Contracts and Middleware for Safe SOA Applications

Beata Sarna-Starosta, R.E.K. Stirewalt and Laura K. Dillon
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
{bss,stire,ldillon}@cse.msu.edu

Abstract

This position paper concerns formal methods for developing safe service-oriented architectures (SOAs) with support for resource management. We seek an approach to building such SOAs based on the specification of service requirements as declarative contracts, and the enforcement of these contracts at the level of the middleware. Using hierarchical containers that provide the necessary middleware services, we expect to guarantee certain safety properties by construction and raise the level of model abstraction for verifying other necessary properties.

1. Introduction

The service-oriented architecture (SOA) paradigm facilitates component reuse, interoperability, scalability, and flexibility in assembling mission-specific applications. However, SOA-based systems cannot yet support critical tasks because the underlying technologies fail to provide sufficient guarantees of safety or quality of service (QoS). To do so, SOA developers need models, tools, and middleware infrastructures that enable the design, implementation, and verification of SOA-based applications.

Safety problems arise when services are composed. Instances, called *processes*, of a composite service application may execute for long periods of time and must be robust in the face of remote-service failures. Long-running processes could degrade server performance, possibly to the point of causing the server to deny service to its customers. To address these issues, the developer must typically augment an otherwise simple interaction protocol with logic that, e.g., monitors the time spent waiting for a response and that triggers some form of remediation when too much time has elapsed. Flaws in these protocols can lead to safety violations and may be exploited by attackers.

To develop safe SOAs requires the ability to design and package services as components, to specify compositions, and to reason about global safety properties and QoS. Herein lies the dilemma: Safety and QoS are complex non-functional concerns, whose effects are global and thus difficult to verify. In prior work, we developed an approach, called Szumo [2, 16], to building systems with strong guarantees relating to synchronization of concurrent threads—a similarly global and cross-cutting non-functional concern. Many of the safety issues that confront the developer of a composite service bear a striking resemblance to those addressed by Szumo. This paper describes these safety issues by way of a concrete example and argues for the use of a Szumo-inspired solution to address them.

In Szumo, design and implementation of concurrency is divided between the developer and middleware. In lieu of writing low-level synchronization code, developers declare *synchronization contracts*, which are explicit and which are dynamically negotiated at run time by a middleware. In addition to raising the level of abstraction in development, the use of such *contract-enabled middleware* relegates a class of assurance obligations to the middleware, thereby enabling a larger class of obligations to be discharged using abstract (and thus more compact) models. The use of contracts and middleware in this capacity is an example of *design for verification* (D4V) [10, 6, 11], a general approach to dealing with complexity in which the developer concedes some degree of design freedom in order to enable efficient verification of system properties with strong guarantees. In the sequel, we postulate that a similar approach should enable the development of safe SOAs. A key insight is the use of *hierarchical containers*, which build on the theory of *connector wrappers* [15] and our prior work in implementing connector wrappers [14]. The remainder of the paper fleshes out these ideas in more detail.

2. Problem Description

SOAs must ensure safety while guaranteeing a level of service necessary to support clients, be they human users or other (composite) services. By *safety*, we mean that integrity should not be compromised as a side-effect of authorized use; whereas by *level of service*, we mean to include issues of usability, availability, and QoS. Services communicate with each other by sending and receiving asynchronous messages, and service interactions often involve complex protocols of message exchange. The protocols describe communication among multiple services, possibly running on different servers, and may involve operations on remote servers' resources. Furthermore, it is possible that several constituent services running concurrently perform operations on different resources as parts of a single transaction. To properly implement a distributed transaction across multiple constituent services requires elaborate protocols of message exchange among the constituents. If the constituent services are developed in an ad hoc manner, subtle implementation flaws could result in safety problems and degraded levels of service.

2.1. An example SOA

Consider a very simple web-based SOA application that allows customers to make on-line arrangements for flights with one specific airline (say, Delta) and accommodations with one specific hotel network (say, Hilton). This *TripManager* SOA handles customers' requests by contacting two component web services, *DeltaService* and *HiltonService*, responsible for finding, respectively, flights and accommodations that satisfy the customers' travel constraints. Given a request to arrange only a flight or an accommodation, *TripManager* forwards the request, along with the customer's constraints, to the respective vendor service. The service processes the request, and returns to *TripManager* with a set of results including the flight or lodging options meeting the customer's requirements and, possibly, alternative options "close" to meeting these requirements. The *TripManager* forwards the results to the customer who then may choose one.

If no flight/accommodation is selected, the transaction is aborted, either by an explicit termination of *TripManager* by the customer, or by *TripManager*'s time-out in case the customer abandons the process. If the customer does select a flight/accommodation, *TripManager* forwards the relevant information back to the appropriate vendor service, which finalizes the transaction. The transaction finalization proceeds according to a two-phase protocol. In the first phase, *TripManager* notifies *DeltaService* (*HiltonService*) that it intends to commit to the flight (hotel room) selected by the customer, and waits for an acknowledgment from the vendor service, confirming that the requested resource is available. If *DeltaService* (*HiltonService*) confirms the request, the resource in question is *provisionally allocated* to the requesting *TripManager* service. In the second phase, *TripManager* sends to the vendor service a notification of commitment. The vendor service then allocates the resource to the customer (charging their credit card accordingly), and returns to *TripManager* with a final acknowledgment.

Whether or not the customer decides to purchase the requested flight or accommodation, processing the request involves asynchronous message exchange between the involved services. Furthermore, *TripManager* must follow interaction protocols established with *DeltaService* and *HiltonService*, so that the flight or lodging option initially reported as available is indeed available after being selected by the customer.

This scenario is additionally complicated when the request involves arrangements for both flight and accommodation, as *TripManager* must then ensure consistency between the results returned by *DeltaService* and *HiltonService*. For example, suppose a customer selects a combination of flight and accommodation that suits her needs. The *TripManager* cannot just forward the selected flight information to *DeltaService* and the selected accommodation information to *HiltonService* with instructions to finalize, because one might succeed and the other might fail (for instance, a vendor's server might crash). The difficulty lies in ensuring that these transactions are processed atomically, that is, either neither transaction or both transactions finalize while maintaining an acceptable level of service. The customer should not wait too long to learn if the trip arrangements are successfully finalized, but if *TripManager* is too hasty in aborting an attempt to finalize a set of options, it risks not finding good solutions.

2.2. Inter-process coordination issues

Consider two instances of the *TripManager* SOA, *TripManager*$_1$ and *TripManager*$_2$, each independently processing a request to arrange travel and accommodation for a different customer. Each instance must be careful to book the hotel and the corresponding flight as an atomic transaction. To achieve this, the *TripManager* will commit a transaction in two phases, first sending two notifications of intent—for the selected flight to *DeltaService*, and for the selected hotel room to *HiltonService*, then waiting for acknowledgments from both before starting the second phase, which finalizes the commit, causing the constituent services to update their respective databases.

When the two instances of *TripManager* perform these sequences of operations concurrently, the following situation may occur. Assume that both *TripManager*$_1$ and *TripManager*$_2$ begin the first phase of their respective commits by sending messages to both *DeltaService* and *HiltonService*. Suppose also that the message from *TripManager*$_1$ arrives at *DeltaService*$_1$ and is acknowledged before the corresponding message from *TripManager*$_2$ arrives at *DeltaService*$_2$, and that the message from *TripManager*$_2$

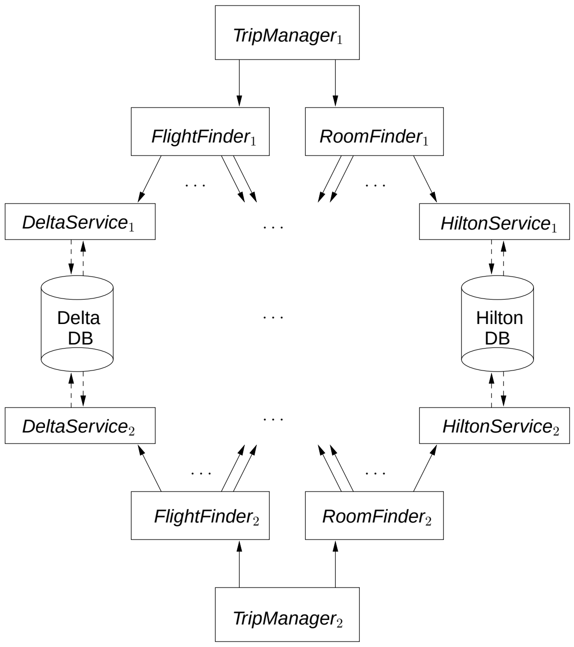

Figure 1. Concurrent operation of two *TripManager* processes

arrives at *HiltonService*$_2$ and is acknowledged before the corresponding message from *TripManager*$_1$ arrives at *HiltonService*$_1$. At this point neither of the two *TripManager* processes may finalize and must re-attempt, which could cause the same scenario to repeat until the *TripManager* processes time out.

The problems of service coordination are exacerbated in a more realistic scenario, in which the *TripManager* service can make travel and accommodation arrangements with multiple vendors. To facilitate this capability, *TripManager* might coordinate two subordinate services, *FlightFinder* and *RoomFinder*, which serve as intermediaries, each communicating with the individual hotel and airlines services. This situation is illustrated in **Figure 1**. To properly handle this case, the commit initiated by *TripManager* will need to compose hierarchically.

2.3. Intra-process coordination issues

In addition to protocol flaws among multiple interacting services, a single service could exhibit safety problems if it is multi-threaded, and the concurrent threads operate over shared data without proper synchronization. In the context of web services, concurrency is realized by means of a *flow* construct provided by WS-BPEL [1]. Each of the concurrently operating flows may read and modify shared variables. For instance, to improve its performance, an SOA may deploy separate flows—each responsible for interacting with a different component service—and embed logic to compensate for those components that do not respond in a reasonable time or do not produce acceptable results. With distribution and the need to synchronize come the kinds of complexity that lead to safety problems.

For example, given a customer's request to find a flight to a specific destination, on a specific date and within a specific price range, *TripManager* forwards this request to *FlightFinder*. The *FlightFinder* service may then delegate a primary flow to look for the flights to that destination exactly meeting the customer's constraints, and auxiliary flows to look for the options where only the time or price constraint is satisfied in case both constraints cannot be met. Each flow writes its results to a separate variable, and after all flows complete execution (or their execution exceeds allowed time and is terminated), the best results are copied to a shared buffer storing the service's output. The flows must be synchronized to prevent races on the shared buffer and to ensure that the best solution over all solutions computed by the flows is among those returned by the service.

In addition to the issues of flow-level parallelism, there is also the issue of when to collect the resources allocated to a process, especially if the threads allocated to this process are either mutually blocked or waiting on a message from a remote service that has crashed. Continuing with our example, after *FlightFinder* receives all results supplied by its provider services within an acceptable time frame, these results should be returned to *TripManager*, and all the resources allocated to *FlightFinder* (including the resources for the flows which did not complete their execution within that time frame) should be collected.

3. Research Directions

We see parallels between many of the safety issues that confront the developer of composite services and those addressed by Szumo, our approach to designing and analyzing systems for properties involving the synchronization of concurrent threads [2, 16]. In Szumo, the tasks of system design and implementation are divided between the developer and the middleware. The developer enhances the specification of the system objects with declarative contracts which describe the objects' requirements for exclusive resource access. At run time, the middleware first composes and then negotiates these *local* contracts to ensure that the exclusive access requirements are satisfied *globally*, i.e., at the system level.

We believe this idea of contract-enabled middleware could be applied in the context of SOAs, with *containers* playing the role of the middleware (**Section 3.1**). Our Szumo middleware could be promoted into a *flow container*, with which to synchronize the interaction of concurrent flows within a given service (**Section 3.2**). In addition, we believe the essential ideas could be extended to automate the coordination of services that participate in hierarchical, atomic transactions. In this case, the developer of a composite service would specify a contract that declares how the service will need to atomically commit operations with its suppliers. These contracts would then be negotiated by means of *inter-process containers*, thereby alleviating the service developer from having to code up this tedious and error-prone logic (**Section 3.3**).

3.1. Containers

A container is an entity that encapsulates a software component from its environment—all interaction with the component from the outside is routed through the container, which may add new capabilities and enforce safety constraints by intercepting and issuing messages to and from the component. We believe that contract-aware containers can automate the implementation of complex service interaction protocols and thereby dramatically simplify the addition of reliability enhancements, such as failsafes, bounded retry, and failover, by developers.

We envision two types of containers—flow and inter-process containers—that could help ameliorate the safety and level of service issues in the context of SOAs. **Figure 2** shows an example of a container-enhanced SOA-based system. Here, S_1 and S_2 represent the servers that are running web service processes. P_1, P_2, and P_3 are instances of three different web services executing on S_1, whereas P_4 and P'_4 are two instances of the same service executing on S_2. Each process has one or more flows, represented as white squares. The arrows connect the constituents of two

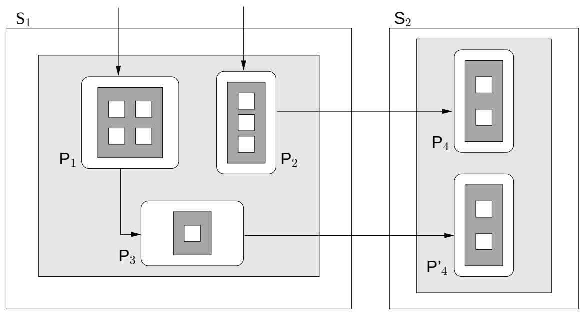

Figure 2. Container-enhanced SOA

different composite processes—one comprising P_1, P_3 and P'_4, and the other comprising P_2 and P_4—running across the two servers. The flow and inter-process containers are depicted as darkly- and lightly-shaded boxes drawn within and around the component processes, respectively.

3.2. Flow container

To address the problems of concurrency within a single web service process, a *flow container* would encompass the flows of an individual process and govern the execution of these flows based on the satisfaction of programmer-supplied synchronization contracts. The container would synchronize concurrent process flows attempting to access shared resources, and automate the negotiation of these flows for exclusive resource access. Flow containers would provide flows with certain safety guarantees, including freedom from simple races among accesses to shared variables by multiple concurrent flows.

Figure 2 depicts the flow containers as boxes with dark shading, surrounding the flows within each process. The WS-BPEL specification provides for a notion of containers to hold shared resources [1]. There is also related work on process containers for web services, but the synchronization aspects of their behavior are left open [7].

3.3. Inter-process container

An *inter-process container* could allow multiple concurrent processes to safely commit atomic transactions involving multiple resources.We envision such a container encompassing all processes running on a given server and governing their interaction with other processes by negotiating contracts between a client and its suppliers. **Figure 2** depicts inter-process containers as boxes with light shading surrounding the processes within a server.

In our running example, to ensure atomic processing of travel and accommodation requests, the *TripManager* service might declare a contract that defines conditions under which it needs to perform atomic commits involving the airline and hotel services. Following Szumo syntax, such a contract might be expressed in the form:

$$\mathrm{finalizing} \Longrightarrow \mathit{FlightFinder} \wedge \mathit{RoomFinder}$$

In English, this states that when the *TripManager* service is in a $\mathrm{finalizing}$ state (i.e., when $\mathrm{finalizing}$ is true), it needs to commit transactions involving the processes bound to *FlightFinder* and *RoomFinder*. Consequently, when coding up the *TripManager* service, the developer would write code to signal when a transaction should finalize rather than write the communication code needed to implement *TripManager*'s role in the commit protocol. At run time, the container would enforce that execution of code that signals entry into a $\mathrm{finalizing}$ state succeeds if the container is able to successfully execute the two-phase commit protocol with the constituent services. Otherwise, the container could either block execution of the statement until the time when it is able to successfully execute the commit or fail. We envision a number of different policies being applicable here, including the use of bounded retry and failover.

If contracts, such as these, were explicitly declared and available at run time, an inter-process container could use them to automate the customization of a commit protocol with the subordinate services, thereby alleviating the developer of *TripManager* from this design burden. Moreover, it is possible that the contracts could be used to inform protocols that attempt to avoid the repeated service denial (due to failure to commit) presented earlier. This example is reminiscent of the behavior of two threads negotiating for exclusive access to the same sets of resources while attempting to avoid or recover from deadlock. The strategies used in these cases require protocols that cannot be implemented by a service in isolation.

3.4. Design for verification

In Szumo, the negotiation of contracts guarantees freedom from a large class of typical concurrency errors, including simple data races and a large class of deadlocks and starvation. These guarantees eliminate the need to analyze systems with regard to such errors, and enable construction of abstract models that facilitate reasoning about other properties. This general approach—conceding some degree of freedom during design in order to enable verification—has been called design for verification (D4V) [10, 6].

The vision outlined in this position paper suggests an approach to D4V for SOAs. Containers at multiple levels would ensure certain guarantees regarding concurrency, information flow and performance. These guarantees enable assumptions that systems assembled from services that execute within the containers will (or will not) exhibit certain behaviors, eliminating the need to explicitly verify related properties. We believe these guarantees will also permit verification of application-specific properties of SOAs based on abstract (and thus compact) behavioral models, which facilitate simpler and more efficient analyses. In cases where verification is not feasible, contracts provide information that may be used to guide and automate testing of the SOAs and of the individual services.

4. Related Work

The ideas discussed in this paper are most closely related to two research areas: modeling transactions for composite services and using contracts to express service requirements.

Transaction design has been studied extensively in the context of database systems [3, 5]. The long-running, asynchronous nature of SOA application processes exacerbates problems of multidatabase transaction management. Thus, the traditional notion of transactions has been augmented to address problems of SOAs [18, 17, 8, 12, 4]. Much of this work focuses on seamless multidatabase operations in scenarios where the collection of databases being accessed changes dynamically during execution. Proposed solutions generally require a developer to define services using new, special-purpose formalisms, specify transactional requirements (e.g., concurrency or QoS) at a low level, or provide global scheduling schemes for service components. In contrast, we propose that the developer should express local transactional requirements at a high level, in a manner that integrates well with existing service specification languages, and that the data processing algorithms needed to enforce the contracts be encapsulated in middleware. The example in this paper illustrates only transactional requirements, but we believe this contract-based middleware approach can be extended to also support other important aspects of service execution, such as security and some QoS requirements.

Contracts have been used to represent service-level agreements (SLAs)—mutual QoS responsibilities among collaborating services. Existing SLA specification languages (e.g., WSLA [9], SLAng [13]) enable developers to formalize these agreements, which may then be enforced automatically by appropriately extending network routers, database management systems, middleware, and/or web servers. SLAs do not typically express transactional requirements of the type illustrated in our example. Ideas from SLA specification languages, however, may prove useful in suggesting how to extend contracts to express various QoS requirements.

5. Summary and Challenges

This position paper calls for research to ensure safety and QoS in SOA applications. The SOA paradigm plays a significant role in today's software engineering, with service compositions implementing more and more critical tasks. Thus, the reliability of SOA-based systems is increasingly important. At the same time, safety and QoS are complex, non-functional concerns with global effects on system behavior. Furthermore, the cross-cutting nature of these concerns makes them difficult to define and verify.

We advocate for a contract-based D4V approach to building SOAs with strong guarantees of safety and QoS. In this approach, developers express safety requirements of an individual service as declarative contracts enhancing the service's specification. When multiple services are assembled into a composite SOA system, their contracts are composed and enforced (negotiated) by the middleware. Negotiation of contracts of the constituent services provides the composite system with certain safety guarantees by construction, thus enabling the analysis of other properties at a higher level of abstraction. We envision the use of process containers to implement this concept of contract-enabled middleware in the SOA environment. Our paper focuses on web-based SOA applications; however, the approach may be also applied to other applications that make use of SOAs.

Realization of our vision will require addressing many open challenges, including:

- What extensions to our contract language will be necessary to adapt it to the SOA setting and to express a wider range of concerns, such as QoS or security?
- Will the overhead incurred by composing and negotiating contracts in middleware be acceptable?
- Will the introduction of analysis abstractions for containers permit derivation of compact system models and support systematic testing?

References

[1] Web Services Business Process Execution Language (WS-BPEL). `http://docs.oasis-open.org/-wsbpel/2.0/wsbpel-specification-draft.html`.

[2] R. Behrends and R. E. K. Stirewalt. The Universe Model: An approach for improving the modularity and reliability of concurrent programs. In *Proc. of FSE'2000*, 2000.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[4] S. Bhiri, C. Godart, and O. Perrin. Transactional patterns for reliable web services compositions. In *Proc. of the 6th Intl. Conf. on Web Engineering (ICWE '06)*, pages 137–144, 2006.

[5] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181–240, 1992.

[6] T. Bultan and A. Betin-Can. Scalable software model checking using design for verification. In *Proc. of the IFIP Working Conference on Verified Software: Theories, Tools, Experiments*, 2005.

[7] A. Charfi and M. Mezini. An aspect-based process container for BPEL. In *Proceedings of the 1st workshop on Aspect oriented middleware development (AOMD'05)*, 2005.

[8] K. Haller, H. Schuldt, and C. Türker. Decentralized coordination of transactional processes in peer-to-peer environments. In *Proc. of the 14th ACM Intl. Conf. on Information and Knowledge Management (CIKM'05)*, pages 28–35, 2005.

[9] IBM. Web Service Level Agreement (WSLA) Language Specification, 2003.

[10] P. Mehlitz and J. Penix. Design for verification using design patterns to build reliable systems. In *Proc. of the* 6^{th} *ICSE Workshop on Component-Based Software Engineering*, 2003.

[11] B. Sarna-Starosta, R. E. K. Stirewalt, and L. K. Dillon. A model-based design-for-verification approach to checking for deadlock in multi-threaded applications. In *Proc. of* 18^{th} *Intl. Conf. on Softw. Eng. and Knowledge Eng.*, 2006.

[12] B. Schmit and S. Dustdar. Model-driven development of web service transactions. *Intl. Journal of Enterprise Modeling and Information Systems Architecture*, 1(1), 2005.

[13] J. Skene, D. Lamanna, and W. Emmerich. Precise service level agreements. In *Proc. of the 26th Intl. Conf. on Software Engineering (ICSE'04)*, 2004.

[14] J. H. Sowell and R. E. K. Stirewalt. A feature-oriented alternative to implementing reliability connector wrappers. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems III*. Springer, 2005.

[15] B. Spitznagel and D. Garlan. A Compositional Formalization of Connector Wrappers. In *Proc. of the 2003 International Conference on Software Engineering*, Portland, Oregon, USA, May 2003.

[16] R. E. K. Stirewalt, R. Behrends, and L. K. Dillon. Safe and reliable use of concurrency in multi-threaded shared memory sytems. In *Proc. of the* 29^{th} *Annual IEEE/NASA Software Engineering Workshop*, 2005.

[17] C. Wichert, A. Fent, and B. Freitag. The compositionality and transactional execution of electronic services. Technical Report MIP-0010, Universitt Passau (FMI), 2000. `http://daisy.fmi.unipassau.de/papers/`.

[18] J. J. Yang and G. E. Kaiser. Jpernlite: an extensible transaction server for the world wide web. In *Proc. of the 9th ACM Conf. on Hypertext and hypermedia (HYPERTEXT '98)*, pages 256–266, 1998.