Tool Supported Analysis of Web Services Protocols

Abinoam P. Marques Jr.[1]

Brazilian Health Informatics Society
Natal-RN, Brazil

Anders P. Ravn[2] Jiří Srba[2,3] Saleem Vighio[2,4]

Department of Computer Science, Aalborg University
Aalborg, Denmark

Abstract

We describe an abstract protocol model suitable for modelling of web services and other protocols communicating via unreliable, asynchronous communication channels. The model is supported by a tool chain where the first step translates tables with state/transition protocol descriptions, often used e.g. in the design of web services protocols, into an intermediate XML format. We further translate this format into a network of communicating state machines directly suitable for verification in the model checking tool UPPAAL. We introduce two types of communication media abstractions in order to ensure the finiteness of the protocol state-spaces while still being able to verify interesting protocol properties. The translations for different kinds of communication media have been implemented and successfully tested, among others, on agreement protocols from WS-Business Activity.

Keywords: Service Oriented Computing, Communication Channels, Model Checking, Verification

1 Introduction

Service oriented computing is gaining in popularity, mainly because the Internet offers a widespread, cheap and efficient infrastructure. Thus there is an

[1] Email: abinoam@gmail.com
[2] Email: {apr,srba,vighio}@cs.aau.dk
[3] The author is partially supported by Ministry of Education of Czech Republic, MSM 0021622419.
[4] The author is supported by Quaid-e-Awam University of Engineering, Science, and Technology, Nawabshah, Pakistan, and partially by the Nordunet3 project COSoDIS.

incentive to develop applications as clients that dynamically and flexibly connect to available services over the net using for instance the Web Services (WS) protocols. For simple client-server applications this may work without many considerations of error-handling, as errors can be often handled just as exceptions in standard sequential programs. However, when several services, possibly from different organizations, are involved, more sophisticated coordination protocols need to be employed in order to implement distributed transactions supporting roll-back or other compensation mechanisms. Therefore the OMG body (www.omg.org) has put much effort into developing protocol standards to handle these issues. They include protocols for atomic transactions [13], coordination [15] and for general web services business activities [14].

Protocols are distributed algorithms and that makes them hard to analyse as they contain several local-state machines that evolve independently and communicate through message exchanges. A further difficulty is asynchronous, perhaps even unreliable communication media. In the standards, protocols are often described by a combination of state/transition tables for the individual state machines, global communication graphs, and concise English text. This is useful for understanding the intent and purpose of a protocol but may be insufficient for in-depth analysis of the possible behaviours under different communication assumptions. Here formal notations like process algebras, temporal logics and automata-based formalisms are often used [7]. We discuss possible protocol formalizations in Section 2.

We have recently worked on analysing protocols using model checking techniques. The first result was an analysis of the atomic transaction protocol [17], heavily inspired by its corresponding TLA encoding [8]. Among other points, this work demonstrated that message exchange through asynchronous media is hard to model via handshake synchronization between automata. We used the experience in a more detailed study of the BAwCC coordination protocol [18] and found a fault in the protocol design [19].

During this work we have seen that it is far from simple to prepare the analysis; many hours are spent on understanding the protocol 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. For instance, the encoding of the BAwCC protocol into the model checker UPPAAL [21] presented in [18] ends up with 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. This process can to a large extent be automated, and the main contributions of this paper are a tool chain and abstraction techniques, presented in Section 3, that were developed for that purpose. The components of the tool chain are detailed in Section 4.

As a further contribution, the tool chain is used to analyse the Subser-

vice Termination and Alternating Bit Protocols, the first one being used as a running example. More importantly, we have applied the tool chain to some recently studied web services protocols as well as to the BAwPC protocol [14] that has not been previously verified. The automatic approach showed a large degree of flexibility and the verification results are summarized in Section 5. We observe that it is now a matter of hours to conduct an experiment with a proposed protocol. It should also be feasible for protocol developers to use our tool without any deep knowledge of the particular model checker we employ in our tool chain. This perspective and future work are discussed in Section 6.

Related Work. Reachability analysis is a well-known technique for the analysis of small communication protocols (see e.g. [22,1]). An approach most related to our work was presented in [12]. Here the authors perform a static analysis of 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 [4], partial decidability results exist for lossy communication channels [6], however with nonprimitive recursive complexity [20] which puts them among the hardest decidable problems. In our approach we provide a practical solution that allows to analyze complex protocols like the ones from WS-Business activity in a matter of seconds. Recently Lohmann [9] surveys possible communication models and divides them into (i) ordered/unordered, (ii) bounded/unbounded and (iii) single/multiple buffer communication. For bounded media different nonblocking sending strategies are discussed as well. In our paper we focus both on ordered and unordered as well as single and multiple buffer communication strategies, but our main goal is to argue about the behaviour of protocols with unbounded communication via the use of model checking techniques that however allow us to verify only bounded media. Moreover we consider unreliable communication policies.

2 Protocol Modelling

Web services protocols are usually described by means of state/transition tables (see e.g. [14]) that specify the behaviours of the protocol roles on *inbound events* (received messages) and on *outbound events* (sent messages). A small example of such a table describing a Subservice Termination Protocol (STP) is presented in Figure 1. The protocol contains three roles A, B and C, all of them are initially in their *Active* states. Once the role A executes the out-

ROLE	A			
	MESSAGES \STATES	Active	AwaitingB	Ended
OUTBOUND	exitB	goto Active		
INBOUND	preparingB	goto AwaitingB	goto AwaitingB	goto Invalid
INBOUND	exitedB		goto Ended	goto Ended
ROLE	B			
	MESSAGES \STATES	Active	AwaitingC	Ended
OUTBOUND	preparingB		goto AwaitingC	
OUTBOUND	exitC		goto AwaitingC	
OUTBOUND	exitedB			goto Ended
INBOUND	exitB	send preparingB goto AwaitingC		
INBOUND	exitedC		send exitedB goto Ended	
ROLE	C			
	MESSAGES \STATES	Active	Ended	
OUTBOUND	exitedC		goto Ended	
INBOUND	exitC	send exitedC goto Ended		

Fig. 1. State/Transition Table of Subservice Termination Protocol (STP)

bound event *exitB*, it waits on confirmation from B that it is preparing for termination and once B is exited, it will reach the *Ended* state. Similarly, the role B waits for the *exitedC* event from role C before it can terminate. Moreover, once the role A receives the message *exitedB* from B and enters its *Ended* state, the arrival of the message *preparingB* will lead to an *Invalid* state as the messages arrived in a wrong order. The protocol moreover compensates for the possibility of messages being lost by repeatedly retransmitting all outbound events.

2.1 Abstract Protocol Model

As our aim is to provide a tool supported analysis of web services protocols like the STP example, we need to formalize the notion of a protocol. We shall use an automata-based approach as it is convenient for our purposes, but as a part of our tool chain we provide a front end that accepts state/transition tables created in a spreadsheet application and translates them into our automata model, essentially a conventional Mealy machine.

Definition 2.1 An *abstract protocol model* is a pair $(Msgs, Roles)$ where $Msgs$ is a finite set of *messages* with $\diamond \in Msgs$ being the *empty message* and $Roles$ is a finite set of *roles* such that for every role $A \in Roles$ we have its description $D_A = (S_A, \longrightarrow_A)$ where S_A is a finite set of *states* and $\longrightarrow_A \subseteq S_A \times Msgs \times Msgs \times S_A$ is the set of the transitions of the role A.

Whenever $(s, m, m', s') \in \longrightarrow_A$ we shall write $s \xrightarrow{m,m'}_A s'$ or simply $s \xrightarrow{m,m'} s'$ if the role A is clear from the context. The meaning is that the role A in its current state s is ready to receive a message m and after that it sends the message m' and changes its state to s'. The messages m and m' can be empty,

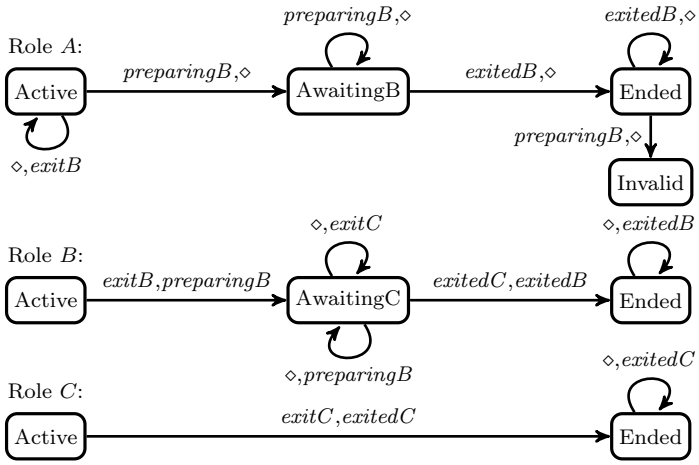


Fig. 2. Formal Specification of the STP Protocol

meaning that the transition can happen either without receiving any message or without sending any message. If both m and m' are empty, this represents an internal transition.

A formal automata-based model of the STP protocol is depicted in Figure 2. One can easily verify that this abstract protocol model describes the same behaviour as the state/transition tables in Figure 1.

2.2 Communication Policy

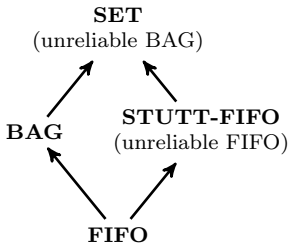
In order to define the semantics of the abstract protocol model, we need to discuss the communication policies. We shall discuss *asynchronous communication* policies with different reliability requirements on the communication medium. We consider e.g. FIFO (First In First Out) communication channels representing a perfect order-preserving communication or, as the other extreme, unreliable (lossy and duplicating) communication policy where messages can be reordered. We can abstract the possible medium implementation by its interface.

Definition 2.2 A *communication medium* is a data structure *Medium* providing the following three operations:

- $send : Medium \times Msgs \to Medium$,
- $available : Medium \times Msgs \to \{true, false\}$, and
- $receive : Medium \times Msgs \to Medium$.

Given a current medium $med \in Medium$ and a message $m \in Msgs$, the operation $med.send(m)$ updates the communication medium with the message m that was sent by one of the roles, $med.available(m)$ answers whether the message m is available for receiving (without modifying the medium) and finally $med.receive(m)$ receives the message m and updates the communication

medium accordingly. The empty message $\diamond$ is always available and sending or receiving the message $\diamond$ has no effect on the medium.



We provide a few examples of possible medium implementations for some of the classical communication policies. For the perfect FIFO policy, we model the medium as a queue. The operation $send(m)$ simply enqueues m at the end of the queue, $available(m)$ checks whether m is at the head of the queue and $receive(m)$ removes the message m from the front of the queue. Similarly for order-preserving but lossy and duplicating policy (called STUTT-FIFO for stuttering FIFO), the call $send(m)$ adds the message m at the end of the queue only if it is not already present there (if the last sent message was m then sending m does not change the queue). When a message m is received then an arbitrary number of messages before m can be dequeued (lossiness) but the message itself stays in the queue (duplication). As another example, a perfect medium, which can however reorder messages, can be modelled as a multiset (we call it BAG). Mathematically, the medium can be represented as a function f from the set of messages $Msgs$ to nonnegative integers. The operation $send(m)$ is then implemented as $f(m) := f(m) + 1$, $available(m)$ is simply returning $f(m) > 0$ and $receive(m)$ is equivalent to $f(m) := f(m) - 1$. Finally, an unreliable medium with reordering can be represented as a set SET of messages that have been already sent. Now $send(m)$ means SET := SET $\cup \{m\}$, availability is checking the presence of the message in SET and receive does not modify SET, this models duplication of messages.

2.3 Semantics of Abstract Protocol Model

Let us assume a given communication policy. The semantics of an abstract protocol model $(Msgs, Roles)$ where $Roles = \{A_1, A_2, \ldots, A_n\}$ is given as transition system with states (configurations) of the form $(s_1, s_2, \ldots, s_n, med)$ where $s_i \in S_{A_i}$ for all i, $1 \leq i \leq n$, are the current states of all roles and $med \in Medium$ represents the current content of the communication medium. The transitions are defined by $(s_1, \ldots, s_i, \ldots, s_n, med) \Rightarrow (s_1, \ldots, s_i', \ldots, s_n, med')$ whenever $s_i \xrightarrow{m,m'}_{A_i} s_i'$ is a transition of the role A_i such that $med.available(m)$ is true and $med' = (med.receive(m)).send(m')$. By $\Rightarrow^*$ we denote the reflexive and transitive closure of $\Rightarrow$.

Consider now our running example from Figure 2. Clearly, with FIFO policy the communication medium this protocol is unbounded due to the possibility of unbounded message retransmission. Also STUTT-FIFO is unbounded due to e.g. the alternating resubmission of the messages *exitedB* and *exitedC*

in the ended states. Due to the resubmission of messages also BAG makes the medium of our example protocol unbounded. On the other hand, for SET the state-space remains finite and we can construct it algorithmically.

2.4 Analysis of Abstract Protocol Models

In the analysis of protocols we are interested in state reachability problems.
State Reachability Problem: given a target state s of a role A_i and the initial configuration of the protocol $c^0 = (s_1^0, s_2^0, \ldots, s_n^0, med^0))$ where med^0 contains no messages, we ask whether there is a reachable protocol configuration $c = (s_1, \ldots, s_{i-1}, s, s_{i+1}, \ldots, s_n, med)$ such that $c^0 \Rightarrow^* c$ for some states $s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n$ and some medium configuration med.

The state reachability problem can provide safety guarantees about the protocol behaviour. Considering our example from Figure 2 with the SET communication policy, it is possible for role A to reach an invalid state. This can be interpreted as an error in the protocol design as the protocol is not immune to reordering of messages. On the other hand, we may want to verify that for order-preserving communication policies, the protocol is safe (does not enter any invalid state). However, this is impossible to do in a fully automatic way as the model with FIFO communication has a full Turing power [4] and the corresponding transition system cannot be enumerated as it has in general infinitely many reachable configurations. There are similar problems with infinite state-spaces for the media STUTT-FIFO and BAG. For BAG the state reachability problem can be shown equivalent to the EXPSPACE-complete coverability problem on Petri nets (see e.g. [5]). In this paper we shall suggest different techniques that will allow us to efficiently answer the state reachability problem on a practically interesting set of protocol models.

Returning to the hierarchy of communication media, it is easy to see that if we show that a state is not reachable under the SET policy, it will not be reachable in any other policy below it. On the other hand, if a state is reachable in FIFO communication, then it will be reachable (with exactly the same trace) also in any communication policy above it.

We shall conclude this section with the discussion of some instances of the state reachability problem relevant for the verification of WS protocols. We assume that all roles contain at least three states called *Active*, *Ended*, *Invalid* representing the state where each role starts, where it ends (both successfully or with a failure) and finally a state representing inconsistency in the protocol design. Such states are often present in the standard specification documents for web services like WS-BA [14]. The following questions will be of interest.

- **Boundedness:** We ask whether starting with all roles in their *Active* states and the empty medium with a given (finite) capacity, is it the case that for all executions the medium does not exceed its capacity?

- **Correctness:** We ask whether starting with all roles in their *Active* states and the empty communication medium, do all roles avoid entering their *Invalid* states in all possible executions?
- **Termination:** We ask whether starting with all roles in their *Active* states and the empty communication medium, is there an execution where all roles reach their *Ended* states?
- **Deadlock-Freeness:** We ask whether starting with all roles in their *Active* states and the empty medium, is there a possible continuation from any reachable configuration but the one where all roles are in their *Ended* states?

The aim is to design protocols that are correct (cannot reach invalid states), can terminate and have no deadlocks. We call such protocols *safe*. As already discussed, most protocols that communicate over FIFO-like channels are not bounded. In the next section we discuss possible approaches that will allow us to prove that the protocol in question is *safe* even for an unbounded medium.

3 Abstractions of Communication Media

Let us consider a situation where a given protocol is sensitive to the order of message arrivals (and hence cannot be proved safe with the SET medium), but at the same time we wish to automatically establish its safety with respect to order-preserving communication policies like FIFO or STUTT-FIFO. We shall now suggest two abstraction strategies to tackle the problem that FIFO and STUTT-FIFO channels are in general unbounded. Both strategies provide an over-approximation of the communication medium, meaning that if the protocol is proved correct under the abstracted communication policy, it will be correct also under FIFO and STUTT-FIFO.

The main reason for introducing the abstractions is to guarantee that the state-space of the corresponding transition system becomes bounded and automatic analysis can be performed. As the problem is in general undecidable [4] and model checking of protocols communicating over FIFO channels is hence impossible, the proposed abstractions are not universal. On some protocols, the abstractions may not guarantee boundedness of the medium. On others the abstractions may be too coarse and thus they may not allow us to verify safety of the protocol, even though it is actually safe for the perfect FIFO communication. Nevertheless, as we demonstrate in our case studies provided in Section 5, the proposed abstractions are sufficient to establish safety of several well-known WS protocols.

3.1 Multiple Channel Optimization

Under the perfect FIFO or STUTT-FIFO communication policy, one can think of each sent message as being time-stamped. The property of the medium is

that it delivers the messages in the order in which they were time-stamped. In other words, the global order of messages is preserved and the medium can be seen as one universal FIFO or STUTT-FIFO channel.

In our first abstraction, called *multiple channel optimization*, we will 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 channel do not synchronize in any way. We may possibly create a separate channel for each message which would in result give us either a communication policy equivalent to BAG (when applied to FIFO) or SET (when applied to STUTT-FIFO). This will clearly not help us with the automatic analysis as we apply the abstractions only to protocols that are not correct under the BAG or SET communication policies. Hence we instead introduce a more refined multiple channel optimization.

The idea is that for each message m that appears in the protocol description, we will compute the function $\mathit{recipients}(m)$ which contains all roles that can possibly receive the message m. On our running STP protocol example from Figure 2 we get the following: $\mathit{recipients}(\mathit{exitB}) = \{B\}$, $\mathit{recipients}(\mathit{preparingB}) = \{A\}$, $\mathit{recipients}(\mathit{exitC}) = \{C\}$, $\mathit{recipients}(\mathit{exitedB}) = \{A\}$, $\mathit{recipients}(\mathit{exitedC}) = \{B\}$. In the STP protocol each message has exactly one recipient, hence the sets are singletons. In general scenarios that include broadcast or multi-party communication, a message can have several recipients.

Formally, for a given protocol $(\mathit{Msgs}, \mathit{Roles})$ where each role $A \in \mathit{Roles}$ has its description $(S_A, \longrightarrow_A)$ we define for each $m \in \mathit{Msgs} \setminus \{\diamond\}$ its recipients: $\mathit{recipients}(m) = \{A \in \mathit{Roles} \mid (s, m, m', s') \in \longrightarrow_A,\ s, s' \in S_A,\ m' \in \mathit{Msgs}\}$. Let $\mathit{channels} = \{\mathit{recipients}(m) \mid m \in \mathit{Msgs}\}$ serve as names of newly introduced communication channels. Now every time a message m is sent, it arrives to the channel $\mathit{recipients}(m)$ and whenever a role checks the availability of a message, it does so on the channel $\mathit{recipients}(m)$. 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 optimization process described above can be run in a fully automatic way (as implemented in our tool) and it has proved particularly useful to verify protocols like Business Agreement with Coordinator Completion protocol from the WS-Business Activity coordination framework [14].

3.2 Unordered Messages

We shall now discuss another abstraction technique, motivated by the fact that the multiple channel optimization may not be sufficient to achieve boundedness of the communication medium as it can be seen e.g. in our STP protocol from Figure 2. The reason here is that the role B can receive the messages

exitB and *exitedC* and both these messages can be repeatedly retransmitted in an arbitrary order, causing the unboundedness of the FIFO and STUTT-FIFO medium, even with multiple channel optimization.

Nevertheless, the protocol is still correct (as also formally verified in Section 5) in the sense that there are no invalid states reachable as long as messages cannot be reordered. In order to show this, we may notice that for example the ordering of the message *exitB* relative to the other messages does not seem to be relevant. We will mark it (using the symbol * in our tool implementation) as a message where it is not necessary to preserve its ordering. Formally, this means that we introduce an additional communication channel behaving as a SET medium such that all marked messages are sent/received to/from this channel. If all messages get marked then we get the SET communication medium. As showed later, marking the single message *exitB* is sufficient to prove the boundedness of the medium, while at the same time allowing us to prove that no invalid states can be reached.

One issue with this abstraction is the selection of messages for marking, as it is up to the designer of the protocol to mark unordered messages. In principle one may automate the process by exploring all combinations of marked/unmarked messages, however, for larger protocols this may not be computationally feasible due to exponentially many such combinations. The development of possible heuristics so that the markings of messages that are more likely to work (provide a bounded medium but still avoid the reachability of invalid states) are enumerated first is left for future research.

4 Automatic Analysis and Tool Support

We shall now outline a solution to the state reachability problem for communication protocols. The answer to this problem is provided by automatic translation of the state/transition tables into an intermediate XML format (denoted as part (i) in Figure 3), followed by a translation to networks of timed automata suitable for a direct verification in the model checker UPPAAL [3,21] (denoted as part (ii) in Figure 3).

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 proposed 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 state/tables of nontrivial size in our applications.

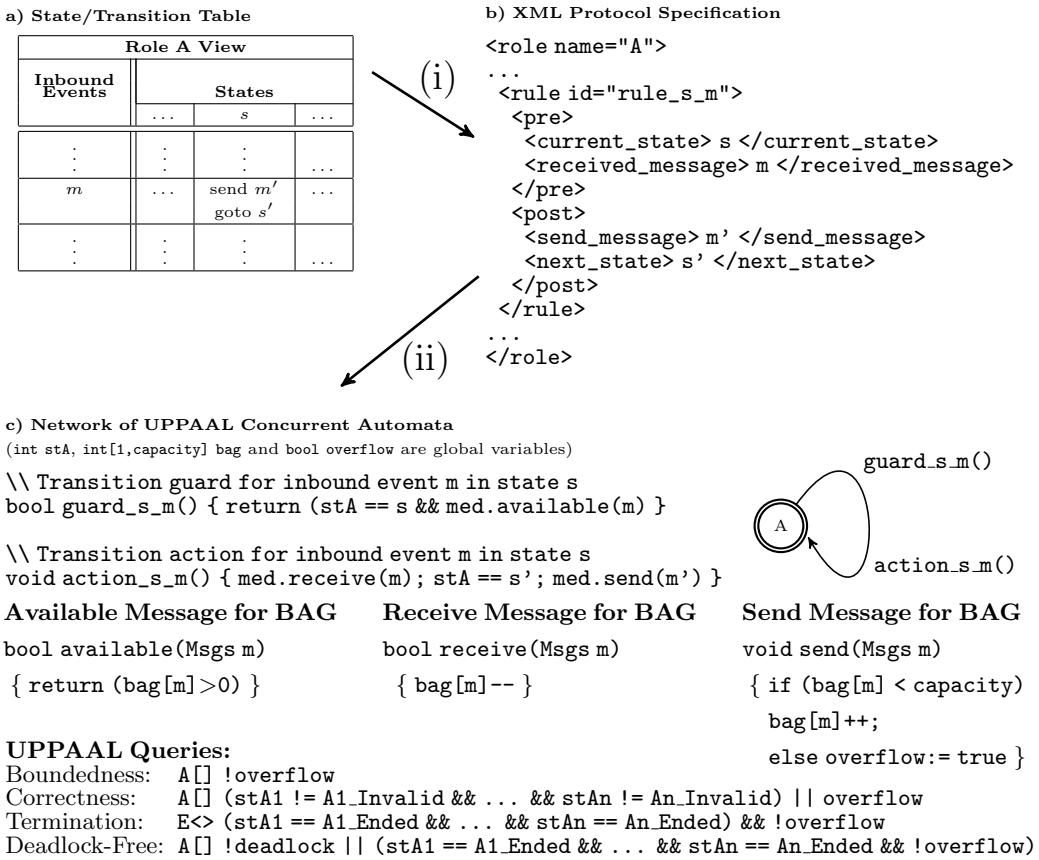


Fig. 3. Process of Automatic Analysis of WS-Protocols with the Medium BAG

In the second part of the translation, the XML description is further encoded into networks of communicating finite automata in the UPPAAL [21] style. UPPAAL is a tool for modelling, simulation and verification of networks of finite automata communicating via handshake and via shared variables. It allows the user to include a restricted C-like syntax for describing guards and updates of transitions and one of the main UPPAAL features is also the explicit treatment of timing information by using real-valued clocks. In the present paper we do not exploit the timing aspects but this will be considered in our future work. We refer the reader to [3] for a detailed introduction into UPPAAL modelling language.

In part (ii) of our translation each `pre` tag is converted into a transition guard (written as a function in C-like syntax accepted by UPPAAL) and each `post` tag is translated to an action (again represented in C-like syntax) that is performed should the transition be executed. The transition forms a loop for every rule of the given role as the data is stored in global variables. The states and message names are declared as global constants and global variables keep track of the states of each role (variable `stA` for role A in our example) as well as of the current content of the communication medium. In our example we

chose to demonstrate the obvious implementation of the communication policy BAG. 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, the global flag `overflow` is set to true. As UPPAAL allows a large set of C-constructs in its syntax, the more advanced media like FIFO and STUTT-FIFO (including the abstractions) are implemented in the expected way as in any other imperative programming language. The details can be found in our publicly available tool chain.

Finally, in Figure 3, we describe how our protocol related questions of boundedness, correctness, termination and deadlock-freeness are formulated in the UPPAAL query language (assuming the role names `A1, A2, ..., An`). The queries are formulated in a subset of CTL logic used in UPPAAL (for more info see [3]). Intuitively, the path quantification `A[]` stands for "for all reachable configurations holds that" and `E<>` stands for "there is a reachable configuration such that".

Tool Details

Translations (i) and (ii) from Figure 3 are implemented in the open source tool `csv2uppaal` available at [11]. 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, in command line mode, the tool calls the UPPAAL verification engine to verify the properties of boundedness, correctness, termination and deadlock-freeness. For Mac OS we provide additionally also a graphical user interface as shown in the appendix. The outcome of the verification is the 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. The 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 more advanced queries that are protocol specific. For example in our running STP protocol from Figure 2 we verified an additional property saying that role A can enter the state *Ended* only after the roles B and C already reached their *Ended* states. This query is formulated as

```
A[] (stA!=A_Ended || (stB==B_Ended && stC==C_Ended))
```

Buffer Type	Properties	BAwCC		BAwPC		STP	ABP
		Org.	Enh.	Org.	Enh.		
BAG	Boundedness	no	no	no	no	no	no
	Correctness	NO	NO	NO	yes	NO	NO
SET	Boundedness	YES	YES	YES	**YES**	YES	YES
	Correctness	NO	NO	NO	**YES**	NO	NO
FIFO	Boundedness	no	no	no	no	no	no
	Correctness	yes?	yes	yes?	yes	yes	yes
STUTT-FIFO	Boundedness	no	no	no	no	no	no
	Correctness	NO	yes	NO	yes	yes	yes
multiple channel STUTT-FIFO	Boundedness	no	**YES**	no	YES	no	**YES**
	Correctness	NO	**YES**	NO	YES	yes	**YES**
multiple channel reorder STUTT-FIFO	Boundedness	YES	YES	YES	YES	**YES**	YES
	Correctness	NO	YES	NO	YES	**YES**	YES

Fig. 4. Summary of Verification Results (Org. means original, Enh. means enhanced)

and UPPAAL confirms that it holds.

5 Applications

In order to investigate the applicability of our proposed framework, we carried out experiments on case studies ranging from well-known academic examples like Alternating Bit Protocol (ABP) [2,10] and Subservice Termination Protocol (STP) described in this paper, to larger-size protocols from WS-Business Activity specification [14], namely Business Agreement with Coordination Completion (BAwCC) and Business Agreement with Participant Completion (BAwPC). The appendix presents the full details.

State/transition tables were described in OpenOffice as spreadsheets and then automatically verified using our `csv2uppaal` tool. The verification results are presented in Figure 4. As all considered protocols were deadlock-free and terminating (apart from ABP where termination is not desirable), we list only the answers for boundedness of the medium and correctness of the protocols (absence of invalid states). The answers "YES" and "NO" in capital letters mean that the tool returned a conclusive answer on the instances in question. The answer "yes" stands for the fact that even though the tool on this concrete medium was not bounded, we were able to conclude the answer using our abstraction techniques (in bold font are marked the prominent positive results that imply correctness for all other less-abstract media). Finally, the answer "no" on boundedness stands for the fact that for any chosen

medium capacity (where the verification terminated within a reasonable time limit) the answer was negative. As boundedness is an undecidable problem, a more precise answer cannot be obtained automatically in general. However, a manual examination of error traces revealed that for all our instances the medium was really unbounded.

The ABP was proved correct for order-preserving communication media by considering the STUTT-FIFO with multiple channel abstraction. For unordered asynchronous communication the protocol is (as expected) not correct. Similarly the STP was proved correct for all order-preserving communications by marking the message *exitB* as unordered and using multiple channel STUTT-FIFO communication.

Both in BAwCC and BAwPC protocols we discovered an error for all considered communication media except for perfect FIFO, where the correctness seems to be valid, though we were not able to prove it in automatic way, hence the answers "yes?". Example of a trace leading to an invalid state is given in the appendix and it has been communicated to the OASIS body. The main reason for the problems is a confusion on retransmission of messages in the ended states. We suggested fixes to the protocols and designed enhanced versions of both protocols that are presented in the appendix. The enhanced BAwPC protocol now turned out to be correct for the most general SET communication (and hence also for any less abstract one) while the enhanced version of BAwCC still contains traces leading to invalid states. The issue here is more subtle and it has been announced on the OASIS discussion forum [16] and a correction is currently under development. Using our automatic analysis we were able to identify that the issue is connected with reordering of messages and not with the lossiness of the medium (the protocol is incorrect even for BAG).

6 Conclusion

We presented an automatic, tool-supported framework for modelling and analysis of communication protocols with a particular focus on web services protocols. A particular strength of our solution is the possibility to choose different models of communication media and various abstractions in order to prove protocol correctness. The approach was successfully tested on e.g. protocols from WS-BA, where in one case we confirmed the presence of a fundamental problem in the protocol design and in the other one we suggested an improvement in the specification sufficient to automatically validate its correctness.

Our tool chain is modular as the state/transition tables are first translated to an intermediate XML format that is further translated to UPPAAL automata. In the future work we plan to provide different front ends that will accept other popular formats for describing protocols, including parameteri-

zation, simple data structures, and timing aspects.

References

[1] Barghouti, N., N. Nounou and Y. Yemini, *An integrated protocol development environment*, in: *Protocol Specification Testing and Verification VI* (1987), pp. 63–69.

[2] Bartlett, K. A., R. A. Scantlebury and P. T. Wilkinson, *A note on reliable full-duplex transmission over half-duplex links*, Commun. ACM **12** (1969), pp. 260–261.

[3] Behrmann, G., A. David and K. Larsen, *A tutorial on UPPAAL*, in: *Proc. of SFM-RT'04*, number 3185 in LNCS (2004), pp. 200–236.

[4] Brand, D. and P. Zafiropulo, *On communicating finite-state machines*, J. ACM **30** (1983), pp. 323–342.

[5] Esparza, J., *Decidability and complexity of Petri net problems — An introduction*, in: W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, LNCS **1491**, Springer Berlin / Heidelberg, 1998 pp. 374–428.

[6] Finkel, A., *Decidability of the termination problem for completely specified protocols*, Distributed Computation **7** (1994), pp. 129–135.

[7] Holzmann, G., "Design and Validation of Computer Protocols," Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[8] Johnson, J., D. E. Langworthy, L. Lamport and F. H. Vogt, *Formal specification of a web services protocol*, Journal of Logic and Algebraic Programming **70** (2007), pp. 34–52.

[9] Lohmann, N., *Communication models for services*, in: *Proc. of ZEUS'10*, CEUR Workshop Proceedings **563** (2010), pp. 9–16.

[10] Lynch, W. C., *Computer systems: Reliable full-duplex file transmission over half-duplex telephone line*, Commun. ACM **11** (1968), pp. 407–410.

[11] Marques, A., A. Ravn, J. Srba and S. Vighio, *The tool csv2uppaal*, available at http://www.cs.aau.dk/~srba/csv2uppaal.zip.

[12] Naumovich, G., L. Clarke and L. Osterweil, *Verification of communication protocols using data flow analysis*, SIGSOFT Softw. Eng. Notes **21** (1996), pp. 93–105.

[13] Newcomer, E. and I. R. (chairs), *Web services atomic transaction version 1.2* (2009), http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec.html.

[14] Newcomer, E. and I. R. (chairs), *Web services business activity version 1.2* (2009), http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os/wstx-wsba-1.2-spec-os.html.

[15] Newcomer, E. and I. R. (chairs), *Web services 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.

[16] OASIS, *Discussion forum, report on error trace in BAwCC* (2011), http://markmail.org/message/xgnyonkihfif5vz2.

[17] Ravn, A., J. Srba and S. Vighio, *A formal analysis of the web services atomic transaction protocol with UPPAAL*, in: *Proc. of ISOLA'10)*, LNCS **6416** (2010), pp. 579–593.

[18] Ravn, A., J. Srba and S. Vighio, *Modelling and verification of web services business activity protocol*, in: P. Abdulla and K. Leino, editors, *Proc. of TACAS'11*, LNCS **6605** (2011), pp. 357–371.

[19] Robinson, I., *Answer in WS-BA discussion forum, July 14th, 2010* (2010), http://markmail.org/message/wriewgkboaaxw66z.

[20] Schnoebelen, P., *Verifying lossy channel systems has nonprimitive recursive complexity*, Information Processing Letters **83** (2002), pp. 251–261.

[21] UPPAAL, http://www.uppaal.com.

[22] Vuong, S. T., D. D. Hui and D. D. Cowan, *Valira — a tool for protocol validation via reachability analysis*, in: *Protocol Specification, Testing and Verification VI* (1987), pp. 35–41.

APPENDIX

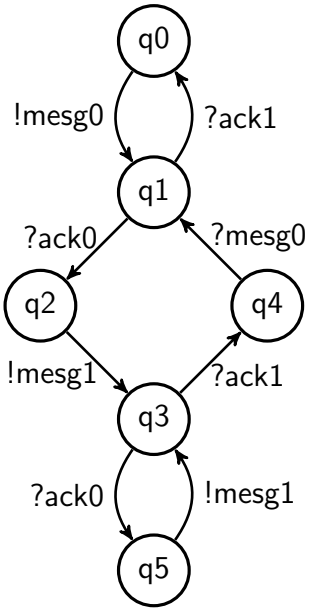


Fig. A.1. ABP: Sender

A Appendix

A.1 Alternating Bit Protocol

The Alternating Bit Protocol [2] [10] (ABP for short), is a simple data-link layer protocol. The protocol involves three roles: A Sender, a Receiver and the medium which is used to send and receive data and acknowledgments between the sender and the receiver. The operation of the protocol is quite simple, initially the sender sends a bit (0 or 1) to the medium, the medium receives the bit and sends to the receiver. The receiver receives a bit and sends an acknowledgment back to the medium. The medium then sends the acknowledgment to the sender. The sender after receiving the acknowledgment, flips the bit and starts all over again. There are possibilities that the bits can be lost by the medium. Figures A.1 and A.2 show the abstract protocol models of the sender and the receiver, respectively. The sent messages are prefixed with '!' and the received ones with '?'. The graphs were taken directly from [7] and the receiver was equipped with an invalid state that is entered if the sender and receiver get out of synchrony.

A.2 WS-Business Activity

The WS-BA specification [14] has been developed to overcome the problems of traditional transaction processing systems where the transactions only spans for very short time and suffer from resources locking. WS-BA is part of Web Services Transaction Framework (WSTF) and is specially designed to support long-running business transactions in order to achieve a consistent agreement

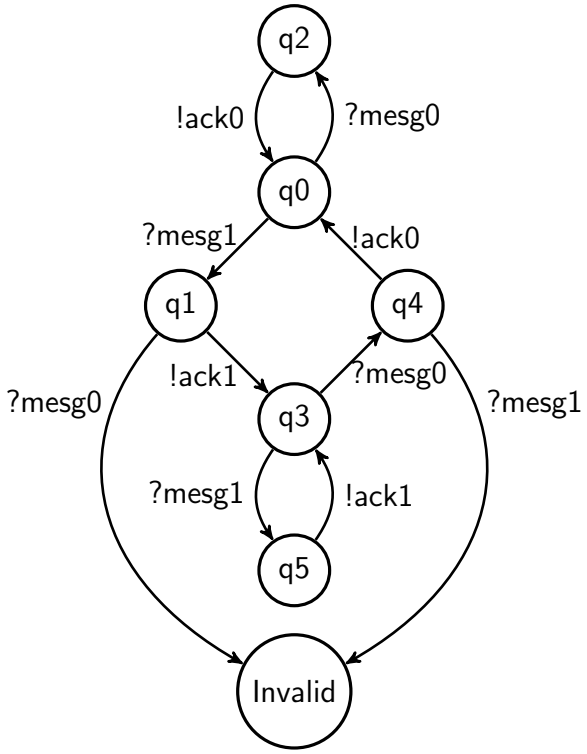


Fig. A.2. ABP: Receiver

on the outcome of these transactions. Following WS-Coordination framework, it involves two coordination types which are used for coordinating activities among distributed systems. These coordination types include: Atomic Outcome, in which all the parties involved in the transaction always reach the same outcome i.e. either closed or canceled/compensated and Mixed Outcome, in which some of the participants can reach a cancel or a compensate decision and others can reach a closed decision. Each of these coordination types can be used in two coordination protocols supported by WS-BA specification. These coordination protocols include: Business Agreement with Coordination Completion(BAwCC) and Business Agreement with Participant Completion (BAwPC).

A participant registered for BAwCC completion protocol relies on its coordinator to tell it when it has received all requests to perform its work with a business activity. On the other hand a participant registered for BAwPC does not rely on its coordinator and knows when it has completed all its work for a business activity.

WS-BA specification describes the behaviour of the underlying protocols using the state/transition diagrams accompanied with English language description. Figure A.4 and Figure A.5 show the abstract behaviour of the BAwCC and BAwPC protocols respectively between a coordinator and a participant.

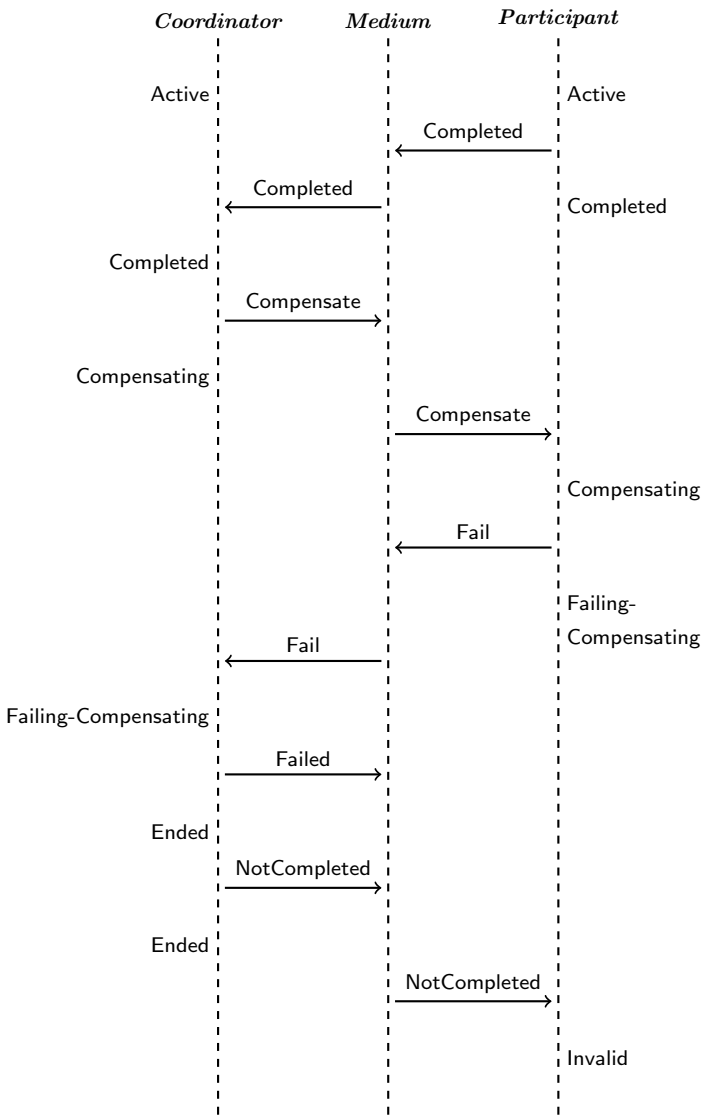


Fig. A.3. Error Trace in BAwPC Leading to Invalid State

We analyze correctness of BAwPC and BAwCC protocols based on the reply we received from the designers of the protocol via their discussion forum [19]. Validation is performed with a simple reachability property, which ensures that the protocol never reaches a situation where either the transaction coordinator or the participant enter invalid states (only upon receipt of messages) and at the same time there is no buffer overflow. Initially, we verified this property under SET communication policy, however, the property turned out to be false for both the protocol types. An error trace in BAwPC which leads to the Invalid state is presented in Figure A.3.

A careful look at error traces suggests that the problem happens because once the coordinator or the participant enters Invalid state, it is still possible

for them to send different types of messages other than the messages which led them to Invalid states. This causes a confusion at the coordinator or the participant side when they are in states in which these messages are are not intended to be received and therefore, the protocol violates correct operation by reaching into Invalid states.

Therefore, based on our analysis, we provide some fixes to the original protocols by introducing three additional ended states for each role. State/transition tables of our enhanced protocols are given in this appendix later on.

After these fixes, we verified the same validity property under SET communication policy and proved that BAwPC protocol achieves correct operation by not reaching into the Invalid states. However, BAwCC protocol type did not satisfy this property even after fixes. After looking at error traces we found that the problem happens when the messages are received in a different order in which they were sent. On the other hand the enhanced BAwCC protocol is proved to be correct under FIFO, LOSSY-FIFO and STUTT-FIFO communication policies using our abstractions.

For completeness we include a short description of both protocols.

A.2.1 Business Agreement with Coordination Completion

A state/transition diagram for BAwCC is shown in Figure A.4. 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 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.

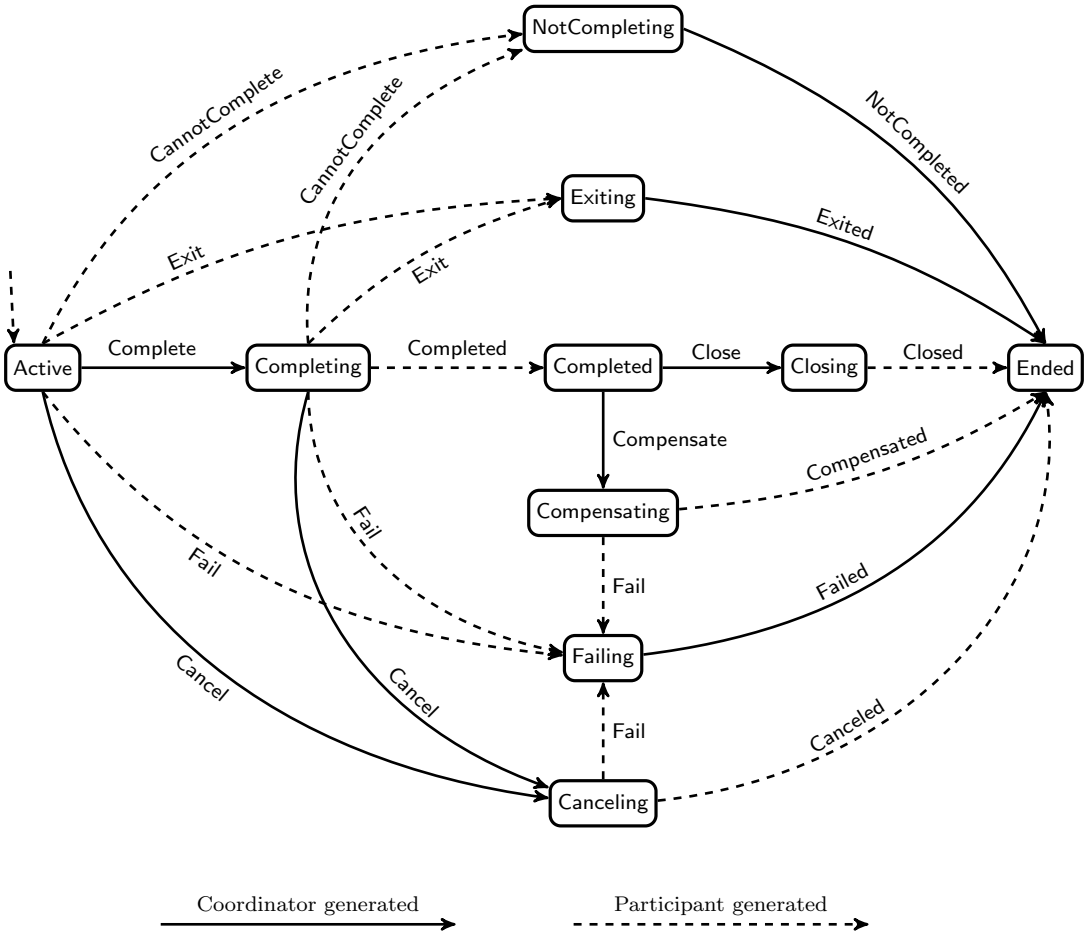


Fig. A.4. Business Agreement with Coordinator Completion

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 dis-

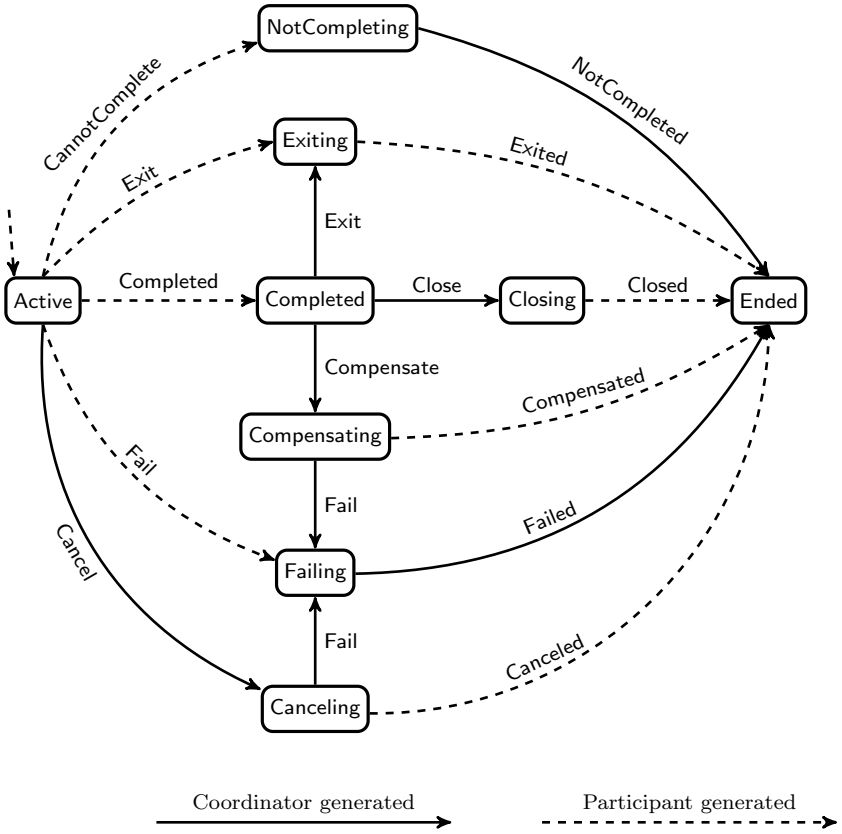


Fig. A.5. Business Agreement with Participant Completion

carded 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.

A.2.2 Business Agreement with Participant Completion

In this type of protocol, the participant knows when it has completed all its work for a business activity and therefore, it sends a Completed notification to the coordinator to signal that it has successfully completed its work. Once the coordinator receives Completed notification then it proceeds to the Completed state. The coordinator in Completed state 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 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 state. A CannotComplete notification informs the coordinator that the participant can not successfully complete its work. 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 coordinator will reply with the Exited message and will forget about the participant. The participant will also forget about the activity. In Active state 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.

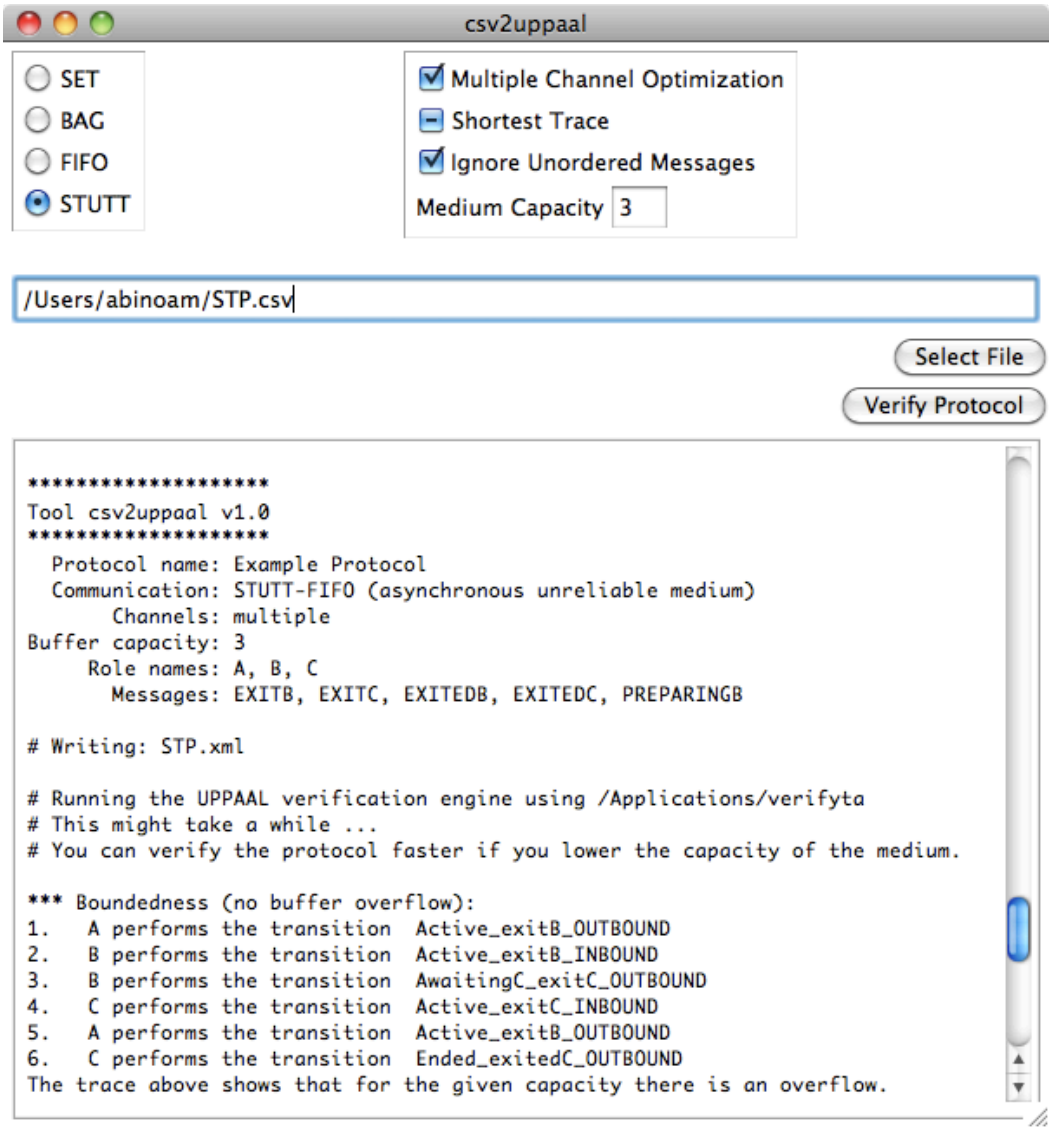


Fig. A.6. Screenshot of `csv2uppaal` GUI on Mac OS X

State/Transition Tables for WS-BA Protocols

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* Canceling	Exiting	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-*	*Invalid State* NotCompleting	Exiting	*Invalid State* Ended
Completed	*Invalid State* Active	*Invalid State* Canceling	Completed	Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-*	*Invalid State* NotCompleting	*Invalid State* Exiting	*Invalid State* Ended
Fail	Failing-Active	Failing-Canceling	Failing-Completing	*Invalid State* Completed	*Invalid State* Closing	Failing-Compensating	Failing-*	*Invalid State* NotCompleting	*Invalid State* Exiting	*Invalid State* Ended
CannotComplete	NotCompleting	*Invalid State* Canceling	NotCompleting	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-*	NotCompleting	*Invalid State* Exiting	*Invalid State* Ended
Canceled	*Invalid State* Active	*Forget* Ended	*Invalid State* Completing	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-*	*Invalid State* NotComleting	*Invalid State* Exiting	Ended
Closed	*Invalid State* Active	*Invalid State* Canceling	*Invalid State* Completing	*Invalid State* Completed	*Forget* Ended	*Invalid State* Compensating	*Invalid State* Failing-*	*Invalid State* NotComleting	*Invalid State* Exiting	Ended
Compensated	*Invalid State* Active	*Invalid State* Canceling	*Invalid State* Completing	*Invalid State* Completed	*Invalid State* Closing	*Forget* Ended	*Invalid State* Failing-*	*Invalid State* NotCompleting	*Invalid State* Exiting	Ended

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**	**Completed**	**Closing**	**Compensating**	**Failing (Active, Canceling, Completing)**	**Failing (Compesating)**	**NotCompleting**	**Exiting**	**Ended-Canceled**	**Ended-Closed**	**Ended-Compensated**	**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-Canceled	*Ignore* Ended-Closed	*Ignore* Ended-Compensated	*Ignore* 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-Canceled	*Send Fail* Ended-Closed	*Send Fail* Ended-Compensated	*Ignore* 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	*Ignore* Ended-Canceled	*Send Closed* Ended-Closed	*Ignore* Ended-Compensated	*Ignore* 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	*Ignore* Ended-Canceled	*Ignore* Ended-Closed	*Send Compensated* Ended-Compensated	*Ignore* 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-Canceled	*Ignore* Ended-Closed	*Ignore* Ended-Compensated	*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 Invalid* NotCompleting	*Forget* Ended	*Ignore* Ended-Canceled	*Ignore* Ended-Closed	*Ignore* Ended-Compensated	*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-Canceled	*Ignore* Ended-Closed	*Ignore* Ended-Compensated	*Ignore* 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* Canceling	Exiting	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-*	*Invalid State* NotCompleting	Exiting	*Invalid State* Ended-Canceled	*Invalid State* Ended-Closed	*Invalid State* Ended-Compensated	*Invalid State* Ended
Completed	*Invalid State* Active	*Invalid State* Canceling	Completed	Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-*	*Invalid State* NotCompleting	*Invalid State* Exiting	*Invalid State* Ended-Canceled	*Invalid State* Ended-Closed	*Invalid State* Ended-Compensated	*Invalid State* Ended
Fail	Failing-Active	Failing-Canceling	Failing Completing	*Invalid State* Completed	*Invalid State* Closing	Failing-Compensating	Failing-*	*Invalid State* NotCompleting	*Invalid State* Exiting	*Invalid State* Ended-Canceled	*Invalid State* Ended-Closed	*Invalid State* Ended-Compensated	*Invalid State* Ended
CannotComplete	NotCompleting	*Invalid State* Canceling	NotCompleting	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-*	NotCompleting	*Invalid State* Exiting	*Invalid State* Ended-Canceled	*Invalid State* Ended-Closed	*Invalid State* Ended-Compensated	*Invalid State* Ended
Canceled	*Invalid State* Active	*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
Closed	*Invalid State* Active	*Invalid State* Canceling	*Invalid State* Completing	*Invalid State* Completed	*Forget* Ended-Closed	*Invalid State* Compensating	*Invalid State* Failing-*	*Invalid State* NotComleting	*Invalid State* Exiting	*Invalid State* Ended-Canceled	Ended-Closed	*Invalid State* Ended-Compensated	*Invalid State* Ended
Compensated	*Invalid State* Active	*Invalid State* Canceling	*Invalid State* Completing	*Invalid State* Completed	*Invalid State* Closing	*Forget* Ended-Compensated	*Invalid State* Failing-*	*Invalid State* NotCompleting	*Invalid State* Exiting	*Invalid State* Ended-Canceled	*Invalid State* Ended-Closed	Ended-Compensated	*Invalid State* Ended

Enhanced 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-Failed**	**Ended-Exited**	**EndedNotCompleted**	**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	*Ignore* Ended-Failed	*Resend Exited* Ended-Exited	*Ignore* Ended-NotCompleted	*Ignore* 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-Failed	*Ignore* Ended-Exited	*Ignore* Ended-NotCompleted	*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-Failed	*Ignore* Ended-Exited	*Ignore* Ended-NotCompleted	*Ignore* 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	*Ignore* Ended-Failed	*Ignore* Ended-Exited	*Resend NotCompleted* Ended-NotCompleted	*Ignore* 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-Failed	*Ignore* Ended-Exited	*Ignore* Ended-NotCompleted	*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-Failed	*Ignore* Ended-Exited	*Ignore* Ended-NotCompleted	*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-Failed	*Ignore* Ended-Exited	*Ignore* Ended-NotCompleted	*Ignore* 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
Compens-ate	*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* 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* Ended-NotCompleted	*Invalid State* Exiting	*Invalid State* Ended-Failed	*Invalid State* Ended-Exited	Ended-NotCompleted	*Invalid State* Ended

BusinessAgreementWithParticipantCompletion protocol
(Participant View)

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

BusinessAgreementWithParticipantCompletion protocol (Participant View)											
Outbound Events	**States**										
	Active	**Canceling**	**Completed**	**Closing**	**Compensating**	**Failing-Active**	**Failing-Canceling**	**Failing-Compesating**	**NotCompleting**	**Exiting**	**Ended**
Exit	Exiting	*Invalid State* Canceling	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-Active	*Invalid State* Failing-Canceling	*Invalid State* Failing-Compensating	*Invalid State* NotCompleting	Exiting	*Invalid State* Ended
Completed	Completed	*Invalid State* Canceling	Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-Active	*Invalid State* Failing-Canceling	*Invalid State* Failing-Compensating	*Invalid State* NotCompleting	*Invalid State* Exiting	*Invalid State* Ended
Fail	Failing-Active	Failing-Canceling	*Invalid State* Completed	*Invalid State* Closing	Failing-Compensating	Failing-Active	Failing-Canceling	Failing-Compensating	*Invalid State* NotCompleting	*Invalid State* Exiting	*Invalid State* Ended
CannotComplete	NotCompleting	*Invalid State* Canceling	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-Active	*Invalid State* Failing-Canceling	*Invalid State* Failing-Compensating	NotCompleting	*Invalid State* Exiting	*Invalid State* Ended
Canceled	*Invalid State* Active	*Forget* Ended	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-Active	*Invalid State* Failing-Canceling	*Invalid State* Failing-Compensating	*Invalid State* NotCompleting	*Invalid State* Exiting	Ended
Closed	*Invalid State* Active	*Invalid State* Canceling	*Invalid State* Completed	*Forget* Ended	*Invalid State* Compensating	*Invalid State* Failing-Active	*Invalid State* Failing-Canceling	*Invalid State* Failing-Compensating	*Invalid State* NotCompleting	*Invalid State* Exiting	Ended
Compensated	*Invalid State* Active	*Invalid State* Canceling	*Invalid State* Completed	*Invalid State* Closing	*Forget* Ended	*Invalid State* Failing-Active	*Invalid State* Failing-Canceling	*Invalid State* Failing-Compensating	*Invalid State* NotCompleting	*Invalid State* Exiting	Ended

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

BusinessAgreementWithParticipantCompletion protocol (Coordinator View)											
Outbound Events	**States**										
	Active	**Canceling**	**Completed**	**Closing**	**Compensating**	**Failing-Active**	**Failing-Canceling**	**Failing-Compesating**	**NotCompleting**	**Exiting**	**Ended**
Cancel	Canceling	Canceling	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-Active	*Invalid State* Failing-Canceling	*Invalid State* Failing-Compensating	*Invalid State* NotCompleting	*Invalid State* Exiting	*Invalid State* Ended
Close	*Invalid State* Active	*Invalid State* Canceling	Closing	Closing	*Invalid State* Compensating	*Invalid State* Failing-Active	*Invalid State* Failing-Canceling	*Invalid State* Failing-Compensating	*Invalid State* NotCompleting	*Invalid State* Exiting	*Invalid State* Ended
Compensate	*Invalid State* Active	*Invalid State* Canceling	Compensating	*Invalid State* Closing	Compensating	*Invalid State* Failing-Active	*Invalid State* Failing-Canceling	*Invalid State* Failing-Compensating	*Invalid State* NotCompleting	*Invalid State* Exiting	*Invalid State* Ended
Failed	*Invalid State* Active	*Invalid State* Canceling	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Forget* Ended	*Forget* Ended	*Forget* Ended	*Invalid State* NotCompleting	*Invalid State* Exiting	Ended
Exited	*Invalid State* Active	*Invalid State* Canceling	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-Active	*Invalid State* Failing-Canceling	*Invalid State* Failing-Compensating	*Invalid State* NotCompleting	*Forget* Ended	Ended
NotCompleted	*Invalid State* Active	*Invalid State* Canceling	*Invalid State* Completed	*Invalid State* Closing	*Invalid State* Compensating	*Invalid State* Failing-Active	*Invalid State* Failing-Canceling	*Invalid State* Failing-Compensating	*Forget* Ended	*Invalid State* Exiting	Ended

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

EnhancedBusinessAgreementWithParticipantCompletion protocol (Participant View)

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

EnhancedBusinessAgreementWithParticipantCompletion protocol
(Coordinator View)

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

EnhancedBusinessAgreementWithParticipantCompletion protocol (Coordinator View)														
Outbound Events	**States**													
	Active	**Canceling**	**Completed**	**Closing**	**Compensating**	**Failing-Active**	**Failing-Canceling**	**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*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*
	Canceling	Canceling	Completed	Closing	Compensating	Failing-Active	Failing-Canceling	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*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*
	Active	Canceling	Closing	Closing	Compensating	Failing-Active	Failing-Canceling	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*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*
	Active	Canceling	Compensating	Closing	Compensating	Failing-Active	Failing-Canceling	Failing-Compensating	NotCompleting	Exiting	Ended-Failed	Ended-Exited	Ended-NotCompleted	Ended
Failed	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Forget*	*Forget*	*Forget*	*Invalid State*	*Invalid State*		*Invalid State*	*Invalid State*	*Invalid State*
	Active	Canceling	Completed	Closing	Compensating	Ended-Failed	Ended-Failed	Ended-Failed	NotCompleting	Exiting	Ended-Failed	Ended-Exited	Ended-NotCompleted	Ended
Exited	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Forget*	*Invalid State*		*Invalid State*	*Invalid State*
	Active	Canceling	Completed	Closing	Compensating	Failing-Active	Failing-Canceling	Failing-Compensating	NotCompleting	Ended-Exited	Ended-Failed	Ended-Exited	Ended-NotCompleted	Ended
NotCompleted	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Invalid State*	*Forget*	*Invalid State*	*Invalid State*	*Invalid State*		*Invalid State*
	Active	Canceling	Completed	Closing	Compensating	Failing-Active	Failing-Canceling	Failing-Compensating	Ended-NotCompelted	Exiting	Ended-Failed	Ended-Exited	Ended-NotCompleted	Ended