such as security, performance, quality of service, and behavioral correctness; and model-based transformation and development techniques.

In the NESSI road map most of the research of Sensoria is located in the service integration layer, although the research touches both the semantic layer and the infrastructure layer, as well as the crosscutting areas of management services, interoperability, and security. Sensoria addresses grand challenges from the areas of service foundations (dynamic reconfigurable architectures, end-to-end security), service composition (composability analysis operators for replaceability, compatibility, and conformance; QoS-aware service composition; business-driven automated composition); service management (self-configuring services); and service engineering (design principles for engineering service applications, associating a service design methodology with standard software development, and business process modeling techniques). In the addressed grand challenges, the Sensoria research consortium focuses on solid mathematical foundations—the project as a whole therefore is located in the "foundational plane" of the SOC Road Map (see chapter 1).

In the remaining sections, we present the Sensoria development approach and illustrate its building blocks. Section 7.2 contains an overview of the Sensoria project and introduces the case study that will be used throughout the chapter.

Today, both UML and SCA are commonly used to specify service-oriented systems, either individually or in combination. Therefore we show two alternative approaches to modeling: the Sensoria UML extension for services and the SCA-inspired SRML, in sections 7.3 and 7.4, respectively.

When orchestrating service-oriented systems it is often necessary to find trade-offs between conflicting system properties. In section 7. 5 we show how soft constraints can be used to model context-dependent restrictions and preferences in order to perform dynamic service selection.

In section 7.6, to illustrate our approach to qualitative and quantitative analysis, we show two of the calculi which are used in the Sensoria development environment, but normally hidden from the developers. The COWS calculus can be used for performing security modeling and enforcement on service orchestrations; in particular, we add annotations for ensuring confidentiality properties. The PEPA process algebra is used to validate the performance of orchestrations regarding previously defined service-level agreements in section 7.7.

An expanded version of this paper is available as LMU-IFI-PST technical report 0702 (Wirsing et al. 2007a).

7.2 Sensoria

Sensoria is one of the three integrated projects of the Global Computing Initiative of FET-IST, the Future and Emerging Technologies section of the European Commission. The Sensoria Consortium consists of 13 universities, two research institutes, and four companies (two SMEs) from seven countries.[2]

7.2.1 Sensoria Approach

The core aim of the Sensoria project is the production of new knowledge for systematic and scientifically well-founded methods of service-oriented software development. Sensoria provides a comprehensive approach to the visual design, formal analysis, and automated deployment of service-oriented applications. The Sensoria techniques enable service engineers to model their applications on a very high level of abstraction, using service-oriented extensions of the standard UML or standard business process models, and to transform those models to be able to use formal analysis techniques as well as generate executable code.

The Sensoria techniques and tools are built on mathematical theories and methods. These formal results are complemented by realistic case studies for important application areas including telecommunications, automotive, e-learning, and e-business. The case studies are defined by the industrial partners to provide continuous practical challenges for the new techniques of services engineering and for demonstrating the research results. This approach is shown in figure 7.1.

An important consideration of the Sensoria project is "scalable analysis for scalable systems." Our vision is that developers of service-oriented systems can develop at a high level of abstraction with support from tools that analyze the system models, provide feedback about the reliability of the system and possible problems, establish the validity of functional and nonfunctional requirements, and manage the deployment to different platforms. We base our tools on formal languages with well-defined properties which make it possible to establish their correctness. Two aspects are particularly important to make the process practical for commercial software development: (1) analysis results are translated back into the familiar modeling notations so that developers do not need to understand the calculi used to perform the analysis, and (2) the analyses not only are usable for "toy examples," but also scale to realistic problems. The process may be customized to various application domains and different iterative and agile process models, such as the Rational Unified Process (RUP), Model-Driven Development (MDD), or Extreme Programming (XP). Figure 7.2 displays one step in this development process.

The developer usually works with higher-level input models, but also with program code, and uses the Sensoria development environment (downloadable from Sensoria project 2007) to perform qualitative or quantitative analysis and to generate output (e.g., new models or code that can be deployed on various platforms). Our tools within the

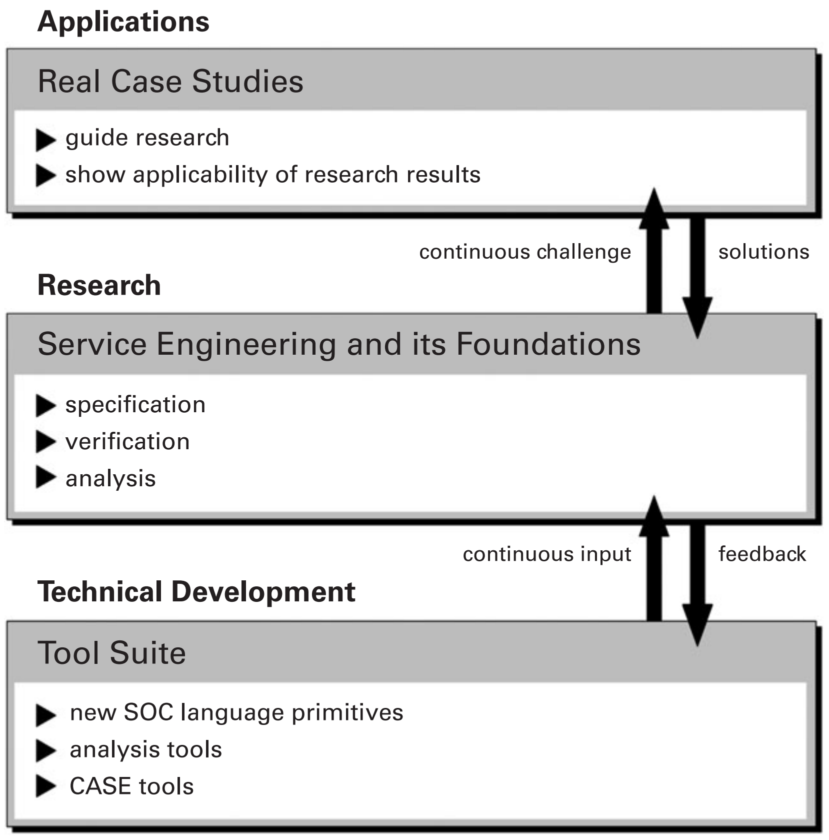


Figure 7.1
Sensoria innovation cycle

qualitative and quantitative analyses can, for example, perform checks of functional correctness of services, early performance analysis, prediction of quantitative bottlenecks in collaborating services, and verification of service-level agreements.

7.2.2 Research Themes

The research themes of Sensoria range across the whole life cycle of software development, from requirements to deployment, including reengineering of legacy systems.

Modeling Service-Oriented Software The definition of adequate linguistic primitives for modeling and programming service-oriented systems enables model-driven development for implementing services on different global computers, and for transforming legacy systems into services using systematic reengineering techniques. Modeling front ends allows

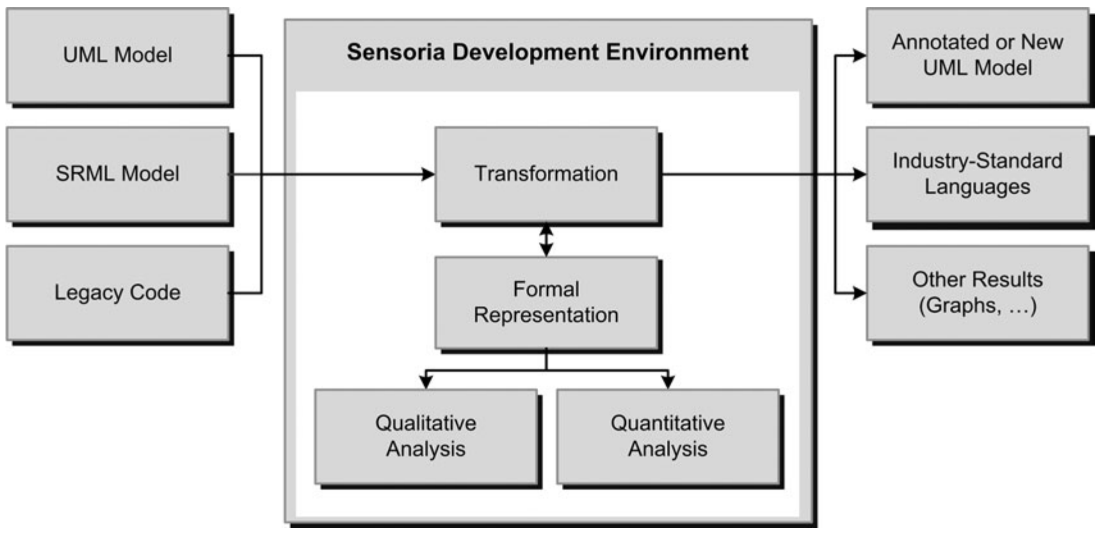

UML Model
SRML Model
Legacy Code
Sensoria Development Environment
Transformation
Formal Representation
Qualitative Analysis
Quantitative Analysis
Annotated or New UML Model
Industry-Standard Languages
Other Results (Graphs, ...)

Figure 7.2
Sensoria development process

designers to use high-level visual formalisms such as the industry standard UML. Automated model transformations allow generation of formal representations from engineering models for further development steps.

Formal Analysis of Service-Oriented Software Based on Mathematically Founded Techniques Mathematical models, hidden from the developer, enable qualitative and quantitative analysis supporting the service development process and providing the means for reasoning about functional and nonfunctional properties of services and service aggregates. Sensoria results include powerful mathematical analysis techniques, particularly in program analysis techniques, type systems, logics, and process calculi for investigating the behavior and the quality of service of properties of global services.

Deploying and Runtime Issues of Service-Oriented Software The development of sound engineering techniques for global services includes deployment mechanisms with aspects such as runtime self-management, service-oriented middlewares, and model-driven deployment, as well as reengineering legacy systems into services, thus enabling developers to travel the last mile to the implementation of service-oriented architectures.

7.2.3 Automotive Case Study

Today, computers embedded in cars can access communication networks such as the Internet, and thereby provide a variety of new functionalities for cars and drivers. In the future, instead of having to code these functionalities, services will be able to be discovered at run-time and orchestrated so as to deliver the best available functionality at agreed levels of quality. A set of possible scenarios of the automotive domain are examined to illustrate the scope of the Sensoria project. In the following we focus on one of them, the car repair scenario.

In this scenario, the diagnostic system reports a severe failure in the car engine that implies the car is no longer drivable. The in-vehicle repair support system is able to orchestrate a number of services (garage, backup car rental, towing truck) that are discovered and bound at that time according to levels of service specified at design time (e.g., balancing cost and delay). The owner of the car deposits a security payment before the discovery of the services is triggered.

7.3 UML Extension for Service-Oriented Architectures

Within the Sensoria approach, a service engineer can model services using a specialization of the UML covering static, dynamic, and quality-of-service aspects of services.

For the static aspects of service-oriented software systems, the UML specialization ranges from rather simple, stereotyped language extensions for introducing services to more complicated structures such as dependency relations between services and their contextual relationships to resources and legacy systems. Modeling dynamics of service-oriented software, in particular orchestration and choreography of services, is supported by primitives for interaction and activity modeling that take into account possible failures and quality-of-service aspects. (These nonfunctional extensions are not covered in this chapter. Interested readers are referred to Wirsing et al. 2006.)

7.3.1 Modeling Structures of SOA

The structure of a service-oriented architecture can be visualized by UML deployment and composite structure diagrams. A deployment diagram is used to represent the—usually nested—nodes of the architecture (i.e., hardware devices and software execution environments). Figure 7.3 shows a UML deployment diagram of the car and its environment as first approximation to an architecture model. The nodes are connected through communication paths that show the three types of communication that characterize the automotive domain: intravehicle communication (nonnamed connections), intervehicle communication, and communication among vehicle and environment, such as the communication with the car manufacturer or a remote discovery server. The components that are involved in the execution of service orderings are a service discovery which may be local or external to the car, a reasoner for service selection, and a service orchestrator.

In addition to UML deployment diagrams, which give a static view of the architecture, we use UML structure diagrams to represent the evolving connections within the service-oriented architecture of the vehicle and its environment. Three types of connections are

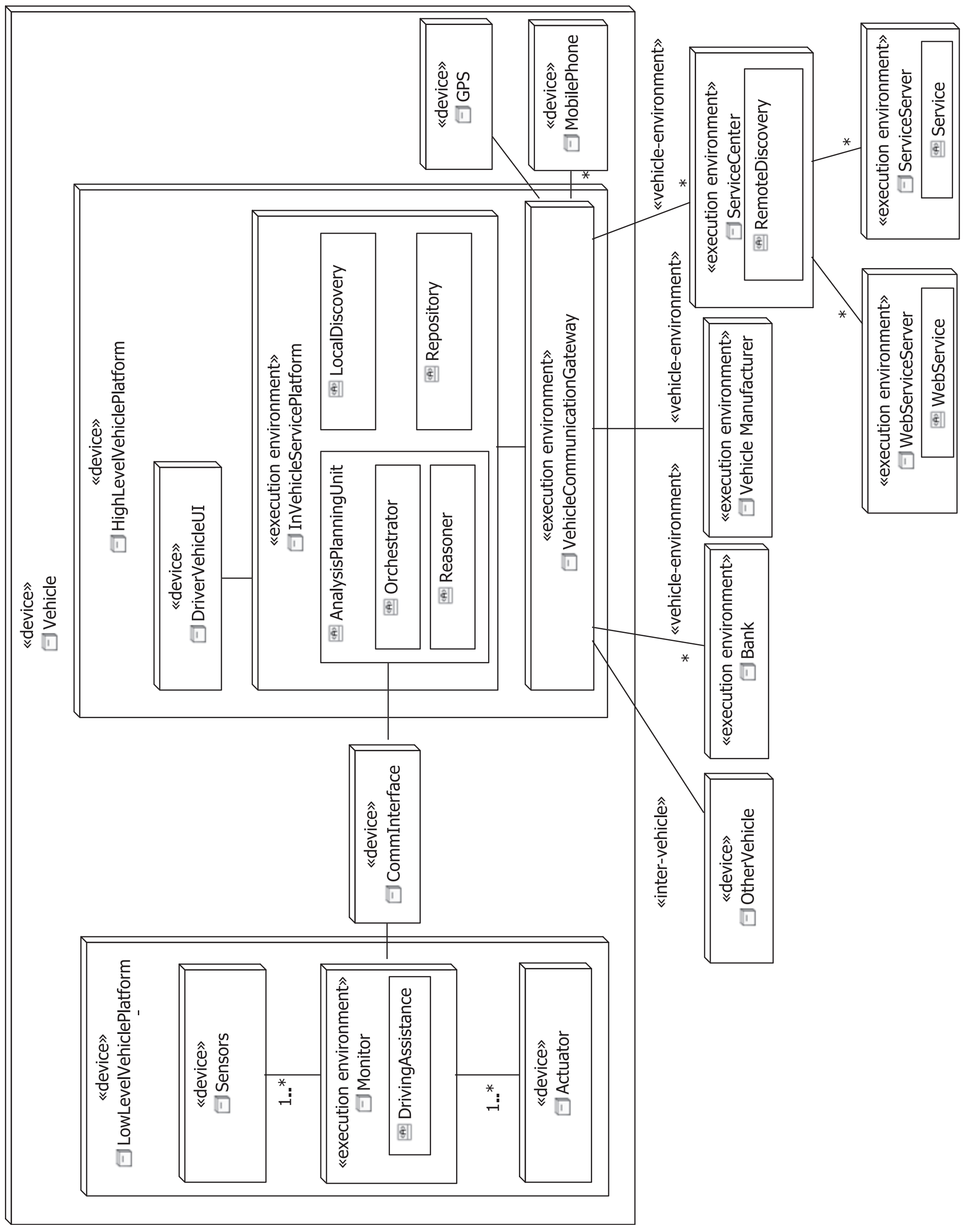

«device» Vehicle
«device» HighLevelVehiclePlatform
«device» DriverVehicleUI
«execution environment» InVehicleServicePlatform
AnalysisPlanningUnit
Orchestrator
Reasoner
LocalDiscovery
Repository
«execution environment» VehicleCommunicationGateway
«device» GPS
«device» MobilePhone
«device» CommInterface
«device» LowLevelVehiclePlatform
«device» Sensors
«execution environment» Monitor
DrivingAssistance
«device» Actuator
1..*
1..*
*
«vehicle-environment»
«execution environment» ServiceCenter
RemoteDiscovery
«execution environment» ServiceServer
Service
«execution environment» WebServiceServer
WebService
«execution environment» Vehicle Manufacturer
«execution environment» Bank
«inter-vehicle»
«device» OtherVehicle

Figure 7.3
Simplified architecture of car and car environment

identified: discovery connection, permanent connection (as in modeling of non-service-oriented architectures), and temporary connections. In order to be invoked, services need to be discovered before the binding to the car's onboard system takes place. Thus the discovery connection is based on the information provided by a discovery service. We distinguish a temporary connection which is, for example, one from the car's onboard system to a known service, such as the car manufacturer's discovery service, from a permanent connection. Permanent connections wire components as in traditional software. For a graphical representation and more details about these connections, the reader is referred to Wirsing et al. (2006).

7.3.2 Modeling Behavior of SOA

The behavior of a service-oriented system is mainly described by the service orchestration which defines the system's workflow. Modeling orchestration of services includes specifying interactions among services, modeling transactional business processes using concepts developed for long-running transactions, and specifying nonfunctional properties of services.

In the modeled business process of the on-road car repair scenario, the orchestration is triggered by an engine failure or a sensor signal such as low oil level. The process starts with a request from the orchestrator to the bank to charge the driver's credit card with the security deposit payment, which is modeled by an asynchronous UML action `requestCardCharge` for charging the credit card. The number of the card is provided as output parameter of the UML stereotyped call action. In parallel to the interaction with the bank, the orchestrator requests the current position of the car from the car's internal GPS service. The current location is modeled as input to the `requestLocation` action and subsequently used by the `findLocalServices` and `findRemoteServices` interactions, which retrieve a list of services. For the selection of services the orchestrator synchronizes with the reasoner to obtain the most appropriate services. Service ordering is modeled by the UML actions `orderGarage`, `orderTowTruck`, and `orderRentalCar`, following a sequential and parallel process, respectively.

Figure 7.4 shows a UML activity diagram of the orchestration of services in the on-road car repair scenario. We use stereotyped UML actions indicating the type of interactions (`send, receive, sendAndReceive`), and model input and output parameters of the interactions with UML pins stereotyped with outgoing and incoming arrows (abbreviations for send and receive, respectively). These pins are not linked to any edge, but specify variables containing data to be sent or target variables to accept the data received; constraints prohibit illegal combinations such as send actions with input pins. Interactions match operations of required and provided interfaces of the services. Services are defined as ports of UML components.

The key technique to handle long-running transactions is to install compensations (e.g., based on the Saga concepts; Bruni et al. 2005). Modeling of compensations is not directly supported in UML. We provide a UML extension within Sensoria for the modeling of these compensations. The extension consists of two modeling primitives—`Scope` and `CompensationEdge`—and corresponding stereotypes for UML activity diagrams. A `Scope` is a structured activity node that groups activity nodes and activity edges. The grouping mechanism provided is convenient for the definition of a compensation or fault handler for a set of activities; at the same time, fault and compensation handlers can also be attached directly to actions. Compensation handlers are defined by linking an activity or scope with a `compensationEdge` stereotyped activity edge to the compensation handler. The handler in turn may again consist of a single action (e.g., see `cancelGarage` in figure 7.4). On completion of an activity node with an associated compensation handler, the handler is installed and is then executed upon the occurrence of an unhandled exception in the continued execution of the orchestration.

The UML stereotypes defined for orchestration are part of the Sensoria UML Profile. The use of such a UML extension has direct advantages for the design of model transformations, especially for deployment transformations.

7.4 The Sensoria Reference Modeling Language

SRML is a language for modeling composite services understood as services whose business logic involves a number of interactions among more elementary service components as well the invocation of services provided by external parties. SRML offers modeling primitives inspired by the Service Component Architecture (SCA) (Beisiegel et al. 2005) but addressing a higher level of abstraction in which one models the business logic of the domain. SRML models both the structure of composite services and their behavioral aspects. As in SCA, interactions are supported on the basis of service interfaces defined in a way that is "independent of the hardware platform, the operating system, the hosting middleware and the programming language used to implement the service" (Fiadero et al. 2006).

An SRML module declares one or more components, a number of "requires-interfaces" which specify services that need to be provided by external parties, and (at most) one "provides-interface" that describes the properties of the service offered by the module. Components have a tight coupling (performed at design time) and offer a distributed orchestration of the parties involved in the service. The coupling between components and external parties is looser and is established at runtime through the discovery, selection, and binding mechanisms that are supported by the underlying service middleware.

A number of wires establish interaction protocols among the components and between the components and the external parties that instantiate the interfaces. A wire establishes a binding between the interactions that both parties declare to support and defines the interaction protocol that coordinates those interactions. Figure 7.5 illustrates the SRML module for the on-road car repair scenario.

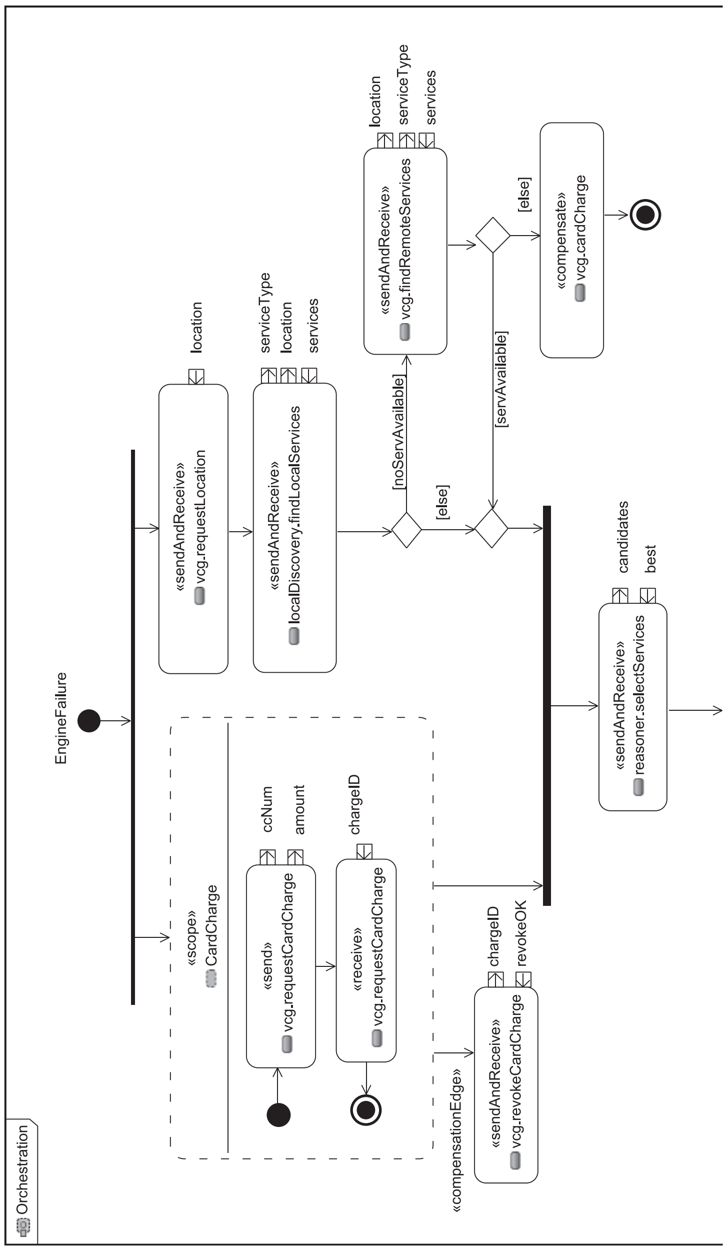


Figure 7.4
Orchestration in the on road car repair scenario

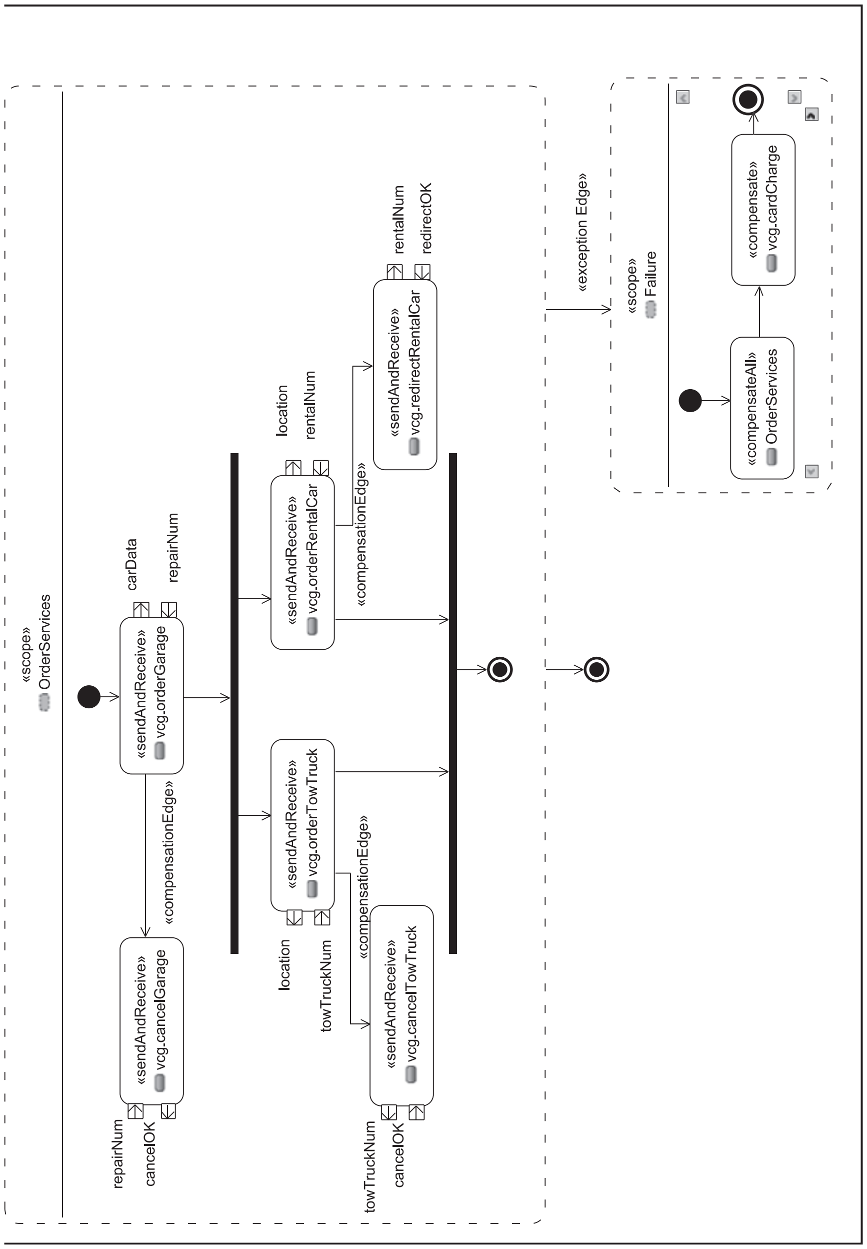
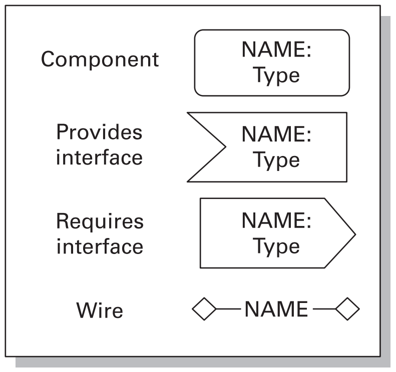
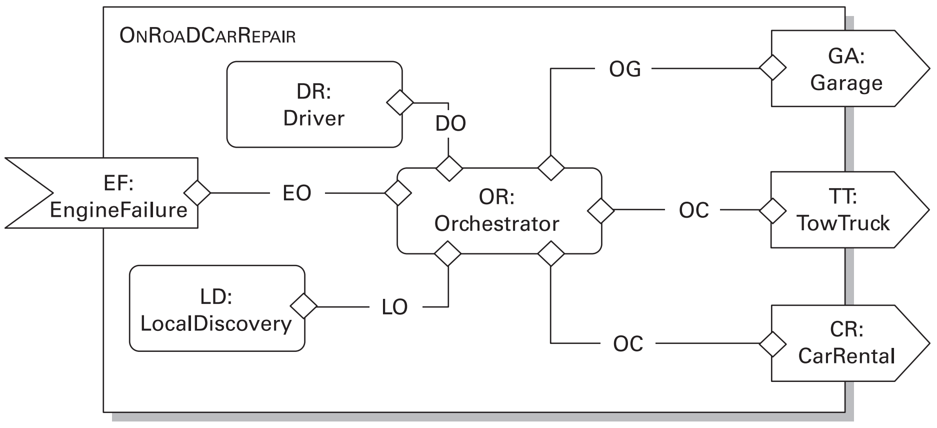


Figure 7.4
(continued)



(a)



(b)

Figure 7.5
(a) Entities in a SRML module, (b) SRML module for the on road car repair scenario

We are developing an editor for SRML based on Eclipse Modeling Framework (EMF). The editor relies on an SRML metamodel consisting of an EMF tree; graph transformation techniques can be used to automate the encoding between (parts of) SRML and other languages for which an EMF metamodel exists. Thus, for example, we are providing an encoding from BPEL processes into SRML modules (Bocci et al. 2007). More specifically, we extract high-level declarative descriptions of BPEL processes that can be used for building more complex modules, possibly including components defined in other (implementation or modeling) languages for which an encoding into SRML exists. We are using the same approach to develop mappings between SRML and other modeling

```
INTERACTIONS

      rcv engineFailure

      s&r askUsrDetails

         ⊠ cust:customerID

            ccID:bankAccountID

            dest:location

            workRelated:boolean

           ⊿t:time

      s&r currentLocation

         ⊠ l:location

      snd confirmation

      s&r requestCharge

         🔔 cust:customerID

            ccID:bankAccountID

            amount:moneyVal

         ⊠ certif:paymentCertificate

      s&r orderGarage

         🔔 …

      s&r rentACar

         🔔 …

      s&r orderTowTruck

         🔔 …
```

Figure 7.6
Specification of the interactions for the business role *Orchestrator*

languages/calculi developed in Sensoria. For instance, we are encoding SRML into COWS in order to provide SRML with an operational semantics for the process of service discovery.

7.4.1 Specifying SRML Entities

The specification of each SRML entity (components, external interfaces, and wires) addresses properties of the interactions supported by that entity.

Figure 7.6 presents the interactions specified in the business role fulfilled by the orchestrator. SRML supports conversational interaction types—send-and-receive (s&r) and receive-and-send (r&s)—involving initiation (`interaction`🔔 denotes the event of initiating interaction), reply (`interaction`⊠ denotes the reply event of `interaction`), and other events for handling specific aspects of conversations, such as committing, canceling, and revoking a deal (see Fiadero et al. 2006 for an exhaustive list of types of interactions and interaction events). Interactions can have (🔔-parameters for transmitting data when the interaction is initiated and (⊠-parameters for carrying a reply).

The different types of entities involved in a module are specified in SRML using three different but related languages. All the languages provide a specification of the supported interactions. The languages for defining business roles, business protocols, and interaction protocols differ in the way they define the behavioral interface.

A component is a computational unit represented in a module by what we call a business role. A business role declares the interactions in which the component can become involved, as well as the execution pattern that orchestrates those interactions (i.e., the orchestration). The orchestration is specified through a set of variables that provide an abstract view of the state of the component and by a set of transition rules.

The transition rules can be derived from UML sequence or activity diagrams and refined with additional information. Each transition has a *trigger*, typically the occurrence of an interaction event, a *guard* identifying the states in which it can take place, the *effects* that change the local state (s′ denotes the value of the local variable s after the transition), and the *sends* which specify the interaction events that are sent. The following is a transition of the business role *Orchestrator*.

```
transition startProcess
  triggeredBy engine Failure🔔?
  guardedBy s=Begin
  effects s'=FailureDetected
  sends currentLocation🔔!
    ∧ askUsrDetails🔔!
```

The external interfaces are defined in a module through what we call business protocols. Business protocols specify the interactions similarly to business roles. Their behavior abstracts from details of the local state of the co-party and specifies the protocol that the co-party adheres to as a set of properties concerning the causality of the interactions. The following is the behavior for the business protocol Engine Failure.

```
initiallyEnabled engineFailure🔔?
engineFailure🔔? enables confirmation🔔!
```

The behavior is described by two statements. The first one ensures that the interaction *engineFailure*🔔? is enabled from the beginning of the session and that it remains enabled until it occurs. In the second one, confirmation🔔! will be accepted only after the occurrence of *engineFailure*🔔?.

7.4.2 Service-Level Agreement

SRML offers primitives for modeling "dynamic" aspects concerned with configuration, session management, and service-level agreement (SLA). In particular, SRML supports the definition of attributes and constraints for SLA, using the algebraic techniques developed in Bistarelli et al. (1997) and Bistarelli (2004) for constraint satisfaction and optimization. Section 7.5 presents this calculus in more detail, together with an example, and shows how the constraints can be modeled in SRML.

7.5 Soft Constraints for Selecting the Best Service

The reasoning component of a service-oriented system (see figure 7.3) has to achieve a compromise between different goals of the system. Soft constraints are a promising way to specify and implement such reasoning mechanisms.

Soft constraints are an extension of classical constraints to deal with nonfunctional requirements, overconstrained problems, and preferences. Instead of determining just a subset of admissible domain elements, a soft constraint assigns a grade to each element of the application domain. Bistarelli, Montanari, and Rossi (Bistarelli et al. 1997) and Bistarelli (2004) have developed a very elegant theory of soft constraints where many different kinds of soft constraints can be represented and combined in a uniform way over so-called constraint semirings (*c-semirings*).

In Sensoria we are developing a language which extends the c-semiring approach with possibilities to specify preferences between constraints and to vary the constraints according to a context. This simplifies the description of behaviors of systems in a dynamically changing environment and with complex trade-offs between conflicting properties.

A context is a boolean expression which can guard a constraint. For example, the distance to the destination might determine whether the quick availability of a rental car is important or not. In this case, "distance < 20km" is a context that can restrict the applicability of a constraint. Variables appearing in contexts are called *context variables*; variables appearing only in constraints but not in contexts are called *controlled variables*. In the car repair scenario, the context variables will specify, among other things, the *distance* to the destination or whether the journey is *work-related*. The controlled variables represent properties of offers (e.g., the cost or quality of an offer). A soft constraint is a mapping from (domains of) controlled variables into a c-semiring.

In the car repair scenario we use soft constraints to specify the preferences of the users. The constraints map the controlled variables into a c-semiring where values are natural numbers; 0 means not acceptable, 1 means barely acceptable, and higher numbers represent higher values of acceptance. For example, the user may prefer a garage that can repair the car as quickly as possible, with a duration of more than two days not being acceptable:

fastRepair: $[\textit{garage-duration} \,|\, n \to 148 \lfloor 48 \to n \rfloor]$

We also may want the repair to be done cheaply, but only if the trip is not work-related. Repairs costing more than 1000 euros are still acceptable, but only barely:

cheapRepair: **in context** $\neg$*work-related*?
assert $[\textit{garage-cost} \,|\, \mathrm{n} \to \lfloor 1000 \to \mathrm{ni}\ 1]$ **end**

Users might not consider all constraints to be equally important. For example, a user who prefers fast but expensive repairs can specify this as a preference between constraints: fastRepair $>$ cheapRepair. In this case an offer that results in a faster repair will always be preferred; the price will be used only to break ties between equally fast repairs.

From a set of constraints and preferences the reasoner can compute either the best solutions or a set of all solutions that are better than a certain threshold. Two techniques that are used for solving soft constraint systems are branch search and bound search (Wirsing et al. 2007c) and dynamic programming (Bistarelli 2004).

The constraints presented in this section can be modeled in SRML using the following syntax, where S is a c-semiring, D is a finite set (domain of possible elements taken by the variables), and V is a totally ordered set (of variables).

7.5.1 Constraint System

S is `<[0..1000],max,min,0,1000>`
D is {nn∈N:1 ≤ n ≤ 1000}
V is `{DR.askUsrDetails⊠.workRelated, GA.cost, GR.duration,...}`

The variable *DR.askUsrDetails⊠.workRelated* is a Boolean context variable that is true if the trip is work-related. The variables *GA.cost* and *GR.duration* represent the cost and availability of the garage service, respectively. We show the SRML specification of the *fastRepair* and *cheapRepair* constraints. Notice that we represent the context as a variable of the constraint.

7.5.2 Constraints

```
fastRepair is  <{GA.duration},def1>
          s.t. def1(n)=48/n;
cheapRepair is <{GA.cost},def2> s.t. def2(n)=
          if DR.askUsrDetails⊠.workRelated
               then 1 else 1000/n;
```

7.6 A Process Calculus for Service-Oriented Systems

COWS (Calculus for Orchestration of Web Services [Lapadula et al. 2007b]) is a recently designed process calculus for specifying, combining, and analyzing service-oriented applications while modeling their dynamic behavior. We present (an excerpt of) COWS's main features and syntax while modeling some simplified components of the on-road car repair scenario. The type system of COWS (Lapadula et al. 2007a) enables us to verify confidentiality properties (e.g., that critical data such as credit card information is shared only with authorized partners). The complete specification, including compensation activities, can be found in Wirsing et al. (2007b).

7.6.1 Service Orchestration with COWS

To start with, we present the COWS term representing the "orchestration" of all services within the scenario of section 7.3:

$[p_{car}]$ *(Orchestrator | LocalDiscovery | Reasoner | SensorsMonitor) | Bank | OnRoadRepairServices*

The services above are composed by using the *parallel composition* operator _ | _ 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*, *LocalDiscovery*, *Reasoner*, and *SensorsMonitor*), and only to them.

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

$$[\mathrm{x}_1, \ldots, \mathrm{x}_n]\ (p_{car} \cdot o_{engfail}?\langle x_1, \ldots, x_n\rangle.s_{engfail} + p_{car} \cdot o_{lowoil}?\langle x_1, \ldots, x_n\rangle.s_{lowoil})$$

This term uses the *choice* operator _+_ to pick one of those alternative "recovery" behaviors whose execution can start immediately. For simplicity, only "engine failure" and "low oil" situations are taken into account.

The *receive-guarded prefix* operator $p_{car} \cdot o_i?\langle x_1, \ldots, x_n\rangle.\_$ expresses that each recovery behavior starts with a *receive* activity of the form $p_{car} \cdot o_i?\langle x_1, \ldots, x_n\rangle$ corresponding to reception of a request emitted, when a failure arises, by *SensorsMonitor* (a term representing the behavior of the "low level vehicle platform" of figure 7.3). *Receives*, together with *invokes*, written as $p \cdot o!\langle e_1, \ldots, e_n\rangle$, are the basic communication activities provided by COWS.

Besides input parameters and sent values, they indicate an *end point* (i.e., a pair composed of a partner name p and an operation name o, through which 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. An interservice communication takes place when the arguments of a receive and of a concurrent invoke along the same end point do match, 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, variables $x_1, \ldots, x_n$, declared local to *Orchestrator* by means of the delimitation operator, are initialized by the receive leading the recovery activity with data provided by *SensorsMonitor*.

The recovery behavior $\mathrm{s}_{\mathrm{engfail}}$, executed when an engine failure occurs, is

[pe,oe,xloc,xlist] ((requestCardCharge | requestLocation.findServices)
| $p_e \cdot o_e?\langle\ \rangle.p_e \cdot o_e?\langle\ \rangle$selectServices.orderGarage. (orderTowTruck | orderRentalCar))

$p_e \cdot o_e$ is a scoped end point along which successful termination signals (i.e., communications that carry no data) are exchanged to orchestrate execution of the different components. Variables x_{loc} and x_{list} are used to store the value of the current car's GSP position and the list of closer on-road services discovered. To present the specification of $s_{engfail}$ in terms of the UML actions of figure 7.4, we have used an auxiliary "sequence" notation (e.g., in *requestLocation.findServices*). This notation indicates that execution of *requestLocation* terminates before execution of findServices starts. Indeed, *requestLocation .findServices* actually stands for the COWS term

$p_{car} \cdot o_{reqLoc}!\langle\ \rangle \mid p_{car} \cdot o_{respLoc}?\langle x_{loc}\rangle.$
$(p_{car} \cdot o_{findServ}!\langle x_{loc}\rangle$
$\mid p_{car} \cdot o_{found}?\langle x_{list}\rangle.p_e \cdot o_e!\langle\ \rangle + p_{car} \cdot o_{notFound}?\langle\ \rangle)$

where *requestLocation* and *findServices* are

$requestLocation \triangleq p_{car} \cdot o_{reqLoc}!\langle\ \rangle \mid p_{car} \cdot o_{respLoc}?\langle x_{loc}\rangle$
$findServices \triangleq p_{car} \cdot o_{findServ}!\langle x_{loc}\rangle$
$\quad \mid p_{car} \cdot o_{found}?\langle x_{list}\rangle.p_e \cdot o_e!\langle\ \rangle + p_{car}o_{notFound}?\langle\ \rangle$

Bank, the last service we show, can serve multiple requests simultaneously. This behavior is modeled by exploiting the *replication* operator * _ to spawn in parallel as many copies of its argument term as necessary. The definition of *Bank* is

$* \; [x_{cust}, x_{cc}, x_{amount}, O_{checkOK}, O_{checkFail}]$
$p_{bank} \cdot O_{charge}? \; \langle x_{cust},\ x_{cc},\ x_{amount}\rangle.$
$(\langle$perform some checks and reply $O_{nocheck}OK$ or $O_{checkFail}\rangle$
$\mid p_{bank} \cdot O_{checkOK}?\langle\ \rangle \cdot x_{cust} \cdot O_{chargeOK}!\langle\ \rangle$
$+ p_{bank} \cdot O_{checkFail}? \; \langle\ \rangle \cdot x_{cust} \cdot O_{chargeFail}!\langle\ \rangle)$

Once prompted by a request, in contrast to *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.

7.6.2 Using Types for Verifying Service Properties

One advantage of using COWS as modeling language is that it already provides some tools for analyzing the models and verifying the properties they enjoy. For example, the type system introduced in Lapadula et al. (2007a) for checking data security properties on COWS terms is a practical and scalable way to provide evidence that a large number of applications enjoy some given properties: from the *type soundness* of the language as a whole, it follows that all well-typed applications do comply with the properties stated by their types. The types permit expression of policies constraining data exchanges in terms of *regions* (i.e., sets of 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's being accessed (by unwanted services) through unauthorized partners. The language operational semantics uses these annotations to guarantee that computations proceeds according to them.

To provide a flavor of the properties that can be expressed and enforced by using the type system, we illustrate some properties relevant for the scenario modeled above. First, a driver in trouble must be assured that information about his credit card and GSP position cannot become available to unauthorized users. To this aim, the credit card identifier *ccId*, communicated by activity *requestCardCharge* to the service *Bank*, can be annotated with the policy $\{p_{bank}\}$, which allows *Bank* to receive the datum but prevents it from transmitting the datum to other services. Similarly, the car's GSP position stored in x_{loc}, used by services *orderGarage*, *orderTowTruck*, and *orderRentalCar*, can be annotated with the regions $\{x_{garage}\}$, $\{xt_{owTruck}\}$, and $\{x_{rentalCar}\}$, respectively, to specify different policies for the same datum, according to the invoked services. Notably, these policies are not statically fixed, but depend on the partner variables x_{Garage}, $x_{towTruck}$, and $x_{rentalCar}$, and thus will be determined by the values that these variables assume as computation proceeds. As a final example property, we mention that by using only appropriate regions, including the customer partner name, one can guarantee that critical data of on-road services, such as cost and quality of the service supplied, are not disclosed to competitor services.

7.7 Quantitative Analysis of Service-Level Agreements

In the car repair scenario the quantitative issue of concern relates to how long it will take from the point of engine failure until both a tow truck and a garage have been ordered, and the tow truck is on its way to help the stranded driver. If the duration of each of the service activities which need to take place along the way (*requestLocation*, *findServices*, *orderGarage*, ...) was known exactly, then this calculation would simply involve adding up these times to give the total duration. However, as is usual in service-oriented systems, none of these durations will be known exactly in this case, and the calculation needs to be based on the expected average duration of each of the atomic service events. In this setting, where only the average duration is known, and not the variance or higher moments, the correct distribution to model with is the exponential distribution. Thus, this aspect of the problem naturally lends itself to Markovian representation and analysis, and that is the approach we will take here.

For the quantitative analysis of such systems we use Performance Evaluation Process Algebra (PEPA) (Hillston 1996), which extends classical process algebras by associating a duration with each activity. Thus, where classical process algebras such as CCS and CSP deal with instantaneous actions which abstract away from time, PEPA has continuous-time activities whose durations are quantified by exponentially distributed random variables. Thus PEPA is a *stochastic* process algebra which describes the evolution of a process in continuous time. The operational semantics of the language gives rise to a continuous-time, finite-state stochastic process called a continuous-time Markov chain (CTMC), which can be used to find the steady-state probability distribution over a model. From this it is straightforward to compute conventional performance measures such as utilization or throughput. Here we are instead performing transient analysis of the CTMC, where one considers the probability distribution at a chosen instant of time. It is possible to use these results to perform more complex quantitative analysis, such as computing response-time measures and first passage-time quantiles, as used in service-level agreements. It is also possible to perform scalability analysis by using an altogether different

Table 7.1
Activities and minimum and maximum values of the rates from the model

		Value		
Activity	Rate	min	max	Meaning
EngineFailure	r_1	1.0	1.0	Arbitrary value—measurement only begins at the end of this event
RequestLocation, FindServices	r_2	0.9	1.1	Location information can be transmitted in one minute, with little variance, service discovery is similar
CurrentLocation	r_3	0.25	1.25	The driver processes location information and decides to act on this, with high variance
SelectServices	r_4	1.9	2.1	The reasoner takes around thirty seconds to make a decision, with little variance
SelectServices	r_5	0.25	1.25	The on-road repair service takes slightly less than one minute to process orders, with high variance

All times are expressed in minutes. Thus a rate of 1.0 means that something happens once a minute (on average). A rate of 2.0 means that the associated activity happens twice a minute on average, or that its mean or expected duration is thirty seconds, which is an equivalent statement.

representation based on ordinary differential equations. For example, we have investigated with PEPA how models of Web Service execution scale with increasing client population sizes (see Wirsing et al. 2006).

The PEPA process algebra is a compact formal language with a small number of combinators. Components perform activities. Activities have a type and rate specified using *prefix* (.) so that (α, r) denotes performing activity α at rate r and (α, $\top$) is a partner for this where the other partner in the cooperation determines the rate. Alternative behaviors can be composed in a choice (+). Parallel composition of components uses CSP-style synchronization over a set of activity types ($\bowtie$). Private behavior can be hidden (/).

To perform quantitative analysis, we assign exponentially distributed rates to the activities of our model. Table 7.1 explains the meaning of these. In our modeling we were unwilling to assume that we even knew precisely the average duration of the atomic service events, and required only that these values lie in a range between some minimum and some maximum average value. The less certainty we have about the actual value of the rate, the wider this range must be.

We analyzed the model first using the PEPA Workbench (Gilmore and Hillston 1994), which confirmed that the model had no deadlocks and no transient states, and that all local states of all components were reachable and that all activities were live. We used the ipc/Hydra tool chain (Bradley and Knottenbelt 2004) to assess the service against the following compound service-level agreement (SLA-1) on the orchestration overall.

At least 30 percent of engine failures lead to the tow truck being ordered within 15 minutes and at least 60 percent of engine failures lead to the tow truck being ordered within 30 minutes.

At first sight it might seem that SLA-1 is so generous with the time bounds and confidence bounds that it will easily be satisfied, and that a more demanding SLA-2 could be posed instead: "30 percent of failures dealt with in 10 minutes, 90 percent in 30." However, there is a possibility that each atomic service event in the orchestration will take longer than average and that each of these is operating at its minimum rate, leading to a much longer overall time than might be expected.

We assess SLA-1 using the passage-time quantile computation capabilities provided by ipc/Hydra. We vary rates r2 to r5 across five possible values, leading to $5 \times 5 \times 5 \times 5 = 625$ experiments to be performed. The graphs of computed probability against experiment number for time bounds of 15 minutes and 30 minutes for all 625 experiments are shown in figure 7.7. Using both of these graphs, we determine that SLA-1 is satisfied across the values of the rates of the model, but that the time bounds and confidence bounds are tight, and the more demanding SLA-2 would not be satisfied by some of the combinations of rate values used in the model.

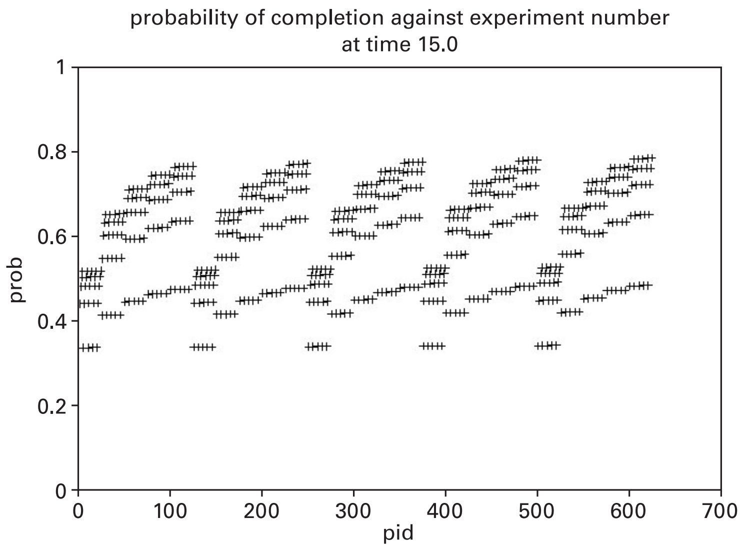
7.8 Concluding Remarks

Service-oriented computing and service-oriented architecture are having a huge impact on IT-based business organizations across the world. However, there are still many open issues regarding the development, analysis, and deployment of such software, some of which touch the very foundations of service-oriented computing.

As a remedy, the EU project Sensoria is developing a novel comprehensive approach to the visual design, formal analysis, and automated deployment of service-oriented software systems where foundational theories, techniques, and methods are fully integrated in a pragmatic software engineering approach.

Compared to other research projects in the field of service-oriented computing, Sensoria focuses heavily on scalable quantitative and qualitative analysis techniques based on formal foundations which can be embedded into a practical software development process. SODIUM,[3] in contrast, focuses on service-modeling semantic support for service discovery, but leaves aside dynamic reconfiguration, model transformation, service deployment, service extraction, and analysis techniques. PLASTIC,[4] on the other hand, is more focused toward development, deployment, and management of service-oriented adaptive applications. Resource management is a major aspect in the PLASTIC approach. The SeCSE project[5] is geared toward development and runtime support for service-oriented computing, again considering formal analysis techniques of services as a secondary field of activity. Sensoria's main research focus and asset in the service-oriented community is therefore the combination of a pragmatic approach with formal theories for service-oriented computing, supporting both development and analysis of service-oriented systems.

We have illustrated service modeling in high-level languages such as SRML or the Sensoria extension of UML, service selection with soft constraints, and analysis of qualitative



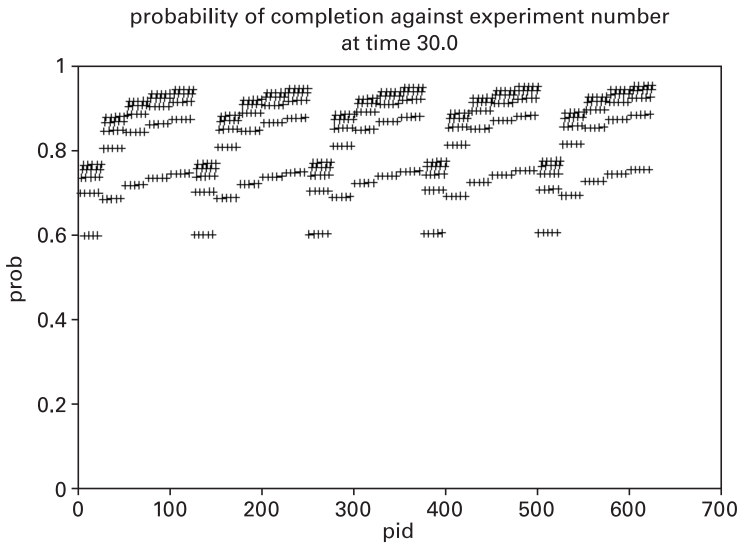


Figure 7.7
Graph of probability of completing the passage from engine failure to completion of order of tow truck within fifteen and thirty minutes plotted against experiment number over all 625 experiments

and quantitative service properties with process calculi such as COWS and PEPA. Further research of the Sensoria project addresses topics including resource consumption of services, security, reconfiguration, deployment, and reengineering of legacy systems into services. To facilitate practical application of these results, we are distributing the Sensoria development environment under an open-source license.

To learn more, please visit our website http://www.sensoria-ist.eu/.

Acknowledgment

The work published in this chapter is partly funded by the European Community under the Sixth Framework Program, contract FP6–IST-2005–016004 SENSORIA. The work reflects only the authors' views. The Community is not liable for any use that may be made of the information contained therein.

Notes

1. http://www.sensoria-ist.eu/.
2. LMU München (coordinator), Germany; TU Denmark at Lyngby, Denmark; FAST GmbH München, S and N AG, Paderborn (both Germany); Budapest University of Technology and Economics, Hungary; Università di Bologna, Università di Firenze, Universitá di Pisa, Universitá di Trento, ISTI Pisa, Telecom Italia Lab Torino, School of Management Politecnico di Milano (all Italy); Warsaw University, Poland; ATX Software SA, Universidade de Lisboa (both Portugal); Imperial College London, University College London, University of Edinburgh, University of Leicester (all United Kingdom).
3. http://www.atc.gr/sodium/.
4. http://www.ist-plastic.org/.
5. http://secse.eng.it/.

References

Beisiegel, Michael, Henning Blohm, Dave Booz, Jean-Jacques Dubray, Adrian Colyer, Mike Edwards, Don Ferguson, Bill Flood, Mike Greenberg, Dan Kearns, Jim Marino, Jeff Mischkinsky, Martin Nally, Greg Pavlik, Mike Rowley, Ken Tam, and Carl Trieloff. 2005. Building Systems Using a Service Oriented Architecture. White paper, SCA Consortium. htttp://www.oracle.com/technology/tech/webservices/standards/sca/pdf/SCA%'_White_Paper1_09.pdf.

Bistarelli, Stefano. 2004. *Semirings for Soft Constraint Solving and Programming*. LNCS 2962. Berlin: Springer.

Bistarelli, Stefano, Ugo Montanari, and Francesca Rossi. 1997. Semiring-based constraint satisfaction and optimization. *Journal of the ACM* 44(2): 201–236.

Bocchi, Laura, Yi Hong, Antónia Lopes, and José Luiz Fiadeiro. 2007. From BPEL to SRML: A formal transformational approach. In *Proceedings of the 4th International Workshop on Web Services and Formal Methods* (WS-FM '07). Berlin: Springer.

Bradley, Jeremy T., and William J. Knottenbelt. 2004. The ipc/HYDRA tool chain for the analysis of PEPA models. In *Proceedings of the 1st International Conference on the Quantitative Evaluation of Systems* (QEST 2004), pp. 334–335.

Bruni, Roberto, Gianluigi Ferrari, Hernán Melgratti, Ugo Montanari, Daniele Strollo, and Emilio Tuosto. 2005. From theory to practice in transactional composition of web services. In M. Bravetti, L. Kloul, and G. Zavattaro, eds., *Formal Techniques for Computer Systems and Business Processes: Proceedings of WS-FM 2005, 2nd International Workshop on Web Services and Formal Methods*. LNCS 3670, pp. 272–286. Berlin: Springer Verlag.

Fiadeiro, José Luiz, Antónia Lopes, and Laura Bocchi. 2006. A formal approach to service component architecture. In *Web Services and Formal Methods*. LNCS 4184, pp. 193–213. Beerlin: Springer.

Gilmore, Stephen, and Jane Hillston. 1994. The PEPA Workbench: A tool to support a process algebra-based approach to performance modelling. In *Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. LNCS 794, pp. 353–368. Berlin: Springer Verlag.

Hillston 1996. ▮▮▮

Hillston, Jane. 1996. *A Compositional Approach to Performance Modelling*. Cambridge: Cambridge University Press.

Lapadula, Alessandro, Rosario Pugliese, and Francesco Tiezzi. 2007a. Regulating data exchange in service oriented applications. In *Proceedings of the IPM International Symposium on Fundamentals of Software Engineering* (FSEN '07). LNCS 4767, pp. 223–239. Berlin: Springer.

Lapadula, Alessandro, Rosario Pugliese, and Francesco Tiezzi. 2007b. A calculus for orchestration of Web Services. In *Proceedings of the 16th European Symposium on Programming* (ESOP '07). LNCS 4421, pp. 33–47. Berlin: Springer.

Sensoria Project. 2007. Web site for the Sensoria Development Environment. http://svn.pst.ifi.lmu.de/trac/sct.

Wirsing, Martin, Allan Clark, Stephen Gilmore, Matthias Hölzl, Alexander Knapp, Nora Koch, and Andreas Schroeder. 2006. Semantic-based development of service-oriented systems. In E. Najn et al., eds., *Proceedings of the 26th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems* (FORTE '06). LNCS 4229, pp. 24–45. Berlin: Springer.

Wirsing, Martin, Laura Bocchi, Allan Clark, José Luiz Fiadeiro, Stephen Gilmore, Matthias Hölzl, Nora Koch, Philip Mayer, Rosario Pugliese, and Andreas Schroeder. 2007a. Sensoria: Engineering for Service-Oriented Overlay Computers. Technical Report 0702, Ludwig-Maximilians-Universität München, Institut für Informatik, PST.

Wirsing, Martin, Rocco De Nicola, Stephen Gilmore, Matthias Hölzl, Roberto Lucchi, Mirco Tribastone, and Gianluigi Zavattaro. 2007b. Sensoria process calculi for service-oriented computing. In Ugo Montanari, Don Sannella, and Roberto Bruni, eds., *Second Symposium on Trustworthy Global Computing* (TGC 2006). LNCS 4661. Berlin: Springer.

Wirsing, Martin, Grit Denker, Carolyn Talcott, Andy Poggio, and Linda Briesemeister. 2007c. A rewriting logic framework for soft constraints. *Electron. Notes in Theoretical Computer Science* 176(4): 181–197.