Modelling and Verification of Web Services Business Activity Protocol

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. WS-Business Activity specification defines two coordination protocols in order to ensure a consistent agreement on the outcome of long-running distributed applications. We use the model checker UPPAAL to analyse the Business Agreement with Coordination Completion protocol type. Our analyses show that the protocol, as described in the standard specification, violates correct operation by reaching invalid states for all underlying communication media except for the perfect FIFO. Based on this result, we propose changes to the protocol. A further investigation of the modified protocol suggests that messages should be received in the same order as they are sent so that a correct protocol behaviour is preserved. Another important property of communication protocols is that all parties always reach their final states. Based on the verification with different communication models, we prove that our enhanced protocol satisfies this property for asynchronous, unreliable, order-preserving communication whereas the original protocol does not.

1 Introduction

Numerous protocols from the web services protocol stack [9] are currently in active development in order 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 their standards [7, 17]. 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 [3] and TLC [7] can be applied to analyse protocol behaviours and verify general correctness criteria.

In this paper we consider the WS-Coordination framework [12] which, among others, includes the WS-Atomic Transaction (WS-AT) [10] and WS-Business

* The author is partially supported by the Ministry of Education of Czech Republic, grant no. MSM 0021622419.

** The author is supported by Quaid-e-Awam University of Engineering, Science, and Technology, Nawabshah, Pakistan, and partially by the Nordunet3 project COSoDIS.

Activity (WS-BA) [11] standards. The WS-AT specification provides 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 [7] where the protocol was formalized in the TLA$^+$ [8] language as well as using the UPPAAL tool and networks of communicating timed automata [15]. In [15], 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 which (to the best of our knowledge) has not yet been automatically verified in the literature. It consists of two coordination protocols: Business Agreement with Participant Completion (BAwPC) and Business Agreement with Coordinator Completion (BAwCC). We focus on BAwCC in our analysis. It is more complex in its behaviour and has a larger number of states, transitions and messages than BAwPC. We develop several UPPAAL [3] models related to the WS-BA protocols based on the state-tables provided in the standard specification (see [11] or the appendix for the complete tables). We use with advantage the C-like constructs available in UPPAAL and the model of the BAwCC protocol contains more than 600 lines of C code. Our tool supported analysis 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 UPPAAL, we suggest fixes to the protocol. Moreover, in contrast to [7, 15], we do not limit our analyses to only one type of asynchronous communication policy where messages can be reordered, lost and duplicated, but study different communication mechanisms (see e.g. [1]). This fact appears crucial as even the fixed protocol behaves correctly only for some types of communication media, whereas for others it still violates the correctness criteria.

Another important property of web services applications is that they should terminate in consistent end states, irrelevant of the actual behaviour of the other participating parties [6]. 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 [13]) 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 under suitable timing constraints used for tire-outs, we can guarantee the termination property for the fixed protocol, at least for the communication policies where the protocol is correct.

The rest of the paper is organized as follows. In Section 2, we give an overview of the web services business activity protocol and discuss different types of com-

munication policies. Section 3 introduces the UPPAAL modeling approach used in the case study. Properties of the original and the fixed protocols are discussed in Sections 4 and 5. Section 6 describes the termination property and its verification. Finally, Section 7 gives a summary and suggestions for the future research. The appendix contains a full overview of the state-transition tables of the original and modified BAwCC protocol.

2 WS-Business Activity Protocol

WS-Business Activity (WS-BA) [11] and WS-Atomic Transaction (WS-AT) [10] both built on top of WS-Coordination specification [12] 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 [5] for simple short-lived activities, whereas WS-BA provides protocols used for long-lived business activities with relaxation of ACID properties.

WS-BA [11] 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-Business Agreement with Participant Completion (BAwPC) and WS-Business Agreement with Coordination Completion (BAwCC) that we shall focus on. A participant registers for one these two protocols, which are managed by the coordinator of the activity.

2.1 Business Agreement with Coordination Completion

A state-transition diagram for BAwCC is shown in Figure 1. Note that the figure depicts a combined view and the concrete coordinator and participant states are abstracted away. The complete transition tables are listed in the appendix.

A participant registered for this protocol 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 new requests within the current business activity and it is time to complete the processing. The Complete message is followed by the Completed message by the participant, provided it can successfully finish its work. This protocol also introduces a new 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

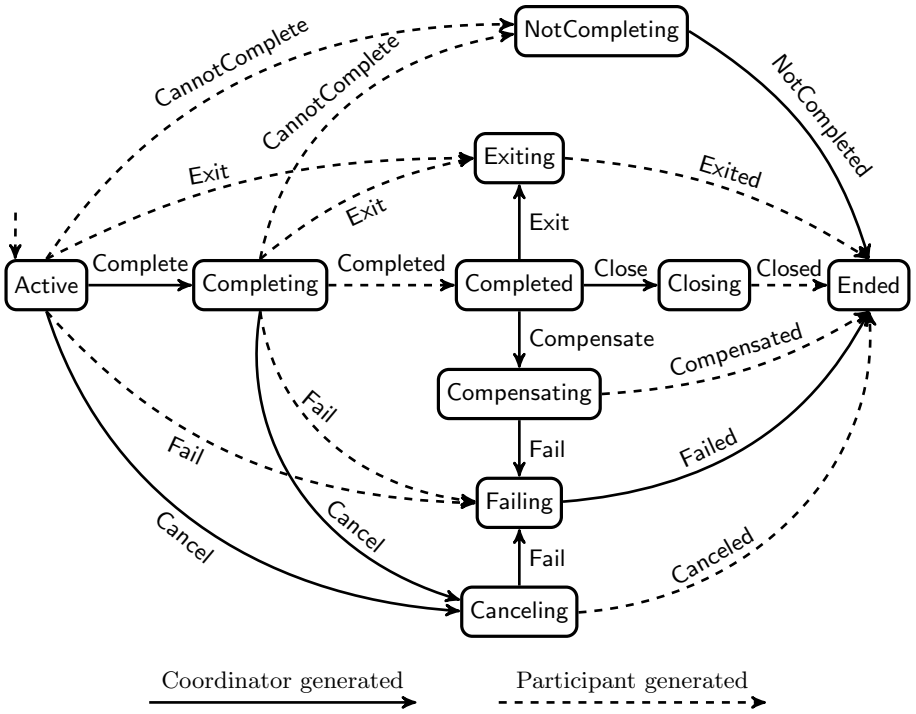

Fig. 1. Business Agreement with Coordinator Completion

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 and to restore the recorded data to its initial state. A participant in response can either send a Compensated or a 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 can not 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 and 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 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 [7] 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 protocol is 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.

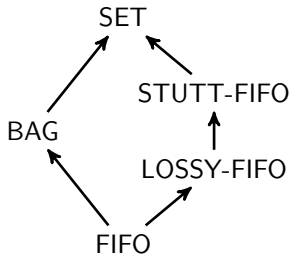

Fig. 2. Communication media

- *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 [7], 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 [7, 15] for 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.
- *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.* In order 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.

Figure 2 shows the relationship 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 introduce the validity of any safety property for e.g. the SET medium, this result will hold also for any other medium below it and finding an error trace in the protocol with e.g. the FIFO medium implies the presence of such a trace also in any other 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 industrial applications that we are aware of. Although, as remarked above, FTP will work this way if the application level avoids 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 otherwise impossible as the size of such channels is not bounded in our setting.

3 Formal Modelling of BAwCC in UPPAAL

The WS-BA standard [11] provides a high-level description of the WSwCC protocol. It is essentially a collection of protocol behaviours described in English accompanied by diagrams like the graph shown in Figure 1 and state-transition tables for the parties involved in the protocol. See Figure 3 a) for a fragment of such a table and the appendix for a complete collection of the tables.

Figure 3 a) describes how the transaction coordinator, being in its internal state Closing, handles the message Complete arriving from the participant. It will simply resend to the participant the message Close and remain in the state Closing. The table also 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 behaviour is given in Figure 3 b). The syntax should be readable even without any prior knowledge of the tool, but we refer the interested reader to [3] 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_Msg` and `Receive_Msg` that take care of sending and receiving of messages via the bit-vectors `msgTC` and `msgP`. The code is shown only for the simplest SET implementation. For BAG, FIFO, LOSSY-FIFO and STUTT-FIFO the code is more complex but implemented in a standard way. 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`.

The transitions described in the state tables are then implemented in the expected way as shown by the two examples in Figure 3 b). The final timed automata model then 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.) is performed via 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 more than 600 nonempty lines of code. The complete UPPAAL model can be downloaded at [14].

4 Analysis of BAwCC

As already noted, WS-BA relaxes the ACID principles and allows for a mixed outcome of a transaction. Therefore, we cannot expect that all parties of the protocol agree on the outcome, as it was the case for WS-AT protocols [7, 15].

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

Inbound Events	**States**				
	...	Closing	...	Exiting	...
⋮	⋮	⋮	...	⋮	⋮
Complete	...	resend Close goto Closing	...	...	...
⋮	⋮	⋮	...	⋮	⋮
CannotComplete	...	goto Invalid-State	...	...	...

c) Uppaal encoding with SET Communication (simplified):
Send_Msg() and Receive_Msg() 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_Msg(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 Receive_Msg(MsgP r) {
  return msgP[r]; }
```

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

```
bool guard() {
 return Receive_Msg(COMPLETE_P) && stTC == TC_CLOSING;  }

void action() {  Send_Msg(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 Receive_Msg(CANNOT_COMPLETE_P) && stTC == TC_CLOSING; }
```

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

Instead, we focus on the analysis of 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 WS-

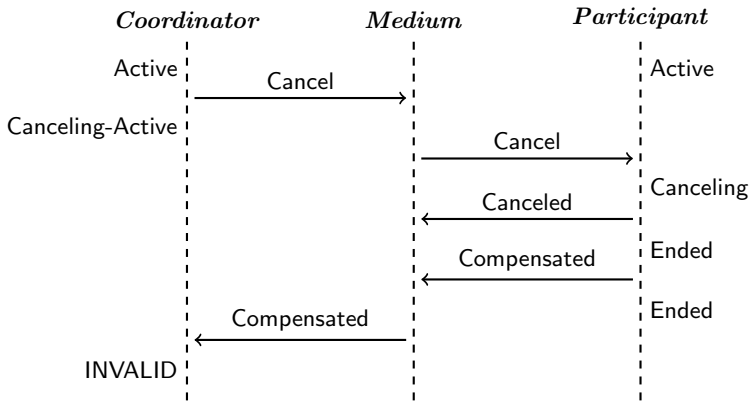

Fig. 4. Error trace in BAwCC leading to an invalid state

BA specification but we contacted the designers via their discussion forum and received (citing [16]):

> "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 BusinessAgreement-WithCoordinationCompletion (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. This requirement is easily formulated in the UPPAAL query language (a subset of TCTL) as follows.

```
E<> (tc.INVALID || par.INVALID) && !overflow
```

This is a safety property asking whether there is a protocol execution in which either the transaction coordinator (process called `tc`) or the participant (process called `par`) enters the state `INVALID` while at the same time there was no buffer overflow. We have checked this property for all five communication policies we consider and the property surprisingly turned out to be true for all of them except for FIFO. The tool automatically generated an error trace, seen in Figure 4. It is easy to see that this trace is executable both for LOSSY-FIFO and BAG communication (and hence also for any other above them in the hierarchy in Figure 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 error 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 [4], there is no hope to establish the correctness of the protocol with unbounded FIFO communication in a fully automatic way.

Another interesting question we can ask about the protocol is whether the communication medium is bounded for BAwCC or not. This can be done by asking the following UPPAAL query.

```
E<> overflow
```

Verification results show that all communication media except for 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.

5 Enhanced BAwCC

Given the verification results in the previous section, we found the BAwCC protocol not completely satisfactory as even a simple relaxation of the perfect communication policy results in incorrect behaviour. Taking into account that the protocols in WS-AT avoided invalid states even under the most general SET communication, we shall further analyze the protocol and suggest an improvement.

The error trace in Figure 4 hints at the source of problems. Once a participant reaches the Ended state, it it 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 Figure 4, 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 side. A similar problem can occur in a symmetric way.

In our proposed fix to the BAwCC protocol, we introduce three additional end states, both for the participant as well as for the coordinator, in order to avoid the confusion. The complete state tables of the enhanced protocol are given in the appendix. We modelled and verified the enhanced protocol in UPPAAL and the results are as follows.

Under the STUTT-FIFO communication, the medium is 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 this is

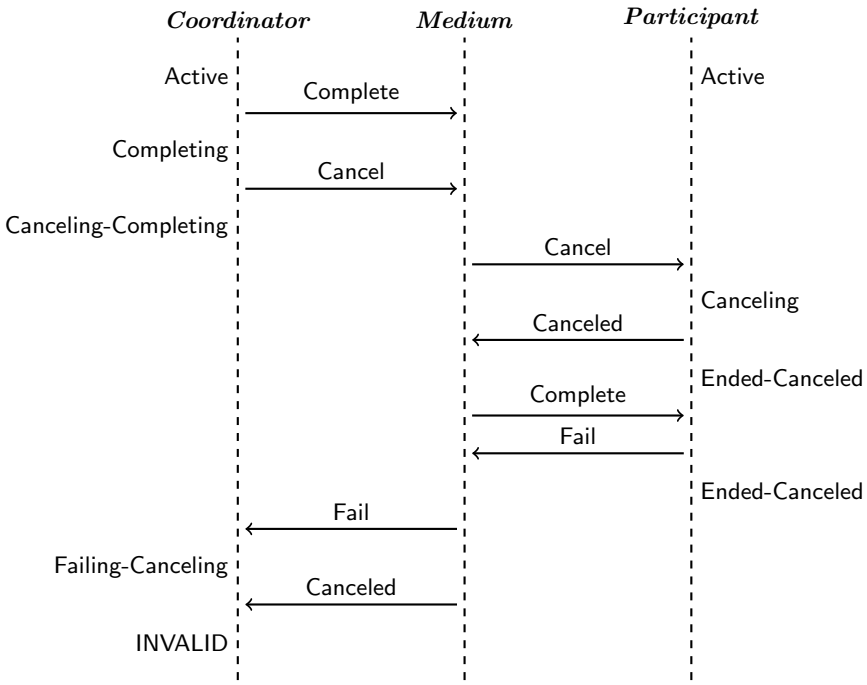

Fig. 5. Error trace in enhanced BAwCC leading to an invalid state

a safety property, the positive result holds automatically also for 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 Figure 5. 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 be 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.

6 Termination under Fairness

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

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

The semantics is that in any maximal computation of the protocol, we 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 fails 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 therefor focus on our enhanced BAwCC protocol and the communication medium STUTT-FIFO where the protocol is correct and the medium bounded. A positive result will imply termination also for LOSSY-FIFO and FIFO.

A quick query about termination in UPPAAL shows that it fails the property 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 is retransmitted over and over. 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 and 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 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 behaviour, 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 the introduction it is essentially the "execution delay" of ATP [13]. 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 any more. Figure 6 shows the 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 can separate 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. The clock x represents the time delay since the last progress transition occurred (it is reset to 0 by any progress transition)

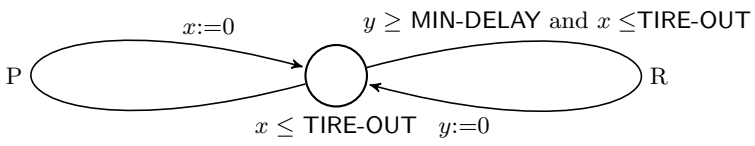

Fig. 6. Tire-outs modelling; P is a progress transition, R is a retransmission transition

and clock y represents the time elapsed since the last retransmission. These two clocks restrict the behaviour of the retransmission transitions so that they are enabled only if at least the minimal delay has passed since last retransmission and the clock x has not exceeded the tire-out limit. The presence of the invariant $x \leq$ TIRE-OUT then ensures a progress.

Using the tire-out modeling as described above we were able to automatically verify that the enhanced BAwCC protocol with the STUTT-FIFO communication policy satisfies 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 the two constants we can experiment with different timing options while making (automatically) sure that the termination property is preserved.

7 Conclusion and Future Work

We provided a formal Uppaal model of the Business Agreement with Coordinator Completion (BAwCC) protocol from the WS-BA specification. The model is based on the state-transition tables provided in the specification. We also introduced several ways to model the communication medium, starting with perfect FIFO channels and ending up with lossy, duplicating and orderless medium. We have verified that the protocol may enter invalid states for all communication policies apart from the 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 protocol in Uppaal, we suggested an enhanced protocol which distinguishes among three different ways of entering the ended states. This protocol is correct also for all imperfect media based on FIFO but may still reach invalid states if more liberal communication is assumed. By introducing timing constraints (tire-outs) to the protocol behaviour, we were also able to verify the termination property for imperfect FIFO communication. Figure 7 gives the summary of the results for all five communication policies and the original and enhanced protocols. Correctness stands for the absence of invalid states in protocol executions, boundedness describes whether the communica-

Buffer Type	Properties	BAwCA Protocol	
		Original	**Enhanced**
SET	Correctness	No	No
	Boundedness	Yes	Yes
	Termination	No	No
BAG	Correctness	No	No
	Boundedness	No	No
	Termination	No	No
STUTT-FIFO	Correctness	No	**Yes**
	Boundedness	No	**Yes**
	Termination	No	**Yes**
LOSSY-FIFO	Correctness	No	**Yes**
	Boundedness	No	No
	Termination	No	**Yes**
FIFO	Correctness	Yes?	**Yes**
	Boundedness	No	No
	Termination	Yes?	**Yes**

Fig. 7. Overview of verification results for BAwCC and enhanced BAwCC

tion channels have bounded size and termination guarantees that during any protocol behaviour, all parties eventually reach their final (ended) states.

To conclude, the BAwCC protocol seems correct for the perfect FIFO communication as provided e.g. by the FTP of TCP/IP. We assume that the protocol was also mainly tested in this setting and hence the tests did not discover any problematic behaviour. On the other hand, the protocol contains 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 protocol also to frameworks with unreliable communication but as we demonstrated, some fixes have to be applied to the protocol in order to guarantee the correct operation also in this case. In any case, WS-BA specification is not explicit about the assumptions on the communication medium, but this should be perhaps considered for the future design of protocols.

Finally, the manual creation of UPPAAL models for WS-BA protocols was a long and time demanding process and in our future work we will try to automate the process of creating timed automata templates directly from the state-transition tables. For widely used, standardized protocols, this is probably not going to find defects. Yet, in concrete implementations some optimizations and specializations may be included, and here a tool support may assist in validating the effect of presumably small innocent changes.

Acknowledgement. The authors are grateful to the anonymous reviewers for their comments on the perspective of this work.

References

1. Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, Dai-Wei. Wang, and L. Zuck. Reliable communication over unreliable channels. *J. ACM*, 41(6):1267–1297, 1994.
2. K.R. Apt, N. Francez, and S. Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988.
3. G. Behrmann, A. David, and K.G. Larsen. 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)*, number 3185 in LNCS, pages 200–236. Springer-Verlag, 2004.
4. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.
5. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
6. Paul Greenfield, Dean Kuo, Surya Nepal, and Alan Fekete. Consistency for web services applications. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1199–1203. VLDB Endowment, 2005.
7. 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.
8. L. Lamport. *Specifying Systems*. Addison-Wesley, 2003.
9. B. Mathew, M. Juric, and P. Sarang. *Business Process Execution Language for Web Services 2nd Edition*. Packt Publishing, 2006.
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. E. Newcomer and I. Robinson (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.
12. E. Newcomer and I. Robinson (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.
13. X. Nicollin and J. Sifakis. The algebra of timed processes, ATP: Theory and application. *Information and Computation*, 114(1):131–178, 1994.
14. A.P. Ravn, J. Srba, and S. Vighio. UPPAAL model of the WS-BA protocol. Available in the UPPAAL example section at http://www.uppaal.org.
15. A.P. Ravn, J. Srba, and S. Vighio. 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)*, volume 6416 of *LNCS*, pages 579–593. Springer-Verlag, 2010.
16. I. Robinson. Answer in WS-BA discussion forum, July 14th, 2010. http://markmail.org/message/wriewgkboaaxw66z.
17. F. H. Vogt, S. Zambrovski, B. Gruschko, P. Furniss, and A. Green. 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)*, pages 21–28. IEEE Computer Society, 2005.

BusinessAgreementWithCoordinationCompletion protocol (Participant View)											
Inbound Events	**States**										
	Active	**Canceling**	**Completing**	**Completed**	**Closing**	**Compensating**	**Failing (Active, Canceling, Completing)**	**Failing (Compesating)**	**NotCompleting**	**Exiting**	**Ended**
Cancel	Canceling	*Ignore* Canceling	Canceling	*Resend Completed* Completed	*Ignore* Closing	*Ignore* Compensating	*Resend Fail* Failing-*	*Ignore* Failing-Compensating	*Resend CannotComplete* NotCompleting	*Resend Exit* Exiting	*Send Canceled* Ended
Complete	Completing	*Ignore* Canceling	*Ignore* Completing	*Resend Completed* Completed	*Ignore* Closing	*Ignore* Compensating	*Resend Fail* Failing-*	*Ignore* Failing-Compensating	*Resend CannotComplete* NotCompleting	*Resend Exit* Exiting	*Send Fail* Ended
Close	*Invalid State* Active	*Invalid State* Canceling	*Invalid State* Completing	Closing	*Ignore* Closing	*Invalid State* Compensating	*Invalid State* Failing-*	*Invalid State* Failing-Compensating	*Invalid State* NotCompleting	*Invalid State* Exiting	*Send Closed* Ended
Compensate	*Invalid State* Active	*Invalid State* Canceling	*Invalid State* Completing	Compensating	*Invalid State* Closing	*Ignore* Compensating	*Invalid State* Failing-*	*Resend Fail* Failing-Compensating	*Invalid State* NotCompleting	*Invalid State* Exiting	*Send Compensated* Ended
Failed	*Invalid State* Active	*Invalid State* Canceling	*Invalid State* Completing	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Forget* Ended	*Forget* Ended	*Invalid State* NotCompleting	*Invalid State* Exiting	*Ignore* Ended
Exited	*Invalid State* Active	*Invalid State* Canceling	*Invalid State* Completing	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-*	*Invalid State* Failing-Compensating	*Invalid State* NotCompleting	*Forget* Ended	*Ignore* Ended
NotCompleted	*Invalid State* Active	*Invalid State* Canceling	*Invalid State* Completing	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-*	*Invalid State* Failing-Compensating	*Forget* Ended	*Invalid State* Exiting	*Ignore* Ended

BusinessAgreementWithCoordinationCompletion protocol (Participant View)

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

BusinessAgreementWithCoordinationCompletion protocol (Coordinator View)

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

BusinessAgreementWithCoordinationCompletion protocol (Coordinator View)

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

Enhanced BusinessAgreementWithCoordinationCompletion protocol (Participant View)

Inbound Events	States											
	Active	**Canceling**	**Completing**	**Closing**	**Compensating**	**Failing (Active, Canceling, Completing)**	**Failing (Compesating)**	**NotCompleting**	**Exiting**	**Ended-Canceled**	**Ended-Closed**	**Ended-Compensated**	**Ended**
Cancel	Canceling	*Ignore*	Canceling	*Ignore*	*Ignore*	*Resend Fail*	*Ignore*	*Resend CannotComplete*	*Resend Exit*	*Send Canceled*	*Ignore*	*Ignore*	*Ignore*
		Canceling		Closing	Compensating	Failing-*	Failing-Compensating	NotCompleting	Exiting	Ended-Canceled	Ended-Closed	Ended-Compensated	Ended
Complete	Completing	*Ignore*	Completing	*Resend Completed*	*Ignore*	*Resend Fail*	*Ignore*	*Resend CannotComplete*	*Resend Exit*	*Send Fail*	*Send Fail*	*Send Fail*	*Ignore*
		Canceling		Completed	Completing	Failing-*	Failing-Compensating	NotCompleting	Exiting	Ended-Canceled	Ended-Closed	Ended-Compensated	Ended
Close	*Invalid State*	*Invalid State*	*Invalid State*	Closing	*Invalid State*	*Invalid State*	*Ignore*	*Ignore*	*Invalid State*	*Ignore*	*Send Closed*	*Ignore*	*Ignore*
	Active	Canceling	Completing		Compensating	Failing-*	Failing-Compensating	NotCompleting	Exiting	Ended-Canceled	Ended-Closed	Ended-Compensated	Ended
Compensate	*Invalid State*	*Invalid State*	*Invalid State*	Compensating	*Invalid State*	*Invalid State*	*Ignore*	*Invalid State*	*Invalid State*	*Ignore*	*Ignore*	*Send Compensated*	*Ignore*
	Active	Canceling	Completing		Closing	Failing-*	Failing-Compensating	NotCompleting	Exiting	Ended-Canceled	Ended-Closed	Ended-Compensated	Ended
Failed	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Forget*	*Forget*	*Invalid State*	*Invalid State*	*Ignore*	*Ignore*	*Ignore*	*Ignore*
	Active	Canceling	Completing	Completed	Closing	Ended	Ended	NotCompleting	Exiting	Ended-Canceled	Ended-Closed	Ended-Compensated	Ended
Exited	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Forget State*	*Ignore*	*Ignore*	*Ignore*	*Ignore*
	Active	Canceling	Completing	Completed	Compensating	Failing-*	Failing-Compensating	NotCompleting	Ended	Ended-Canceled	Ended-Closed	Ended-Compensated	Ended
NotCompleted	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Forget*	*Invalid State*	*Ignore*	*Ignore*	*Ignore*	*Ignore*
	Active	Canceling	Completing	Completed	Compensating	Failing-*	Failing-Compensating	Ended	Exiting	Ended-Canceled	Ended-Closed	Ended-Compensated	Ended

Enhanced BusinessAgreementWithCoordinationCompletion protocol (Participant View)

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

Enhanced BusinessAgreementWithCoordinationCompletion protocol (Coordinator View)

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

Enhanced BusinessAgreementWithCoordinationCompletion protocol (Coordinator View)

Outbound Events	States												
	Active	**Canceling (Active,) (Completing)**	**Completing**	**Completed**	**Closing**	**Compensating**	**Failing (Active, Canceling, Completing, Compensating)**	**NotCompleting**	**Exiting**	**Ended-Failed**	**Ended-Exited**	**EndedNot-Completed**	**Ended**
Cancel	Canceling-Active	Canceling-*	Canceling-Completing	*Invalid* *State* Completed	*Invalid* *State* Closing	*Invalid* *State* Compensating	*Invalid* *State* Failing-*	*Invalid* *State* NotCompleting	*Invalid* *State* Exiting	*Invalid* *State* Ended-Failed	*Invalid* *State* Ended-Exited	*Invalid* *State* Ended-NotCompleted	*Invalid* *State* Ended
Complete	Completing	*Invalid* *State* Canceling-*	Completing	*Invalid* *State* Completed	*Invalid* *State* Closing	*Invalid* *State* Compensating	*Invalid* *State* Failing-*	*Invalid* *State* NotCompleting	*Invalid* *State* Exiting	*Invalid* *State* Ended-Failed	*Invalid* *State* Ended-Exited	*Invalid* *State* Ended-NotCompleted	*Invalid* *State* Ended
Close	*Invalid* *State* Active	*Invalid* *State* Canceling-*	*Invalid* *State* Completing	Closing	Closing	*Invalid* *State* Compensating	*Invalid* *State* Failing-*	*Invalid* *State* NotCompleting	*Invalid* *State* Exiting	*Invalid* *State* Ended-Failed	*Invalid* *State* Ended-Exited	*Invalid* *State* Ended-NotCompleted	*Invalid* *State* Ended
Compensate	*Invalid* *State* Active	*Invalid* *State* Canceling-*	*Invalid* *State* Completing	Compensating	*Invalid* *State* Closing	Compensating	*Invalid* *State* Failing-*	*Invalid* *State* NotCompleting	*Invalid* *State* Exiting	*Invalid* *State* Ended-Failed	*Invalid* *State* Ended-Exited	*Invalid* *State* Ended-NotCompleted	*Invalid* *State* Ended
Failed	*Invalid* *State* Active	*Invalid* *State* Canceling-*	*Invalid* *State* Completing	*Invalid* *State* Completed	*Invalid* *State* Closing	*Invalid* *State* Compensating	*Forget* Ended-Failed	*Invalid* *State* NotComleting	*Invalid* *State* Exiting	Ended-Failed	*Invalid* *State* Ended-Exited	*Invalid* *State* Ended-NotCompleted	*Invalid* *State* Ended
Exited	*Invalid* *State* Active	*Invalid* *State* Canceling-*	*Invalid* *State* Completing	*Invalid* *State* Completed	*Invalid* *State* Closing	*Invalid* *State* Compensating	*Invalid* *State* Failing-*	*Invalid* *State* NotComleting	*Forget* *State* Ended-Exited	*Invalid* *State* Ended-Failed	Ended-Exited	*Invalid* *State* Ended-NotCompleted	*Invalid* *State* Ended
NotComp-leted	*Invalid* *State* Active	*Invalid* *State* Canceling-*	*Invalid* *State* Completing	*Invalid* *State* Completed	*Invalid* *State* Closing	*Invalid* *State* Compensating	*Invalid* *State* Failing-*	*Forget* *State* Ended-NotCompleted	*Invalid* *State* Exiting	*Invalid* *State* Ended-Failed	*Invalid* *State* Ended-Exited	Ended-NotCompleted	*Invalid* *State* Ended