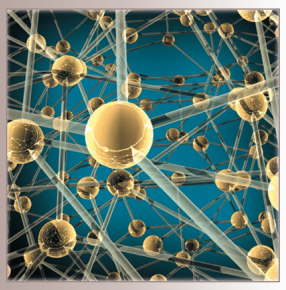

Separating Operational and Control Behaviors

A New Approach to Web Services Modeling

The ability to develop flexible business applications is one of the ultimate objectives behind the use of Web services. Before taking part in such applications, each Web service should be modeled so that service engineers can monitor its execution and identify and address design problems early on. The authors propose a novel approach for modeling Web services that distinguishes operational behavior, which defines the business logic underpinning the Web service's functioning, and control behavior, which guides the operational behavior's execution progress by identifying the actions to take and the constraints to satisfy. The authors' prototype system assists service engineers in specifying, enforcing, and monitoring these behaviors, thereby achieving a better design.

Quan Z. Sheng
University of Adelaide, Australia

Zakaria Maamar
Zayed University, UAE

Hamdi Yahyaoui
Kuwait University, Kuwait

Jamal Bentahar
Concordia University, Canada

Khouloud Boukadi
University of Blaise Pascal, France

As evidenced by academic and industrial initiatives that address issues ranging from composition to discovery, semantics, and security, Web services are poised to fulfill the promise of making separate applications interoperate smoothly.[1–4] However, despite the tremendous efforts put into this technology, several challenges lie ahead before Web services can become the technology of choice for developing flexible, cross-enterprise business applications. For example, Web services are still largely perceived as simple, passive components that react upon request only, and the de facto standard for defining them — namely, the Web Services Description Language (WSDL) — doesn't show how they function or how to oversee their executions. Such drawbacks reduce Web services' participation in complex business applications, confining them to simple case studies.[2,5]

Our approach for modeling Web services uses a richer description that isolates a service from any orchestration scenario before it abstracts and separates its behavior into two types. *Operational behavior* is application-dependent, illustrating the logic that underpins the functionality of the Web service. In con-

1089-7801/10/$26.00 © 2010 IEEE

trast, *control behavior* is application-independent, controlling the Web service's operational behavior and building its execution progress. The coordination of operational and control behaviors at runtime involves conversational messages that convey details between them, thus keeping them both aware of their respective status.

A second benefit of behavior separation involves testing, analyzing, and debugging. A Web service engineer can review a Web service's business logic through its operational behavior without affecting the way the control behavior frames the service execution's progress. This is possible because both behaviors are designed independently. By separating these behaviors, our approach also helps verify a Web service design's accuracy, soundness, and completeness. For instance, a deadlock during a conversation session between operational and control behaviors can indicate a problem in the Web service design's soundness. Given that testing and debugging tasks are still difficult and that early detection can help address problems at design time,[2,6] the techniques proposed here can be extremely valuable in practice.

Web Service Behavior Modeling

Our approach to modeling a Web service starts by identifying its control and operational behaviors and then engaging these behaviors in conversations. We use statecharts to model both behaviors,[7] but other formalisms such as Petri nets would work, too.

Operational and Control Behaviors

As mentioned earlier, operational behavior is domain-application dependent (for example, real estate) and changes from one scenario to another according to different requirements, such as user (minimum age to submit a home-loan application) or security (maximum size of a password). In contrast, control behavior guides a Web service's business logic execution.

A Web service's behavior is a five tuple $B = <S, L, T, s_0, F>$, where S is a finite set of state names, $s_0 \in S$ is the initial state, $F \subseteq S$ is a set of final states, L is a set of labels, and $T \subseteq S \times L \times S$ is the transition relation. Each transition is composed of a source state $s_{sr} \in S$, a target state $s_{tg} \in S$, and a transition label $l \in L$. We qualify this transition as intra-behavior. A Web service's control and operational behaviors are instances of the service's behavior, denoted by $B^{co} = <S^{co}, L^{co}, T^{co}, s_0^{co}, F^{co}>$ and $B^{op} = <S^{op}, L^{op}, T^{op}, s_0^{op}, F^{op}>$, respectively.

Figures 1b and 1c depict, respectively, the control and operational behaviors of WeatherWS, a Web service that delivers a five-day weather-forecast report. After analyzing several real weather Web services, such as the US National Oceanic and Atmospheric Administration's (NOAA's) National Digital Forecast Database (NDFD) XML/SOAP service (Figure 1a), we came up with the operational behavior for WeatherWS as Figure 1b illustrates. When a user submits a place, either via a zipcode or a city name, WeatherWS checks its correctness. WeatherWS then converts the correct place into longitude and latitude coordinates, which it uses to retrieve the weather forecast from a weather database. If access to the database succeeds, WeatherWS delivers the report; otherwise, the operation is cancelled.

Figure 1c shows WeatherWS's control behavior, using several states extracted from transactional Web services,[8] including `activated`, `not activated`, `done`, `aborted`, `suspended`, and `compensated`, but we could use other types of states if needed without deviating from the control behavior's original purpose. A path in a Web service's behavior is a finite sequence of states and transitions starting from one state (say, s_i) and ending at another state (say, s_j):

$$p_{i \to j} = s_i \xrightarrow{l_i} s_{i+1} \xrightarrow{l_{i+1}} s_{i+2} \cdots, s_{j-1} \xrightarrow{l_{j-1}} s_j$$

such that

$$\forall i \le k \le j - 1: (s_k, l_k, s_{k+1}) \in T.$$

For example, `notActivated` $\xrightarrow{start}$ `activated` $\xrightarrow{commitment}$ `done` is a path in WeatherWS's control behavior (see Figure 1c).

Engaging Behaviors in Interaction

The operational and control behaviors' connection is specified when the service engineer works out the conversation sessions that involve the behaviors. Specification means how and when a state in the operational behavior communicates with other states in the control behavior, and vice versa. This is illustrated with the transitions between this state and the associated paths that the mapping procedure produces. We define the specification operation as

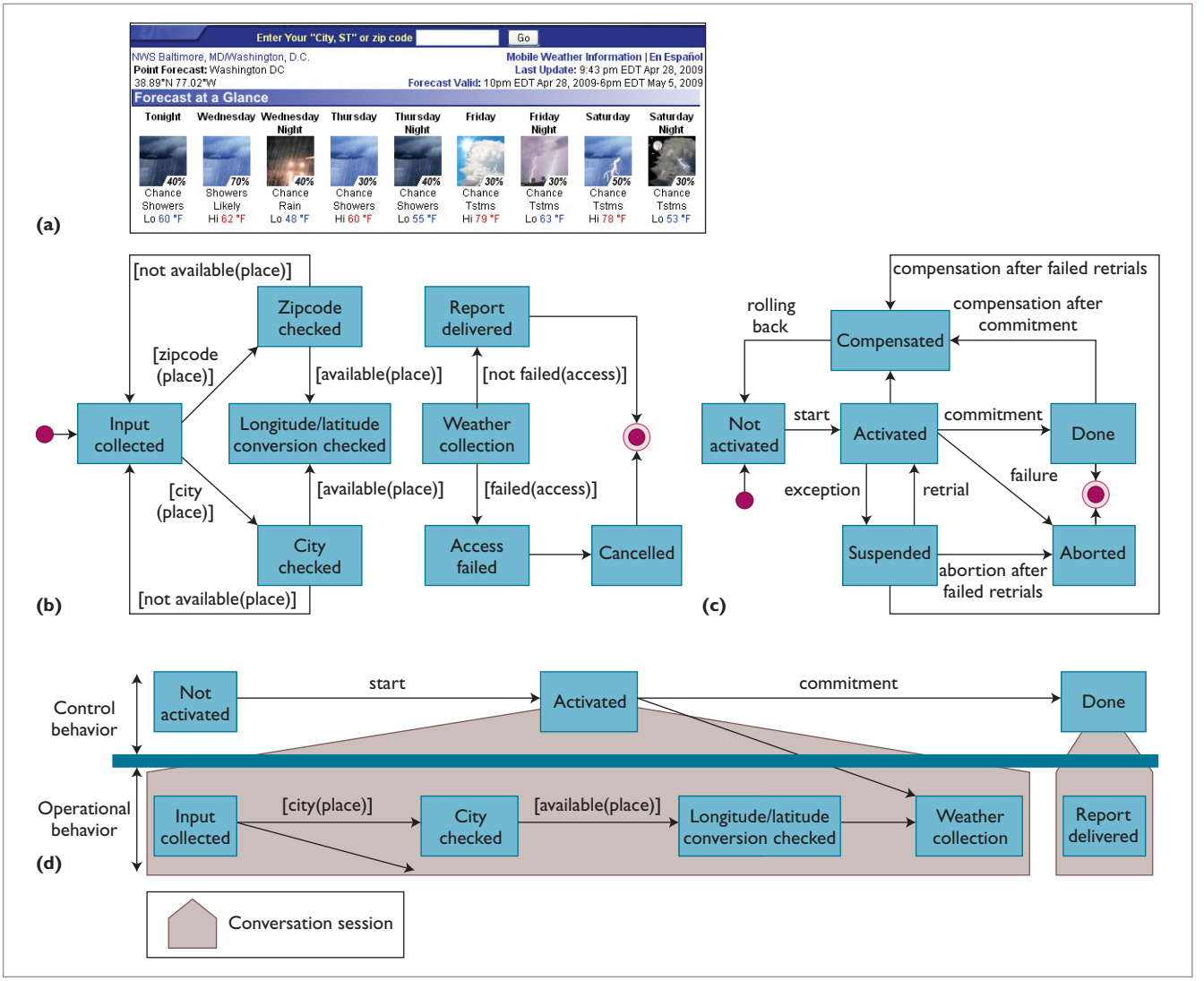

Figure 1. Example Web service. In our approach to Web service modeling, we break down (a) the five-day weather forecast from the US National Oceanic and Atmospheric Administration into its (b) operational and (c) control behaviors and specify (d) the interactions between the two behaviors for the Web service.

follows. Let L_S be the set of labels associated with the transitions connecting states in the operational and control behaviors and P^{op} be the set of paths in the Web service's operational behavior starting from any state in this behavior. We define the specification's operation by using $Spec : S^{co} \rightarrow 2^{L_S \times P^{op} \times L_S}$ and $Next: S^{co} \times P^{op} \rightarrow L^{co} \times S^{co}$.

The specification function associates each state in the control behavior with a set of (possibly empty) triples, each of which contains

- the label of the transition from s^{co} to the first state in a mapped path's operational behavior,
- the mapped path $p_{i \rightarrow j}$ itself, and
- the label of the transition from the last state in the mapped path's operational behavior back to s^{co}.

The partial function `Next` associates a given state in the control behavior and its respective mapped path in the operational behavior with the next state (and its incoming transition label) that the control behavior must now take on. Transitions that connect states in independent statecharts are qualified as *inter-behavior*.

Figure 2 shows the two types of transitions in the conversations that occur between WeatherWS's operational and control behaviors: intra-behavior (solid lines) and inter-behavior (dashed lines). In Figure 2, the inter-behavior

transitions are labeled as $label_1$, $label_2$, and $label_3$, and the state `activated` is associated with two triples as follows: Spec(`activated`) = {($label_1$, $path_1$, $label_2$), ($label_1$, $path_2$, $label_3$)}, where $path_1$ = `Input-collected` $\xrightarrow{city(place)}$ `City-checked` $\xrightarrow{available(place)}$ `Longitude/latitude-conversion` → `Weather-collection` $\xrightarrow{not\ failed(access)}$ `Report-delivered`, and $path_2$ = `Input-collected` $\xrightarrow{city(place)}$ `City-checked` $\xrightarrow{available(place)}$ `Longitude/latitude conversion` → `Weather-collection` $\xrightarrow{failed(access)}$ `Access-failed` → `Cancelled`. The two corresponding functions `Next` are represented as `Next`(`activated`, $path_1$) = (`commitment`, `done`), and `Next`(`activated`, $path_2$) = (`failure`, `aborted`).

Figure 2 shows the initiation of WeatherWS in the control behavior with the `activated` state, which it takes on following receipt of a user's input. Because of the (`activated`, $label_1$, `Input-collected`) inter-behavior transition, WeatherWS takes on the `Input-collected` state in the operational behavior, where WeatherWS's execution begins by checking the input's correctness, translating the input into longitude and latitude coordinates, and using a dedicated weather database to search the weather forecast for the input place. Afterward, two exclusive cases exist, as Figure 2 shows. In Figure 2a, everything goes fine, and the Web service delivers a five-day weather-forecast report to the user. Because of the inter-behavior transition of (`Report-delivered`, $label_2$, `activated`), WeatherWS successfully completed its operation by transiting from `activated` to `done` states in the control behavior. However, in Figure 2b, access to the database fails because WeatherWS's operational behavior indicates `Access-failed` and `Cancelled` states. Because of (`Cancelled`, $label_3$, `activated`) inter-behavior transition, WeatherWS terminated its operation in a failure by transiting from `activated` to `aborted` states in the control behavior.

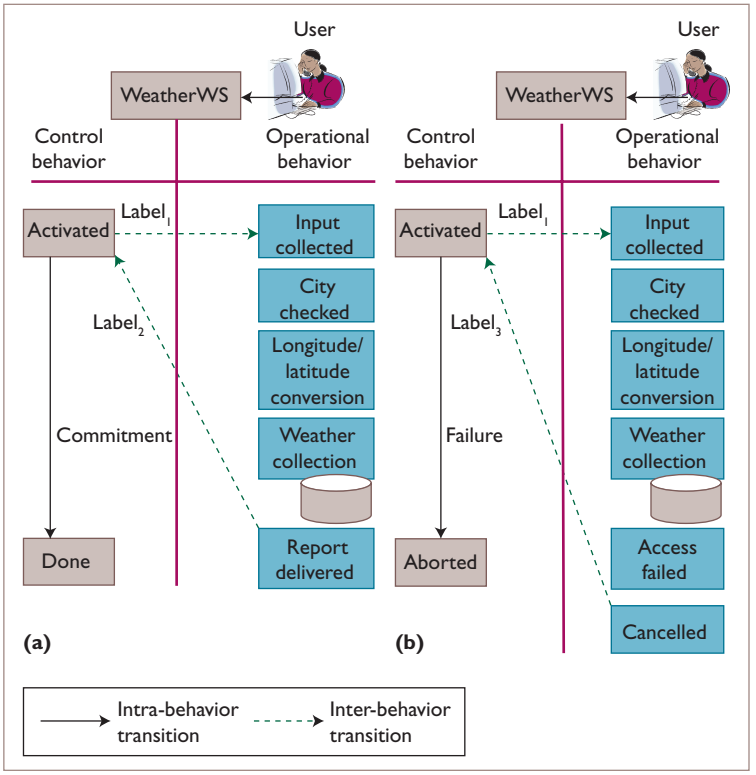

Figure 2. Conversation between WeatherWS's control and operational behaviors. For (a) the success case and (b) the failure case, solid lines represent intra-behavior transitions in conversation and dashed lines represent inter-behavior transitions.

Control and Operational Behavior Conversations

To determine appropriate conversational messages between operational and control behaviors, we can draw some analogies between these behaviors and networking protocols, resulting in seven types of messages: *sync*, *ping*, *success*, *ack*, *fail*, *delay*, and *syncreq*. Table 1 summarizes their descriptions. Each message type is associated with a category of performative, either *initiation* (from control to operational behaviors to start an execution) or *outcome* (from operational to control behaviors to report on the execution's status).

All conversational messages have input arguments with different structures. To ease the description, we group input arguments into three parts:

- str_{P1} consists of common arguments for all messages, including `ID`, `Name`, `From`, `To`, and `Trigger`.
- str_{P2} consists of arguments from the control to the operational behaviors, including `Authorized activity-time`, `Authorized passivity-time`, and `Required participants`.
- str_{P3} consists of arguments from the operational to the control behaviors, including `Counter-part ID`, `Effective activity-time`, `Effective passivity-time`, and `Execution nature`.

Figure 3a shows examples of how conversa-

Table 1. Messages implementing inter-behavior transitions.

Message type	Performative category	Description
`Sync(str)`	Initiation	Originates from a control state and targets an operational state. The purpose is to trigger the operational states' execution (including the targeted operational state) in a conversation session. `Sync` is a blocking message, which makes the control state wait for a notification back from the last operational state to execute in this conversation session.
`Success(str)`	Outcome	Originates from an operational state and targets the control state that submitted `Sync`. The purpose is to inform this control state of the operational states' successful execution in a conversation session and to return the execution thread back to this control state as well. `Success` is coupled with `Sync`.
`Fail(str)`	Outcome	Originates from an operational state and targets the control state that submitted `Sync`. The purpose is to notify this control state of the operational states' failure to execute in a conversation session and to return the execution thread back to this control state as well. `Fail` is the opposite of `Success` and is coupled with `Sync`.
`Syncreq(str)`	Outcome	Originates from an operational state and targets the control state that submitted `Sync`; denotes request for another synchronization. `Syncreq` is motivated by the possibility of unhandled exceptions in this operational state's conversation session and thus requires actions to be taken.
`Delay(str)`	Initiation	Originates from a control state and targets the operational state that was `Sync`'s destination. The purpose is to inform this operational state of the unacceptable delay in execution so that corrective actions will be taken. The targeted operational state reacts to the delay by submitting `Success`, `Fail`, or `Syncreq` to the control state, depending on this execution's current status.
`Ping(str)`	Initiation	Originates from a control state and targets the operational state that was `Sync`'s destination. The purpose is to check the liveness of the operational states in the targeted operational state's conversation session.
`Ack(str)`	Outcome	Originates from an operational state and targets a control state. The purpose is to confirm the operational states' liveness in a conversation session. `Ack` is coupled with `Ping`. If no `Ack` is received within a time limit, the control state submits another `Sync` to the operational state prior to declaring this conversation session's failure.

tional messages implement some inter-behavior transitions in Figure 2. In particular,

- `Sync(str)` handles (`activated`, label$_1$, `Input-collected`), where `str` is populated with $str_{P1} \cup str_{P2}$ (Case 1 in Figure 3a).
- `Fail(str)` handles (`Cancelled`, label$_3$, `activated`), which is correlated with (`activated`, label$_1$, `Input-collected`). Here, `str` is populated with elements from $str_{P1} \cup str_{P3}$ (Case 2 in Figure 3a).
- `Ack(str)` handles (`activated`, label$_1$, `Input-collected`), where `str` is populated with elements described as Case 4 in Figure 3a.

Conversation sessions include sequences of conversational messages (or inter-behavior transitions). A sequence is an ordered list of messages that are consistently put together. The following are some possible sequences of messages where "." stands for "next":

- `Sync.Success (resp., Sync.Fail)` refers to synchronization followed by `success (resp., failure)`.
- `Sync.Delay.Syncreq.Sync.Success` refers to synchronization followed by delay, request for resynchronization, synchronization, and finally by success, as Figure 3b shows.

Moreover, we can represent all possible sequences of conversational messages using a combination of if-then rules. Let n be the number of conversational messages exchanged dur-

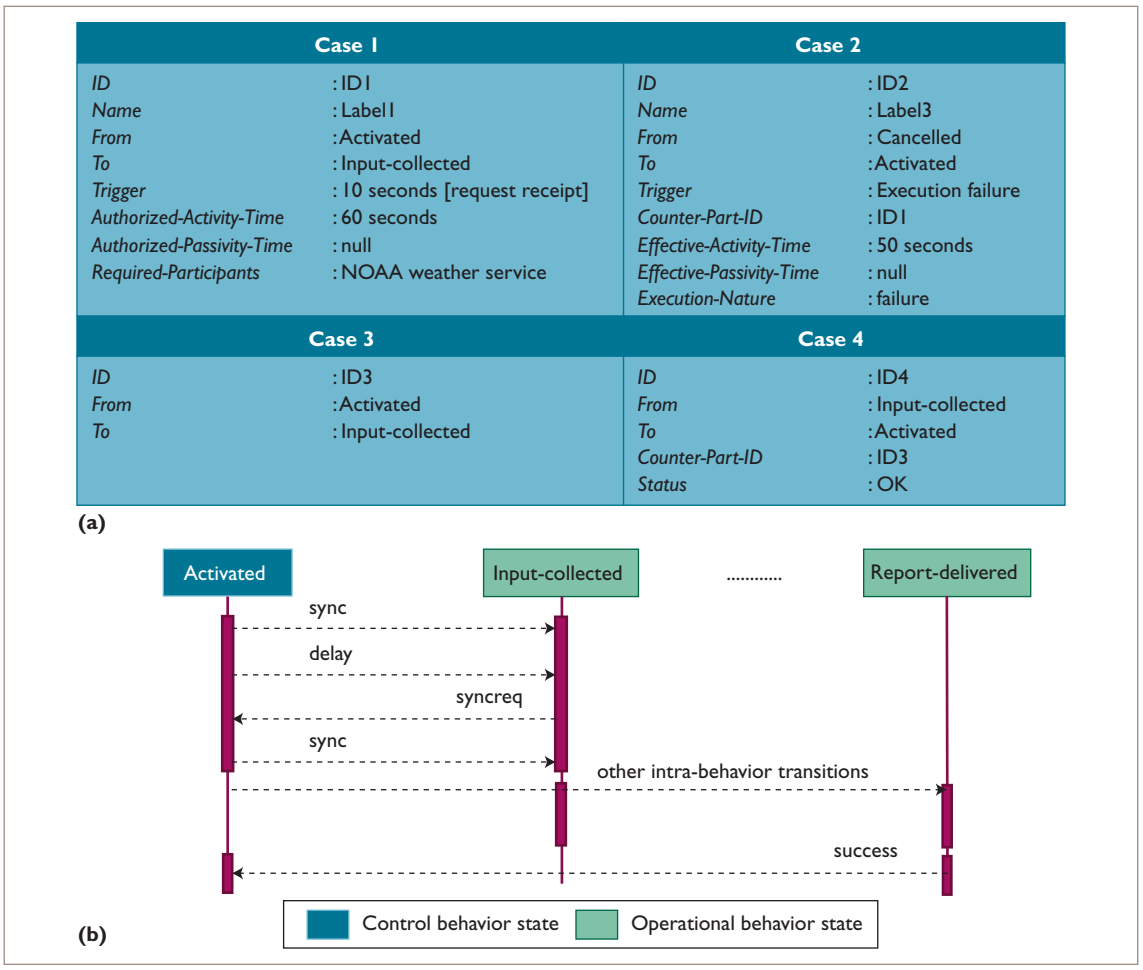

Figure 3. Examples of Web service conversation design. (a) Conversational messages. (b) A sequence of conversational messages.

ing a session, m_1, ..., m_7 be the conversational messages in Table 1, and $m_i(t)$ ($i \in \{1, ..., 7\}$) be a conversational message sent at time t from a state in the control behavior (with respect to the operational behavior) to a state in the operational behavior (with respect to the control behavior). We can represent conversational messages as follows:

$$\forall t \in [0, n-2], m_i(t) \Rightarrow \bigvee_{j \in J} m_j(t+1),$$

with $J \subseteq \{1, ..., 7\}$ and $i \neq j$, where $m_i(0)$ = Sync $\vee$ $m_i(0)$ = Ping and $m_i(n-1)$ = Success $\vee$ $m_i(n-1)$ = Fail $\vee$ $m_i(n-1)$ = Ack.

The right side of the formula defines the possible valid continuations after the input m_i. Using this formula, we can specify some conditions that help us determine that a sequence of messages is well-formed. For instance, to avoid deadlock situations, such as `Sync.Delay.Delay...`, we can put some restrictions on the way these sequences are developed. Examples of restrictions include the following:

- Each `Delay` message in a sequence should be followed by a `Success`, `Fail`, or `Syncreq` message, that is, `Delay`(t) $\Rightarrow$ `Success`(t + 1) $\vee$ `Fail`(t + 1) $\vee$ `Syncreq`(t + 1).
- Each `Sync` message in a sequence should be followed by a `Success`, `Fail`, or `Delay` message, that is, `Sync`(t) $\Rightarrow$ `Success`(t + 1) $\vee$ `Fail`(t + 1) $\vee$ `Delay`(t + 1).
- Each `Ping` message in a sequence should be followed by an `Ack` message, that is, `Ping`(t) $\Rightarrow$ `Ack`(t + 1).

In our work, a Web service design's soundness and completeness indicate a well-formed conversation between the operational and control behaviors. We can now determine if a Web service's operational and control states were properly executed (for example, a Web service

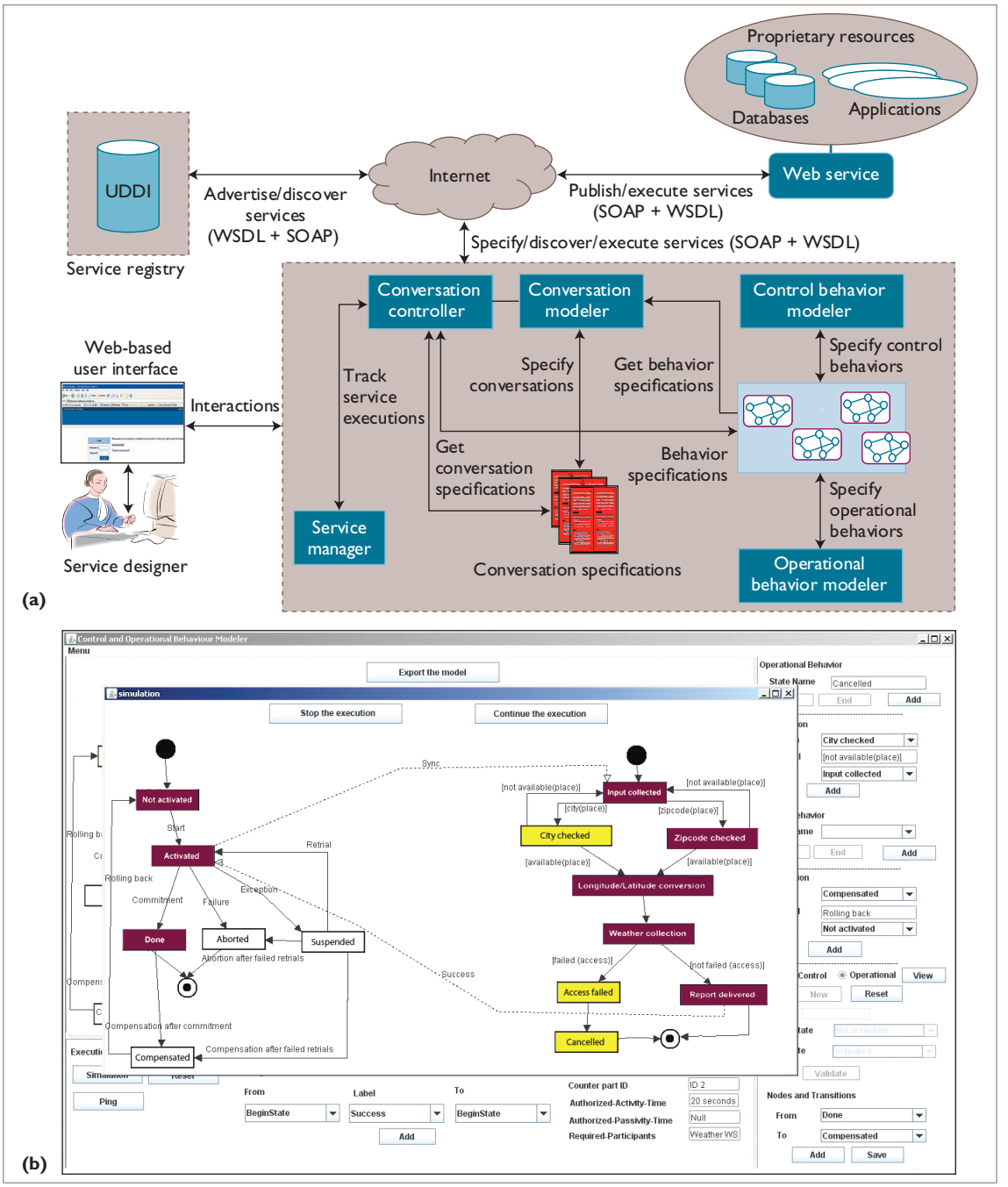

Figure 4. System implementation. (a) Prototype architecture consisting of the `ControllerBehaviorModeler`, `OperationalBehaviorModeler`, `ConversationModeler`, `ConversationController`, *and* `ServiceManager`*; and (b) service behavior specification and enforcement via a visual interface.*

isn't indefinitely waiting for an `Ack`), detect errors at design time (for example, having a `Sync` without a corresponding synchronization matching transition), and verify Web services' nonfunctional properties (for example, `Delay` messages submitted upon response-time verification). Details about how to verify Web service design appear elsewhere.[9]

System Implementation

We validated our approach using a prototype system implemented in Java. Figure 4a shows its main modules. Service engineers access our system's CASE-like tool for Web services design via a dedicated user interface. Specifically, the `ControlBehaviorModeler` and the `OperationalBehaviorModeler` assist engineers in

Related Work in Web Services

Our work is at the cross section of several initiatives that examine Web services from different perspectives, such as conversation, behavior modeling, and management.

From a conversational perspective, two specifications back the added value of conversations to Web services: the Web Services Conversation Language (WSCL; www.w3.org/TR/wscl10) and the Web Service Choreography Interface (WSCI; www.w3.org/TR/wsci). The former describes a Web service's external behavior in terms of acceptable sequences for service invocation, and the latter supports message correlation, message choreography, and service operation compensation. However, these specifications don't examine a Web service's "internal structure," focusing instead on its surrounding environment.

From a behavioral modeling perspective, representative specifications include Semantic Markup for Web Services (OWL-S, formerly DAML-S; www.w3.org/Submission/OWL-S) and Web Service Semantics (WSDL-S; www.w3.org/Submission/WSPL-S). The former organizes a Web service's description along three categories — profile (what it does), process model (how it operates internally), and grounding (how it accepts requests) — and the latter provides a lightweight approach for creating semantic descriptions of Web services. Unfortunately, these specifications don't discuss how to verify a Web service's design.

From a management perspective, several specifications exist. The W3C Web Services Architecture Working Group, for example, suggests a Web service life cycle and how it processes requests (www.w3.org/TR/2004/NOTE-wslc-20040211). However, the life cycle modeling doesn't separate operational from control concerns, making the maintenance of Web services a tedious task.

Essentially, our approach stresses the (intra-) conversation sessions that Web services initiate and thus goes beyond the simple use of conversations as a means of interaction. Compared to the aforementioned specifications and existing initiatives on Web service conversations,[1–3] this is the first work that makes operational and control behaviors accessible ("interactible") to each other via conversations.

References

1. L. Ardissono, A. Goy, and G. Petrone, "Enabling Conversations with Web Services," *Proc. 2nd Int'l Joint Conf. on Autonomous Agents & Multi-Agent Systems* (AAMAS 03), ACM Press, 2003, pp. 819–826.
2. B. Benatallah, F. Casati, and F. Toumani, "Web Service Conversation Modeling, A Cornerstone for E-Business Automation," *IEEE Internet Computing*, vol. 8, no. 1, 2004, pp. 46–54.
3. Z. Maamar, Q.Z. Sheng, and B. Benatallah, "Towards a Conversation-Driven Composition of Web Services," *Web Intelligence and Agent Systems*, vol. 2, no. 2, 2004, pp. 145–150.

specifying a Web service's control and operational behaviors, respectively. The `ConversationModeler` takes a Web service's behavior specifications as input and produces conversation specifications as output (that is, the inter-transitions and message sequences between the two behaviors). The respective modeler then translates all these specifications into XML documents for subsequent processing.

The `ConversationController` implements functions to support the conversations between operational and control behaviors. Specifically, it provides methods for managing conversation sessions, triggering transitions, and communicating with the `ServiceManager`, which is responsible for managing Web service execution. Through the `ConversationController` and the `ServiceManager`, the service engineer can track and analyze (if necessary) the service's execution according to its conversation definition (for example, whether messages are received and sent in an appropriate order).

Figure 4b shows the behavior enforcement. Upon receiving a user request, WeatherWS moves from the `notActivated` state to the `activated` state in the control behavior (that is, the left statechart diagram), where it submits a `Sync` message along with necessary details. After execution, the `Report-delivered` state returns a `Success` message back to the `activated` state in the control behavior.

The design, development, and verification of complex Web services remain a burden on those on the front line of satisfying the promise of flexible, cross-enterprise business applications. However, appropriate design guidelines are practically nonexistent, market pressure forces software to be released without proper verification, and the complexity of today's business scenarios requires both flexible and reliable Web services. Our approach attempts to tackle these challenges from the ground up, during Web service modeling and design. Future work will examine the restrictions that transactional properties place on Web service behavior.

Acknowledgments

Quan Z. Sheng's work was partially supported by the Aus-

tralian Research Council Discovery Grant DP0878367. Jamal Bentahar's work was partially supported by the Natural Sciences and Engineering Research Council of Canada (341422-07), FQRSC (Quebec), and FQRNT (Quebec). We thank the anonymous reviewers for their valuable feedback.

References

1. M. Mrissa et al., "A Context-Based Mediation Approach to Compose Semantic Web Services," *ACM Trans. Internet Technology*, vol. 8, no. 1, 2007, p. 4.
2. M.P. Papazoglou et al., "Service-Oriented Computing: State of the Art and Research Challenges," *Computer*, vol. 40, no. 11, 2007, pp. 38–45.
3. Q.Z. Sheng, B. Benatallah, and Z. Maamar, "User-Centric Services Provisioning in Wireless Environments," *Comm. ACM*, vol. 51, no. 11, 2008, pp. 130–135.
4. K. Verma and A. Sheth, "Semantically Annotating a Web Service," *IEEE Internet Computing*, vol. 11, no. 2, 2007, pp. 83–85.
5. Q. Yu et al., "Deploying and Managing Web Services: Issues, Solutions, and Directions," *Very Large Databases J.*, vol. 17, no. 3, 2008, pp. 537–572.
6. Y. Kambayashi and H.F. Ledgard, "The Separation Principle: A Programming Paradigm," *IEEE Software*, vol. 21, no. 2, 2004, pp. 78–87.
7. D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 4, 1996, pp. 293–333.
8. S. Bhiri, O. Perrin, and C. Godart, "Ensuring Required Failure Atomicity of Composite Web Services," *Proc. 14th Int'l World Wide Web Conf.* (WWW 05), ACM Press, 2005, pp. 138–147.
9. M. Kova et al., "A Formal Verification Approach of Conversations in Composite Web Services Using NuSMV," *Proc. 8th Int'l Conf. Software Methodologies, Tools and Techniques* (SoMet 09), IOS Press, 2009, pp. 245–261.

Quan Z. Sheng is a senior lecturer in the School of Computer Science at the University of Adelaide. His research interests include Web services, business process integration, data integration, and Internet computing. Sheng has a PhD in computer science from the University of New South Wales. Contact him at qsheng@cs.adelaide.edu.au.

Zakaria Maamar is a full professor in the College of Information Technology at Zayed University, Dubai, UAE. His research interests include Web services, social networks, and context-aware computing. He has a PhD in computer science from Laval University, Quebec City. Contact him at zakaria.maamar@zu.ac.ae.

Hamdi Yahyaoui is an assistant professor in the Mathematics and Computer Science Department at Kuwait University. His research interests include Web services, mobile computing, and security. Yahyaoui has a PhD in computer science from Laval University, Quebec City. Contact him at hamdi.yahyaoui@ku.edu.kw.

Jamal Bentahar is an assistant professor in the Concordia Institute for Information Systems Engineering at Concordia University, Montreal. His research interests include multi-agent systems, Web services, trust and reputation, and argumentation. Bentahar has a PhD in computer science and software engineering from Laval University, Quebec City. Contact him at bentahar@ciise.concordia.ca.

Khouloud Boukadi is a teaching assistant in the LIMOS laboratory at the University of Blaise Pascal, Clermont-Ferrand, France. Her research interests include Web services, context-aware computing, and security. Boukadi has a PhD in computer science from Ecole des Mines, Saint Etienne, France. Contact her at boukadi@isima.fr.