Leveraging Synchronization Contracts for Compositional Reasoning in Design of Services

Laura K. Dillon\textsuperscript{a}, Yi Huang\textsuperscript{a}, and R.E.K. Stirewalt\textsuperscript{b}

\textsuperscript{a}Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824, USA;
\textsuperscript{b}LogicBlox, Inc., Atlanta, GA 30309, USA

EXTENDED ABSTRACT

1. INTRODUCTION

The servlets platform is the preferred Java technology for implementing services on the Web.\textsuperscript{1} Based on a request/response programming model, the servlet programming model is simple and intuitive, yet very general, permitting flexible composition of services. Servlets are deployed to a container, which is meant to free a service developer to concentrate on design and implementation of a service’s “business logic.” Toward this end, the container automates details of messaging protocols and of resource and life-cycle management. In particular, the container manages threads executing in the servlets it hosts in order that the servlets can respond to requests from multiple, autonomous clients in a timely manner. The platform thus simplifies programming and promotes portability.

Details of synchronizing concurrent service executions, however, have proven difficult to relegate to a container.\textsuperscript{2,3} A web service typically manages dynamically changing numbers of service executions on behalf of autonomous clients. For example, a multi-way chat service manages a distinct service execution for each client that signals to start (or join an existing) chat. The lifetimes of different service executions may overlap, and the service must respond in a timely manner to any legal message intended either for an extant service execution or to initiate a new one. In processing a message, a thread advances the service logic of the messaged service execution and records sufficient state information that other threads can pick up where it left off when processing subsequent messages for this same service execution. In processing a message, the thread may access shared libraries and shared software components, and may even access and modify the execution states of other service executions. Thus, a servlet typically mixes code implementing a service’s business logic with code to acquire and release locks on shared resources in order to prevent data races and yet still allow concurrent processing of non-interfering messages. This synchronization code is prone to subtle synchronization errors, which are notoriously difficult to diagnose and correct. It also tends to obscure the business logic.

In prior work, we developed an approach to address these problems based on synchronization contracts, which declare the resources needed for processing messages, and we showed that the approach simplifies programming of services.\textsuperscript{3,4} More recently, we undertook a feasibility study, which showed that our contract-based approach complements best development practices in the IP telecommunication domain and does not incur significant runtime overhead for a realistic service.\textsuperscript{5} However, this study also produced insights into how we might refine the approach to further improve transparency between a state-machine design and an implementation of that design as a servlet. We believe the refined approach will also permit us to leverage synchronization contracts for compositional reasoning about designs of services expressed as state machines and their implementations.

This extended abstract outlines our ongoing research on these ideas. We first provide some general background on design and implementation of services (Section 2). We then describe the contract-based approach for developing services (Section 3), which the next release of our reference synchronization framework for servlets will support. Finally, we outline plans for supporting compositional reasoning about service designs and their implementations (Section 4) and provide summary remarks (Section 5).

Author E-mail: L.K.D., Y.H.: \{ ldillon, huangyi7 \}@cse.msu.edu; R.E.K.S.: kurt.stirewalt@logicblox.com
2. BACKGROUND

Our contract-based approach complements best practice in design and development of services. Specifically, it targets common servlets platforms, like Apache Tomcat, Oracle Weblogic, IBM WebSphere, or SUN GlassFish (Section 2.1), and it employs a state machine design of a service’s business logic as the primary design artifact (Section 2.2).

2.1 Common servlets platform

Modern servlets platforms require an application’s state is separated from the application’s logic, with the former encapsulated in sessions and the latter encapsulated in one or more servlets. Briefly, an application session represents the states of all ongoing executions of a service via protocol sessions—e.g., SIP sessions and HTTP sessions. A protocol session objectifies a service execution, which in turn controls a media connection between two endpoints, one of which is the service. Because sessions are separate from application logic, a single servlet can host multiple threads. The servlets container provides an API for navigating and querying a message and its associated sessions.

Because endpoints are autonomous, the container cannot predict when or in what order messages will arrive. Thus, to ensure responsiveness, it executes in a dedicated thread, listening for incoming messages on the network and for outgoing messages generated by the threads it manages. Upon receiving a message bound for a service that it hosts, the container determines the servlet to route the message to and retrieves the message’s associated sessions or, if the message is not associated with pre-existing protocol and application sessions, it creates the needed sessions. It then dispatches a thread to process the message, passing that thread the message and the associated servlet and sessions. For brevity, we refer to the information passed to the thread as the message-processing context. After dispatching the thread, the container returns to listening for other messages.

To process a message, a thread invokes a special service method in the servlet-container API. This method, in turn, invokes various message-specific methods, collectively referred to as doXXX methods. For instance, it invokes doInvite to process an INVITE message. The application programmer overrides the doXXX methods to effect the business logic for a service.

2.2 Current design practice

A best practice in development of IPT services is to represent a service’s business logic as a state machine and program a servlet to emulate the transitions of this machine. We refer to this state machine as a business machine. By convention, each transition in a business machine represents the effects of processing a message. A state indicates how far the service execution has progressed in processing previous messages, information needed to correctly advance the business logic in processing the subsequent message.

*it also invokes the doRequest method, as INVITE is a type of SIP request message
We illustrate the key ideas in Figure 1 for a dial-up dating service, which a subscriber calls to be matched up with another subscriber. The business machine (on the left) represents the logic for one service execution. We depict states as rounded rectangles and transitions as arrows; the arrow style distinguishes transitions that correspond to a message processed by this service execution (solid arrow) from those that correspond to a message processed by some other service execution (dashed arrow). We label transitions with a name and with information about the message processed and any other messages created and sent in processing this first message. The message processed, listed first, is denoted Role?msg, where msg indicates the type of the message and role indicates the role played by the message’s “other” endpoint (i.e., the endpoint connecting with the dating service via the message’s protocol session). A message that is created and sent is denoted ss!msg, where msg indicates the type of the message and ss indicates the intended recipient—the “other” endpoint of the session that ss refers to. Because servlets are stateless, a thread finds all information it uses in processing a message by navigating the APIs of the message-processing context.† The legend (right side of Figure 1) describes the roles (Role Values) of the remote endpoints, other information needed in processing messages for this example (Session Attributes, API macros), and the message types (Message Types). In general, the transitions on a business machine also need to specify updates to session data and other resources. We omit these updates in the figure for conciseness.

Briefly, when a subscriber, say Mary, calls the dating service, her phone sends an initial INVITE message to the servlet container that hosts the dating service. The thread dispatched to process this message initializes an execution of the dating service on Mary’s behalf, objectifying the new service execution as a SIP session, which we refer to as Mary’s session. It also creates a new SIP session to use in creating and sending an INVITE message to a media server, which supports data collection by voice and touch tones. This processing effects transition T1. Mary’s service execution then arranges for Mary’s phone to exchange information with the media server (effecting T2 and T3). While Mary’s phone and the media server are connected, Mary’s service execution is in state TalkingWithMediaServer. When Mary’s service execution attempts to connect Mary’s phone and the peer’s phone (effecting T9, the dashed transition of the peer’s service execution, and T11); otherwise, the peer is no longer available, and Mary’s service execution arranges for Mary to select a different peer (effecting T4). At any time, Mary can terminate her service execution by hanging up (effecting one of the transitions into state Terminating). In Mary’s service execution, Mary’s phone plays the Caller role and the media server plays the Msr role.

3. DEVELOPING A SERVICE WITH SYNCHRONIZATION CONTRACTS

Synchronization contracts (3.1) facilitate programming a service to emulate a business machine in two ways. First, they enable a straightforward method for programming a service so that the code is easily traceable to the business machine (Section 3.2). Second, they permit use of a synchronization framework to implement the synchronization logic (Section 3.3).

3.1 Synchronization contracts

A synchronization contract consists of a set of transition clauses, each corresponding to a transition in the business automaton and containing:

- a name clause — specifying a unique transition name‡
- a guards clause — specifying a set of boolean guards, which express conditions on the message-processing context
- a resources clause — specifying a set of handles, which reference resources that could potentially be shared

†the message and the associated protocol session; this simple example, does not require an application session
‡not supported in the latest release of our framework, at http://www.cse.msu.edu/sens/szumo/SNeF/, name clauses will be in the next release.
A transition clause is interpreted as follows: The name identifies a unique transition \( t \) in the business automaton, the guards identify conditions that determine when processing a message corresponds to firing \( t \), and the handles designate the shared resources that a thread might access in evaluating the guards and processing the message. A servlet satisfies its contract if this interpretation is accurate—in other words, whenever the guards of a transition clause are all true for a given message-processing context, all data read or written is either local to the thread or local to some resource accessible through one of the handles.

### 3.2 Programming the servlet

Briefly, the service programmer performs the following steps to effect a design of a business machine as a servlet. First, she selects session attributes in which to store information about the state of a service execution and macros that use the servlet-container API to calculate other information needed in processing messages. Then, she writes the synchronization contract and the doXXX methods. For the synchronization contract, she writes a set of transition clauses in terms of these attributes and macros. Each doXXX method begins by calling a framework function, getTransitionName, which acquires the resources the thread needs to process the message and returns the name of the transition to be effected. The next statement, a conditional, selects a branch based on a transition name and effects the designated transition. Finally, she makes the servlet class that implements the service inherit from a framework class, which itself inherits from a base class defined by the servlet-container API.

<table>
<thead>
<tr>
<th>Name</th>
<th>Guards</th>
<th>Handles</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>mtype == OK</td>
<td>this</td>
</tr>
<tr>
<td>T2</td>
<td>role == Msr, mtype == OK, subscriber.cs == InvitingMediaServer</td>
<td>this, subscriber</td>
</tr>
<tr>
<td>T3</td>
<td>role == Caller, mtype == ACK, cs == ConnectingMS2Caller</td>
<td>this, ms</td>
</tr>
<tr>
<td>T9</td>
<td>role == Msr, mtype == MESSAGE, mpeer.cs != TalkingWithMediaServer, subscriber.cs == TalkingWithMediaServer</td>
<td>this, subscriber, mpeer, mpeer.ms</td>
</tr>
</tbody>
</table>

Table 1. Names, guards, and handles for some example transition clauses

Table 3.2 illustrates names, guards, and handles for some example transition clauses for our dating service. It assumes the service programmer chooses the attributes and macros described in Figure 1.

### 3.3 Middleware Synchronization Framework

The purpose of the synchronization framework is to free a service programmer from having to write low level code to infer what resources a thread needs in processing a message and then to acquire them; instead, she declares a synchronization contract, which is automatically negotiated when a doXXX method invokes the getTransitionName function. Based on the contract provided by the service programmer, the getTransitionName function dynamically infers the resources a thread will need to process the message. It then proceeds to negotiate on behalf of the thread to acquire the needed resources, potentially blocking the thread and yielding to another thread contending for a resource that it needs in order to be fair or to avoid deadlock. In any case, the middleware returns control to the application code only once the needed resources are acquired. On completion of message processing, the synchronization framework automatically releases all resources held by a thread.

Benefits of handling synchronization in this manner include enhanced traceability of application code to a business state machine, simplified reasoning about correctness, better support for reuse of services, and flexibility in permitting use of “pluggable” synchronization protocols. Additionally, it can guarantee absence of data races and deadlock provided the service satisfies its contract and the synchronization framework correctly implements negotiation.

### 4. COMPOSITIONAL REASONING

An important rationale for basing the implementation of a service on a business machine is that state machine models are intuitive, and they can be flexibly composed and rigorously analyzed. For example, given some number of clients who invoke a service, we can model the overall behavior of the service as a process in Finite
State Processes (FSP)\(^1\) by (1) writing a process \(B\) representing the business machine, (2) for each client, creating the client process \(c:B\), in which actions of \(B\) are prefixed with a unique client name \(c\), (3) identifying shared transitions,\(^2\) and (4) forming the parallel composition of the client processes under a renaming that forces shared transitions to occur simultaneously. Using the Labeled Transition System Analyzer (LTSA) tool\(^3\) we can then interactively simulate scenarios of interest, formally verify or refute temporal properties, and generate error traces. For instance, we might verify that the dating service never connects two different subscribers to the same peer and that a subscriber cannot be in state \texttt{Pairing} if the selected peer is in state \texttt{Terminating}. Another use of composition in reasoning about services is in modeling a service composed of multiple sub-services. We can obtain a model of the composition by composing appropriately specialized models of the sub-services, and then we can analyze the composite model to check for errors in the overall business logic. For instance, to check if a subscriber who phones the dating service will eventually be connected to the media server provided she does not hang up first, we might analyze an FSP process obtained by composing a process representing some number of concurrent dating-service executions with a FSP process representing the same number of concurrent media-service executions (produced from a business machine describing the media service’s business logic) under an appropriate renaming to capture the messaging semantics. To reduce potential for state explosion, we might hide actions in the sub-processes not pertinent to the property or to interactions between the dating-service and media-service executions and minimize the sub-processes prior composing them.

Synchronization contracts come into play because they can simplify the problem of programming a servlet that correctly emulates a business machine. A potential pitfall in the analysis scenario outlined above is that the validity of such analysis depends on transitions being atomic, but the processing of messages by a service is not atomic. Thus, to ensure the analysis results hold for an actual service, we need to know, not just that the \texttt{doXXX} methods correctly update the execution state, but also that each behavior of the service is equivalent to a behavior in which messages are processed serially. The contract-based method for developing a servlet (Section 3.2) facilitates showing that the \texttt{doXXX} methods correctly implement the business logic by producing methods that do not mix synchronization code with business code and by using a conditional statement whose conditions explicitly identify the names of the transitions effected by the branches. In addition, if a servlet satisfies its synchronization contract, invoking the \texttt{getTransitionName} function creates an \textit{isolated zone},\(^5\) consisting of all resources a thread might access in processing the message—thereby guaranteeing that the message-processing operations in a servlet that uses synchronization contracts are serializable. Moreover, in the service domain, it should be possible to determine the handles a thread might use when processing a message by inspecting the code, since the thread locates each resource it accesses using the servlet-container API to navigate from the message-processing context to the resource.

The previous discussion assumes that the \texttt{getTransitionName} is correct. Specifically, it assumes

- if the value returned by \texttt{getTransitionName} designates transition \(t\) of the business automaton and if \(g\) and \(H\) denote, respectively, the guard and the set of handles in the transition clause engendered by \(t\), then
  - \(g\) is true and
  - \texttt{getTransitionName} acquires locks on all resources referenced by handles, if any, in \(H\)
- the locking strategy does not incur deadlock.

Thus, any deadlock-free strategy for acquiring multiple resources can be used in implementing \texttt{getTransitionName}. We have evaluated three different strategies, which provide different performance tradeoffs.\(^6\)

5. SUMMARY

In summary, we believe that synchronization contracts provide substantial benefits for developing and reasoning about services. A servlet developer writes synchronization contracts in lieu of writing low level synchronization code. A generic synchronization framework then uses the contracts to automatically synchronize threads running

\(^1\text{e.g., the dashed transition in } c:B \text{ from } c:\text{TalkingWithMediaServer} \text{ to } c:\text{Pairing}[c'] \text{ and the corresponding solid transition in } c':B \text{ from } c':\text{TalkingWithMediaServer} \text{ to } c':\text{Pairing}[c']\)
in the container. Encapsulating the synchronization logic within a well-engineered framework simplifies programming, evolution, and maintenance, while preventing data races and deadlock. The added transparency between a state machine design of a service’s business logic and the implementation provides benefits for verification—for example, allowing a service developer to check that an implementation correctly effects transitions of a business machine. The approach also supports compositional reasoning about services based on business-machine designs.

ACKNOWLEDGMENTS

Partial support was provided for this research by NSF grant CCF 0702667, LogicBlox Inc., and AT&T Research Laboratory. The authors also thank G. Bond, E. Cheung, H. Purdy, T. Smith, V. Subramonian, and P. Zave for indispensable explanations, feedback and guidance.

REFERENCES