An Integrated Workbench for Model-Based Engineering of Service Compositions

Howard Foster, *Member*, *IEEE*, Sebastian Uchitel, *Member*, *IEEE Computer Society*, Jeff Magee, *Member*, *IEEE*, and Jeff Kramer, *Member*, *IEEE Computer Society*

Abstract—The Service-Oriented Architecture (SOA) approach to building systems of application and middleware components promotes the use of reusable services with a core focus of service interactions, obligations, and context. Although services technically relieve the difficulties of specific technology dependency, the difficulties in building reusable components is still prominent and a challenge to service engineers. Engineering the behavior of these services means ensuring that the interactions and obligations are correct and consistent with policies set out to guide partners in building the correct sequences of interactions to support the functions of one or more services. Hence, checking the suitability of service behavior is complex, particularly when dealing with a composition of services and concurrent interactions. How can we rigorously check implementations of service compositions? What are the semantics of service compositions? How does deployment configuration affect service composition behavior safety? To facilitate service engineers designing and implementing suitable and safe service compositions, we present in this paper an approach to consider different viewpoints of service composition behavior analysis. The contribution of the paper is threefold. First, we model service orchestration, choreography behavior, and service orchestration deployment through formal semantics applied to service behavior and configuration descriptions. Second, we define types of analysis and properties of interest for checking service models of orchestrations, choreography, and deployment. Third, we describe mechanical support by providing a comprehensive integrated workbench for the verification and validation of service compositions.

Index Terms—Service-oriented architecture, composite services, services models, Web services modeling, analysis, validation.

1 Introduction

As the adoption of a Service-Oriented Architecture (SOA) approach and the more general notion of Service-Oriented Computing (SOC) gains popularity, tool support for the increasing number of standards and complex configuration dependencies is expected. While there are specific tools for certain service aspects, there is currently little to support the engineer in building complex service interactions using complementing standards across the standard spectrum. These standards have been designed to cover the service data requirements, interface descriptions, process requirements, and behavior specifications of collaborating services yet only together will they provide a complete environment for the benefits of services to be realized. Current integrated development environments, such as Visual Studio.NET (for Microsoft.NET), focus on the function or code behind a service (illustrated by the focus of consuming or implementing the service rather than the scope of how that service is expected to be used and composed with other services). In other words, there is a gap between the provision of tools for building a service (the components and interface of a service) and the interactions that the service will provide or require in the environment that it is used.

Service orchestration languages, such as the Web Services Business Process Execution Language (WS-BPEL) [1], aim to fulfill the requirement of a coordinated and collaborative service invocation specification to support the interactions of a local process with multiple service partners. However, an orchestration alone does not fulfill the requirement of an assured collaboration in cross-enterprise service domains. Participating services must adhere to policies set out to support these collaborative roles in a services architecture with obligations to constrain the interactions between services. While policies are generally considered to be resource access based (e.g., security and access control permissions), obligations are equally important in ensuring that collaboration is conducted in an appropriate manner and that the behavior exhibited by participating clients is suitable for given scenarios. This issue is collectively wrapped up in the term Service Choreography. Recent standards efforts have produced choreography languages, such as the Web Services Choreography Description Language (WS-CDL) [2]. In addition, the design and implementation of service components in this architecture style must support the original policies as defined by the service owner and their enterprise. These interacting services can be constructed using various emerging standards and managed by multiple parties in their domain of interest and as such the task of linking these activities across workflows within this domain is crucial. Therefore, of clear interest is the need to support such engineering tasks as process verification, partner service usability, and other properties to verify the

- *H. Foster, J. Magee, and J. Kramer are with the Department of Computing, Imperial College London, Huxley Building, South Kensington Campus, London SW7 2AZ, UK.*
E-mail: {howard.foster, j.magee, j.kramer}@imperial.ac.uk.
- *S. Uchitel is with the Departamento de Computación, Universidad de Buenos Aires, Pabellon 1, Ciudad Universitaria, Buenos Aires C1428EGA, Argentina, and the Department of Computing, Imperial College London, Huxley Building, South Kensington Campus, London SW7 2AZ, UK.*
E-mail: suchitel@dc.uba.ar.

Manuscript received 12 May 2009; revised 21 Oct. 2009; accepted 6 Mar. 2010; published online 28 Apr. 2010.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSCSI-2009-05-0123.
Digital Object Identifier no. 10.1109/TSC.2010.19.

roles of service users and their actions [3]. There is also high value in providing a simulated process mechanism to visually compare expected with simulated results of service behavior which can increase expectations of a successful outcome prior to deployment.

To address the issues discussed above, we propose a holistic approach to engineering service compositions. More specifically, we abstract formal behavioral models from interacting service orchestrations, their choreography policies, and deployment architecture scenarios. The formal models are used as input in model-checking techniques to determine the safety and correctness of properties from design, implementation, and collaboration. Our aim is to provide greater assurance that service requirements are implemented safely and correctly in terms of service composition specifications and configuration.

The paper is structured as follows: In Section 2, we provide a background to the concepts of service orientation, while highlighting aspects of service behavior in service orchestrations, choreography, and deployment scenarios. In Section 3, we provide formal models of service orchestrations, choreography, and deployment through structural mappings from their design and implementation into a process model which can be compiled to form finite state machines of service composition behavior. In Section 4, we analyze these models for different behavioral properties of service compositions. In Section 5, we discuss an implementation of the analysis techniques and our experience with both industry and academia in using the techniques. Section 6 provides a discussion on this and related work in the area, and Section 7 concludes the paper with a summary and view on future work.

2 Background

2.1 Concepts of Service Orientation

The Service-Oriented Model (SOM) [4], illustrated in Fig. 1, is a relationship model described by the World-Wide Web (W3C) Services Architecture Group as *"to explicate the relationships between an agent and the services it provides and requests."* The fundamental elements for service composition are *service*, *service goal state*, *task*, and *role*. A service is an abstract resource that represents a capability of performing tasks, while its goal state is driven from the requirements of some person or organization's point of view. Furthermore, a service may take a particular role in performing a task. The model also serves to exhibit the relationships between service elements, and links these with service choreography. While the SOM model serves as a useful reference map in considering the principles of SOAs, there is clearly a gap between principle and implementation. In this work, we consider the design and implementations of service orchestrations and their choreography. This leads us to structurally map the relationships between elements of the SOM (with a focus on choreography) to formal models of behavior. For example, a *goal state* is achieved by a *service task* and is a result of a service task *action*. In effect, our approach in this paper is to specifically measure how the relationships are achieved and to ensure they are specified correctly to achieve a goal state. To achieve this, however, we shall see that this also requires mapping of other relationships (such as service interface, role, and semantics).

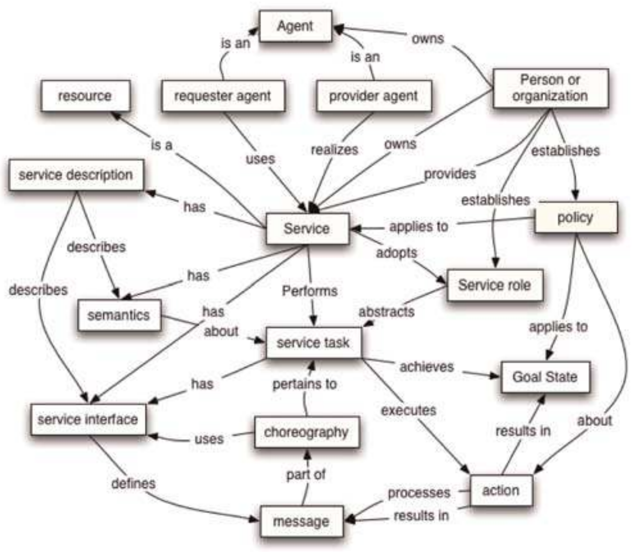


Fig. 1. The W3C SOM.

2.2 Service Choreography and Orchestration

2.2.1 Choreography and Obligations

Service choreography describes the overall behavior in a service composition between the services and their partners for one or more service goals. Choreography in general terms, and by dictionary definition, explores the wider aspects of interactions, often referenced by a similarity to arranging dance or ballet group sequences. The W3C Web Services Architecture [4] describes choreography as *"the sequence and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state."* Choreography is typically initiated by an event (trigger) or a request message and is executed by multiple partner interactions, potentially occurring concurrently between different partners. Such interactions during this choreography pose questions such as: Can messages be sent and received in any order by differing partners? What are the rules governing a sequence of choreographies? Can a global view of the overall exchange of messages between different partners be sufficiently modeled (e.g., for verification and monitoring a coordinated behavior)?

2.2.2 Orchestration and Process

A Service Orchestration is a process within a choreography that provides the necessary actions to achieve a partners obligations in that choreography. When used in the context of SOA, the orchestration part is often related to a role similar to that of a music orchestra conductor, who sequences and times the necessary steps to perform a musical act. Each musician has a role to play and performs a task to contribute to the overall score. A service orchestration coordinates the interactions between different services, offered by different service partners and providing different roles depending on the context of the orchestration. As a way to describe these service orchestrations, the WS-BPEL orchestration language was created. Note that although service choreography is referred to on the SOM model, service orchestration is not. Interestingly, one can think of orchestration as a *kind of* service (rather than a unique element of the model) produced

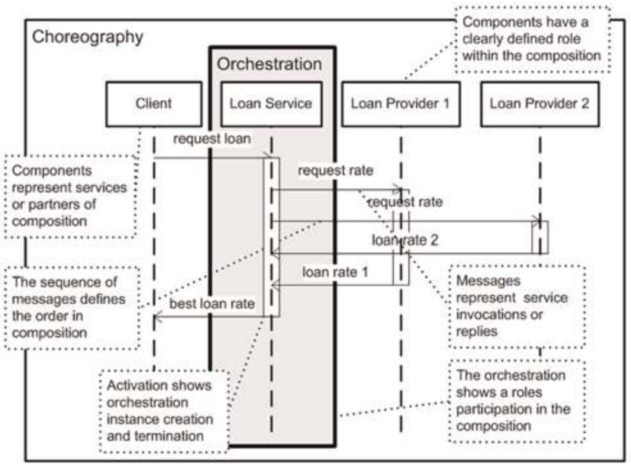

Choreography
Orchestration
Components have a clearly defined role within the composition
Client
Loan Service
Loan Provider 1
Loan Provider 2
request loan
request rate
request rate
loan rate 2
loan rate 1
best loan rate
Components represent services or partners of composition
The sequence of messages defines the order in composition
Messages represent service invocations or replies
Activation shows orchestration instance creation and termination
The orchestration shows a roles participation in the composition

Fig. 2. A scenario of service choreography and orchestration interactions in a (basic) MSC.

by a service requester or provider. A service orchestration can be implemented in any modern programming language; however, all types of orchestration languages support a number of common concepts. There is service invocation with a number of styles such as synchronous (wait for reply) or asynchronous invocation (to receive a reply at a later point). Invocations can also be performed concurrently, repeating or being chosen depending on values within the orchestration environment (i.e., using variables and conditional transitions).

2.2.3 *Abstract View of Choreography and Orchestration*

We relate the design of service choreography and orchestration as a set of scenarios (illustrated in Fig. 2) with a mapping between the elements of a basic Message Sequence Chart (bMSC) and those in building service composition specifications. Using MSCs has the benefit of building compositions incrementally. The choreography of the scenario is shown as the multipartner view of interactions across the scenario, while orchestration is the view of a single partner's interactions with other partners in the scenario. The example scenario depicts a choreography for an *obtain the best loan rate* goal. In this example, the design has focused on a central orchestration, that of the *Loan Service*, which coordinates the service goal and interacts between a client and two *Loan Providers*.

2.3 Service Deployment and Constraints

Service Engineers must also carefully consider the use of resources as part of service orchestration processes and especially when combining a number of orchestrations in a single deployment environment. As we have described in Section 2.2.2, service orchestration languages support a number of synchronization primitives to invoke other services. WS-BPEL also includes a primitive to determine the concurrent execution of a block of statements. In order to avoid concurrency problems, such as lost updates or inconsistent analyses, the language supports locking primitives so that variables that maintain state can be accessed in mutual exclusion. The combination of these primitives means that service orchestrations that are not written carefully may deadlock or exhibit other safety or liveness

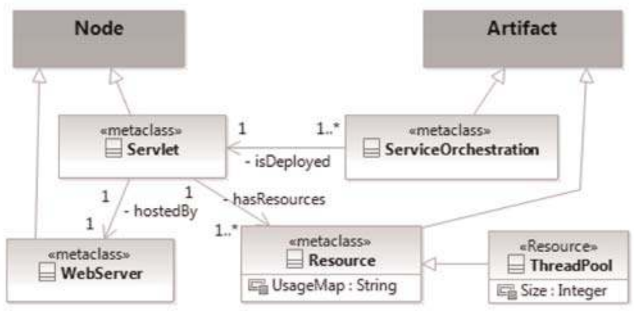


Fig. 3. A metamodel for service orchestration deployment.

property violations due to resource issues. Process related resources are typically defined in one of three groups [5], that of 1) Processor (thread pools, priority mechanisms, and intraprocess mutexes), 2) Communication resources (protocol properties, connections, etc.), and 3) Memory (buffering requests in queues and bounding the size of a thread pool). One such resource that is commonly configured with multiple process instances and interactions is that of a shared thread pool.

2.3.1 *Abstract View of Deployment*

There are a number of Architecture Description Languages (ADLs) we could use to describe a deployment architecture, including Darwin [6] and UML [7]. An abstract view, however, defines a metamodel that can be used in any ADL. One such metamodel is illustrated as UML2 in Fig. 3, showing the relationships between service artifacts and system architecture nodes for service deployment. One or more service orchestrations (of type ServiceOrchestration) are modeled as artifacts which are deployed on to servlet nodes. A service orchestration can only be deployed to one servlet instance. Servlets are hosted on Web server nodes (a Web server is a Web container which manages the creation and deletion of servlet instances). A servlet also has predefined resource allocations, which are modeled as one or more objects of type Resource node.

3 Modeling Service Compositions

In this section, we describe how to create formal models of service composition design, implementations, and deployment configurations. Formal models are constructed using a process algebra which represents the behavioral and architectural configuration of service compositions.

3.1 Behavior Semantics in FSP

We define the behavioral and structural semantics of each service composition artifact in terms of a Labeled Transition System (LTS) [8]. Labels can represent different things depending on the context the system is used in. Typical uses of labels include representing input expected, conditions that must be true to trigger the transition, or actions performed during the transition. We use LTSs to describe the formal behavior of service specifications, both in design and implementation models. LTSs can be modeled using the Finite State Process (FSP) notation [9] which can be compiled into LTSs using the Labeled Transition System Analyzer (LTSA) tool [10]. FSP is a textual notation (technically a process calculus) for concisely describing and reasoning about concurrent programs. FSP is designed to be easily machine readable, and thus provides a preferred language to specify abstract processes. FSP supports a range of operators to define a process model representation which is given in an online [11]. Initially, to enable a common representation for service interactions we define a template for two partners, a type of interaction and an interaction operation name. These interaction templates are labeled *p1_p2_primitive_op* where p1 is the local process partner name, p2 is the service partner, primitive is the activity and op is the name of the operation requested.

The mappings to support our analysis are provided complete in an Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSC.2010.19.

3.2 Interaction Design Models

In this section, we discuss the use of MSCs for service composition modeling and how MSC design models are synthesized to FSP models to represent service composition behavior.

3.2.1 Specification

MSCs can provide visual aids to design requirements specifications for service compositions, yet their combined behavior is difficult to analyze by human observation. We have already provided an example of an MSC for service compositions in Fig. 2. To represent different service composition scenarios, we base our models on the International Telecommunications Union Telecommunication Standardisation Sector (ITU-T) recommendation Z20 [12] which provides two levels of sequence chart composition. First, there is a basic MSC (bMSC) which defines the components and message sequences between them. Second, there is a high-level MSC (hMSC) which defines the ordering of the bMSC. In this way, a requirements engineer can specify the different scenarios for message sequencing and compose these to a service system architecture model. Note that the ITU-T Z20 recommendation aligns with that of the Unified Modeling Language Version 2 (UML2) Sequence Chart notation (particularly on grouping messages). The process of synthesizing these MSC scenarios to LTSs provides a way to computationally and mechanically analyze these scenarios to determine whether the behavior specified is desirable given a complete system behavior model. To aid accessibility in design, we also support the mapping of UML2 Sequence Charts.

3.2.2 Mappings and Models

The semantics of MSC message names (or labels) is not constrained. As such, we can apply the template pattern (described in Section 3.1) to message names in these specifications to associate the type of message activity for service interactions, where c1 (component 1) in the MSC represents p1 (partner 1) and c2 represents p2 (partner 2). Additionally, a type of interaction represents an invocation (invoke), receive (receive), or response (reply). This template is only necessary if the ITU-T recommendation is followed, otherwise if it is UML2 then the language of UML2 Sequence Charts includes a direction indicator of

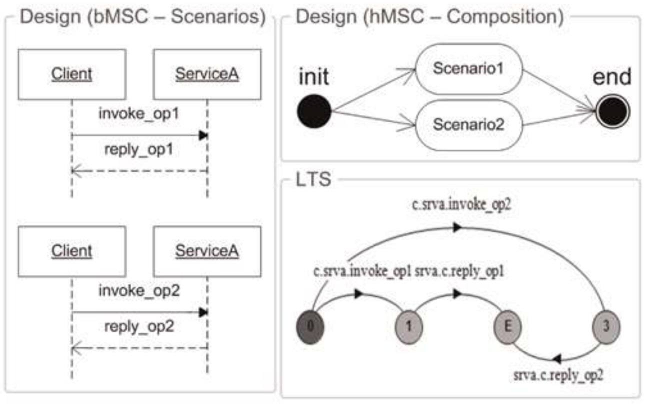


Fig. 4. Basic and high-level MSC and synthesis to LTS model.

messages (i.e., invoke, reply, synchronous, or asynchronous, etc.). Thus, we have a set of profiles which can be applied to MSCs depending on the language used to specify them. Referring back to Fig. 4, note that the LTS transitions are labeled with abbreviated partner names for visual clarity (e.g., c for Client, srva for Service A).

A formal syntax and semantics for MSCs based upon the ITU-T recommendation and a corresponding algorithm to synthesize MSCs to LTSs is described in [13]. Once the MSCs are specified, we use this algorithm to synthesize LTS models from MSC specifications. The general idea of the algorithm is to build local FSP processes that correspond to portions of MSC component behavior and to combine them in such a way as to provide a complete MSC behavior. The algorithm identifies beginning and end states of each component, in each MSC scenario, and then combines their instance and behavior relationships into a single behavior architecture model for analysis.

3.3 Service Orchestration

3.3.1 Specification

We use the WS-BPEL specification as an example implementation of service orchestrations. WS-BPEL defines a series of constructs to describe a service composition process, where a local partner in the composition executes a series of service interactions. A basic structure for a WS-BPEL process is outlined in Fig. 5.

3.3.2 Behavioral Mapping to FSP

Our behavioral mapping of WS-BPEL to FSP groups activities by their related areas in specification. Primitive activities in WS-BPEL are those which define basic interaction activities between the local process and services defined by partners of the orchestration, such as invoking a partner service operation or receiving and replying to an operation request from a service partner. Interaction primitives are modeled in FSP with labels for partners, the type of interaction (either *invoke*, *receive*, or *reply*) and the name of the operation. Additionally, the *terminate* primitive takes any labeled transition activity in the orchestration and immediately transitions to the end state of the process. Our orchestration model includes an action set (a list of actions in the process). The terminate activity is represented by a choice at each action state to either continue or end the process. Any primitive activity may also have external

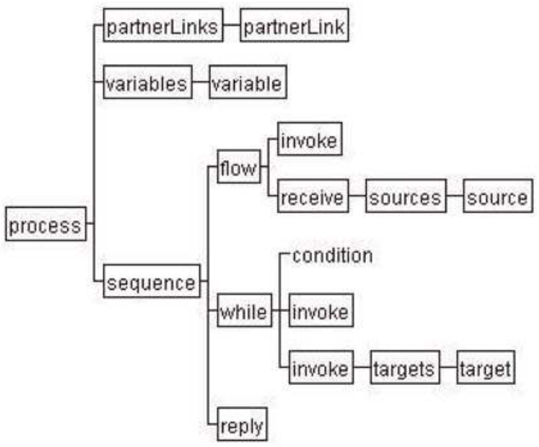


Fig. 5. Example structure of a WS-BPEL orchestration.

dependencies outside of the scope (specified as a structured link activity).

Structured activities are those which define ordering or behavioral constraints in the sequencing of activities, such as whether activities are executed in sequence, concurrently or are linked to the successful completion of other activities. The *sequence* activity construct is used to scope a sequence of activities in the order they are given in that scope. We represent sequence activities as a sequence composition process in FSP. Alternatively, a *flow* creates an execution scope that executes each activity in the scope concurrently. The flow construct maps directly to a parallel composition process in FSP. Linked activities, represented by an attribute of either *target* or *source* are pre- and postactivity execution conditions. They are used to guard when activity transitions can be made given the requirement that other activities have successfully completed. Both source and target links are modeled as synchronized sequence activities in the FSP model. The *while* activity provides a construct to perform a repetitive execution of activities until a boolean condition is evaluated to true. The while activity is represented in FSP in two parts. First, using the variable expression evaluation described at the beginning of this section. The second part is to represent recursion and use the FSP if-then-else with alternative process transitions depending on whether the evaluation is true or false. The *switch* construct is also a conditional activity, selecting either a particular case or an "in all other cases" path of execution. Each *case* branch builds an FSP guarded activity, while the last case activity model includes an *otherwise* activity. Finally, the pick activity awaits the occurrence of one event in a set of events and then performs the activity associated with the event that occurred. The events can occur in any order; hence, we represent the available activities as a choice composition in FSP. Fig. 6 illustrates some structural activity mappings to LTSs.

Scoping can be used to group activities and declare handlers for either local or global fault-handling and compensation recovery actions. Scoping is related event handling in WS-BPEL, to provide a mechanism to support concurrently receiving messages while the orchestration process is executing. To model these activities, a sequence composition is used to model the event activity which is then composed with a global event manager. The set of global

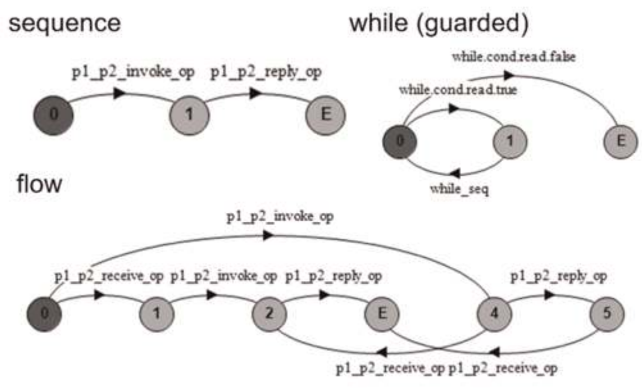


Fig. 6. Example WS-BPEL structure activities as LTSs.

event actions are collated for the entire orchestration process. In a similar way to event handling, fault handling provides a mechanism to capture error events (such as the failure to invoke a particular service). The difference to the more generic event handling is that *fault handler* activities (when activated) cause the immediate terminate or nonfault activities in the scope. Thus, we need to model the fault handler activities as alternative paths in the model. This is achieved using a guarded sequence composition in FSP. The events which identify faults are added to the scope composition and in a similar model to the terminate activity, a fault event raised causes the nonfault activity processes to end while execution continues with the relevant fault sequence process. Finally, compensation handlers are very similar to fault handlers; however, compensation focuses on concurrent recovery actions rather than exception handling and can be executed directly with the *compensate* activity.

3.3.3 Interaction Models

WS-BPEL, hosted as a Web service, requires an interface description to advertise its offered services (methods) and message types. This description is in the form of the Web Service Description Language (WSDL), which specifies the service ports, operations, and message data types used. To link service interactions, we build port connector models for interacting orchestration processes by use of an interaction matching process. The process is summarized as follows: For every orchestration process in a composition, we extract all the interaction activities (i.e., invoke, receive, and reply). For each invocation activity, a partner role is selected and a partner service port referenced. The port is used to determine which connector model is applicable. Given a selected interface definition, the operation for invocation is referenced. Finally, depending on the invocation style (i.e., synchronous or asynchronous), a port connector bridge is built (synchronizing the interaction activities). The sequence is then repeated for all other invocations in the selected and other composition processes. In WS-BPEL, invocations specified with the *invoke* construct and only an input variable declare a one-way or asynchronous operation. To model this, we use an asynchronous port model (Fig. 7a). Conversely, invocations specified with both input and output variables define a synchronous request and reply ("rendezvous") interaction. To model this, we use a synchronous port model (Fig. 7b).

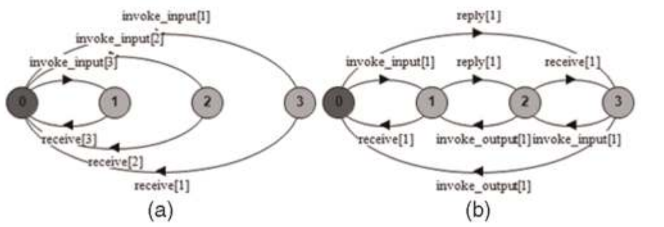


Fig. 7. Port connector models for asynchronous (a) and synchronous (b) interactions.

3.3.4 Architecture and Analysis Models

A complete architecture model for a WS-BPEL process and its interactions is built to represent the composed behavior of the service orchestration activities. The architecture model can then be composed as a unique single identifier when combining different orchestration architecture models in a single service composition analysis model.

3.4 Service Choreography

3.4.1 Specification

We use the WS-CDL specification as an example language to describe service choreography and map the constructs of this language to the semantics of LTSs using FSP models. An example choreography specification of WS-CDL is provided in Fig. 8. A WS-CDL package consists of choreographies that specify one or more scenarios of interaction activities between different partners. At the choreography package level, general aspects common to all choreographies are defined, for example, participant roles, types, channels for communication, and information (data) sets. Within each choreography scenario are activities which specify interactions, exceptions, workunits, and finalization steps.

3.4.2 Structural Mapping to FSP

Primitive activities in WS-CDL are similar to those for interactions in WS-BPEL; however, in addition, the *perform* construct passes the control flow to a named choreography in the specification. The perform activity is modeled as a sequence composition of a choreography process in FSP. Also, the primitive actions of *silentAction* and *noAction* indicate either a nonobservable action that should take place or that no action should take place for a particular partner in the choreography, respectively. Both silentAction and noAction are transformed to a FSP transition action. Defined at the beginning of a choreography package are participant, relationship, and communication types. These are defined using the participantType, relationshipType, roleType, and channelType, respectively. All types generate a mapping list (to build a reference map of participants and their relationships). These are also referenced later in analysis. More significant is the use of channelTypes. A channelType realizes a point of collaboration between participantTypes (with a specific roleType) by specifying where and how information is exchanged. A channelType can declare a channel that is used for one interaction (once), or used by only one participantType (distinct) or that it can be used multiple times by different participantTypes (shared). Additionally, a channelType can also be restricted to a type of exchange, either both request and respond (request-respond), requests only (request), or responses only (respond). The declaration and restrictions on using a channel are included in the internal mapping built.

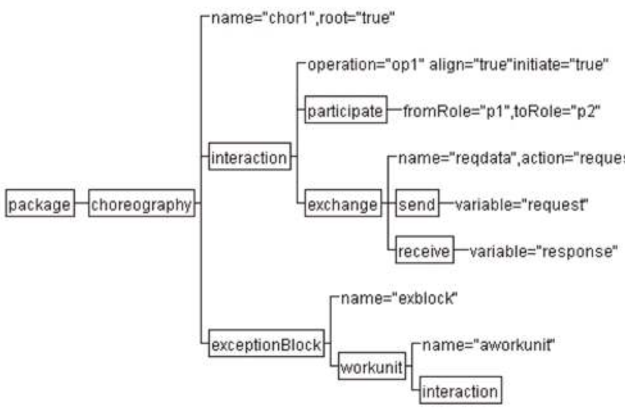


Fig. 8. Example structure of a WS-CDL choreography.

An *Interaction* in WS-CDL is a complex construct to define an interaction between two partners in the choreography. The interaction construct specifies a participate child element which declares the participation relationshipType, and the roleType to and from of the participants. Additionally, another child element specifies the exchange of information, naming the operation (service method) and the informationType. A child of the exchange element specifies the send and receive variables, along with an optional exception handler (causeException) to cater for errors that may occur during an exchange. We build an action in FSP for each of the interaction elements, creating a sequential composition of the form *p1_p2_type_op* (where type is either request, receive, or respond).

Structured constructs in WS-CDL are also similar to those found in WS-BPEL, including *sequence*, *parallel* which is identical to the WS-BPEL *flow* construct and *choice*. Hence, we reuse the semantics used for WS-BPEL and apply the same translation to a FSP. WS-CDL also provides the structure called *workunit*. A workunit can be used to scope a set of activities for execution, having a guarded condition for execution and also repeating if necessary. Thus, the workunit in WS-CDL is represented as a *while* and *if* type construct.

Fault handling in WS-CDL is declared using *exceptionBlocks*. An exceptionBlock is identified by a unique name and can be referenced in the causeException part of the exchange activity. An exceptionBlock must define a *workunit* to describe the behavior of actions to take if an exception is raised. To model this in FSP, we create a new sequential composition process which includes the translation of all activities defined in the workunit and alternative transition path for the exceptionBlock.

3.4.3 Interaction Models

An interaction channelType for a particular partnershipType might be defined to be only of action type *respond*, while a choreography interaction exchange may attempt a *request* on this channel. For interaction models, we declare a *PortModel* (again, similar for those defined for WS-BPEL) which composes the types of exchange actions (request, receive, or reply) with that of the restrictions specified on a particular channel. Choreography interaction models allow us to link orchestration and choreography models, while also checking constraints on channel usage.

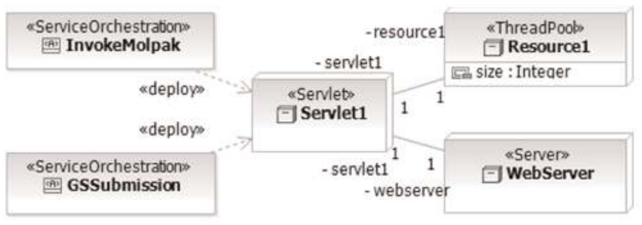


Fig. 9. A service orchestration deployment configuration model.

3.4.4 *Architecture and System Models*

A complete model for WS-CDL choreography specifications and their interactions is built to represent the architecture of the service choreography package. This composes the models produced from translation of each choreography in the WS-CDL specification to FSP, their channel port connector models, and the interactions specified on these channels. As with the WS-BPEL architecture models, we generate unique behavior models based upon the name of each choreography, while the overall package specification provides a system model of the composed choreography architecture models.

3.5 Service Deployment

3.5.1 *Specification*

Using the metamodel for service deployment described in Section 2.3.1, we provide an example deployment diagram in Fig. 9 showing two service orchestrations (named "InvokeMolpak" and "GSSubmission") which are configured to be deployed to a single servlet. The diagram is related to an analysis case study described later in Section 4. The servlet has a designated resource of type ThreadPool. Note that the resources can be stereotyped to define which resource type is being specified. The servlet is also associated with a Web server, which acts as a container for one or many servlets.

3.5.2 *Structural Mapping to FSP*

First, each service orchestration (as WS-BPEL) is identified in the deployment design model and mapped to FSP as described in Section 3.3. The orchestration translations are stored in a lookup list along with their corresponding architecture model. The number of orchestrations in the model is also stored, for reference later in process mapping. Second, each Servlet element is selected and again a list generated for reference. Each servlet is then considered for associated resource elements and identified as a particular type. The type we currently focus on is of type *ThreadPool*; however, other resource types may be introduced in future work. We model the management of threads in a shared thread pool as a sequence of processes which "get" or "put" a resource from a container (Fig. 10). The shared thread pool (TPOOL) is a container for *poolsize* number of threads and represents the service orchestration server technology stack for allocating and releasing threads as required by the orchestration processes. When a process is composed with this thread pool, those interactions which acquire a thread (represented by the first conditional statement of "$(\mathrm{t} > 0)$ $\mathrm{get} \rightarrow \mathrm{TPOOL}[\mathrm{t} - 1]$") increase the acquired threads by one if there exists unallocated threads in the pool. Alternatively, a completed interaction may free a thread which is represented by the statement "$\mathrm{put} \rightarrow \mathrm{TPOOL}[\mathrm{t} + 1]$," adding a thread back to the available resources. Each servlet, if associated with a ThreadPool is composed with one or more TPOOL processes to include resource usage within the behavior model. Finally, a SERVER process is created which composes both the service orchestrations and the servlet composition SERVLETS.

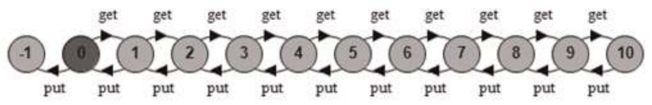


Fig. 10. LTS for a shared ThreadPool Resource model.

3.5.3 *Interaction and Resource Models*

To associate the actions of service orchestrations and resource usage, we map the service orchestration actions to particular resource model actions. First, we add an action for the create and terminate service orchestration instances. In the WS-BPEL specification, a process may be instantiated by containing at least one "start activity." This may either be designated on a "receive" activity or a "pick" through the use of a "createInstance" attribute. There is no restriction for the number of activities which may create an instance of a process, and there are further semantics for how these correlate on a given process. Therefore, a createInstance action can occur in multiple activities, but only one may actually create an instance of a process. In our current mapping capability, we assume that one activity will be designated to create an instance of a process. For behavior modeling purposes, it is only necessary to include a create and terminate action in the mapping model (immediately before the first activity and after the last activity, respectively).

Second, for each orchestration architecture, we scan each of the orchestration process interactions and gather those which are resource-operator activities (activities which cause acquisition or release of resources). In the case of the WS-BPEL notation, these are invoke, receive, or reply. Additionally, we add a resource and activity mapping for the createInstance and terminateInstance activities. These are added as a resource "get" and "put," respectively. A final task is that of generating a pool-resource-map process. To generate this, we need to define a process stub for each combination of orchestrations sharing the server resource pool and represent that a number of instances of these processes can exist at a given time.

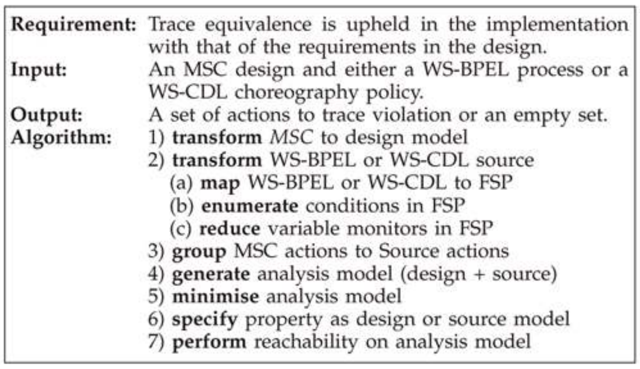
3.5.4 *Architecture and System Models*

A complete model for deployment, linking the service compositions and mappings is built to represent an analysis model of the service composition deployment behavior. This combines the behavior translated from the service orchestrations and the behavior mappings between the orchestration interactions and resource usage associated with the servlets.

4 Analyzing Service Compositions

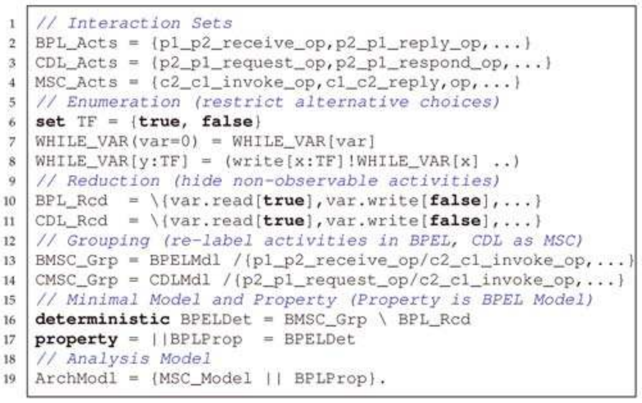
4.1 Overview

In this section, we provide service composition analysis examples from two case studies. In design, interactions,



```
Requirement: Trace equivalence is upheld in the implementation
             with that of the requirements in the design.
Input:       An MSC design and either a WS-BPEL process or a
             WS-CDL choreography policy.
Output:      A set of actions to trace violation or an empty set.
Algorithm:   1) transform MSC to design model
             2) transform WS-BPEL or WS-CDL source
                (a) map WS-BPEL or WS-CDL to FSP
                (b) enumerate conditions in FSP
                (c) reduce variable monitors in FSP
             3) group MSC actions to Source actions
             4) generate analysis model (design + source)
             5) minimise analysis model
             6) specify property as design or source model
             7) perform reachability on analysis model
```

Fig. 11. Algorithm for design analysis.



```
1  // Interaction Sets
2  BPL_Acts = {p1_p2_receive_op,p2_p1_reply_op,...}
3  CDL_Acts = {p2_p1_request_op,p2_p1_respond_op,...}
4  MSC_Acts = {c2_c1_invoke_op,c1_c2_reply,op,...}
5  // Enumeration (restrict alternative choices)
6  set TF = {true, false}
7  WHILE_VAR(var=0) = WHILE_VAR[var]
8  WHILE_VAR[y:TF] = (write[x:TF]!WHILE_VAR[x] ..)
9  // Reduction (hide non-observable activities)
10 BPL_Rcd  = \{var.read[true],var.write[false],...}
11 CDL_Rcd  = \{var.read[true],var.write[false],...}
12 // Grouping (re-label activities in BPEL, CDL as MSC)
13 BMSC_Grp = BPELMdl /{p1_p2_receive_op/c2_c1_invoke_op,...}
14 CMSC_Grp = CDLMdl /{p2_p1_request_op/c2_c1_invoke_op,...}
15 // Minimal Model and Property (Property is BPEL Model)
16 deterministic BPELDet = BMSC_Grp \ BPL_Rcd
17 property = ||BPLProp  = BPELDet
18 // Analysis Model
19 ArchModl = (MSC_Model || BPLProp).
```

Fig. 12. FSP of abstraction for analysis model.

and obligations analysis we illustrate the approach using a case study from the UK Police IT Organisation (PITO). The case study and approach are fully described in [14]. In one scenario, a Police Officer performs an enquiry on a suspect which is composed of concurrent interactions between a central enquiry service, vehicle registration service, number plate service, and personal enquiry service. For deployment analysis, we undertook a case study with the Software Engineering Group of University College London (UCL), who had experienced deadlock situations when executing a service composition (called Molpak) for predicting polymorphs in organic crystal structures [15]. Although the original deployment configuration (illustrated in Fig. 9) appears simple, the nature of the interactions provided a challenging model-checking scenario. A client invokes a central orchestration which invokes up to 38 InvokeMolpak orchestrations in parallel. Each of these orchestrations invokes the GSSubmission orchestration to submit jobs to a grid resource manager. The GSSubmission polls a resource manager awaiting completion of each analysis task.

Common to all types of analysis are a series of steps to abstract a combined model of specification and implementation in preparation for analysis. The types of abstraction we use are: **Enumeration**—representing the range of the values of a continuous variable as a set of abstracted terms (partitioning variables in to value parts), **Reduction**—the technique to decrease the size of individual parts of a system while preserving relevant characteristics needed to verify the behavior of the system, **Grouping**—the many-to-one mapping of variables or entities (actions) into a single descriptor.

Note that the accuracy of the analysis and algorithms presented here is based upon the correctness and coverage of operator mappings detailed in Section 3.

4.2 Design Analysis

Service composition *Design Analysis* focuses on the verification of the implementations of composition interaction sequences compared with that of a design formed by the possible scenarios that a service orchestration can fulfill. The essence of this verification is to prove that trace equivalence is upheld in the service composition implementation and the requirements specified of it in the design models. However, it is also the case that the design model can be checked against that of the implementation. Switching properties provide a mechanism to check additional behaviors observed in both models.

4.2.1 Process

An algorithm for design analysis is illustrated in Fig. 11. The process takes as input an MSC design specification, together with a service orchestration implementation or a choreography policy specification. First, a pair of LTSs is generated from the design model and the orchestration or choreography implementations. The translation mapping of the MSC to LTS uses the technique discussed in Section 3.2.2, while the WS-BPEL to LTS steps are discussed in Section 3.3.2, and WS-CDL steps in Section 3.4.2.

The abstraction steps are further illustrated as FSP examples in Fig. 12, initially building a set of interactions for each source model (lines 2-4). To abstract the orchestration and choreography models for design analysis, we enumerate the variables used in control-flow constructs (e.g., choice, while, etc.). In the case of a *choice* construct, the choice alternatives are numbered 0..N (where N is the number of alternatives). In the case of a *while* construct, the enumeration is either *true* or *false* on the condition specified (lines 6-8). Enumerating these variables allow us to reduce the possible alternative transition paths by a lookup of previously assigned values and conditions using the variables. The reduction also hides the FSP variable *.read* and *.write* actions, although optionally these can be selected as visible to enable interactive animation of the process. This reduction does not change the behavior of the orchestration or choreography models, but hides the actions from the complete activity set used in model analysis, as these are not normally specified in MSC design specifications.

We then group the interactions in all models which require us to relabel the interaction actions between the design specification and implementation models (lines 13-14). First, we map the MSC activities (e.g., invoke_op (c1c2)) by replacing c1 with p1, and c2 with p2. Furthermore, if c2 invokes c1 but the WS-BPEL interactions represent this as a receive (i.e., they are equivalent actions) then these must also be mapped. Similarly, if a WS-CDL interaction *requests* from p1 to p2, then these must be mapped to an *invoke* in the design specification. Receive and reply interactions are handled using the same principle.

A final preparation activity to perform analysis is to produce a model which represents a minimal, deterministic representation and specify the property model for analysis. A minimal model means that a trace in the original process

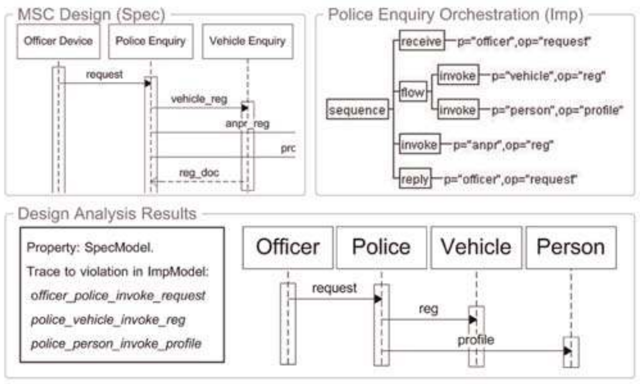

MSC Design (Spec)
Officer Device
Police Enquiry
Vehicle Enquiry
request
vehicle_reg
anpr_reg
prc
reg_doc
Police Enquiry Orchestration (Imp)
sequence
receive p="officer",op="request"
flow
invoke p="vehicle",op="reg"
invoke p="person",op="profile"
invoke p="anpr",op="reg"
reply p="officer",op="request"
Design Analysis Results
Property: SpecModel.
Trace to violation in ImpModel:
officer_police_invoke_request
police_vehicle_invoke_reg
police_person_invoke_profile
Officer
Police
Vehicle
Person
request
reg
profile

Fig. 13. Results for design analysis of MSC and WS-BPEL.

```
Requirement: Check the composition of interactions in collaborating
             service orchestrations.
Input:       One or more pairs of WS-BPEL processes and inter-
             face (WSDL) specifications.
Output:      A set of actions to trace violation or an empty set.
Algorithm:   1) generate WS-BPEL interface table
             2) for each WS-BPEL process do
                (a) transform WS-BPEL process to imp model
                (b) for each invoke activity do
                    1. lookup partner in interface table
                    2. if partner exists
                       a. get port for partner linktype
                       b. generate port connector for interaction
                       c. add port connector to process port list
                (c) for each port connector in process port list do
                    1. add port to imp model
             4) generate analysis model (from set of processes)
             5) perform reachability on analysis model
```

Fig. 14. Algorithm for interactions analysis.

leads to an END state if and only if the trace leads to an END state in a deterministic process. The step combines the grouped BPEL interactions and reduced BPEL activities (line 16). An architecture model is produced by composing the property (line 17) and MSC model together (line 19).

4.2.2 Results

In the example illustrated in Fig. 13, we show a verification and violation of the Police Enquiry orchestration against an MSC design specification. In this case, the resulting trace violation is due to an incorrect implementation of concurrent invocations for *vehicle* and *person* requests. In the MSC design specification, it is expected that *anpr* immediately follows a *vehicle* request. The service engineer can examine the violation and decide whether the corrective action is to enhance the design specification to accommodate the behavior or to modify the orchestration.

4.3 Interactions Analysis

The aim of interactions analysis is to check the composition of collaborating service orchestrations. Orchestrations that collaborate will undertake an interaction cycle, whereby one invokes a service operation on a partner and the partner must oblige by receiving this request. If the client of the service expects a reply to be given to this request then the partner must provide a reply (either by specifically replying or invoking a response back to the client).

4.3.1 Process

An algorithm for interactions analysis is illustrated in Fig. 14. First, there is an interface compatibility check which ensures that the service interface description of the partner and the definition of the parter by the client are aligned. Second, the interaction cycles (defined by interaction ports, as discussed in Section 3.3.3) are checked for any violations. The inputs required for both these checks are a set of service orchestrations and their service interfaces. Given these inputs, a series of model abstractions are carried out as follows: Enumeration is repeated as for the design analysis enumeration focusing on the variables used in control-flow constructs. Reduction is also repeated as for the design analysis; however, for interactions analysis each port connector model is analyzed as to whether there exists a partner orchestration included in the analysis. If a partner does not exist for an invoke interaction (by an unknown partner result from the matching algorithm detailed in Section 3.3.3), the port connector model for the client is removed to hide this from analysis. Grouping is already included in the port connector models of the orchestrations; however, to complete an analysis model, the orchestration processes and port connector models are composed together to form a single analysis model.

4.3.2 Results

In the example illustrated in Fig. 15, we show a verification and violation of the interactions specified between the Police Enquiry and Vehicle Enquiry orchestrations. In this case, the resulting violation is due to an incorrect sequencing of *reply* and *receive* in the Vehicle Enquiry orchestration. The service engineer can examine the violation and decide appropriate corrective actions, in this case, switching the order of *reply* with *receive*.

4.4 Obligations Analysis

The aim of obligations analysis is to compare multipartner service choreography policies and their *obligations* with that of individual partner service implementations. The obligations analysis considers one partner's role in the choreography and checks their obliged interactions set out in the choreography policy. Using the same approach, each partner implementation in the choreography can be checked. In fact, one of the reported aims of WS-CDL is to be used as a

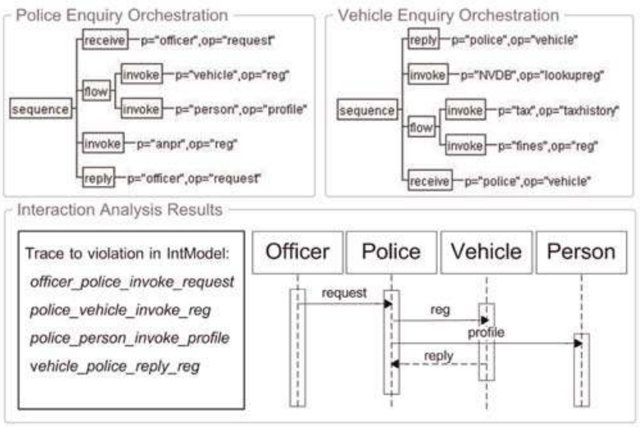

Police Enquiry Orchestration
sequence
receive p="officer",op="request"
flow
invoke p="vehicle",op="reg"
invoke p="person",op="profile"
invoke p="anpr",op="reg"
reply p="officer",op="request"
Vehicle Enquiry Orchestration
sequence
reply p="police",op="vehicle"
invoke p="NVDB",op="lookupreg"
flow
invoke p="tax",op="taxhistory"
invoke p="fines",op="reg"
receive p="police",op="vehicle"
Interaction Analysis Results
Trace to violation in IntModel:
officer_police_invoke_request
police_vehicle_invoke_reg
police_person_invoke_profile
vehicle_police_reply_reg
Officer
Police
Vehicle
Person
request
reg
profile
reply

Fig. 15. Results for orchestration interactions analysis.

```
Requirement: Check that partner service implementations fulfill
             choreography policies.
Input:       One or more WS-BPEL processes and one WS-CDL
             choreography specification. Property as either name
             of partner role or empty (for all partner roles).
Output:      A set of actions to trace violation or an empty set.
Algorithm:   1) transform WS-CDL to spec model
             2) for each WS-BPEL process do
                (a) transform WS-BPEL process to imp model
                (b) map interactions to spec activities
                (c) build exception handling list
                (d) map throw to CDL causeException
             3) if role is not empty
                (a) hide activities not equal to role action
                (b) set property to role orchestration
             4) generate analysis model (imp+spec)
             5) perform reachability on analysis model
```

Fig. 16. Algorithm for obligations analysis.

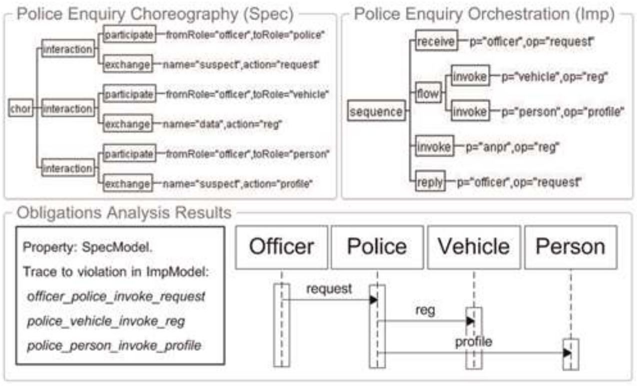

Police Enquiry Choreography (Spec)
chor
interaction
participate
fromRole="officer",toRole="police"
exchange
name="suspect",action="request"
interaction
participate
fromRole="officer",toRole="vehicle"
exchange
name="data",action="reg"
interaction
participate
fromRole="officer",toRole="person"
exchange
name="suspect",action="profile"
Police Enquiry Orchestration (Imp)
sequence
receive
p="officer",op="request"
flow
invoke
p="vehicle",op="reg"
invoke
p="person",op="profile"
invoke
p="anpr",op="reg"
reply
p="officer",op="request"
Obligations Analysis Results
Property: SpecModel.
Trace to violation in ImpModel:
officer_police_invoke_request
police_vehicle_invoke_reg
police_person_invoke_profile
Officer
Police
Vehicle
Person
request
reg
profile

Fig. 17. Results for obligations analysis.

specification which can be distributed between partners. This approach assists partners in checking their own implementations against the wider multipartner policy.

4.4.1 Process

The obligations analysis algorithm (Fig. 16) takes as input a WS-CDL choreography policy specification and one or more service orchestration implementation in WS-BPEL. The abstractions necessary from the models produced by these inputs and translation to FSP are as follows: The enumeration is repeated as for the design analysis. Reduction is also repeated as for the design analysis; however, the interactions of the service choreography policy will initially contain all actions by all specified roles in the choreography. The abstraction hides any actions that are not identified as the selected partner role for analysis. In a similar approach to the grouping abstraction in the design analysis, the actions of the choreography and service partner orchestrations must be mapped. This step of abstraction first groups the related request and response actions in the choreography specification with invoke and reply actions in the service orchestrations. Second, the fault handling in WS-BPEL can be mapped to catching exceptions in the choreography policy by mapping the *throw* construct actions of the orchestration to the *causeException* of the choreography.

4.4.2 Results

In the example illustrated in Fig. 17, we show a verification and violation of the Police Enquiry orchestrations when checked against an overall Choreography policy. The resulting violation is similar to the design check in Section 4.2, where there exists an incorrect implementation of concurrent invocations for *vehicle* and *person* requests. In the Choreography policy specification, it is expected that *person* immediately follows a *vehicle* request; however, the concurrent operation of these invocations in the orchestration may mean that the order is reversed. The service engineer can examine the violation and decide whether the corrective action is to enhance the choreography specification to accommodate the behavior or to modify the orchestration.

4.5 Deployment Analysis

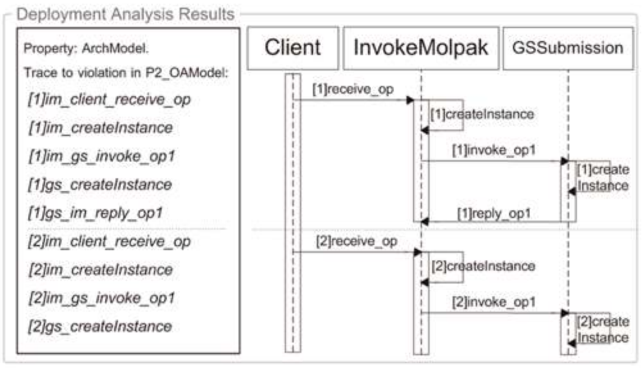
In more complex orchestrations, or where there are multiple processes hosted in a single servlet, the assessment of resources required and the ability of the infrastructure to support multiple client requests becomes increasingly difficult to estimate. A process model, however, can provide a formal specification of the interactions and can be composed with resource models to detect where possible deadlocks may occur given a limited number of resources available. We outlined this as a composed model of service orchestration and deployment diagrams in Section 3.5.4. The aim of deployment analysis is to check whether the behavior specified in service orchestrations and the available resources as part of a deployment description are safe given a number of instances and interactions of the service orchestrations.

4.5.1 Process

A deployment analysis algorithm (illustrated in Fig. 18) takes as input a set of service orchestrations in WS-BPEL and a deployment diagram. The modeling is as described in Section 3.5; however, there are a number of preparation steps to be undertaken prior to the analysis of the model produced. The enumeration abstraction is repeated as for the design analysis enumeration, focusing on the variables used in service orchestration control-flow constructs. Additionally, a number of instances of the service composition (number of client requests to start the orchestrations)

```
Requirement: Check that deployment configuration of compositions
             and their behaviour does not lead to deadlock in
             resource usage.
Input:       One or more WS-BPEL processes, their interface
             (WSDL) specifications and one deployment diagram.
Output:      A set of actions to trace violation or an empty set.
Algorithm:   1) transform deployment diagram to spec model
             2) for each servlet in spec model do
                (a) build resource model
             3) for each WS-BPEL process do
                (a) transform WS-BPEL process to imp model
                (b) for each invoke activity do
                    1. lookup partner in interface table
                    2. if partner exists
                       a. get port for partner linktype
                       b. generate port connector for interaction
                       c. add port connector to process port list
                (c) for each port connector in process port list do
                    1. add port to imp model
                (d) add create and terminate instance activities
                (e) map activites to resource model
                (f) add resource model to spec
             4) generate analysis model (imp+spec)
             5) perform reachability on analysis model
```

Fig. 18. Algorithm for deployment (resource) analysis.


Deployment Analysis Results

Property: ArchModel.
Trace to violation in P2_OAModel:
[1]im_client_receive_op
[1]im_createInstance
[1]im_gs_invoke_op1
[1]gs_createInstance
[1]gs_im_reply_op1
[2]im_client_receive_op
[2]im_createInstance
[2]im_gs_invoke_op1
[2]gs_createInstance

Fig. 19. Sample results from deployment analysis.

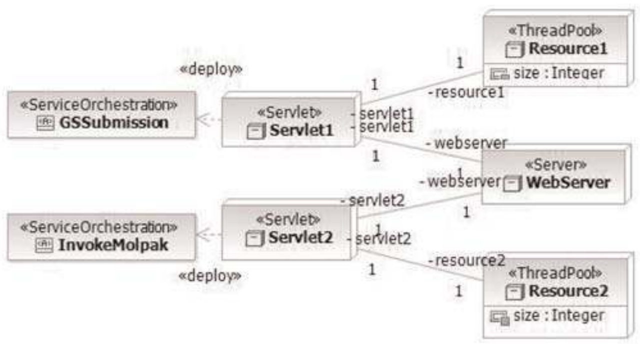


Fig. 20. Refined architecture for Molpak solution.

can be defined. This creates a set of process compositions with a range of 1..N (where N is the number of clients to simulate). Reduction is also repeated as for the design analysis; however, as for the interactions analysis, each port connector model is analyzed as to whether there exists a partner orchestration included in the analysis. If a partner does not exist, the port connector is removed to hide this from deployment analysis. Finally, the grouping abstraction specifically constructs a relabeling of actions in the service orchestrations to either acquire (get) or release (put) a thread resource instance from the resource model. Additionally, a *createInstance* and *terminateInstance* action is added to these lists to represent the new creation of a service orchestration or its termination.

4.5.2 Results

Using the Molpak case study from UCL, we performed the analysis on the deployment model (Fig. 9) and the set of interacting service orchestrations. The resulting violation trace is illustrated in Fig. 19. Note that we show concurrent interactions for two clients in square brackets. The reason for this deadlock is an exhausted resource thread pool allocation, at the point of creating a new instance of the GSSubmission orchestration. The concurrent allocation of thread resources for client requests 1 and 2 is the cause for the shared pool limit to be reached, with each request waiting for a response which can never be received. The deadlock situation has occurred whereby neither of the invocations can be replied to as the GSSubmission orchestration would also require further threads to undertake its activities (invocations of other services). The solution in this case is architectural, since the behavior is acceptable yet the deployment configuration is conflicting with server resource usage. The solution is to split the orchestrations across two individual thread pools, as illustrated in Fig. 20.

5 Implementation and Experience

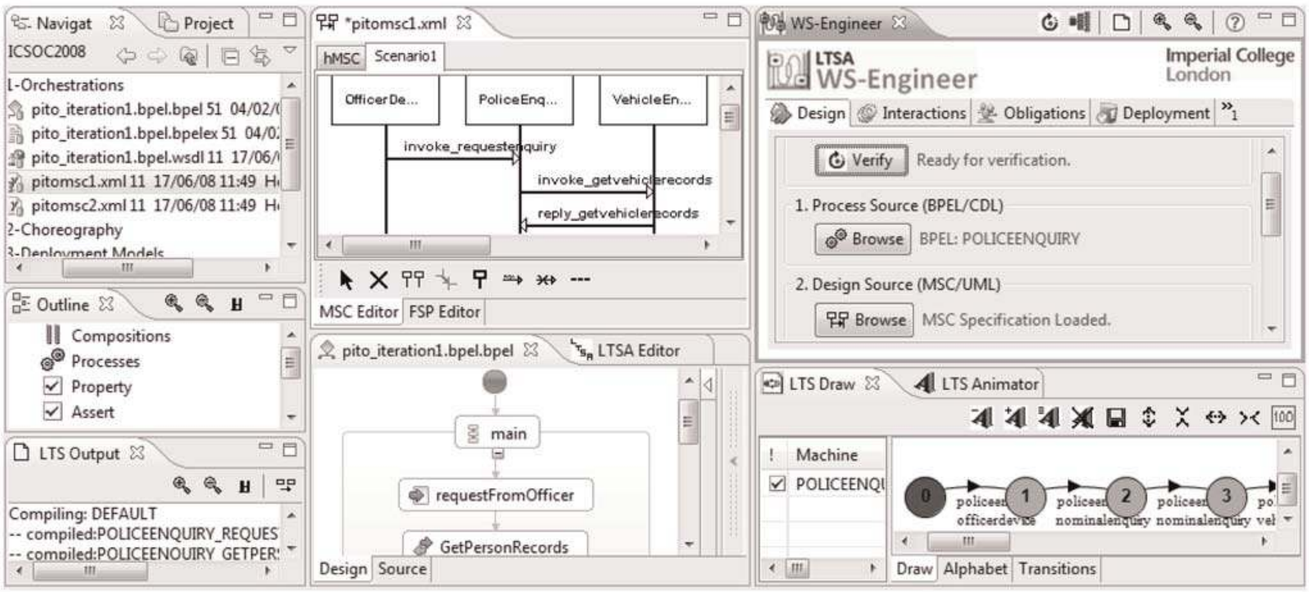
The analysis types discussed in Section 4 have been implemented as an Eclipse IDE plug-in, supporting a mechanical analysis approach to aid service engineers in developing service compositions. The plug-in (known as *WS-Engineer* [16]) is illustrated in Fig. 21, and is built on top of the LTSA [10] originally written by Jeffrey N. Magee of the Department of Computing, Imperial College London. One of the key principles of the WS-Engineer tool is to hide the necessary underlying formalisms from service engineers, allowing them to concentrate on implementing correctly service requirements and not on how to model them. WS-Engineer provides both a visual view for engineers to select the different source artifacts and then simply *Verify* these under the type of analysis selected or compose analysis through an open API for integration into other tools. The WS-Engineer tool is available at http://www.ws-engineer.net.

We have been fortunate to have some challenging business and academic *service composition scenarios* presented to us, covering a wide ranging domain of service applications. We have already shown two examples of our experience, namely that in verifying service orchestrations and choreography for the UK Police IT Organisation (PITO) and for deployment analysis in the Molpak example with the University College London (UCL). More recently, we have been part of a European Union project called Sensoria (http://www.sensoria-ist.eu), which aims to provide formal techniques for the software engineering of services.

Our technical experience has focused on closely aligning the analysis techniques with existing service orchestration and choreography tools. For example, we carried out a series of tests using the WS-BPEL examples provided by International Business Machines (IBM) with their BPWS4J Engine. We have also worked closely with the WS-CDL group of the World-Wide Web Consortium (W3C), carrying out a series of modeling tests from their WS-CDL exit tests examples and tools. This has been verified by several members of the group.

6 Discussion and Related Work

We discuss similar or related work in two areas, namely service composition specification modeling and formal service modeling with verification of service compositions. For service specification modeling, Papazoglou and Yang [17] suggested that service specifications should simply describe all interfaces of a set of operations that are available to the service client for invocation. Broader requirements have been specified in [18], [19], [20] where the authors describe an approach to specifying business processes through a subset of the UML profile (driven from a process class with attributes and methods). The behavior of the interacting process classes is given using an activity graph. While these works show partnered processes working together, it is unclear how multiple scenarios of each process would be specified. In [21], however, the authors provide examples of service behavior in



```
Compiling: DEFAULT
-- compiled:POLICEENQUIRY_REQUES
-- compiled:POLICEENQUIRY_GETPER
```

Fig. 21. WS-Engineer workbench plug-in for eclipse IDE with MSC, WS-BPEL Editors, and LTS Views.

both UML Sequence and Activity diagrams, yet revoke back to building requirements in a pi-calculus algebra to represent the concurrent and alternative paths possible in a composite Web service specification. More elaborate techniques have acted as a bridge between model-driven approaches and those that specify directly in a process algebra. For example in [22], the authors use an extended version of the TROPOS methodology, featuring a modeling framework proposed in [23] to capture business requirements and then generate orchestration (WS-BPEL) source from these requirements. In [24], [25], [26], Petri-net-based models are used to specify the semantics of Web service specifications, their compositions, and the communication between services. A direct mapping is mentioned between service specification and Petri nets yet no examples were given for this. For service design specifications, it appears that these works concentrate on a single scenario with the disadvantage that multiple scenarios mean changing a single specification, rather than an incremental or evolutionary approach to add to design specifications as requirements change.

One of the earlier proposals for formal analysis of composition implementations was given in [27]. In this the author suggests that due to the nature of the software assets (the compositions in this case) being deployed to the Internet, the risk of a bug in such a composition impacts are much greater than that of conventional system deployments. The author of this work has also provided analysis of compositions in terms of those implemented in the Web Service Flow Language (WSFL) [28], which is one of a group of specifications that have been used to create WS-BPEL, and implements a mapping between WSFL and Promela (the language of the SPIN tool) [29]. The work provides a useful reference point on mapping XML schemas (as Web service specifications are typically defined in XML). In [30], [31], Web service specifications are described in LOTOS. The authors extend the common mapping theme between the algebra and WS-BPEL by providing rules for a two-way process. They also confirm, however, that due to the expressive and flexible structure of LOTOS, the mapping from LOTOS to WS-BPEL clearly does not preserve the structure of a process. More recently, coverage of WS-BPEL modeling has increased, and in [32] the authors use the model checker Bogor (which supports different property languages using LTL, CTL, etc.).

To the best of our knowledge, little has been published on combining service specifications, their implementations, and deployment scenarios with formal models and their analysis. We believe that providing analysis for these crosscutting concerns elevates the effectiveness of analysis and provides a much richer workbench for integrating with existing and future service engineering toolsets.

7 Conclusion

We have presented a formal rigorous approach to analyzing service compositions from differing viewpoints and using varied properties to greatly increase the assurance that engineered service compositions are safe and correct. To enable this, we described a series of semantic mappings from service composition specifications and implementations to LTSs and used these transition systems as models to analyze important aspects of service orchestration and choreography. We have defined and implemented several analysis types, including design analysis, orchestration and choreography analysis, and also architectural concerns with service orchestration deployment configurations and limited resource usage. One of the key goals of our work has been to provide a core platform for others to explore other areas of service composition analysis. Our future work will consider how dynamic service composition analysis can be combined with rigorous software engineering principles to provide safe and correct behavior adaptation of systems in a services architecture. Toward this, we have already produced some key work in the self-management and adaptation of service composition using the concept of *Service Modes* [33].

Acknowledgments

This work has been partly supported by the EU FET-IST Global Computing 2 project SENSORIA (IST-3-016004-IP-09), grant PAE-PICT-2272, and by two IBM Eclipse Innovation Awards (2005 and 2006). The authors would also like to thank Professor Wolfgang Emmerich (University College London) and Michael Hu (UK Police IT Organisation) for the example scenarios they provided.

References

[1] A. Alves et al., "Web Service Business Execution Language (WS-BPEL) v2.0," OASIS, OASIS Standard, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, Apr. 2007.

[2] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon, "Web Services Choreography Description Language Version 1.0," W3C, W3C Candidate Recommendation, http://www.w3.org/TR/ws-cdl-10, Nov. 2005.

[3] R. Akkiraju, H. Flaxer, H. Chang, T. Chao, L.J. Zhang, F. Wu, and J.J. Jeng, "A Framework for Facilitating Dynamic e-Business via Web Services," *Proc. Workshop Object-Oriented Web Services at ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01),* 2001.

[4] D. Booth and D.H. Haas, "Web Services Architecture (WS-A)," World Wide Web Consortium (W3C), Working Group Note, http://www.w3.org/TR/ws-arch, Feb. 2004.

[5] I. Pyarali, M. Spivak, R. Cytron, and C.S. Douglas, "Evaluating and Optimizing Thread Pool Strategies for Real-Time CORBA," *Proc. ACM SIGPLAN Workshop Languages, Compilers and Tools for Embedded Systems,* 2001.

[6] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying Distributed Software Architectures," *Proc. Fifth European Software Eng. Conf. (ESEC '95),* 1995.

[7] OMG, "Unified Modelling Language (UML) 2.1.1," Object Management Group, Specification, www.uml.org, Feb. 2007.

[8] R. Milner, *Communication and Concurrency.* Prentice-Hall, 1989.

[9] J. Magee, J. Kramer, and D. Giannakopoulou, "Analysing the Behaviour of Distributed Software Architectures: A Case Study," *Proc. Fifth IEEE Workshop Future Trends of Distributed Computing Systems,* 1997.

[10] J. Magee and J. Kramer, *Concurrency—State Models and Java Programs,* second ed. John Wiley, 2006.

[11] J. Magee, "FSP Language Specification," http://www.doc.ic.ac.uk/ltsa/fsp, 2009.

[12] ITU-T-Z20, "Formal Description Techniques (FDT) Message Sequence Chart (MSC) (Z20)," Int'l Telecomm. Union, Telecomm. Standardisation Sector, ITU-T Recommendation, http://www.itu.int/ITU-T/2005-2008/com17/languages/Z120.pdf, Apr. 2004.

[13] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of Behavioral Models from Scenarios," *IEEE Trans. Software Eng.*, vol. 29, no. 2, pp. 99-115, Feb. 2003.

[14] H. Foster, S. Uchitel, J. Magee, J. Kramer, and M. Hu, "Using a Rigorous Approach for Engineering Web Service Compositions: A Case Study," *Proc. Services Computing Conf. (SCC '05),* 2005.

[15] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S.L. Price, "Grid Service Orchestration Using the Business Process Execution Language (BPEL)," *J. Grid Computing,* vol. 3, nos. 3/4, pp. 283-304, http://dx.doi.org/10.1007/s10723-005-9015-3, 2005.

[16] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "WS-Engineer: A Tool for Model-Based Verification of Web Service Compositions and Choreography," *Proc. Int'l Conf. Software Eng. (ICSE '06),* 2006.

[17] M. Papazoglou and J. Yang, "Design Methodology for Web Services and Business Processes, Technologies for E-Services," *Lecture Notes in Computer Science,* Springer-Verlag, 2002.

[18] T. Gardner, "UML Modelling of Automated Business Process with Mapping to BPEL4WS," *Proc. First European Workshop Object Orientation and Web Services (EOOWS '03),* 2003.

[19] S. Iyengar, "Business Process Integration Using UML and BPEL4WS," *Proc. XML Conf. and Exposition (XML '03),* 2003.

[20] K. Mantell, "From UML to BPEL," technical report, IBM DeveloperWorks, 2003.

[21] S. Woodman and E.D. Palmer, "Notations for the Specification and Verification of Composite Web Services," *Proc. Eighth IEEE Int'l Enterprise Distributed Object Computing (EDOC),* 2004.

[22] M. Pistore and A.M. Roveri, "Requirements-Driven Verification of Web Services," *Proc. Int'l Workshop Web Services and Formal Methods (WS-FM '04),* 2004.

[23] E. Yu, "Towards Modeling and Reasoning Support for Early Requirements Engineering," *Proc. Third Int'l Symp. Requirements Eng. (RE '97),* 1997.

[24] R. Hamadi and B. Benatallah, "A Petri Net-Based Model for Web Services Composition," *Proc. Third IEEE Int'l Conf. Web Services (ICWS),* 2004.

[25] X. Yi and K.J. Kochut, "Towards Efficient Integration of Complex Web Services Using a Unified Model for Protocol and Process," *Proc. Fifth Int'l Conf. Internet Computing (IC '04),* 2004.

[26] X. Yi and K. Kochut, "Process Composition of Web Services with Complex Conversation Protocols: A Colored Petri Nets Based Approach," *Proc. Design, Analysis, and Simulation of Distributed Systems Symp. (DASD '04),* 2004.

[27] S. Nakajima, "Model-Checking Verification for Reliable Web Service," *Proc. Workshop Object-Oriented Web Services at ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA),* 2002.

[28] F. Leymann, "Web Services Flow Language Specification (WSFL 1.0)," technical report, IBM, 2001.

[29] S. Nakajima, "On Verifying Web Service Flows," *Proc. Int'l Symp. Applications and the Internet (SAINT '02),* 2002.

[30] A. Ferrara, "Web Services: A Process Algebra Approach," *Proc. Second Int'l Conf. Service Oriented Computing (ICSOC '04),* 2004.

[31] G. Salaun and A. Ferrara, "Negotiation among Web Services Using LOTOS/CADP," *Proc. European Conf. Web Services (ECWS '04),* 2004.

[32] D. Bianculli, C. Ghezzi, and P. Spoletini, "A Model Checking Approach to Verify BPEL4WS Workflows," *Proc. Int'l Conf. Service-Oriented Computing and Applications (SOCA '07),* 2007.

[33] H. Foster, "Architecture and Behaviour Analysis for Engineering Service Modes," *Proc. Second Workshop Principles of Eng. Service Oriented Systems (PESOS '09),* May 2009.

Howard Foster received the PhD degree in 2006 from Imperial College London in the area of rigorous software engineering for service compositions and has more than 12 years industrial experience as an IT consultant for leading business and IT professional service organizations. He is a research fellow with the Department of Computing, Imperial College London. His research focuses on software services, compositions, choreography, and model-based verification and validation techniques. He is also actively working in the area of software self-management and behavioral synthesis of software components. He is a regular program and workshop committee member for leading software engineering conferences and also for software service focused conferences and workshops. He is a member of the IEEE.

Sebastian Uchitel holds a professorship at the Department of Computing, FCEN, University of Buenos Aires, and a readership at Imperial College London. His research interests are in behavior modeling and analysis of requirements and design for complex software-intensive systems. His research focuses on partial behavior modeling, including scenario-based specifications, behavior model synthesis, and modal transition systems. His research also includes goal-oriented requirements engineering, reliability, software architectures, and service-oriented architectures. He was an associate editor of the *IEEE Transactions on Software Engineering* and is currently an associate editor of the *Requirements Engineering Journal*. He was a program cochair of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006) held in Tokyo and is the program cochair of the 32nd IEEE/ACM International Conference on Software Engineering (ICSE 2010). He has recently been distinguished with the Philip Leverhulme Prize. He is a member of the IEEE Computer Society.

Jeff Magee is a professor of computing and is currently both the head of the Department of Computing and the deputy principal of the Faculty of Engineering at Imperial College. His research is primarily concerned with the software engineering of complex systems, currently focusing on the software architecture of self-adaptive systems. He is the author of more than 150 journal and conference publications and has coauthored a book on concurrent programming entitled *Concurrency—State Models and Java Programs*. He was a coeditor of the *IEE Proceedings on Software Engineering* and an associate editor of *ACM Transactions on Software Engineering and Methodology*. He was a program cochair of the 24th International Conference on Software Engineering and chair of the ICSE Steering Committee from 2002 to 2004. He was a member-at-large of the ACM SIGSOFT committee from 2002 to 2005. He was the corecipient of the 2005 ACM SIGSOFT Outstanding Research Award for his work in distributed software engineering. He is a chartered engineer and fellow of the BCS. He is also a member of the IEEE.

Jeff Kramer is a professor of computing and senior dean of Imperial College London. His research interests include rigorous techniques for requirements engineering; software specification, design, and analysis; and software architectures, particularly as applied to distributed and adaptive software systems. He was the editor-in-chief of the *IEEE Transactions on Software Engineering* (2006-2009) and the corecipient of the 2005 ACM SIGSOFT Outstanding Research Award for his research work in distributed software engineering. He was a program cochair of the 21st IEEE/ACM International Conference on Software Engineering (ICSE 1999) and is the general cochair of the 32nd IEEE/ACM International Conference on Software Engineering (ICSE 2010). He is a coauthor of a recent book on concurrency, coauthor of a previous book on distributed systems and computer networks, and author of more than 200 journal and conference publications. He is a chartered engineer, a fellow of the Royal Academy of Engineering, a fellow of the IET, a fellow of the BCS, a fellow of the ACM, and a member of the IEEE Computer Society.