Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 105 (2004) 147–158

www.elsevier.com/locate/entcs

Formal Specification of a Web Services Protocol

James E. Johnson, David E. Langworthy, Leslie Lamport

Microsoft

Friedrich H. Vogt

University of Technology Hamburg-Harburg

Abstract

We describe a use of formal methods to specify and check a Web Services protocol. The Web Services Atomic Transaction protocol was specified in TLA$^+$ and checked with the TLC model checker. A modest effort revealed oversights that caused unanticipated behaviors of the protocol; these were corrected by clarifications and changes to the protocol.

Keywords: specification, standards, tla, verification

1 Introduction

Web Services (WS) are meant to overcome the integration problem for heterogeneous distributed applications [2]. A large number of protocols that use the SOAP [9] conventions for message exchange are now under development. Current and future applications will depend on the correctness of these protocols. As with any distributed system, WS protocols can permit obscure behaviors that are hard to understand, and it will be very difficult to debug them. Since the dependence on Internet-based systems is increasing dramatically, it is important that these protocols operate correctly.

Our goal is to apply formal methods to the process of standardizing WS protocols. To test our approach, we selected a non-trivial example—namely, the WS Atomic Transaction protocol [3]. (Two of the authors were involved in

doi:10.1016/j.entcs.2004.02.022

designing the protocol.) Our experience, reported here, shows that writing and model checking a TLA$^+$ specification can help eliminate errors and ambiguities in a protocol. TLA$^+$ can be used to help achieve the reliability required in the design of a WS protocol standard. We also believe that writing formal specifications can aid in the process of designing WS protocols.

2 Example: Web Service Atomic Transaction

In a distributed application system, resources like databases and/or caches are accessed by different processes. If the resources' state changes must satisfy the well-known ACID properties [6], updates must use a commit protocol. WS Atomic Transaction (WS-AT) is a protocol designed for this purpose.

The transaction protocol is begun by an *initiator* process and controlled by a *transaction coordinator*. *Participant* processes that manage the resources register for the transaction. At some point, the initiator can decide to abort or try to commit the transaction. Roughly speaking, the goal of the protocol is to guarantee that the initiator and the participants agree on whether the transaction is committed or aborted.

All of the protocol's messages go to or from the transaction coordinator. Any communication between the initiator and the participants is application specific and is not part of the protocol.

WS-AT is based on the well-known two-phase commit (2PC) protocol [1]. However, it has two non-standard features. The first is the registration procedure. Registration is often ignored in textbook descriptions of 2PC. The second non-standard feature is that it distinguishes two classes of participants, called *volatile* and *durable*, based on the type of resource that they are expected to manage. A cache is an example of a volatile resource; a conventional database is a durable resource. The standard 2PC protocol uses two phases, prepare and commit. WS-AT uses three phases, prepare for volatile participants, prepare for durable participants, and commit. A participant may register any time before the beginning of the prepare phase for its class. Thus, durable participants can still be registering during the volatile participants' prepare phase.

3 Formal Specification and TLA$^+$

A WS protocol standard should describe precisely the behavior relevant for interoperability. It should omit internal implementation details such as when records are written to stable storage. It should permit someone to implement one of the parties in the protocol without knowing anything about how the other parties are implemented. Therefore, a standard should be an unambiguous and complete description of the allowed behavior of the protocol's participants. WS standards have formal mechanisms to specify the format of XML data structures [10] and service interfaces [5]. However, they generally do not have methods of precisely specifying the complete behavior of a protocol. They instead employ informal descriptions of the protocol that can be imprecise, ambiguous, or incomplete; they often fail to consider unusual cases.

The need for precision and completeness in a standard naturally suggests the use of formal methods. Such methods use a well-defined language with a precise semantics for writing formal specifications of a protocol's allowed behaviors. Tools can be applied to analyze those behaviors and help check the correctness of the protocol.

There is no generally accepted method of formally specifying a WS protocol. State tables that are given in some specifications are a step in that direction, but they are usually not written as precisely as formal specifications.

TLA$^+$ [8] is a language for writing high-level specifications of concurrent systems. Unlike most specification languages, it is based on mathematics rather than programming-language constructs. This makes it extremely expressive. It has no built-in constructs for operations like sending a message, but the language is expressive enough that such operations are easily defined within a specification. Thus, instead of having to use some particular message passing semantics built into the language (for example, lossless FIFO delivery), one can specify whatever assumptions one wants to make about message delivery.

The WS-AT protocol is straightforward enough that it should be easy to describe in any serious specification language. However, as explained in Section 4 below, there is one place in the specification where what one process does depends upon the internal state of another process. This can be hard to model in some languages designed expressly for distributed systems. TLA$^+$ has been used to describe a wide range of concurrent systems, so we are confident of its ability to specify any desired WS protocol.

Because it is so mathematical, TLA$^+$ seems foreign to most engineers. However, we have found that engineers can very quickly learn to read TLA$^+$ specifications. Learning to write TLA$^+$ specifications seems to be about as hard as learning a new programming language.

4 Modeling the Protocol in TLA$^+$

In the summer of 2003, we began a small project to write a TLA$^+$ specification of the WS Atomic Transaction protocol. Two of the authors are researchers experienced in using formal methods; the other two are experts on this particular protocol, having participated in its design. The project lasted $2\frac{1}{2}$ months. It was a background activity, so the actual time spent was not large—perhaps $1\frac{1}{2}$ man-months. The result of the project was a TLA$^+$ specification of the protocol, written by the researchers, and two higher-level specifications, written by the designers with the assistance of the researchers.

The protocol specification has been read by the designers to determine that it corresponds to their idea of what the protocol does. It has been checked with the TLC model checker to determine that it is complete (specifies what should occur on the receipt of any possible message) and that it satisfies the basic correctness property of the protocol (agreement on the outcome of the transaction).

The researchers were initially given a preliminary version of the official specification [3]. (Its state table was the part that they found most useful.) They met with the designers three times, and they also asked questions of the designers by email.

The first task we faced was deciding what kind of specification to write. There is no notion of a "right" specification of a system. A specification is an abstraction that is meant to serve some purpose. We found that there were basically two ways to model the protocol. One was to faithfully formalize the description of the protocol in the official specification. The second was to write a simpler model whose primary goal was to verify the completeness and correctness of the protocol. We chose the second option. We therefore collapsed multiple states from the official specification's state table into single states in the TLA$^+$ specification. This reduced the total number of protocol states, making the specification easier to check. Other modeling questions that we faced were:

- Whether communication between the initiator and the coordinator should be internal or by messages. The designers decided it was all right to let it be internal.
- What assumptions should be made about the message-passing infrastructure. The designers decided that the specification should allow messages to be reordered, lost, and duplicated.

Additional questions arose because some aspects of the protocol were not described clearly enough in the official specification to permit their formal specification. An important example of this was the registration procedure. The researchers did not understand why a *Register* message from a new participant could not arrive after the coordinator had forgotten all about the transaction. This could not be resolved by inspecting the official specification because registration is described in a different standard (the WS-Coordination protocol standard [4]). The official specification did not clearly explain the interaction between these two protocols. We eventually decided to let the TLA$^+$ specification describe the necessary synchronization between registering processes and the coordinator without indicating how it is achieved. Thus, the specification has an action by one process enabled by a predicate on the state of another process, without describing how the first process learns about the second process's state. (This is easily expressed in TLA$^+$, but would be difficult in a specification language based on communicating automata.)

Most of the effort consisted of resolving these issues, many of which were discovered only while writing the specification. The complete specification consists of about 350 lines of TLA$^+$ plus 500 lines of comments. Once one understands what a protocol does and how is should be modeled, actually writing its specification is not hard.

The specification is well suited to model checking. TLC checked that the basic agreement property is satisfied by a model containing four participants. For this model, the protocol has about 500,000 reachable states, with its longest non-repeating behavior containing 45 states. TLC checked it in about $4\frac{1}{4}$ minutes on a 2-processor, 2.4GHz PC. The protocol is straightforward enough that the specification is highly unlikely to contain any error that would manifest itself only on a larger model, given the topology of the communication patterns and the indistinguishability of transaction participants.

Our effort did not reveal any major issues with the core durable 2PC protocol. This was to be expected, since that part of the protocol is very well understood. However, it did expose several problems with registration and with the volatile 2PC protocol. The use of the WS-Coordination protocol to control registration within a transaction commit protocol is new, as is the use of separate volatile and durable 2PC protocols. The interaction of these new features turned out to be more complicated than expected. Model checking revealed behaviors of the protocol that were not anticipated by the designers. This led to clarifications and changes to the official specification.

5 Overview of the TLA$^+$ Protocol Specification

The complete TLA$^+$ specification with comments is too long to present here; and like most formal specifications, it would be very hard to understand without comments. Instead, we describe its “flavor”, showing just small pieces of the specification. The complete specifications are posted on the Web [7].

Recall that the protocol involves an initiator, a transaction coordinator (TC), and a set of participants. The TC exchanges messages with the participants. As noted above, we modeled the initiator and TC as a single process. The specification declares the constant parameter *Participant*, which represents the set of participants.

We specify only the safety properties of the protocol (what is permitted to happen), not its liveness properties (what must eventually happen). This enables a very simple representation of message passing. A variable *msgs* represents the set of all messages that have ever been sent. An action that, in an implementation, would be enabled by the receipt of certain messages is, in the specification, enabled by the existence of those messages in *msgs*. Recall that we decided to allow messages to be lost, duplicated, or received out of order. Since *msgs* is a set, messages can be received in any order. Loss of a message is represented by simply not executing that action, even though the action is enabled. (Because we specify only safety properties, there is no requirement that an enabled action is ever executed.) Duplicate message delivery is allowed because messages are never removed from *msgs*, so once a message is in the set, the action of receiving that message is always enabled.

There are three other variables: *iState* is a record describing the initiator's state, *tcData* is a record describing the data maintained by the coordinator, and *pData* is an array, where $pData[p]$ is a record describing the state of participant p.

Correctness of the protocol is expressed by invariance of a state predicate *Consistency*. It asserts that the protocol is not in an inconsistent final or finishing state—that is, where one process thinks the protocol committed and another thinks it has aborted. It has two separate conjuncts, one asserting what is true if the initiator has reached the *committed* state, and the other asserting what is true if a participant has reached the *committed* state. These two conjuncts are not logically independent, but we have not eliminated the redundancy in order to make it clear what is being asserted. The definition is as follows. We do not expect the reader to understand the predicate in detail, but rather to appreciate the essentially mathematical nature of the specification. TLA$^+$ uses the convention that a list of expressions bulleted by $\land$ or $\lor$ represents their conjunction or disjunction, and indentation is used to eliminate parentheses.

```
$Consistency \triangleq$
    $\land\ (iState = \text{“committed”})$
        $\Rightarrow\ \lor\ \land\ tcData.st = \text{“ended”}$
        ∧ $tcData.res = $ "committed"
        ∧ $\forall\, p \in Participant :$
            ∨ $pData[p].st = $ "unregistered"
            ∨ ∧ $pData[p].st\ \ = $ "ended"
              ∧ $pData[p].res \in\ \{$"?", "committed"$\}$
      ∨ ∧ $tcData.st = $ "committing"
        ∧ $\forall\, p \in Participant :$
            ∨ $pData[p].st \in \{$"unregistered", "prepared"$\}$
            ∨ ∧ $pData[p].st\ \ = $ "ended"
              ∧ $pData[p].res \in\ \{$"?", "committed"$\}$

∧ $\forall\, p \in Participant :$
    ∧ $pData[p].st\ \ = $ "ended"
    ∧ $pData[p].res = $ "committed"
    $\Rightarrow$ ∧ $iState = $ "committed"
       ∧ ∨ ∧ $tcData.st\ \ = $ "ended"
             ∧ $tcData.res = $ "committed"
             ∧ $iState\ \ = $ "committed"
         ∨ $tcData.st = $ "committing"
       ∧ $\forall\, pp \in Participant :$
           ∨ $pData[pp].st \in \{$"unregistered", "prepared"$\}$
           ∨ ∧ $pData[pp].st\ \ = $ "ended"
             ∧ $pData[pp].res \in \{$"?", "committed"$\}$
```

A TLA$^+$ safety specification has the form $Init \wedge \Box[Next]_{vars}$, where $Init$ is a predicate describing the initial state, $Next$ is a formula that describes the next-state relation, and $vars$ is the tuple of specification variables. The bulk of a specification consists of the definition of $Next$. Its high-level definition is

$$Next \triangleq TCInternal \vee TCRcvMsg \vee PInternal \vee PRcvMsg$$

where the four disjuncts have the following meaning:

- $TCInternal$ describes the "spontaneous" steps of the initiator and of the TC—that is, steps not taken in response to receipt of a message.
- $TCRcvMsg$ describes the response of the TC to receipt of a participant's message.
- $PInternal$ describes the participants' spontaneous steps.
- $PRcvMsg$ describes the participants' responses to a message from the TC.

The initiator does not send or receive explicit messages.

Action $TCInternal$ is defined to be $A \wedge (\text{UNCHANGED } pData)$, where A is a disjunction of formulas describing the different operations performed by the

TC or initiator. Here is one of those disjunctions. (An action describes a state change as a relation between the new and old values of the variables, where a primed occurrence of a variable represents the new value and an unprimed occurrence represents the old value.)

$\vee$ The TC ends the volatile prepare and begins the durable prepare. It does this when it has received *Prepared* or *ReadOnly* messages from every participant that registered as volatile, and it sends a *Prepare* message to every participant that registered as durable.

$$
\begin{aligned}
&\wedge\ tcData.st = \text{“preparingVolatile”}\\
&\wedge\ \forall\, p \in Participant : tcData.reg[p] \neq \text{“volatile”}\\
&\wedge\ tcData' = [tcData\ \text{EXCEPT}\ !.st = \text{“preparingDurable”}]\\
&\wedge\ msgs' = msgs \cup [type : \{\text{“Prepare”}\},\\
&\qquad\qquad\qquad\qquad dest : \{p \in Participant :\\
&\qquad\qquad\qquad\qquad\qquad tcData.reg[p] = \text{“durable”}\}]\\
&\wedge\ \text{UNCHANGED}\ iState
\end{aligned}
$$

Action *TCRcvMsg* has the form $\exists m \in msgs : B$, where B is a disjunction, each disjunct representing the receipt of a different type of message m. We have written these disjuncts in a rather unusual way. A typical subaction has the form

$$(B_1 \wedge P_1) \vee \ldots \vee (B_n \wedge P_n)$$

where the B_i (called the “guards”) are mutually exclusive state predicates and the P_i describe the new values of variables. If none of the guards is true, then the subaction equals FALSE and no step is possible. To be able to check the completeness of our specification, we wanted TLC to flag an error if it evaluated this subaction and found none of the guards true. So, we instead wrote the subaction as

$$\text{CASE}\ B_1 \rightarrow P_1 \ \Box\ \ldots\ \Box\ B_n \rightarrow P_n$$

These two expressions have the same meaning if exactly one of the guards is true. But the latter expression is undefined if none of the guards is true, causing TLC to report an error. Here is one of the disjuncts in the definition of *TCRcvMsg*.

$\vee$ m is a *Committed* message.

$\wedge\ m.type = \text{“Committed”}$

$\wedge$ CASE The normal case, in which the TC is in the *committing* state. In this case, it sets the element of *tcData.reg* corresponding to the sender to *committed*.

$$
\begin{aligned}
&tcData.st = \text{“committing”}\\
&\rightarrow\\
&tcData' = [tcData\ \text{EXCEPT}\ !.reg[m.src] = \text{“committed”}]
\end{aligned}
$$

```
            □   If the TC has forgotten the transaction, then the transaction has
                been committed and m is ignored.
              ∧ tcData.st  = "ended"
              ∧ tcData.res = "committed"
            →
              UNCHANGED tcData
        ∧ UNCHANGED ⟨iState, pData, msgs⟩
              The initiator's state and the participants' data are unchanged, and no
              messages are sent.
```

The definitions of *PInternal* and *PRcvMsg* are similar, except they involve an additional existential quantification over the set of participants.

The specification ends with the definition of *Spec*, the complete specification, and a theorem asserting the invariance of *Consistency* and of *TypeOK*, a state predicate describing the types of the variables. (TLA$^+$ is an untyped language.)

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

$$\text{THEOREM } Spec \Rightarrow \Box(TypeOK \land Consistency)$$

This is the theorem that TLC checked.

6 More Abstract Specifications

We have also written two higher-level specifications—a 70-line *abstract* specification that describes only the behavior of the initiator and the participants, and an intermediate-level *shared-memory* specification that also describes the transaction coordinator but eliminates all messages. We have checked that both of these higher-level specifications are implemented by the protocol specification described above.

The purpose of Web services is to provide a framework of interoperable standards that allow the development of secure and reliable transactional systems and applications. We therefore want to specify not only WS-AT, but also systems built on top of it. Our abstract specification is useful for this because it is simpler than the complete specification.

The complete WS-AT protocol specification can be verified quickly on a modern processor. But once this is done, it does not need to be done again every time a client system is validated. TLC requires 253 seconds to check all behaviors of the full specification for a system with four participants. But it can check the behaviors of the abstract specification with 4 participants in less than 7 seconds. This savings in time, accompanied by a large reduction in the size of the state space, will allow the checking of client systems on larger models. The procedure can be carried further by providing abstract specifications of the client systems to permit their clients to be modeled and validated more efficiently.

The abstract specification can also be used as the specification of other protocols. For example, one could obtain a new protocol by replacing 2PC with a commit protocol that uses a different communication topology. If the new protocol satisfied the abstract specification, then a client could safely use it in place of WS-AT.

Before specifying the fully abstract protocol, we specified its behavior using shared memory. The shared-memory specification is about the same length as the complete protocol specification. It specifies the requirements of the WS-AT protocol that are independent of what messages are sent. All the internal data structures and basic transitions are preserved from the complete protocol specification.

During the design of WS-AtomicTransaction, there was a great deal of discussion about the details of the protocol's messages. The shared-memory specification, in conjunction with a detailed design, would have helped to clarify the issues. It would have provided a quick test of whether a proposed change to the specification impacted the semantics, or if it was merely a different way of expressing the same basic protocol.

The higher-level specifications can also be used for explanation and experimentation. The two parts of any design activity are determining what to build and how to build it. These two parts form the two levels of a design—namely, the requirements and the detailed design. Each level informs the other through an iterative design activity. TLA$^+$ provides a *lingua franca* that can be used to specify both levels precisely. The TLC model checker can automatically verify that the detailed design satisfies the requirements. This allows changes in the design to be checked quickly for their impact on the requirements. Model checking can also show how changing the requirements affects the current detailed design.

The 2PC protocol is very well established and well understood, so with WS-AT it was clear from the beginning what we were building. With other new systems, this is often not the case. The requirements develop along with the detailed design. As soon as users have a clear understanding of what a system actually does, they think of new applications—many with new requirements. Any tool that increases the amount, fidelity, or speed of this feedback accelerates the design process.

7 Conclusion

The WS-AT official specification contains a great deal of detail that is not captured in the TLA$^+$ specification. It very carefully describes data formats and some messaging interfaces. These are details of the protocol that are ignored by the TLA$^+$ specification, which describes only its behavioral properties. Such details are straightforward and fairly non-controversial.

The TLA$^+$ specification therefore describes the part of the protocol that is generally left imprecise in current specifications, and ignores those aspects of the protocol that are already specified quite precisely. TLA$^+$ thus complements the approach taken in most existing standards. Having a TLA$^+$ specification can help prevent incompatibility among different implementations.

A precise description of a proposed standard can also help avoid misunderstanding among the writers of the standard. In a standards committee, discussions can become very abstract. It is easy for different people to interpret statements in different ways. A mathematical formula is unambiguous.

Writing formal specifications can help in designing a protocol. Being able to check quickly if a protocol specification satisfies a more abstract specification can speed the design process. A high-level specification of the protocol's requirements can also be used to check the correctness of a protocol's clients.

TLA$^+$ is a particularly good language for writing standards because it is based on ordinary mathematics. This makes its semantics particularly simple, since the semantics of a specification language is, by definition, a translation into mathematics of specifications written in the language. As one engineer said about TLA$^+$, "If I want to find out what an expression means, I can look it up in a math book."

We are now specifying other WS protocols in TLA$^+$.

Acknowledgement

Colin Campbell found a minor error in our specification when translating it into the AsmL specification language.

References

[1] Bernstein, P. A., V. Hadzilacos and N. Goodman, "Concurrency Control and Recovery in Database Systems," Addison-Wesley, Reading, Massachusetts, 1987.

[2] Box, D., *Understanding GXA*, Technical report, Microsoft Corporation (2002), URL `http://msdn.microsoft.com/library/en-us/dngxa/html/understandgxa.asp`.

[3] Cabrera, F. et al., *Specification: Web Services Atomic Transaction (WS-AtomicTransaction)* (2002), BEA Systems, International Business Machines Corporation, Microsoft Corporation, Inc. URL `http://msdn.microsoft.com/ws/2003/09/wsat/`.

[4] Cabrera, L. F. et al., *Web Services Coordination (WS-Coordination)* (2003), BEA Systems, International Business Machines Corporation, Microsoft Corporation. URL http://msdn.microsoft.com/ws/2003/09/wscoor/.

[5] Christensen, E., F. Curbera, G. Meredith and S. Weerawarana, *Web Services Description Language (WSDL) 1.1* (2001), W3C Note, URL http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[6] Gray, J. and A. Reuter, "Transaction Processing: Concepts and Techniques," Morgan Kaufmann Publishers, Inc., San Francisco, 1993.

[7] Johnson, J. E., D. E. Langworthy, L. Lamport and F. H. Vogt, *Specification of the Web Services Atomic Transaction Protocol*, URL http://research.microsoft.com/users/lamport/tla/ws-at.html. It can also be found by searching the Web for the 26-letter string obtained by removing the dashes from "`wsatom-ictransac-tion-tlaspec`".

[8] Lamport, L., "Specifying Systems," Addison-Wesley, Boston, 2003.

[9] Mitra, N., *Soap version 1.2 part 0: Primer*, Technical report, World Wide Web Consortium (W3C) (2003), URL http://www.w3.org/TR/2003/REC-soap12-part0-20030624/.

[10] Thompson, H. S., D. Beech, M. Maloney and N. Mendelsohn, *Xml schema part 1: Structures* (2001), W3C Recommendation, URL http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/.