A Formal Analysis of the Web Services Atomic Transaction Protocol with UPPAAL

Anders P. Ravn, Jiří Srba*, and Saleem Vighio**

Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, DK-9220 Aalborg East, Denmark.
{apr,srba,vighio}@cs.aau.dk

Abstract. We present a formal analysis of the Web Services Atomic Transaction (WS-AT) protocol. WS-AT is a part of the WS-Coordination framework and describes an algorithm for reaching agreement on the outcome of a distributed transaction. The protocol is modelled and verified using the model checker UPPAAL. Our model is based on an already available formalization using the mathematical language TLA$^+$ where the protocol was verified using the model checker TLC. We discuss the key aspects of these two approaches, including the characteristics of the specification languages, the performances of the tools, and the robustness of the specifications with respect to extensions.

1 Introduction

Web Services (WS) are distributed applications that interoperate across heterogeneous networks and provide services that are hosted and executed on remote systems. Web services infrastructures employ one or more layers of a web service protocol stack (see e.g. [8]), containing various standardization initiatives on aspects which need to be implemented and described in web services environments. Many protocols in the stack use the SOAP [15] conventions and are currently at various adoption stages, ranging from approved standards to proposals.

Several protocols for web services require transactional support in order to preserve consistency. A classical transaction terminates with two possible outcomes: *committed* or *aborted*. In the committed case, the outcome is made persistent and visible outside the transaction, whereas in the aborted case, all the actions taken during the transaction are cancelled. Standards for supporting transactions among web services include the WS-Coordination framework [9] developed by BEA Systems, IBM and Microsoft. Web Services Atomic Transaction (WS-AT) is a part of this framework. This specification defines three coordination protocols that are used by distributed applications which require consistent agreements on the outcome of short-lived distributed activities.

* The author is partially supported by the Ministry of Education of The Czech Republic, project 1M0545 — Institute for Theoretical Computer Science.

** The author is supported by Quaid-e-Awam University of Engineering, Science, and Technology, Nawabshah, Pakistan.

Web services protocols are in general nontrivial and their correctness is not obvious. Therefore we model WS-AT as a network of abstract state machines communicating via shared variables and, beside some other properties, verify its correctness using the model checker UPPAAL [1]. Verification of communication protocols is in general not a new topic (see e.g. [3]) but WS-AT was formally specified only recently, and analysed in [4] using the language TLA$^+$ [5] and its model checker TLC [7]. The TLA$^+$ formalization of the protocol remained very useful for the creation of our UPPAAL model. In fact, we have transferred the state transition tables specified in TLA$^+$ into our UPPAAL model so that we can make a fair comparison of the two specification languages.

We compare the TLA$^+$ model with our abstract state machine model with respect to several criteria. First of all, we consider the performance of the verification tools TLC and UPPAAL. We were able to verify the protocol for up to five participants but the verification in UPPAAL was significantly faster. Then we discuss the foundations of the two approaches as TLA$^+$ is based on a formal mathematical language while UPPAAL automata rely on imperative programming constructs and transition graphs. We mention the expressiveness of these formalisms and consider the robustness of the models with respect to a wider applicability in other protocols with a particular focus on measuring the quality of service. In conclusion, the two formalization languages complement each other and we discuss a combination of these approaches for future applications in the specification and analysis of web services protocols.

The rest of the paper is organized as follows. In Section 2, we give an overview of the web services atomic transaction protocol. Section 3 discusses the TLA$^+$ modelling approach. A UPPAAL model of the protocol is presented in Section 4. Model properties and verification results are discussed in Section 5. Finally, Section 6 provides a detailed comparison and discusses ideas for future work.

2 Overview of WS-Atomic Transaction Protocol

WS-Coordination [9] is a specification framework for the description of protocols that coordinate the actions of applications in a distributed environment. WS-Atomic Transaction [10] (or WS-AT for short) is a part of this framework that defines an atomic transaction coordination type based on the well-known ACID (Atomicity, Consistency, Isolation, Durability) principle [2]. WS-AT defines three specific agreement coordination protocols: Completion, Volatile two-phase commit, and Durable two-phase commit. The goal of the protocols is to reach an agreement between a protocol initiator and a number of protocol participants on whether the transaction should be committed or aborted. It is following the "all or nothing" policy with no compensation mechanism. All communication between the initiator and the participants is established via a transaction coordinator. See Figure 1 for the parties involved in the protocol and their communication.

The *completion protocol* is a simple communication scheme between the initiator and the coordinator. It is essentially used by the initiator to ask the

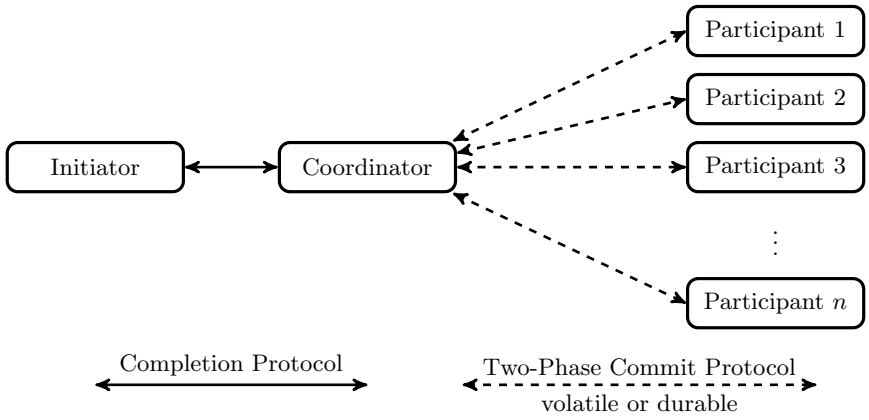

Fig. 1. WS-AT communication scheme

coordinator to try to commit or abort the transaction. The other two protocols are both based on the *two-phase commit* (2PC) protocol (see Figure 2) which coordinates registered participants in reaching their commit or abort decisions. First, coordinator invites the registered participants to prepare for committing the transaction, on which a participant can either vote for abort and terminate, or answer that it is either prepared to commit or is read-only (meaning that the participant's commit does not require any further action). The second phase of the protocol handles the actual commit, provided that the first phase was successful. In the *volatile* variant of 2PC protocol, the specification describes the communication between the coordinator and participants managing volatile resources (like a cache register). The *durable* variant deals with the coordinator-participant conversation for participants managing durable resources (like a database register). The WS-AT protocol combines the two protocols into a new three-phase (i) prepare volatile, (ii) prepare durable and (iii) commit protocol. Moreover it allows the participants to register for the protocol at any time before the prepare phase of their respective category is completed; thus as an example, durable participants can still register while the registration of volatile participants is already closed.

3 Formalization and Modelling of the Protocol

The WS-AT standard [10] provides a high-level description of the protocol. It is a collection of protocol behaviours described in English accompanied by diagrams like the two-phase commit state transition graph presented in Figure 2 and state tables for the parties involved in the protocol, see part a) in Figure 3. Unfortunately, there is no generally accepted method for formally specifying WS protocols and, as documented in [4], the WS-AT description is not sufficiently precise for a direct formalization. For example, the roles in the protocol are not

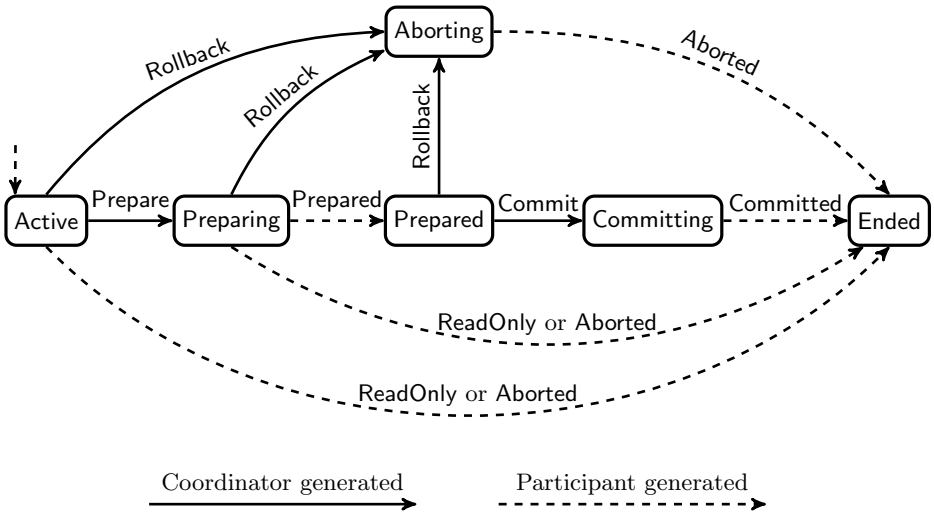

Fig. 2. Two-phase commit state transition graph

sufficiently separated from each other, which causes confusion in the protocol description as well as in the state tables. The description is silent also about what kind of communication between the parties in the protocol is assumed, as well as what data the coordinator stores about each participant.

Let us take a look at Figure 3 a) describing how the transaction coordinator handles the message Prepared arriving from some participant. The WS-AT description says that if the coordinator is in the state Preparing and receives the message Prepared, then it should register the vote and change its state to Prepared. How this rule should be interpreted when another participant sends its Prepared message is not explicitly formulated and the WS-AT description says only that a coordinator with multiple participants can be understood as a collection of independent coordinator state machines, each with its own state. Furthermore, the state tables do not describe the details of how and when the decision about commit or abort is made.

The WS-AT description can be formalized using the TLA$^+$ language as shown in Figure 3 b) taken from [4]. TLA$^+$ [5] is a formal mathematical language for specifying high-level descriptions of distributed systems. The language is very expressive; it uses predicate logic with first order quantification, which allows for expressing the protocol behaviour in a rather elegant way. There are no built-in constructs for protocol primitives like message passing, but they can be encoded using the mathematical formalism, for example as sets in case of the message passing. The TLA$^+$ expression in Figure 3 b) describes that if there is a message m of the type Prepared in the set *msgs* containing all messages sent so far, and the transaction coordinator is in the state preparingVolatile and the sender of the message is registered as volatile, or the coordinator is in preparingDurable and the

a) WS-AT:
The coordinator accepts the message Prepared. Upon receipt of this notification, the coordinator knows the participant is prepared and votes to commit the transaction.

Inbound Events	**States**			
	...	Preparing	Prepared	...
⋮	⋮	⋮	⋮	⋮
Prepared	...	Record vote; goto Prepared	...	...
⋮	⋮	⋮	⋮	⋮

b) TLA$^+$:

$\exists m \in msgs : m.type =$ "Prepared"
$\quad \wedge$ CASE
$\quad\quad \vee \wedge \; tcData.st =$ "preparingVolatile"
$\quad\quad\quad \wedge \; tcData.reg[m.src] =$ "volatile"
$\quad\quad \vee \wedge \; tcData.st =$ "preparingDurable"
$\quad\quad\quad \wedge \; tcData.reg[m.src] =$ "durable"
$\quad \longrightarrow$
$\quad\quad \wedge \; tcData' =$ [tcData EXCEPT !.reg[m.src] = "prepared"]
$\quad\quad \wedge$ UNCHANGED $msgs$
$\quad \Box$
$\quad \vdots$

c) Uppaal Timed Automata:
An edge in the coordinator automaton with the construct select `parId: Participant`, guard `guard9(parId)` and update `action9(parId)`, where

```
bool guard9(Participant parId) {
 if ((msgSrc[parId][PREPARED] == true) &&
  ((tcData.st == TC_PREPARING_VOLATILE && tcData.reg[parId] == TC_VOLATILE)
     ||
  (tcData.st == TC_PREPARING_DURABLE && tcData.reg[parId] == TC_DURABLE)))
   return true;
 return false;
}

void action9(Participant parId) {
 tcData.reg[parId] = TC_PREPARED;
}
```

Fig. 3. Specification of a selected WS-AT transition in TLA$^+$ and UPPAAL

sender is registered as durable, then the coordinator will note that this particular participant is now prepared to commit the transaction. In TLA$^+$ it is necessary to explicitly assert that this rule does not change the current set of messages. This is given by the clause "UNCHANGED *msgs*".

During the formalization of WS-AT in TLA$^+$, the authors in [4] had to make a few design decisions. First of all, it was agreed that the completion protocol between the initiator and the coordinator is modelled via internal communication as one single process. The 2PC protocol is modelled via unreliable asynchronous message passing, where the messages can be reordered, lost or duplicated. Internal timing events like "expires times out" in the state tables are modelled via nondeterminism, which provides a safe over-approximation of the behaviour but disallows the verification of any time-dependent properties. The full TLA$^+$ model is described in [4] and its correctness has been verified using the TLC model checker [7] for up to four protocol participants. In [4, 7], it is concluded that the formalization of the protocol was nontrivial and a discussion with designers involved in the formulation of WS-AT was necessary, because the WS-AT definition employs informal descriptions being imprecise, ambiguous and often fail to consider unusual cases.

The authors in [4] support their choice of TLA$^+$ as modelling language by the argument that there is a place in the specification where one process depends on the internal state of another process, and that this can be hard to model in some languages designed expressly for distributed systems.

In the section to follow, we explain an alternative way to model the WS-AT protocol with a network of state machines as provided by the model checker Uppaal [12, 1] (see Figure 3 part c) for an example of Uppaal syntax) and compare the advantages and disadvantages of both approaches. We also explain how the difficulty with rules that depend on internal states of other processes can be solved in our approach via the use of global variables and a special form of process templates.

4 The UPPAAL Model

In this section we provide the details about our Uppaal model of WS-AT protocol. Uppaal [12] is a tool for modelling, simulation and verification of networks of timed automata. The language allows to describe communicating abstract state machines with handshake synchronization and communication via shared variables. It provides a powerful C-like syntax for describing guards and updates on transitions. Uppaal allows also real time clocks. However, this feature is not used in the present model. We refer the reader to [1] for a thorough introduction to the Uppaal modelling language.

The protocol model in Uppaal consists of a process that models the initiator together with the transaction coordinator (TC for short) and a participant template that can be instantiated to as many participant processes as we want to consider.

4.1 Global Declarations

Global variable declarations of the protocol model contain the set of states for the initiator, for TC, and for the participants. We also define a set of registration

types and outcomes for TC and the participants. Finally, we model the set of messages sent between TC and participants as a bit-vector.

The protocol model consists of n participants. A type `Participant` identifies a participant using the indices among 0, 1, ..., $n-1$.

```
const int NO_OF_PARTICIPANTS = n;
typedef int[0,NO_OF_PARTICIPANTS-1] Participant;
```

Initiator and TC related declarations. The initiator's state is stored in the variable `iState` which may contain one of the following values.

```
iState ∈ {I_ACTIVE, I_COMMITTED, I_ABORTED, I_COMPLETING}
```

The information about TC is stored in the variable `tcData`. It is defined as a variable of the record type `DataTC`.

```
typedef struct {StateTC st; RegTC reg[Participant];
                ResTC res;} DataTC;
DataTC tcData;
```

The record type `DataTC` contains three components.

- A variable `st` of type `StateTC` represents all possible control states of TC.

```
st ∈ {TC_ACTIVE, TC_PREPARING_VOLATILE, TC_PREPARING_DURABLE,
      TC_ABORTING, TC_COMMITTING, TC_ENDED}
```

- The array `reg[Participant]` is defined as `RegTC` type and stores the registration state (known to the TC) for each participant `parId`.

```
reg[parId] ∈ {TC_UNREGISTERED, TC_VOLATILE, TC_DURABLE,
              TC_PREPARED, TC_READ_ONLY, TC_COMMITTED}
```

- Finally, a variable `res` of type `ResTC` represents the outcome of the protocol.

```
res ∈ {TC_COMMITTED_RES, TC_ABORTED_RES }
```

Participants' related declarations. The array `pData[Participant]` represents the data maintained by each participant and is declared as `DataP` record type.

```
typedef struct {StateP st; RegP reg; ResP res;} DataP;
DataP pData[Participant];
```

The record type `DataP` contains three components.

- A variable `st` of type `StateP` represents the control states of a participant.

```
st ∈ {P_UNREGISTERED, P_PREPARED, P_REGISTERING, P_ACTIVE,
      P_PREPARING, P_ENDED}
```

- A variable `reg` of type `RegP` records the registration status of a participant.

```
reg ∈ {P_VOLATILE, P_DURABLE}
```

- Finally, `res` of type `ResP` represents the outcome of the protocol as recorded by the given participant.

```
res ∈ {P_READ_ONLY, P_COMMITTED, P_ABORTED }
```

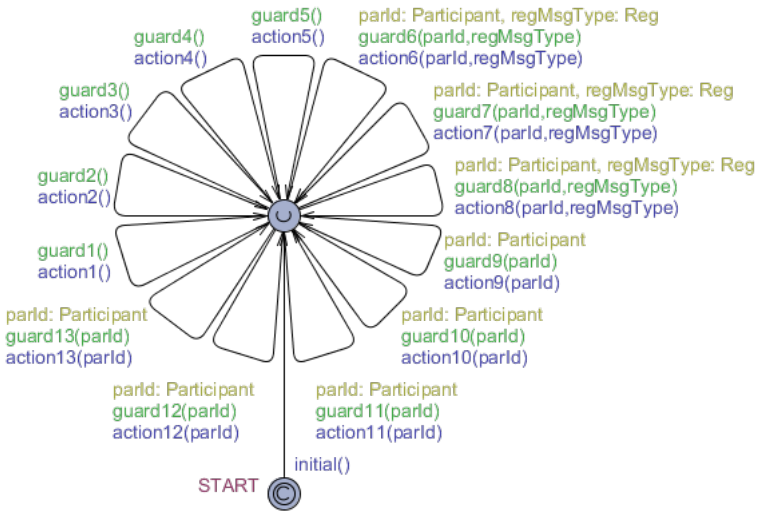

Fig. 4. Initiator-Coordinator process

4.2 Messages

Following the TLA$^+$ model, we decided to model the completion protocol between the initiator and the TC by a direct communication. For the communication between TC and participants, we adopted the model of asynchronous message passing where messages can be reordered, duplicated or lost. Unlike in TLA$^+$, we model messages as two dimensional Boolean arrays. There are two types of messages, either sent by the TC to a concrete participant or sent by a participant to the TC.

- The array `msgDest[Participant][MsgsTC]` stores messages of type `MsgsTC` sent from the TC to a participant where `MsgsTC` = {`REGISTER_RESPONSE, PREPARE, COMMIT, ROLLBACK`}. Given a participant `parId` and a message `msg` of type `MsgsTC`, the array element `msgDest[parId][msg]` has the value true if and only if the TC has already sent the message `msg` to the participant `parId`.
- The array `msgSrc[Participant][MsgsP]` represents messages of type `MsgsP` sent from a participant to TC where `MsgsP` = {`PREPARED, READ_ONLY, COMMITTED, ABORTED, REGISTER_VOLATILE, REGISTER_DURABLE`}. As before, if a participant `parId` has already sent a message `msg` to the TC, then `msgSrc[parId][msg]` has the value true, otherwise it is false.

This representation ensures that duplicate messages are ignored, and that the arrival order of messages is ignored as well.

4.3 Initiator-Coordinator Process

The model for the Initiator-Coordinator process is shown in Figure 4. The execution starts in the location START from which the protocol is set to its initial values by the function `initial()`. After this initial phase, the model has just one location which is urgent (no time elapse is allowed). Each of the transitions in the model has a guard and an update, both modelled as a function in C-like code. Due to the space limitation we present here in detail only rule 9, as already displayed in Figure 3 c). The rule has a select statement `parId: Participant` which is a convenient UPPAAL abbreviation for a set of transitions where the `parId` variable is instantiated to all possible participant identities (as defined by the type `Participant`).

Referring to Figure 3 c), the boolean function `guard9(parId)` has a parameter `parId`. The function checks if the TC has already received a prepared message from the selected participant by the test `(msgSrc[parId][PREPARED] == true)`. It also checks if the TC is currently in the preparing volatile phase of the protocol `(tcData.st == TC_PREPARING_VOLATILE)` and the registration state for the participant that the TC has recorded is volatile `(tcData.reg[parId] == TC_VOLATILE)`. The guard is satisfied also if the current TC's state is preparing durable `(tcData.st == TC_PREPARING_DURABLE)` and TC's recorded registration state of the participant is durable `(tcData.reg[parId] == TC_DURABLE)`. If one of these conditions is satisfied then the guard returns true, otherwise it returns false.

If the guard `guard9(parId)` is true then the transition can be executed and the function `action9(parId)` is called for the selected participant. The call of `action9(parId)` simply sets the TC's registration state for the participant `parId` to prepared `(tcData.reg[parId] = TC_PREPARED)`.

The reader may observe that some rules in Figure 4 do not have any select statements, others select a `parId`, and a few select the registration type of messages `regMsgType` as well. The full UPPAAL model is available as [11].

4.4 Participant Process

The template for the participants has a similar shape as the initiator-coordinator template. The final UPPAAL model contains one copy of this template for each participant in the network. Like the initiator-coordinator process, the participant process also starts in the location `START` and performs the initialization first. The participant model then consists of loop-transitions with guards and updates (actions) numbered from 14 to 22. Consult [11] for the complete model.

Following the TLA$^+$ specification, we encoded into the rules the behaviour of each participant. An example of such a rule is a situation when the participant identified as `id` receives a rollback message from the TC. As long as the participant is in one of the four prescribed states (defined in the code to follow), it will be able to read this message and end the transaction with the aborted outcome and confirm this by sending a message to the TC.

```
Consistency ≜
∧ (iState = "committed")
  ⇒ ∨  ∧ tcData.st = "ended"
       ∧ tcData.res = "committed"
       ∧ ∀ p ∈ Participant :
            ∨ pData[p].st = "unregistered"
            ∨ ∧ pData[p].st = "ended"
              ∧ pData[p].res = {"?" , "committed"}
    ∨  ∧ tcData.st = "committing"
       ∧ ∀ p ∈ Participant :
            ∨ pData[p].st = {"unregistered" , "prepared"}
            ∨ ∧ pData[p].st = "ended"
              ∧ pData[p].res = {"?" , "committed"}
∧ ∀ p ∈ Participant :
  ∧ pData[p].st = "ended"
  ∧ pData[p].res = "committed"
  ⇒ ∧ iState = "committed"
    ∧ ∨ ∧ tcData.st = "ended"
        ∧ tcData.res = "committed"
        ∧ iState = "committed"
      ∨ tcData.st = "committing"
    ∧ ∀ pp ∈ Participant :
        ∨ pData[pp].st = {"unregistered" , "prepared"}
        ∨ ∧ pData[pp].st = "ended"
          ∧ pData[pp].res = {"?" , "committed"}
```

Fig. 5. Consistency Property in TLA$^+$

```
bool guard22() {
  return (msgDest[id][ROLLBACK] == true &&
     (pData[id].st == P_REGISTERING || pData[id].st == P_ACTIVE ||
      pData[id].st == P_PREPARING || pData[id].st == P_PREPARED))
}

void action22() {
  pData[id].st = P_ENDED; pData[id].res = P_ABORTED;
  msgSrc[id][ABORTED] = true;
}
```

The other participant rules follow a similar pattern.

5 Model Properties and Verification Results

We discuss now the properties of the UPPAAL WS-AT model we described in the previous section and compare its verification results with those of the TLC model checker.

5.1 Model Properties

Consistency: The main correctness requirement is that the participants together with the initiator unanimously agree to commit or abort the transaction. This property is called consistency in [4], and its formulation in TLA$^+$ can be seen in Figure 5. Consistency is a safety property, and it is expressed by an invariant assertion. It states that the protocol is never in an inconsistent configuration where one process thinks that the transaction is committed while another process claims that it was aborted. There are two separate conjuncts in the invariant, one asserting what should be true if the initiator reached the decision to commit, and the other one asserting what is true if a participant has reached the commit decision.

We verified the same consistency property given in [4] by reformulating it to the UPPAAL syntax. The UPPAAL expression for checking consistency is given as the function `Consistency()` in Figure 6. The query is then formulated in UPPAAL's CTL logic as `A□ Consistency()` which checks whether on all computations every state satisfies the consistency invariant. It is no surprise that the WS-AT protocol satisfies this property as it is a rather standard protocol and it was recently verified using the model checker TLC [4], resulting in some modifications and improvements in the official specification.

Rules Usage: The next question one can ask is whether all rules implemented in the UPPAAL model are actually necessary, in other words if for any given rule there is some execution where the rule is actually used.

For this purpose we introduce an `observer`, which is a function added to the update of every transition in our model which simply records the number of the rule that was executed.

```
typedef int[1,22] Rules; bool flag[Rules];
void observer(int x) {   flag[x] = true; }
```

As we numbered all the rules in our models (see e.g. Figure 4) it is now easy to add the calls `observer(1)`, ..., `observer(22)` to the updates of the transitions representing the rules 1 to 22, respectively. Now in order to verify whether for example the rule 9 is ever used, we ask UPPAAL the query `E◇ flag[9]`. In this way we verified (again as expected) that all rules specified in the protocol are actually used at some execution.

5.2 Performance Results

We measured the time needed for the verification of the consistency property, which is the most time demanding one as it searches the whole state-space. The tests were performed on a iMac 27in, 4 GB 1067 MHZ RAM, 3.06 GHz Intel Core 2 Duo and Leopard Snow operating system. We used UPPAAL 4.1.2 and TLA Toolbox version 1.1.0, both with the default settings. The results are shown below along with the results obtained using the TLC model checker for the protocol description given in [4]. Execution times are rounded up to seconds and we also report on the number of explored states.

```
bool Consistency() {
 return InitiatorCommittedOK() && ParticipantCommittedOK();
}
bool InitiatorCommittedOK() {
 return iState != I_COMMITTED ||
      ( tcData.st == TC_ENDED && tcData.res == TC_COMMITTED_RES &&
             AllParticipantsCommitted() ) ||
      ( tcData.st == TC_COMMITTING && AllParticipantsCommitting() );
}
bool AllParticipantsCommitted() {
 for (p=0; p<NO_OF_PARTICIPANTS; p++)
  if (!(pData[p].st == P_UNREGISTERED || (pData[p].st == P_ENDED &&
        (pData[p].res == P_READ_ONLY || pData[p].res == P_COMMITTED))))
  return false;
return true;
}
bool AllParticipantsCommitting() {
 for (p=0; p<NO_OF_PARTICIPANTS; p++)
  if (!((pData[p].st == P_UNREGISTERED || pData[p].st == P_PREPARED) ||
      (pData[p].st == P_ENDED &&
       (pData[p].res == P_READ_ONLY || pData[p].res == P_COMMITTED))))
  return false;
return true;
}
bool ParticipantCommittedOK() {
 for (p=0; p<NO_OF_PARTICIPANTS; p++)
  if (pData[p].st == P_ENDED && pData[p].res == P_COMMITTED) {
    if ( !InitCoorCommittedOrCommiting() || !AllParticipantsCommitting() )
      return false;
 }
return true;
}
bool InitCoorCommittedOrCommiting() {
 return iState == I_COMMITTED &&
        ( (tcData.st == TC_ENDED && tcData.res == TC_COMMITTED_RES) ||
          tcData.st == TC_COMMITTING);
}
```

Fig. 6. Consistency property in UPPAAL

Performance Results for Checking Consistency				
Number of participants	**TLC**		**UPPAAL**	
	Time	States	Time	States
1	1s	132	1s	143
2	1s	2 082	1s	2 621
3	6s	32 244	2s	50 537
4	1m 49s	504 306	33s	1 014 497
5	40m 37s	8 000 412	14m 36s	21 100 793

Tool	**TLC**	**UPPAAL**
specification language	TLA$^+$	timed automata network with shared variables
necessary user's background	mathematical	programming
expressiveness of spec. language	very expressive, infinite sets, relations, quantifiers, co-inductive approach	restricted, communicating state machines, C-like (but finite) data-structures, inductive approach
model checker characteristics	restricted to bounded domains, exhaustive search	verifies the full specification language (with time)
modelling/verification speed	fast modelling, slower verification	slower modelling, faster verification
verification of time/cost features	manual encoding necessary but no verification support	straightforward modelling and state-of-the-art verification support
parameterized reasoning	modelling yes, verification no	modelling yes, verification no

Fig. 7. Comparison of the model checkers TLC and UPPAAL

Comparison of the verification results indicates that UPPAAL is more efficient than the model checker TLC in terms of execution time, even though it actually explores more states. Beyond five participants, it is almost certain that UPPAAL will run out of RAM (and start swapping) and TLC may take a very long time (probably days). However, we have not tried to optimize the UPPAAL model in any way yet and we believe that related tools, like for example UPPAAL CoVer [14], may significantly improve its performance.

6 Comparison and Conclusion

We conclude by discussing the key aspects of the two approaches presented for formalization and verification of WS protocols. A summary table is in Figure 7.

Perhaps the main difference is that TLC can analyse (a subset of) the TLA$^+$ language which is based on mathematical reasoning: first order logic and a simple set theory. Reading of TLA$^+$ specifications requires training, but the authors claim that it is about as difficult as learning a new programming language [4]. The UPPAAL model of WS-AT may look more familiar to engineers even without any prior training in concurrency theory. The model in fact uses only a limited set of UPPAAL primitives. For example no synchronization between processes is employed as all message passing is asynchronous and modelled using

shared variables. The rules of the protocol are encoded in a C-like programming language.

Yet, UPPAAL requires a lower-level encoding of some protocol fragments like message passing. Messages in UPPAAL are encoded as bit-vectors, while TLA$^+$ offers an elegant and easy to read set notation. This necessarily implies a more verbose encoding of message passing in UPPAAL, but also allows for more control and a possible optimization of the performance. We have not looked in detail into the code optimization yet, but it seems that e.g. the UPPAAL CoVer tool [14] may bring a further improvement in the verification performance.

We note that while in TLA$^+$ specification of WS-AT we can count up to 33 rules used in the model, the UPPAAL code implements only 22 rules. While still modelling the protocol at exactly the same level of abstraction, the reason is that in the TLA$^+$ language one has to explicitly assert what variables an execution of a rule leaves unchanged. This is due to the co-inductive approach used in TLA$^+$ and user that thinks in imperative programming terms may find it confusing. As UPPAAL uses an inductive approach (what is not described in the rules, is not allowed), we eliminate the need to consider rules that have no effect on the protocol behaviour.

Another point we shall discuss is the possibility to extend the approaches with quantitative analysis. The quality of service has recently become an important aspect and one may wish to explicitly model for example time and cost attributes of a protocol. Already the WS-AT specification mentions the time aspects, citing [10]: "A coordination context may have an Expires attribute. This attribute specifies the earliest point in time at which a transaction may be terminated solely due to its length of operation." Both in TLA$^+$ and UPPAAL specifications the time-outs are currently modelled using a nondeterministic choice, which on one hand provides a safe over-approximation of the behaviour, but on the other hand does not allow us to ask time related queries. While time features can be specified in both formalisms, TLC does not provide any verification support for it. The analysis of time aspects in UPPAAL is straightforward, as UPPAAL is a state-of-the-art tool for continuous time modelling and (automatic) verification. Moreover, the UPPAAL related tool UPPAAL CORA [13] for cost-optimal reachability will allow an easy addition of cost features for analysis and verification of a variety of quality of service questions. For further discussion on this topic the reader may consult also [6].

To sum up, it is possible to verify the consistency of the WS-AT for up to five participants both in TLC and UPPAAL. The main problem in the process is actually the understanding of the WS-AT specification which, in its textual form, is incomplete and imprecise. The authors in [4] relied during the formalization phase on two experts that participated in designing the WS-AT protocol. Our modelling task was easier because TLA$^+$ is a fully formal language and we could find all the answers about the behaviour of the protocol in their specification. We can roughly conclude that TLA$^+$ is a more suitable language for higher-level specification of WS protocols because of it succinctness and flexibility. On the other hand, any protocol that is described in UPPAAL timed automata framework

can be also verified, which is not the case for the TLC model checker. The experimental results also show that the UPPAAL engine is noticeably faster than TLC and hence more suitable for complex protocols. For further applicability in automatic verification UPPAAL provides readily available extensions with time and cost, features that can be encoded in TLA$^+$ specifications but not necessarily verifiable in TLC nor any presently available tool.

The formalization of WS-AT shows the need for introducing a standard for the description of WS protocols. The current practice is insufficient and the standardized protocols can be ambiguous and incomplete. In our future work, we plan to investigate a higher-level language that would share some of the advantages of TLA$^+$ as specification formalism, while being more targeted directly towards WS protocols, easily understandable by software engineers, and allowing an automatic translation to verification tools such as UPPAAL.

Acknowledgments. We would like to thank Kaustuv Chaudhuri, Leslie Lamport and Stephan Merz for answering our questions related to TLC.

References

[1] G. Behrmann, A. David, and K.G. Larsen. A tutorial on UPPAAL. In *Proceedings of SFM-RT'04*, volume 3185 of *LNCS*, pages 200–236. Springer-Verlag, 2004.
[2] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
[3] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.
[4] J.E. Johnson, D. E. Langworthy, L. Lamport, and F. H. Vogt. Formal specification of a web services protocol. *Journal of Logic and Algebraic Programming*, 70(1):34–52, 2007.
[5] L. Lamport. *Specifying Systems*. Addison-Wesley, 2003.
[6] L. Lamport. Real-time model checking is really simple. In *Proceedings of CHARME'05*, volume 3725 of *LNCS*, pages 162–175. Springer, 2005.
[7] L. Lamport and Y. Yu. TLC — the TLA+ model checker, 2003. http://research.microsoft.com/en-us/um/people/lamport/tla/tlc.html.
[8] B. Mathew, M. Juric, and P. Sarang. *Business Process Execution Language for Web Services 2nd Edition*. Packt Publishing, 2006.
[9] E. Newcomer and I. Robinson (chairs). Web services coordination (WS-coordination) version 1.1, 2007. http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.1-spec-os/wstx-wscoor-1.1-spec-os.html.
[10] E. Newcomer and I. Robinson (chairs). Web services atomic transaction (WS-atomic transaction) version 1.2, 2009. http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec.html.
[11] A.P. Ravn, J. Srba, and S. Vighio. UPPAAL model of the WS-AT protocol. Available in the UPPAAL example section at http://www.uppaal.com/.
[12] UPPAAL. http://www.uppaal.com.
[13] UPPAAL CORA. http://www.cs.aau.dk/∼behrmann/cora/.
[14] UPPAAL CoVer. http://www.hessel.nu/CoVer/.
[15] W3C. SOAP version 1.2 part 0: Primer (second edition), 2007. W3C Recommendation.