Proceedings of the 5th International Conference on Enterprise Information Systems (ICEIS). Angers, France, April 2003.

PATTERN BASED ANALYSIS OF EAI LANGUAGES – THE CASE OF THE BUSINESS MODELING LANGUAGE

Petia Wohed[*], Erik Perjons
Stockholm University/The Royal Institute of Technology
Forum 100, 164 40 Kista, Sweden
{*petia, perjons*}*@dsv.su.se*

Marlon Dumas, Arthur ter Hofstede
Queensland University of Technology
GPO Box 2434, Brisbane QLD 4001, Australia
{*m.dumas, a.terhofstede*}*@qut.edu.au*

Key words: Enterprise Application Integration, Workflow and Communication patterns

Abstract: Enterprise Application Integration (EAI) is a challenging area that is attracting growing attention from the software industry and the research community. A landscape of languages and techniques for EAI has emerged and is continuously being enriched with new proposals from different software vendors and coalitions. However, little or no effort has been dedicated to systematically evaluate and compare these languages and techniques. The work reported in this paper is a first step in this direction. It presents an in-depth analysis of a language, namely the Business Modeling Language, specifically developed for EAI. The framework used for this analysis is based on a number of workflow and communication patterns. This framework provides a basis for evaluating the advantages and drawbacks of EAI languages with respect to recurrent problems and situations.

1 INTRODUCTION

The paradigmatic shift from a functional to a process (cross-functional) oriented approach of information system development during the last years, raises new demands for integrating the underlying information systems. The technical problems elicited during such integration have become the main topic of Enterprise Application Integration (EAI), the purpose of which is the integration of individual applications into a seamless whole, with minimum coding effort.

One of the techniques for EAI is to extract all the process logic and to encapsulate it in one place (the Process Broker), separated from the application logic which is implemented by the underlying applications. A process broker communicates with the surrounding applications by means of messages and executes according to the logic implemented in it (Linthicum, 00).

During the last years a number of techniques and platforms for developing process brokers have been developed by different software vendors, e.g. (IBM, 02; IONA, 02; HP, 02; Vitria, 02). These platforms implement proprietary modelling languages and notations. With the continuously increasing number of such languages, the need of an in-depth comparison of them becomes more and more obvious. The work presented in this paper is a first step in this direction. An analysis framework based on a number of workflow and communication patterns has been used for analyzing one of these languages, namely the Business Modeling Language (BML) (Wåhlander et al., 01; Johannesson and Perjons, 01).

The contribution of the paper is twofold: (i) the in-depth analysis of BML; and (ii) the assemblage and testing of the framework through this analysis. BML was selected as the starting language for the analysis, because it is a message oriented process modeling language based on communicating state machines. This feature clearly distinguishes BML from the activity-based process modeling languages implemented by traditional Workflow Management Systems (WFMS), for which an extensive comparison has already been provided in (Aalst et al., 02b). Although the analysis provided here is specifically targeted to BML, many of the results are applicable to other languages of the same family, such as SDL[1].

The framework used for the analysis is composed of two parts:

- A **set of workflow patterns** identified by van der

[*]Research conducted while at the Queensland University of Technology.

[1]SDL - Structured and Description Language (ITU, 01) was developed by the International Telecommunication Union for the purposes of specifying telecommunication services.

Aalst et al. (Aalst et al., 02b). A Workflow (WF) pattern is an abstracted form of a recurring situation related to the ordering of the activities in a workflow. The WF patterns defined in (Aalst et al., 02b) are language and domain independent, making them a suitable analysis instrument. Besides, they have been successfully used for comparing a number of commercial WFMS (Aalst et al., 02b), and for analyzing the expressive power of UML activity diagrams (Dumas and ter Hofstede, 01).

- A **set of communication patterns** described in (Ruh et al., 01). Since messages are the way in which a process broker communicates with the applications that it integrates, this is a necessary part for achieving a comprehensive analysis.

This framework is complementary to the one presented in (Söderström et al., 02). This latter framework addresses a similar problem: the continuously increasing number of new process modeling languages and the need to understand and compare them. However it is intended to be used by IS/IT-managers, business strategists and other stakeholders involved in business process management, which puts it at a higher level of abstraction than the framework proposed in this paper.

The rest of the paper is structured as follows. Section 2 gives an overview of the EAI domain and introduces briefly the BML language. In sections 3 and 4 the BML language is evaluated against the set of workflow and communication patterns. Finally, section 5 concludes the work and gives directions for future research.

2 BACKGROUND

2.1 EAI

We adopt Linthicum's definition of enterprise application integration as "the unrestricted sharing of data and business processes among *any* connected applications and data sources in the enterprise" (Linthicum, 00). The point of departure when applying EAI is facilitating sharing of data and processes without applying extensive changes to the existing application and data structures.

The approaches used for EAI have evolved from deploying Point-to-Point solutions, to Message Broker architectures, and then to Process Broker architectures (see fig. 1, reprinted from (Johannesson and Perjons, 01)). This line of development clearly reveals the two main forces driving it, namely: 1) complexity reduction, by moving from a Point-to-Point to a Message Broker architecture, and 2) process logic extraction and encapsulation, which is the essence of extending the message brokers into process brokers. Separating in this way the process logic from the application logic improves flexibility and facilitates maintenance.

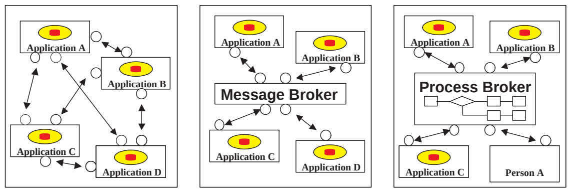

Figure 1: Architectures of EAI

A technology related to the Process Brokers is that of Workflow Management Systems (WFMS). The WFMS were initially designed for managing communication based on document routing. Whereas the first generation of WFMS aimed to support simple task coordination, the next generation workflow technologies put the business process in focus. Following the development further, the workflow community has extended the problem domain to even span over enterprise-wide and inter-organizational workflows in the last years. A result from this is the development of the Wf-XML language (WfMC, 00), by the Workflow Management Coalition. The Wf-XML language defines a number of XML message templates, the instances of which can be used for initiating, monitoring and controlling business processes in/by remote systems.

2.2 Overview of BML

The Business Modeling Language (BML) (Wåhlander et al., 01; Johannesson and Perjons, 01) is a message oriented process modeling language based on communicating state machines. A process diagram (see the right-hand side in fig. 2) models the states of a process and what is performed/processed during the transitions between these states. During the transitions messages can be sent, automated tasks can be performed, and/or automated decisions can be made. For each process diagram there is a number of process instances, that are created at runtime. The process instances execute independently of each other, but can communicate by sending and receiving messages. Each instance has an input queue (left-hand side in fig. 2), where received messages are stored. A process instance can either be waiting in a state or performing a transition from one state to another. A transition is initiated when a message in the input queue is consumed or a timer has expired.

Following the example in fig. 2, a process instance starts in a *Start* state (a circle with the name Start). Only the messages *m1* from *process a* and *m2* from *process c* can initiate a transition. The message *m1* is first in the queue and is therefore consumed, and the process instance performs a transition to the state *Wait for Event 1*. During the transition a message *m3* is sent to *human actor D* and a *timer T1* is started. Thereafter the message *m9* is first in the queue. Since only message *m5* can initiate a further transition from *Wait for Event 1*, message *m9* is discarded (sent to the back of the queue). The next message in the queue is then *m5*, which initiates *Automated Business Decision 1* and the process instance continues following the path satisfying the specified business rule.

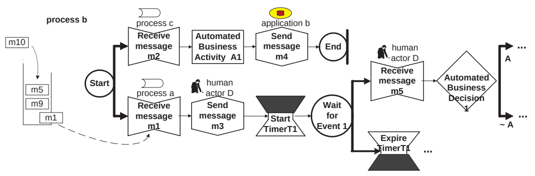

Figure 2: A process instance with the input queue

The main BML symbols are as follows. **Wait for Event** and **Start**: a process instance is waiting in a Wait for Event state until a message is received or a timer has expired. The starting state is indicated by a Wait for Event symbol named Start. The end of the flow of a process is described by the **End** symbol. **Receive Message** describes the consumption of a message from the input queue and **Send Message** describes the sending of a message. **Automated Business Activity** defines operations that will be performed on the process instance. **Automated Business Decision** defines how the control flow is dynamically changed depending on different business rules. **Start Timer** and **Expire Timer** capture the notion of time, which facilitates delays supervision. When a timer is started it is provided with a timeout value. **Application** and **Human actor** are both symbols denoting external actors and like the **Process** symbol they are used to model the sender/recipient of a message. Each process diagram has furthermore a data model that describes the data handled in the model as well as the structure of the different messages exchanged during the process execution.

3 WORKFLOW PATTERNS IN BML

In this section, we consider a subset of the 20 workflow patterns presented in (Aalst et al., 02b), and we discuss how and to what extent these patterns can be expressed in BML. The patterns considered here have been selected because they put forward important strengths, weaknesses, or specificities of BML with respect to alternative activity-based business process modelling languages. In particular, we do not consider patterns that directly correspond to primitive BML constructs such as the "sequence" pattern, the "exclusive choice" pattern (whereby one among several branches is chosen based on a condition), and the "simple merge" pattern (the dual of the "exclusive choice", i.e. several branches of which only one is active at a moment, reconverge into one single branch).

3.1 Parallel Split

Description A point in the process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order[2].

Example After activity *new_cellphone_subscription_order* the activity *insert_new_subscription* in Registry application (Home Location Registry application) and *insert_new_subscription* in Mobile answer application are executed in parallel.

Solution In many contemporary workflow languages, this pattern is captured by a primitive operator which creates two parallel branches (or threads) within the same process. In contrast, BML does not support multiple threads within a single process instance. Instead, a parallel split is realized by simultaneously creating instances of two or several (sub-)processes, which then run in parallel (see fig. 3). This feature is inherent to languages based on communicating state machines (SDL for example) since a state machine can only be in one state at a time[3].

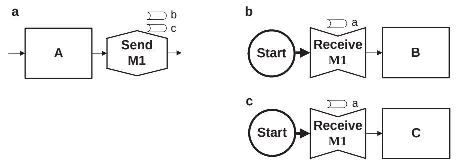

Figure 3: Parallel Split

3.2 Synchronization

Description A point in the process where multiple parallel branches (i.e. sub-processes or activities) converge into one single thread of control, thus synchronizing multiple threads (see also (WfMC, 99)). It is an assumption of this pattern that after an incoming branch has been completed, it cannot be completed again while the merge is still waiting for other

[2]This definition is close to the definition introduced by WFMC in (WfMC, 99)

[3]Harel's statecharts (Harel, 87) and similar formalisms are an exception. They are based on state machines but support parallel branches.

branches to be completed. Also, it is assumed that the threads to be synchronized belong to the same global process instance (i.e., to the same "case" in workflow terminology).

Example Activity *archive* is executed after the completion of both activity *send_tickets* and activity *receive_payment*. Obviously, the synchronization occurs within a single global process instance: the *send_tickets* and *receive_payment* must relate to the same client request (i.e. no synchronization occurs between the payment of client X and the sending of the tickets to client Y).

Solution As discussed in the previous pattern, the "threads" to be synchronized are modelled in BML as distinct processes running in parallel. These parallel processes are denoted by *b* and *c* in fig. 4. To be able to synchronize, *b* and *c* send a message to an already running instance of a process called *a*, which acts as a *synchronizer*. The synchronizer waits for messages from both *b* and *c* and then triggers the activity or sub-process to be performed after the synchronization point, which in our example is activity *D*.

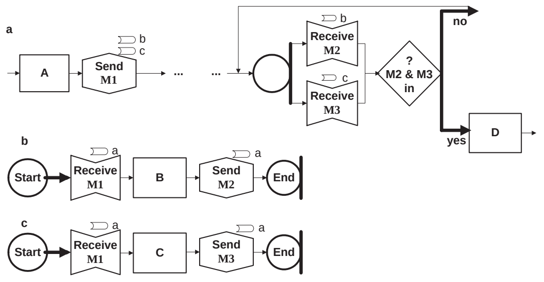

Figure 4: Synchronization

This solution presupposes that the instances of *b* and *c* know to which instance of *a* they have to send their synchronization messages. Hence, instances of *b* and *c* must be provided at creation time with the identifier of the instance of *a* which will act as their synchronizer. This in turn means that for every synchronization point there is a corresponding parallel split, such that when the split is reached, it is known for certain that the synchronization point will be reached as well, and therefore, it is clear that an instance of *a* needs to be created and its identifier needs to be given to both *b* and *c*. This is not the case in the presence of arbitrary (non-structured) cycles, i.e. cycles going from a point within a block delimited by a parallel split and a synchronization point, to a point preceding this region. In these cases, when the parallel split is reached, it is not known for certain whether the corresponding synchronization point will be reached or not. Indeed, it may happen that the arbitrary cycle in the middle of the block is taken, so that the process quits the block before reaching the synchronization point. (Kiepuszewski et al., 00) analyzes the properties of processes containing such arbitrary cycles, and identifies situations in which such cycles cannot be removed (or "unfolded") without changing the behaviour of the process. However, we have not yet found practical situations involving such "unfoldable" arbitrary cycles in the context of EAI.

3.3 Multi-Merge

Description A point in a process where two or more branches reconverge without synchronization. If more than one branch gets activated, possibly concurrently, the activity following the merge is started for every action of every incoming branch.

Example Two activities *audit_application* and *process_applications* running in parallel, should both be followed by an activity *close_case*, which must be executed twice if the activities *audit_application* and *process_applications* are both executed.

Solution Two different solutions apply for this pattern. The first one is as proposed by van der Aalst et al. in (Aalst et al., 02b). It is based on replication of the activities triggered after the merge, so that they are included in each parallel process. The second solution (see fig. 5) avoids this replication by starting a new process instance of a separate process, where the activities following the merge are placed. In this way a new process instance is created each time one of the triggering tasks is completed. As with the "Synchronization" pattern, this solution assumes that for every synchronizing merge, there is a single corresponding multi-choice point (or parallel split), that precedes it and that starts the sub-processes to be synchronized.

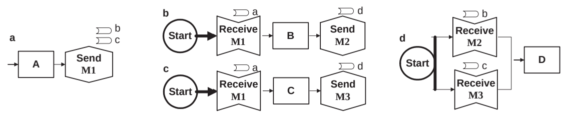

Figure 5: Multi-Merge

3.4 N out of M Join

Description A point in the process where M parallel threads converge into one. The subsequent activity or sub-process should be activated once N out of these M threads have completed ($N \leq M$). The completion of all remaining threads is ignored. This pattern has been identified in (Casati et al., 95), where it is called *partial join*. The "Synchronization" pattern presented above is a special case of the N out of M pattern, where N = M.

Example To improve query response time, a complex search is sent to two alternative databases over the Internet. The first one that comes up with the result should proceed the flow. The second result is ignored.

Solution As in the "Synchronization" pattern, the M incoming branches are modelled in BML as sub-processes running in parallel (sub-processes *b1,...,bM* in fig. 6). These sub-processes are triggered from a parent process *a*. Upon completion, each of these sub-processes sends a message back to *a*, which waits until it receives N messages from the sub-processes *b1,...,bM*, before proceeding with the next activity (task *C* in fig. 6). Assuming that the process *a* never returns to the state *Wait M2*, after the execution of task *C*, the messages from *b1,...,bM* received from this point on, will not be consumed.

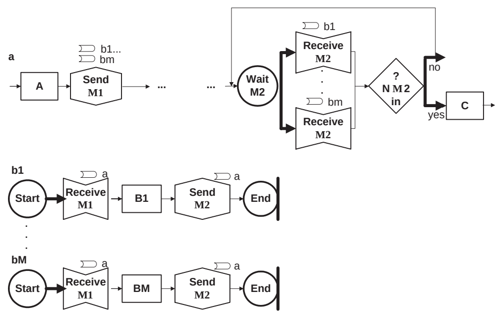

Figure 6: N out of M join

This solution assumes that the process *a* is not involved in a loop. If this was the case, process *a* would come back to the state *Wait M2* in the second iteration of the loop, and consequently, messages from *b1,...,bM* that should be ignored, would now be consumed. In other words, completion messages related to the first iteration of the loop would be mixed with completion messages related to the second iteration. To avoid this happening, a solution is to transform the loop into a recursive call to process a^4. In other words, when an instance of process *a* reaches a point where a loop should be taken, a new instance of process *a* is created to handle the next iteration of the loop. Eventually, this new instance will create instances of *b1,...,bM* and will synchronize them. However, each instance of *b1,...,bM* will be associated to a unique instance of *a*, thereby eliminating the possibility of mixing instances created in different iterations of the loop. However, this transformation of loops into recursive calls is only possible if the loops are structured (i.e. there are no unfoldable arbitrary cycles), as discussed in the "Synchronization" pattern.

[4] In MQSeries workflow, loops are also modelled through such recursive or iterative invocations to a process.

3.5 Multiple Instances without a Priori Runtime Knowledge

Van der Aalst et al. present a family of patterns that involve the creation of multiple instances of an activity or subprocess (Aalst et al., 02b). For space reasons, we only consider the most complex of these patterns. The other patterns of this family can be handled by restricting the solution to this pattern.

Description A point in a workflow process where an activity B is enabled multiple times. The number of instances of B that need to be enabled is not known until all these instances have been created. After all the created instances of B have completed, a terminating activity E has to be executed once.

Example When booking a trip, the activity *book_flight* is executed multiple times if the trip involves multiple flights. Once all bookings are made, an invoice is sent to the client. How many bookings are made is only know at runtime through interaction with the user.

Solution In BML, this pattern is implemented by three processes that communicate with each other: one "wrapping" the activity *B*, called process *b*; a second one process *a*, for initiating the necessary number of instances of the process *b*; and a third one process *c*, that waits for the completion of all initiated instances of process *b* before executing activity *E*. Process *c* receives from process *a* the number *n* of instances of process *b* that it needs to wait for, before initiating task *E*. This solution is presented in figure 7. Initializing of the counters *n* and *i* to zero is done at start time.

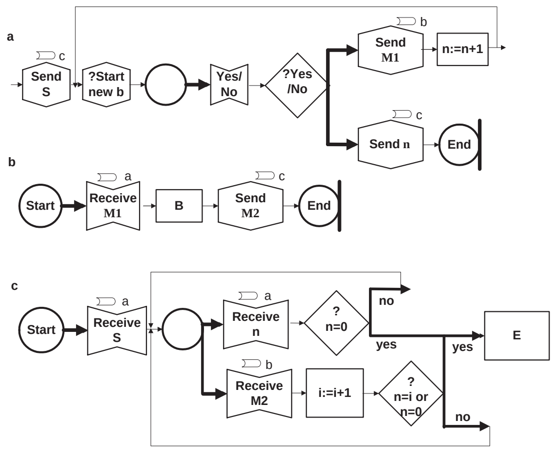

Figure 7: MI without a priori runtime knowledge

3.6 Deferred Choice

Description A point in a process where one among several alternative branches is chosen based on information which is not necessarily available when this point is reached. This differs from the normal exclusive choice, in that the choice is not made immediately when the point is reached, but instead several alternatives are offered to the process broker, and the choice is delayed until a signal is received.

Example When a contract is finalised, it has to be reviewed and signed either by the director or by the operations manager, whoever is/are available first. The process broker will notify both the director and the operations manager that the contract is to be reviewed: the first one available will review it.

Solution There is a construct in BML for handling this situation, namely the *Wait for Event* state. This contrasts with other workflow modelling languages where the choices between branches are always made immediately when the point of choice is reached, and there is no straightforward way of expressing the fact that the choice needs to be delayed.

3.7 Interleaved Parallel Routing

Description A set of activities is executed in an arbitrary order. Each activity in the set is executed exactly once. The order is decided at run-time: it is not until one activity is completed that the decision on what to do next is taken. In any case, no two activities in the set can be active at the same time.

Example At the end of each year, a bank executes two activities for each account: *add_interest* and *charge_credit_card_costs*. These activities can be executed in any order. However, since they both update the account, they cannot be executed at the same time.

Solution The idea is to wrap the activities to be interleaved in separate BML sub-processes. A master process is responsible for deciding which activity is to be executed in the first place. After taking this decision (based on user input for example), the master process sends an invocation message to the first activity. When this activity is completed, the sub-process sends a message back to the master process. The master process then decides which activity will be executed in the second place, and sends an invocation message to the sub-process wrapping this second activity, and so on. A solution based on a similar principle is described in the context of UML activity diagrams in (Dumas and ter Hofstede, 01).

3.8 Milestone

Description A given activity can only be enabled if a certain milestone has been reached which has not yet expired. A milestone is defined as a point in the process where a given activity has finished and an activity following it has not yet started.

Example After having placed a purchase order, a customer can withdraw it at any time before the shipping takes place. To withdraw an order, the customer must complete a withdrawal request form, and this request must be approved by a customer service representative. The execution of the activity *approve_order_withdrawal* must therefore follow the activity *request_withdrawal*, and can only be done if: (i) the activity *place_order* is completed, and (ii) the activity *ship_order* has not yet started.

Solution The milestone is modelled by a *Wait for Event* state with two outgoing branches. One of the branches is triggered by the expiry of the milestone (the beginning of the activity *ship_order* in the above example), while the other branch is triggered by a request for executing the activity linked to the milestone (activity *approve_order_withdrawal* in the example). This is illustrated in figure 8, where activity *C* is only enabled if activity *A* has completed and activity *B* has not yet started.

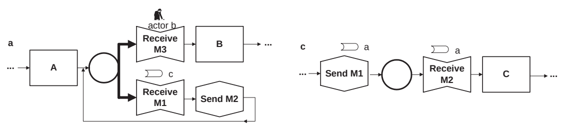

Figure 8: Milestone

4 COMMUNICATION PATTERNS IN BML

In this section we consider the communication patterns presented in (Ruh et al., 01), and describe their implementation in BML.

Communication is realized by exchanges of messages between points located in different processes. In message oriented process languages, the points of communication are explicitly modelled by two symbols: one for sending and another one for receiving a message. In BML a message may also be sent from a process (or application) to itself, which implies that a pair of send-receive messages does not necessarily need to be split across two different processes. Two types of communications are distinguished, namely synchronous and asynchronous communication.

Synchronous Communication

Synchronous communication denotes the situation in which the sender and the receiver coordinate their processing according to their communication.

4.1 Request/Reply

Description One application/process A sends a request to an application/process B and blocks (i.e., waits for reply) until B sends a reply. After receiving the reply A continues processing. The task performed then usually depends on the response it gets from B.

Example After the customer has booked a flight, the booking system asks (requests) the customer which payment method he/she prefers. Depending of the customer's response, the booking system performs different activities. No customer activities can be performed before the response is received.

Solution The solution for this pattern is shown in fig. 9. Naturally, two processes/applications *a* and *b* between which the communication is realized are modeled (see, alt1). After sending a message to process *b*, process *a* gets into a *Wait for Event* state, waiting for a response from *process b*. After receiving the response from *process b* a choice is made based on it, and the execution of *process a* continues according to this choice.

This solution is applicable when the owner of process/application *a* also has sufficient control of the process/application *b* and can guarantee that process/application *b* always sends a reply back to *a*. However, if it is not the case, e.g. if *b* is an external process/application, it is better to include an extra check in the process/application *a*, limiting the waiting for response time. This is typically done by including a timer in *a* (see alt2) and specifying the execution path in case the time expires before any response from *b* has been registered.

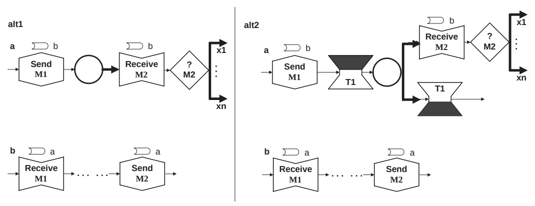

Figure 9: Request/Reply

4.2 One-Way

Description One-way communication is a special case of Request/Reply when the sender A only requires a confirmation for receipt from the receiver B, before continuing.

Example After the customer has ordered products, the supplier confirms to the customer that the order is received. The customer does not perform any activities until a confirmation is received.

Solution Since this is a special case of the Request/Reply pattern the solution is very close to the one presented above. The only differences are that process/application *b* sends a notification for receipt immediately after consuming *a*'s message. When receiving it *a* simply continues to execute without taking into consideration the content of the notification.

4.3 Synchronous Polling

Description Synchronous Polling partially allows the sender A to continue processing while waiting for a reply from the receiver B. A needs, though, to periodically stop and check for the expected reply.

Example During a game session, the system continuously checks if the customer has terminated the game.

Solution The solution for this pattern (see fig. 10) is implemented through *checkpoints*, i.e. states in which the process can either: (i) consume the message received from *b* before continuing with its other tasks; or (ii) if the message from *b* has not yet arrived, simply continue with its tasks until the next checkpoint is reached. Checkpoints are modelled by a *Wait for Event* state with two outgoing branches: one that will be fired if the message from *b* is already available, and the other pointing to a timer whose value is set to zero, indicating the absence of waiting time but the presence of an interruption.

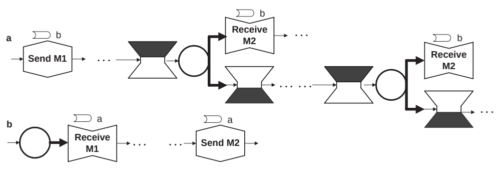

Figure 10: Synchronous Polling

Asynchronous Communication

In contrast to synchronous communication, asynchronous communication does not require the sender to synchronize processing with communication. The sender sends a message and continues processing immediately.

4.4 Message Passing

Description An application A sends a message to an application B and continues processing.

Example When an order is received, a log is notified, before the system execute the order.

Solution Message passing can simply be represented by the *Send message* symbol alone.

4.5 Publish/Subscribe

Description An application A sends messages to a number of other applications which have previously lodged an expression of interest in receiving this type of message(s).

Example An organisation offers information about the products to its customers. If the customers are interested in receiving such information, they have to notify a system, which lists interested customer. When product information is going to be distributed to the customers, the organisation requests the current list, including the customer's addresses.

Solution There is no construct in BML that directly captures the concept of subscription list. It is however possible to model the control-flow of a subscription handling process (see process *b* in fig. 10). The process *b* is initiated when a request for a subscription list is made. From there on, actors can subscribe or unsubscribe by sending *Start* or *Delete subscription* messages to the subscription process *b*. The subscription list can then be used for publishing, i.e. sending information to the subscribers on the list. The publication is done by process *a* in the figure, requesting and receiving the subscription list from *b*. This solution does not capture the details of the data manipulation required to maintain the subscription list.

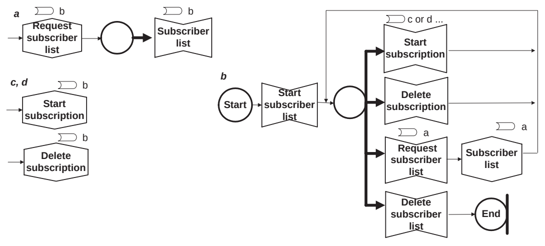

Figure 11: Publish/Subscribe

4.6 Broadcast

Description A message is sent to every application in a system. The receiver decides whether it is interested in the message or not. If it has interest the request is processed according to the logic programmed into the receiver, otherwise the message is ignored.

Example Before a system is shut down, every client connected to it is informed about the situation.

Solution In BML this is represented by a *Send message* symbol, where all the applications are explicitly specified as recipients of this message. The way in which the different recipients react to the message is implemented in the corresponding application logic.

5 CONCLUSION

In this paper a framework based on existing workflow and communication patterns was used for an in-depth analysis of BML. BML, as a language based on communicating state machines, differs from the traditional activity-based workflow languages. Heavily focused on modeling communication, it provides a comparatively simple representation for most of the communication patterns. At the same time, this emphasis on modelling the communication, does not seem to imply major drawbacks on its ability to model typical workflow (i.e. task-driven) scenarios. Most of the workflow patterns can be represented in BML in a relatively simple way.

In particular, state-based patterns such as the deferred choice and the milestone patterns can be captured naturally in BML, since BML integrates the concept of states in between the processing of two business activities. These state-based patterns are difficult, and sometimes impossible to express in existing workflow modeling languages (Aalst et al., 02b). On the other hand, modeling parallel threads in BML involves decomposing the process into several sub-processes: one per parallel thread. As a result, for processes with a high degree of parallelism, the representation in BML will contain an equally large number of distinct sub-process, which may affect the comprehensibility of the overall process model.

A limitation of BML identified during our analysis relates to the modeling non-structured workflows. However, given that non-structured workflows cause problems even for many existing WFMS, this is not a drawback specific to BML.

A summary of the analysis on BML is presented in Table 1. The table also shows a comparison of BML with two Workflow Modelling Languages: IBM's MQSeries Workflow and TIBCO's InConcert, both of which are key components of the EAI solutions of their respective vendors (IBM's WebSphere MQ and TIBCO's ActiveEnterprise). The ratings for MQSeries Workflow and InConcert in the table are taken from (Aalst et al., 02b) where an analysis of major commercial WFMS is provided. A '+' in a cell of the table refers to direct support; a '–' refers to indirect support (i.e. there is no cunstruct which directly support the pattern, but a work-around solution has to be applied); '+/–' is an intermediate ranking; and 'ne' means that we have not (conclusively) evaluated the support for the pattern in the language.

The framework used in this paper can be applied to the analysis of other EAI languages. Furthermore, as EAI is closely related to Web service composition, it is possible to use the same framework for evaluating Web service composition languages. In (Wohed et al., 02; Aalst et al., 02a) we have reported the evaluations for BPEL4WS and BPML.

	BML	MQS	InC
Sequence	+	+	+
Parallel Split	+	+	+
Synchronization	+	+	+
Exclusive Choice	+	+	+/–
Simple Merge	+	+	+/–
Multi Choice	+	+	+/–
Synchronizing Merge	+	+	+
Multi-Merge	+	–	–
N out of M Join	+/–	–	–
Arbitrary Cycles	–	–	–
Implicit Termination	+	+	+
MI without Synchronization	+	–	–
MI with a priori Design Time Knowledge	+	+	+
MI with a priori Runtime Knowledge	+/–	–	–
MI without a priori Runtime Knowledge	+/–	–	–
Deferred Choice	+	–	–
Interleaved Parallel Routing	–	–	–
Milestone	+	–	–
Cancel Activity	ne	–	–
Cancel Case	ne	–	–
Request/Reply	+	ne	ne
One-Way	+	ne	ne
Synchronous Polling	+	ne	ne
Message Passing	+	ne	ne
Publish/Subscribe	+/–	ne	ne
Broadcast	+	ne	ne

Table 1: Comparison of BML against MQSeries Workflow (MQS) and InConcert (InC)

Acknowledgement

We would like to thank Prof. Wil van der Aalst for his valuable comments on an earlier version of this paper.

REFERENCES

Aalst, W. v. d., Dumas, M., ter Hofstede, A., and Wohed, P. (02a). Pattern-Based Analysis of BPML (and WSCI). Technical report, FIT-TR-2002-05, Queensland University of Technology, Brisbane.

Aalst, W. v. d., ter Hofstede, A., Kiepuszewski, B., and Barros, A. (02b). Workflow patterns. Technical report FIT-TR-2002-2, Queensland University of Technology. Accessed from `http://www.tm.tue.nl/it/research/patterns`. To appear in Distributed and Parallel Databases, Kluwer.

Casati, F., Ceri, S., Pernici, B., and Pozzi, G. (95). Conceptual modeling of workflows. In Papazoglou, M., editor, *Proc. of the 14th Int. Object-Oriented and Entity-Relationship Modelling Conference (OOER'95)*, volume 1021 of *LNCS*, pages 341–354. Springer Verlag.

Dumas, M. and ter Hofstede, A. (01). UML activity diagrams as a workflow specification language. In Gogolla, M. and Kobryn, C., editors, *Proc. of the 4th Int. Conf. on the Unified Modeling Language (UML01)*, volume 2185 of *LNCS*, pages 76–90. Springer Verlag.

Harel, D. (87). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

HP (02). HP Process Manager, Interactive Edition. Accessed Sep 02, `www.ice.hp.com/cyc/af/00/101-0601.pdf`.

IBM (02). IBM WebShere MQ software. Accessed Sep 02, `www.ibm.com/software/ts/mqseries`.

IONA (02). Orbix E2A Application Server Platform. Accessed Sep 02, `www.iona.com/products/appserv.htm`.

ITU (01). Int. Telecommunication Union, Specification and Description Language (SDL). Accessed Nov 01 from `www.itu.int/rec/recommendation.asp?type=products&parent=T-REC-z`.

Johannesson, P. and Perjons, E. (01). Design principles for process modelling in enterprise application integration. *Information Systems*, 26(2):165–184. Special Issue on Practical Applications of Agents.

Kiepuszewski, B., ter Hofstede, A., and Bussler, C. (00). On structured workflow modelling. In Wangler, B. and Bergman, L., editors, *Proc. of the 12th Int. Conf. on Advanced Information Systems Engineering (CAiSE00)*, volume 1789 of *LNCS*, pages 431–445. Springer Verlag.

Linthicum, D. S. (00). *Enterprise Application Integration*. Addison-Wesley.

Ruh, W., Maginnis, F., and Brown, W. (01). *Enterprise Application Integration: A Wiley Tech Brief*. John Wiley and Sons, Inc.

Söderström, E., Andersson, B., Johannesson, P., Perjons, E., and Wangler, B. (02). Towards a framework for comparing process modelling languages. In Pidduck, A., Mylopoulos, J., Woo, C., and Özsu, M., editors, *14th Int. Conf. on Advanced Information Systems Engineering (CAiSE02)*, volume 2348 of *LNCS*, pages 600–611. Springer Verlag.

Vitria (02). Business ware: The leading integration plattform. Accessed Sep 02 from `www.vitria.com/library/brochures/vitria_businessware_brochure.pdf`.

WfMC (00). Workflow Management Coalition Workflow Standard - Interoperability Wf-XML Binding. Accessed Aug 02 from `www.wfmc.org/standards/docs/Wf-XML-1.0.pdf`.

WfMC (99). Workflow Management Coalition: Terminology and Glossary, WFMC-TC-1011. Accessed Feb 99 from `www.wfmc.org`.

Wohed, P., van der Aalst, W., Dumas, M., and ter Hofstede, A. (02). Pattern-Based Analysis of BPEL4WS. Technical report, FIT-TR-2002-04, Queensland University of Technology, Brisbane.

Wåhlander, C., Nilsson, M., and Törnebohm, J. (01). Visuera PM Modeler. Accessed Jun 02 from `www.visuera.com/en/index.htm`.