BAMBERGER BEITRÄGE

ZUR WIRTSCHAFTSINFORMATIK UND ANGEWANDTEN INFORMATIK

ISSN 0937-3349

Nr. 65

# Modelling and Validating Business Collaborations: A Case Study on RosettaNet

Andreas Schönberger

March 2006

FAKULTÄT WIRTSCHAFTSINFORMATIK UND ANGEWANDTE INFORMATIK

OTTO-FRIEDRICH-UNIVERSITÄT BAMBERG

Distributed and Mobile Systems Group

Otto-Friedrich Universität Bamberg

Feldkirchenstr. 21, 96052 Bamberg, GERMANY

Prof. Dr. rer. nat. Guido Wirtz

http://www.uni-bamberg.de/en/fakultaeten/wiai/faecher/informatik/lspi/

Due to hardware developments, strong application needs and the overwhelming influence of the net in almost all areas, distributed and mobile systems, especially software systems, have become one of the most important topics for nowadays software industry. Unfortunately, distribution adds its share to the problems of developing complex software systems. Heterogeneity in both, hardware and software, concurrency, distribution of components and the need for interoperability between different systems complicate matters. Moreover, new technical aspects like resource management, load balancing and deadlock handling put an additional burden onto the developer. Although subject to permanent changes, distributed systems have high requirements w.r.t. dependability, robustness and performance.

The long-term common goal of our research efforts is the development, implementation and evaluation of methods helpful for the development of robust and easy-to-use software for complex systems in general while putting a focus on the problems and issues regarding the software development for distributed as well as mobile systems on all levels. Our current research activities are focussed on different aspects centered around that theme:

- *Robust and adaptive Service-oriented Architectures*: Development of design methods, languages and middleware to ease the development of SOAs with an emphasis on provable correct systems that allow for early design-evaluation due to rigorous development methods and tools. Additionally, we work on approaches to autonomic components and container-support for such components in order to ensure robustness also at runtime.
- *Agent and Multi-Agent (MAS) Technology:* Development of new approaches to use Multi-Agent-Systems and negotiation techniques, for designing, organizing and optimizing complex distributed systems, esp. service-based architectures.
- *Peer-to-Peer Systems*: Development of algorithms, techniques and middleware suitable for building applications based on unstructured as well as structured P2P systems. A specific focus is put on privacy as well as anonymity issues.
- *Context-Models and Context-Support for small mobile devices*: Investigation of techniques for providing, representing and exchanging context information in networks of small mobile devices like, e.g. PDAs or smart phones. The focus is on the development of a truly distributed context model taking care of information reliability as well as privacy issues.
- *Visual Programming- and Design-Languages*: The goal of this long-term effort is the utilitization of visual metaphores and languages as well as visualization techniques to make design- and programming languages more understandable and, hence, easy-to-use.

More information about our work, i.e., projects, papers and software, is available at our homepage. If you have any questions or suggestions regarding this report or our work in general, don't hesitate to contact me at guido.wirtz@wiai.uni-bamberg.de

Guido Wirtz

Bamberg, April 2006

# Modelling and Validating Business Collaborations: A Case Study on RosettaNet

Andreas Schönberger

Lehrstuhl für Praktische Informatik, Fakultät WIAI

**Abstract** The way business processes are organised heavily influences the flexibility and the expenses of enterprises. The capability to address changing market needs in a timely manner and to provide appropriate pricing is indispensable in a world of internationalisation and growing competition. Optimising processes that cross enterprise boundaries potentially is a key success factor in achieving this goal but it requires the information systems of the participating enterprises to be consistently integrated. This gives rise to some challenging tasks. The personnel involved in building up business collaborations comes from different enterprises with different business vocabulary and background which requires extensive communication support. The lack of central technical infrastructure, typically prohibited by business politics, calls for a truly distributed and computer-aided collaboration structure, so that the resulting complexity must be handled somehow. Nevertheless robustness is an important factor in building business collaborations as these may exchange goods of considerable value.
This technical report proposes the use of a two step modelling approach that separates business logic, modelled in the so-called *centralised perspective* (CP), from its distributed implementation, modelled in the so-called *distributed perspective* (DP). The separation of these perspectives enables business people to concentrate on business issues and to solve communication problems in the CP whereas technical staff can concentrate on distribution issues. The use of stringent modelling rules is advised in order to provide the basis for formal analysis techniques as one means to achieve robustness.
Considering the choreography of RosettaNet Partner Interface Processes (PIPs) as the subject of analysis, UML activity diagrams for modelling the CP and WSBPEL for modelling the DP are described as enabling techniques for implementing the proposed two step modelling approach. Further, model checking is applied to validate the CP and DP models in order to detect errors in early design phases. As the adequacy of model checking tools highly depends on the detailed modelling techniques as well as the properties to be checked, a major part of our discussion covers relevant properties and requirements for a model checker.

**Keywords** validation, verification, UML, business process modelling, distributed system, business to business integration, distributed consensus, RosettaNet, WSBPEL

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| 2PC | Two-Phase-Commit Protocol |
| BOV | Business Operational View |
| CP | Centralised Perspective |
| CTL | Computation Tree Logic |
| DP | Distributed Perspective |
| FSV | Functional Service View |
| IFV | Implementation Framework View |
| MC | Micro-choreography |
| MCP | Media Control Protocol |
| OIU | Order Information User |
| PIP | Partner Interface Process |
| PIPXP | PIP Execution Protocol |
| PLTL | Propositional Linear Time Logic |
| RTC | Real Time Clock |
| UML | Unified Modeling Language |
| WSBPEL | Web Services Business Process Execution Language |

# 1 Introduction

Business processes define how enterprises produce and exchange goods and services. The flow of information is the main leverage for controlling business processes and heavily influences the flexibility and the expenses of enterprises. One example for being flexible is giving the customer the possibility to configure the product himself, e.g. choosing the colour of product parts or choosing add-ons that provide more functionality. But this requires enterprises to quickly integrate customer preferences in the current production schedule, i.e. the flow of information must be well-designed and well-managed. One example for reducing costs is minimising the time until goods and services are billed which again requires a well-designed and well-managed flow of information. Enterprises can use flexibility and minimal expenses to realise strategies like *providing a high customer value* or *offering the best price*. Hence, optimising business processes is a core technique for enterprises to continue to exist in stiff competition. Optimising business processes can be done by automating the flow of information. Automating the flow of information of business collaborations between enterprises, i.e. optimising cross-enterprise business processes, still has unexploited potential to furnish competitive advantages but gives rise to hard challenges as well.
The personnel involved in building up business collaborations frequently comes from different enterprises with different business vocabulary and background which requires extensive communication support. The lack of central technical infrastructure, typically prohibited by business politics, demands for truly distributed computation so that the resulting complexity must be handled somehow. Nevertheless, robustness is an important factor in building business collaborations as these may exchange goods of considerable value. Therefore we propose the following approach to building up business collaborations:

1. **Modelling a business collaboration from a centralised perspective.** Communication between personnel from different enterprises can be supported by first focussing on the business logic of the collaboration. The so-called *centralised perspective* (CP) specifies the abstract business state of the collaboration, the events that trigger state changes, so-called micro-choreographies that consistently perform state changes and the control flow of the collaboration. Interpreting the CP as the common view of all collaboration participants on the collaboration progress is key to understanding the CP. Thus the state under consideration reduces to the facts the collaboration participants have to agree upon and micro-choreographies can be interpreted as single actions that change state leaving out message passing details. This modelling metaphor makes modelling quite simple and does not require technical experts, who possibly don't know business logic very well, to create the model. Nevertheless, the modelling technique applied should have clear semantics to avoid misunderstandings between collaboration participants and to provide the foundation of (semi-)automated generation of a distributed implementation. The model of the CP then gives context to a distributed implementation of the collaboration which is a similar approach as pursued by WS-CAF/WS-Context ([OAS03]).

2. **Modelling a business collaboration from a distributed perspective.** The so-called *distributed perspective* (DP) models the implementation of the CP in a distributed environment by specifying a representation of the abstract business state and by specifying protocols for performing micro-choreographies that ensure distributed consensus.

Complexity is handled in a twofold way. First, the global view on business logic is already fixed when modelling the DP, so technical aspects can be focussed. Second, the concept of micro-choreography helps in unitising the implementation model, which decomposes the overall task. Further, tasks to be fulfilled on the DP are achieving agreement on the start of micro-choreographies, proving that the DP conforms to the CP and integrating local business politics of the collaboration participants.

3. **Applying model checking techniques to ensure robustness.** Business collaborations possibly exchange goods and services of considerable value. Therefore, a robust design is needed in order to avoid losses. This means that the protocol to perform a collaboration should ideally be defined in such a way that errors are only possible by violating the protocol thus identifying a responsible person to call to account for losses in case an error occurs. This goal requires to look at all possible protocol runs for a given environment. As the set of possible runs in a concurrent environment grows quickly, time-consuming and error-prone manual analysis should be extended by automated analysis techniques such as model checking. Model checkers compute all possible runs of a given protocol and offer the possibility to check properties. Model checking is suited to perform validation in early design phases because models are relatively small and undetected errors cause high costs. Major problems in applying model checking techniques are the identification of relevant properties to check and the choice of the right model checker. Therefore this technical report gives a taxonomy of relevant properties of business collaborations and identifies requirements for model checkers. A core requirement for applying model checking is that the model checker in use can be applied directly to the models of the CP and DP or, at least, to enhanced models of the CP and DP because having to model the collaboration twice would be too costly.

In practice, the approach proposed must be supported by suitable technologies and tools. As the relationships between enterprises are numerous, i.e. enterprises have many partners, and dynamic, i.e. enterprises acquire and lose partners, these technologies and tools should be based on standards. Standards support quick and low-cost automation of business collaborations. We propose UML activity diagrams ([OMG03]) and WSBPEL ([IBM03]) as enabling, but not exclusive, technologies for modelling the CP and the DP respectively. UML activity diagrams offer a visual notation which supports the communication functionality of the CP. WSBPEL and in particular Web Services are platform and programming language independent which is important for connecting heterogeneous systems. Both, UML activity diagrams and WSBPEL, rely on standards.
A case study on choreographing RosettaNet *Partner Interface Processes* (PIPs) is conducted to evaluate the approach proposed in conjunction with UML activity diagrams and WSBPEL. RosettaNet suits well as the subject of our case study because it is a standard itself and a major part of the RosettaNet standard is devoted to standardising message contents of business collaborations, an important task that is not addressed by our approach. The use of model checkers, namely TCM/TATD[1] for validating the CP and SPIN[2] for validating parts of the DP, is shown by applying them to the models of the case study. The use of these tools is not shown in a general way because model checking depends on the detailed modelling techniques

[1]`http://wwwhome.cs.utwente.nl/~tcm/tatd.html`
[2]`http://spinroot.com/spin/whatispin.html`

as well as the properties to be checked. A more important result of our work is therefore the creation of a taxonomy of relevant properties for business collaborations and the identification of requirements for a model checker.

This technical report is structured as follows. Section 2 gives a short introduction to the technologies applied. Section 3 explains our modelling approach in detail and shows how UML activity diagrams and WSBPEL can be used for modelling the two complementary perspectives. Section 4 introduces the use case of the case study and describes the modelling of the use case in detail. In section 5 the modelling results are validated and a taxonomy of properties as well as core requirements for a model checker are given. Section 6 then discusses related work. Finally, section 7 concludes the paper and identifies future work.

# 2 Basics

The technologies and concepts used in this report range from *modelling systems* in general, *UML 1.5 activity diagrams* ([OMG03]), *Web Services*[3] and *WSBPEL* ([IBM03]) to *model checking* as well as *RosettaNet*[4]. A comprehensive introduction to all of these is out of scope of this report. Therefore, only a short introduction to *model checking* and *RosettaNet* is provided here.

## 2.1 Model Checking

Model checking is a technique for validating finite systems. Finiteness is a requirement because, in general, the whole state space is being explored for validation purposes. Usually, temporal logic is used to formulate properties that are then proved to be true or not. Temporal logic is particularly useful to express relationships between multiple states of the state space under consideration. The atoms of temporal logic formulae refer to single states of a state space so that, normally, their formalisation depends on the type of system to be validated.
A model checker is a tool for applying model checking techniques to particular types of systems. Model checkers differ in input language for specification of systems, in query language for formalisation of properties and in algorithms for verifying properties. The main functionality of a model checker is the application of verification algorithms as well as the translation of input specifications and properties into a structure suitable for verification. Input and query language are crucial for intuitive use of a model checker. Frequently, a system must be translated to the input language of a model checker before it can be validated, so intuitive use is important. Ideally, the paradigm of the input language fits the system under consideration. For example, the widely-used model checker SPIN uses the paradigm of concurrent processes that exchange messages via message channels.
Model checkers are frequently used for validating concurrent systems. As the components of a concurrent system are asynchronously executed, the state space of such systems usually increases exponentially in the number of components and states of components. This necessitates the use of automated methods for analysis. Unfortunately, the validation of systems often fails because available validation resources, in terms of CPU cycles and main memory, do not suffice for the huge state space of concurrent systems. This problem is also known as the *state explosion problem*. If the state space of a system is too huge or not finite then abstraction mechanisms can be applied.
To do so, a system S is translated into an abstract system S' that typically has a smaller state space. Then S' is validated instead of S and the validation results are transferred to S. Figure 1 shows an abstraction example (taken from [BBF01], p.111). The number of states in the left system have been reduced by collapsing the states in a dotted area to one state respectively. The transitions of S have been transferred to S' according to the following rule: If a transition starts from/ends in a state in S than it starts from/ends in its corresponding collapsed state in S'. The result of this abstraction is shown on the right side of the figure. S' has a smaller state space than S but it allows for more system runs. Apparently, not all properties that hold for S' also hold for S. For example, it can be shown that, only using A transitions, every state in

[3]`http://www.w3.org/2002/ws/`
[4]`http://www.rosettanet.org/`

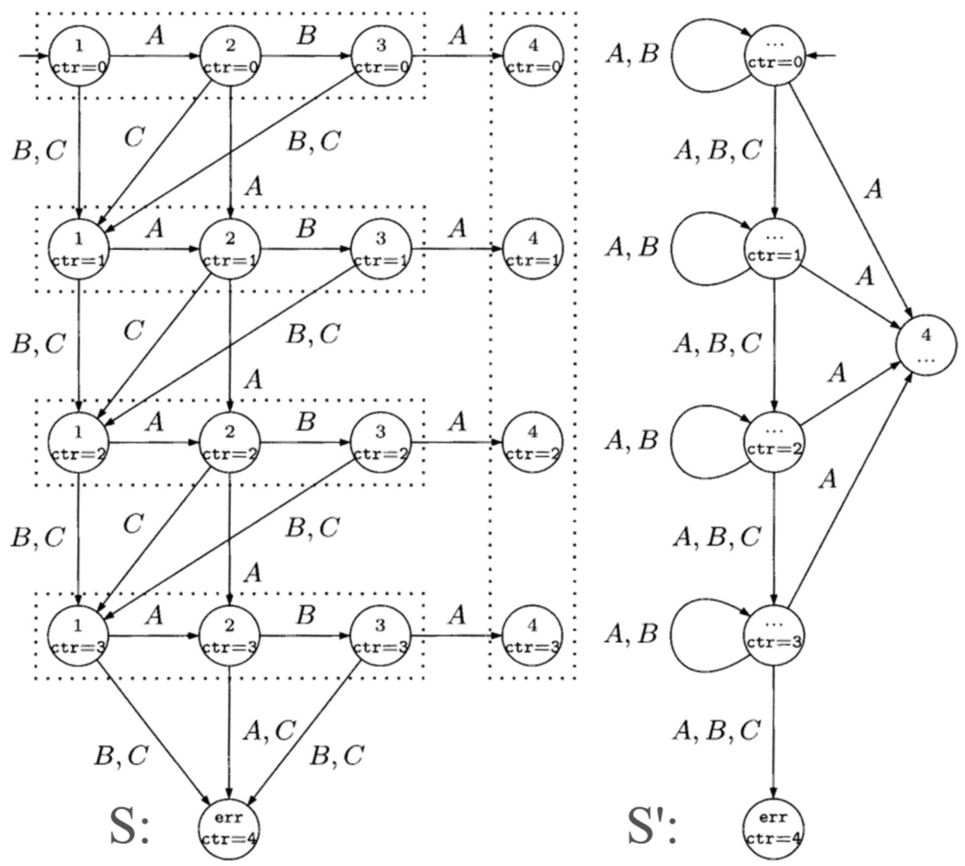

Figure 1: Abstraction of system S to S'

S' is reachable. This property does not hold for S.
In order to decide whether a property can safely be transferred or not, the distinction between *safety properties* and *liveness properties* is useful.
Informally speaking, a safety property expresses that a particular state will never be reached whereas a liveness property expresses that a particular state will be reached if some conditions are met. The classification is easier to make following the rule in [BBF01], p.84, for identifying safety properties:

> [...]when a safety property is violated, it should be possible to instantly notice it. If this depends on the remainder of the behaviour, then a behaviour which would end immediately would not violate the safety property. Hence we can only notice it, in the current state, relying on events which occurred earlier[...]

Safety properties that hold for S' also hold for S. The reason is that S' allows strictly more system runs than S does. This can be shown by the fact that any transition that can be taken in S can also be taken in S'. The transfer of safety properties from S' to S is only correct, but not complete. If a safety property holds in S' it also holds in S, but if a safety property does not hold in S' then it is not necessarily the case that it does not hold in S as well. Generally, the transfer of liveness properties is not admissible.
In this report, only abstractions that allow for strictly more system runs are used so that the transfer of properties follows the rules just described.

## 2.2 RosettaNet

RosettaNet is a non-profit standards organisation dedicated to supporting B2B integration and endorsed by over 500 companies worldwide. Founded in 1998, RosettaNet defines business messages and rules for its electronic exchange. Therefore RosettaNet uses technology and ideas from Open-edi ([ISO04]), UN/CEFACT Modeling Methodology (UMM, [UN/01]) and ebXML[5]. The core RosettaNet standards are Partner Interface processes (PIPs) and the RosettaNet Implementation Framework (RNIF) [Ros02], [Dam04]. PIPs, classified in clusters like cluster *3 Order Management* and segments like segment *3A Quote and Order Entry*, describe the application context, the content and parameters for the electronic exchange of one or two business documents. The RNIF in turn provides a metamodel for PIPs and details the technology for their execution.
PIPs describe the exchange of business documents at three levels, namely the *Business Operational View* (BOV), the *Functional Service View* (FSV) and the *Implementation Framework View* (IFV).
The BOV describes a PIP from a business perspective. This includes an informal textual description of the application context of the PIP and an UML activity diagram visualizing the PIP. In that diagram, the roles of the business partners involved are represented by swimlanes. The first activity of such a diagram is stereotyped with a *Business Transaction Type* according to UMM ([UN/01] chapter 1, p.14 f.)). Figure 2 shows an example of such a diagram that further visualises business documents to be exchanged as object flows. Finally, the BOV specifies start and end states of a PIP execution and *Business Process Activity Controls* like *Time to Perform* for the overall PIP.
For each role of the BOV a component is defined in the FSV that is responsible for exchanging

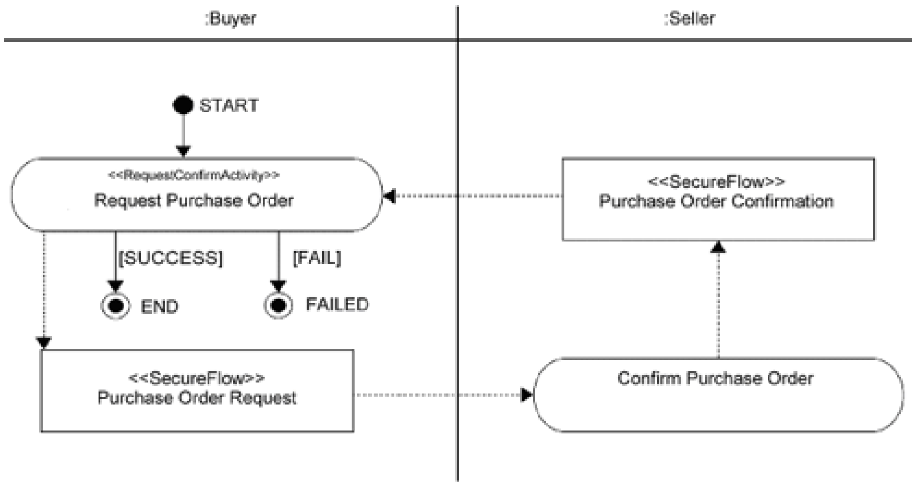

Figure 2: UML activity diagram visualising the BOV of a PIP

business documents as *Actions* and control messages as *Signals*. The exchange of each message is detailed by *Message Exchange Controls* and finally the intended order of message exchanges is represented by an UML sequence diagram.
The main task of the IFV is the detailed specification of the business documents to exchange which is done in a xsd-file. Moreover the IFV specifies encryption details of the messages to be

[5]`http://www.ebxml.org/`

exchanged.
Apart from defining a metamodel for PIPs, a major part of the RNIF is devoted to specifying a reliable protocol for exchanging the messages of a PIP. This is different from the information in the FSV where only the idealised flow of messages is given. RNIF defines four variants of message exchange protocols as *Business Message Patterns* ([Ros02] p.75 ff) that can be used to type a PIP.

# 3 An approach for modelling business collaborations

As already pointed out in the introduction, automating business collaborations needs modelling support for communication purposes. Models provide this support by defining a unified system of concepts and sometimes even semantics for these concepts. Models are further useful for accomplishing analysis, design and documentation tasks. Subsection 3.1 introduces the two-step modelling approach referred to in the introduction. Subsections 3.2 and 3.3 show how UML activity diagrams and WSBPEL can be applied to modelling the CP and DP respectively.

## 3.1 The core approach

The idea for separating the CP and the DP of a business collaboration bases on the insight that collaboration participants first have to agree upon what to do in a collaboration from a global point of view before specifying a collaboration from a local point of view. A business collaboration can be interpreted as a single business process that spans multiple enterprises. The purpose of the CP is to model this business process by identifying relevant states and actions. The purpose of the DP is to specify the distributed implementation of the overall business process.

### 3.1.1 The centralised perspective

The definition of state is crucial for modelling the CP on business collaborations. The goal of any business collaboration is achieving a new and common state, e.g. signing a new contract. Moreover the applicability of transactions and the way transactions are conducted within a collaboration may depend on what steps have been taken before. The exchange of an order of goods may require a quote to be exchanged beforehand and refer to prices that are fixed in the quote. Interpreting a collaboration as a single business process, we decided *to model state explicitly in our approach and to define it as the common view of the collaboration partners on the progress of the collaboration* (process state in the following). A process state is composed of the relevant attributes of the collaboration, e.g. the information if a contract has already been signed or if certain resources are free or not. Process states are therefore the basis for communication between collaboration participants who negotiate which process states should be reached and which attribute values should be associated with these process states. Clearly, it is impossible in a truly distributed environment to ensure that the collaboration participants have the very same view on process states at every point in time. So it is more adequate to think of a process state as an abstract business state. The collaboration participants always reside in the same business state or if any participant has changed its state then all other participants must make a change to the same state in finite time and no further state changes are allowed until all participants have reached that state. Such a semantics can be achieved by using distributed consensus mechanisms. An alternative approach is to let the views of the collaboration participants on progress diverge but then the number of process states to model would possibly grow exponentially. That is why the use of so-called *micro-choreographies*[6] is proposed to consistently change process states by means of two-phase-commit protocols (2PC in the following). If no communication is possible at all the use of so-called *distributed time-outs* is envisaged to make state changes, e.g. for releasing valuable resources of a participant. A distributed time-out is only allowed if communication was successful beforehand, e.g. it can be agreed upon the reservation time of a resource reservation while agreeing upon the reservation itself. The mechanism to negotiate a distributed time-out while performing a 2PC can be found in the following excursus.

**Excursus: Distributed time-out after 2PC.** This excursus presents the negotiation of a distributed time-out during the execution of a 2PC run. A distributed time-out can then be fired without any further communication.
The discussion of this excursus concentrates on when the collaboration participants have to activate timers that fire the time-out events. Therefore, figure 3 shows an idealised run of 2PC. The labels used in figure 3 are used in the following discussion where labels *tX* represent points in time and labels *VR*, *VC*, *GC* and *ACK* represent the messages exchanged. The object instances *A* and *B* represent two participants of a 2PC run. To demonstrate the viability of negotiating a distributed time-out during a 2PC run, two scenarios are analysed:

- **Scenario 1.** In this scenario, B reserves resources in favour of A. B uses a distributed time-out to release these resources after a certain amount of time. But in any case, it should be provided that A can have faith in the reservation by B until A triggers a time-out event himself. Therefore B has to trigger a time-out event after A does. This can be accomplished by triggering the timer of A earlier than the timer of B.
  The first step in doing so is that B reserves resources immediately before t3. If the 2PC run ends with a result of *Abort* afterwards, B releases the resources and no distributed time-out is needed at all. Otherwise, A activates its timer immediately before t5 whereas B activates its timer directly after t6. Let the duration of the timers be *d*. Then the time-out event of A is triggered at *t5 + d* at the latest and the time-out event of B is triggered at *t6 + d* at the earliest. As t5 necessarily precedes t6, the time-out event of A is triggered earlier than the time-out event of B if the clock drift of both is acceptable (see below). Hence, the goal is achieved.
  Clearly, message losses must be taken into account. If VR and VC messages are lost, the emerging situations can easily be handled by aborting the 2PC run. If GC or ACK messages are lost, the situation is more complex.
  The loss of such messages are detected by A by not receiving any ACK messages. In this case A repeatedly sends GC messages until he either receives an ACK message or a 2PC timer (not to confuse with the timer for distributed time-outs), activated by A before sending the first ACK message, has run out. In either case, A then proceeds in the business collaboration. Even if the 2PC timer mentioned runs out, A can have faith in the reservation of the resources because B reserved these before t3.
  B detects the loss of GC or ACK messages by either never receiving a GC message or by receiving multiple GC messages. If B receives no GC messages at all in a certain time

[6]The notion of *micro-choreography* is inspired by the fact that collaboration participants have to exchange a set of messages according to some strict rules for implementing consistent changes of process states.

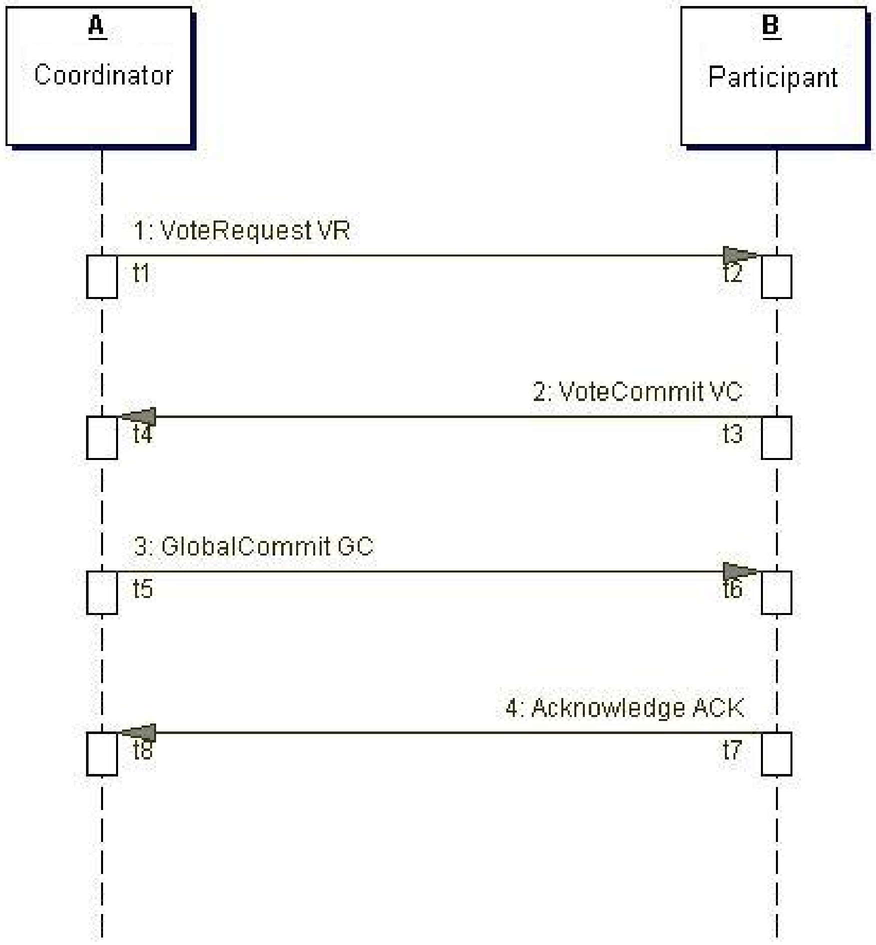

Figure 3: Idealised run of 2PC protocol

period, manual intervention is needed to resolve the state of A. Duplicate GC messages can be answered by B with ACK messages but this is not necessary because A can proceed without ACK messages as well.
If ACK messages are lost A can only proceed after a 2PC timer has run out. In the meantime, B could already have need in performing further micro-choreographies. In order to avoid unnecessary delays in the execution of the collaboration, B can stop A from waiting for ACK messages by using *piggy-backing*. Therefore B adds an ACK message to the first message of the new micro-choreography. If A receives the message of the new micro-choreography it receives the missing ACK message as well and can terminate the 2PC run.

- **Scenario 2.** In this scenario A reserves resources for B. As this a symmetric case to scenario 1, the timer of B must be activated earlier than the timer of A.
In scenario 2, the reservation of resources is done by A immediately before t5. If the reservation is not successful A aborts the 2PC run, otherwise *Commit* is the only possible result of the 2PC run. B starts its timer immediately before t3 whereas A starts its timer after successfully having reserved the resources. This means A starts its timer after t4.

Let *d* be the duration of the timers. Then, the time-out event of B is triggered at *t3 + d* at the latest whereas the time-out event of A is triggered at *t4 + d* at the earliest. As t3 necessarily precedes t4 the time-out event of B is fired earlier than the time-out event of A if the clock drift of A and B is acceptable (see below). Hence, the goal is achieved. Clearly, the timer of B can only tentatively be activated and must be deactivated if the 2PC run terminates with a result of *Abort*. The tentative activation of the timer of B does not mean B can begin new micro-choreographies assuming that resources are reserved while the reservation by A fails later on. This situation to happen would require that B begins a new micro-choreography before the current 2PC run is terminated. To do so is not allowed.
The correctness of the mechanism for scenario 2 in the case of message losses can be confirmed by looking at the properties of 2PC because 2PC already considers message losses. If A and B detect an *Abort* result of the 2PC run, then either no timers for distributed time-out have been activated or the activation has been cancelled. No resources are then reserved either. If A detects a *Commit* result and B blocks continuously (because GC messages get lost), then B cannot use the resources reserved. But this is not a problem because the collaboration participants must ensure that message losses are not the standard case. If A and B both detect a *Commit* result it only has to be considered that B must activate its timer before he sends the first (of possibly multiple) VC messages. As the point in time at which the first VC message is sent precedes the point in time at which the first VC message is received, the time-out event of B is fired earlier as described above. Further cases do not exist.

It is noteworthy that the mechanism described here does not require the clocks of the participants to be synchronised. This is because the activation of timers relies on time intervals and on the order of message dispatch and message receipt. A problem to be handled is clock drift, i.e. the clock of A could run faster than the clock of B or the other way round. This problem can be solved by multiplying the timer duration of the participant who reserves resources for the other participant by a factor of $\lambda$. The value of $\lambda$ can be deduced by looking at the maximum clock deviation of customary *Real Time Clocks* (RTC in the following). According to manufacturers of RTCs, the clock deviation ranges between 3 ppm approx. and 100 ppm approx. depending on the particular RTC product. To safely calculate $\lambda$, a clock deviation of 300 ppm approx. is assumed. This means that a RTC runs 0.0003 * 3600 = 1.08 seconds per hour too fast or too slow as maximum. This means that the clock drift of A and B can be at most approx. 2 seconds per hour. Hence 1.0006 should be a safe value for $\lambda$. The collaboration participants have to ensure that the clock deviation assumed is not exceeded. Maybe time buffers have to be introduced for handling clock deviation excesses.
One could criticise that the proposed mechanism for realising distributed time-outs needs manual intervention in certain situations. But these situations are fairly rare assuming reasonable investment in communication infrastructure and 2PC is used here at the level of business processes. Manual intervention can therefore be justified.

**end of excursus**

Having defined process states and micro-choreographies as core concepts of the CP, the relation of these concepts has to be clarified. A process state cannot be directly changed into another process state. Either a global event, i.e. a distributed time-out, or a local event that triggers a micro-choreography must be used to leave a process state. A distributed time-out must have been agreed upon before by means of distributed consensus and always points to exactly one other process state. Local events can be detected locally, but after detection of a local event the execution of a micro-choreography must be negotiated by means of a protocol because another participant might have detected a local event as well. The execution of a micro-choreography can lead to multiple process states depending on the content of the business messages that have to be exchanged during a micro-choreography. It is important to note that no data other than the content of the business messages that has been agreed upon by 2PC may be used to route between multiple process states. It is reasonable to divide the result of a micro-choreography in a technical result and a business result to perform the actual routing. The technical result is a boolean value that is true whenever all relevant business messages of a micro-choreography have successfully been exchanged, interpreted and agreed upon by 2PC. It is not sensible to represent a business result as either *SUCCESS* or *FAILURE* because it is not really clear if a cancellation of an order is a success or a failure nor is the set of possible business results always restricted to two values, e.g. in a case where an offer may be accepted, rejected or be decided upon at a later point in time. Finally, it is useful to introduce roles in the CP in order to prevent the modellers from specifying processes that are extraordinarily hard to implement. Roles can be used to identify the 2PC coordinator of micro-choreographies, the participant in charge for detecting local events that trigger micro-choreographies as well as to make clear whose resources are released by means of distributed time-outs.

### 3.1.2 The distributed perspective

While the distributed environment of business collaborations can be largely ignored in the CP, the DP must respect the restrictions of distributed computing. As business collaborations usually cannot be built on a central technical infrastructure, automation based on the *shared memory* paradigm is hardly possible. Considering insecure communication media, e.g. communication over the internet, the *message passing* paradigm is adequate for automating business collaborations. The following constraints must be addressed.

- There are no assumptions about how long a message travels from sender to receiver.
- Messages can overtake each other.
- Messages can be lost or be duplicated.
- Finally the clocks of the collaboration participants must be assumed to be not synchronised.

As already mentioned in the discussion of the CP, such an environment prevents the collaboration participants from having exactly the same view on collaboration progress at every point in time. But it can be made sure that any local process of a participant is at most one step behind the local process of another participant by using the well-known 2PC. The detailed protocols for specifying the implementation of the collaboration, in particular the implementation of micro-choreographies, depend to some extent on the application domain and on the particular use case. This subsection therefore discusses the tasks for specifying the DP and gives detailed protocols only where appropriate. Generally speaking, the task of the DP is to represent the process states of a collaboration and to specify the number, types and order of messages to be exchanged. This task can be decomposed into the following packages.

**Representing process states.** Process states are abstract business states. Thus the current process state can be represented by a single variable of an enumeration type, that contains all possible process states. The attributes of a process state can then be deduced by relating the value of the variable to a process state in the CP.

**Executing micro-choreographies.** The messages to exchange within a micro-choreography (MC) depend on the state changes that should be performed. Arbitrary business documents can be exchanged within a MC. No matter what business documents are exchanged, at the end of a MC a 2PC run has to be performed in order to achieve distributed consensus whether all business documents have successfully been transmitted and interpreted or not. The coordinator of the 2PC run can be determined by choosing the participant who has received the last business document. Typically, all business messages then successfully have been exchanged and interpreted and the only task of the 2PC is to agree upon that. The result of such a 2PC run then always will be *Commit*, except communication failures prevent the participants from concluding such a result. In order to avoid blocking processes, manual intervention is needed in the typical *2PC blocking situation*. The notification of such a blocking situation then must be performed reliably which can be safely done because the notification can be performed locally. More detailed treatment can be found in literature (e.g. [TS02], section 7.5.1). Throughout this technical report the protocol for executing MCs will be called *MC execution protocol*.

**Triggering the execution of micro-choreographies.** A MC is usually triggered by the collaboration participant who sends the first message of the MC. In some process states there may be multiple MCs that can be triggered, maybe by different collaboration participants. As the execution of a MC may lead to new process states where other MCs can be triggered the participant with the privilege to trigger a MC must be determined by a protocol. Such a protocol must also take into account that distributed time-outs can lead to process state changes and thus affect the applicability of MCs. Ideally, such a protocol negotiates an instance number for the execution of the MC as well. The so-called *media control protocol* (MCP) is introduced here to solve these tasks.
The discussion assumes two collaboration participants. In process states in which only one collaboration participant can trigger new MCs the MCP task is easy to solve. That is why only the case where both collaboration participants can trigger a MC is described. The protocol proposed can also be applied to the case where only one participant can trigger a MC. Model checking has been applied to develop the following protocol. The development and validation of the protocol is described in section 5.
Considering the constraints of the distributed environment (cf. p.12) the receiver of a message can hardly be expected to really be ready to receive the message. The reason is that he might have time-outed before. To be precise the main task of the MCP, i.e. determining who begins

the next MC, is redefined as following: *At any point in time only one collaboration participant has the right to trigger a MC*, i.e. to send the first message of the MC. If the other participant still waits for the first message to arrive, i.e. he does not time-out before, and if the message transmission does not take too long, then the MC can successfully be triggered by delivering the first message. Clearly, the first message of a former MC could be delivered as well but this message then has the wrong instance number. The MCP proposed here is not fair, i.e. one participant can continuously prevent the other from beginning MCs by acquiring the right to begin MCs himself. Only one participant of a MCP run can prevent the other from triggering MCs. A MCP run starts when a new process state is reached and it stops when a process state is left. In the following the privileged participant is called *coordinator* whereas the disadvantaged participant simply is called *participant*.

In order to acquire the right to trigger a MC the coordinator as well as the participant first have to request their communication right. If they are granted the right to trigger a MC and they didn't change their MCP state in the meantime, then they can send the first message of the MC. Outdated messages of a MCP run are handled by means of sequence ids. To generate these ids the coordinator as well as the participant hold two sequence counters each. One counter counts the number of own requests and is incremented before any new request. The second counter contains the number of requests of the partner. The detailed usage is described below. Figure 4 shows the protocol machine of the coordinator. The intended behaviour is displayed in black colour. The states and transitions that are only introduced because of the constraints of the distributed environment are displayed in blue colour. The coordinator starts the MCP in state *wait_part*. If he receives a communication request (*com_req*) in this state, he has to check whether the sequence id contained in the request is bigger than the value of the counter for the participant. If this is the case, the coordinator accepts the request for communication and sends a *grant* message to the participant that contains the sequence id of the *com_req* just received. Afterwards the coordinator waits for the first message of a MC (*mc*) that then must use the sequence number of the *grant* message. If he receives such a message the MC begins and the coordinator therefore switches to state *partner_mc*. At this point the MC is executed. After the MC has terminated the coordinator switches to state *checkInternal* and checks if he has to trigger some MCs on his own. The coordinator also switches to state *checkInternal* if he time-outs in state *wait_part*. If there are no MCs to trigger in state *checkInternal*, the coordinator switches to state *wait_part* and waits for another *com_req* sent by the participant. Otherwise the coordinator increments the counter for his own requests and uses it as the sequence id of a *lock_req* message he sends to the participant in order to request communication himself. If the coordinator then receives a *lock* message that carries a sequence id which has the same value as the sequence id he used in his last *lock_req*, the coordinator switches to state *my_mc* and then sends the first message of the MC adding the value of his request counter as sequence id. After the MC has terminated the coordinator switches to state *wait_part* and checks for new *com_reqs* to arrive. All other transitions are used to handle communication errors. To do so, outdated messages are deleted or states are left by means of time-out because some expected messages do not arrive. Figure 5 shows the protocol machine of the participant. The participant starts a MCP run in state *checkInternal*. If he doesn't need to execute any MCs, the participant switches to state *wait_lock* and waits there for *lock_req* messages of the coordinator to arrive. The participant checks if an arriving *lock_req* message carries a sequence id that is bigger than his partner counter. If this is the case, the participant updates his partner counter with the value of the sequence id and sends back a *lock* message carrying the same value. If no

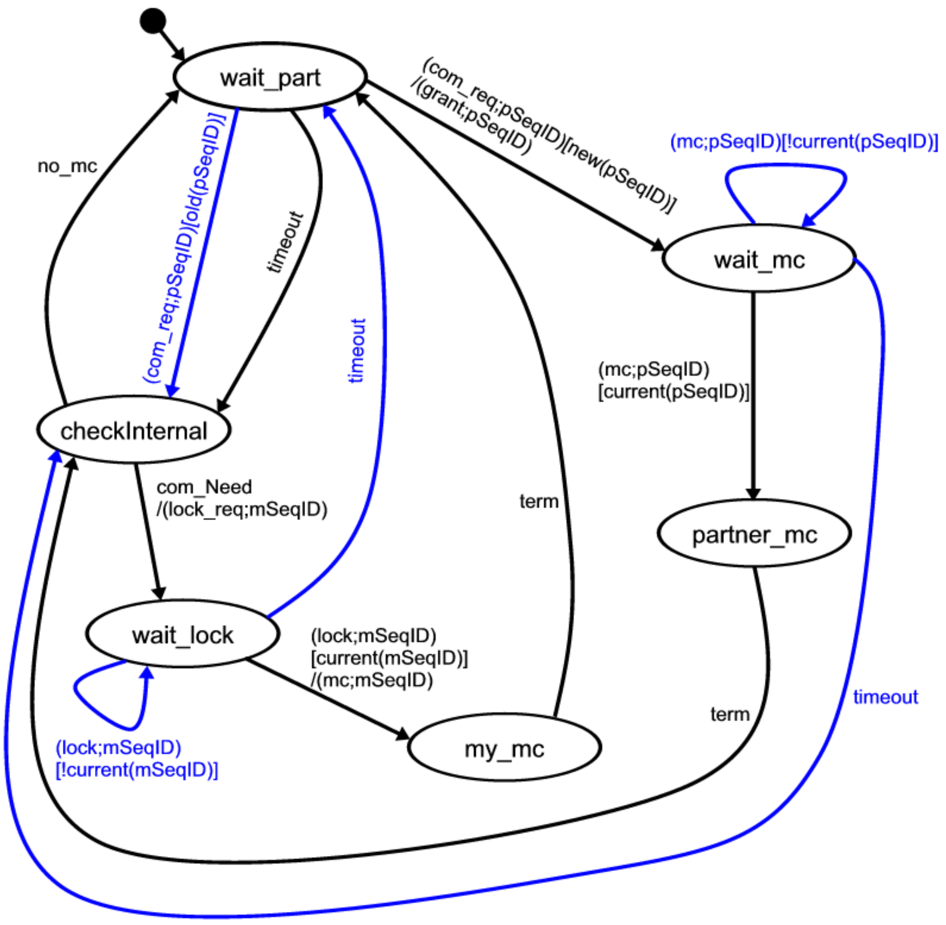

Figure 4: Coordinator automaton of MCP

*lock_req* message arrives in state *wait_lock* the participant time-outs and then switches to state *checkInternal* to check the need for own MCs again.
If the participant has received a valid *lock_req* in state *wait_lock* he switches to state *wait_mc*. There, the participant waits for the first message of the negotiated MC (*mc*) to arrive that must carry a sequence id that is the same as his partner counter. If such a *mc* message arrives the participant switches to state *partner_mc* and then performs the rest of the MC. After the MC has terminated, the participant switches to state *checkInternal* again to check the need for own MCs. If any MCs ought to be executed the participant increments his request counter and then uses its value as a sequence id of a *com_req* message he sends to the coordinator. Afterwards the participant switches to state *wait_grant*. If a *grant* message arrives with a sequence id that is the same as the participant's request counter, the participant switches to state *my_mc* and performs the MC. After the termination of the MC the participant reaches state *wait_lock* again and waits for *lock_req* messages. In state *wait_grant* the participant can receive a *lock_req* instead of a *grant* as well. If such a *lock_req* message contains a valid sequence id the participant must send back a matching *lock* message analogously to state *wait_lock* and then switch to state *wait_mc*. This behaviour resolves the conflict that emerges from both partners trying to acquire the right to trigger a MC. All other transitions and states are used to handle communication errors analogously to the coordinator protocol machine.

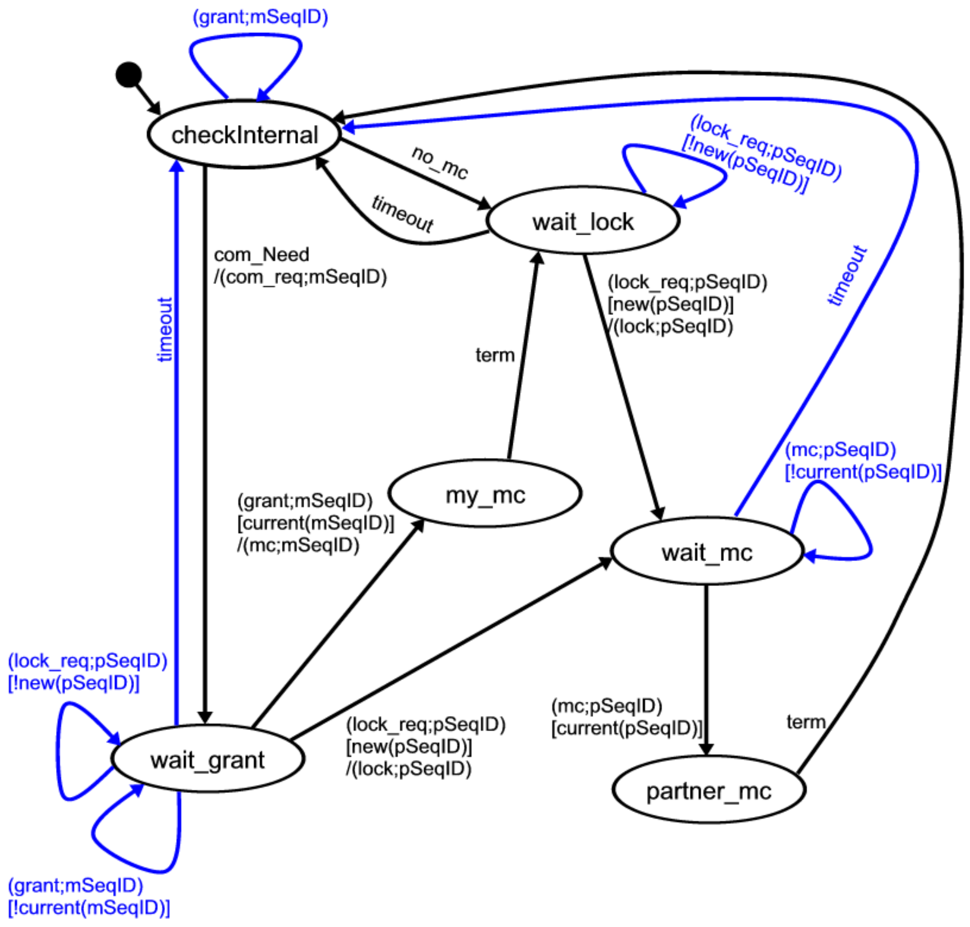
(grant;mSeqID)
checkInternal
no_mc
timeout
wait_lock
(lock_req;pSeqID) [!new(pSeqID)]
com_Need /(com_req;mSeqID)
(lock_req;pSeqID) [new(pSeqID)] /(lock;pSeqID)
timeout
timeout
term
my_mc
(grant;mSeqID) [current(mSeqID)] /(mc;mSeqID)
(mc;pSeqID) [!current(pSeqID)]
wait_mc
(lock_req;pSeqID) [!new(pSeqID)]
wait_grant
(lock_req;pSeqID) [new(pSeqID)] /(lock;pSeqID)
(mc;pSeqID) [current(pSeqID)]
term
partner_mc
(grant;mSeqID) [!current(mSeqID)]

Figure 5: Participant automaton of MCP

The MCP described so far only ensures that at any point in time at most one communication partner has the right to trigger a MC, i.e. at most one partner is in the state labelled *my_mc*. The second partner would need a permission to enter the same state. But as the first partner does not handle any requests before the MC has finished (then state *my_mc* is left), the second partner cannot gain such a permission.
It has not yet been described in detail how the instance numbers are provided for the execution of MCs. The communication partners can choose the sequence id of the request message they used to acquire the right to trigger a MC as an instance number of the MC. These sequence ids can easily be made unique by using a scheme like the following. The coordinator starts his sequence id counter (request counter) with a value of 1 whereas the participant starts with 0. Both partners then always increment their counters by 2.

Distributed time-outs (which lead to new process states) can be handled by checking if the respective timer has run out in a MCP state that is reached again and again. The start states of the communication partners can be used for this task, i.e. *wait_part* for the coordinator and *checkInternal* for the participant. If a distributed time-out event is detected, the current MCP run is terminated, the new process state (not a new MCP state) is entered and a new MCP run is started. As the clocks of the communication partners cannot be assumed to be tightly synchronised and as the transmission time of messages that are used to negotiate a distributed time-out beforehand must be considered, a distributed time-out can lead to an intermediate state. The communication partners then reside in different process states until the second partner has detected the distributed time-out as well. There could be MCs that are applicable for a communication partner in both of these process states. Then a MC could be started with the communication partners assuming different process states which could lead to diverging local views on the collaboration progress. This clearly is unintended. To avoid diverging views the current process state can be added to *lock_req* and *com_req* messages. If the process state contained in such a message does not correspond to the local process state when evaluating such a message, no answer is sent back. The communication partner who has not performed the distributed time-out will then eventually detect and perform it.

The description how MCs are integrated in the MCP deviates from reality for demonstration and analysis purposes. The first message of a MC is not transmitted by the MCP as presented in the protocol machines but by the MC itself. The transmission of the first MC messages is added to the MCP because otherwise it would not be so apparent when MCs are performed. Further, the outgoing transitions from *my_mc* and *partner_mc* are simplifying reality. Any MC includes a 2PC run at the end. So, the communication partners could be blocked[7] while performing the MC. Then the non-blocking partner could not acquire the right to trigger MCs because his partner wouldn't answer. This problem can be solved by piggybacking the result of the last 2PC run to any new *lock_req* or *com_req* message. The blocking partner would then first read the 2PC result, then switch MCP state and finally evaluate the new request message.

Finally process state changes that emerge from MC runs must be considered. If the result of a successful MC run leads to a process state change the respective partner would terminate the current MCP run, switch the process state and eventually start a new MCP run. If the other partner blocked during the MC run the partners would then reside in different process states. As opposed to the situation that emerges from distributed time-outs, in this situation, progress can only be achieved by freeing the blocking partner. But this definitely would lead to a process state change of the blocking partner as well.

**Generating and interpreting business documents.** When automating existing business collaborations, the business logic for generating and interpreting business documents is implicitly, i.e. in the heads of business people, or explicitly, i.e. in business software, present. Further this logic depends on each particular use case. That is why these tasks are not described in detail in this technical report. Instead, the assumption is made that the evaluation of any business document can be represented by a finite set of values. It is also noteworthy that the generation and interpretation of business documents is a lever for applying local business politics to the collaboration. In this technical report the so-called *internal process* is responsible for generating and interpreting business documents.

[7] At most one partner can be blocked because of 2PC properties

**Detecting events of the real world and changing the real world.** Business collaborations are driven by changes in the real world (events) and affect the real world. It is assumed that there are already systems that are able to detect events and change the state of the real world when automating business collaborations. Nevertheless, there must be a clear concept on how to relate these existing systems to the CP and DP on the collaboration. Details depend on particular use cases. Thus this report does not provide general integration rules but shows how such existing systems can be integrated in the use case analysed in section 4. Again, it is noteworthy that integration of such systems is a lever for applying local business politics to the collaboration. In this technical report the so-called *internal process* is responsible for detecting events of the real world and for changing the real world.

**Specifying the control flow of the collaboration.** Control flow is needed to route between process states and MCs. The choice of the MCs to execute depends on events from the real world. The choice of new process states depends on distributed time-outs and the results of MC runs. Further constructs like decision, loop, parallel flow or synchronization are needed to fine-tune the control flow. In this technical report the so-called *local process* is responsible for the control flow of the collaboration. This *local process*, augmented by the MCP and MC execution protocol will be called the *protocol process* in this report.

To complete this subsection, an assumption implicitly contained in the approach proposed here is pointed out. Before any changes to the real world can be made a 2PC run has to be successfully executed at the end of a MC run. A 2PC run to complete successfully requires that every business document of a MC has to be successfully exchanged and interpreted. From this follows that the business documents must be interpreted again (after the 2PC run) to perform the changes to the real world. The application of 2PC would be useless if business documents would not be available or could not be interpreted after the 2PC run. Therefore the assumption must be made that a business document exchanged during a MC is stored safely and that once a business document was interpreted successfully, it can be interpreted successfully again and again.

## 3.2 Using UML activity diagrams for modelling the centralised perspective

To efficiently use the concepts introduced, an appropriate modelling language has to be provided. We found that UML 1.5 activity diagrams are a good choice for the following reasons. UML 1.5 activity diagrams ([OMG03]) are a language suitable for modelling business processes as stated in [DH01]. Activity diagrams are a visual language thus supporting the communication task of a model. There's a large user community using activity diagrams so that a modelling approach based on activity diagrams can easily be adopted in practice. UML is a standardised language which provides the evolution and free-of-charge use of the language and a more or less common meaning of the modelling elements simplifying the exchange of models. Finally, the artefacts of our modelling approach can easily be mapped to activity diagram elements as described below. This proposal for modelling the behavioural view on the CP with UML 1.5 activity diagrams can easily be modified to be used with UML 2.0 statecharts.

**Process states** can be represented by state machine states as activity diagrams are a special case of state machines in UML 1.5. Such states can be hierarchically composed. Thus all attributes that make up the process state of a collaboration can conveniently be modelled as substates of the state machine state. Any micro-choreography or distributed time-out that can be applied in a certain process state is connected to that process state by a transition.

**Micro-choreographies** can be modelled as activities where a single activity models a whole MC. Note that the way distributed consensus is achieved within the activity is not explicitly shown. A consistent outcome of the MC is only implicitly required. The point in time at which MCs get triggered can be captured by events that are added to the transitions that lead into an activity. Such events are always local to one participant. The name of the participant who detects the event can be added to the event identifier in order to make clear that he is responsible for detecting it. Further, guards of incoming transitions to an activity can be used to condition the triggering of the MC. Variables in such guards can be local to one participant as this participant can decide upon the triggering of a MC without consulting his collaboration partners. Activities terminate when the distributed commit protocol that has to be executed at the end of each MC has finished. Therefore no events are allowed in outgoing transitions of an activity. The result of a MC can be visualised as an object flow with multiple result types. The result of a MC execution can then be used in guards to route the collaboration.

**Distributed time-outs** can be represented as UML *when* or *after* events of transitions. Distributed time-outs are designed for situations where no communication is possible, so these transitions directly connect two process states. While distributed time-outs are detected in process states, their corresponding timers are activated in MCs. This leads to unclear semantics that may vary from use case to use case. The following questions therefore have to be answered. If there are multiple transitions that lead into a process state and are preceded by a MC, which of these transitions activate timers for a distributed time-out? If there are multiple distributed time-outs in a process state, must all or only a part of them be activated by a MC? If there are loops in the collaboration, is the timer of a distributed time-out triggered each time a MC leads to a process state or only the first time?
The possibility of loops in a collaboration gives rise to another semantics question. If a process state is left before the timer of a distributed time-out event is detected, should the corresponding timer then continue to progress, should it halt until the process state is reached again or should it be completely cancelled?
Two hints can be given for answers to this questions without looking at use cases. If the execution of a MC results in technical failure no timers of distributed time-outs may be affected at all because in extreme cases a participant may not even have noticed that his partner has tried to execute a MC. And, if timers are not completely cancelled when leaving a process state, then the distributed time-out events they fire can only be applied if the collaboration resides in the corresponding process state or at the moment this process state is reached again.

**Control flow** is supported by activity diagrams with various node types. Alternative flows of execution can be visualised by decision and merge nodes whereas parallel flows of execution can be visualised by fork and join nodes. Further, transitions can be used to specify the relation between process states and micro-choreographies. Process states may have multiple incoming and outgoing transitions. Outgoing transitions of a process state always have an event that shows which real world event leads to the execution of the transition, i.e. local events of the

participants that trigger MCs or global distributed time-out events. Transitions accompanied by local events end in activities whereas transitions accompanied by global events end in new process states.
Micro-choreographies always have exactly one incoming and one outgoing transition. Such an outgoing transition ends in a decision process that determines the next process state depending on the outcome of the micro-choreography. Therefore, the outcome is divided into a technical and a business result. If the execution of a MC fails technically the process always returns to the process state from which the MC was initiated. This case can be represented as a guard with the constant *technical failure* (TF). If the execution of a MC succeeds technically, the next process state depends on the content of the business documents that have been exchanged. This circumstance can be represented by using the names of the relevant business documents as the initial part of variable names that are evaluated in guards to determine the next process state. The restriction that there may only be one incoming transition to a micro-choreography is made to guarantee an unique process state to fall back to in case of a TF. Hence, the same MC may be modelled multiple times within a collaboration.
Guards can be used as described in the discussion of process states and MCs. Finally transitions may be accompanied by actions. Actions are useful for manipulating local variables of the participants for routing purposes. Note that actions may not be used to manipulate process states because distributed consensus would therefore be needed.

**Roles** are a means to specify the tasks a collaboration participant is obliged to fulfil and are useful for the purposes described in section 3.1.1. In activity diagram modelling, swimlanes can be used to visualise roles. Further, role names can be used as part of event identifiers to meet the tasks described in section 3.1.1.

**Business documents** store the information that is negotiated during a collaboration. In activity diagram modelling, object flows can be used to reference the structural definition of such documents thus abstracting from the details of such documents.

## 3.3 Using WSBPEL for modelling the distributed perspective

While a modelling language for the CP must support the communication task of models, it is the primary task of a modelling language for the DP to provide a concise model that can easily be executed. WSBPEL is such a language that describes a business collaboration at the level of Web Service calls. WSBPEL models can be executed by deploying them on adequate process engines providing the right binding information. WSBPEL is well-suited for modelling the DP of business collaborations for various reasons. The modeller can take the point of view of one collaboration participant interacting with his partners of the collaboration via message exchanges which is precisely the task of modelling the DP. Web Services are platform and programming language independent thus supporting the integration of heterogeneous systems which are likely to be found when integrating business processes. And, as well as activity diagrams, WSBPEL has a large user community and is a standardised language.

The main tasks in modelling the DP have already been identified in subsection 3.1.2. The task of *representing process states* can be fulfilled by using a single global variable with an

enumeration type that enumerates all possible process states. *XML Schema*[8] can be used to create such an enumeration type. The task of *specifying the control flow of the collaboration* can then be accomplished by referencing that global process state variable. The whole business collaboration from a single participant's point of view can be represented in WSBPEL as a WSBPEL *while* loop waiting for termination of the collaboration. Within this loop a WSBPEL *switch* construct can be used to select the right code depending on the content of the global process state variable. The condition for terminating the collaboration is then manipulated in process state logic. The WSBPEL *scope* construct can be used to support the specification of the process state logic of a particular process state. A WSBPEL scope can be used to restrict the validity of variables to a particular code fragment and it offers the WSBPEL *onAlarm* construct for specifying relative and absolute timers, i.e. timers with a time period or with a fixed point in time. These timers can be used to model distributed time-outs by writing appropriate variables when running out. Therefore, it has to be considered that the timer of a distributed time-out is activated during the execution of a MC that has been triggered in a different process state. Thus it makes no sense to use relative timers because a collaboration participant could be blocked during the 2PC run of the MC while his partner is already able to proceed. As relative timers are activated when the WSBPEL scope of a process state gets activated, there would be no control over the sequence of starting timers. Absolute timers that refer to global datetime variables should be used instead. Then an absolute point in time can be computed for this timer when performing the 2PC run of the respective MC. When such a timer runs out a variable can be written to store the information that a distributed time-out event has been fired. Unfortunately, there's a drawback of this approach. A 2PC run could block that long that the point in time the timer runs out already has passed when the WSBPEL scope of the new process state is reached. The WSBPEL standard does not define whether a time-out event should be fired or not in such a situation which may lead to compatibility problems among WSBPEL engines. Therefore the assumption is made that a process state is entered before a respective timer of a distributed time-out, if any, has run out. The fact, that the duration of a reservation at the level of business processes is long relative to the time needed to perform a 2PC run and resolving blocking situations, justifies this assumption. Special scenarios that do not meet this assumption need different approaches.
The task of *triggering the execution of micro-choreographies* is addressed by using the MCP (p. 13). As already stated above a new MCP run is started each time a new process state is reached. To represent the MCP in WSBPEL, a loop can be used that is inserted directly within the WSBPEL scope of a process state. Within this loop different code fragments can be applied depending on the current process state. MCP states can be represented by a variable with an enumeration type that enumerates all MCP states. An example of different code fragments depending on process states is the choice between the *coordinator* and the *participant* role of the MCP. In order to detect distributed time-outs, the variables storing the information about distributed time-out events can be read before switching over the MCP states in the MCP loop. Clearly, a distributed time-out can then only be detected when a MCP state changes. A MCP state change can be deferred if a 2PC run blocks. But then a MC run is just being performed and distributed time-outs shall not be applied during a MC run.

Finally a WSBPEL model of the DP must also address the tasks of *detecting events of the real world and changing the real world*, *generating and interpreting business documents* as well as

[8]`http://www.w3.org/XML/Schema#dev`

*executing micro-choreographies* identified in subsection 3.1.2. But these tasks are application dependent. That is why a discussion of these tasks as well as a more detailed treatment of the tasks above is given in section 4.

# 4 A RosettaNet use case

This section is devoted to the evaluation of the modelling approach proposed in the last section by applying it to a use case. Application dependent details are discussed as well. We chose a choreography of RosettaNet PIPs as the subject of our case study for some good reasons. First of all, RosettaNet is a dedicated B2B integration standard that is in use and has a large user community including enterprises like Intel or Sony. Thus we do not have to guess business needs. A large part of the RosettaNet specifications is devoted to the standardisation of the content of business documents. While such domain specific details are not the focus of our work they surely have big impact on real-world collaboration scenarios. Hence, analysing RosettaNet choreographies already takes the next step to applying our modelling approach to real world scenarios. Finally, there's a practical need for modelling support in choreographing PIPs as this task is not standardised yet.
The case study is taken from segment 3A of the PIP directory: *Quote and Order Entry* and is composed of 9 different PIPs of this segment. The use case is a binary collaboration of business partners. The two business partners take the roles of *buyer* and *seller* throughout the whole collaboration. The overall goal of the composition is the negotiation of a contract and of contract changes. The collaboration terminates as soon as the buyer has received the goods and services he requested. In a real world scenario more PIPs can be used to define a comprehensive model of the collaboration including detailed treatment of tasks like *transportation and distribution* (segment 3B) or *returns and finance* (segment 3C). The description of the use case starts with informally explaining the usage of the selected PIPs. Each PIP is identified by its name and the respective *Business Message Pattern* as well as the *Business Transaction Type* are given in parentheses.

- **PIP 3A1: Request Quote (Asynchronous Two-Action Activity; Request/Confirm).** This PIP can be used to start the collaboration. The buyer requests a quote for some particular goods and services. The seller answers with either *Quote*, *NoQuoteAvailable* or with *ReferralToOtherProvider*. In the first case the seller reserves resources for the buyer, but the buyer is not obliged to accept the quote. In case of the other two answers the collaboration is terminated immediately.

- **PIP 3A10: Notify of Quote Acknowledgement (Asynchronous Single-Action Activity; Notification).** If the buyer has received a valid quote he is obliged to use this PIP to inform the seller whether his quote is generally acceptable or not. If not, the collaboration is terminated immediately. Otherwise the seller prolongs the reservation of resources and waits for an order. The buyer is still not obliged to accept the quote.

- **PIP 3A4: Request Purchase Order (Asynchronous Two-Action Activity; Request/Confirm).** This PIP can either be used to start the collaboration or to sign a contract after having confirmed a quote to be acceptable with the help of PIP 3A10. The buyer sends a quote to the seller that can be answered with either an *Accepted*, a *Rejected* or a *Pending* message. If the answer is *Accepted* the parties have signed a legally binding contract. If the answer is *Rejected* the collaboration terminates immediately. If the answer is *Pending*, the buyer waits until the seller notifies him about the decision with PIP 3A7 or, after some time has elapsed, the buyer queries the decision with PIP 3A5.

- **PIP 3A5: Query Order Status (Asynchronous Two-Action Activity; Query/Response).** The buyer can use this PIP either if there is a valid contract, or if the decision of the seller about a quote (PIP 3A4) is still pending or if the decision of the seller about a contract change request (PIP 3A8) is still pending. The answer of the seller has to be evaluated depending on which of these situations applies.
  In the first case, the seller can either provide new information about order progress or just tell that no progress has been achieved. In case of the other two situations the seller can either send an *Accepted*, a *Rejected* or a *Pending* message. If an order has not yet been decided upon, a new contract is signed (*Accepted*), the collaboration is terminated immediately (*Rejected*) or the decision is further postponed (*Pending*). If a contract change request has not yet been decided upon, either the current contract is replaced by a new one (*Accepted*), the current contract remains valid (*Rejected*) or the decision is further postponed (*Pending*).

- **PIP 3A6: Distribute Order Status (Asynchronous Single-Action Activity; Information Distribution).** If there is a valid contract, the seller can use this PIP to communicate information about order progress to the buyer.

- **PIP 3A7: Notify of Purchase Order Update (Asynchronous Single-Action Activity; Notification).** The seller must trigger this PIP if he has sent a *Pending* message in a PIP of type 3A4 or 3A8 before. The seller may only send an *Accepted* or a *Rejected* message, but no *Pending* message. Analogously to PIP 3A5 a new contract is then signed (*Accepted*) or the collaboration is terminated/the current contract remains valid (*Rejected*).
  Further, the seller can use this PIP to request contract changes on his own. As PIP 3A7 is a Single-Action Activity, i.e. only one business message can be exchanged, the buyer cannot directly answer such a request. To answer a contract change request, the buyer should either use PIP 3A8 or PIP 3A9. If the buyer does not so or if the execution of these PIPs fails, the seller awaits a time-out event and then concludes that the current contract remains valid.

- **PIP 3A8: Request Purchase Order Change (Asynchronous Two-Action Activity; Request/Confirm).** This PIP is usually used by the buyer to request a contract change. The seller can then either send *Accepted* to confirm the change, *Rejected* to keep the current contract or *Pending* to postpone the decision. If the seller does not provide a pending decision in time the buyer awaits a time-out and then concludes that the current contract is still valid.
  Further, this PIP is used to answer a contract change request initiated by the seller with PIP 3A7. If the buyer wants to reject the change request, he sends a *Purchase Order Change Request* message that exactly contains the data of the current contract. The current contract then remains valid no matter what the seller answers. To be concise the seller should send a *Rejected* message. If the buyer is about to accept the change request of the seller (with modifications) then he sends a *Purchase Order Change Request* message (with modifications). The seller may then only respond with an *Accepted* message to sign a new contract or with a *Rejected* message to keep the current contract.

- **PIP 3A: Request Purchase Order Cancellation (Asynchronous Two-Action Activity; Request/Confirm).** This PIP is usually used by the buyer to cancel a

current contract. Further the buyer can offer the cancellation of a contract instead of a contract change requested by the seller (PIP 3A7).
In both cases the seller can only send an *Accepted* message to rescind the contract or a *Rejected* message to keep the current contract.

- **PIP 3A: Notify of Purchase Order Information (Asynchronous Single-Action Activity; Notification).** The buyer uses this PIP to inform a so-called *Order Information User* (*OIU*) about the conclusion and cancellation of contracts and about new contract versions.

## 4.1 The centralised perspective

This subsection models the use case introduced according to the approach proposed. This detailed description only covers the part of the collaboration that suffices to sign a contract. The whole model can be provided by the authors upon request. The term *PIP* used in the following description is to be interpreted as a special type of the term *micro-choreography* used in former sections.

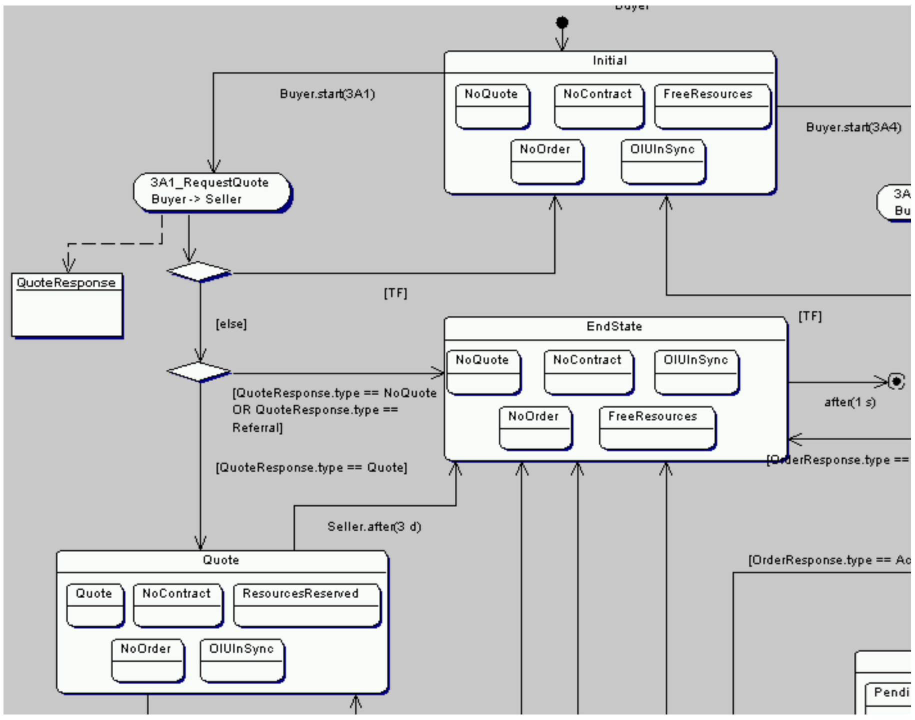

Figure 6: Use case: Path from state *Initial* to state *Quote*

Figure 6 shows the initial part of the business collaboration. In state *Initial* there is neither a quote, nor an order, nor a contract. The necessary resources for fulfilling a contract are still free and the *OIU* has knowledge about the current collaboration progress. The first PIP of the collaboration can only be triggered by the buyer which is represented by local events *Buyer.start(3A1)* and *Buyer.start(3A4)*. If the execution of PIP 3A1 succeeds technically and if the seller sends a *QuoteResponse* of type *Acceptable*, both partners proceed to state *Quote*. The

*Quote* state indicates that a quote exists and that the necessary resources for that quote are reserved. The *OIU* still has knowledge about the collaboration progress because the exchange of a contract is not an event he has to be noticed about. If the seller answers with a *QuoteResponse* of type *Referral* to recommend another provider or *NoQuote* to state he cannot fulfil the request, then the process is terminated. In this case the transition to state *EndState* is fired. This state is only introduced to specify the relevant variable values of the collaboration before termination. Immediately after having reached state *EndState*, the transition to the UML activity diagram end node is fired. If the execution of PIP 3A1 fails (*TF*), both participants return to process sate *Initial*. During the execution of PIP 3A1 the seller reserves resources for the buyer that can be released by means of the distributed time-out event *Seller.after(3d)* without additional communication. Clearly the algorithm for negotiating distributed time-outs (p. 9) has to be applied. Further, the MCP has to ensure that such a distributed time-out event is not processed while the participants are executing a PIP. Figure 6 relates UML activities and PIPs by giving the name of the PIP to the activity. Additionally, the name of the participant who triggers the PIP is appended followed by the name of the participant who receives the first business message. Further the object flow *QuoteResponse* represents the last business document exchanged during PIP 3A1. Generally the result of a PIP execution can be deduced from the last business document exchanged during a PIP. This object flow can then be used in guards to route the control flow of the collaboration according to the business result of the PIP. In figure 6 the guard *QuoteResponse.type == Quote* checks if the seller is able to offer an appropriate quote for the needs of the buyer. *type* can either be a field of or a function on *QuoteResponse* that represents a finite set of values.

Figure 7 shows how state *AcceptableQuote* can be reached from *Quote* by executing PIP 3A10. If this PIP execution fails technically, both participants stay in state *Quote*. If the execution succeeds technically the next state to be reached depends on business document *QuoteEval* that the buyer sends to the seller. If *QuoteEval* states that the seller's quote is basically acceptable the reservation of resources is prolonged. Thus the distributed time-out event *Seller.after(6d)* is defined. If the buyer does not consider the quote of the seller to be acceptable, the next state is *EndState* and the collaboration is terminated. The seller can then release his resources.
After having reached state *AcceptableQuote* the buyer can use PIP 3A10 to terminate the collaboration by stating that he does not accept the seller's quote. This enables the seller to release its resources as soon as possible. In state *AcceptableQuote* PIP 3A10 can only be used to terminate the collaboration. If the execution does not fail technically, state *EndState* is necessarily reached.

The last possibility to leave state *AcceptableQuote* is executing PIP 3A4 in order to conclude a contract, shown in figure 8. Again, technical failure of the execution forces the participants to remain in state *AcceptableQuote*. If the execution succeeds technically the next state depends on the seller's response (*OrderResponse*). If the seller rejects the order of the buyer the participants stay in *AcceptableQuote*. The reservation of resources is still valid in this situation. If the seller accepts the order, both participants reach state *ContractOS* representing a valid contract. At the same time the seller starts the processing of the order using the resources reserved. As the *OIU* has to be informed about the conclusion of contract, sub state *OIUOutOfSync* indicates his information shortcoming.
Finally the seller can postpone his decision by answering *Pending* which leads to process state

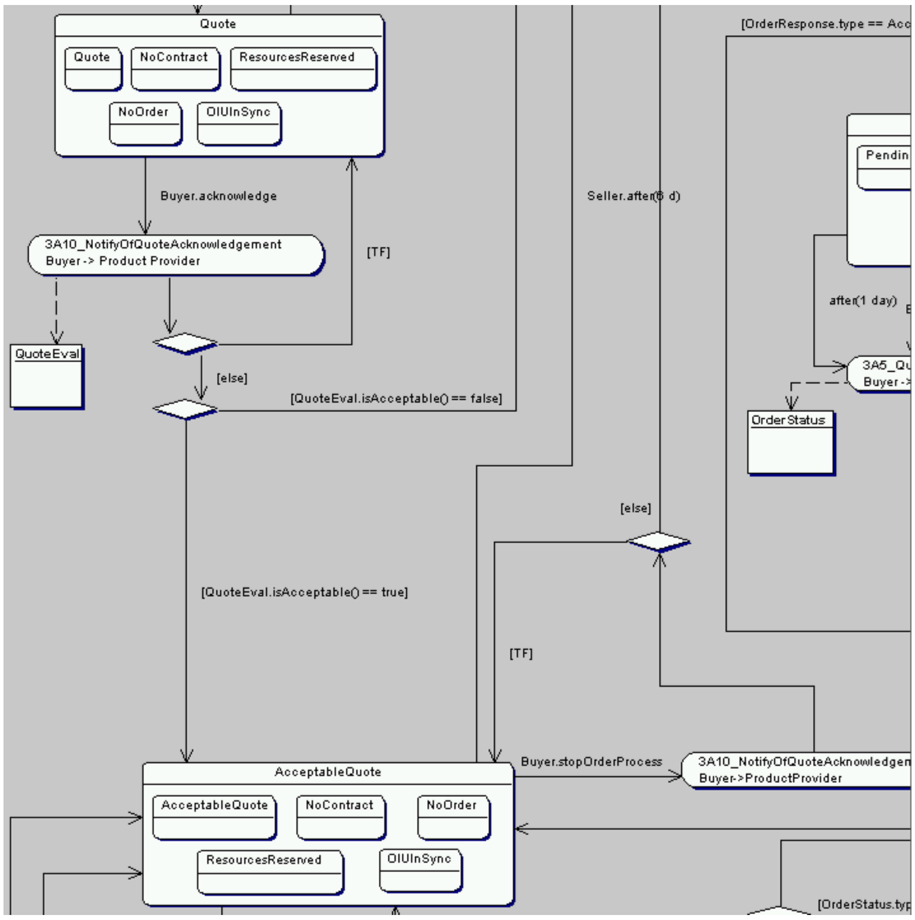

Figure 7: Use case: Path from state *Quote* to state *AcceptableQuote*

*PendingOrder2*. Except of triggering PIP 3A5 state *PendingOrder2* is not discussed in detail. PIP 3A5 can be triggered by two different events. The buyer can query the seller's decision at any point in time which is represented by event *Buyer.infoRequest*. Additionally, the seller's decision should be polled for on a daily basis, represented by event *after(1d)*. Clearly, this event has to be detected by the buyer as well because only the buyer can trigger PIP 3A5.

Process state *ContractOS*, shown in figure 9, can only be left by executing PIP 3A13. But PIP 3A13 actually is only designed for communication between buyer and *OIU*. To preserve the concept of common global state the implementation of PIP 3A13 is therefore adapted. This work assumes a 2PC to be run at the end of any PIP. The seller can play the role of a participant of the 2PC run of PIP 3A13 and thus be informed about *OIU* synchronisation. If the execution of PIP 3A13 succeeds, buyer and seller reach state *ContractIS* representing a valid contract with a synchronised *OIU*. This is also the end of the process fragment described here. In *ContractIS*, the buyer as well as the seller can request contract changes. The buyer can further request the cancellation of the contract and query the processing progress. *ContractIS* is the only process state from which a successful termination of the collaboration is possible. When the buyer detects the fulfilment of the contract he tries to terminate the collaboration using PIPs. If there are technical errors in terminating the collaboration the local variable *Buyer.cf.failcount* is incremented as a reminder for the need to repeatedly try to terminate the collaboration. In order to keep the use case in size PIP 3A13 is used to terminate the

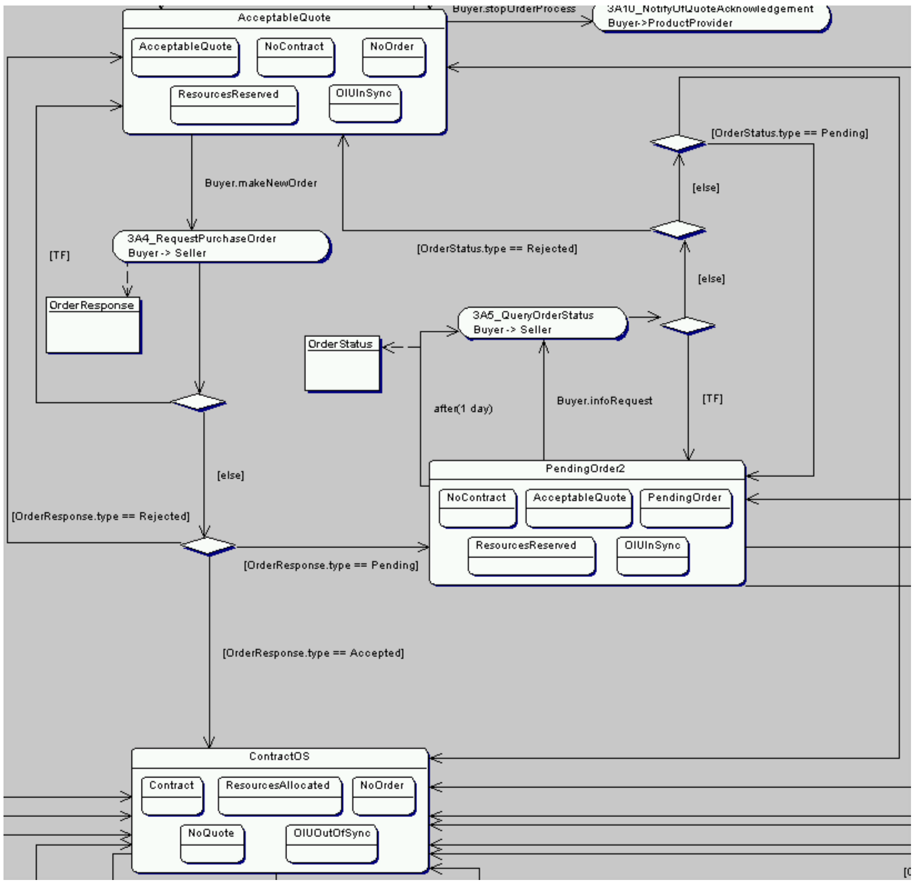

Figure 8: Use case: Path from state *AcceptableQuote* to state *ContractOS*

collaboration instead of more appropriate RosettaNet PIPs for stock receipt and factoring.

Apparently, the modelling of the use case widens the notion of process state defined on p. 8 as *the common view of the collaboration partners on the progress of the collaboration*. This has been done in order to provide an adequate modelling of the *OIU*. The *OIU* does not trigger PIPs himself but is only informed about contracts and contract changes by the buyer. Strictly speaking, if the PIPs of the use case were used exactly as proposed then only the buyer would always have complete knowledge of the collaboration progress because he is the only one to participate in each PIP execution. The *OIU* does not actively influence the collaboration and is only informed about major events. Thus it is acceptable that he has exact knowledge of the progress only in some process states. To represent the knowledge of the *OIU* the sub states *OIUInSync* and *OIUOutOfSync* are used. Any process state with sub state *OIUOutOfSync* can only be left if PIP 3A13 is successfully executed.
The seller however participates actively in the collaboration and therefore always needs complete knowledge of the collaboration progress. From a business point of view it is not interesting for the seller whether the *OIU* is synchronised or not. But from a protocol point of view the synchronisation of the *OIU* is relevant because an unsynchronised *OIU* could prevent the buyer from executing PIPs. Without information about *OIU* synchronisation, the seller cannot distinguish between *ContractOS* and *ContractIS*. So if the buyer was still in state ContractIS, the seller could trigger PIPs that necessarily would fail technically because the buyer cannot respond. To avoid this situation the pragmatic approach of integrating the seller into the 2PC

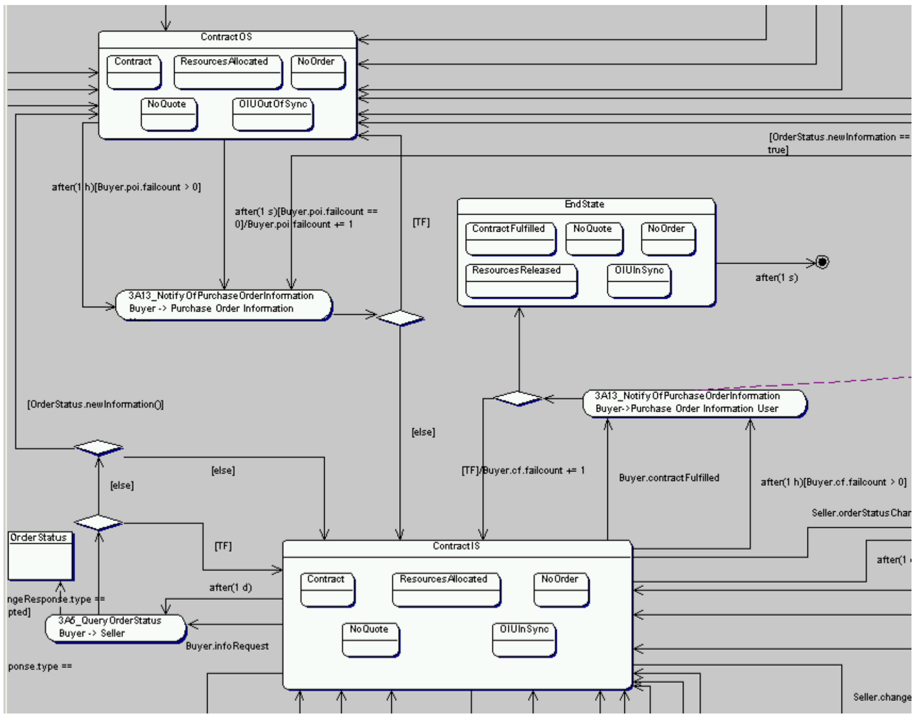

Figure 9: Use case: Path from state *ContractOS* to state *EndState*

run of PIP 3A13 is chosen. Thus the seller can distinguish between *ContractOS* and *ContractIS*.

## 4.2 The distributed perspective

The modelling of the DP has to consider the tasks identified in section 3.1.2. These tasks are partly addressed in earlier sections as far as they are application independent (cf. section 3.3). This section introduces concepts for solving the outstanding tasks in section 4.2.1 and provides a WSBPEL model of the use case in section 4.2.2.

### 4.2.1 Conceptual modelling

According to section 3.3 the tasks of *executing micro-choreographies* and of *detecting events of the real world and changing the real world* as well as *generating and interpreting business documents* still have to be addressed. This section begins with executing micro-choreographies. In tribute to the use case the protocol for solving this task is called *PIP execution protocol* here.

**PIP execution protocol** The goal of the PIP execution protocol is distributed consensus among the participants of a PIP execution. RosettaNet PIPs can be classified according to *Business Message Pattern* (cf. section 2). This section actually introduces a separate protocol for each of the classes *Asynchronous Single-Action Activity* and *Asynchronous Two-Action Activity*. The remaining classes are seldom used and therefore not discussed here. Each protocol has been developed and validated with the help of model checking (see section 5 for details).

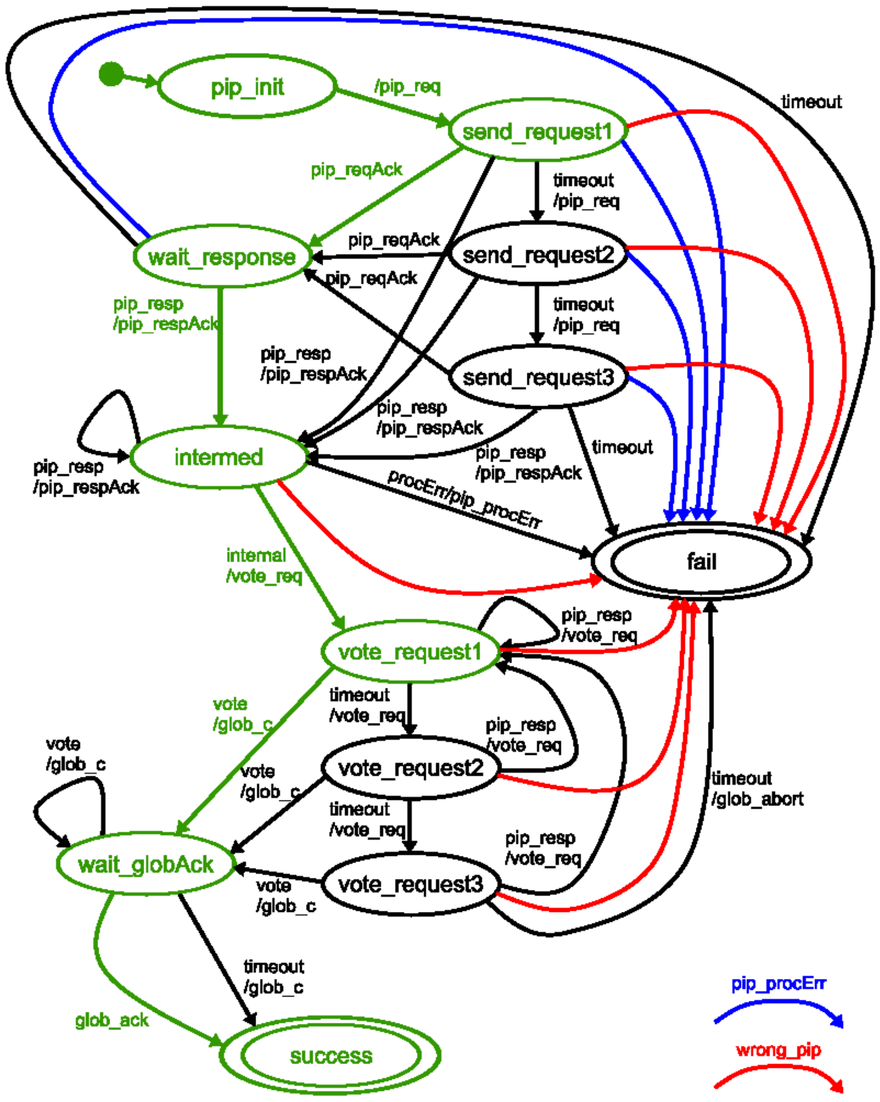

Figure 10: Sender automaton of a Two-Action Activity

This section only presents the end version of each protocol.
In order to ensure a consistent outcome of a PIP execution the RosettaNet specification of a PIP is extended by 2PC. 2PC identifies the roles of *2PC-Coordinator* and *2PC-Participant* that are associated with the participants of a PIP execution as follows. The participant who receives the last business document of the PIP takes the role of 2PC-Coordinator while the remaining participant takes the role of 2PC-Participant. Naming the sender of the first business document of a PIP as *sender* and naming the corresponding partner as *receiver*, the 2PC-Coordinator of a Single-Action Activity is the *receiver* whereas the 2PC-Coordinator of a Two-Action-Activity is the *sender*. As the 2PC-Participant cannot terminate before the 2PC-Coordinator has concluded and distributed the result of the PIP execution, deviating views of the PIP participants on the PIP result are not possible. This is a key difference to the RosettaNet standard in use according to which the sender of the last business document (2PC-Participant) can terminate before the receiver of the last business document has concluded a result.

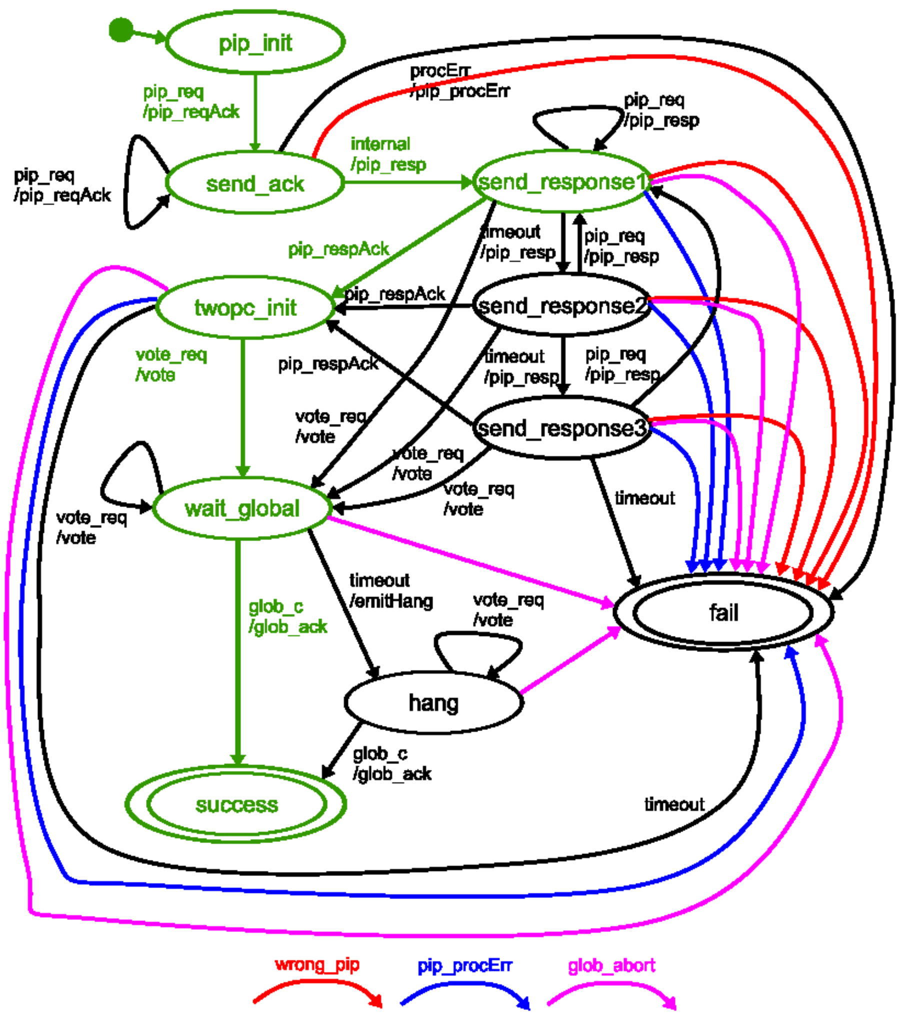

Figure 11: Receiver automaton of a Two-Action Activity

As pointed out before the result of a PIP execution can be divided into technical success and business success. Technical success means that any business document of a PIP has successfully been exchanged and evaluated and that the PIP participants have agreed on that. Business success depends on the evaluation policy for business documents. Moreover it is hard to give a precise definition for the notion of *business success*. Therefore the PIP execution protocol only provides a consistent outcome regarding the technical success. As a 2PC run cannot be started before the last business document of a PIP has been exchanged the intended result of the 2PC is fixed in advance. Communication failures are then the only cause for a 2PC run not to conclude a *global commit*.

Figure 10 shows the protocol machine of the *sender* of a Two-Action-Activity. The states and transitions in green colour represent an ideal protocol run with FIFO channels without errors and with sufficiently fast PIP participants. In such an ideal run, the *sender* begins in state *pip_init*, sends the first business document and awaits an acknowledgement of receipt for this business document in state *send_request1*. If the *receiver* sends this acknowledgement (*pip_reqAck*), the *sender* switches to state *wait_response* and waits for the business document

of the *receiver*. When receiving that business document the *sender* acknowledges its receipt with *pip_respAck* and switches to state *intermed*. The state *intermed* represents the process of evaluating the *receiver's* business document and is left when the *internal process* which is responsible for evaluating business documents has reported successful processing. The *sender* then sends a *Vote Request* (*vote_req*) to start a 2PC run and switches to state *vote_request1*. The *receiver* answers with a *vote* message causing the *sender* to send a *Global Commit* (*glob_c*) and to switch to state *wait_globAck*. Deviating from standard 2PC the *receiver's Vote* message is not divided into *Vote-Commit* and *Vote-Abort* because the intended result is always *Commit* (see above). The *sender* leaves state *wait_globAck* when the *receiver* has acknowledged the receipt of *glob_c* with *glob_ack*.
All other transitions and states except of state *fail* are made necessary by the assumptions made about the environment (cf. p. 12). State *fail* can also be caused by a processing error in the *internal process*. These assumptions necessitate that all states except *pip_init*, *intermed*, *success* and *fail* can be left by means of time-out[9]. When leaving a state with a time-out, the *sender* assumes that the *receiver* didn't receive his last message and sends it again. If this occurs repeatedly, the *sender* finally switches to state *fail*. If the *sender* has sent a *vote_req* before reaching *fail* it could be the case that a *receiver's vote* message has been lost thus causing the *receiver* to block. Therefore the *sender* sends a *glob_abort* when firing the transition from *vote_request3* to *fail* in order to free the *receiver*. There's no need for time-outs in *pip_init* and in *intermed*. The *sender* leaves *pip_init* immediately after having sent the first business message. State *intermed* is left after communication between the PIP execution process and the internal process which is assumed to be secure.
Some transitions are needed because the *sender* can receive the same message more than once. In this case the *sender* assumes the *receiver* not to have received a message and resends it. The receipt of a duplicated *receiver's* message also may indicate that the *receiver's* protocol process has not yet terminated. In such a case the *sender* therefore switches back from state *vote_request3* or *vote_request2* to *vote_request1*.
Two different messages could cause the current state to be left in states *send_requestX*. The *receiver* sends *pip_reqAck* and *pip_resp* in sequence. As *pip_resp* can only be sent after *pip_reqAck* has been sent and as *pip_reqAck* does not carry a business document to evaluate, the receipt of *pip_resp* in states *send_requestX* directly leads to *intermed* possibly without ever having received a *pip_reqAck*.
The red and blue transitions of the *sender* automaton are used to handle the *receiver's* error messages. A *pip_procErr* (blue transitions) announces that the *receiver* was unable to read the *sender's* business document and therefore has terminated his protocol run. *pip_procErr* messages do not have to be processed any more when the *sender* has reached state *intermed* because then the *sender* has already received a *pip_resp*. The *receiver* only sends a *pip_resp* if he was able to read the *sender's* business document before.
A *wrong_pip* message announces that the *receiver* has already terminated its protocol run (by means of time-outs) and therefore has considered a *sender's* message to be outdated. After having received a *vote* message the *sender* does not process any *wrong_pip* messages because then the *receiver* cannot terminate the protocol run by means of time-outs any more.
According to the assumptions made about the environment, messages can overtake each other. Thus messages of former protocol runs can interfere with messages of the current protocol run.

[9]Not to confuse with a distributed time-out.

To distinguish between the two the MCP provides a PIP instance number that is used to tag the messages of the current protocol run. If a message carries the wrong PIP instance number, it is discarded and answered with a *wrong_pip* carrying the wrong PIP instance number. The handling of PIP instance numbers is not shown in figure 10.

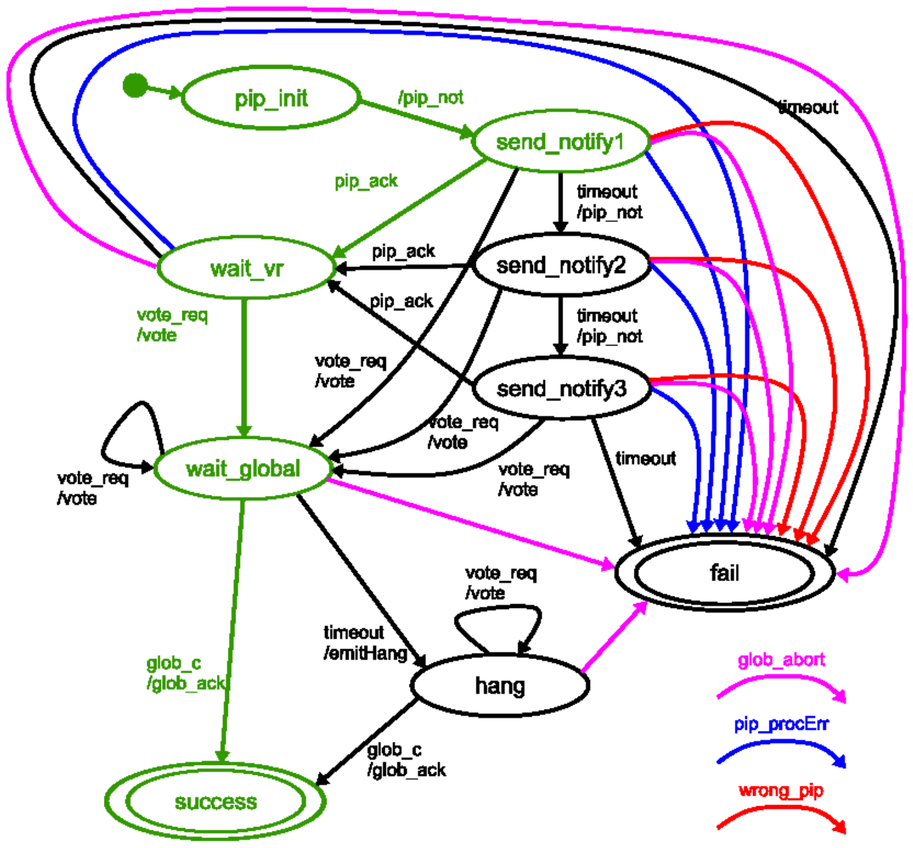

Figure 12: Sender automaton of a Single-Action Activity

Figure 11 shows the *receiver* automaton of a Two-Action Activity. Its structure is complementary to the *sender* automaton of a Two-Action Activity. The choice of colours also corresponds. Therefore only major specialities of the *receiver* automaton are presented.
The state representing the situation that the internal process evaluates a business document is called *send_ack* instead of *intermed*. From a protocol point of view the major difference is that the *receiver* cannot always terminate his protocol run without manual intervention because the *receiver* has to wait for the *sender's* decision after having sent at least one *vote*. If the *receiver* does not receive an appropriate message, both a *Global Abort* and a *Global Commit* decision are possible. If all *vote* messages have been lost the *sender* could conclude *Abort* by repeatedly firing time-out transitions. If at least one *vote* has been received and all *glob_c* messages have been lost the *sender* could also conclude *Commit*. That is why the *receiver* leaves state *wait_global* by firing a time-out transition that ends in state *hang* if no *glob_c* or *glob_abort* arrive. If a *glob_c* or a *glob_abort* arrives afterwards, the *receiver* terminates his protocol run according to the message received. But as there is no guarantee that such a message will eventually arrive, manual intervention may be needed. Therefore the message *emitHang* is sent when switching to state *hang*. *emitHang* represents any reliable mechanism to notify a human of the blocking situation. It is noteworthy that no *wrong_pip* messages may be processed in state *hang* because

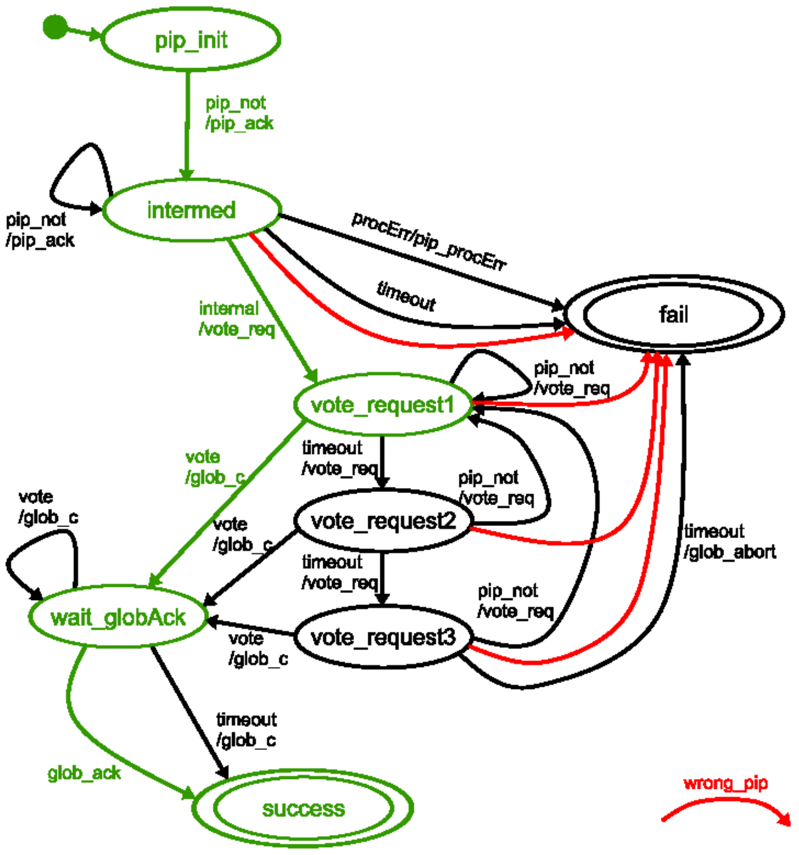

Figure 13: Receiver automaton of a Single-Action Activity

these messages do not carry information about the success of the PIP execution. The same holds for state *wait_global*.
As *glob_abort* messages may arrive in nearly any state of the *receiver* automaton the according transitions in figure 11 have been painted in pink to save labels.

The protocol machines of a Single-Action Activity (figures 12 and 13) are built analogously to the protocol machines of a Two-Action Activity. A textual description for these automata is therefore omitted. According to RosettaNet terminology the first business document to be exchanged is called *pip_not* as an abbreviation for *Notification* instead of *pip_req* (*Request*).

The protocol machines introduced here do not change the behaviour of the RosettaNet protocol for executing PIPs in use except for integrating a 2PC run. Therefore the adaptation of existing implementations should be easily possible.
Finally an implicit assumption of the PIP protocol is that once a business document has been successfully evaluated, it can repeatedly be evaluated. This assumption is necessary because the internal process must check readability, e.g. in state *intermed*, before the global result of the 2PC run is concluded. The actual processing of the business document and the impact on the real world however may not be done until the global result already has been concluded. The assumption makes sure that a PIP may not fail technically after technical success has been announced.

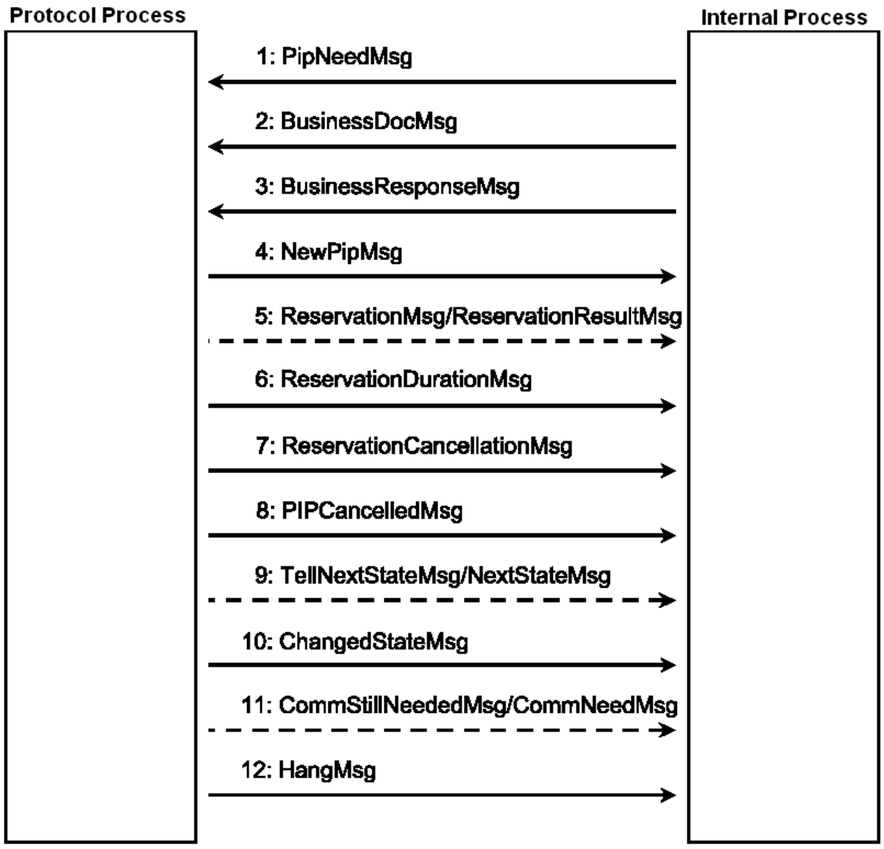
Protocol Process
Internal Process
1: PipNeedMsg
2: BusinessDocMsg
3: BusinessResponseMsg
4: NewPipMsg
5: ReservationMsg/ReservationResultMsg
6: ReservationDurationMsg
7: ReservationCancellationMsg
8: PIPCancelledMsg
9: TellNextStateMsg/NextStateMsg
10: ChangedStateMsg
11: CommStillNeededMsg/CommNeedMsg
12: HangMsg

Figure 14: Message exchange between protocol process and internal process

**Internal Process** In section 3.1.2 the so-called *internal process* has been assigned the responsibility to address the tasks of *detecting events of the real world and changing the real world* as well as *generating and interpreting business documents*. It has also been pointed out that probably, there already exist running systems which fulfil these tasks when integrating business processes. Therefore in this technical report an interface consisting of a set of messages is presented to encapsulate the internal process. As WSBPEL is used to model the protocol process the exchange of these messages by means of Web Services is a natural choice. Figure 14 shows the messages to be exchanged with solid arrows for asynchronous calls and dotted arrows for synchronous calls. The arrows always point to the receiver of the call. The following description details format and functionality of the calls. *ProcessId* and *PIPInstanceId* are used as message parts to identify the whole collaboration between seller and buyer or to identify the PIP instance respectively. The content of a message is informally provided in the format *message name<message component1*, *message component2*, *...>*:

1. **PipNeedMsg<ProcessId, PIP>** With this message the internal process declares need for a particular *PIP* that should be executed because of an event in the real world. After having sent this message, the internal process waits until the protocol process announces the possibility to execute the *PIP* with an according message. The design of the MCP forces the internal process to be able to accept other messages as well because the other collaboration partner could be the first to execute a PIP.

2. **BusinessDocMsg<ProcessId, PIPInstanceId, BusinessMessage>** After having

been informed about the possibility to execute a PIP, the internal process uses this message to provide the first business document of the respective PIP for the protocol process. Afterwards the internal process awaits further messages of the protocol process regarding the PIP execution.

3. **BusinessResponseMsg<ProcessId, PIPInstanceId, Valuation, BusinessMessage>** The internal process uses this message to inform the protocol process whether it has been possible to process a message of the collaboration partner or not. The message part *Valuation* can either have the value *pip_proceed* or *pip_processing_error* to announce this decision. The message part *BusinessMessage* contains a business document as the answer to the collaboration partner's message. If the respective PIP only specifies one business document to be exchanged, *BusinessMessage* may be empty as well.

4. **NewPipMsg<ProcessId, PIPInstanceId, PIP, BusinessMessage>** The protocol process uses this message to inform the internal process of a new PIP to be executed. The message part *PIP* identifies the type of PIP to be executed, e.g. PIP 3A4. The message part *BusinessMessage* carries the partner's business document. The internal process is expected to answer with a BusinessResponseMsg after some time. Concurrently, the internal process must be able to receive further informations like the cancellation of the PIP.

5. **ReservationMsg<ProcessId, BusinessMessage> / ReservationResultMsg<ProcessId, Result>** If resources must be reserved during a PIP execution, the protocol process uses this synchronous call to request the reservation of resources. The message part *BusinessMessage* carries the last business document that has been exchanged during the current PIP. It is assumed that the internal process can deduce the resources to reserve from this BusinessMessage. The reservation of resources is not restricted in time at this point of the PIP execution.
   The internal process uses *ReservationResultMsg* to declare whether the reservation was possible or not. The message part *Result* is a boolean value that indicates the success of reservation. The internal process will receive further reservation information at a later point of the PIP execution.

6. **ReservationDurationMsg<ProcessId, Duration>** The protocol process uses this message to fix the time period of the latest reservation that has been performed. The message part *Duration* specifies the duration of the reservation in seconds. The reason for separating *ReservationMsg* and *ReservationDurationMsg* is the protocol for negotiating distributed time-outs (cf. p. 9).

7. **ReservationCancellationMsg<ProcessId>** The protocol process uses this message to cancel the latest reservation. This message could be sent because of a distributed time-out event or because the current PIP execution that lead to the reservation has failed technically.

8. **PIPCancelledMsg<ProcessId, PIPInstanceId>** The protocol process uses this message to inform the internal process that the current PIP execution failed. The necessity to free resources cannot be deduced from this message because PIPs can fail before resources have been reserved.

9. **TellNextStateMsg<ProcessId, PIPInstanceId, BusinessMessage> / NextStateMsg<ProcessId, ProcessState>** The protocol process uses this message to ask the internal process which is the next process state to be reached after having successfully executed a PIP. The message part *BusinessMessage* carries the last business document that has been exchanged during the PIP. The internal process uses this business document to determine the next process state. The internal process sends the name of the next process state using the message part *ProcessState*. At first the sight, it is paradoxical to ask the internal process to determine the next process state. According to the description above, this is the task of the protocol process. But the protocol process would need business logic to evaluate complex business documents to determine the next process state. This business logic is already implemented in the internal process. As business documents have to be evaluated relative to a process state, the internal process must have knowledge about process states anyway. So the cost for implementing the logic for evaluating transitions between process states in the internal process should be rather moderate.

10. **ChangedStateMsg<ProcessId, ProcessState>** The only situation in which this message is sent by the protocol process is the transition between two process states because of a distributed time-out event. The message part *ProcessState* carries the new process state.

11. **CommStillNeededMsg<ProcessId, PIP> / CommNeedMsg<ProcessID, IsNeeded>** Before executing a PIP the internal process uses a *PipNeedMsg* to declare the need for a certain PIP. Before such a PIP can be executed it could be the case that the other collaboration participant triggers a PIP execution. Thus the requested PIP execution might not be needed any more when the protocol process has acquired the right to execute it. Therefore the protocol process uses this synchronous call to ask the internal process if the requested PIP is still needed. The internal process uses the boolean field *IsNeeded* to state whether the requested PIP is still needed or not. If not, the protocol process releases his right to execute a PIP.
    Otherwise, the internal process is expected to send a *BusinessDocMsg* with the first business document of the PIP to be executed.

12. **HangMsg<ProcessId, PIPInstanceId>** The protocol process uses this message to announce its blocking during a 2PC run. Manual intervention is then needed. This message represents and can be replaced by any mechanism that reliably detects and announces blocking situations, e.g. monitoring.

Not only does the interface introduced encapsulate the internal process, it also moves the whole application specific business logic from the protocol process to the internal process. Thus the implementation of the protocol process can be automated based on the information of the CP and the structure of the PIPs (Single-Action/Tow-Action) used. The implementation could also be done manually, but as the MCP and the PIP execution protocol are highly complex this is an error-prone task. The next section describes how WSBPEL can be used to specify the protocol process. It is assumed that communication between protocol process and internal process is reliable, i.e. message losses, message duplicates and overtaking messages related to this interface are not dealt with.

### 4.2.2 WSBPEL realisation

This section details and extends the description of section 3.3 adapting it to the use case introduced. All relevant concepts are introduced in a general manner and, as far as possible, discussed at the example of process state *PendingOrder1*. A model of the whole WSBPEL process is not given because it can automatically be generated. Process descriptions created with XML based standards such as WSBPEL typically have a complex structure and are therefore hard to use. This requires the use of tools to create process descriptions.
The WSBPEL examples below have been created using *Cape Clear SOA Editor 4.8*[10] for editing WSDL interfaces as well as *Oracle BPEL Designer for Eclipse 3.0*[11] for editing WSBPEL processes. The following description requires basic knowledge of WSDL and WSBPEL.

The presentation of the WSBPEL model starts with the global process loop as described in section 3.3. Listing 1 shows the WSBPEL code of the global process loop. In this and in the following listings, unimportant code fragments are replaced with XML comments, immediately closed WSBPEL tags or with WSBPEL *empty* tags with self-descriptive names.

Listing 1: Global process loop

```
<process>
 <partnerLinks/>
 <!-- definition of global variables -->
 <variables>
  <variable name="procTerminated" type="xsd:boolean"/>
  <variable name="pState" element="tns:globProcessStates"/>
  <!-- more variables go here -->
 </variables>
         <sequence>
  <!-- global process loop begins here -->
  <while name="globLoop" condition="boolean(bpws:getVariableData('
      procTerminated'))">
   <switch name="swProcState">
    <!-- choice of processState QUOTE -->
    <case condition="bpws:getVariableData('pState') = 'Quote'">
     <empty name="quoteCode"/>
    </case>
    <case condition="bpws:getVariableData('pState','','/tns:
        globProcessStates') = 'AcceptableQuote'">
     <empty name="acceptableQuoteCode"/>
    </case>
    <!-- more cases go here -->
   </switch>
  </while>
  <!-- more code -->
  </sequence>
</process>
```

For leaving the global process loop its condition is manipulated within process state code fragments. The MCP is used within process states to agree upon the collaboration partner to

[10] http://www.capescience.com/soaeditor/

[11] http://www.oracle.com/appserver/bpel_home.html

begin the next PIP. Process states are represented by WSBPEL scopes because these offer the possibility to define local variables that are only valid within the scope.
Further, *onAlarm* tags of a WSBPEL scope are useful for implementing timers with relative time periods or with absolute points in time. These timers can then be used to realise the concept of distributed time-outs by writing a variable whenever a timer runs out. When conceiving a WSBPEL realisation for distributed time-outs the algorithm for negotiating distributed time-outs (cf. p. 9) has to be considered. According to this algorithm the distributed time-out of a process state is negotiated during a PIP execution that has been started in another process state. As the 2PC-Participant of a respective 2PC run can block it does not make sense to trigger a relative timer with a static time period when entering the new process state. If the collaboration participant who needs the resources reserved blocks for a long time, it could be the case that the timer for the distributed time-out in the new process state does not run out before the other participant has released his resources. This would violate the goals of the distributed time-outs introduced. Instead, absolute points in time should be computed during a 2PC run that are then used as the deadline of an absolute timer of the new process state. However this method is not absolutely safe either. It could be the case that a collaboration participant blocks that long that the deadline of the distributed time-out has already passed when the new process state is reached. The WSBPEL standard does not specify whether a WSBPEL implementation must trigger a time-out event when an onAlarm tag is activated after its deadline has passed. This could lead to inconsistencies among different WSBPEL implementations. Therefore the assumption is made that the deadline of an onAlarm tag for distributed time-outs has not passed before that onAlarm tag is activated, i.e. before the new process state is reached. This assumption is sensible because the execution of a 2PC run at the end of a PIP takes relatively little time compared to the duration of resource reservation, even if manual resolution of a blocking situation is considered. There might be special cases that need a different distributed time-out realisation because this assumption does not apply.
Apart from timers a WSBPEL scope also offers *onMessage* tags to receive event messages. According to the WSBPEL standard such event messages can be received concurrently while the actual activities of the scope are performed. *onMessage* tags thus are well-suited for receiving *PipNeedMsg* messages. Unfortunately the WSBPEL standard does not provide a precise semantics for *onMessage* tags either. The WSBPEL specification ([IBM03]) states on p. 83, section 13.5.4.2 that arbitrarily many messages may be received via onMessage blocks as long as the activities of the according scope have not finished: "When such an event occurs, the corresponding activity is carried out. However, the event remains enabled, even for concurrent use". When receiving a message via an onMessage block the message to be received is saved to a variable. As such a variable can be used within onMessage blocks of concurrently active scopes, WSBPEL provides the *variableAccessSerializable* attribute for scopes ([IBM03] p.84). If this attribute is set to *true*, a WSBPEL implementation must serialise access to such a variable from tow different scopes. But this does not solve the problem that a single onMessage block of one and the same scope could be active multiple times at one point in time. If only a single onMessage block would be used to receive *PipNeedMsg* messages, it could be the case that two *PipNeedMsg* messages requesting different PIPs would write the variable for receiving messages two times before the according activity is performed. Then only the need for one kind of PIP would be registered. A solution for this problem is to replace the *PipNeedMsg* that carries a PIP parameter with multiple types of *PipNeedMsg* messages, one for each kind of PIP that can be requested. The concurrent use of an onMessage block for registering the need for PIPs is

then no problem because the same kind of PIP is requested.
Listing 2 shows the WSBPEL scope for process state *PendingOrder1*. The *eventHandlers* tag enumerates the events that are waited for during this scope is active. The *onMessage* tag in listing 2 is used to register the need for execution of PIP 3A7. The attributes of this tag identify the Web Service that is used to receive the corresponding message. The Web Service itself is described in a separate WSDL file. When the *onMessage* tag for requesting PIP 3A7 is executed the need for PIP 3A7 is recorded in a variable that is valid in the whole WSBPEL process. Thus the PIP can be executed in another process state without requesting it again, provided that it is applicable in that other process state.
The *onMessage* tag is followed by an *onAlarm* tag. This *onAlarm* tag specifies the timer for the distributed time-out of process state *PendingOrder1*[12]. The variable *psPendingContract1_expireTime* has been written before during the execution of a PIP that led to process state *PendingOrder1*.
The next *onAlarm* tag is used for implementing the transition between process state PendingOrder1 and PIP 3A7 that is annotated with after*(1 hour) [Seller.pendingOrder.failcount > 0]* in the CP. The variable *clockTickOneHour* that is local to scope *PendingOrder1* is therefore set to true after one hour and it is evaluated appropriately in the scope activities.

Listing 2: Scope for representing a process state

```
<scope name="PendingOrder1">
 <variables>
  <!-- variables only visible within PendingOrder1 -->
  <variable name="leaveState" type="xsd:boolean"/>
  <variable name="swState" element="tns:mcStatesParticipant"/>
  <!-- more variables go here -->
 </variables>
 <eventHandlers>
  <onMessage partnerLink="internalProcessLink" portType="tns:
      SellerPortType" operation="receive_a7Need" variable="
      temp_a7rec">
   <assign name="setComNeed">
   <copy> <!-- remember need for PIP 3A7 -->
     <from expression="true"></from>
     <to variable="deferredPip3A7"/>
    </copy>
   </assign>
  </onMessage>
  <onAlarm until="bpws:getVariableData('
      psPendingContract1_expireTime')">
   <assign name="setTimeOut">
    <copy><!-- excute distributed timeout asap -->
     <from expression="true"></from>
     <to variable="procStateTimedOut"/>
    </copy>
   </assign>
  </onAlarm>
  <onAlarm for="'PT1H'">
   <assign name="setClockTick">
```

[12]PendingOrder1 does not have a distributed time-out in the CP. It is introduced here only for demonstration purposes

```
    <copy><!-- one hour has passed -->
     <from expression="true"></from>
     <to variable="clockTickOneHour"/>
    </copy>
   </assign>
  </onAlarm>
 </eventHandlers>
 <sequence name="seqPendingOrder1">
  <empty name="scope activities go here"/>
 </sequence>
</scope>
```

The activities of a process state scope are organised in a loop that continuously switches over the states of the MCP. To determine the exact set of states to switch over, the MCP role of the collaboration participant (*Coordinator* or *Participant*) must be known. This role can be a different one from process state to process state of the protocol process. The seller, whose protocol process is described here, takes the role of *Participant* in state *PendingOrder1* of the use case. Listing 3 shows the main structure of activities in scope *PendingOrder1*. After each MCP state change the distributed time-out event is checked. If it has not yet been registered the WSBPEL switch *mcPartSwitch* chooses the next MCP state. Thus a distributed time-out event can only be detected if a MCP state change occurs. If a PIP execution blocks the next MCP state change could take a long time. But during a PIP execution, even if blocking, the processing of distributed time-outs is not allowed, so this adequate for the CP.

Listing 3: Loop within a process state scope

```
<sequence name="seqPendingOrder1">
 <!-- variable initialization goes here -->
 <while name="wPendingOrder1" condition="boolean(bpws:
     getVariableData('leaveState')) = false">
  <switch name="pStateTimeout">
   <case condition="bpws:getVariableData('procStateTimedOut') =
       true">
   <!-- recognize distributed timeout -->
    <assign name="leaveState">
     <copy>
      <from expression="true"></from>
      <to variable="leaveState"/>
     </copy>
     <copy>
        <!-- switch to next state -->
      <from expression="'End'"></from>
      <to variable="pState" query="/tns:globProcessStates"/>
     </copy>
    </assign>
    <!-- inform internal process about new process state -->
   </case>
   <otherwise>
    <switch name="mcPartSwitch">
     <!-- switch between states of media control protocol -->
     <case condition="bpws:getVariableData('swState','','/tns:
         mcStatesParticipant') = 'checkInternal'">
       <!-- checkInternal activities go here -->
      </case>
      <case condition="bpws:getVariableData('swState','','/tns:
         mcStatesParticipant') = 'wait_grant'">
       <!-- wait_grant activities go here -->
      </case>
      <!-- more cases go here -->
     </switch>
    </otherwise>
   </switch>
  </while>
</sequence>
```

Listing 4 shows the WSBPEL code for MCP state *wait_grant* as a representative for MCP states. In this state the MCP *Participant* waits for a *grant* message that provides the right to trigger a PIP. A *lock_req* message could arrive concurrently and announce the *Coordinator's* wish to trigger a PIP. If the *Participant* doesn't receive a message at all he awaits a MCP time-out and then switches to MCP state *checkInternal*. The task of providing an adequate WSBPEL implementation for this behaviour is to choose between two messages and a time-out. The WSBPEL *pick* tag does exactly that. Within a *pick* multiple *onMessage* tags for receiving different types of messages and at most one *onAlarm* tag for specifying a relative or absolute timer can be used. If there are multiple *onMessage* branches the first branch in the code is the first to be checked at runtime. Alternatively, a single *onMessage* branch can be used to receive a general message container and the right code for processing the various message types within the container could be chosen depending on its content.
Nevertheless the WSBPEL *pick* tag does not suffice to implement MCP state *wait_grant* because outdated messages must be handled. These outdated messages possibly cause the *pick* tag to be executed multiple times. A WSBPEL *while* loop is used to reactivate the *pick* block each time an outdated message has been processed. Each time the *pick* block is reactivated, its *onAlarm* timer is reactivated too. A relative timer with a static duration therefore is not adequate because then the time needed until a time-out event is fired would greatly depend on the number of outdated messages that arrive. Alternatively, one could try to use an absolute timer with a deadline. This approach fails either because WSBPEL does not specify what to do if a deadline has already passed when an *onAlarm* tag gets activated. Thus it could be the case that the deadline is passed while an outdated message is processed. In such a case, if no further messages arrive, then the protocol process could block permanently.
In order to provide an appropriate implementation for a MCP *wait_grant* state a separate WSBPEL scope is used. A relative *onAlarm* timer of that scope is used to detect the MCP time-out for *wait_grant*. Within this *scope*, a *pick* block is put into a *while* block for being able to repeatedly process incoming messages. Further the *onAlarm* branch of the *pick* is used to specify a relative timer with a short duration. Each time this timer runs out, the MCP time-out for *wait_grant* is checked for. Listing 4 shows the mechanism in WSBPEL.

Listing 4: Representation of state wait_grant

```
<case condition="bpws:getVariableData('swState','','/tns:
   mcStatesParticipant') = 'wait_grant'">
 <scope name="sWaitGrant">
  <variables>
```

```
   <variable name="wg_timedOut" type="xsd:boolean"/>
   <variable name="wg_leave" type="xsd:boolean"/>
   <variable name="temp" messageType="tns:containerMessage"/>
  </variables>
  <eventHandlers>
   <onAlarm for="'PT30S'">
    <assign name="setTimeout">
     <copy>
      <from expression="true"></from>
      <to variable="wg_timedOut"/>
     </copy>
    </assign>
   </onAlarm>
  </eventHandlers>
  <sequence name="seqWaitGrant">
   <!-- initialize variables -->
   <while name="waitForMessages" condition="(bpws:getVariableData
       ('wg_timedOut') = false) and (bpws:getVariableData('wg_leave
       ') = false)">
    <pick name="pickMessage">
     <onMessage partnerLink="buyerLink" portType="tns:
         SellerPortType" operation="receiveBuyerMessage" variable="
         temp">
      <!-- message handling depends on message content -->
     </onMessage>
     <onAlarm for="'PT5S'">
      <empty name="pollStateTimeout"/>
     </onAlarm>
    </pick>
   </while>
   <!-- more code goes here -->
  </sequence>
 </scope>
</case>
```

After having used the MCP to negotiate the triggering of a PIP, the PIP execution protocol has to be applied. A WSBPEL scope is used to represent a PIP execution in order to define local variables. The detailed actions of the PIP execution protocol depend on the *Business Message Pattern* of the PIP (*Single-Action/Two-Action Activity*) and the role of the process within the PIP (*Sender/Receiver*). The general structure however can be organised in a loop that switches between the different PIP execution protocol states in each cycle. This loop terminates when the technical success of the PIP has been decided. In process state *PendingOrder1* the *seller* takes the *sender's* role in PIP 3A7. Listing 5 shows the WSBPEL model of PIP 3A7. PIP execution states *send_notifyX* have been merged to a single state *send_notify* with a variable to count the number of business documents that already have been sent. Analogously to MCP state *wait_grant*, a separate scope is used for representing a single PIP execution state if multiple message receipts and a time-out can occur.

Listing 5: Representation of PIP 3A7

```
<scope name="do3A7">
 <variables>
```

```
  <variable name="result" type="xsd:string"/>
  <!-- scope variables go here-->
 </variables>
 <sequence name="seqPIP">
  <assign name="initPipVars">
   <!-- initialize variables here -->
  </assign>
  <while name="pipStateLoop" condition="bpws:getVariableData('
      result') = 'undecided'">
   <switch name="switchPipStates">
    <case condition="bpws:getVariableData('pip_state','','/tns:
        pip1ActionStatesSender') = 'pip_init'">
    </case>
    <case condition="bpws:getVariableData('pip_state','','/tns:
        pip1ActionStatesSender') = 'send_notify'">
    </case>
    <!-- more cases go here -->
   </switch>
  </while>
 </sequence>
</scope>
```

If a protocol process takes the sender's role during a PIP execution, it has to ask the internal process in state *pip_init* if the execution of the PIP is still necessary. If this is not the case the *while* loop surrounding scope is directly terminated. Otherwise either PIP execution state *fail* or *success* will eventually be reached. Depending on these states various variables for the *MCP* and *local process* are set and messages are sent to the *internal process*. In the case of state *success* the *internal process* is asked which process state should be reached next and the variable for controlling the global process loop is accordingly manipulated. Further, if the protocol process takes the *sender's role* in the PIP, the variable for registering the need for the respective PIP has to be set to false (*deferredPIP3A7* in case of PIP 3A7). If a PIP should be repeatedly triggered (once per hour in case of PIP 3A7) in case of a failed PIP execution, the failure has to be registered in an appropriate variable.
Resource reservations cannot be dealt with in states *success* and *fail* because the algorithm for negotiating distributed time-outs (cf. p. 9) has to be applied. Considering PIP 3A7 this means that resources have to be reserved before the first *vote* message in PIP execution state *send_notify* or *wait_vr* has been sent. If reservation of resources fails the *seller* does not send a *vote* and the PIP execution fails as well. If the reservation is successful the *seller* waits until he has reached state *success* before he sends the duration of reservation to the internal process. This is done because the reserved resources must not be released before the buyer has concluded they are not reserved any more. When reaching state *fail* a *ReservationCancellationMsg* message must not be sent to the internal process because *fail* can be reached without having reserved resources before. Therefore *ReservationCancellationMsg* must be sent after an error in states *wait_global* or *hang* has occurred.
It is noteworthy that the implementation of PIP state *hang* does not have an *onAlarm* branch. This is done to implement the blocking situation in a 2PC run. Despite the fact that no loop is used in state *hang* the processing of messages from outdated protocol runs is possible. After having processed such a message in state *hang* the implementation of the PIP execution pro-

tocol routes the control flow back to state *hang* automatically. Listing 6 shows the WSBPEL code of *hang*.

Listing 6: Representation of state hang

```
<case condition="bpws:getVariableData('pip_state','','/tns:
   pip1ActionStatesSender') = 'hang'">
 <sequence xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-
    process/" name="sequence-1">
  <receive createInstance="no" name="recHangMessage" partnerLink="
     buyerLink" portType="tns:SellerPortType" operation="
     receiveBuyerMessage" variable="partContTemp"/>
  <switch name="oldMsg">
   <case condition="bpws:getVariableData('partContTemp','container
      ','/newPipEnv/pipInstanceNumber') = bpws:getVariableData('
      pipInstanceNumber')">
    <sequence name="handleNew">
     <!-- some code omitted here -->
     <switch xmlns="http://schemas.xmlsoap.org/ws/2003/03/business
        -process/" name="whichMessage">
      <case condition="(bpws:getVariableData ('partnerContainer','
         container','/newPipEnv/pipProtocolMsg') = 'glob_abort')">
       <!-- handle global abort message -->
      </case>
      <case xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-
         process/" condition="bpws:getVariableData('
         partnerContainer','container','/newPipEnv/pipProtocolMsg
         ') = 'vote_request'">
       <!-- handle vote_request message -->
      </case>
      <case xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-
         process/" condition="(bpws:getVariableData ('
         partnerContainer','container','/newPipEnv/pipProtocolMsg
         ') = 'glob_c')">
       <!-- handle global commit message -->
      </case>
     </switch>
    </sequence>
   </case>
   <otherwise>
    <!-- handle message from previous protocol run -->
   </otherwise>
  </switch>
 </sequence>
</case>
```

The approach introduced uses the PIP execution protocol within the MCP. Thus one collaboration participant could block at the level of the PIP execution protocol while the other participant already sends messages at the level of the MCP again. Generally, the blocking participant then cannot react to these MCP messages because their receipt is not expected at the level of the PIP execution protocol. As stated above, this problem can be addressed using the concept of *piggybacking* and can be implemented using a common message container for MCP and PIP execution protocol. This container consists of the process instance id, the current process state, a MCP sequence number, type of MCP message and a PIP envelope. Generally, at the level of the MCP all message parts except the PIP envelope are processed whereas at the level of the PIP execution protocol, only the PIP envelope is processed. Only when sending *com_req* or *lock_req* messages (MCP level), the latest PIP envelope that has been sent is sent again. Considering the message order of the PIP execution protocol this PIP envelope either contains a *glob_abort* or a *glob_c* message. If the receiver of such a message container is still blocked, he first checks the PIP envelope and is then freed again. A PIP envelope consists of PIP type (e.g. 3A4), the PIP execution protocol message type (e.g. *pip_req*), the PIP instance id and, if necessary, a business document.

**Particularities of the WSBPEL realisation** Even though FIFO channels without message losses and duplicates are assumed for communication between protocol process and internal process, the PIP instance id of these messages must be evaluated. The reason for that is the concurrent receipt of messages by the protocol process from the collaboration partner and the internal process. For example in PIP state *send_ack* the receiver's protocol process can receive both *wrong_pip* messages from the collaboration partner and a *BusinessResponseMsg* from the internal process. If a *wrong_pip* arrives earlier than a *BusinessResponseMsg* then the PIP execution is terminated and the *BusinessResponseMsg* is delivered next time an appropriate state is reached. But then the *BusinessResponseMsg* is outdated and must be deleted.

The WSBPEL model described here does not use *message correlation* ([IBM03], p.45-52), a mechanism for correlating messages and WSBPEL process instances. For binary collaborations such as the use case described here WS-Addressing ([Ora05], section 6-7) can be used instead. If the use of correlation sets is intended, then the PIP instance id must not be part of a collaboration set. Otherwise outdated messages from former protocol runs could not be handled.

Finally there is no default functionality in WSBPEL for manipulation of datetime values, e.g. for computing the deadline for process state timers. To solve this problem embedded Java can be used in some WSBPEL engines but then processes in use are not WSBPEL compliant any more. Alternatively the problem can be solved by manipulating string variables or by implementing a dedicated Web Service for datetime operations.

# 5 Validating business collaborations

From our point of view validation in early design phases is a key success factor in building complex systems. Design errors are costly if they are detected late in the development cycle. Model checking is a technique that supports detecting errors by completely exploring the state space of a system. That is why errors that emerge from very improbable situations are more easily found because they are not simply overlooked. Thus we have applied model checking techniques to our case study as well. Model checking is frequently applied by formalising properties one suspects to hold for a certain model. The model checker then either verifies the property or finds an execution that violates the property (or it runs out of resources). For the interested reader more information about model checking can be found in ([BBF01], [CGP99], [Hol03]).

The rest of this section first identifies and classifies relevant properties for the use case in section 4. Then requirements of a model checker to reliably validate these properties are discussed. Finally we describe the use of TCM/TATD and SPIN for model checking our use case.

## 5.1 Collaboration properties that should be checked

Before identifying relevant properties of a model it should be clear that the analysis of a property depends on the amount of information in the model. Therefore some very interesting properties of the use case in section 4 cannot be model checked. A very apparent case is the legitimacy of decisions of a collaboration participant. It would probably be illegitimate if a seller does not accept the purchase order of a buyer if the buyer exactly obeys the rules of a quote he has received from the seller beforehand. To do such an analysis, detailed information about the structure of business documents would be necessary, not to mention a formal definition of the notion of *exactly obeying the rules of a quote.*
A less apparent case of properties that cannot be analysed with our model can be encountered when looking at events. Imagine a process state in which a buyer can initiate two different PIPs, one to notify the success of a collaboration and one to cancel the whole collaboration. Assume further that the buyer has detected success and hence triggers a local event to notify his collaboration partner. Then he surely would not try to cancel the collaboration afterwards. In our model there's no information if such event sequences are admissible or not. This means that the system we model allows more executions than are possible in the real world. This has consequences depending on what kind of property is analysed, i.e. if the property to be analysed is a safety or a liveness property. According to [BBF01], p.84, the truth value of a safety property can be decided in any particular state of the state space only by looking at the sequence of states that led to that particular state. If this is not the case, i.e. if the truth value depends on future states, the property under consideration is a liveness property. Assuming no modelling mistakes have been made, successfully verifying a safety property means that the property also holds in the real world whereas not being able to do so not necessarily means the property does not hold in the real world. Regarding liveness properties theoretically no results at all can be carried over to the real world from verifying a given property in the model.

Accepting these restrictions we still claim that model checking is useful for finding errors in our model.

As the identification of properties to check, probably one of the most important parts in validating systems, is not inherently supported by model checking, validating systems still requires the validator to have a lot of experience and analytical skills. We propose a taxonomy of relevant properties to at least help the validator in finding the right properties to check. An intuitive approach to identifying relevant properties of a given system is looking at its purpose in detail and trying to derive properties that ensure that purpose.
There are three main aspects of business collaborations which are relevant for validation.

- **Sanity of the centralised perspective**
- **Conformance of the distributed implementation to the centralised perspective**
- **Interference of local business politics with the centralised perspective**

### 5.1.1 Sanity of the centralised perspective

In essence, *sanity of the CP* amounts to a sensible order of process states, PIPs (respectively MCs) as well as local events and distributed time-outs. This description only suffices to clarify the scope of *sanity of the CP*, so a more detailed classification of relevant properties is given in the following.

**Generic properties** refer to the process nature of a model and should hold in any model that represents a process. A frequently demanded property of process models is absence of livelocks and deadlocks to ensure that progress is eventually achieved. [Esh02], p.171, defines further properties that should hold in a process model. A property that is closely related to absence of deadlocks and livelocks is that processes should always terminate in a defined end state. This is only possible if absence of deadlocks and livelocks is given. If it does not hold that processes always terminate in a defined end state then it must be at least provided that this is possible. A further property in [Esh02] refers to processes that are modelled with activity diagrams and requires that any state of a process model can be reached and that any transition of the model can be fired. Finally [SO00] demands for activity diagram models that *lack of synchronisation* should be impossible, i.e. there should be no situation in which two instances of the same node are active.

**Domain specific properties** refer to the details of B2B integration. Most business processes require resources to be reserved. So one important property is that at the end of any business process, all resources should be free. A further property is that a state should only be reachable if all its substates have been created by an appropriate micro-choreography execution or by a distributed time-out, i.e. there should be no state changes without distributed consensus. From our point of view the creation of a more comprehensive set of properties is the task of B2B standards consortia such as RosettaNet. [JM99] defines property patterns that could be applied to find the right properties. These patterns are *Sequence*, *Combined Occurrence/Exclusion*, *Consequence* and *Precedence*.

**Application specific properties** cannot be defined before a specific application context is given, but finding properties can be supported by looking at the purpose of a collaboration as proposed above and by applying the patterns just named. One property that should hold in our use case is that a process state with a signed contract should only be reachable if *PIP 3A4 Request Purchase Order* has been executed successfully.

### 5.1.2 Conformance of the distributed implementation to the centralised perspective

This aspect can be investigated at two levels. The first is application dependent and checks whether the same process states are reachable in the CP and DP and whether the same transitions can be fired. In other words, the equality of the CP and DP on a particular use case can be analysed. Formal methods that stem from the area of process algebra can be applied to this question. The approach proposed here, suggests a (semi-)automatic generation of the DP based on the information in the CP. Assuming this generation to be correct, addressing conformance on this level has minor priority and is not further investigated here.

Rather, the aspect of conformance is researched at the second level of application independent infrastructure functionality. This infrastructure functionality is obliged to ensure the assumptions made in the CP regarding the execution of PIPs (MCs), the triggering of PIPs (MCs) and the management of distributed time-outs. The MCP and the PIP execution protocol are assigned to this task and therefore have to be analysed. But before identifying relevant properties, assumptions about the environment in which these protocols operate should be clarified.

**Assumptions about the environment**
Section 3.1.2 defines important assumptions about the communication facilities the participants of the business collaboration use. These constraints have to respected when validating the MCP and the PIP execution protocol.
Further, assumptions about time-outs have to be made. Section 3.1.1 only states that the participant who reserves resources for his partner has to maintain this reservation longer than his partner. The correctness of the algorithm for negotiating distributed time-outs is also informally explained in that section. Therefore the general assumption is made that time-outs can occur at any point in time. According to p. 5 the proof of liveness properties then only has limited validity because non-deterministic choice of time-outs generates a state space that allows more protocol runs than concrete choice of timer values would do. Assuming fixed timer values, the *Coordinator* of a MCP run for example maybe would always generate a time-out event in state *wait_pip* before the *Participant* of the MCP run sends a *pip* message after having received an appropriate *grant*. The *Participant* would then be unable to execute a PIP. On the other hand the right choice of time-outs may also guarantee that each participant of the MCP can trigger a PIP in finite time and thus solve the MCP fairness problem.

The assumptions introduced in section 3.1.2 and in this section are an important parameter in validating the use case because they heavily influence which properties can be verified and which not. For example there shouldn't be a blocking situation in the PIP execution protocol if no messages get lost. Therefore it is sensible to perform the validation in multiple steps with different assumptions and different properties.

**Relevant properties of the MCP**
Absence of deadlocks is a frequently postulated property for protocols. This property is also required for the MCP because the collaboration participants have to negotiate the triggering of PIPs repeatedly. Considering the functionality of the MCP, extensively discussed in section 3.1.2, the property that at any point of time only one collaboration participant has the right to trigger a PIP, must be validated. Further, it can be presumed that each collaboration participant will be allowed to trigger a PIP after finite time if only the communication medium is error-free and protocol states are not left prematurely by the participants. This presumption should also be analysed with the help of model checking. Finally, the MCP must consider distributed time-outs. Processing such a distributed time-out event leads to leaving the current process state as well as terminating the current and starting a new MCP run. If both participants detect a distributed time-out at the same time, this behaviour is no problem. If not, the situation arises where participants execute different MCP runs in different process states. As long as this is the case, MCP messages cannot be processed because they do not carry the expected process state names. It has already been informally pointed out on p. 16 that including process state names in MCP messages suffices to correctly handle distributed time-outs. Analysis with means of model checking is therefore omitted.
Table 1 shows the validation levels of the MCP. *Successful triggering of PIPs* is omitted on level two, because the possibility of loosing messages prevents that the acquisition of the right to trigger PIPs can be guaranteed.

| Level | Environment | Property |
|---|---|---|
| 1 | • No message losses<br>• FIFO channels<br>• Finite message transmission time<br>• Duplicate messages<br>• Time-outs at arbitrary points in time | • Absence of deadlocks<br>• Mutual exclusion regarding the right to trigger PIPs<br>• PIPs can successfully be triggered, if *Coordinator* and *Participant* do not leave state *wait_pip* prematurely |
| 2 | • Message losses<br>• Overtaking messages<br>• Arbitrary message transmission time<br>• Duplicate messages<br>• Time-outs at arbitrary points in time | • Absence of deadlocks<br>• Mutual exclusion regarding the right to trigger PIPs |

Table 1: Media Control Protocol validation levels

**Relevant properties of the PIP execution protocol** The functionality of the PIP execution protocol is extensively discussed in section 4.2.1. The distinction between the two protocol variations for *Single-Action Activity* PIPs and *Two-Action Activity* PIPs is not relevant for the

purpose of identifying properties, because these variations only differ in the number of messages to exchange but not in the goals that should be achieved. The most important property of the PIP execution protocol is guaranteeing a consistent outcome of a PIP execution. This property is required to hold no matter in which environment the collaboration participants interact. In order to guarantee a consistent outcome, blocking situations must be accepted. Therefore the termination of the PIP execution protocol can only be demanded for if messages are assumed not to get lost and to be delivered in finite time (otherwise the *2PC-Participant* could block in state *hang*). The last property to validate is that technical success of the PIP execution protocol is possible, i.e. both participants reach state *success*. Clearly this property does not have very big significance. Nevertheless it is postulated because executing a 2PC at the end of a PIP constrains the PIP execution and must not prohibit success entirely.
There are two assumptions that cannot be checked because the model introduced here does not contain enough information. First, a consistent business result depends on the application of a uniform evaluation function to the business documents by both participants. Second, a business document that can be read once must be readable again and again (cf. p. 34). Apart from that, timers for generating distributed time-out events must be activated. The correctness of the algorithm for appropriately activating such timers is informally discussed in section 3.1.1 and is not validated here. Finally the 2PC of PIP 3A13 has been extended to include three participants, especially the *seller*. Special treatment of this extension is omitted.
Table 2 shows the validation levels of the PIP execution protocol.

| Level | Environment | Property |
|---|---|---|
| 1 | • No message losses<br>• FIFO channels<br>• Finite message transmission time<br>• Duplicate messages<br>• Time-outs at arbitrary points in time | • Consistent outcome<br>• Both participants terminate<br>• Technical success possible |
| 2 | • No message losses<br>• Overtaking messages<br>• Finite message transmission time<br>• Duplicate messages<br>• Time-outs at arbitrary points in time | • Consistent outcome<br>• Both participants terminate<br>• Technical success possible |
| 3 | • Message losses<br>• Overtaking messages<br>• Arbitrary message transmission time<br>• Duplicate messages<br>• Time-outs at arbitrary points in time | • Consistent outcome<br>• 2PC-Coordinator terminates<br>• 2PC-Participant terminates or reaches *hang*<br>• Technical success possible |

Table 2: PIP execution protocol validation levels

### 5.1.3 Interference of local business politics with the centralised perspective

The modelling of a business collaboration introduced in this report bases on the CP and its implementation specified in the DP. Local business politics of the collaboration participants are not explicitly modelled. Rather they are encapsulated by means of the internal process interface (cf. p. 35). The internal process is responsible for *detecting events of the real world and changing the real world* as well as *generating and interpreting business documents.* These tasks clearly affect properties of the CP like absence of deadlocks and livelocks. A detailed analysis of how local business politics affect properties of the CP is helpful for optimising local business processes without breaking the whole collaboration and therefore a valuable source of information. This analysis is still ongoing work and therefore omitted here.

## 5.2 Requirements of a model checker

Applying methodologies efficiently and reliably needs tool support. That is why we identify core requirements that any model checker should meet for efficient use. Clear and well-documented formal semantics, simulation capabilities as well as resource usage feedback are frequently postulated requirements for model checkers. In our opinion any model checker should also fulfil the following requirements.

**Model building support** is needed because any model checker needs some input language it works with. Support can be given in different flavours. When building models from scratch a visual input language is useful as well as consistency checks. There should also be some well-documented patterns to help the inexperienced user to start building validation models. When validating existing models there should be a way to (semi-) automatically translate or import the existing model into the input language of the model checker.

**Query building support** is frequently given by offering some kind of temporal logic. This surely is a step in the right direction but temporal logic is still hard to use for non computer scientists. Verifying properties sometimes fails not because of errors in the model but because of errors in the formalisation of properties.

More requirements come into play when choosing an appropriate model checker for a given model. We claim that the following requirements should be satisfied for validating models that have been created according to our modelling approach with activity diagram modelling.

**Adequacy of input language.** For model validation to be successful, the description of the original model should be similar to the paradigm of the model checker input language. Ideally the model to be validated should already be described with the input language of the model checker, but this is rarely the case. The reason for this requirement is that the translation from one model to another is an error-prone task and that it is hard to justify that verification results for the transformed model are also amenable to the original model. The bigger the difference in paradigms between the original and the transformed model, the harder is the translation process and the transfer of verification results. For our case study it would be best if the respective model checkers would accept activity diagrams as an input language for checking the

CP and a state machine based input language for checking the DP.

**Adequacy of query language.** As stated above, formalising properties needs support. First of all the query language must offer predicates to relate the atoms of formulae to modelling elements of the input language. Assuming activity diagrams as the input language of the model checker an example for such a predicate is the control flow is in node X. Assuming a state machine based input language such predicates should be capable to capture the current state of a process, the state of message buffers or the value of control variables.

It would be best if the query language of a model checker was extensible because different application domains potentially need different predicates. Regarding the CP of our use case as an application domain it would very helpful to have an open type system for states and for micro-choreographies. As aforementioned, PIPs can be put multiple times in a model, so having the possibility to refer to the type of PIPs would greatly simplify validations that are only related to certain kinds of PIPs, e.g. a PIP of type A is never directly executed after a PIP of type B. Otherwise one would have to collect all occurrences of a PIP of a certain kind in a model by hand which is part of the property to be verified. Another requirement for the efficient formalisation of properties according to our modelling approach is a predicate to capture PIP-executions with a certain result. The formalisation of properties like if a PIP of type A has never been executed with result B then state C is never reached could then easily be done.

Finally the query language should be as expressive as possible. Model checkers frequently only offer some type of propositional temporal logic for query purposes but the availability of predicate logic or a higher order logic would facilitate query building. Clearly, the more expressive a query language is the harder is the verification problem, but this is another topic. Regarding the DP of our use case the formalisation of the *MCP* property that PIPs can successfully be triggered could be facilitated by having the possibility to use quantifiers in logic formulae. Thus the fact that a *request for triggering a PIP* with a fixed, but arbitrary sequence id $x$ is followed by a *permission message to trigger a PIP* with the same sequence id $x$ could be more easily formalised.

## 5.3 Validating the use case

The properties identified in subsection 5.1 have been validated on a pentium 2.4 GHz machine with 1 GB RAM and Windows XP Professional installed. VMWare Workstation 4.5.2 was used to install Fedora Linux Core 3 as a guest OS on this machine. The Fedora installation was assigned 604 MB of RAM. The following subsections discuss the validation of the CP and DP respectively.

### 5.3.1 Validating the centralised perspective

The validation of the CP has been performed using TCM/TATD[13] (version 2.20), a tool that has been extended with prototype model checking functionality by Hendrik Eshuis during a Phd project ([Esh02]). TATD internally uses NuSMV[14] (installed in version 2.1.0) for validation purposes. We used TATD in our case study because it is directly applicable to activity diagrams and it uses a dedicated formal activity diagram semantics ([Esh02]). In principle the requirement of adequacy is therefore met. As a prototype tool TATD does not fully meet some important requirements like simulation capabilities or resource usage feedback but it is sufficient for evaluation purposes. It is necessary to explain the basics of Eshuis' semantics in order to talk about validation with TATD. Roughly speaking, the semantics used is based on the notions of *configuration* and *step*. A configuration is a bag of activity nodes and state nodes that are active at a certain point in time. Steps transform configurations in other configurations by computing and firing maximal sets of transitions that can be fired in parallel. The computation of new configurations is based on the presence of events and the valuation of guards. Non-determinism is used to simulate events of the outside world and to determine the value of guards. The state space of models is explored by computing all possible sequences of configurations.

TATD offers two quite expressive kinds of temporal logic, namely subsets of PLTL and CTL (cf. [BBF01], p. 35-37, for both), for formalising properties. In order to relate formulae to models, TATD offers the IN(*node name*) predicate to state that a node with name *node name* is part of the current configuration and the use of boolean guard variables.

A noteworthy feature of TATD is the usage of *strong fairness*. Speaking in terms of activity diagrams, strong fairness means that if a guard[15] potentially will be evaluated infinitely often or if the presence of an event potentially is tested infinitely often then, eventually, that guard will be evaluated to true or the event will occur. Thus a loop with a guard that never becomes true or an event that never occurs cannot prevent the termination of a process. Generally, a verification run with TATD only considers system runs that are strongly fair. Adding the assumption of TATD that any activity instance will eventually terminate this means that the property of *termination* of processes can potentially be proven to be true with TATD. If the validator does not need or want strong fairness, it can also be turned off during validation.

**Modelling with TATD** TATD does not fully support UML 1.5 activity diagrams nor does it provide XMI import/export functionality. This is why the remodelling of the use case needs some clarification. The following discussion only refers to changes that are relevant for validation purposes. TATD does not support substates in state machine states, so process states are modelled without substates. This makes the formalisation of properties harder because substates cannot be referenced directly but only indirectly via their superstates. The collaboration in itself remains the same.

Events are not transferred from the CP to the TATD model. Thus, if there are multiple outgoing transitions of a process state then TATD chooses the transition to fire non-deterministically. From the point of view of TATD functionality this is nearly the same as if events would be

[13] `http://wwwhome.cs.utwente.nl/~tcm/tatd.html`

[14] `http://nusmv.irst.itc.it/`

[15] contradictions excluded

used in the TATD model because events are produced non-deterministically either. Only events that are generated by firing a transition do not occur non-deterministically, but such events are not part of the CP. So the only difference that emerges from not modelling events is that in verification runs without strong fairness a process cannot be prevented from terminating by an event that never occurs. In the CP events are used to leave process states. If events do not occur a process can block in a particular process state. This is an evident result and does not need support from a model checker. Whether events are modelled in the TATD model or not is therefore irrelevant for the use case of this report. From the point of view of the CP a non-deterministic choice of transitions is also adequate because the *internal process*, which is responsible for generating events, is encapsulated. Assumptions about its behaviour are therefore hard to make. The effect of this encapsulation has already been discussed on page 47. A positive side effect of not modelling events in the TATD model is state space reduction during validation because the order of event generation does not have to be considered.
Finally guards are transferred from the CP to the TATD model as follows. TATD only allows boolean variables within guards. For the use case of this report all guards can be built from boolean variables. As all boolean variables of a TATD model have global scope, modelling guards as presented in figure 15 is intuitive at first sight. The figure shows two sections of the TATD model that determine the result of a PIP execution. As variables have global scope the result of PIP 3A7 is determined by other variables (*techFail_8*, *rejected_8*) as the result of PIP 3A5 (*techFail_9*, *rejected_9*)[16] in order to avoid side effects. But the TATD mechanism for determining variable values does not need that separation. TATD determines the value of variables non-deterministically. This is adequate for the use case of this report because the business logic for evaluating business documents is encapsulated within the *internal process.* A variable in a TATD model is assigned a value by an activity node if the variable is used within a guard of a transition that either ends in the activity node or starts from the activity node. In the case of outgoing transitions the variable is assigned a value before firing a transition, in the case of incoming transitions the variable is assigned a value after firing a transition. Thus variables for determining the result of activity nodes can be reused because each activity node assigns a new value to a reused variable. Note that reuse of variables is only allowed in the use case of this report because the control flow of the use case is never split and thus variables cannot be updated by two different activity nodes at the same time. Figure 16 shows reuse of variables *rejected* and *techFail* for modelling the example of figure 15. Reuse of variables heavily influences execution time of validation runs. This has been empirically deduced by validating simple formulae like $F\ FINAL$ for an extract of the CP with and without reuse of variables. For this particular extract of the CP some validation runs without reuse of variables needed approx. 50 times more time than validation runs with reuse of variables.

**Validation results** Regarding our classification of relevant properties, TATD checks an essential generic property by default, i.e. the reachability of every node and the possibility to fire every transition. Other generic properties like absence of deadlocks have to be encoded with temporal logic. The formula

$$G\ (FINAL \lor (X\ true))$$

[16] *else* is a predefined expression that evaluates to true if all other guards of a decision node evaluate to false

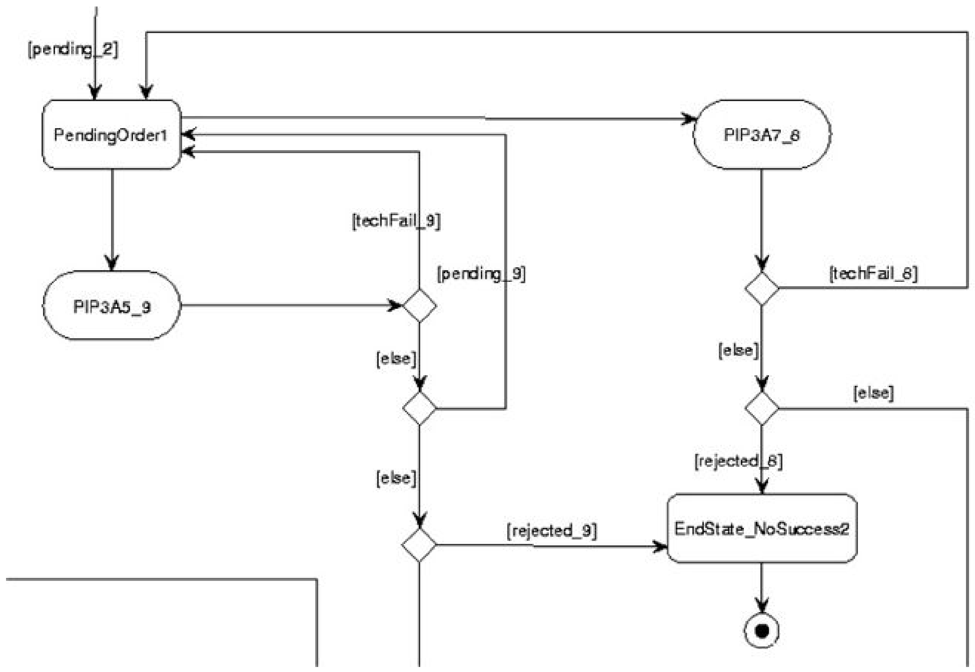

Figure 15: Separate variables for each guard

encodes absence of deadlocks by stating that at any point in time (G[17]) either an end state is reached (constant *FINAL*) or there is a transition to the next state (X true[18]). Absence of deadlocks can be verified for the use case described here.
Termination of processes can be validated by checking formula

$$F\ FINAL$$

with *strong fairness* turned on. In this case the property turns out to be true for the use case described here. If strong fairness is turned off then at least possible termination can be proved be verifying formula

$$E\ F\ FINAL.$$

From a practical point of view it is much more interesting to falsify a formula like

$$G\ \neg IN\,(EndState\_Success)$$

because the falsification of this formula does not only prove that termination is possible but also furnishes a counter example that leads to termination.
Finally, absence of livelocks cannot be verified for the given use case. Livelocks occur when particular states of a system are reached infinitely often. Livelocks cannot be avoided in the use case of this report because the internal processes are modelled non-deterministically. According to the CP it is possible that a buyer triggers PIP 3A5 infinitely often in process state ContractIS and that the execution of PIP 3A5 repeatedly fails. This results in a loop through ContractIS and PIP 3A5. As absence of livelocks cannot be verified at all, at least the whole set of possible loops should be determined in order to give hints for a an adequate implementation of the internal process. When running TATD with strong fairness turned on no loops at all

[17]G(lobally) is implicitly preceded by A(lways), i.e. in all possible runs at any point in time
[18]in the neXt state, the constant *true* holds

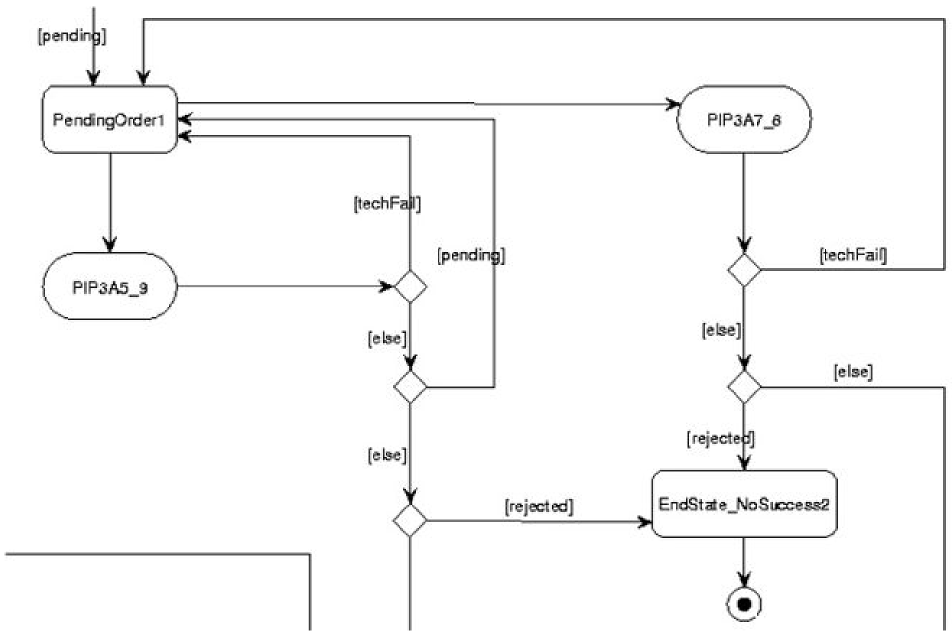

Figure 16: Reuse of variables in guards

can be detected because any guard will eventually become true and any event will eventually be generated. When running TATD with strong fairness turned off always the same loop is detected. In order to detect more loops formulae of the type

$$\neg G\ F\ \ (IN\,(nodeName))$$

can be used. Such a formula states that node *nodeName* cannot be reached infinitely often ($\neg G\ F$). If this is not the case, there must be a livelock that TATD presents as a counter example[19]. Figure 17 shows such a loop that has been detected by validating formula

$$\neg G\ F\ \ (IN\,(PIP3A7\_14))$$

in red and blue colour. But figure 17 also shows that this approach does not detect all possible loops in a process specification because multiple loops could run through the same node. For example the loop consisting of nodes ContractIS, PIP3A7_14, PendingContractChangeSI, PIP3A8_21 contains node PIP3A7_14 as well. All in all manual detection of loops is a very cumbersome task that should be done in an automated manner by a tool instead.

The validation of domain specific and application specific properties can be done with TATD in theory but in practice the formalisation of properties frequently is clumsy because the query language of TATD is not adapted to the specific needs of the use case, i.e. states and PIPs cannot be typed and there's no predicate for referring to a PIP execution with a certain outcome. For example the property *if state ContractIS is ever reached then PIP 3A4 must have been successfully executed before* can be encoded as follows:

*PIP3A4 execution - step 1:*
$F\ a\ \rightarrow\ (\neg\ a\,U\ \ (b\ \vee\ c\ \vee\ d\ \vee\ e))$

[19]Strong fairness has to be turned of therefore.

$$
\begin{aligned}
a &:= IN\,(ContractIS)\\
b &:= IN\,(Pip3A4\_2)\ \wedge\ X\,(\neg\, techFail\ \wedge\ \neg\, rejected\ \wedge\ \neg\, pending)\\
c &:= IN\,(Pip3A4\_5)\ \wedge\ X\,(\neg\, techFail\ \wedge\ \neg\, rejected\ \wedge\ \neg\, pending)\\
d &:= IN\,(PendingOrder1)\\
e &:= IN\,(PendingOrder2)
\end{aligned}
$$

*PIP3A4 execution - step 2:*

$$
\begin{aligned}
&F\ f\ \rightarrow\ (\neg\, f\ U\ g)\\
f &:= IN\,(PendingOrder1)\ \vee\ IN\,(PendingOrder2)\\
g &:= IN\,(Pip3A4\_2)\ \vee\ IN\,(Pip3A4\_5)
\end{aligned}
$$

The first step tests the reachability of node ContractIS without neither successful execution of a PIP of type PIP3A4 (formula b, c) nor reaching a process state of type PendingOrder (formula d, e). The second step then tests if a state of type PendingOrder can be reached without executing a PIP of type PIP3A4 before. *techFail*, *rejected* and *pending* are boolean variables that represent the result of an execution of PIP3A4. The X operator refers to the state in the next configuration whereas the U operator states that its left-hand-side holds until its right-hand-side holds and that the right-hand-side will hold finally. The semantics of the U operator requires the use of the implications because the case where state ContractIS is never reached would not be allowed otherwise. One may wonder why the outcome of a PIP is not directly encoded with the intended result but with the conjunction of the negations of the alternative results. The reason for this is that each boolean variable of a guard is assigned a truth variable independently. This means that for example variable *rejected* and variable *pending* for determining the outcome of PIP3A4_2 can be true at the same time. The actual result then depends on the order in which the variables are tested.

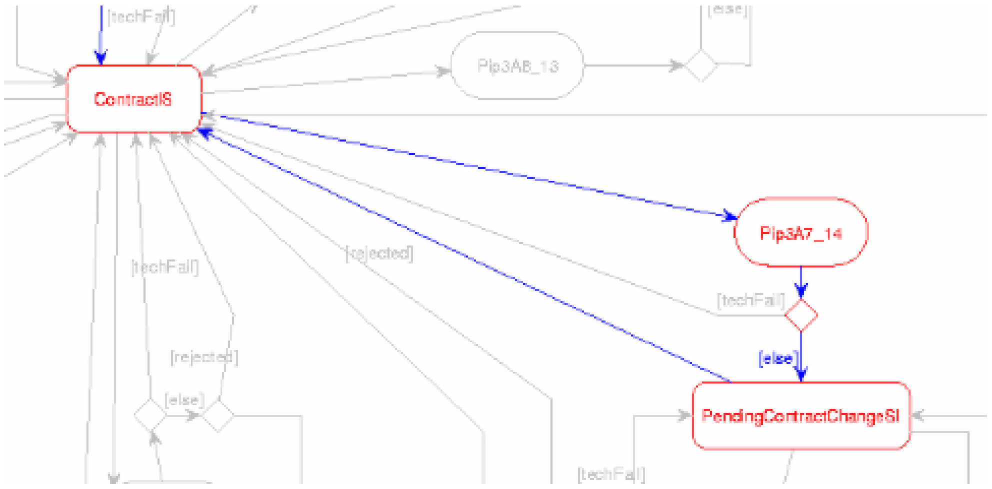

Figure 17: A loop through PIP 3A7_14

**Judgement** The verification of generic properties is supported by TATD to a large extent. The verification of domain and application specific properties with TATD basically is possible as well but the formalisation of properties is not sufficiently supported. Availability of substates for process states and availability of types for activities and states is needed for formalising properties more efficiently and more reliably. In practice, the way of formalising domain and application specific properties presented in this report is too error-prone and cumbersome. Further, simulation capability and detailed resource usage feedback would be needed in practice. When judging TATD, it must be considered that it is a prototype implementation and that implementations of various UML activity diagram semantics are rather rare. Serious alternatives to using TATD could not be found.

### 5.3.2 Validating the distributed perspective

As already pointed out in subsection 5.1.2, the validation of the DP is only performed at the level of application independent infrastructure functionality, i.e. the properties identified for MCP and PIP execution protocol in table 1 and 2 are validated. The validation has been performed on the machine described on page 53. SPIN in version 4.2.5 has been chosen as model checker for validation. Xspin in version 4.2.3 has been used as a visual frontend to SPIN. SPIN suits well for validating the MCP and PIP execution protocol specifications because the paradigm of SPIN of communicating automata is basically the paradigm of these specifications. SPIN is a widely spread and well documented model checker. A detailed description of SPIN functionality and in particular a detailed description of SPIN's input language Promela is omitted here. A detailed introduction to SPIN can be found in the SPIN reference ([Hol03]) and on the SPIN homepage[20]. In the following, knowledge of SPIN and Promela is presumed.

Duplicate messages are not included in the validation models of the MCP and the PIP execution protocol. As soon as an original message has been sent, a duplicate message can be delivered at any point in time. Thus, the size of validation models that consider duplicate messages is frequently prohibitive for available validation resources. On the other side duplicate messages can easily be handled. The protocol machines cannot discriminate whether an original message or its duplicate message is received first and a duplicate message cannot be delivered before its corresponding original message has been sent. Thus the analysis of duplicate messages can be reduced to the case of receiving a message a second time. Looking at the protocol machines of the use case, it is apparent that any message (i.e. a message type with particular content, not a message instance) causes at most one state transition in the protocol machines. If a message is delivered a second time, it is ignored in the MCP and the PIP execution protocol either ignores the message or sends a message that has already been sent again. Thus, duplicate messages do not have essential influence on the functionality of the protocol machines. Sending a message a second time in case of the PIP execution protocol can be interpreted as a duplicate message as well. Hence, duplicate messages can be left out in the validation models of the use case without limiting the validity of validation results. Clearly this is only true if the argumentation just presented is true. From the model checking method point of view leaving out duplicate messages is not allowed because this constrains the state space of the validation models. This theoretical limitation is accepted in order to fulfil resource restrictions.
Messages from past protocol runs are not included in the validation models either. Such messages can be easily identified looking at the process state field of MCP messages or looking at the PIP instance id field of PIP execution protocol messages. The MCP protocol machines can

[20]`http://spinroot.com`

simply ignore messages from past protocol runs. When receiving a message from a past protocol run, PIP execution protocol machines answer with a *wrong_pip* message that carries the PIP instance id of the received message. *wrong_pip* messages with an outdated PIP instance id are simply discarded by the receiving protocol machine and therefore do not affect the protocol run during which they are sent. Messages from past protocol runs can therefore be excluded from validation models without restricting the validity of validation models. *wrong_pip* messages are excluded from validation models for the same reasons. Clearly, from a methodical point of view, the excluded message types must be considered as not validated and as a potential source for errors.

**MCP** The validation of the MCP is being discussed using the final version of the MCP. Instead of discussing the validation of prior versions of the MCP, a major bug of a prior version is presented that has been detected using model checking.
When the *Participant* of the MCP waits for a *grant* message in state *wait_grant* he can be interrupted by a *lock_req* message of the *Coordinator* that must be handled with higher priority. Intending to alleviate this disadvantage of the *Participant* a prior version of the MCP had the *Participant* remind such an interruption. In case of such an interruption and a time-out event in the subsequent state *wait_pip* the *Participant* should directly return to state *wait_grant* to await the outstanding *grant* message instead of switching to state *checkInternal*. That version did not guarantee mutual exclusion for the right to trigger a PIP execution which can be shown by the following scenario.
The *Participant* begins with sending a *com_req* message and then switches to state *wait_grant* to await a matching *grant* message. The *Coordinator* receives the *com_req* message in state *wait_part*, sends a matching *grant* message back and then switches to state *wait_pip*. There, the *Coordinator* detects a time-out event and then sends a *lock_req* message. As the *lock_req* message travels faster than the *grant* message, the *lock_req* message arrives first. The *Participant* sends a *lock* message when receiving *lock_req* and then switches to state *wait_pip*. There the *Participant* detects a time-out event and then directly switches back to state *wait_grant* because he has been interrupted before. There the *Participant* receives the outstanding *grant* message and thus switches to state *my_pip*. The *Coordinator* receives the *lock* message and thus switches to state *my_pip* too. Hence, both have the right to trigger a PIP.

Before describing the validation of the properties of table 1, a general problem in validating the MCP must be discussed. Theoretically, the validation of the MCP is not possible at all because its model is not finite. This infinity is caused by the use of sequence ids that can grow arbitrarily. The sequence ids correlate request and permission of the right to trigger PIP executions and therefore provide essential functionality of the MCP that cannot be excluded from the validation model. The MCP model can be transformed into a finite model according to the following possibilities.
The first possibility is to define an upper bound for sequence ids. Then the behaviour of the MCP protocol machines must be adapted for the case of reaching that bound. New requests could then always carry the same sequence id or no more requests at all could be sent. In either case the state space of the MCP validation model would be changed such that it allows other system runs than the original MCP model. Thus, any validation results from a model with an upper bound for sequence ids rely on the assumption that the property under consideration is

not affected by that bound.
The second possibility is to build an abstraction of the MCP model that only captures the relations between the sequence ids of the MCP protocol machines. The idea of this approach is that the actual value of a sequence id is not relevant but only if it is the biggest sequence id up to the point of consideration or if it equals another sequence id. These sequence id predicates can be captured by boolean variables that then must be manipulated accordingly during state transitions and process local actions. For example a boolean variable with global scope could be introduced into a PROMELA model that tells whether it is possible that the MCP *Participant* knows the current value of the MCP *Coordinator* sequence id. This variable would have to be set to false whenever the *Coordinator* increments its sequence id locally and it would have to be set to true whenever the *Participant* receives a *lock_req* message. [FJ00] describes how to build an abstraction for a PROMELA model that is quite similar to the MCP model. The problem with building an abstract model like this is that the abstract model allows more system runs than the original model. According to section 2.1 the proof of liveness properties is therefore not admissible using an abstract model. The proof of safety properties is only sound but not complete using an abstract model. Yet another problem is that building the abstract model is an error-prone process which further restricts the validity of validation results. Building abstract models in a sensible way needs formal foundation in order to ensure conformance between abstract and original model. Such foundation is omitted in [FJ00]. [Wol86] also treats abstraction and is frequently cited in this work area. Unfortunately, its results are not applicable because it only handles the abstraction of variables that do not affect the control flow of a protocol.

**MCP, validation level 1** With respect to the properties to validate, for the first validation level of the MCP an upper bound has been defined for sequence ids to make the validation model finite. According to [BBF01], p. 84, the property of successful triggering a PIP under certain circumstances is not a safety property. If it was a safety property it would have to be decidable in any state of the validation state space without looking at future states. Assume that the MCP *Coordinator* has received a valid *lock* message and that the MCP *Participant* is in state *wait_pip*. If the *Participant* leaves the state because he detects a time-out event, the property is violated. But if the *Participant* stays in *wait_pip* until he receives a *pip* message, the property holds. Hence the property is not a safety property. This, in turn, is a reason for not building an abstract model because only safety properties can be analysed then.
Clearly, using an upper bound for sequence ids restricts the validity of a verified property to protocol runs that only use sequence ids smaller than the bound. When reaching the bound for sequence ids the validation protocol machines are designed to keep on sending requests with the bound as the sequence number because this disables the smallest number of transitions of the original MCP protocol machines.

The complete validation model for the first validation level of the MCP can be found in appendix A in listing 13. The MCP is modelled in PROMELA with a separate process for *Participant* and *Coordinator* respectively. The states of the protocol machines are represented using labels in the process definitions. State transitions are realised using *goto* statements and the property of PROMELA models of being processed top-down. The MCP must consider the need for PIP execution that is declared by the *internal process*. As the *internal process* is encapsulated in this report, non-determinism is used to model the need for PIP execution. Further, the MCP functionality is basically independent of particular PIP types, so the need for PIP execution can be represented by two boolean variables *pComNeed* and *cComNeed* that indicate whether the *Participant* and the *Coordinator* respectively have need for PIP execution or not. Once the need for PIP execution is declared, it is kept until a PIP has been successfully triggered. The code in listing 7 is therefore used to set the boolean variables respectively (the example shows the *Participant* need) when the *Participant* or the *Coordinator* enters state *checkInternal.* According to PROMELA semantics, both branches of the if construct depicted are always executable. Hence the branch to execute is chosen non-deterministically. The first branch sets the need for PIP execution. The second branch simply keeps the old value of the PIP need variable. *cComNeed* and *pComNeed* are not set back to false until the respective process reaches label *my_pip*, i.e. until the need for PIP execution has been satisfied. The validation model thus represents an *internal process* that declares the need for PIP execution at arbitrary points in time and keeps this need until a PIP could be successfully triggered. Even the case of PIP execution that failed technically is represented by this simple model. A technical failure means that the need for PIP execution is not satisfied. As the next time the code in listing 7 is executed potentially results in the first branch to be chosen this case is covered as well. Listing 13 shows this code in lines 62-65 and 117-120.

Listing 7: Non-deterministic declaration of the need for PIP execution

```
if
:: pComNeed = true;
:: skip;
fi;
```

Sequence ids are represented by two process-scope byte variables *mySeqId* and *otSeqId* for each process. *mySeqId* represents the sequence id of the requests of the owning process whereas *otSeqId* keeps the biggest sequence id of a request of the respective other process. A separate message buffer is used for each different message type in order to optimise the PROMELA model. Thus, the order of different message types in a message buffer does not have to be considered during validation (cf. [Hol03], p. 123 f.). The bound of the sequence ids and the size of message buffers is determined by the available computing resources.

The results of validating the PROMELA model in listing 13 are the following. SPIN checks absence of deadlocks by default, so a simple verification run without any PLTL property suffices to prove absence of deadlocks.
The SPIN formula

$$[]\ !\ (participant[1]@my\_pip\ \&\&\ coord[0]@my\_pip)$$

has been used to prove mutual exclusion with respect to the right to trigger PIPs. The SPIN formula corresponds to the PLTL formula $G\ \neg\ (p\ \wedge\ q)$ which means that at no point in time both, $p$ and $q$, may be true. Atoms $p$ and $q$ are used to reference PROMELA labels. To do so the SPIN construct

$$Process\ name\,[Process\ id]\,@Label$$

can be used. *participant*[1]@*my_pip* states that the control flow of the *participant* process (which carries process id 1 as the second process in the PROMELA model) is currently at label

*my_pip*. Being at label *my_pip* corresponds to having the right to trigger PIPs. The property of mutual exclusion can be verified in a SPIN run. Note that SPIN implicitly checks the formula for all possible system runs. This corresponds to an implicit quantification of the formula with the CTL $A$ operator.
The property that PIPs can be successfully triggered if *Coordinator* and *Participant* do not leave *wait_pip* prematurely is the hardest to validate. The first problem is formalising the property in a general way. A PIP execution can be successfully triggered if a request with sequence id $x$ is answered with a permission message with the same sequence id $x$ and if the respective process does not leave state *wait_pip* prematurely. A general formulation of the property would require to make a statement for arbitrary $x$ but this would require the availability of predicate logic style quantifiers. SPIN only offers PLTL. Instead the formula can iteratively be checked for a particular value of $x$. The next problem is that the property cannot be formalised in a single formula because the complexity of the verification problem grows with the length of the formula to verify. For the validation resources available such a formula would be too long. Instead the property is formalised separately for the *Participant* and the *Coordinator* process. Concerning the *Participant*, he can successfully trigger a PIP if he does not leave state *wait_grant* because of a time-out event after having sent a *com_req* message. Further the *Participant* must receive a matching *grant* message and the *Coordinator* mustn't leave *wait_pip* until having received a *pip* message. Finally the *Participant* mustn't be interrupted in state *wait_grant* by a *Coordinator* *lock_req* message. For the purpose of presentation the formula to validate this property is decomposed. The actual formula is given in PLTL syntax whereas its atoms are given in SPIN constructs:

$$\begin{array}{ll}
\textit{TimeoutP:} & G\ ((a\ \wedge\ b)\ \rightarrow\ (\neg\, c\ U\ (d\ \wedge\ e))) \\
 & a\ :=\ coord[0]@wait\_pip \\
 & b\ :=\ coord[0] : otSeqId == 2 \\
 & c\ :=\ coord[0]@checkInternal \\
 & d\ :=\ coord[0]@rec\_pip \\
 & e\ :=\ coord[0] : temp == 2 \\
 & \\
\textit{InterruptP:} & G\ \neg\ f \\
 & f\ :=\ participant[1]@interrupt \\
 & \\
\textit{SuccessP:} & (TimeoutP\ \wedge\ InterruptP)\ \rightarrow\ G\ ((g\ \wedge\ h)\ \rightarrow\ F\ (i\ \wedge\ k)) \\
 & g\ :=\ participant[1]@sent\_com\_req \\
 & h\ :=\ participant[1] : mySeqId == 2 \\
 & i\ :=\ coord[0]@partner\_pip \\
 & k\ :=\ coord[0] : otSeqId == 2
\end{array}$$

Referencing labels with SPIN constructs has already been introduced above. The value of process-scope variables can be captured in a similar way by using the following syntax:

$$Process\ name\,[Process\ id] :\ Variable\ denominator$$

The whole formula *SuccessP* is an implication. The right-hand side formalises successfully triggering a PIP. A PIP can be successfully triggered, if a request with a particular sequence

id sent by the *Participant* ($g \wedge h$), at any point in time ($G$), eventually ($F$) will be followed by the state in which the *Coordinator* has received a PIP with the same sequence id ($i \wedge k$). The left-hand side of the implication captures conditions for a successful triggering of a PIP by the *Participant*. Formula *InterruptP* states that the *Participant* mustn't be interrupted by a *grant* message. Formula *TimeoutP* states that the *Participant* does not leave state *wait_grant* prematurely by means of a time-out event and that the *Coordinator* does not leave sate *wait_pip* prematurely either.
The formula for the *Coordinator* can be formalised analogously, except that the *Interrupt* formula is not needed. A detailed discussion is therefore omitted here.

*TimeoutC:*

$$\begin{aligned}
&G\ ((a\ \wedge\ b)\ \rightarrow\ (\neg\, c\ U\ (d\ \wedge\ e)))\\
&a\ :=\ participant[1]@wait\_pip\\
&b\ :=\ participant[1] : otSeqId == 3\\
&c\ :=\ participant[1]@checkInternal\\
&d\ :=\ participant[1]@rec\_pip\\
&e\ :=\ participant[1] : temp == 3
\end{aligned}$$

*SuccessC:*

$$\begin{aligned}
&TimeoutC\ \rightarrow\ G\ ((f\ \wedge\ g)\ \rightarrow\ F\ (h\ \wedge\ i))\\
&f\ :=\ coord[0]@sent\_lock_req\\
&g\ :=\ coord[0] : mySeqId == 3\\
&h\ :=\ participant[1]@partner\_pip\\
&i\ :=\ participant[1] : otSeqId == 3
\end{aligned}$$

Both formulae, *SuccessP* and *SuccessC*, can be verified if *weak fairness* is turned on during a SPIN run. *Weak fairness* guarantees that a process will not be prevented from taking a single step forever. *Weak fairness* is frequently needed for verifying liveness properties.
Note that successfully verifying formulae *SuccessP* and *SuccessC* is of limited value. Both formulae are implications, i.e. they can only evaluate to false if the left-hand sides of the implications evaluate to true. It can be proved that these left-hand sides can evaluate to true by falsifying the negations of the left-hand sides. Nevertheless, the left-hand sides heavily constrain the set of possible system runs without requiring the processes to communicate. Formulae *SuccessP* and *SuccessC* do not really prove that successfully triggering PIPs can be guaranteed, rather they identify sufficient, not necessarily necessary, conditions for successfully triggering a PIP.

**MCP, validation level 2** The second level of MCP validation adds message duplicates, overtaking messages and long travelling messages to the validation problem. This leads to a heavy increase in size of the validation model that cannot be handled with the available resources. In order to get a sufficiently small validation model, abstraction techniques can be applied according to [FJ00]. Not only restricts such an abstraction strategy the validity of validation results, but it also adds to the distance between original model and validation model as it requires another abstraction step. If this additional abstraction step does not have formal semantics, it is a very error-prone task to build the final validation model. From a practical point of view the benefit of doing so is questionable. This step is therefore omitted here. A model for validation level 2 based on abstraction can be received upon request.

**PIP execution protocol** The PIP execution protocol comprises two variants for appropriately handling Single-Action-Activities and Two-Action-Activities. Both variants have the same goals, so the properties to check are the same too. The complete validation models of the Two-Action variant can be found in appendix A, the models of the Single-Action variant can be received upon request. In this section, only the validation of the Two-Action variant (PIPXP in the following) is discussed in detail. Analogously to the validation of the MCP, the validation of early versions of the PIPXP is omitted. Instead the following major bug that has been detected using model checking is discussed:
As already pointed out above, the intended result of a 2PC run at the end of PIPXP is always *Commit*. This can lead to assuming that *Global Abort* messages are not obligatory. But leaving out *Global Abort* messages leads to more blocking situations than necessary as can be seen from the following scenario. Suppose a communication media that does not lose messages and delivers any message after finite time. Suppose further that a 2PC-Coordinator sends a *Vote Request* several times. As the 2PC-Coordinator does not receive a response message, he finally decides to switch silently (*Global Abort* is excluded from the protocol) to state *fail*. In this situation it could be the case that the 2PC-Participant receives the *Vote Requests* and replies to them with *Vote* messages. After having sent the first *Vote* message, the 2PC-Participant cannot terminate its protocol run without receiving a *Global Commit* (or a *Global Abort*) message. As the transmission of a *Vote* message may take longer than the time-out of the 2PC-Coordinator, the 2PC-Participant would get stuck without *Global Abort* messages despite the fact that every message is eventually delivered.

The validation of PIPXP is basically possible because its state space is finite. Further, PLTL is sufficiently powerful to formalise properties of table 2. In practice, a problem in validating the PIPXP is to avoid exhausting the given resources by keeping the validation model in size, particularly for validation levels two and three. That is why the following modifications have been applied to the PIPXP (state and message identifiers are taken from the protocol machines):

1. In sender's state *vote_requestX* the receipt of *pip_resp* messages is not responded to instead of sending a *vote_req* message. The reason for that is that *vote_req* messages are possibly sent multiple times anyway. Accordingly, the transitions from *vote_request2* to *vote_request1* and from *vote_request3* to *vote_request1* are removed either.

2. Analogously to state *vote_requestX*, in receiver's state *send_responseX* the receipt of *pip_req* messages is not responded to because *pip_resp* messages (triggered by *pip_req* messages in the original model) are possibly sent multiple times anyway. The transitions from *send_response2* to *send_response1* and from *send_response3* to *send_response1* are removed either.

3. In sender's state *wait_globAck*, *glob_c* messages are not sent as a reaction to *vote* messages because *glob_c* messages can be sent multiple times nevertheless.

4. States *send_request3*, *vote_request3* and *send_response3* are removed from the validation model because they just represent a message retransmission. Transitions from these states

to state *fail* are replaced by transitions to state *fail* from *send_request2*, *vote_request2* and *send_response2*.

5. Finally, a modification is applied that is not visible in the protocol machines. When delivering a message to the respective receiver's buffer it is tested if the buffer already contains a message with the same content. This massively reduces the size of message buffers as each distinct message is buffered at most once. Note that the number of different messages amounts to the number of different message types because the PIP instance id can be neglected when analysing a single protocol run. Note also, that this does not exclude duplicate messages from the model because a message can be delivered to a buffer after the consuming process already has taken a message with the same content from the buffer.

Theoretically, the validity of validation results is affected by the measures just described because the state space of the validation allows strictly less protocol runs than the original model. Practically, the loss of validity is marginal because the number of message retransmissions must be limited anyway for model checking purposes. Determining the exact value of this limit is rather a problem of available resources than a problem of protocol correctness. Figures 18 and 19 show the automata actually validated as opposed to the original automata shown in figures 11 and 10.

For presentation purposes the automata depicted in figure 18 and figure 19 are used as a common model for the validation levels identified in table 2. The formalisation of model properties to check is the same for every validation level as well.
The property *consistent outcome* is checked by introducing a separate PROMELA process as depicted in listing 8. The most important part of this process is the *assert* statement. If the boolean expression of a PROMELA *assert* statement is not true when executing the *assert* in a verification run, SPIN detects an error. The expression

$$resultS \; == \; 0 \; || \; resultR \; == \; 0 \; || \; resultS \; == \; resultR$$

represents the condition for a consistent outcome of a protocol run. The content of the expression is that either, the outcome of the sender process (resultS) or of the receiver process (resultR) is not yet decided, or, that the outcome of both processes is the same. According to PROMELA semantics the code for checking the *assert* statement can be executed at any point in time. Thus, if there is a situation that contradicts the consistency condition then this situation will be detected in a verification run. The *atomic* construct is not necessary for ensuring consistency but only an optimisation that reduces the state space of the validation model according to [Rui01], p. 151f.

Listing 8: Process ensuring the consistency of PIPXP runs

```
active proctype consistent(){
end:
  atomic{!(resultS == 0 || resultR == 0 || resultS == resultR) ->
  assert (resultS == 0 || resultR == 0 || resultS == resultR )};
}
```

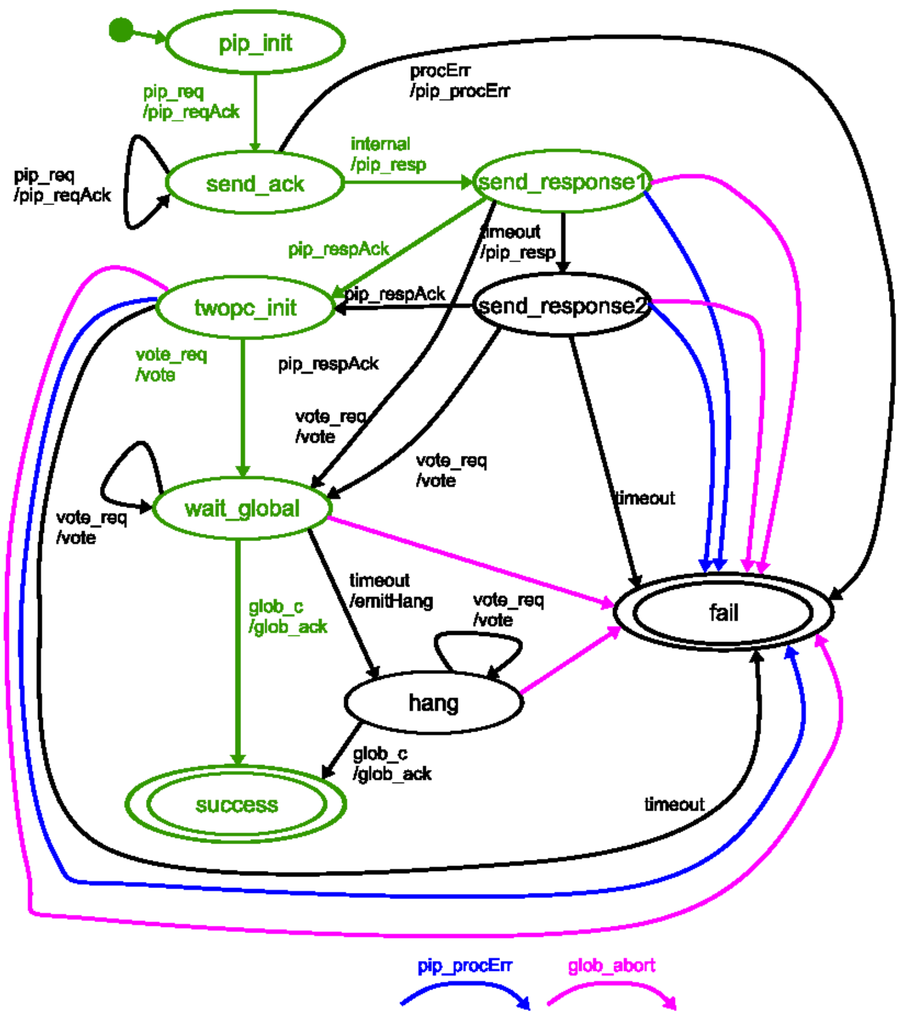

Figure 18: Receiver automaton actually validated

The termination property of table 2 can simply be formalised by introducing a label in the sender and receiver processes and stating that this label must eventually be reached. Using PLTL syntax for the formula and SPIN expressions for its atoms this property can be formalised as follows:

*Termination:* $(F\ p)\ \wedge\ (F\ q)$

$p\ :=\ send[0]@term$

$q\ :=\ rec[1]@term$

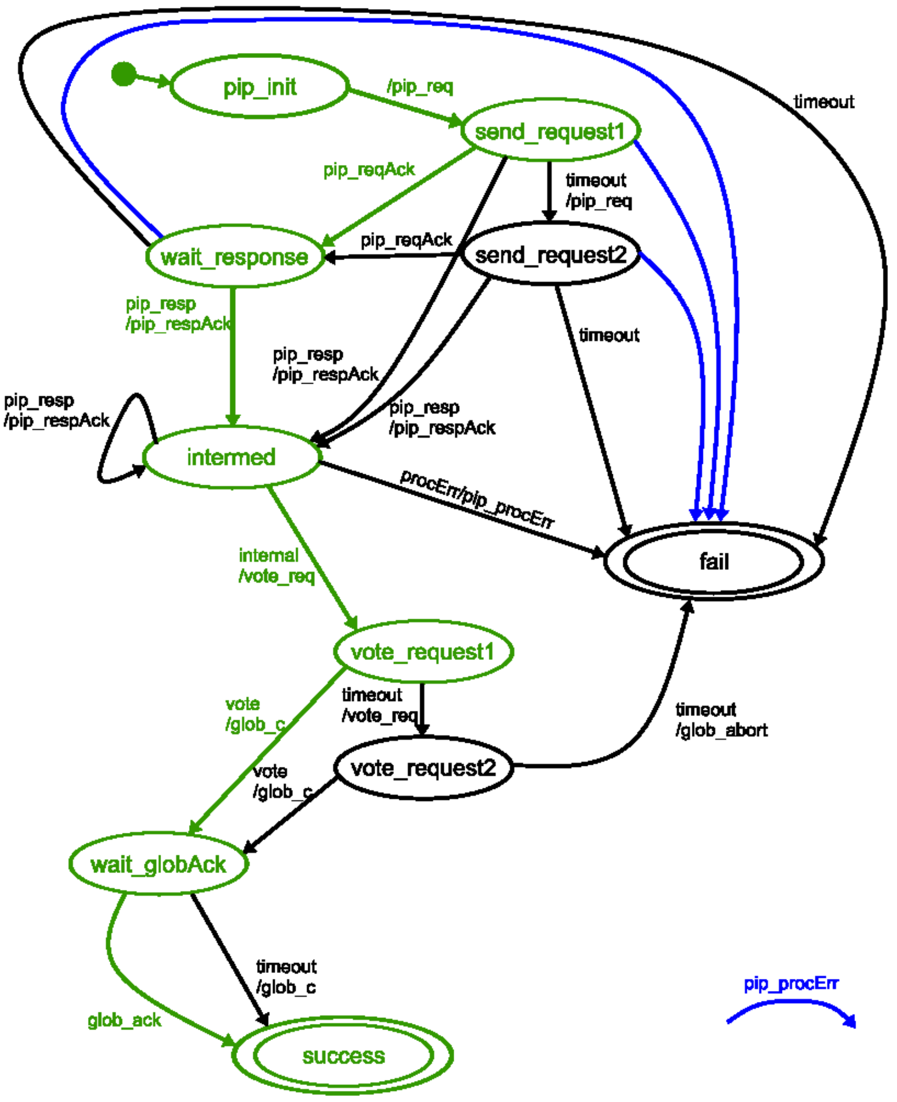

Figure 19: Sender automaton actually validated

This formula must be slightly adapted for the third validation level, because the PIPXP *Receiver* can get stuck in state *hang*:

$$
\begin{array}{ll}
\textit{Termination3:} & (F\ p)\ \wedge\ (F\ (q\ \vee\ r)) \\
 & p\ :=\ \textit{send}[0]@\textit{term} \\
 & q\ :=\ \textit{rec}[1]@\textit{term} \\
 & r\ :=\ \textit{rec}[1]@\textit{hang}
\end{array}
$$

The last property of possible success of PIPXP cannot be directly formalised in PLTL, because PLTL does not support existential expressions. A PIP execution is successful if both result variables indicate success. In order to verify that success is possible, it can equivalently be falsified that the negation of success always holds. Therefore the following formula has to be falsified for proving that success is possible:

$$\begin{array}{ll} \textit{PossibleSuccess:} & G\ \neg\ (p\ \wedge\ q) \\ & p\ :=\ \textit{resultS}\ ==\ 1 \\ & q\ :=\ \textit{resultR}\ ==\ 1 \end{array}$$

**PIP execution protocol, validation level 1** The complete validation model for level 1 can be found in appendix A, listing 14. Sender and receiver are modelled as a separate process each. Labels are used to structure these processes according to their protocol states. Analogously to the MCP validation, the interaction between the PIPXP and the sender's/receiver's internal processes is modelled by means of non-determinism. An example for using non-determinism is depicted in listing 9 that shows the evaluation of the PIP's first business document. According to PROMELA semantics, statements *rToEve!pip_procErr*, representing a processing error, and *goto send_response*, representing successful processing, are always executable. Hence the PROMELA *if* construct chooses non-deterministically between the two options. Note that the first statement is always executable because the buffer of the respective message channel will never be full.

Listing 9: Non-deterministic processing of business documents

```
send_ack:
  do
  :: eveToPipReq?pip_req, temp ->
    if
    :: temp == pipId -> rToEve!pip_reqAck, pipId;
    :: else;
    fi;
  :: rToEve!pip_procErr, pipId; goto failure;
  :: goto send_response;
  od;
```

Analogously to the MCP validation, a separate message channel is defined for each pair of message type and message sender. A difference to the MCP is the introduction of separate processes for message channel manipulation (eve processes in the following). The eve processes' task is to model the insufficiencies of the communication media, particularly for validation levels 2 and 3. Their only task on this validation level is to ensure that messages with same content are at most contained once in a receiver's message buffer. Listing 10 shows the procedure for dispatching *vote_req* messages by an eve process. If there is already a *vote_req* in *Receiver's* message buffer (line 2), the message is discarded. Otherwise (line 3), the message is put into *Receiver's* message buffer.

Listing 10: Filtering messages

```
1    if
2    :: eveToVoteReq?[vote_req, eval(temp)];
3    :: else -> eveToVoteReq!vote_req, temp;
4    fi;
```

Property *Termination* of table 2 can successfully be verified while process *consistent* ensures a consistent outcome of the sender and receiver process. Formula *PossibleSuccess* can be falsified which proves the possibility of successful PIP execution.

**PIP execution protocol, validation level 2** The complete validation model for level 2 can be found in appendix A, listing 15. The difference to validation level 1 is that messages can overtake each other. Listing 11 shows an extension to the eve processes for modelling overtaking messages. In each iteration of an eve process' loop a message can either be processed by the dispatching procedure of level 1 or it can be put at the end of the message queue. Thus a message can be overtaken by any other message. Whether every possible order of message delivery can be produced with the model of listing 11 depends on the number of messages and buffer length. Suppose channel *sToEve* has a buffer length of two, then there is no possibility that the first two messages of the sender are overtaken by all other messages of the sender.

Listing 11: Modelling overtaking messages

```
1 do
2 :: atomic{sToEve?msg, temp -> sToEve!msg, temp};
3 :: if
4                /*Dispatch messages depending on message type*/
5    fi;
6 od;
```

Except for *Termination*, the verification of model properties furnishes the same results as on validation level 1. The problem with verifying *Termination* is caused by the model for overtaking messages. Line 2 of listing 11 is possibly executed infinitely often. Thus arbitrary transmission time is modelled as well, but it shouldn't. To overcome this problem, the *Termination* formula can be extended by a condition that prohibits arbitrary transmission time:

$$\begin{array}{ll} \textit{FiniteMsgTravel:} & (F\ G\ p)\ \wedge\ (F\ G\ q) \\ & p\ :=\ len\,(sToEve) == 0 \\ & q\ :=\ len\,(rToEve) == 0 \end{array}$$

Taking *FiniteMsgTravel* as the precondition of an implication, *Termination* can be proved:

$$FiniteMsgTravel\ \rightarrow\ Termination$$

The formula of *FiniteMsgTravel* is based on the observation that messages in the validation model under consideration have finite transmission time if and only if message channels *sToEve* and *rToEve* eventually ($F$) get empty permanently ($G$). This claim can be justified by the following informal proof[21]:
If a message buffer eventually gets empty permanently, then a message cannot be taken from and put into the same buffer again and again. Otherwise the message buffer would not eventually get empty permanently.

[21]The proof can be done with SPIN for the less complex model Single-Action PIP execution protocol

Sender and receiver only send a finite number of messages. None of these messages may be taken from und put into the same buffer again and again. Hence, rToEve and sToEve eventually get empty permanently.

**PIP execution protocol, validation level 3** Validation level 3 differs from level 2 in arbitrary message transmission times and lost messages. This adds to the complexity of the verification problem. As the validation of level 2 is quite close to exhaust available validation resources, the model of the eve processes is adapted as shown in listing 12. The resulting new model allows messages to be delivered to the receiving process at most once. As multiple delivery of the same message only triggers resending messages and does not cause state changes (cf. above), the use of this new model does not constrain the validity of validation results. The new model contains boolean variables to remember that a message has already been delivered. When receiving a message the eve processes decide non-deterministically whether the message gets lost or not. This is done using a Promela *inline* statement (*chooseVal*). In order to include overtaking messages, line 16 of listing 12 contains a Promela *skip* statement. A message can be prevented from being delivered by repeatedly executing this *skip* statement. Thus, messages can overtake each other. As line 16 can be executed infinitely often, arbitrary transmission times are modelled as well. Note further, that the new model allows for arbitrary message ordering as opposed to the model of validation level 2 (cf. p. 70).

Listing 12: Model of communication media for PIPXP validation level 3

```
1  do
2  :: atomic{sToEve?pip_req, temp -> chooseVal(gotPipReq);};
3  :: atomic{sToEve?pip_respAck, temp -> chooseVal(gotRespAck);};
4  :: atomic{sToEve?vote_req, temp -> chooseVal(gotVoteReq);};
5  :: atomic{sToEve?glob_c, temp -> chooseVal(gotGlobC);};
6  :: atomic{sToEve?glob_abort, temp -> chooseVal(gotGlobAbort);};
7  :: atomic{sToEve?pip_procErr, temp -> chooseVal(gotProcErr);};
8  :: else ->
9    if
10   :: gotPipReq && !sentPipReq -> eveToPipReq!pip_req, temp;
        sentPipReq = true;
11   :: gotRespAck && !sentRespAck -> eveToPipRespAck!pip_respAck,
        temp; sentRespAck = true;
12   :: gotVoteReq && !sentVoteReq -> eveToVoteReq!vote_req, temp;
        sentVoteReq = true;
13   :: gotGlobC && !sentGlobC -> eveToGlobC!glob_c, temp; sentGlobC
        = true;
14   :: gotGlobAbort && !sentGlobAbort -> eveToGlobAbort!glob_abort,
        temp; sentGlobAbort = true;
15   :: gotProcErr && !sentProcErr -> eveToRProcErr!pip_procErr, temp
        ; sentProcErr = true;
16   :: skip; /*Messages can travel arbitrarily long*/
17   fi;
18 od;
19
20 inline chooseVal(cVar){
21   if
22   :: cVar = true;
```

```
  :: skip; /*Messages can be lost*/
  fi;
}
```

Model properties *Termination3* and *PossibleSuccess* can be verified using SPIN. Again a consistent outcome of the sender and the receiver process is ensured by process *consistent*.

**Judgement** All in all, SPIN suits well for validating concurrent systems. The necessity for heavily adapting the MCP emerges from the general need for finite models when using model checking and not from insufficiencies of SPIN. The validation of the MCP shows that there are properties that cannot be formalised because the expressiveness of the SPIN query language PLTL is restricted. Apart from that, SPIN is ready to be used in practice as it offers high-performance verification algorithms, an adequate query language that considers the trade-off between expressiveness and validation complexity, simulation functionality and a comfortable front-end for analysing counter-examples. The main problem in using SPIN (as in using any other model checker) is providing a model that does not exhaust validation resources. This may introduce subtle intricacies as well as validating a system that does not match the communicating automata approach would do.

# 6 Related work

First of all we want to emphasize that the idea of transforming consistent states into consistent states by means of transactional activities is well-known from database theory. But we apply this concept in the domain of B2B integration by defining abstract business states as the common view of collaboration partners on the progress of the collaboration.

Regarding the *centralised perspective*, we defined a business collaboration as a single business process and we modelled *state explicitly as the common view of the collaboration partners on the progress of the collaboration.* Further, we defined transactional *micro-choreographies* to consistently change these process states. To our knowledge, this way of modelling business collaborations has not been proposed before. Nonetheless there are two standards suited for modelling business collaborations that use some similar concepts.

The Web Services Choreography Description Language Version 1.0 *WS-CDL 1.0* [W3C05] is a *Candidate Recommendation* of the W3C[22] that can be used to describe collaborations

> by defining, from a global viewpoint, their [i.e., the collaboration participants] common and complementary observable behaviour; where ordered message exchanges result in accomplishing a common business goal.

([W3C05], section *Abstract*). A WS-CDL description is composed of structural and behavioural elements. *Role types*, *relationship types* and *participant types* are the main structural description elements. A *role type* describes a publicly visible unit of behaviour by referring to WSDL[23] interfaces. A *relationship type* relates exactly two *role types* and optionally identifies subsets of the behaviour of role types that can be used within that *relationship type. Participant types* offer the possibility to combine multiple *role types* in one logical unit.
The main behavioural description elements are *choreographies* and *activities.* These elements refer to structural description elements. The root of a WS-CDL description is a *choreography*[24]. *Choreographies* are composed of *activities.* There are multiple types of activities. The atomic unit of WS-CDL descriptions are *interaction activities* that define the exchange of messages between *role types* and therefore refer to *relationship types. Perform activities* can be used to define the execution of a *choreography* within another *choreography*, which means that *choreographies* can be hierarchically composed. *Ordering structures* can be used to define the control flow of *activities* whereas *work units* can be used to group *activities* together and to conditionally execute them depending on the evaluation of guards.
Before starting the actual comparison between the approach introduced in this report and WS-CDL, remember that the discussion is targeted at the centralised perspective of the approach introduced and note that this perspective does not necessarily use UML activity diagrams. Interaction activities and choreographies resemble micro-choreographies described in this report. A consistent outcome of interaction activities and choreographies can be achieved by setting the *align* attribute or the *coordination* attribute respectively to true. As opposed to our micro-choreographies, a consistent outcome is not required for every interaction/choreography as both

[22]`http://www.w3.org`

[23]`http://www.w3.org/2002/ws/desc/`

[24]Actually, the root is a *package*, but logically a *choreography* can be thought of as the root

*align* and *coordination* default to false. Clearly, this can be specified nonetheless. WS-CDL does not define common states of the collaboration participants that should be achieved after having executed an interaction/choreography, but using the *align* and *coordination* attribute, local variables of collaboration participants can be required to hold the same value. Thus the *process states* of the centralised perspective can be emulated. In this report, events haben been proposed to trigger the execution of micro-choreographies. WS-CDL does not use events to trigger interactions/choreographies but uses an *initiate* attribute for interactions instead. If set to true, the *initiate* indicates that the surrounding choreography can be started by the first message of the interaction. So events and the initiate attribute somehow have the same purpose. The main difference comes into play, when the party to trigger the next micro-choreography or interaction/choreography is not unique and more than one collaboration participant tries to trigger the next micro-choreography or interaction/choreography at the same time. The approach introduced in this paper requires that the right to trigger executions must be negotiated (cf. p. 12) whereas WS-CDL states that in case two interactions are marked as *initiators* the first performed interaction establishes the collaboration (cf. [W3C05], section 5.7 *Choreography Life-line*). As the right to trigger a micro-choreography must be negotiated the approach introduced in this paper can handle concurrent events. WS-CDL runs into problems if the choreography under consideration is not a top-level choreography. If it is top-level, then two choreographies might actually be installed. But if it is not a top-level choreography, then the participants would surely want to stay in the same collaboration. But if so, how is it determined which of two concurrent interactions has been triggered first? One solution is to allow multiple initiating interactions only for top-level choreographies but then events cannot be emulated for process states. Finally there is no built-in support in WS-CDL for distributed time-outs as specified in this report. There is a *timeout* element for interactions, but it applies to interactions and not to process states.
Summing up, most of the concepts of the approach introduced here can be emulated by WS-CDL but the main difference is the level on which both operate. The centralised perspective of this report requires the modeller to think in micro-choreographies, process states, events and distributed time-outs. Particularly, micro-choreographies can be used in an abstract manner. WS-CDL, however, requires the modeller to think in WSDL interfaces, message exchanges and much more technical detail. So it is reasonable to think of the centralised perspective more in terms of business process modelling and to think of WS-CDL more in terms of technical specification.

The second standard suitable for modelling business collaborations is the ebXML BPSS v2.0.2 standard[25]. BPSS models a collaboration based on Business Transactions that exchange business documents and then composes Business Collaborations out of Business Transactions. Business Collaborations can be composed hierarchically. BPSS Business Transactions are quite similar to micro-choreographies of this report and BPSS Business Collaborations resemble compositions of micro-choreographies. Further the notion of state the BPSS specification uses is quite similar to the notion used in this report.

> The state of the Business Collaboration is logical between the parties interacting in a peer-to-peer rather than a controlled environment. The virtual state of the

[25]`http://www.oasis-open.org/committees/tc_home.php?wgabbrev=ebxml-bp`, ebXML BPSS v2.0.2 is not yet an official specification

> Business Collaboration lies with the involved partners. (cf. [Oas06], section 3.4.1)

BPSS also identifies state as a key element of collaborations.

> The choreography is specified in terms of Business States, and transitions between those Business States. (cf. [Oas06], section 3.4.11.1)

Unfortunately, state is not explicitly modelled in BPSS as the common view of the collaboration participants on the progress of the collaboration. This is a major difference to the work presented here. Further, BPSS does not define an event concept for triggering Business Transactions and does not specify the need for a protocol for negotiating the right to trigger Business Transactions. BPSS also defines the need for reliable messaging in section 3.2.

> The ebXML Message Service Specification provides a reliable messaging infrastructure. This is the basis upon which the ebBP technical specification builds its protocol for business state alignment using Business Signals. (cf. [Oas06], section 3.2)

The need for such a service could be overcome by implementing a 2PC on top of an unreliable medium, so defining this requirement constrains the class of distributed systems that BPSS collaborations can be built on. This report shows how a collaboration can be implemented on top of an unreliable communication medium using WSBPEL without requiring the application programmer to code the 2PC implementation himself. Nonetheless, requiring a reliable messaging infrastructure surely is an admissible design decision. Another design decision is that Business Transactions must either fail or succeed from a business point of view.

> A Business Transaction MUST succeed or fail from both a technical and business protocol perspective. (cf. [Oas06], section 3.4.2)

P. 12 of this report explains reasons why such a requirement does not make sense in every collaboration setting.

We defined the *distributed perspective* of this report as the implementation of the CP that uses the CP as its context. We further stated that message passing is a suitable paradigm for realising this implementation. Clearly the idea of implementing a business collaboration using message passing is not new. But the work in this report simplifies the implementation as the global view on business logic is already completely fixed when it comes to do the implementation. Another approach that is quite similar to ours in the sense of defining a context for distributed implementations can be found in [OAS03], but the definition of context is left unspecified. Note further, that as opposed to many other approaches our proposal for implementing the DP does not need a reliable messaging infrastructure nor does it need synchronized clocks. We also proposed the MCP for negotiating the right to trigger the next micro-choreography. Looking at the case study, the assumptions we made surely do not make the task of programming a business collaboration very easy but truly reliable systems are hard to build and the protocol process encapsulates large parts of the complexity.

Regarding the whole approach of modelling business collaborations of this report, ebXML[26] offers similar functionality in the sense of providing a common model of the collaboration (i.e. BPSS as already discussed) and functionality for implementing a distributed system that conforms to that common model (i.e. the ebXML Messaging Services 2.0 [Oas02]). The comparison between the CP of this work and BPSS applies. Further this report does not offer a general-purpose messaging facility as ebXML messaging does but only defines the requirement for ensuring a consistent outcome of micro-choreographies and for negotiating the triggers of these. Standard 2PC and the MCP are proposed for fulfilling this requirement. Particularly, an equivalent to the MCP cannot be found in ebXML.

A lot of related work is done by the Web Services community with respect to composing services (e.g. [BCH05], [SG04]). The composition of services definitely is necessary for integrating businesses but the approaches (excluding WS-CDL) we know all act on the level of single service calls and not on the level of transactional micro-choreographies as we do.

Looking at RosettaNet as our use case, [Dog02] have proposed a framework for executing multiple PIPs, but they did not define a modelling approach for creating PIP compositions. RosettaNet itself has not yet defined a standard for composing PIPs either.

Regarding the validation of UML activity diagrams, the basis for doing so is defining a suitable semantics as Rik Eshuis did. Alternative semantics have been defined ([Por01], [BCR00]), particularly the UML 1.5 standard itself defines an activity diagram semantics in terms of Statecharts. Unfortunately, theses semantics are either not implemented or respective tools are not freely available.

Further, there are some taxonomies of properties for model checking properties ([JM99], [Esh02]), but these differ from ours in being not that detailed or are defined for a different application area.

Regarding the validation of the DP we followed the approach of building an abstract model of the infrastructure protocols and validating these using SPIN. [FBS04] presented a way for applying model-checking directly to WSBPEL specifications. This potentially can be used for analysing how the local business politics of collaboration partners in the distributed perspective interfere with business properties in the centralised perspective which is future work.

Summing up, the main contributions of the work presented in this report are:

- An easy-to-use approach for modelling business collaborations from a centralised perspective that respects a distributed execution environment.
- A clear definition of the tasks that have to be fulfilled when implementing a business collaboration in a distributed environment with respect to the centralised perspective.
- Protocols for negotiating the right to trigger a micro-choreography and for agreement upon distributed time-outs during the execution of a 2PC.
- A case study on RosettaNet using UML activity diagrams and WSBPEL that proves that

[26]`http://www.ebxml.org/`

the modelling approach is viable and the application of model checking techniques to the models produced.

- A taxonomy of properties that should be checked when modelling business collaborations and a collection of model checker requirements.

# 7 Conclusion, practical experience and future work

Our approach proposes the modelling of business collaborations from a centralised and a distributed perspective. For the centralised perspective, a modelling concept based on common business states and transactional micro-choreographies to change these states was introduced. We also showed how this concept can be visualised by UML 1.5 activity diagrams and we described how RosettaNet PIP choreographies can be built following these ideas. Modelling according to stringent rules, as we did in our case study, forms the basis for (semi-) automatic validation and execution of models. For the distributed perspective, the core tasks for building an implementation model were identified and the MCP was introduced for negotiating the triggering of micro-choreographies. In the case study, it was shown how WSBPEL can be used for implementing infrastructure functionality and that a reliable communication medium is not necessarily required. Further, an interface for separating that infrastructure functionality from the *internal process*, i.e. the tasks of generating and interpreting business documents as well as detecting events of the real world and changing the real world, was defined.
We identified and classified core properties that should hold in business collaborations and we identified important requirements that a model checking tool should meet to enable efficient and reliable validation. Evaluation results from using a prototype model checker, TATD, and a state-of-the-art model checker, SPIN, show that, in principle, model checking is helpful for detecting bugs in early design phases. Execution of models can be achieved by defining mappings from process states and micro-choreographies to implementations. Considering our case study, micro-choreographies can be implemented by using RosettaNet PIP implementations. Note that our modelling approach is inherently model-driven. The centralised perspective only specifies the business logic of a collaboration that can be implemented by various technologies, e.g. Web Services and WSBPEL.

Applying model checking to a system requires the system to be appropriately modelled as well as the identification of relevant properties. In this context, *appropriate* means that automatic processing of the model of a system must be possible. Practical experience shows that adhering to this condition already helps in detecting errors. Further, identifying relevant properties forces the modeller to precisely formulate the goals of a model and to align the actual model with these goals.
In this report, WSBPEL was used to model the use case from a distributed perspective. As WSBPEL models are executable if the right binding information is provided, they can also be interpreted as distributed implementations. That is why great care has been applied to the translation of the protocol machines to WSBPEL. Thus, bugs in the WSBPEL specification regarding timers with deadline-valued expressions and regarding the receipt of multiple messages within one *onMessage* event handler of a single *scope* have been detected.
Finally, complexity can be greatly reduced by separating a centralised and a distributed perspective. While, the business logic can be focussed from the centralised perspective, the implementation aspects, particularly with respect to an unreliable communication medium, can be focussed from the distributed perspective.

Future work is required to extend the proposed modelling approach for multi-party collaborations and a hierarchical composition technique should be defined. For validating models, a more detailed taxonomy of business collaboration properties is desirable as well as an intuitive approach to formalise these properties. Access to advanced validation technology for business people is a long-term goal. Moreover it is desirable to know how the local business politics of collaboration partners in the distributed perspective interfere with business properties in the centralised perspective. Applying model checking techniques directly to WSBPEL models therefore should be explored. Finally, researching the strengths and weaknesses of different modelling languages in representing our modelling concept is an interesting area of ongoing work.

## References

[BBF01] B. Berard, M. Bidoit, and A. Finkel et. al. *Systems and Software Verification : Model-Checking Techniques and Tools.* Springer-Verlag, Berlin, 1 edition, August 2001.

[BCH05] D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web service interfaces. In *Proc. of the 14th internat. conference on World Wide Web*, pages 148–159. ACM Press, 2005.

[BCR00] Egon Boerger, Alessandra Cavarra, and Elvinia Riccobene. An ASM Semantics for UML Activity Diagrams. In *AMAST '00: Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology*, pages 293–308, London, UK, 2000. Springer-Verlag.

[CGP99] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, Cambridge, MA, USA, 1999.

[Dam04] Suresh Damodaran. B2B integration over the Internet with XML: RosettaNet successes and challenges. In *Proc. of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 188–195, New York, 2004. ACM Press.

[DH01] M. Dumas and A. H. M. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. *Lecture Notes in Computer Science*, 2185, 2001.

[Dog02] A. Dogac et. al. An ebXML Infrastructure Implementation through UDDI Registries and RosettaNet PIPs. ACM SIGMOD Internat.l Conference on Management of Data, 2002.

[Esh02] H. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling.* PhD thesis, University of Twente, Netherlands, 2002.

[FBS04] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting BPEL web services. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM Press.

[FJ00] Elena Fersman and Bengt Jonsson. Abstraction of Communication Channels in Promela: A Case Study. In *Proceedings of the 10th SPIN Workshop*, pages 187–204, Portland, Oregon, USA, 2000.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker.* Addison-Wesley Pearson Education, September 2003.

[IBM03] IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems. *Business Process Execution Language for Web Services*, 1.1 edition, May 2003.

[ISO04] ISO/IEC. *Information technology - Open-edi reference model.* ISO/IEC, 2 edition, May 2004.

[JM99] W. Janssen and R. Mateescu et. al. Model Checking for Managers. In *Proc. of the 5th and 6th Internat. SPIN Workshops*, pages 92–107. Springer-Verlag, 1999.

[Oas02] Oasis Open. *ebXML Message Service Specification.* Oasis Open, 2.0 edition, April 2002.

[OAS03] OASIS Open. Web Services Composite Application Framework (WS-CAF), 2003.

[Oas06] Oasis Open. *ebXML Business Process Specification Schema Technical Specification.* Oasis Open, 2.0.2 edition, January 2006.

[OMG03] OMG. *OMG Unified Modeling Language Specification.* Object Management Group, Inc., 250 First Ave. Suite 100 Needham, MA 02494, U.S.A., 1.5 edition, March 2003.

[Ora05] Oracle. *Oracle BPEL Process Manager Developers Guide 10g Release 2.* Oracle, June 2005.

[Por01] Ivan Porres. *Modeling and Analyzing Software Behavior in UML.* PhD thesis, Abo Akademi University, Finland, November 2001.

[Ros02] RosettaNet, www.rosettanet.org. *RosettaNet Implementation Framework: Core Specification*, v02.00.01 edition, March 2002.

[Rui01] T. Ruijs. *TOWARDS EFFECTIVE MODEL CHECKING.* PhD thesis, University of Twente, Netherlands, 2001.

[SG04] David Skogan and Roy Gronmo et. al. Web Service Composition in UML. In *Proc. of the Enterprise Distributed Object Computing Conference, Eighth IEEE International (EDOC'04)*, pages 47–57, 2004.

[SO00] W. Sadiq and M. E. Orlowska. Analyzing process models using graph reduction techniques. *Information Systems*, 25(2):117–134, 2000.

[TS02] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems - Principles and Paradigms.* Prentice Hall, Upper Saddle River, NJ, 2002.

[UN/01] UN/Cefact. UN/CEFACT' s Modelling Methodology N090 Revision 10, November 2001.

[W3C05] W3C. *Web Services Choreography Description Language.* W3C, 1.0 edition, November 2005.

[Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 184–193, New York, NY, USA, 1986. ACM Press.

# A Promela validation models

Listing 13: MCP, validation level 1

```
#define SEQ_BOUND       6
#define incr(x)         x++;

mtype = {pip, com_req, grant, lock_req, lock};

chan toPartLockReq = [5] of {mtype, byte};
chan toPartGrant = [5] of {mtype, byte};
chan toPartPip = [5] of {mtype, byte};

chan toCoordComReq = [5] of {mtype, byte};
chan toCoordLock = [5] of {mtype, byte};
chan toCoordPip = [5] of {mtype, byte};

bool cComNeed = false;
bool pComNeed = false;

/* #### COORD #### */
active proctype coord() {
  byte mySeqId = 0, otSeqId = 0, temp;
  bool runOut = false;

  xr toCoordComReq;
  xr toCoordLock;
  xr toCoordPip;

  xs toPartLockReq;
  xs toPartGrant;
  xs toPartPip;

wait_part:
  if
  :: atomic { nempty(toCoordComReq) -> toCoordComReq??com_req,
     temp;}
    if
    :: temp > otSeqId -> otSeqId = temp; toPartGrant!grant,
       otSeqId;
sent_grant: goto wait_pip;
    :: else -> goto checkInternal;
    fi;
  :: atomic {else -> goto checkInternal;}
  fi;

wait_pip:
  do
  :: atomic { nempty(toCoordPip) -> toCoordPip??pip, temp;}
rec_pip:
    atomic {
    if
    :: temp == otSeqId; goto partner_pip;
    :: temp < otSeqId;
    :: temp > otSeqId; assert (temp <= otSeqId);
    fi;
    }
  :: atomic{ else -> goto checkInternal;};
  od;

partner_pip:
  goto checkInternal;

checkInternal:
progress:
  atomic {
  if
  :: cComNeed = true;
  :: skip;
  fi;
  }
  if
  :: atomic{!cComNeed -> goto wait_part;};
  :: atomic{ else -> incr(mySeqId);
    if
    :: mySeqId > SEQ_BOUND -> mySeqId = mySeqId - 1; runOut = true
       ;
    :: else -> skip;
    fi;
    }
    toPartLockReq!lock_req, mySeqId;
sent_lock_req: goto wait_lock;
  fi;

wait_lock:
  do
  :: atomic{ nempty(toCoordLock) -> toCoordLock??lock, temp;};
got_lock:
    atomic {
    if
    :: temp == mySeqId && !runOut -> goto my_pip; /*runOut needs
       to be considered because a lock can be delivered after a
       second lock-req -> SEQID-BOUND*/
    :: temp == mySeqId && runOut -> goto wait_part;
    :: temp < mySeqId;
    :: temp > mySeqId; assert (temp <= mySeqId);
    fi;
    }
  :: atomic {else -> goto wait_part;};
  od;

my_pip:
 toPartPip!pip, mySeqId; cComNeed = false;
 goto wait_part;

term: skip; /*this code is never reached */
```

```
}


/* #### PARTICIPANT #### */
active proctype participant() {
  byte mySeqId = 0, otSeqId = 0, temp;
  bool runOut = false;

  xr toPartLockReq;
  xr toPartGrant;
  xr toPartPip;

  xs toCoordComReq;
  xs toCoordLock;
  xs toCoordPip;

checkInternal:
  atomic {
  if
  :: pComNeed = true;
  :: skip;
  fi;
  }
 do
  :: atomic {nempty(toPartGrant) -> toPartGrant??grant, temp;};
rec_grant_old: skip;
  :: else ->
    if
    :: atomic { !pComNeed -> goto wait_lock;};
    :: atomic {else -> incr(mySeqId);
      if
      :: mySeqId > SEQ_BOUND -> mySeqId = mySeqId - 1; runOut =
         true;
      :: else -> skip;
      fi;
      }
      toCoordComReq!com_req, mySeqId;
sent_com_req: goto wait_grant;
    fi;
  od;

wait_lock:
  do
  :: atomic { nempty(toPartLockReq) -> toPartLockReq??lock_req,
     temp;}
    if
    :: temp > otSeqId -> otSeqId = temp; toCoordLock!lock, otSeqId
       ; goto wait_pip;
    :: else;
    fi;
  :: else -> goto checkInternal;
  od;

```

```
wait_pip:
  do
  :: atomic { nempty(toPartPip) -> toPartPip??pip, temp;}
rec_pip:    atomic {
     if
     :: temp == otSeqId -> goto partner_pip;
     :: temp < otSeqId;
     :: temp > otSeqId; assert (temp <= otSeqId);
     fi;
     }
  :: atomic { else -> goto checkInternal; }
  od;

wait_grant:
  do
  :: atomic { nempty(toPartLockReq) -> toPartLockReq??lock_req,
      temp;};
     if
     :: temp > otSeqId -> otSeqId = temp;
interrupt: toCoordLock!lock, otSeqId; /*interrupt = true;*/ goto
    wait_pip;
     :: else;
     fi;
  :: else ->
     if
     :: atomic { nempty(toPartGrant) -> toPartGrant??grant, temp;};
rec_grant:      atomic{
       if
       :: temp == mySeqId && !runOut -> goto my_pip; /*runOut needs
            to be considered because a lock can be delivered after a
            second lock-req -> SEQID-BOUND*/
       :: temp == mySeqId && runOut -> goto checkInternal;
       :: temp < mySeqId;
       :: temp > mySeqId; assert (temp <= mySeqId);
       fi;
       }
     :: else -> goto checkInternal;
     fi;
  od;

partner_pip:
  goto checkInternal;

my_pip:
  toCoordPip!pip, mySeqId; pComNeed = false; goto wait_lock;

term: skip;  /*this code is never reached */

}
```

Listing 14: Two-Action PIP execution protocol, validation level 1

```
#define doTimeout        else
#define E_BUF    2
#define incr(x) x++;

mtype = {pip_req , pip_reqAck , pip_resp , pip_respAck , pip_procErr ,
    vote_req , vote , glob_c , glob_abort , glob_ack};

/* to eve */
chan sToEve = [3] of {mtype, byte};
chan rToEve = [3] of {mtype, byte};

/* eve to Rec */
chan eveToPipReq = [E_BUF] of {mtype, byte};
chan eveToPipRespAck = [E_BUF] of {mtype, byte};
chan eveToVoteReq = [E_BUF] of {mtype, byte};
chan eveToGlobC = [E_BUF] of {mtype, byte};
chan eveToGlobAbort = [E_BUF] of {mtype, byte};
chan eveToRProcErr = [E_BUF] of {mtype, byte};

/*eve to Send*/
chan eveToPipReqAck = [E_BUF] of {mtype, byte};
chan eveToPipResp = [E_BUF] of {mtype, byte};
chan eveToVote = [E_BUF] of {mtype, byte};
chan eveToGlobAck = [E_BUF] of {mtype, byte};
chan eveToSProcErr = [E_BUF] of {mtype, byte};

byte resultS = 0, resultR = 0; /*0=Undecided, 1=Success, 2=Failed
    */

/*################### SENDER ####################*/
active proctype send(){
  byte pipId = 1;
  byte retries = 0;
  byte temp;

pip_init: sToEve!pip_req , pipId; incr(retries);


send_request:
  do
  :: atomic{nempty(eveToSProcErr) -> eveToSProcErr?pip_procErr ,
      temp;}
     if
     :: temp == pipId ->  goto failure;
     :: else;/*other process must be in fail anyway*/
     fi;
  :: else ->
     if
     :: atomic {nempty(eveToPipReqAck) -> eveToPipReqAck?pip_reqAck
         , temp;};
        if
        :: temp == pipId -> goto wait_response;
        :: else;
        fi;
      :: atomic {nempty(eveToPipResp) -> eveToPipResp?pip_resp, temp
        ;};
        if
        :: temp == pipId -> sToEve!pip_respAck, pipId; goto intermed
           ;
        :: else;
        fi;
      :: doTimeout ->
        if
        :: d_step{retries < 2; incr(retries);} sToEve!pip_req, pipId
           ;
        :: retries == 2; goto failure;
        fi;
      fi;
  od;


wait_response:
  do
  :: atomic{nempty(eveToSProcErr) -> eveToSProcErr?pip_procErr,
     temp;
    if
    :: temp == pipId -> goto failure;
    :: else;
    fi;}
  :: else ->
    if
    :: atomic{ nempty(eveToPipResp) -> eveToPipResp?pip_resp, temp
       ;};
      if
      :: temp == pipId -> sToEve!pip_respAck, pipId; goto intermed
         ;
      :: else;
      fi;
    :: doTimeout -> goto failure;
    fi;
  od;


intermed:
  do
  :: atomic{ nempty(eveToPipResp) -> eveToPipResp?pip_resp, temp
     ;};
    if
    :: temp == pipId -> sToEve!pip_respAck, pipId;
    :: else;
    fi;
  :: sToEve!pip_procErr, pipId; goto failure; /*processingError*/
  :: goto vote_request;
  od;
```

```


vote_request:
  sToEve!vote_req, pipId;
  retries = 1;

  do
/*  :: atomic{ nempty(eveToPipResp) -> eveToPipResp?pip_resp, temp
   ;};
     if
     :: temp == pipId -> sToEve!vote_req, pipId; goto vote_request;
     :: else;
     fi; */ /*#CHANGE# vote_reqs are being sent multiple times
        anyway */
  :: atomic{ nempty(eveToVote) -> eveToVote?vote, temp;};
     if
     :: d_step{ temp == pipId -> retries = 0;} sToEve!glob_c, pipId
        ; goto wait_globAck;
     :: else;
     fi;
  :: doTimeout ->
     if
     :: d_step{retries < 2; incr(retries);} sToEve!vote_req, pipId;
     :: d_step{retries == 2; retries = 0;} sToEve!glob_abort, pipId
        ; goto failure;
     fi;
  od;


wait_globAck:
  retries = 1;

  do
  :: atomic{ nempty(eveToGlobAck) ->  eveToGlobAck?glob_ack, temp
      };
     if
     :: atomic{temp == pipId -> goto success;}
     :: else;
     fi;
/*  :: atomic {nempty(eveToVote) -> eveToVote?vote, temp};
     if
     :: temp == pipId -> sToEve!glob_c, pipId;
     :: else;
     fi;*/ /*#CHANGE# glob_cs are being sent multiple times anyway
        */
  :: doTimeout ->
     if
     :: retries < 2 -> sToEve!glob_c, pipId; incr(retries);
     :: atomic{retries == 2; goto success;}
     fi;
  od;


```

```
failure : resultS = 2; goto term ;

success: resultS = 1; goto term;

term: skip;
}


/*################ RECEIVER ################*/
active proctype rec(){
  byte pipId = 0; /*to be initialised*/
  byte retries = 0;
  byte temp;

pip_init:
  if
  :: atomic{nempty(eveToPipReq) -> eveToPipReq?pip_req, temp;};
     pipId = temp; rToEve!pip_reqAck, temp; goto send_ack;
  :: doTimeout -> goto failure;
  fi;

send_ack:
  do
  :: eveToPipReq?pip_req, temp ->
     if
     :: temp == pipId -> rToEve!pip_reqAck, pipId;
     :: else;
     fi;
  :: rToEve!pip_procErr, pipId; goto failure;
  :: goto send_response;
  od;


send_response:
  rToEve!pip_resp, temp; retries = 1;

  do
  :: atomic{nempty(eveToRProcErr) -> eveToRProcErr?pip_procErr,
     temp;};
     if
     :: temp == pipId -> atomic{retries = 0; goto failure;}
     :: else;
     fi;
  :: atomic{nempty(eveToGlobAbort) -> eveToGlobAbort?glob_abort,
     temp;};
     if
     :: temp == pipId -> atomic{retries = 0; goto failure;}
     :: else;
     fi;
  :: else ->
     if
/*     :: atomic{nempty(eveToPipReq) -> eveToPipReq?pip_req, temp
   ;};
```

```
        if
        :: temp == pipId -> rToEve!pip_resp, pipId; goto
           sendResponse;
        :: else;
        fi; */ /* #CHANGE# pip_resps are being sent multiple times
           anyway*/
      :: atomic{nempty(eveToPipRespAck) -> eveToPipRespAck?
         pip_respAck, temp;};
        if
        :: temp == pipId -> atomic{retries = 0; goto twopc_init;}
        :: else;
        fi;
      :: atomic{nempty(eveToVoteReq) -> eveToVoteReq?vote_req, temp
         ;};
        if
        :: temp == pipId -> rToEve!vote, pipId; atomic{retries = 0;
           goto wait_global;}
        :: else;
        fi;
      :: doTimeout ->
        if
        :: retries < 2; rToEve!pip_resp, pipId; incr(retries);
        :: atomic{retries == 2; retries = 0;} goto failure;
        fi;
      fi;
    od;

twopc_init:
  do
  :: atomic{nempty(eveToRProcErr) -> eveToRProcErr?pip_procErr,
     temp;};
    if
    :: temp == pipId -> goto failure;
    :: else;
    fi;
  :: atomic{nempty(eveToGlobAbort) -> eveToGlobAbort?glob_abort,
     temp;};
    if
    :: temp == pipId -> goto failure;
    :: else;
    fi;
  :: else ->
    if
    :: atomic{ nempty(eveToVoteReq) -> eveToVoteReq?vote_req, temp
       ;};
      if
      :: temp == pipId -> rToEve!vote, pipId; goto wait_global;
      :: else;
      fi;
    :: doTimeout -> goto failure;
    fi;
  od;

```

```
wait_global:
  do
  :: atomic{nempty(eveToGlobAbort) -> eveToGlobAbort?glob_abort ,
     temp;};
     if
     :: temp == pipId -> goto failure;
     :: else;
     fi;
  :: else ->
     if
     :: atomic{nempty(eveToGlobC) -> eveToGlobC?glob_c , temp;};
       if
       :: temp == pipId -> rToEve!glob_ack , pipId; goto success;
       :: else;
       fi;
     :: atomic{ nempty(eveToVoteReq) -> eveToVoteReq?vote_req , temp
        ;};
       if
       :: temp == pipId -> rToEve!vote , pipId;
       :: else;
       fi;
     :: doTimeout -> goto hang; /*hang notification omitted*/
     fi;
  od;

hang:
  do
  :: atomic{nempty(eveToGlobC) -> eveToGlobC?glob_c , temp;};
     if
     :: temp == pipId -> rToEve!glob_ack , pipId; goto success;
     :: else;
     fi;
  :: atomic{nempty(eveToGlobAbort) -> eveToGlobAbort?glob_abort ,
     temp;};
     if
     :: temp == pipId -> goto failure;
     :: else;
     fi;
  :: else ->
     if
     :: atomic{nempty(eveToVoteReq) -> eveToVoteReq?vote_req , temp
        ;};
       if
       :: temp == pipId -> rToEve!vote , pipId;
       :: else;
       fi;
     :: else -> skip;
     fi;
  od;

failure: atomic{ resultR = 2; goto term };

success: atomic{ resultR = 1; goto term};
```

```

term: skip;
}

/* ############# SENDER TO RECEIVER ############# */
active proctype eveSToR(){
xs eveToPipReq;
xs eveToPipRespAck;
xs eveToVoteReq;
xs eveToGlobC;
xs eveToGlobAbort;
xs eveToRProcErr;
xr sToEve;

byte temp = 0;
mtype msg = 0;

end:  do
  :: atomic{ sToEve?pip_req, temp ->
    if
    :: eveToPipReq?[pip_req, eval(temp)];
    :: else -> eveToPipReq!pip_req, temp;
    fi;}
  :: atomic{ sToEve?pip_respAck, temp ->
    if
    :: eveToPipRespAck?[pip_respAck, eval(temp)];
    :: else -> eveToPipRespAck!pip_respAck, temp;
    fi;}
  :: atomic{ sToEve?vote_req, temp ->
    if
    :: eveToVoteReq?[vote_req, eval(temp)];
    :: else -> eveToVoteReq!vote_req, temp;
    fi;}
  :: atomic{ sToEve?glob_c, temp ->
    if
    :: eveToGlobC?[glob_c, eval(temp)];
    :: else -> eveToGlobC!glob_c, temp;
    fi;}
  :: atomic{ sToEve?glob_abort, temp -> eveToGlobAbort!glob_abort,
      temp;}
  :: atomic{ sToEve?pip_procErr, temp -> eveToRProcErr!pip_procErr
      , temp;}
  od;
}


/* ############# RECEIVER TO SENDER ############# */
active proctype eveRToS(){
xs eveToPipReqAck;
xs eveToPipResp;
xs eveToVote;
xs eveToGlobAck;
xs eveToSProcErr;
```

```

xr rToEve;

byte temp = 0;
mtype msg = 0;

end:  do
  :: atomic{ rToEve?pip_reqAck, temp; ->
     if
     :: eveToPipReqAck?[pip_reqAck, eval(temp)];
     :: else -> eveToPipReqAck!pip_reqAck,temp;
     fi;}
  :: atomic{ rToEve?pip_resp, temp; ->
     if
     :: eveToPipResp?[pip_resp, eval(temp)];
     :: else -> eveToPipResp!pip_resp,temp;
     fi;}
  :: atomic{ rToEve?vote, temp; ->
     if
     :: eveToVote?[vote, eval(temp)];
     :: else -> eveToVote!vote,temp;
     fi;}
  :: atomic{ rToEve?glob_ack, temp; -> eveToGlobAck!glob_ack,temp
      ;}
  :: atomic{ rToEve?pip_procErr, temp; -> eveToSProcErr!
      pip_procErr,temp;}
  od;
}


active proctype consistent(){
end:
  atomic{ !(resultS == 0 || resultR == 0 || resultS == resultR) ->
       assert (resultS == 0 || resultR == 0 || resultS == resultR )
      };
}
```

Listing 15: Two-Action PIP execution protocol, validation level 2

```
#define doTimeout        else
#define E_BUF    2
#define incr(x) x++;


mtype = {pip_req, pip_reqAck, pip_resp, pip_respAck, pip_procErr,
    vote_req, vote, glob_c, glob_abort, glob_ack};

/* to eve */
chan sToEve = [3] of {mtype, byte};
chan rToEve = [3] of {mtype, byte};

/* eve to Rec */
chan eveToPipReq = [E_BUF] of {mtype, byte};
chan eveToPipRespAck = [E_BUF] of {mtype, byte};
chan eveToVoteReq = [E_BUF] of {mtype, byte};
chan eveToGlobC = [E_BUF] of {mtype, byte};
chan eveToGlobAbort = [E_BUF] of {mtype, byte};
chan eveToRProcErr = [E_BUF] of {mtype, byte};

/*eve to Send*/
chan eveToPipReqAck = [E_BUF] of {mtype, byte};
chan eveToPipResp = [E_BUF] of {mtype, byte};
chan eveToVote = [E_BUF] of {mtype, byte};
chan eveToGlobAck = [E_BUF] of {mtype, byte};
chan eveToSProcErr = [E_BUF] of {mtype, byte};

byte resultS = 0, resultR = 0; /*0=Undecided, 1=Success, 2=Failed
    */

/*################### SENDER ###################*/
active proctype send(){
  byte pipId = 1;
  byte retries = 1;
  byte temp = 1;

pip_init: sToEve!pip_req, pipId;


send_request:
  do
  :: atomic{nempty(eveToSProcErr) -> eveToSProcErr?pip_procErr,
      temp;
     if
     :: temp == pipId ->  goto failure;
     :: else;/*other process must be in fail anyway*/
     fi;}
  :: else ->
     if
     :: atomic {nempty(eveToPipReqAck) -> eveToPipReqAck?pip_reqAck
         , temp;
```

```
        if
        :: temp == pipId -> goto wait_response;
        :: else;
        fi;}
     :: atomic {nempty(eveToPipResp) -> eveToPipResp?pip_resp, temp
        ;};
        if
        :: temp == pipId -> sToEve!pip_respAck, pipId; goto intermed
           ;
        :: else;
        fi;
     :: doTimeout ->
        if
        :: d_step{retries < 2; incr(retries);} sToEve!pip_req, pipId
           ;
        :: retries == 2; goto failure;
        fi;
     fi;
  od;


wait_response:
  do /* loop is needed to remove messages from former protocol
        runs*/
  :: atomic{nempty(eveToSProcErr) -> eveToSProcErr?pip_procErr,
      temp;
     if
     :: temp == pipId -> goto failure;
     :: else;
     fi;}
  :: else ->
     if
     :: atomic{ nempty(eveToPipResp) -> eveToPipResp?pip_resp, temp
        ;};
        if
        :: temp == pipId -> sToEve!pip_respAck, pipId; goto intermed
           ;
        :: else;
        fi;
     :: doTimeout -> goto failure;
     fi;
  od;


intermed:
  do
  :: atomic{ nempty(eveToPipResp) -> eveToPipResp?pip_resp, temp
      ;};
     if
     :: temp == pipId -> sToEve!pip_respAck, pipId;
     :: else;
     fi;
  :: sToEve!pip_procErr, pipId; goto failure; /*processingError*/
  :: goto vote_request;
  od;


vote_request:
  sToEve!vote_req, pipId;
  retries = 1;

  do
/*  :: atomic{ nempty(eveToPipResp) -> eveToPipResp?pip_resp, temp;};
    if
    :: temp == pipId -> sToEve!vote_req, pipId; goto vote_request;
    :: else -> skip;
    fi; */ /*#CHANGE# vote_reqs are being sent multiple times anyway*/
  :: atomic{ nempty(eveToVote) -> eveToVote?vote, temp;};
    if
    :: d_step{ temp == pipId -> retries = 0;} sToEve!glob_c, pipId; goto wait_globAck;
    :: else;
    fi;
  :: doTimeout ->
    if
    :: d_step{retries < 2; incr(retries);} sToEve!vote_req, pipId;
    :: d_step{retries == 2; retries = 0;} sToEve!glob_abort, pipId; goto failure;
    fi;
  od;


wait_globAck:
  retries = 1;

  do
  :: atomic{ nempty(eveToGlobAck) ->  eveToGlobAck?glob_ack, temp;
    if
    :: temp == pipId -> goto success;
    :: else;
    fi;}
/*  :: atomic {nempty(eveToVote) -> eveToVote?vote, temp};
    if
    :: temp == pipId -> sToEve!glob_c, pipId;
    :: else -> skip;
    fi;*/ /*#CHANGE# glob_cs are being sent multiple times anyway*/
  :: doTimeout ->
    if
    :: retries < 2 -> sToEve!glob_c, pipId; incr(retries);
    :: atomic{retries == 2; goto success;}
    fi;
  od;

```

```

failure : atomic{resultS = 2; goto term ;}

success: resultS = 1;

term: skip;
}


/*################# RECEIVER #################*/
active proctype rec(){
  byte pipId = 0; /*to be initialised*/
  byte retries = 1; /*the first message transmission is already
      accounted*/
  byte temp = 1;

pip_init:
  if
  :: atomic{nempty(eveToPipReq) -> eveToPipReq?pip_req, temp;
    pipId = temp;}; rToEve!pip_reqAck, temp; goto send_ack;
  :: doTimeout -> goto failure;
  fi;

send_ack:
  do
  :: eveToPipReq?pip_req, temp ->
    if
    :: temp == pipId -> rToEve!pip_reqAck, pipId;
    :: else;
    fi;
  :: rToEve!pip_procErr, pipId; goto failure;
  :: goto send_response;
  od;


send_response:
  rToEve!pip_resp, temp;

  do
  :: atomic{nempty(eveToRProcErr) -> eveToRProcErr?pip_procErr,
     temp;
    if
    :: temp == pipId -> retries = 1; goto failure;
    :: else;
    fi;}
  :: atomic{nempty(eveToGlobAbort) -> eveToGlobAbort?glob_abort,
     temp;
    if
    :: temp == pipId -> retries = 1; goto failure;
    :: else;
    fi;}
  :: else ->
    if
```

```
/*      :: atomic{nempty(eveToPipReq) -> eveToPipReq?pip_req, temp
    ;};
        if
        :: temp == pipId -> rToEve!pip_resp, pipId; goto
            sendResponse;
        :: else -> rToEve!wrong_pip, temp;
        fi; */ /*#CHANGE# pip_resps are being sent multiple times
            anyway*/
      :: atomic{nempty(eveToPipRespAck) -> eveToPipRespAck?
          pip_respAck, temp;
        if
        :: temp == pipId -> retries = 1; goto twopc_init;
        :: else;
        fi;}
      :: atomic{nempty(eveToVoteReq) -> eveToVoteReq?vote_req, temp
          ;};
        if
        :: temp == pipId -> rToEve!vote, pipId; atomic{retries = 1;
            goto wait_global;}
        :: else;
        fi;
      :: doTimeout ->
        if
        :: retries < 2; rToEve!pip_resp, pipId; incr(retries);
        :: atomic{retries == 2; retries = 1; goto failure;}
        fi;
      fi;
  od;

twopc_init:
  do
   :: atomic{nempty(eveToRProcErr) -> eveToRProcErr?pip_procErr,
       temp;
      if
      :: temp == pipId -> goto failure;
      :: else;
      fi;}
   :: atomic{nempty(eveToGlobAbort) -> eveToGlobAbort?glob_abort,
       temp;
      if
      :: temp == pipId -> goto failure;
      :: else;
      fi;}
   :: else ->
      if
      :: atomic{ nempty(eveToVoteReq) -> eveToVoteReq?vote_req, temp
          ;};
        if
        :: temp == pipId -> rToEve!vote, pipId; goto wait_global;
        :: else;
        fi;
      :: doTimeout -> goto failure;
      fi;
  od;

wait_global:
  do
  :: atomic{nempty(eveToGlobAbort) -> eveToGlobAbort?glob_abort,
     temp;
    if
    :: temp == pipId -> goto failure;
    :: else;
    fi;}
  :: else ->
    if
    :: atomic{nempty(eveToGlobC) -> eveToGlobC?glob_c, temp;};
      if
      :: temp == pipId -> rToEve!glob_ack, pipId; goto success;
      :: else;
      fi;
    :: atomic{ nempty(eveToVoteReq) -> eveToVoteReq?vote_req, temp
       ;};
      if
      :: temp == pipId -> rToEve!vote, pipId;
      :: else;
      fi;
    :: doTimeout -> goto hang; /*hang notification omitted*/
    fi;
  od;

hang:
  do
  :: atomic{nempty(eveToGlobC) -> eveToGlobC?glob_c, temp;};
    if
    :: temp == pipId -> rToEve!glob_ack, pipId; goto success;
    :: else;
    fi;
  :: atomic{nempty(eveToGlobAbort) -> eveToGlobAbort?glob_abort,
     temp;
    if
    :: temp == pipId -> goto failure;
    :: else;
    fi;}
  :: else ->
    if
    :: atomic{nempty(eveToVoteReq) -> eveToVoteReq?vote_req, temp
       ;};
      if
      :: temp == pipId -> rToEve!vote, pipId;
      :: else;
      fi;
    :: else;
    fi;
  od;

failure: atomic{ resultR = 2; goto term };
```

```

success: resultR = 1;

term: skip;
}

/* ############# SENDER TO RECEIVER ############# */
active proctype eveSToR(){
byte temp = 1;
mtype msg = 0;

end_us:  do
  :: atomic{sToEve?msg, temp -> sToEve!msg, temp};
  :: if
    :: atomic{ sToEve?pip_req, temp ->
      if
      :: eveToPipReq?[pip_req, eval(temp)];
      :: else -> eveToPipReq!pip_req, temp;
      fi;}
    :: atomic{ sToEve?pip_respAck, temp ->
      if
      :: eveToPipRespAck?[pip_respAck, eval(temp)];
      :: else -> eveToPipRespAck!pip_respAck, temp;
      fi;}
    :: atomic{ sToEve?vote_req, temp ->
      if
      :: eveToVoteReq?[vote_req, eval(temp)];
      :: else -> eveToVoteReq!vote_req, temp;
      fi;}
    :: atomic{ sToEve?glob_c, temp ->
      if
      :: eveToGlobC?[glob_c, eval(temp)];
      :: else -> eveToGlobC!glob_c, temp;
      fi;}
    :: atomic{ sToEve?glob_abort, temp -> eveToGlobAbort!
       glob_abort, temp;}
    :: atomic{ sToEve?pip_procErr, temp -> eveToRProcErr!
       pip_procErr, temp;}
    fi;
  od;
}


/* ############# RECEIVER TO SENDER ############# */
active proctype eveRToS(){
byte temp = 1;
mtype msg = 0;

end_us:  do
  :: atomic{rToEve?msg, temp -> rToEve!msg, temp;};
  :: if
    :: atomic{rToEve?pip_reqAck, temp; ->
      if
```

```
        :: eveToPipReqAck?[pip_reqAck, eval(temp)];
        :: else -> eveToPipReqAck!pip_reqAck,temp;
        fi;}
      :: atomic{rToEve?pip_resp, temp; ->
        if
        :: eveToPipResp?[pip_resp, eval(temp)];
        :: else -> eveToPipResp!pip_resp,temp;
        fi;}
      :: atomic{rToEve?vote, temp; ->
        if
        :: eveToVote?[vote, eval(temp)];
        :: else -> eveToVote!vote,temp;
        fi;}
      :: atomic{rToEve?glob_ack, temp; -> eveToGlobAck!glob_ack,temp
         ;}
      :: atomic{rToEve?pip_procErr, temp; -> eveToSProcErr!
         pip_procErr,temp;}
      fi;
   od;
}


active proctype consistent(){
end:
   atomic{ !(resultS == 0 || resultR == 0 || resultS == resultR) ->
        assert (resultS == 0 || resultR == 0 || resultS == resultR )
      };
}
```

Listing 16: Two-Action PIP execution protocol, validation level 3

```
inline chooseVal(cVar){
  if
  :: cVar = true;
  :: skip; /*messages can get lost*/
  fi;
}

#define doTimeout       else
#define E_BUF   2
#define incr(x) x++;

mtype = {pip_req, pip_reqAck, pip_resp, pip_respAck, pip_procErr,
    vote_req, vote, glob_c, glob_abort, glob_ack};

/* to eve */
chan sToEve = [3] of {mtype, byte};
chan rToEve = [3] of {mtype, byte};

/* eve to Rec */
chan eveToPipReq = [E_BUF] of {mtype, byte};
chan eveToPipRespAck = [E_BUF] of {mtype, byte};
chan eveToVoteReq = [E_BUF] of {mtype, byte};
chan eveToGlobC = [E_BUF] of {mtype, byte};
chan eveToGlobAbort = [E_BUF] of {mtype, byte};
chan eveToRProcErr = [E_BUF] of {mtype, byte};

/*eve to Send*/
chan eveToPipReqAck = [E_BUF] of {mtype, byte};
chan eveToPipResp = [E_BUF] of {mtype, byte};
chan eveToVote = [E_BUF] of {mtype, byte};
chan eveToGlobAck = [E_BUF] of {mtype, byte};
chan eveToSProcErr = [E_BUF] of {mtype, byte};

byte resultS = 0, resultR = 0; /*0=Undecided, 1=Success, 2=Failed
    */

/*#################### SENDER ####################*/
active proctype send(){
  byte pipId = 1;
  byte retries = 0;
  byte temp;

pip_init: sToEve!pip_req, pipId; incr(retries);


send_request:
  do
  :: atomic{nempty(eveToSProcErr) -> eveToSProcErr?pip_procErr,
      temp;}
     if
     :: temp == pipId ->  goto failure;
    :: else;/*other process must be in fail anyway*/
    fi;
  :: else ->
    if
    :: atomic {nempty(eveToPipReqAck) -> eveToPipReqAck?pip_reqAck
        , temp;};
      if
      :: temp == pipId -> goto wait_response;
      :: else;
      fi;
    :: atomic {nempty(eveToPipResp) -> eveToPipResp?pip_resp, temp
        ;};
      if
      :: temp == pipId -> sToEve!pip_respAck, pipId; goto intermed
          ;
      :: else;
      fi;
    :: doTimeout ->
      if
      :: d_step{retries < 2; incr(retries);} sToEve!pip_req, pipId
          ;
      :: retries == 2; goto failure;
      fi;
    fi;
  od;


wait_response:
  do /* loop is needed to remove messages from former protocol
      runs */
  :: atomic{nempty(eveToSProcErr) -> eveToSProcErr?pip_procErr,
      temp;
    if
    :: temp == pipId -> goto failure;
    :: else;
    fi;}
  :: else ->
    if
    :: atomic{ nempty(eveToPipResp) -> eveToPipResp?pip_resp, temp
        ;};
      if
      :: temp == pipId -> sToEve!pip_respAck, pipId; goto intermed
          ;
      :: else;
      fi;
    :: doTimeout -> goto failure;
    fi;
  od;


intermed:
  do
  :: atomic{ nempty(eveToPipResp) -> eveToPipResp?pip_resp, temp
```

```
       ;};
     if
     :: temp == pipId -> sToEve!pip_respAck, pipId;
     :: else;
     fi;
  :: sToEve!pip_procErr, pipId; goto failure; /*processingError*/
  :: goto vote_request;
  od;


vote_request:
  sToEve!vote_req, pipId;
  retries = 1;

  do
/*      :: atomic{ nempty(eveToPipResp) -> eveToPipResp?pip_resp,
    temp;};
        if
        :: temp == pipId -> sToEve!vote_req, pipId; goto
            vote_request;
        :: else;
        fi; */ /* #CHANGE# vote_reqs are being sent multiple times
            anyway */
  :: atomic{ nempty(eveToVote) -> eveToVote?vote, temp;};
     if
     :: d_step{ temp == pipId -> retries = 0;} sToEve!glob_c, pipId
        ; goto wait_globAck;
     :: else;
     fi;
  :: doTimeout ->
     if
     :: d_step{retries < 2; incr(retries);} sToEve!vote_req, pipId;
     :: d_step{retries == 2; retries = 0;} sToEve!glob_abort, pipId
        ; goto failure;
     fi;
  od;


wait_globAck:
  retries = 1;

  do
  :: atomic{ nempty(eveToGlobAck) ->  eveToGlobAck?glob_ack, temp
     };
     if
     :: atomic{temp == pipId -> goto success;}
     :: else;
     fi;
/*  :: atomic {nempty(eveToVote) -> eveToVote?vote, temp};
     if
     :: temp == pipId -> sToEve!glob_c, pipId;
     :: else;
     fi;*/ /*#CHANGE# glob_cs are being sent multiple times anyway
```

```
        */
  :: doTimeout ->
    if
    :: retries < 2 -> sToEve!glob_c, pipId; incr(retries);
    :: atomic{retries == 2; goto success;}
    fi;
  od;


failure : resultS = 2; goto term ;

success: resultS = 1; goto term;

term: skip;
}


/*################ RECEIVER ################*/
active proctype rec(){
  byte pipId = 0;
  byte retries = 0;
  byte temp;

pip_init:
  if
  :: atomic{nempty(eveToPipReq) -> eveToPipReq?pip_req, temp;};
    pipId = temp; rToEve!pip_reqAck, temp; goto send_ack;
  :: doTimeout -> goto failure;
  fi;

send_ack:
  do
  :: eveToPipReq?pip_req, temp ->
    if
    :: temp == pipId -> rToEve!pip_reqAck, pipId;
    :: else;
    fi;
  :: rToEve!pip_procErr, pipId; goto failure;
  :: goto send_response;
  od;


send_response:
  rToEve!pip_resp, temp; retries = 1;

  do
  :: atomic{nempty(eveToRProcErr) -> eveToRProcErr?pip_procErr,
     temp;};
    if
    :: temp == pipId -> atomic{retries = 0; goto failure;}
    :: else;
    fi;
  :: atomic{nempty(eveToGlobAbort) -> eveToGlobAbort?glob_abort,
      temp;};
      if
      :: temp == pipId -> atomic{retries = 0; goto failure;}
      :: else;
      fi;
   :: else ->
      if
/*      :: atomic{nempty(eveToPipReq) -> eveToPipReq?pip_req, temp
    ;};
         if
         :: temp == pipId -> rToEve!pip_resp, pipId; goto
              sendResponse;
         :: else;
         fi; */ /* #CHANGE# pip_resps are being sent multiple times
              anyway*/
      :: atomic{nempty(eveToPipRespAck) -> eveToPipRespAck?
           pip_respAck, temp;};
         if
         :: temp == pipId -> atomic{retries = 0; goto twopc_init;}
         :: else;
         fi;
      :: atomic{nempty(eveToVoteReq) -> eveToVoteReq?vote_req, temp
           ;};
         if
         :: temp == pipId -> rToEve!vote, pipId; atomic{retries = 0;
              goto wait_global;}
         :: else;
         fi;
      :: doTimeout ->
         if
         :: retries < 2; rToEve!pip_resp, pipId; incr(retries);
         :: atomic{retries == 2; retries = 0;} goto failure;
         fi;
      fi;
   od;

twopc_init:
   do
   :: atomic{nempty(eveToRProcErr) -> eveToRProcErr?pip_procErr,
        temp;};
      if
      :: temp == pipId -> goto failure;
      :: else;
      fi;
   :: atomic{nempty(eveToGlobAbort) -> eveToGlobAbort?glob_abort,
        temp;};
      if
      :: temp == pipId -> goto failure;
      :: else;
      fi;
   :: else ->
      if
      :: atomic{ nempty(eveToVoteReq) -> eveToVoteReq?vote_req, temp
```

```
        ;};
      if
      :: temp == pipId -> rToEve!vote, pipId; goto wait_global;
      :: else;
      fi;
    :: doTimeout -> goto failure;
    fi;
  od;

wait_global:
  do
  :: atomic{nempty(eveToGlobAbort) -> eveToGlobAbort?glob_abort,
     temp;};
    if
    :: temp == pipId -> goto failure;
    :: else;
    fi;
  :: else ->
    if
    :: atomic{nempty(eveToGlobC) -> eveToGlobC?glob_c, temp;};
      if
      :: temp == pipId -> rToEve!glob_ack, pipId; goto success;
      :: else;
      fi;
    :: atomic{ nempty(eveToVoteReq) -> eveToVoteReq?vote_req, temp
        ;};
      if
      :: temp == pipId -> rToEve!vote, pipId;
      :: else;
      fi;
    :: doTimeout -> goto hang; /*hang notification omitted*/
    fi;
  od;

hang:
  do
  :: atomic{nempty(eveToGlobC) -> eveToGlobC?glob_c, temp;};
    if
    :: temp == pipId -> rToEve!glob_ack, pipId; goto success;
    :: else;
    fi;
  :: atomic{nempty(eveToGlobAbort) -> eveToGlobAbort?glob_abort,
     temp;};
    if
    :: temp == pipId -> goto failure;
    :: else;
    fi;
  :: else ->
    if
    :: atomic{nempty(eveToVoteReq) -> eveToVoteReq?vote_req, temp
        ;};
      if
      :: temp == pipId -> rToEve!vote, pipId;
```

```
        :: else;
        fi;
      :: else;
      fi;
  od;

failure: atomic{ resultR = 2; goto term };

success: atomic{ resultR = 1; goto term};

term: skip;
}

/* ############## SENDER TO RECEIVER ############## */
active proctype eveSToR(){
xr sToEve;

bool gotPipReq = false;
bool  gotRespAck = false;
bool  gotVoteReq = false;
bool  gotGlobC = false;
bool  gotGlobAbort = false;
bool  gotProcErr = false;

bool sentPipReq = false;
bool  sentRespAck = false;
bool  sentVoteReq = false;
bool  sentGlobC = false;
bool  sentGlobAbort = false;
bool  sentProcErr = false;

byte temp = 1;

end: do
   :: atomic{sToEve?pip_req, temp -> chooseVal(gotPipReq);};
   :: atomic{sToEve?pip_respAck, temp -> chooseVal(gotRespAck);};
   :: atomic{sToEve?vote_req, temp -> chooseVal(gotVoteReq);};
   :: atomic{sToEve?glob_c, temp -> chooseVal(gotGlobC);};
   :: atomic{sToEve?glob_abort, temp -> chooseVal(gotGlobAbort);};
   :: atomic{sToEve?pip_procErr, temp -> chooseVal(gotProcErr);};
   :: else ->
      if
      :: gotPipReq && !sentPipReq -> eveToPipReq!pip_req, temp;
         sentPipReq = true;
      :: gotRespAck && !sentRespAck -> eveToPipRespAck!pip_respAck,
         temp; sentRespAck = true;
      :: gotVoteReq && !sentVoteReq -> eveToVoteReq!vote_req, temp;
         sentVoteReq = true;
      :: gotGlobC && !sentGlobC -> eveToGlobC!glob_c, temp;
         sentGlobC = true;
      :: gotGlobAbort && !sentGlobAbort -> eveToGlobAbort!glob_abort
         , temp; sentGlobAbort = true;
      :: gotProcErr && !sentProcErr -> eveToRProcErr!pip_procErr,
        temp; sentProcErr = true;
    :: skip; /*Messages can have arbitrary transmission times*/
    fi;
  od;
}


/* ############# RECEIVER TO SENDER ############# */
active proctype eveRToS(){
xs eveToPipReqAck;
xs eveToPipResp;
xs eveToVote;
xs eveToGlobAck;
xs eveToSProcErr;

xr rToEve;

bool gotReqAck = false;
bool  gotPipResp = false;
bool  gotVote = false;
bool  gotGlobAck = false;
bool  gotProcErr = false;

bool sentReqAck = false;
bool  sentPipResp = false;
bool  sentVote = false;
bool  sentGlobAck = false;
bool  sentProcErr = false;


byte temp = 1;

end: do
  :: atomic {rToEve?pip_reqAck, temp -> chooseVal(gotReqAck);}
  :: atomic {rToEve?pip_resp, temp -> chooseVal(gotPipResp);};
  :: atomic {rToEve?vote, temp -> chooseVal(gotVote);};
  :: atomic {rToEve?glob_ack, temp -> chooseVal(gotGlobAck);};
  :: atomic {rToEve?pip_procErr, temp -> chooseVal(gotProcErr);};
  :: else ->
    if
    ::  gotReqAck && !sentReqAck -> eveToPipReqAck!pip_reqAck,
        temp; sentReqAck  = true;
    ::  gotPipResp && !sentPipResp -> eveToPipResp!pip_resp, temp;
         sentPipResp = true;
    ::  gotVote && !sentVote -> eveToVote!vote, temp;  sentVote =
        true;
    ::  gotGlobAck && !sentGlobAck -> eveToGlobAck!glob_ack, temp;
         sentGlobAck = true;
    ::  gotProcErr && !sentProcErr -> eveToSProcErr!pip_procErr,
        temp;  sentProcErr = true;
    :: skip; /*Messages can have arbitrary transmission times*/
    fi;
  od;
```

```
}


active proctype consistent(){
end:
  atomic{ !(resultS == 0 || resultR == 0 || resultS == resultR) ->
      assert (resultS == 0 || resultR == 0 || resultS == resultR )
     };
}
```

# B List of previous University of Bamberg reports

**Bamberger Beiträge zur Wirtschaftsinformatik**

Stand März 16, 2006

| | |
|---|---|
| Nr. 1 (1989) | Augsburger W., Bartmann D., Sinz E.J.: Das Bamberger Modell: Der Diplom-Studiengang Wirtschaftsinformatik an der Universität Bamberg (Nachdruck Dez. 1990) |
| Nr. 2 (1990) | Esswein W.: Definition, Implementierung und Einsatz einer kompatiblen Datenbankschnittstelle für PROLOG |
| Nr. 3 (1990) | Augsburger W., Rieder H., Schwab J.: Endbenutzerorientierte Informationsgewinnung aus numerischen Daten am Beispiel von Unternehmenskennzahlen |
| Nr. 4 (1990) | Ferstl O.K., Sinz E.J.: Objektmodellierung betrieblicher Informationsmodelle im Semantischen Objektmodell (SOM) (Nachdruck Nov. 1990) |
| Nr. 5 (1990) | Ferstl O.K., Sinz E.J.: Ein Vorgehensmodell zur Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM) |
| Nr. 6 (1991) | Augsburger W., Rieder H., Schwab J.: Systemtheoretische Repräsentation von Strukturen und Bewertungsfunktionen über zeitabhängigen betrieblichen numerischen Daten |
| Nr. 7 (1991) | Augsburger W., Rieder H., Schwab J.: Wissensbasiertes, inhaltsorientiertes Retrieval statistischer Daten mit EISREVU / Ein Verarbeitungsmodell für eine modulare Bewertung von Kennzahlenwerten für den Endanwender |
| Nr. 8 (1991) | Schwab J.: Ein computergestütztes Modellierungssystem zur Kennzahlenbewertung |
| Nr. 9 (1992) | Gross H.-P.: Eine semantiktreue Transformation vom Entity-Relationship-Modell in das Strukturierte Entity-Relationship-Modell |
| Nr. 10 (1992) | Sinz E.J.: Datenmodellierung im Strukturierten Entity-Relationship-Modell (SERM) |
| Nr. 11 (1992) | Ferstl O.K., Sinz E. J.: Glossar zum Begriffsystem des Semantischen Objektmodells |
| Nr. 12 (1992) | Sinz E. J., Popp K.M.: Zur Ableitung der Grobstruktur des konzeptuellen Schemas aus dem Modell der betrieblichen Diskurswelt |
| Nr. 13 (1992) | Esswein W., Locarek H.: Objektorientierte Programmierung mit dem Objekt-Rollenmodell |
| Nr. 14 (1992) | Esswein W.: Das Rollenmodell der Organsiation: Die Berücksichtigung aufbauorganisatorische Regelungen in Unternehmensmodellen |
| Nr. 15 (1992) | Schwab H. J.: EISREVU-Modellierungssystem. Benutzerhandbuch |
| Nr. 16 (1992) | Schwab K.: Die Implementierung eines relationalen DBMS nach dem Client/Server-Prinzip |
| Nr. 17 (1993) | Schwab K.: Konzeption, Entwicklung und Implementierung eines computergestützten Bürovorgangssystems zur Modellierung von Vorgangsklassen und Abwicklung und Überwachung von Vorgängen. Dissertation |

| | |
|---|---|
| Nr. 18 (1993) | Ferstl O.K., Sinz E.J.: Der Modellierungsansatz des Semantischen Objektmodells |
| Nr. 19 (1994) | Ferstl O.K., Sinz E.J., Amberg M., Hagemann U., Malischewski C.: Tool-Based Business Process Modeling Using the SOM Approach |
| Nr. 20 (1994) | Ferstl O.K., Sinz E.J.: From Business Process Modeling to the Specification of Distributed Business Application Systems - An Object-Oriented Approach -. 1st edition, June 1994 |
| | Ferstl O.K., Sinz E.J. : Multi-Layered Development of Business Process Models and Distributed Business Application Systems - An Object-Oriented Approach -. 2nd edition, November 1994 |
| Nr. 21 (1994) | Ferstl O.K., Sinz E.J.: Der Ansatz des Semantischen Objektmodells zur Modellierung von Geschäftsprozessen |
| Nr. 22 (1994) | Augsburger W., Schwab K.: Using Formalism and Semi-Formal Constructs for Modeling Information Systems |
| Nr. 23 (1994) | Ferstl O.K., Hagemann U.: Simulation hierarischer objekt- und transaktionsorientierter Modelle |
| Nr. 24 (1994) | Sinz E.J.: Das Informationssystem der Universität als Instrument zur zielgerichteten Lenkung von Universitätsprozessen |
| Nr. 25 (1994) | Wittke M., Mekinic, G.: Kooperierende Informationsräume. Ein Ansatz für verteilte Führungsinformationssysteme |
| Nr. 26 (1995) | Ferstl O.K., Sinz E.J.: Re-Engineering von Geschäftsprozessen auf der Grundlage des SOM-Ansatzes |
| Nr. 27 (1995) | Ferstl, O.K., Mannmeusel, Th.: Dezentrale Produktionslenkung. Erscheint in CIM-Management 3/1995 |
| Nr. 28 (1995) | Ludwig, H., Schwab, K.: Integrating cooperation systems: an event-based approach |
| Nr. 30 (1995) | Augsburger W., Ludwig H., Schwab K.: Koordinationsmethoden und -werkzeuge bei der computergestützten kooperativen Arbeit |
| Nr. 31 (1995) | Ferstl O.K., Mannmeusel T.: Gestaltung industrieller Geschäftsprozesse |
| Nr. 32 (1995) | Gunzenhäuser R., Duske A., Ferstl O.K., Ludwig H., Mekinic G., Rieder H., Schwab H.-J., Schwab K., Sinz E.J., Wittke M: Festschrift zum 60. Geburtstag von Walter Augsburger |
| Nr. 33 (1995) | Sinz, E.J.: Kann das Geschäftsprozeßmodell der Unternehmung das unternehmensweite Datenschema ablösen? |
| Nr. 34 (1995) | Sinz E.J.: Ansätze zur fachlichen Modellierung betrieblicher Informationssysteme - Entwicklung, aktueller Stand und Trends - |
| Nr. 35 (1995) | Sinz E.J.: Serviceorientierung der Hochschulverwaltung und ihre Unterstützung durch workflow-orientierte Anwendungssysteme |
| Nr. 36 (1996) | Ferstl O.K., Sinz, E.J., Amberg M.: Stichwörter zum Fachgebiet Wirtschaftsinformatik. Erscheint in: Broy M., Spaniol O. (Hrsg.): Lexikon Informatik und Kommunikationstechnik, 2. Auflage, VDI-Verlag, Düsseldorf 1996 |

| | |
|---|---|
| Nr. 37 (1996) | Ferstl O.K., Sinz E.J.: Flexible Organizations Through Object-oriented and Transaction-oriented Information Systems, July 1996 |
| Nr. 38 (1996) | Ferstl O.K., Schäfer R.: Eine Lernumgebung für die betriebliche Aus- und Weiterbildung on demand, Juli 1996 |
| Nr. 39 (1996) | Hazebrouck J.-P.: Einsatzpotentiale von Fuzzy-Logic im Strategischen Management dargestellt an Fuzzy-System-Konzepten für Portfolio-Ansätze |
| Nr. 40 (1997) | Sinz E.J.: Architektur betrieblicher Informationssysteme. In: Rechenberg P., Pomberger G. (Hrsg.): Handbuch der Informatik, Hanser-Verlag, München 1997 |
| Nr. 41 (1997) | Sinz E.J.: Analyse und Gestaltung universitärer Geschäftsprozesse und Anwendungssysteme. Angenommen für: Informatik ’97. Informatik als Innovationsmotor. 27. Jahrestagung der Gesellschaft für Informatik, Aachen 24.-26.9.1997 |
| Nr. 42 (1997) | Ferstl O.K., Sinz E.J., Hammel C., Schlitt M., Wolf S.: Application Objects – fachliche Bausteine für die Entwicklung komponentenbasierter Anwendungssysteme. Angenommen für: HMD – Theorie und Praxis der Wirtschaftsinformatik. Schwerpunkheft ComponentWare, 1997 |
| Nr. 43 (1997): | Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using the Semantic Object Model (SOM) – A Methodological Framework - . Accepted for: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1997 |
| | Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using (SOM), 2nd Edition. Appears in: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1998 |
| Nr. 44 (1997) | Ferstl O.K., Schmitz K.: Zur Nutzung von Hypertextkonzepten in Lernumgebungen. In: Conradi H., Kreutz R., Spitzer K. (Hrsg.): CBT in der Medizin – Methoden, Techniken, Anwendungen -. Proceedings zum Workshop in Aachen 6. – 7. Juni 1997. 1. Auflage Aachen: Verlag der Augustinus Buchhandlung |
| Nr. 45 (1998) | Ferstl O.K.: Datenkommunikation. In. Schulte Ch. (Hrsg.): Lexikon der Logistik, Oldenbourg-Verlag, München 1998 |
| Nr. 46 (1998) | Sinz E.J.: Prozeßgestaltung und Prozeßunterstützung im Prüfungswesen. Erschienen in: Proceedings Workshop „Informationssysteme für das Hochschulmanagement“. Aachen, September 1997 |
| Nr. 47 (1998) | Sinz, E.J.:, Wismans B.: Das „Elektronische Prüfungsamt“. Erscheint in: Wirtschaftswissenschaftliches Studium WiSt, 1998 |
| Nr. 48 (1998) | Haase, O., Henrich, A.: A Hybrid Respresentation of Vague Collections for Distributed Object Management Systems. Erscheint in: IEEE Transactions on Knowledge and Data Engineering |
| Nr. 49 (1998) | Henrich, A.: Applying Document Retrieval Techniques in Software Engineering Environments. In: Proc. International Conference on Database and Expert Systems |

| | |
|---|---|
| | Applications. (DEXA 98), Vienna, Austria, Aug. 98, pp. 240-249, Springer, Lecture Notes in Computer Sciences, No. 1460 |
| Nr. 50 (1999) | Henrich, A., Jamin, S.: On the Optimization of Queries containing Regular Path Expressions. Erscheint in: Proceedings of the Fourth Workshop on Next Generation Information Technologies and Systems (NGITS'99), Zikhron-Yaakov, Israel, July, 1999 (Springer, Lecture Notes) |
| Nr. 51 (1999) | Haase O., Henrich, A.: A Closed Approach to Vague Collections in Partly Inaccessible Distributed Databases. Erscheint in: Proceedings of the Third East-European Conference on Advances in Databases and Information Systems – ADBIS'99, Maribor, Slovenia, September 1999 (Springer, Lecture Notes in Computer Science) |
| Nr. 52 (1999) | Sinz E.J., Böhnlein M., Ulbrich-vom Ende A.: Konzeption eines Data Warehouse-Systems für Hochschulen. Angenommen für: Workshop „Unternehmen Hochschule“ im Rahmen der 29. Jahrestagung der Gesellschaft für Informatik, Paderborn, 6. Oktober 1999 |
| Nr. 53 (1999) | Sinz E.J.: Konstruktion von Informationssystemen. Der Beitrag wurde in geringfügig modifizierter Fassung angenommen für: Rechenberg P., Pomberger G. (Hrsg.): Informatik-Handbuch. 2., aktualisierte und erweiterte Auflage, Hanser, München 1999 |
| Nr. 54 (1999) | Herda N., Janson A., Reif M., Schindler T., Augsburger W.: Entwicklung des Intranets SPICE: Erfahrungsbericht einer Praxiskooperation. |
| Nr. 55 (2000) | Böhnlein M., Ulbrich-vom Ende A.: Grundlagen des Data Warehousing. Modellierung und Architektur |
| Nr. 56 (2000) | Freitag B, Sinz E.J., Wismans B.: Die informationstechnische Infrastruktur der Virtuellen Hochschule Bayern (vhb). Angenommen für Workshop "Unternehmen Hochschule 2000" im Rahmen der Jahrestagung der Gesellschaft f. Informatik, Berlin 19. - 22. September 2000 |
| Nr. 57 (2000) | Böhnlein M., Ulbrich-vom Ende A.: Developing Data Warehouse Structures from Business Process Models. |
| Nr. 58 (2000) | Knobloch B.: Der Data-Mining-Ansatz zur Analyse betriebswirtschaftlicher Daten. |
| Nr. 59 (2001) | Sinz E.J., Böhnlein M., Plaha M., Ulbrich-vom Ende A.: Architekturkonzept eines verteilten Data-Warehouse-Systems für das Hochschulwesen. Angenommen für: WI-IF 2001, Augsburg, 19.-21. September 2001 |
| Nr. 60 (2001) | Sinz E.J., Wismans B.: Anforderungen an die IV-Infrastruktur von Hochschulen. Angenommen für: Workshop „Unternehmen Hochschule 2001“ im Rahmen der Jahrestagung der Gesellschaft für Informatik, Wien 25. – 28. September 2001 |

Änderung des Titels der Schriftenreihe *Bamberger Beiträge zur Wirtschaftsinformatik* in *Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik* ab Nr. 61