Int J Softw Tools Technol Transfer (2013) 15:125–147
DOI 10.1007/s10009-012-0231-4

TACAS 2011

# Model-checking web services business activity protocols

**Abinoam P. Marques Jr. · Anders P. Ravn · Jiří Srba · Saleem Vighio**

Published online: 13 April 2012


**Abstract** Web services business activity (WS-BA) specification defines two coordination protocols BAwCC (Business Agreement with Coordination Completion) and BAwPC (Business Agreement with Participant Completion) that ensure a consistent agreement on the outcome of long-running distributed applications. To verify fundamental properties of the protocols, we provide formal analyses in the model checker UPPAAL. Our analyses are supported by a newly developed tool chain, where in the first step we translate tables with state-transition protocol descriptions into an intermediate XML format, and in the second step we translate this format into a network of communicating state machines directly suitable for verification in UPPAAL. Our results show that the WS-BA protocols, as described in the standard specification, violate correct operation by reaching invalid states for all underlying communication media except for a perfect FIFO. Hence, we propose changes to the protocols and a further investigation of the modified protocols suggests that in case of the BAwCC protocol, messages should be received in the same order as they are sent to preserve correct behaviour, while BAwPC is now correct even for asynchronous, unordered, lossy and duplicating media. Another important property of communication protocols is that all parties always reach, under certain fairness assumptions, their final states. Based on an automatic verification with different communication models, we prove that our enhanced protocols satisfy this property whereas the original protocols do not. All verification results presented in this article were performed in a fully automatic way using our new tool `csv2uppaal`.

**Keywords** Web service · Coordination protocol · Communication media model · Model checking · Analysis tool

The paper is supported by VKR Center of Excellence MT-LAB.
A. P. Marques Jr. is affiliated to the Brazilian Society of Health Informatics.
S. Vighio is supported by Quaid-e-Awam University of Engineering, Science, and Technology, Nawabshah, Pakistan, and partially by the Nordunet3 project COSoDIS.

A. P. Marques Jr.
Brazilian Society of Health Informatics,
Rua Tenente Gomes Ribeiro, 57-sala 33,
Vila Clementino, São Paulo, SP, Brazil
e-mail: abinoam@gmail.com

A. P. Ravn · J. Srba · S. Vighio (✉)
Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, 9220 Aalborg East, Denmark
e-mail: vighio@cs.aau.dk

A. P. Ravn
e-mail: apr@cs.aau.dk

J. Srba
e-mail: srba@cs.aau.dk

## 1 Introduction

Numerous protocols from the web services protocol stack [14] are currently in active development to support communication schemes that guarantee consistent and reliable executions of distributed transactions. As applications depend on the correctness of these protocols, guarantees about their functionality should be given prior to the protocols being put into industrial use. However, design and implementation of these protocols is an error-prone process, partly because of the lack of details provided in the standards [9,27]. Therefore, formal approaches provide a valuable supplement during the discussion and clarification phases of protocol standards. The advantage of formal methods is that automatic tools like UPPAAL [4] and TLC [9] can be applied to verify general correctness criteria of protocols.

In this article, we study the WS-Coordination framework [18] that includes, among others, the standards Web services-Atomic Transaction (WS-AT) [16] and Web services-Business Activity (WS-BA) [17]. The WS-AT specification describes protocols used for simple short-lived activities, whereas WS-BA provides protocols used for long-lived business activities. The WS-AT protocol has recently been in focus in the formal methods community and its correctness has been verified using both the TLC model checker [9] where the protocol was formalized in the TLA$^+$ [10] language as well as using the tool UPPAAL [26] and networks of communicating timed automata [22]. In [22], we discussed the key aspects of the two approaches, including the characteristics of the specification languages, the performances of the tools, and the robustness of the specifications with respect to extensions.

In the present work, we analyse the WS-BA standard that (to the best of our knowledge) has not yet been formally verified in the literature. It consists of two coordination protocols: Business Agreement with Participant Completion (BAwPC) and Business Agreement with Coordinator Completion (BAwCC); we provide a formal verification of both protocol types. In the conference version [23] of this article we verified a manually created UPPAAL model of BAwCC. During this work, we realised that it is far from simple to prepare the analysis; many hours are spent on understanding the protocols and on encoding state-transition tables, messages and communication media into a format accepted by a model checking tool. In particular the encoding part is a tedious and error-prone process, when done manually. The encoding of the BAwCC protocol into the model checker UPPAAL presented in [23] ends up with around 800 lines of C-code and it took at least one person month to do the encoding and check it thoroughly to remove translation bugs. As we demonstrate in this extended presentation, this process can be to a large extent automated in our new translation tool (its preliminary version was presented in [12]). Furthermore we apply our tool chain to the analysis of BAwPC protocol type.

Our analysis of the standard protocols unexpectedly reveals several problems. The *safety* property, that the protocol never enters an invalid state, is checked for a range of communication mechanisms. The main result is that the property is violated by all considered communication mechanisms but perfect FIFO (queue). Based on a detailed analysis of the error traces produced by our tool, we suggest fixes to the protocols. Moreover, in contrast to [9,22], we do not limit our analysis to only one type of asynchronous communication policy where messages can be reordered, lost and duplicated, but study different communication mechanisms mentioned in the literature (see e.g. [1,11]). This fact appears crucial as even the enhanced BAwCC protocol behaves correctly only for some types of communication media, whereas for others it still violates the correctness criteria. On the other hand, for the enhanced BAwPC protocol we were able to automatically prove its correctness even for the most liberal communication policy.

Another important property of web services applications is that they should terminate in consistent end states, irrelevant of the actual behaviour of other participating parties [8]. This kind of property is usually called *liveness* and for most nontrivial protocols it cannot be established without some fairness assumptions, such that if a particular transition is infinitely often enabled then it is also executed. In our setting, we use a more engineering-like approach by introducing tire-outs (delays before an alternative action is chosen, essentially the "execution delay" of ATP [19]) on the resubmission of messages, as this is a likely way this situation is handled in practice. UPPAAL enables us to specify the timing information in a simple and elegant way and our verification results show that with suitable timing constraints for tire-outs and minimum retransmission delays, we can guarantee the termination property for the fixed protocols, at least for the communication policies where the protocols are correct.

*Related work.* Reachability analysis is a well-known technique for the analysis of small communication protocols (see e.g. [3,28]). An approach most related to our work was presented in [15]. Here the authors perform static analysis of a three-way handshake connection establishment protocol and the alternating bit protocol via dataflow static analysis using the tool FLAVERS. They model a communication medium as a finite state automaton, but consider only limited notions of lossiness, media of fixed sizes and do not suggest any abstraction techniques. In our model checking approach, we are able to argue about correctness also for unbounded communication channels and provide an automatic encoding of the communication medium in a more compact way. Even though the verification problems for unbounded communication buffers are in general undecidable [5], partial decidability results exist for lossy communication channels [6], however with nonprimitive recursive complexity [25] which puts them among the hardest decidable problems. In our approach, we provide a practical solution that allows us to analyze complex protocols like the ones from WS-BA in a matter of seconds. Recently Lohmann [11] surveys possible communication models and divides them into (1) ordered/unordered, (2) bounded/unbounded and (3) single/multiple buffer communication. For bounded media different nonblocking sending strategies are discussed as well. In our article we focus both on ordered and unordered as well as single and multiple buffer communication strategies. Yet our main goal is to argue about the behaviour of protocols with unbounded communication via the use of model checking techniques that permit us to verify bounded media only. Moreover, we consider unreliable communication policies.

The rest of the text is organized as follows. In Sect. 2, we give an overview of the WS-BA protocols and discuss

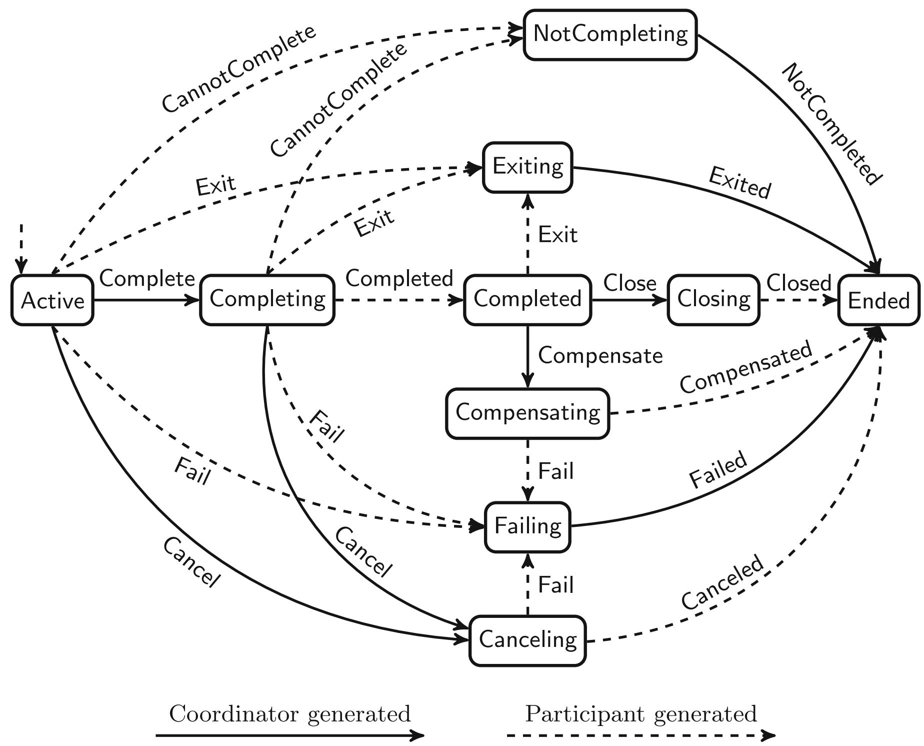


**Fig. 1** Business Agreement with Coordinator Completion

different types of communication policies in Sect. 3. Section 4 introduces the UPPAAL modeling approach used in the case study. Tool details and automatic analysis of the original and the fixed protocols are discussed in Sects. 5–7. Section 8 describes the termination under fairness property and its verification. Finally, Sect. 9 gives a summary and suggestion for the future research. The appendix contains implementation details of communication media and a full overview of the state-transition tables of the enhanced BAwCC and BAwPC protocols; the state-transitions for the original protocols can be found in [17].

## 2 WS-BA protocols

Both WS-BA [17] and WS-AT [16] are based on the WS-Coordination specification [18] and form the Web Services Transaction Framework (WSTF). WS-Coordination describes an extensible framework for coordinating transactional web services. It enables an application service to create a context needed to propagate an activity to other services and to register for coordination protocols. These coordination protocols are described in WS-AT and WS-BA specifications. WS-AT provides protocols based on the ACID (atomicity, consistency, isolation, durability) principle [7] for simple short-lived activities, whereas WS-BA provides protocols for long-lived business activities with relaxation of ACID properties.

WS-BA [17] describes two coordination types: AtomicOutcome and MixedOutcome. In AtomicOutcome the coordinator directs all participants to the same outcome, i.e. either to close or to cancel/compensate. In MixedOutcome some participants may be directed to close and others to cancel/compensate. Each of these coordination types can be used in two coordination protocols: WS-BAwPC and WS-BAwCC. A participant registers for one of these two protocols, which are managed by the coordinator of the activity.

### 2.1 Business Agreement with Coordinator Completion

A high-level state diagram for BAwCC is shown in Fig. 1. Note that the figure depicts a combined view and the concrete coordinator and participant states are abstracted away. The complete transition tables are in [17].

A participant is informed by its coordinator that it has received all requests to perform its work and no more work will be required. In this version of the protocol the coordinator decides when an activity is terminated, so completion notification comes from the coordinator: it sends a Complete message to the participant to inform it that it will not receive any further requests within the current business activity and it is time to complete the processing. The Complete message is followed by the Completed message from the participant, provided it can successfully finish its work. This protocol also introduces a Completing state between Active and Completed states. Once the coordinator reaches the Completed state, it can reply with either a Close or a Compensate message. A Close message informs the participant that the activity has completed successfully. A participant then sends a Closed notification and forgets about the activity. Upon receipt of a Closed notification the coordinator knows that the participant has successfully completed its work and forgets about the participant's state.

A Compensate message, on the other hand, instructs the participant to undo the completed work. A participant in response can either send a Compensated or Fail notification. The Compensated message informs the coordinator that the participant has successfully compensated its work for the business activity, the participant then forgets about the activity and the coordinator forgets about the participant. Upon receipt of a Fail message, the coordinator knows that the participant has encountered a problem and has failed during processing of the activity. The coordinator then replies with a Failed message and forgets about the state of the participant. The participant in turn also forgets about the activity. A participant can also send CannotComplete or Exit messages while being in Active, or Completing states. A CannotComplete notification informs the coordinator that the participant cannot successfully complete its work and any pending work will be discarded and completed work will be canceled. The coordinator replies with a NotCompleted message and forgets about the state of the participant. The participant also forgets about the activity in turn. In case of an Exit message, the coordinator knows that the participant will no longer engage in the business activity; the pending work will be discarded and any work performed will be canceled. The coordinator will reply with the Exited message and will forget about the participant. The participant will also forget about the activity. In Active and Completing states the coordinator can end a transaction by sending a Cancel message. A participant can either reply with a Canceled or a Fail notification. A Canceled message informs the coordinator that the work has been successfully canceled and then the participant forgets about the activity.

### 2.2 Business Agreement with Participant Completion

The BAwPC protocol is similar to BAwCC and differs mainly in the fact that the decision about the completion of work comes first from the participant. A participant in the BAwPC protocol sends a Completed message in its Active state to inform the coordinator about completion of work. The coordinator after receiving this message proceeds to the Completed state. In contrast to BAwCC protocol type, state-transition tables of the BAwPC protocol are lacking the state Completing. We refer the reader to [17] for a detailed description of both protocols, including the full state-transition tables.

## 3 Communication policies

The WS-BA specification is not explicit about the concrete type of communication medium for exchanging messages apart from implicitly expecting that the communication is asynchronous. In [9] the authors (two of them were designers of the specification) studied WS-AT and agreed that one should consider asynchronous communication where messages can be lost, duplicated and reordered. Indeed, the WS-AT protocol was proved correct in this setting. It seems natural to adopt the same communication assumptions also for WS-BA, however, as we show later on, the BAwCC and BAwPC protocols are not correct under such a liberal communication policy. We therefore consider a hierarchy of five different communication policies for asynchronous message passing in our study.

- *Unreliable Unordered Asynchronous Communication*
  In this type of asynchronous communication the messages may arrive in different order than they were sent and the communication medium is assumed to be unreliable as messages can be lost and duplicated. It corresponds well with the elementary UDP protocol of TCP/IP. As argued in [9], this kind of policy is conveniently implemented as a pool of messages mathematically represented by a set. Adding more messages of the same sort to a set has no additional effect and as our correctness property is a safety property, lossiness is implicitly included by the fact that protocol participants are not in any way forced to read messages contained in the pool (see [9,22] for a further discussion on this issue). In the rest of the paper, we call this kind of communication implementation SET.
- *Reliable Unordered Asynchronous Communication*
  This kind of communication still does not preserve the order of messages but it is a completely reliable medium where a message can only be received as many times as it was sent. Therefore, we have to keep track of the number of messages of the same type currently in transit. We can model this communication medium as a multiset (also called a bag) of messages. We refer to this particular implementation of the communication medium as BAG.
- *Reliable Ordered Asynchronous Communication*
  This type of communication channel represents the perfect communication medium where messages are delivered according to the FIFO (first in, first out) policy and they can be neither duplicated nor lost. The problem with this medium is that for most nontrivial protocols there is no bound on the size of the communication buffer storing the queue of messages in transit (thanks to the asynchronous nature of the communication) and automatic verification of protocols using this communication policy is often impossible due to the infinite state-space of possible protocol configurations. We refer to this communication as FIFO. It is essentially implemented by the FTP protocol of TCP/IP. However, FTP avoids unbounded

buffering by having no guarantees on timing. Delivery can be indefinitely delayed. In practice, there will be a timeout on a connection as well; but this is not part of the protocol.

- *Lossy Ordered Asynchronous Communication*
Here we assume an order preserving communication policy like in FIFO but messages can now be also lost before their delivery. The problem with unbounded size of this communication channel remains for most of interesting protocols. We call this policy LOSSY-FIFO.
- *Stuttering Ordered Asynchronous Communication*
To overcome the infinite state-space problem mentioned in the FIFO and LOSSY-FIFO communication policies, we introduce an abstraction that ignores stuttering, i.e. repetition of the same message inside of an ordered sequence of messages. We can also consider it as a lossy and duplicating medium which, however, preserves the order among different types of messages. In practice, this means that if a message is sent and the communication buffer contains the same message as the most recently sent one, then the message will be ignored. Symmetrically, if a message is read from the buffer, it can be read as many times as required providing it is of the same type. This means that the communication buffer can remain finite even if the protocol includes retransmission of messages, as, e.g. both protocols from WS-BA specification do. We call this communication type STUTT-FIFO. It is very close to the behaviour of an actual FTP transmission, because there will be a bound on the number of unacknowledged messages. In our implementation of STUTT-FIFO, we further relax the global order-preserving requirement and introduce several independent communication channels such that only messages sent via the same channel preserve their relative ordering, but two different channels do not preserve the ordering between them. We call it the *multiple channel abstraction*. We may possibly create a separate channel for each message, which would result in the communication SET. However, this will clearly not help us with the automatic analysis, as we apply this abstraction only to protocols that are not correct under the SET communication policy. Hence a more refined but fully automatic multiple channel abstraction is implemented in our tool. The idea is that for each message $m$ that appears in the protocol description, we will by static analysis compute the function *recipients*$(m)$ which contains all roles that can possibly receive the message $m$. Now every time a message $m$ is sent, it arrives to the STUTT-FIFO channel named *recipients*$(m)$, and whenever a role checks the availability of a message $m$, it does so on the channel *recipients*$(m)$. As a result, messages that arrive to the same channel preserve their relative order but messages in two different channels are unordered. The multiple channel abstraction has proved particularly useful to ensure boundedness of the BAwCC and BAwPC protocols as the communication medium is unbounded for the single-channel STUTT-FIFO communication policy. In the rest of the article, whenever mentioning STUTT-FIFO communication policy, we implicitly assume that it uses the multiple channel abstraction.

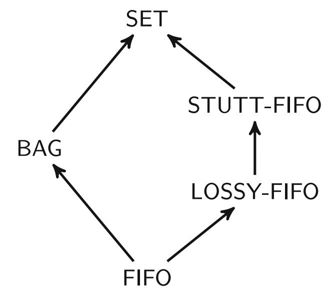


**Fig. 2** Communication policies

Figure 2 depicts the relations among the different communication media. The arrows indicate the inclusions (in the sense of possible behaviours) of the presented media. Hence any protocol execution with the FIFO communication policy is possible also in any other communication type above it. This means that if we can prove the validity of any safety property for, e.g. the SET medium, this result will hold also for any other medium below it. Conversely, finding an error trace in the protocol with, e.g. the FIFO medium implies the presence of such a trace also in any medium above it.

While the communication policies SET, BAG, FIFO and LOSSY-FIFO are well studied, the STUTT-FIFO communication we introduce in this paper is nonstandard and not implemented in any of the industrial applications that we are aware of. Although, as remarked above, FTP will work this way if the application level avoids unbounded retransmission of data. The main reason why we consider this kind of communication is that it allows us to validate the protocols in question while preserving the finiteness of the state-space. Hence we can establish safety guarantees also for the FIFO and LOSSY-FIFO communication policies, which would be impossible otherwise, as the size of such channels is not bounded in our setting.

## 4 UPPAAL encoding of WS-BA protocols

The WS-BA standard [17] provides a high-level description of both of its protocol types. It is essentially a collection of protocol behaviours described in English accompanied by diagrams like the graph shown in Fig. 1 and state-transition tables for the parties involved in the protocols. See Fig. 3a for a fragment of such a table; the appendix and [17] contain a complete collection of the tables.

**Fig. 3** Implementation of selected WS-BA rules in UPPAAL

**(a) WS-BA (BAwCC):**
Coordinator View for Inbound Events (received messages).

| Inbound Events | States | | | | |
|---|---|---|---|---|---|
| | ... | Closing | ... | Exiting | ... |
| ⋮ | ⋮ | ⋮ | ... | ⋮ | ⋮ |
| Completed | ... | resend Close goto Closing | ... | ... | ... |
| ⋮ | ⋮ | ⋮ | ... | ⋮ | ⋮ |
| CannotComplete | ... | goto Invalid-State | ... | ... | ... |

**(b) UPPAAL encoding with SET Communication (simplified):**
Here send() and available() are functions for the coordinator.

```
typedef int[0,6] MsgsTC;
const MsgsTC CANCEL_TC = 0;
const MsgsTC COMPLETE_TC = 1;
const MsgsTC CLOSE_TC = 2;
const MsgsTC COMPENSATE_TC =3;
const MsgsTC FAILED_TC = 4;
const MsgsTC EXITED_TC = 5;
const MsgsTC NOT_COMPLETED_TC = 6;
bool msgTC[MsgsTC];

void send(MsgsTC s) {
 msgTC[s] = true; }
```

```
typedef int[0,6] MsgsP;
const MsgsP EXIT_P = 0;
const MsgsP COMPLETED_P = 1;
const MsgsP FAIL_P = 2;
const MsgsP CANNOT_COMPLETE_P = 3;
const MsgsP CANCELED_P = 4;
const MsgsP CLOSED_P = 5;
const MsgsP COMPENSATED_P = 6;
bool msgP[MsgsP];

bool available(MsgP r) {
  return msgP[r]; }
```

Behaviour of transaction coordinator upon the receipt of the message Completed is modelled by a loop transition with the following guard and update (action).

```
bool guard() {
 return available(COMPLETED_P) && stTC == TC_CLOSING;  }

void action() {  send(CLOSE_TC); stTC = TC_CLOSING; }
```

Behaviour of transaction coordinator upon the receipt of the message CannotComplete is modelled by a transition to a new error-state called INVALID with the following guard and with no update.

```
bool guard() {
 return available(CANNOT_COMPLETE_P) && stTC == TC_CLOSING; }
```

Figure 3a describes how the transaction coordinator, being in its internal state Closing, handles the message Completed arriving from the participant. It will simply resend the message Close and remain in the state Closing. Also the table describes that while being in the state Closing, the coordinator does not expect to receive the message CannotCompensate from the participant, and should this happen, it will enter an invalid state. The UPPAAL implementation of this protocol is given in Fig. 3b. The syntax should be readable even without any prior knowledge of the tool, but we refer the interested reader to [4] for a thorough introduction to UPPAAL. The code in the figure first lists the names of constants that represent messages sent from the transaction coordinator to the participant and vice versa. Then it defines two functions `send` and `available` that take care of sending and checking availability of messages via the bit-vectors `msgTC` and `msgP`. The code is shown only for the simple SET implementation. For BAG, FIFO, LOSSY-FIFO and STUTT-FIFO the code is more complex but implemented in a standard way (using the C-like language in the tool). The only complication is that the data structures representing these four types of communication are in general unbounded, so to ensure automatic verification we introduce a constant upper bound on the buffer size and we register a buffer overflow in a Boolean variable called `overflow`. Details are provided in the appendix.

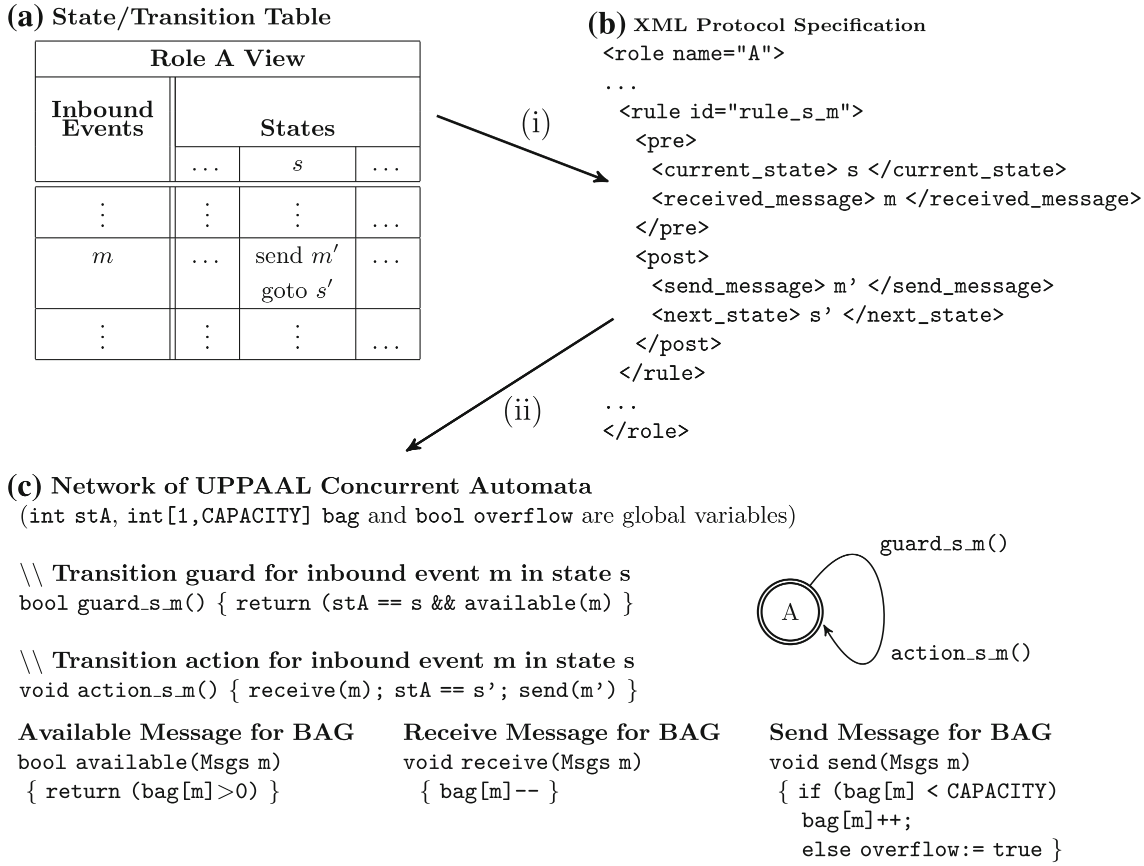


**Fig. 4** Process of automatic analysis of WS-protocols with the medium BAG

The transitions of the state tables are then implemented in the expected way as shown by the two examples in Fig. 3b. The final timed automata model consists of a process for the coordinator with two locations (normal execution and invalid state) and a similar process for the participant running in parallel with the coordinator process. All data management (states, buffer content, etc.) use C-like data structures, as this is an efficient and manageable way to handle this relatively large model. In total the C part of the implementation contains around 800 nonempty lines of code and it was created by a manual translation from the state-transition tables. The complete UPPAAL model of BAwCC can be downloaded at [21] and all protocols are also a part of our tool distribution.

In the next section, we present a tool chain that allows us to generate the UPPAAL models automatically. Compared to 800 lines in the manually created model, the computer-generated model contains 1,400 lines of code. This is mainly due to the fact that several transitions going to invalid states were joined together in the manual model, however, there is essentially no difference in the time needed to verify the models using UPPAAL.

## 5 Automatic tool support

The translation presented in the previous section has been implemented in our tool chain. Here, we assume a general situation where protocols have a finite number of roles communicating over some communication medium. This ensures that the tool is applicable to a wider range of communication protocols, as discussed in more detail also in [12]. Figure 4a describes how the Role A, being in its internal state s handles message m arriving from some other role. It will simply send a message m' to the sending role and change its state from s to s'.

Our formal analysis of such protocols starts by automatically translating the state-transition tables into an intermediate XML format [denoted as part (i) in Fig. 4], followed by a translation to networks of timed automata suitable for a direct verification in UPPAAL [denoted as part (ii) in Fig. 4]. The translation example uses the BAG communication medium this time.

As the reader can see, we created an intermediate XML representation of the state tables. The main motivation is that the translation (i) from state-transition tables to its XML representation can be replaced by another front end, allowing us for example to describe a protocol with some domain-specific language and to translate it automatically into the XML format. This provides a better modularity of our tool chain. The translation (i) is to a large extent a syntactic reformulation of the tables with added explicit definitions of states and messages that allow us to check for typos in the state-transition tables. This has proved useful during the creation of the tables in WS-BA protocols.

In part (ii) of our translation, each `pre` tag is converted into a transition guard and each `post` tag is translated to an action that is performed should the transition be executed. Note that we assume a given capacity of the `bag` data structure (limited by the constant `CAPACITY`). Should the protocol require more messages in transit for the `BAG`, the global flag `overflow` is set to true.

Translations (i) and (ii) from Fig. 4 have been implemented in the open source tool `csv2uppaal` available at [13]. The input state-transition tables are created in standard spreadsheet editors like OpenOffice and saved as csv files (textual representation of the tables). The csv files are then parsed using an awk script that generates the intermediate protocol description in the XML format with elements representing the messages, roles and their states and transition rules with pre and post conditions. The final part of the tool-chain is written in Ruby and generates files directly readable by UPPAAL (concurrent automata descriptions and a query file). Finally, the tool calls the UPPAAL verification engine to verify the properties of boundedness, correctness, termination and, though not discussed in this article, also deadlock-freeness.

The outcome of the verification is a statistics with details about the protocol, medium, roles and messages, the verification results and possibly execution traces if relevant for the verified properties. The traces are printed in a human readable form. Use of the tool chain requires no expertise with the model checker UPPAAL and is accessible to WS protocol designers without any particular training in formal verification. On the other hand, the advanced users may open the generated files in the UPPAAL GUI, experiment with simulating the protocol and ask advanced queries that are protocol specific.

The model-checker UPPAAL is nowadays so efficient that the protocols described in this article were verified within a couple of seconds on a standard laptop. Hence we did not need to apply any further optimization techniques to speed up the verification.

## 6 Analysis of WS-BA protocols

In the analysis of the WS-BA protocols, we first focus on the actual state-transition tables w.r.t. reachability of invalid states. Invalid states appear in the tables both for inbound and outbound messages. The meaning of these states is not clearly stated in the WS-BA specification but we contacted the designers via their discussion forum and received (citing [24]):

> "For outbound events, an Invalid State cell means that this is not a valid state for the event to be produced. . . . For inbound events, an Invalid State cell means that the current state is not a valid state for the inbound message. For example, for Participants in Business Agreement With Coordination Completion (table B.3) the Canceling state is not a valid state for receiving a Close message. There are no circumstances where a Participant in this state should ever receive a Close message, indicating an implementation error in the Coordinator which sent the message. This is a protocol violation . . ."

This means that in the tables for outbound events, messages that lead to invalid states are never sent (and hence omitted in the UPPAAL model) and for inbound events the possibility to enter an invalid state is a protocol violation. We call a protocol that never reaches any of its invalid states *correct* and we shall verify correctness for both protocol types in the WS-BA standard. The UPPAAL formulation of this property is shown below using the UPPAAL query language (a subset of TCTL).

```
A[] ((stTC != INVALID && stP ! =INVALID) || overflow)
```

This is a safety property asking that for all reachable protocol executions the state of the coordinator (called `stTC`) and the participant (called `stP`) is not `INVALID` or there is a buffer overflow.

Another important question we can ask about the protocol is whether the communication medium is actually bounded for WS-BA protocols or not. We call this property *boundedness* and the UPPAAL formulation of this property is

```
A[] !overflow.
```

Hence if the correctness property fails, we have found a real problem in the protocol design. For showing that a protocol is correct, we need to verify the correctness property and at the same time we need to establish that the protocol is bounded.

### 6.1 Analysis of BAwCC protocol

The correctness property for BAwCC protocol type under all five communication policies surprisingly turned out not to hold, except for the FIFO policy.

The tool automatically generated error traces leading to invalid states; one of them is depicted in Fig. 5. It is easy to see that this trace is executable both for LOSSY-FIFO and BAG communication (and hence also for any medium above

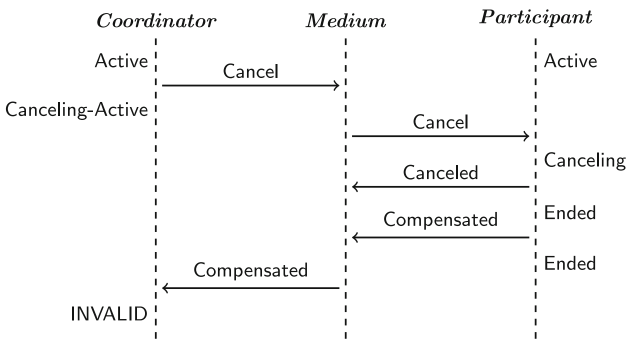


**Fig. 5** Trace in BAwCC leading to an invalid state

them in the hierarchy in Fig. 2). The main point in this trace is that the message Canceled that is sent by the participant is either lost (possible in LOSSY-FIFO) or reordered with the message Compensated (possible in BAG).

It is also clear that this trace cannot be executed in the perfect FIFO communication policy. For FIFO we were able to verify that the protocol is correct for up to six messages in transit (three from coordinator to participant and three in the opposite direction). As perfect FIFO communication is known to have the full Turing power [5], there is no hope to establish the correctness of general protocols with unbounded FIFO communication in a fully automatic way.

Furthermore, verification of boundedness for BAwCC reveals that all communication media except SET can always reach a buffer overflow for any given buffer size that we were able to verify (up to 20 messages in transit). This is a good indication that the communication buffer is indeed unbounded and a simple (manual) inspection of the protocol confirms this fact.

### 6.2 Analysis of BAwPC protocol

Similar to the BAwCC protocol type, the verification of correctness for BAwPC also returned a negative result under all five communication policies except for FIFO policy.

An error trace leading to an invalid state is shown in Fig. 6. For the same reasons as in BAwCC, this trace is executable both for LOSSY-FIFO and BAG communication (and hence also for any medium above them in the hierarchy in Fig. 2). We also found out that all the communication policies in BAwPC reach buffer overflow, except for SET. As a result the WS-BA protocols are not correct whenever the communication medium is lossy or allows for message reordering; something the protocols designers were not aware of during a manual inspection of the protocol behaviours.

## 7 Enhanced WS-BA protocols

Given the verification results in the previous section, we can claim that both WS-BA protocols are not completely

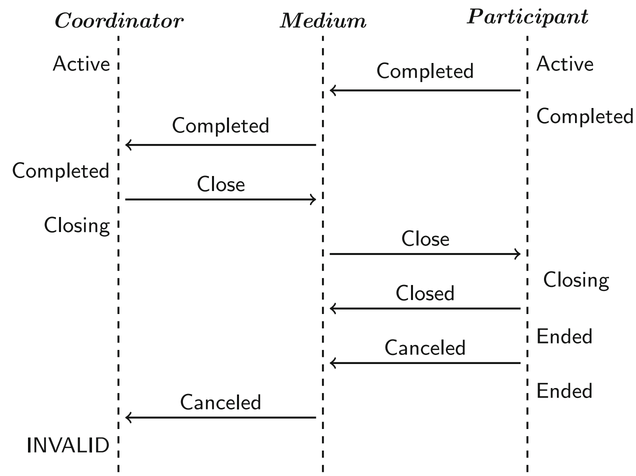


**Fig. 6** Trace in BAwPC leading to an invalid state

satisfactory as even a minor relaxation of the perfect communication policy results in incorrect behaviour. Taking into account that the protocols in WS-AT (studied in [9,22]) avoided invalid states even under the most general SET communication, we shall further analyze the WS-BA protocols and suggest improvements.

### 7.1 Enhanced BAwCC protocol

The error trace for BAwCC shown in Fig. 5 hints at the source of problems. Once a participant reaches the Ended state, it is instructed to forget all state information and just send the last message by which the transition to the Ended state was activated. The problem is that there are three different reasons for reaching the Ended state, but BAwCC allows for the retransmission of all three messages at the same time, whenever the participant is in the state Ended. As seen in Fig. 5, the participant after receiving the message Cancel correctly answers with the message Canceled, but then sends the message Compensated. This causes confusion on the coordinator's side. A similar problem can occur in a symmetric way.

In our proposed fix to the BAwCC protocol, we introduce distinct end states, both for the participant as well as for the coordinator, to avoid the confusion. The complete state tables of the enhanced protocol are given in the appendix. A communication with OASIS body responsible for the WS-BA standard confirmed that this was indeed implicitly assumed, though not reflected in the state-transition tables presented in the standard.

We modelled and automatically verified the enhanced protocol and the results are as follows. Under the STUTT-FIFO communication, the medium is now bounded with no overflow, so all verification results are conclusive. We also established that there is no execution of the modified protocol that leads to an invalid state. As argued before, this positive

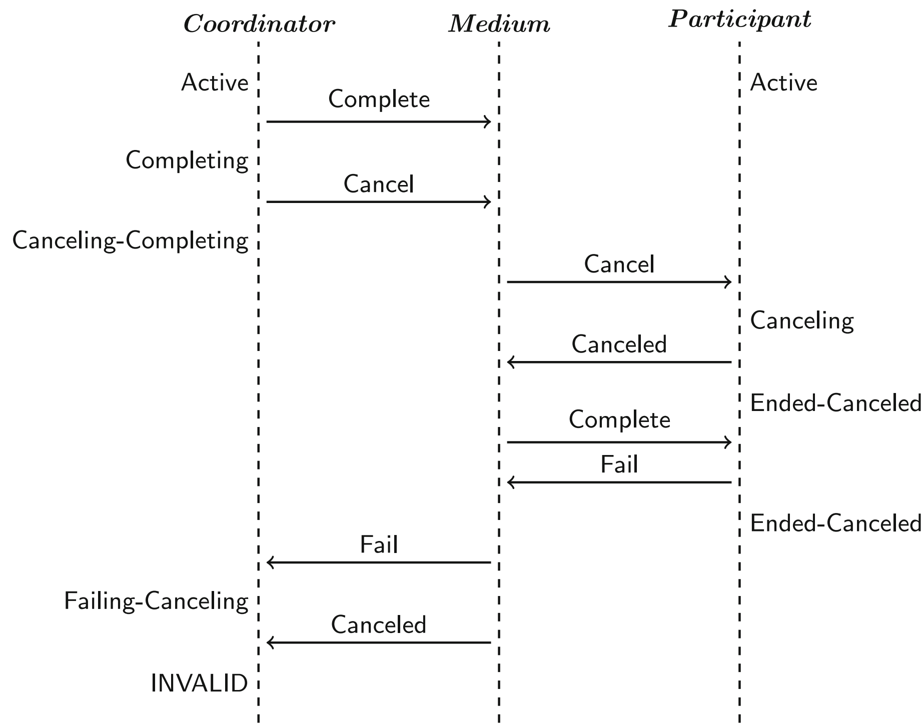


**Fig. 7** Trace in enhanced BAwCC leading to an invalid state

result holds automatically also for any less abstract media like LOSSY-FIFO and FIFO.

However, when considering the media BAG and SET representing a communication where messages can be reordered, the tool still returns error traces like the one depicted in Fig. 7. This problem is more inherent to the protocol design and the reason for the confusion is the fact that the messages Canceled and Fail sent by the participant are delivered in the opposite order.

To conclude, our enhanced protocol, unlike the original one, is immune to lossiness and duplication of messages (stuttering) as long as their order is preserved. Making the protocol robust w.r.t. reordering of messages would, in our opinion, require a substantial and nontrivial redesign of the BAwCC protocol. We have communicated the error trace in Fig. 7 to the OASIS body [20] responsible for WS-BA standards and a correction is currently under development.

### 7.2 Enhanced BAwPC protocol

The introduction of three additional end-states in the BAwPC protocol, both for the participant as well as for the coordinator, leads to the correct protocol behaviour even for the SET communication policy (that at the same time gives bounded state-space) and hence also for any other media considered in this article. Hence, unlike for BAwCC, we have obtained correctness of the protocol under the most general communication medium that allows also for reordering of messages.

## 8 Termination under fairness

In this section we turn our attention to another important property of distributed protocols, namely *termination*. Termination means that as long as the communication parties follow the protocol, any concrete execution will bring them to their end states. In UPPAAL this property for our protocols can be formulated as

```
A <> stTC == TC_ENDED && stP == P_ENDED.
```

The meaning is that in any maximal computation of the protocol, it will eventually reach a situation where the states of the transaction coordinator as well as the participant are `TC_ENDED` and `P_ENDED`, respectively. Termination is hence a liveness property.

It is clear that the original BAwCC and BAwPC protocols fail to satisfy termination, as we can reach invalid states from which there is no further continuation. This is true for all types of communication, except for FIFO, where on the other hand we cannot prove termination due to the unboundedness of the medium. We shall therefore focus on the enhanced BAwCC and BAwPC protocols and the communication medium STUTT-FIFO, respectively, SET where the protocols are correct and the medium bounded.

A quick query about termination in UPPAAL shows that it fails the property also for the enhanced protocols and the tool returns error traces that reveal the reason: there is no bound on the number of retransmissions of messages and this can create infinite process executions where the same message

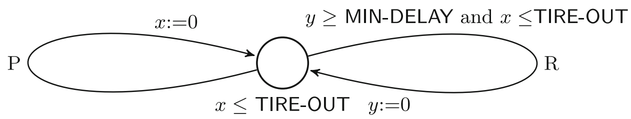


**Fig. 8** Tire-outs modelling; *P* is a progress transition, *R* is a retransmission transition

is retransmitted over and over again. This is to be expected for any nontrivial protocol and in theory the issue is handled by imposing an additional assumption on *fairness* of the protocol execution. This can for example mean that we require that whenever during an infinite execution some action is infinitely often enabled then it has to be also executed. Such assumptions will guarantee that there is a progress in the protocol execution; these assumptions are well studied in the theory (see e.g. [2]).

The complication is that fairness concerns infinite executions and is therefore difficult to implement in concrete applications. Software engineers would typically use only a limited number of retransmissions within a fixed time interval and give up resending messages after a certain time has passed.

So far, we have used UPPAAL only for the verification of discrete systems, but the tool allows us to specify also *timed* automata models and supports their automatic verification. By introducing the timing aspects into the protocol behaviours, we will be able to argue about fairness properties like termination.

We model the retransmission feature using tire-outs. A tire-out imposes a progress in the model and, as already outlined in Sect. 1, it is essentially the "execution delay" of ATP [19]. In our model, we introduce two clocks $x$ and $y$ local both for the coordinator and the participant. We also assume two global constants MIN-DELAY and TIRE-OUT, representing the minimal possible delay between two retransmissions and a tire-out time after which the protocol will not attempt to retransmit the message. Figure 8 shows an implementation of this feature in the protocol model. We already explained that the rules of the protocol are modelled using loops in UPPAAL automata and the discrete data are handled using guards and updates (not shown in the illustration). In the figure, we split all transitions into two categories: progress transitions and retransmission transitions. Retransmission transitions retransmit a message and remain in the same state, while progress transitions change the state of the participant or the coordinator after their execution. The clock $x$ represents the time delay since the last progress transition occurred (it is reset to 0 by any progress transition) and clock $y$ represents the time elapsed since the last retransmission. These two clocks allow us to restrict the behaviour of the retransmission transitions so that they are enabled only if at least the minimal delay has passed since the last retransmission and the clock $x$ has not exceeded the tire-out limit. The presence of the invariant $x \leq$ TIRE-OUT then ensures the progress.

Using the tire-out modeling as described above we were able to verify that both the enhanced BAwCC and BAwPC protocols with STUTT-FIFO (for enhanced BAwPC even with SET) communication policy satisfy the termination property for suitable constants MIN-DELAY and TIRE-OUT where, for example, the minimal delay is set to one time unit and the tire-out deadline to 30 time units. By changing these two constants, we can experiment with different timing options while making sure that the termination property is preserved. The termination check under fairness has been implemented in our tool chain and hence these answers have been provided in a fully automatic way.

For the FIFO communication policy, we observed that neither the original nor the enhanced protocols satisfy termination. The reason is that perfect FIFO communication never looses messages and introduction of tire-outs created new deadlocks as some of the retransmission rules already exceeded their tire-outs but an (old) message at the front of the queue (that some of the retransmission rules could possibly receive before resending another message) is blocking the queue. We failed to observe this fact in the conference version [23] of this article as we were modelling all variants of the protocol manually. The automatization of the verification process allowed us to discover this termination issue with FIFO; in practice this problem can be resolved by timestamping messages and disregarding the ones with a timestamp above a given bound.

## 9 Conclusion and future work

We have described a tool chain for automatic generation and verification of formal UPPAAL models from communication protocol descriptions. We have applied the tool to BAwCC and BAwPC; two nontrivial protocols from the WS-BA specification. The UPPAAL model is generated from the state-transition tables provided in the WS-BA specification. The tool includes several communication medium models, starting with perfect FIFO channels and ending up with a lossy, duplicating and reordering medium. We have verified that the protocols may enter invalid states for all communication policies apart from the perfect FIFO. For FIFO we verified that no invalid states are reachable for up to six messages in transit (three in each direction), however, this is not a guarantee that the protocol is correct for any size of the FIFO buffer.

Based on the analysis of the protocols in UPPAAL, we suggested enhanced versions of the BAwCC and BAwPC protocols where we distinguish among three different ways of entering the ended states. The enhanced BAwPC protocol is correct for all communication media we consider, however, the enhanced BAwCC protocol is correct only for all

**Table 1** Overview of verification results for BAwCC and enhanced BAwCC

| Buffer type | Properties | BAwCC protocol | | BAwPC protocol | |
|---|---|---|---|---|---|
| | | Original | Enhanced | Original | Enhanced |
| SET | Correctness | No | No | No | Yes |
| | Boundedness | Yes | Yes | Yes | Yes |
| | Termination | No | No | No | Yes |
| BAG | Correctness | No | No | No | Yes |
| | Boundedness | No | No | No | No |
| | Termination | No | No | No | Yes |
| STUTT-FIFO | Correctness | No | Yes | No | Yes |
| | Boundedness | No | Yes | No | Yes |
| | Termination | No | Yes | No | Yes |
| LOSSY-FIFO | Correctness | No | Yes | No | Yes |
| | Boundedness | No | No | No | No |
| | Termination | No | Yes | No | Yes |
| FIFO | Correctness | Yes? | Yes | Yes? | Yes |
| | Boundedness | No | No | No | No |
| | Termination | No | No | No | No |

imperfect media based on FIFO, but may still reach invalid states if orderless asynchronous communication is assumed. The problems have been reported to the OASIS body and adjustments of the BAwCC protocol is currently under development.

By introducing timing constraints (tire-outs) to the protocol behaviour, we were also able to verify termination. Table 1 summarizes results for all five communication policies and the original and enhanced protocols. Correctness stands for the absence of invalid states in protocol executions, boundedness defines whether the communication channels have bounded size and termination guarantees that during any protocol behaviour, all parties eventually reach their final (ended) states. The claim that a certain communication medium is unbounded can be automatically verified only up to a certain (constant) capacity of the channels. However, the generated traces causing an overflow can be (in a manual way) easily extended to demonstrate that the media are not bounded for any given size of the communication buffers.

To conclude, the BAwCC and BAwPC protocols seem correct for the perfect FIFO communication as provided, e.g. by the FTP of TCP/IP. We assume that the protocols were also mainly tested in this setting and hence the tests did not discover any problematic behaviour. On the other hand, the protocols contain a number of message retransmissions, which would not be necessary for the perfect medium. This signals that the designers planned to extend the applicability of the protocols also to frameworks with unreliable communication but as we demonstrated, some fixes have to be applied to the protocols to guarantee the correct operations also in this case. The WS-BA specification is not explicit about the assumptions on the communication medium, but this should be perhaps considered for the future designs of communication protocols in the web services community.

## Appendix A: Implementation details

We shall now present details about the implementation of communication media discussed in the article. In the implementation we need to provide for each medium three basic operations: `send`, `available`, and `receive`. The operation `send(s)` updates the communication medium with a message `s` sent by some role of the protocol, `available(r)` returns a boolean value indicating the availability of the message `r` in the medium (but does not change its content) and `receive(r)` receives the message `r` and updates the medium accordingly. We will explain the implementation details for these operators using the C-like constructs supported directly by the tool UPPAAL. We assume a predefined data type `Msgs` enumerating all messages present in the protocol.

### A.1 Medium SET

The SET medium is implemented with a boolean array such that the `send` operation sets the value of the message to true, `available` returns true if the given message exists in the array, and `receive` has no effect as the medium is duplicating and a sent message can be available for multiple resubmission.

```
bool msg_SET[Msgs];

void send(Msgs s) { msg_SET[s]=true; }

bool available(Msgs r) { return msg_SET[r];}

void receive(Msgs r) { skip }
```

### A.2 Medium BAG

This policy is similar to SET with the exception that for each message we remember the exact number of times it was sent and received. As a consequence, the medium is reordering but not lossy nor duplicating. It is naturally implemented as an array of integers representing the number messages currently present in the medium.

```
const int CAPACITY=4;
int msg_BAG[Msgs];
bool overflow=false;

void send(Msgs s) {
  if (msg_BAG[s]==CAPACITY) overflow=true;
  else msg_BAG[s]++; }

bool available(Msgs r) { return msg_BAG[r]>0; }

void receive(Msgs r) { msg_BAG[r]-; }
```

### A.3 Medium FIFO

Under this policy that represents a standard queue, the operation `send` enqueues a message at the end of the queue, `available` checks whether the required message is at the front of the queue and `receive` removes the message from the queue.

```
const int CAPACITY=4;
int bufferSize=0;
typedef int[0,CAPACITY-1] Buffer;
Msgs msg_FIFO[Buffer];
bool overflow=false;

void send(Msgs s) {
  if (bufferSize==CAPACITY) overflow=true;
  else
   { for (i=bufferSize-1; i>=0; i-)
              msg_FIFO[i+1]=msg_FIFO[i];
   bufferSize++;
   msg_FIFO[0]=s; } }

bool available(Msgs r) {
  if (bufferSize==0) return false;
  else return msg_FIFO[bufferSize-1]==r; }

void receive(Msgs r) { bufferSize-; }
```

### A.4 Medium LOSSY-FIFO

LOSSY-FIFO does not allow reordering and duplication of messages, however, messages can get lost. In this policy the `send` operation adds the message at the end of the queue, `available` checks if the message appears at *any* position in the queue (not necessarily only at its front) and `receive` removes the message from the queue, including all messages that were sent before it.

```
const int CAPACITY=4;
int bufferSize=0;
typedef int[0,CAPACITY-1] Buffer;
Msgs msg_FIFO[Buffer];
bool overflow=false;

void send(Msgs s) {
  if (bufferSize==CAPACITY) overflow=true;
  else
   { for (i=bufferSize-1; i>=0; i-)
              msg_FIFO[i+1]=msg_FIFO[i];
   bufferSize++;
   msg_FIFO[0]=s; }
}

bool available(Msgs r) {
  for (i=bufferSize-1; i>=0; i-)
     if (msg_FIFO[i]==r) return true;
  return false;
}

void receive(Msgs r) {
  while (msg_FIFO[bufferSize-1]!=r)
                        bufferSize-;
  bufferSize-;
}
```

### A.5 Medium STUTT-FIFO

This medium represents an order-preserving but lossy and duplicating communication policy. The `send` operation adds a message to the queue but only if the same message is not already present as the most recently sent one. The operation `available` checks if the message exists at any position in the queue and `receive` removes (losses) messages from the front of the queue until the required message is reached, however, this message remains in the queue to allow for duplication.

```
  const int CAPACITY=4;
int bufferSize=0;
typedef int[0,CAPACITY-1] Buffer;
Msgs msg_FIFO[Buffer];
bool overflow=false;

void send(Msgs s) {
  if (bufferSize==CAPACITY) overflow=true;
  else
```

```
    if (msg_FIFO[0]!=s and bufferSize>0) {

           for (i=bufferSize-1; i>=0; i-)
              msg_FIFO[i+1]=msg_FIFO[i];
         bufferSize++;
         msg_FIFO[0]=s; }
    if (bufferSize==0) {
         bufferSize++;
         msg_FIFO[0]=s; }
}
bool available(Msgs r) {
  for (i=bufferSize-1; i>=0; i-)
         if (msg_FIFO[i]==r) return true;
  return false;
}
```

```
  void receive(Msgs r) {
  while (msg_FIFO[bufferSize-1]!=r)
      bufferSize-;
}
```

## Appendix B: State-Transition Tables

The second part of the appendix contains state-transition tables of the enhanced protocols discussed in the article. In the tables some of the states with similar behaviour are listed in a single column and the *asterisks* indicate the fact that the state after performing the corresponding action is not changed.

Enhanced Business Agreement With Coordination Completion protocol (Participant View)

| Inbound Events | States: Active | Canceling | Completing | Completed | Closing | Compensating | Failing (Active, Canceling, Completing) |
|---|---|---|---|---|---|---|---|
| Cancel | Canceling | *Ignore*<br>Canceling | Canceling | *Resend Completed*<br>Completed | *Ignore*<br>Closing | *Ignore*<br>Compensating | *Resend Fail*<br>Failing-* |
| Complete | Completing | *Ignore*<br>Canceling | *Ignore*<br>Completing | *Resend Completed*<br>Completed | *Ignore*<br>Closing | *Ignore*<br>Compensating | *Resend Fail*<br>Failing-* |
| Close | *Invalid State*<br>Active | *Invalid State*<br>Canceling | *Invalid State*<br>Completing | Closing | *Ignore*<br>Closing | *Invalid State*<br>Compensating | *Invalid State*<br>Failing-* |
| Compensate | *Invalid State*<br>Active | *Invalid State*<br>Canceling | *Invalid State*<br>Completing | Compensating | *Invalid State*<br>Closing | *Ignore*<br>Compensating | *Invalid State*<br>Failing-* |
| Failed | *Invalid State*<br>Active | *Invalid State*<br>Canceling | *Invalid State*<br>Completing | *Invalid State*<br>Completed | *Invalid State*<br>Closing | *Invalid State*<br>Compensating | *Forget*<br>Ended |
| Exited | *Invalid State*<br>Active | *Invalid State*<br>Canceling | *Invalid State*<br>Completing | *Invalid State*<br>Completed | *Invalid State*<br>Closing | *Invalid State*<br>Compensating | *Invalid State*<br>Failing-* |
| NotCompleted | *Invalid State*<br>Active | *Invalid State*<br>Canceling | *Invalid State*<br>Completing | *Invalid State*<br>Completed | *Invalid State*<br>Closing | *Invalid State*<br>Compensating | *Invalid State*<br>Failing-* |

Enhanced Business Agreement With Coordination Completion protocol (Participant View)

| Inbound Events | States: Failing (Compesating) | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
|---|---|---|---|---|---|---|---|
| Cancel | *Ignore*<br>Failing-Compensating | *Resend CannotComplete*<br>NotCompleting | *Resend Exit*<br>Exiting | *Send Canceled*<br>Ended-Canceled | *Ignore*<br>Ended-Closed | *Ignore*<br>Ended-Compensated | *Ignore*<br>Ended |
| Complete | *Ignore*<br>Failing-Compensating | *Resend CannotComplete*<br>NotCompleting | *Resend Exit*<br>Exiting | *Send Fail*<br>Ended-Canceled | *Send Fail*<br>Ended-Closed | *Send Fail*<br>Ended-Compensated | *Ignore*<br>Ended |
| Close | *Invalid State*<br>Failing-Compensating | *Invalid State*<br>NotCompleting | *Invalid State*<br>Exiting | *Ignore*<br>Ended-Canceled | *Send Closed*<br>Ended-Closed | *Ignore*<br>Ended-Compensated | *Ignore*<br>Ended |
| Compensate | *Resend Fail*<br>Failing-Compensating | *Invalid State*<br>NotCompleting | *Invalid State*<br>Exiting | *Ignore*<br>Ended-Canceled | *Ignore*<br>Ended-Closed | *Send Compensated*<br>Ended-Compensated | *Ignore*<br>Ended |
| Failed | *Forget*<br>Ended | *Invalid State*<br>NotCompleting | *Invalid State*<br>Exiting | *Ignore*<br>Ended-Canceled | *Ignore*<br>Ended-Closed | *Ignore*<br>Ended-Compensated | *Ignore*<br>Ended |
| Exited | *Invalid State*<br>Failing-Compensating | *Invalid Invalid*<br>NotCompleting | *Forget*<br>Ended | *Ignore*<br>Ended-Canceled | *Ignore*<br>Ended-Closed | *Ignore*<br>Ended-Compensated | *Ignore*<br>Ended |
| NotCompleted | *Invalid State*<br>Failing-Compensating | *Forget* | *Invalid State*<br>Exiting | *Ignore*<br>Ended-Canceled | *Ignore*<br>Ended-Closed | *Ignore*<br>Ended-Compensated | *Ignore*<br>Ended |

Enhanced Business Agreement With Coordination Completion protocol (Participant View)

| Outbound Events | States | | | | | | |
|---|---|---|---|---|---|---|---|
| | Active | Canceling | Completing | Completed | Closing | Compensating | Failing (Active, Canceling, Completing, Compensating) |
| Exit | | *Invalid State* | | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Exiting | Canceling | Exiting | Completed | Closing | Compensating | Failing-* |
| Completed | *Invalid State* | *Invalid State* | | | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Canceling | Completed | Completed | Closing | Compensating | Failing-* |
| Fail | | | | *Invalid State* | *Invalid State* | | |
| | Failing-Active | Failing-Canceling | Failing Completing | Completed | Closing | Failing-Compensating | Failing-* |
| CannotComplete | | *Invalid State* | | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | NotCompleting | Canceling | NotCompleting | Completed | Closing | Compensating | Failing-* |
| Canceled | *Invalid State* | *Forget* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Ended-Canceled | Completing | Completed | Closing | Compensating | Failing-* |
| Closed | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Forget* | *Invalid State* | *Invalid State* |
| | Active | Canceling | Completing | Completed | Ended-Closed | Compensating | Failing-* |
| Compensated | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Forget* | *Invalid State* |
| | Active | Canceling | Completing | Completed | Closing | Ended-Compensated | Failing-* |

Enhanced Business Agreement With Coordination Completion protocol (Participant View)

| Outbound Events | States | | | | | |
|---|---|---|---|---|---|---|
| | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Exit | *Invalid State* | Exiting | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | NotCompleting | | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Completed | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Fail | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| CannotComplete | | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Canceled | *Invalid State* | *Invalid State* | | *Invalid State* | *Invalid State* | *Invalid State* |
| | NotComleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Closed | *Invalid State* | *Invalid State* | *Invalid State* | | *Invalid State* | *Invalid State* |
| | NotComleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Compensated | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | | *Invalid State* |
| | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |

Enhanced Business Agreement With Coordination Completion protocol (Coordinator View)

| Inbound | States | | | | | | |
|---|---|---|---|---|---|---|---|
| Events | Active | Canceling (Active) | Canceling (Completing) | Completing | Completed | Closing | Compensating |
| Exit | | | | | *Invalid State* | *Invalid State* | *Invalid State* |
| | Exiting | Exiting | Exiting | Exiting | Completed | Closing | Compensating |
| Completed | *Invalid State* | *Invalid State* | | | *Ignore* | *Resend Close* | *Resend Compensated* |
| | Active | Canceling-Active | Completed | Completed | Completed | Closing | Compensating |
| Fail | | | | | *Invalid State* | *Invalid State* | |
| | Failing-Active | Failing-Canceling | Failing-Canceling | Failing-Completing | Completed | Closing | Failing-Compensating |
| CannotComplete | | | | | *Invalid State* | *Invalid State* | *Invalid State* |
| | NotCompleting | NotCompleting | NotCompleting | NotCompleting | Completed | Closing | Compensating |
| Canceled | *Invalid State* | *Forget* | *Forget* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Ended | Ended | Completing | Completed | Closing | Compensating |
| Closed | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Forget* | *Invalid State* |
| | Active | Canceling-Active | Canceling-Completing | Completing | Completed | Ended | Compensating |
| Compensated | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Forget* |
| | Active | Canceling-Active | Canceling-Completing | Completing | Completed | Closing | Ended |

Enhanced Business Agreement With Coordination Completion protocol (Coordinator View)

| Inbound | States | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Events | Failing (Active, Canceling, Completing) | Failing (Compesating) | NotCompleting | Exiting | Ended-Failed | Ended-Exited | EndedNot-Completed | Ended |
| Exit | *Invalid State* | *Invalid State* | *Invalid State* | *Ignore* | *Ignore* | *Resend Exited* | *Ignore* | *Ignore* |
| | Failing-* | Failing-Compensating | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Completed | *Invalid State* | *Ignore* | *Invalid State* | *Invalid State* | *Ignore* | *Ignore* | *Ignore* | *Ignore* |
| | Failing-* | Failing-Compensating | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Fail | *Ignore* | *Ignore* | *Invalid State* | *Invalid State* | *Resend Failed* | *Ignore* | *Ignore* | *Ignore* |
| | Failing-* | Failing-Compensating | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| CannotComplete | *Invalid State* | *Invalid State* | *Ignore* | *Invalid State* | *Ignore* | *Ignore* | *Resend NotCompleted* | *Ignore* |
| | Failing-* | Failing-Compensating | NotComppleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Canceled | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Ignore* | *Ignore* | *Ignore* | *Ignore* |
| | Failing-* | Failing-Compensating | NotComppleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Closed | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Ignore* | *Ignore* | *Ignore* | *Ignore* |
| | Failing-* | Failing-Compensating | NotComppleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Compensated | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Ignore* | *Ignore* | *Ignore* | *Ignore* |
| | Failing-* | Failing-Compensating | NotComppleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |

Enhanced Business Agreement With Coordination Completion protocol (Coordinator View)

| Outbound | States | | | | | | |
|---|---|---|---|---|---|---|---|
| Events | Active | Canceling (Active, Completing) | Completing | Completed | Closing | Compensating | Failing (Active, Canceling, Completing, Compensating) |
| Cancel | | | | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Canceling-Active | Canceling-* | Canceling-Completing | Completed | Closing | Compensating | Failing-* |
| Complete | | *Invalid State* | | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Completing | Canceling-* | Completing | Completed | Closing | Compensating | Failing-* |
| Close | *Invalid State* | *Invalid State* | *Invalid State* | | Closing | *Invalid State* | *Invalid State* |
| | Active | Canceling-* | Completing | Closing | Closing | Compensating | Failing-* |
| Compensate | *Invalid State* | *Invalid State* | *Invalid State* | | *Invalid State* | | *Invalid State* |
| | Active | Canceling-* | Completing | Compensating | Closing | Compensating | Failing-* |
| Failed | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Forget* |
| | Active | Canceling-* | Completing | Completed | Closing | Compensating | Ended-Failed |
| Exited | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Canceling-* | Completing | Completed | Closing | Compensating | Failing-* |
| NotCompleted | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Canceling-* | Completing | Completed | Closing | Compensating | Failing-* |

Enhanced Business Agreement With Coordination Completion protocol (Coordinator View)

| Outbound Events | States | | | | | |
|---|---|---|---|---|---|---|
| Events | NotCompleting | Exiting | Ended-Failed | Ended-Exited | EndedNot-Completed | Ended |
| Cancel | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Complete | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Close | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Compensate | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Failed | *Invalid State* | *Invalid State* | | *Invalid State* | *Invalid State* | *Invalid State* |
| | NotComleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Exited | *Invalid State* | *Forget* | *Invalid State* | | *Invalid State* | *Invalid State* |
| | NotComleting | Ended-Exited | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| NotCompleted | *Forget* | *Invalid State* | *Invalid State* | *Invalid State* | | *Invalid State* |
| | Ended-NotCompleted | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |

Enhanced Business Agreement With Participant Completion protocol (Participant View)

| Inbound Events | States | | | | | | |
|---|---|---|---|---|---|---|---|
| | Active | Canceling | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
| Cancel | | *Ignore* | *Resend Completed* | *Ignore* | *Ignore* | *Resend Fail* | *Resend Fail* |
| | Canceling | Canceling | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
| Close | *Invalid State* | *Invalid State* | | *Ignore* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Canceling | Closing | Closing | Compensating | Failing-Active | Failing-Canceling |
| Compensate | *Invalid State* | *Invalid State* | | *Invalid State* | *Ignore* | *Invalid State* | *Invalid State* |
| | Active | Canceling | Compensating | Closing | Compensating | Failing-Active | Failing-Canceling |
| Failed | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Forget* | *Forget* |
| | Active | Canceling | Completed | Closing | Compensating | Ended | Ended |
| Exited | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Canceling | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
| NotCompleted | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Canceling | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |

Enhanced Business Agreement With Participant Completion protocol (Participant View)

| Inbound Events | States | | | | | | |
|---|---|---|---|---|---|---|---|
| | Failing-Compesating | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Cancel | *Ignore* | *Resend CannotComplete* | *Resend Exit* | *Send Calceled* | *Ignore* | *Ignore* | *Ignore* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Close | *Invalid State* | *Invalid State* | *Invalid State* | *Ignore* | *Send Closed* | *Ignore* | *Ignore* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Compensate | *Resend Fail* | *Invalid State* | *Invalid State* | *Ignore* | *Ignore* | *Send Compensated* | *Ignore* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Failed | *Forget* | *Invalid State* | *Invalid State* | *Ignore* | *Ignore* | *Ignore* | *Ignore* |
| | Ended | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Exited | *Invalid State* | *Invalid State* | *Forget* | *Ignore* | *Ignore* | *Ignore* | *Ignore* |
| | Failing-Compensating | NotCompleting | Ended | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| NotCompleted | *Invalid State* | *Forget* | *Invalid State* | *Ignore* | *Ignore* | *Ignore* | *Ignore* |
| | Failing-Compensating | Ended | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |

Enhanced Business Agreement With Participant Completion protocol (Participant View)

| Outbound | States | | | | | | |
|---|---|---|---|---|---|---|---|
| Events | Active | Canceling | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
| Exit | | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Exiting | Canceling | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
| Completed | | *Invalid State* | | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Completed | Canceling | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
| Fail | | | *Invalid State* | *Invalid State* | | | |
| | Failing-Active | Failing-Canceling | Completed | Closing | Failing-Compensating | Failing-Active | Failing-Canceling |
| CannotComplete | | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | NotCompleting | Canceling | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
| Canceled | *Invalid State* | *Forget* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Ended-Canceled | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
| Closed | *Invalid State* | *Invalid State* | *Invalid State* | *Forget* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Canceling | Completed | Ended-Closed | Compensating | Failing-Active | Failing-Canceling |
| Compensated | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Forget* | *Invalid State* | *Invalid State* |
| | Active | Canceling | Completed | Closing | Ended-Compensated2 | Failing-Active | Failing-Canceling |

Enhanced Business Agreement With Participant Completion protocol (Participant View)

| Outbound | States | | | | | | |
|---|---|---|---|---|---|---|---|
| Events | Failing-Compesating | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Exit | *Invalid State* | *Invalid State* | | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Completed | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Fail | | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| CannotComplete | *Invalid State* | | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Canceled | *Invalid State* | *Invalid State* | *Invalid State* | | *Invalid State* | *Invalid State* | *Invalid State* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compesated | Ended |
| Closed | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | | *Invalid State* | *Invalid State* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |
| Compensated | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | | *Invalid State* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Canceled | Ended-Closed | Ended-Compensated | Ended |

Enhanced Business Agreement With Participant Completion protocol (Coordinator View)

| Inbound Events | States: Active | Canceling | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
|---|---|---|---|---|---|---|---|
| Exit | | | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Exiting | Exiting | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
| Completed | | | *Ignore* | *Resend Close* | *Resend Compensate* | *Invalid State* | *Ignore State* |
| | Completed | Completed | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
| Fail | | | *Invalid State* | *Invalid State* | | *Ignore* | *Ignore* |
| | Failing-Active | Failing-Canceling | Completed | Closing | Failing-Compensating | Failing-Active | Failing-Canceling |
| CannotComplete | | | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | NotCompleting | NotCompleting | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
| Canceled | *Invalid State* | *Forget* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Ended | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
| Closed | *Invalid State* | *Invalid State* | *Invalid State* | *Forget* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Canceling | Completed | Ended | Compensating | Failing-Active | Failing-Canceling |
| Compensated | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Forget* | *Invalid State* | *Invalid State* |
| | Active | Canceling | Completed | Closing | Ended | Failing-Active | Failing-Canceling |

Enhanced Business Agreement With Participant Completion protocol (Coordinator View)

| Inbound Events | States: Failing-Compesating | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
|---|---|---|---|---|---|---|---|
| Exit | *Invalid State* | *Invalid State* | *Ignore* | *Ignore* | *Resend Exited* | *Ignore* | *Ignore* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Completed | *Ignore* | *Invalid State* | *Invalid State* | *Ignore* | *Ignore* | *Ignore* | *Ignore* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Fail | *Ignore* | *Invalid State* | *Invalid State* | *Resend Failed* | *Ignore* | *Ignore* | *Ignore* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| CannotComplete | *Invalid State* | *Ignore* | *Invalid State* | *Ignore* | *Ignore* | *Resend NotCompleted* | *Ignore* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Exited | Ended-Failed | Ended-NotCompleted | Ended |
| Canceled | *Invalid State* | *Invalid State* | *Invalid State* | *Ignore* | *Ignore* | *Ignore* | *Ignore* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Closed | *Invalid State* | *Invalid State* | *Invalid State* | *Ignore* | *Ignore* | *Ignore* | *Ignore* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Compensated | *Invalid State* | *Invalid State* | *Invalid State* | *Ignore* | *Ignore* | *Ignore* | *Ignore* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |

Enhanced Business Agreement With Participant Completion protocol (Coordinator View)

| Outbound Events | States Active | Canceling | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
|---|---|---|---|---|---|---|---|
| Cancel | | | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Canceling | Canceling | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
| Close | *Invalid State* | *Invalid State* | | | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Canceling | Closing | Closing | Compensating | Failing-Active | Failing-Canceling |
| Compensate | *Invalid State* | *Invalid State* | | *Invalid State* | | *Invalid State* | *Invalid State* |
| | Active | Canceling | Compensating | Closing | Compensating | Failing-Active | Failing-Canceling |
| Failed | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Forget* | *Forget* |
| | Active | Canceling | Completed | Closing | Compensating | Ended-Failed | Ended-Failed |
| Exited | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Canceling | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |
| NotCompleted | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Active | Canceling | Completed | Closing | Compensating | Failing-Active | Failing-Canceling |

Enhanced Business Agreement With Participant Completion protocol (Coordinator View)

| Outbound Events | States Failing-Compesating | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
|---|---|---|---|---|---|---|---|
| Cancel | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Close | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Compensate | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* | *Invalid State* |
| | Failing-Compensating | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Failed | *Forget* | *Invalid State* | *Invalid State* | | *Invalid State* | *Invalid State* | *Invalid State* |
| | Ended-Failed | NotCompleting | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| Exited | *Invalid State* | *Invalid State* | *Forget* | *Invalid State* | | *Invalid State* | *Invalid State* |
| | Failing-Compensating | NotCompleting | Ended-Exited | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |
| NotCompleted | *Invalid State* | *Forget* | *Invalid State* | *Invalid State* | *Invalid State* | | *Invalid State* |
| | Failing-Compensating | Ended-NotCompelted | Exiting | Ended-Failed | Ended-Exited | Ended-NotCompleted | Ended |

## References


1. Afek, Y., Attiya, H., Fekete, A., Fischer, M., Lynch, N., Mansour, Y., Wang, D.-W., Zuck, L.: Reliable communication over unreliable channels. J. ACM **41**(6), 1267–1297 (1994)
2. Apt, K.R., Francez, N., Katz, S.: Appraising fairness in languages for distributed programming. Distrib. Comput. **2**, 226–241 (1988)
3. Barghouti, N., Nounou, N., Yemini, Y.: An integrated protocol development environment. In: Protocol Specification Testing and Verification VI, pp. 63–69. North-Holland, Amsterdam (1987)
4. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04). LNCS, vol. 3185, pp. 200–236. Springer, Berlin (2004)
5. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983)
6. Finkel, A.: Decidability of the termination problem for completely specified protocols. Distrib. Comput. **7**, 129–135 (1994)
7. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, Menlo Park (1993)
8. Greenfield, P., Kuo, D., Nepal, S., Fekete, A.: Consistency for web services applications. In: VLDB '05: proceedings of the 31st international conference on very large data bases, pp. 1199–1203. VLDB Endowment (2005)
9. Johnson, J.E., Langworthy, D.E., Lamport, L., Vogt, F.H.: Formal specification of a web services protocol. J. Logic. Algebraic Program. **70**(1), 34–52 (2007)
10. Lamport, L.: Specifying Systems. Addison-Wesley, Reading (2003)
11. Lohmann, N.: Communication models for services. In: Proceedings of ZEUS'10, vol. 563 of CEUR Workshop Proceedings, pp. 9–16. CEUR-WS.org (2010)
12. Marques, A.P. Jr., Ravn, A.P., Srba, J., Vighio, S.: Tool supported analysis of web services protocols. In: Proceedings of the 5th International Workshop of Harnessing Theories for Tool Support in Software (TTSS'11), pp. 50–64 (2011)
13. Marques, A.P., Ravn, A.P., Srba, J., Vighio, S.: The tool csv2uppaal. http://csv2uppaal.github.com/csv2uppaal/. Accessed 6 April 2012
14. Mathew, B., Juric, M., Sarang, P.: Business Process Execution Language for Web Services, 2nd edn. Packt Publishing, Birmingham (2006)
15. Naumovich, G.N., Clarke, L.A., Osterweil, L.J.: Verification of communication protocols using data flow analysis. SIGSOFT Softw. Eng. Notes **21**, 93–105 (1996)
16. Newcomer, E., Robinson, I. (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. Accessed 6 April 2012
17. Newcomer, E., Robinson, I. (chairs): Web services business activity (WS-businessactivity) version 1.2 (2009). http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os/wstx-wsba-1.2-spec-os.html. Accessed 6 April 2012
18. Newcomer, E., Robinson, I. (chairs): Web services coordination (WS-coordination) version 1.2 (2009). http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os/wstx-wscoor-1.2-spec-os.html. Accessed 6 April 2012
19. Nicollin, X., Sifakis, J.: The algebra of timed processes, ATP: theory and application. Inf. Comput. **114**(1), 131–178 (1994)
20. OASIS. Discussion forum, report on error trace in BAwCC (2011). http://markmail.org/message/xgnyonkihfif5vz2. Accessed 6 April 2012
21. Ravn, A.P., Srba, J., Vighio, S.: UPPAAL model of the WS-BA protocol. http://www.uppaal.org. Accessed 6 April 2012
22. Ravn, A.P., Srba, J., Vighio, S.: A formal analysis of the web services atomic transaction protocol with UPPAAL. In: Proceedings of the 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'10). LNCS, vol. 6416, pp. 579–593. Springer, Berlin (2010)
23. Ravn, A.P., Srba, J., Vighio, S.: Modelling and verification of web services business activity protocol. In: Abdulla, P.A., Leino, K.R.M. (eds) Proceedings of TACAS'11. LNCS, vol. 6605, pp. 357–371. Springer, Berlin (2011)
24. Robinson, I.: Answer in WS-BA discussion forum, July 14th (2010). http://markmail.org/message/wriewgkboaaxw66z. Accessed 6 April 2012
25. Schnoebelen, Ph.: Verifying lossy channel systems has nonprimitive recursive complexity. Inf. Process. Lett. **83**, 251–261 (2002)
26. UPPAAL. http://www.uppaal.com. Accessed 6 April 2012
27. Vogt, F.H., Zambrovski, S., Gruschko, B., Furniss, P., Green, A: Implementing web service protocols in SOA: WS-coordination and WS-businessactivity. In: Proceedings of the Seventh IEEE International Conference on E-Commerce Technology Workshops(CECW'05), pp. 21–28. IEEE Computer Society, New york (2005)
28. Vuong, S.T., Hui, D.D., Cowan, D.D.: Valira—a tool for protocol validation via reachability analysis. In: Protocol Specification, Testing and Verification VI, pp. 35–41. North-Holland, Amsterdam (1987)