Transforming Dynamic Behavior Specifications from Activity Diagrams to BPEL

Nasser Mousa Faleh . Mustafa
SEECS, University of Ottawa
Ottrawa, Canada
Nmust041@uottawa.ca

Gregor V. Bochmann
SEECS,University of Ottawa
Ottawa, Canada
bochmann@site.uottawa.ca

***Abstract* - The Service-Oriented Architecture (SOA) provided by the Web Services standards supports Model-Driven Development, it allows global business process models described in the Business Process Modeling Notation (BPMN) or as UML Activity Diagrams to be transformed into Web Services components specified by WSDL and/or BPEL. We have experimented the transformation of UML Activity Diagrams to several BPEL processes using the IBM Rational Software Architect (RSA) tool. These diagrams were derived from the specification of global system behavior where each activity may represent some collaboration between several system components in distributed systems. The derived component behaviors assure that the global behavior will be realized by coordinating the actions of the components through the exchange of asynchronous messages. In this paper, we describe how this method can be adapted to the context where the system components will be implemented as BPEL processes. We found out that the IBM Rational tool does not support some important asynchronous message exchange scenarios, and we describe here how the generated BPEL processes can be manually adapted. We also discuss some difficulties that arise in relation with input message buffering. since we assume that the received messages remain in a buffer pool until they are required by the destination process. This message buffering is largely provided by the BPEL execution environment. We explain in this paper how all these problems can be resolved by simple modifications of the automatically generated component behaviors in BPEL.**

Keywords: Distributed applications, Web Services, Service Oriented Architecture, BPEL, UML, Activity diagrams, transformations, race conditions.

I. INTRODUCTION

The increasing demands for more software applications and their complexity have urged researchers in software engineering to think about new tools to simplify the job of software engineers. The standardized Unified Modeling Language (UML) [1] has attracted much attention. UML provides the possibility to define different types of diagrams, generate automatic documentation, and allow software developers to communicate with a high level of abstraction. In addition, UML models can be imported, exported and transformed into other business models through automatic transformations. These models play an important role in Model Driven Architecture (MDA) [2], specifically for the transformation from UML to the Service Oriented Architecture (SOA) [3], including the Business Process Execution Language (BPEL) [4]. There has been a considerable amount of work on the transformation from UML or the Business Process Modeling Notation (BPMN) to BPEL. Most of this previous work considers that the generated BPEL process communicates with other Web Services (WS) through synchronous method invocations for the performance of certain actions. In contrast, we consider in this paper the situation where several BPEL processes running in different servers collaborate by asynchronous message passing. This situation occurs naturally in the context of distributed workflow implementations. The examples considered in this paper arise in the context of a development process of distributed applications where the requirements are first defined at a high level of abstraction in terms of an Activity diagram where each activity is either a collaboration between several parties, or an action that is performed by one of these parties. Our work extends a previous work that was developed by [15,16] for specifying such high-level collaborations, in addition to formulating an algorithm for deriving the specification of the dynamic behavior of each of the involved parties in terms of local actions and the exchange of coordination messages that coordinate the actions of the different parties involved. Moreover, the author of [17] used the Eclipse tool to implement this algorithm and produce the local behavior of each party in the form of a local Activity diagram. For the implementation of these local behaviors in the framework of Web Services, we are therefore interested in transforming the diagrams defining the local behaviors into BPEL processes that are executed on different server computers.

In this paper we describe our experimentation with the transformation of UML-2 Activity diagrams into PBEL processes with asynchronous message passing, using examples coming from the derivation of local behaviors as explained above. In this context, it is important to avoid race conditions between different message reception and sending events. This can be achieved by introducing a local input message pool where received messages are stored until they are requested (consumed) by the application process (that is, the BPEL process). This distinction between reception and consumption of messages is important for the correct operation of the distributed system. The Web Services execution environment that we used for our experiments supported this distinction, however, not to the full degree required. In fact, the application may indicate which type of message it wants to consume, but it cannot indicate which particular message parameter value it is waiting for. Because of this missing feature, we had to introduce additional complications in the translation process in the case that the behavior includes certain types of loops. We also noted that the translation tool we used for our experiments did not correctly deal with the choice between two different types of messages that could be received at a given point during the execution of the behavior.

This paper is structured in three sections. Section II introduces a background on the UML to SOA transformation and an example of an application with asynchronous messages exchanges. Section III discusses the limitations of the UML to BPEL transformations and proposes solution for these limitations. We conclude the paper with conclusions and future work in Section IV.

II. BACKGROUND

A. *Transformation from UML to Service Oriented Architecture (SOA)*

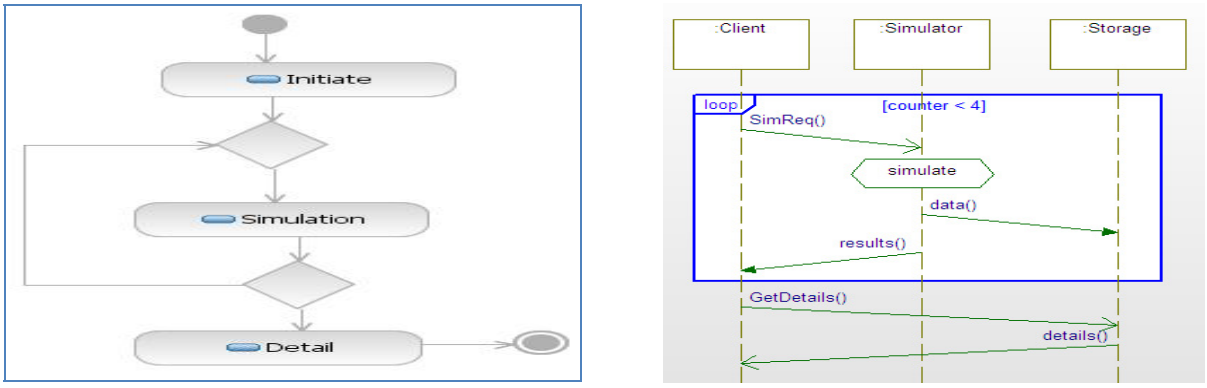
The Service-Oriented Architecture (SOA) provides design principles for loosely integrating services that could be provided by different business domains. Using the Web Services (WS) standards of the WSDL service descriptions and the SOAP communication protocol, these services could either be newly created processes or encapsulations of legacy software systems that are made available over the Internet. Several services may be integrated using a central control process, a design principle called orchestration. Often this control process is described using the Business Process Execution Language (BPEL).

The Web Services standards provide two kinds of communication between different processes: (a) synchronous and (b) asynchronous. For synchronous communication, which is a kind of remote procedure call, one normally distinguishes a *Client* process and a *Server* process. A synchronous communication consists of two message exchanges: First the *Client* sends a message requesting the execution of a specific method with specified parameters. Then at the end of this execution, the *Server* returns a message containing the result information to the *Client*. The *Client*, in the meantime, waits for this answer. In the case of asynchronous communication, any process may, at any time, send a message to another process, and normally continues some local processing. Each reception of a message normally leads to some local processing which may include the transmission of new messages to other processes. The orchestration design principle normally uses synchronous communication; another design principle called choreography favors asynchronous communication. An example of asynchronous communication is a *Client* that sends a change of billing address to the service.

BPEL is a kind of programming language, encoded in XML. It is intended for defining programs that coordinate the execution of Web Services that are provided on different servers, possibly by different organizations. The control flow constructs of the language include sequential execution, alternatives, loops and concurrency, besides the basic operations of updating local variables and calling methods provided by local or remote services. Since these control structures are quite similar to those of UML Activity diagrams and the Business Process Modeling Notation (BPMN), there has been much work on the automatic translation from UML Activity diagrams into BPEL processes. A WS-BPEL meta-model and process interaction meta-model have been defined [5] for generating BPEL processes from UML-2 system models. The author of [6] proposes a model-driven approach for extending UML-2 Activity diagrams (AD) with business process goals and performance measures and describes a mapping into BPEL. A UML-2 extension for SOA – called the UML4SOA profile for modeling service orchestrations was introduced by [7], including the transformation from UML4SOA Activity diagrams into executable languages such as BPEL and Java. Others defined a transformation method from the Business Process Modeling Notation (BPMN) into BPEL [8].

B. *Distributed System Design from Global Service Specifications*

As mentioned in the introduction, the work performed by [15] specified graphical notation based on UML Activity diagrams for distributed applications at a high-level of abstraction called “choreography”. Each activity within the diagram is either a collaboration between several parties involved in the distributed application, or a local action by one of these parties. These collaborations may be further refined in terms of sub-collaboration diagrams, in the form of “choreographies” or simply by a sequence diagram showing the local actions and messages exchanged between the parties. An example is shown in Figure 1 below.



(a) **(b)**

Figure 1: *Client-Simulator-Storage* - (a) High-level view, (b) Sequence diagram

The high-level view of this example application (see Figure 1(a)) includes the *initiate*, *simulation* and *detail* collaborations. The *initiate* collaboration is started by the *Client* to initiate the *Storage.* Figure 1(b) shows the repeated execution of the *simulation* collaboration followed by the *details* collaboration. We now assume that this application involves three parties: the *Client*, the *Simulator,* and the *Storage*. The role of the *Storage* party is to store the detailed data of each simulation, and provide the details of the last *simulation* when requested by the *Client* during the *details* collaboration (which only involves the *Client* and the *Storage*). We further assume that (a) the *simulation* collaboration is initiated by the *Client* by sending a *SimReq* message to the *Simulator*, and is concluded by the *Simulator* sending the detailed results of the simulation to the *Storage* and a summary of the results to the *Client*, and (b) that the *details* collaboration is a simple remote call by the *Client* of the method *GetDetails* provided by the *Storage*. Under these assumptions, this *Client-Simulator-Storage* example will give rise to execution scenarios described by the sequence diagram of Figure 1(b). We note that in such distributed applications one has to distinguish between strong and weak sequencing. The loop in this example is a weak loop, which means that the *Client* may start the next repetition of the loop before the *Storage* has received the *data* message. In applications with weak sequencing, there are often situations of race conditions between the reception of messages from different parties or race conditions between sending and receiving messages. A systematic review of these difficulties is given in [11] and [15]. This is a very simple example. Note, however, that the decision about the termination of the loop will be performed by the *Client* (who chooses between the messages *SimReq* and *GetDetails*). If the loop should be executed 4 times, as indicated in Figure 1(b), then the *Client* should have a local counter variable that is incremented during each execution of the loop. We also note that a race condition may occur at the *Storage* party in the case that the last *data* message encounters a long transmission delay from the *Simulator*, and the subsequently sent *GetDetails* message from the *Client* arrives earlier. If this race condition is not detected, the *Storage* party will return the detailed results of the before-last simulation, which is an error. We will discuss below how such race conditions can be dealt with.

C. *Implementing distributed system designs as Web Services using BPEL*

For the work described in this paper, our objective was to evaluate the possibility of a semi-automatic process for the implementation of distributed applications as Web Services, We starts with the work of [16,17] that describes the high-level of the overall system behavior. Given that their Eclipse software tool provides for the automatic transformation from the high-level global collaboration diagram to local UML Activity diagrams for each of the involved parties, and that the IBM RSA tool provides for the automatic translation from UML Activity diagrams to corresponding BPEL processes, it was natural to combine these two transformation processes into a single automated implementation process starting with the global high-level system behavior description and the identification of the parties involved and ending with a BPEL process for each party running in the IBM WebSphere environment.The following section describes the results of our experiments, explains some difficulties encounters and provides some solutions.

III. UML TO BPEL TRANSLATION: REALIZATION AND LIMITATIONS

We describe in this section the results of our experimentation with the IBM RSA tool in view of automatically transforming a UML Activity diagram representing a given party behavior into a BPEL process that is executed within the IBM WebSphere environment. Most of the difficulties encountered are related to the reception of asynchronous messages and their consumption from the local message pool.

A. *Message consumption from the message pool*

Our experiments confirmed that the WebSphere environment implementation has a message reception pool for each BPEL process from which messages can be consumed in the order in which the BPEL process requests them. In the case of a request of a specific message type, the message is consumed by a *Receive* primitive. In the case that the BPEL process is ready to consume a message among several different message types, the BPEL *Pick* primitive can be used. However, we have not found any way to specify that a message of a given type with a given parameter value should be consumed from the message pool. Also alternatives between sets of several concurrent message consumptions cannot be realized by BPEL processes. How these difficulties can be accommodated is described in the following subsections.

B. *Alternative message receptions*

In UML activity diagrams, a choice may have different semantics: (a) the choice may depend on some local condition (e.g. a Boolean variable with value true or false), or (b) in the case of the choice between two message receptions, the choice will depend on which message is first received (or in the presence of a message pool, which message is available for consumption). We consider as an example the *send* activities in Figure 2: The *Client* party sends either message *A* or message *B* to the *Service* party. The behavior of the latter is shown in Figure 3**:** the *Service* can receive either message. We note that the *Client* makes a decision based on the value of the *condition* variable, while the *Service* has no decision to be made; it receives either A or B, but never both. The transformation function of the RSA tool did not distinguish these two types of decision nodes. Both decision nodes, of the *Client* and of the *Service*, were transformed into a BPEL decision node. This is correct for the *Client*, but not for the *Service*. A decision node between several alternative message receptions should rather be translated into a BPEL *Pick* primitive. We have approached this failure of the automatic transformation function by manually changing the resulting decision node in the BPEL *Service* process into a *Pick* node. The alternative message reception events are represented in the BPEL process using <onMessage> elements. The *Pick* executes the alternative that starts with the reception of the events that occurs first. Figure 4 shows the automatically obtained BPEL process for the *Client* party, while Figure 5 shows the BPEL process for the *Service* party after the manual transformation.

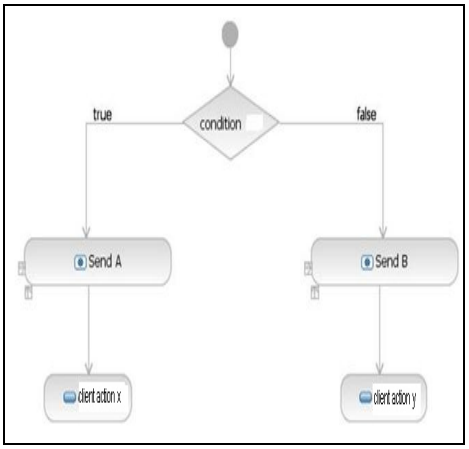


Figure 2: Activity diagram - *Client* behavior.

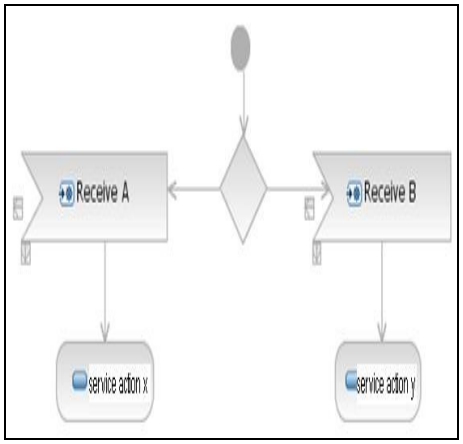


Figure 3: Activity diagram - Service behavior

.

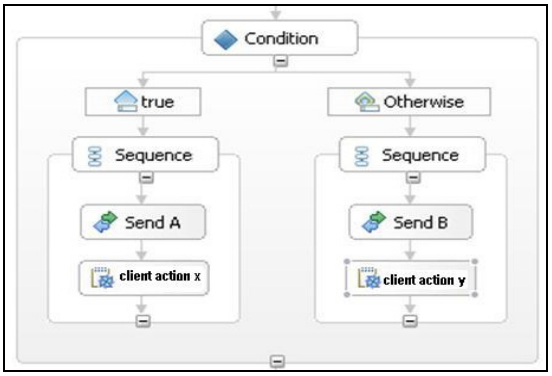


Figure 4: BPEL process of *Client* party with decision node.

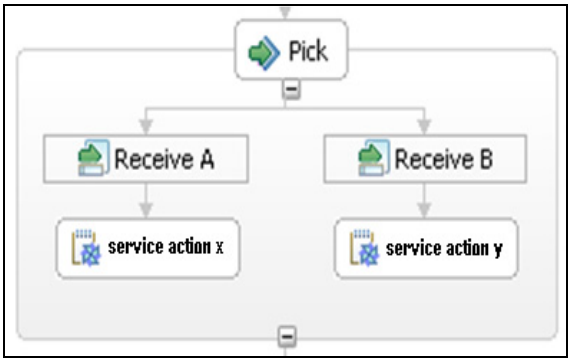


Figure 5: BPEL process of Service party with alternative messages reception

C. ***Alternative with concurrent message reception***

The situation concerning alternative actions becomes more complicated if one alternative involves the concurrent reception of several messages, as illustrated by the *Service* party shown in Figure 6. The second alternative includes the reception of the two messages B and C with respective actions y and z subsequently. In the Activity diagram, each receive activity is modeled as an *accept call action* node which is assigned to a *call event* trigger that specifies the type of message to receive. This behavior cannot be translated into a BPEL process with a ***Pick*** node involving the three message types, since the ***Pick*** node receives only one message, never two. We propose in the following subsections two different solutions to this problem: (a) converting the concurrency into alternatives, and (b) adding an extra message before the concurrent messages.

1. Converting the concurrency to alternatives

It is well-known that concurrency can be implemented by interleaving. As shown in Figure 7, the two concurrent receptions can be modeled by two alternatives, one first receiving B, the other first receiving C. In the case that B is received first, the action y following this reception is executed concurrently with the reception of C and its following action z. This Activity diagram can be translated into BPEL using the method described in Section III part B. The three alternatives of Figure 7 are selected by a *Pick* node which is manually inserted.

2. Adding an additional message before the concurrent receives

An alternative solution is to add an additional message transmission when the concurrent alternative is chosen. This leads to the "equivalent" Activity diagram of Figure 8, which involves an additional message transmission, but has a simple overall structure. In case of receiving the extra message, the reception of the messages B and C will be performed concurrently. The transformation into BPEL can be performed as described in Section III part B.

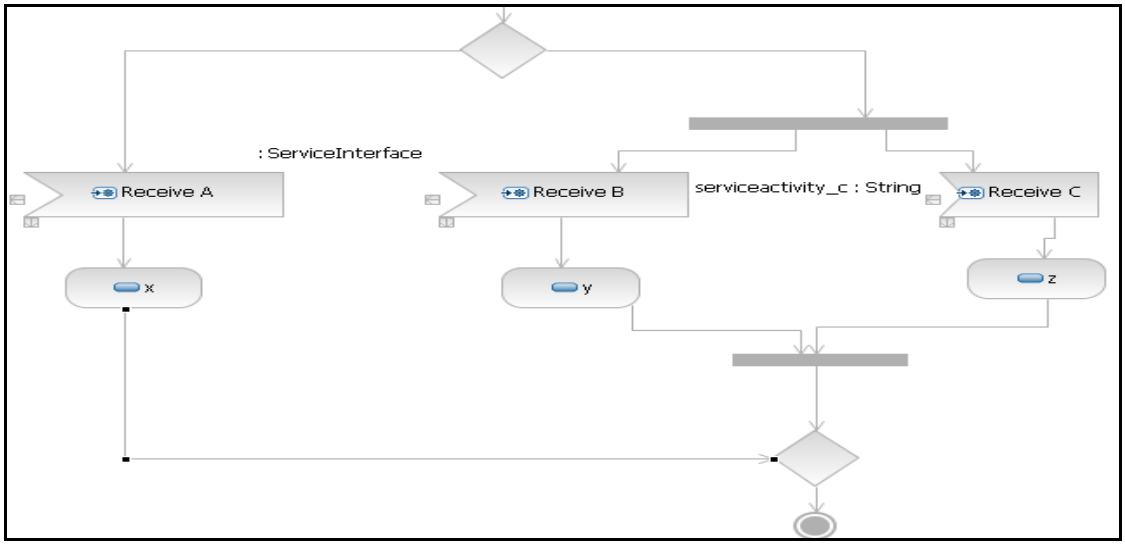


Figure 6: Activity diagram showing the choice between a single and a concurrent message reception

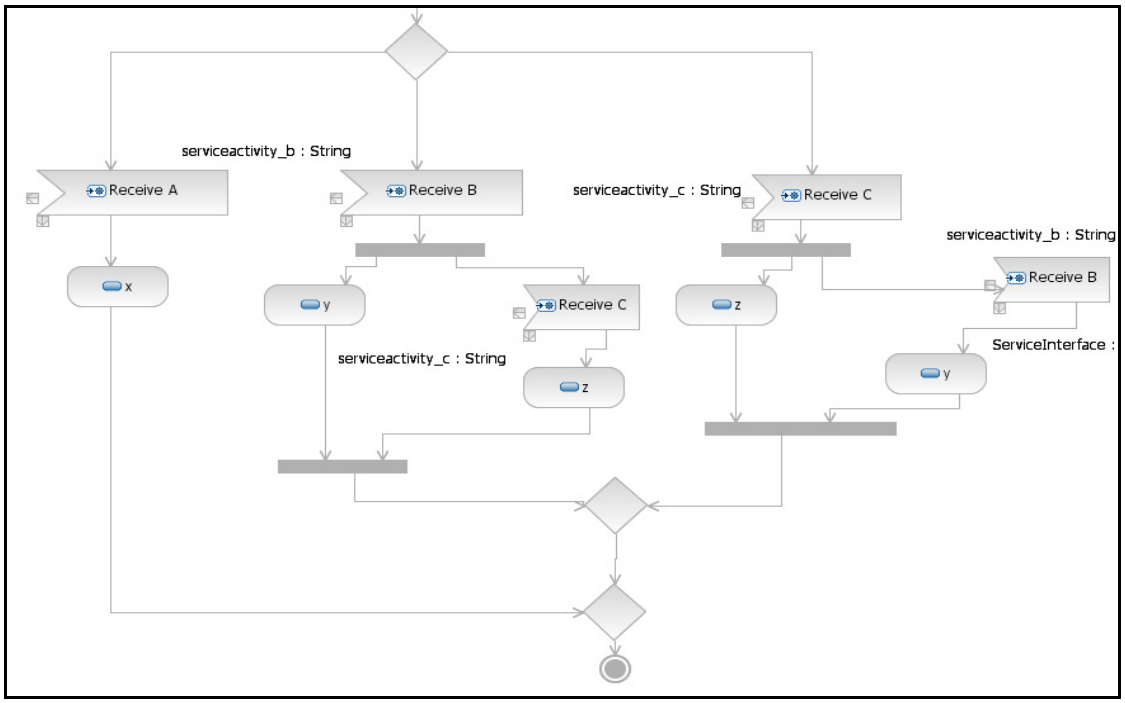


Figure 7: Transformed Activity diagram equivalent to the diagram of Figure 6

D. ***Race conditions and weak loops***

As mentioned in Section II part B, race conditions can be handled by introducing a message reception pool at each party and letting the destination process determine when it is ready to consume any received message, instead of requiring a message to be processed when it is received. For this purpose, it is necessary that the destination process can request the consumption of a given type of message, or one out of several alternative messages, as discussed in the previous subsections. It is therefore necessary that the different messages that may be received by the destination process during different stages of its processing belong to different message types. However, there are situations where the distinction between different message types is not sufficient, but the distinction of which message to consume depends also on the values of certain message parameters. This is the case in the example introduced in Section II part B which is considered here again. The issues concerning races that may occur in the coordination of collaborations between several distributed parties, especially in the presence of weak sequencing, are discussed in detail in [11] and [15]. The example of Section II part B (see Figure 1) contains a weak loop and the possibility of a race between the reception of the last *data* message by the *Storage* party and its reception of the *GetDetails* message. It has been proposed that such races can be handled by including in all messages involved a sequence number which indicates how many times the loop has been executed. Each party involved updates a

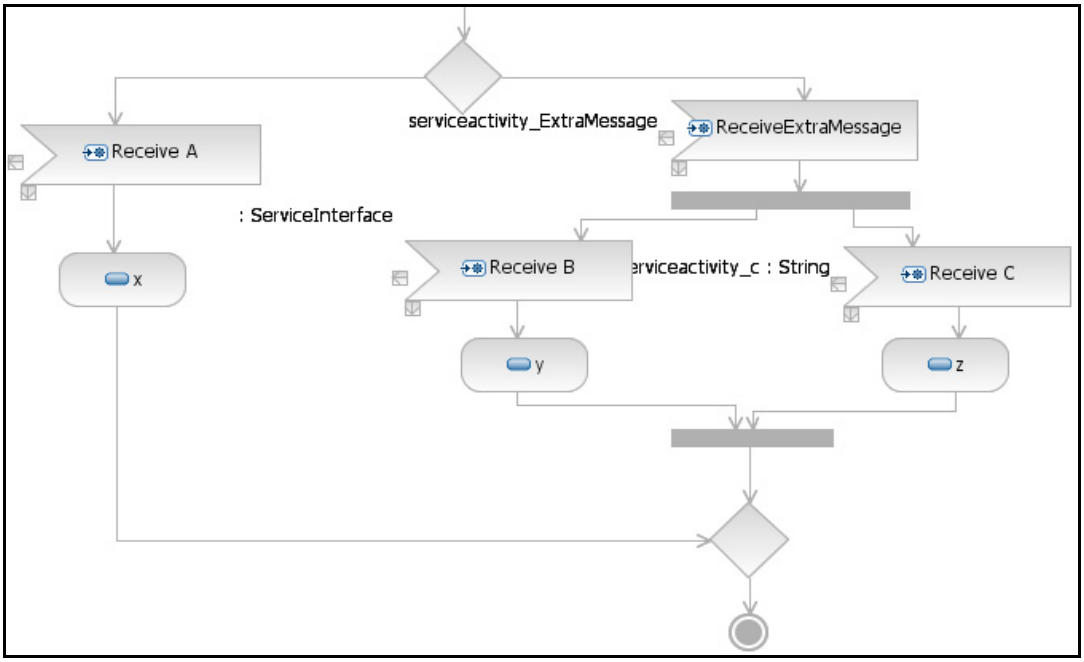


Figure 8: Activity diagram with additional message corresponding to Figure 6.

local counter variable and therefore knows what the value of the sequence number parameter should be for the next message to be consumed. Initially, these counter variables have the value zero. The *Client* party will increment its counter before its value is included in the first *SimReq* message, and the *Simulator* party will forward the received parameter value in the parameters of the message sent. As a result, the *Storage* party will initially only consume a *data* message with parameter value equal to one (one higher than its local counter); it will then increment its counter and wait for the next message. If a *GetDetails* message is received, it can only be consumed if its parameter is one larger than the current counter value of the *Storage* party. This is shown in *Figure 9*, where *Figure 9(a)* represents the behavior of the *Client* party, while *Figure 9(b)* represents the behavior of the *Storage* part. This procedure is included in the distributed system designs generated by our tool mentioned in Section II part B [16, 17]. However, this procedure cannot be translated into our BPEL environment because the message pool of the IBM WebSphere environment can distinguish messages only by their message type, and not by their parameter values.

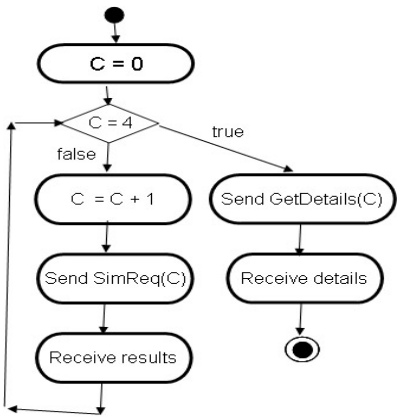


Figure 9(a): Behavior of *Client*

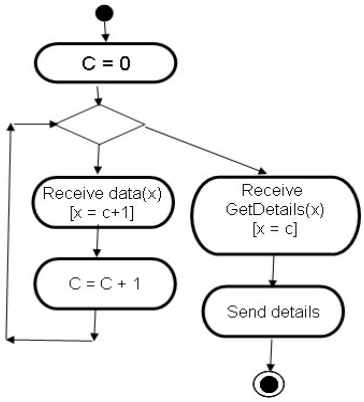
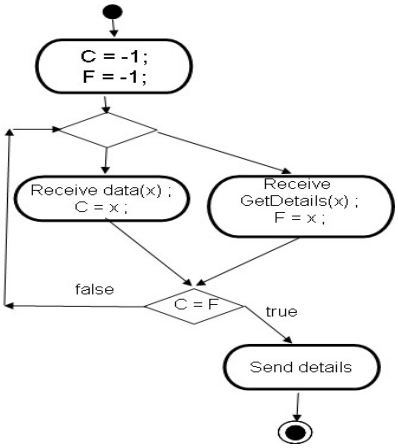
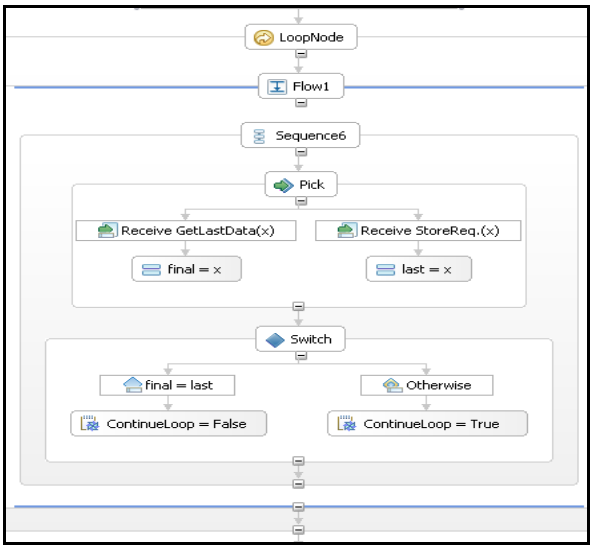


Figure 9(b): Behavior of *Storage*

We have found a solution to this problem by using a different behavior pattern for the process that has to distinguish the parameter values of the messages that control the execution of a weak loop. Instead of the behavior pattern shown in *Figure 9*(b), we propose the behavior pattern shown in *Figure 10*(a) which deals with the example of the *Storage* party in our *Client-Simulator-Storage* example. The party has two local variables, a *counter* which contains the parameter value of the last *data* message received, and a *final* variable which contains the value of the *GetDetails* message parameter if this message has been received. Both variables are initialized to -1. We assume here that messages are delivered in order between any two communicating parties. When either a *data* message or the *GetDetails* message has been received, the loop can terminate if the two variables contain the same value; otherwise some additional *data* message is expected.This behavior pattern can be easily transformed into BPEL as explained in Section III part B. The BPEL behavior obtained for the *Storage* party is shown in *Figure 10*(b).



(a)



(b)

Figure 10: Behavior of *Storage* party
(a) Activity diagram; (b) BPEL process

IV. Conclusion

We have shown how distributed system collaborations can be modeled as UML activity diagrams and be transformed into distributed system designs involving several parties typically implemented as separate processes in different components. We have discussed in detail certain difficulties that arise when one tries to transform the behavior each these separate processes into BPEL in order to obtain an implementation in the context of the Service-Oriented Architecture. Transformations from UML Activity diagrams into BPEL processes have been studied by many researchers from different points of view, such as improving performance, or using notations other than UML. In this paper we consider systems communicating by asynchronous message passing. We have identified certain shortcomings of existing automatic UMLtoBPEL transformations and have proposed solutions that assist in the production of correct transformations. We were able to provide two types of solutions: (a) automatic solutions by modifying the original activity diagram in such a way that the automatic transformation process into BPEL works correctly, and (b) manual intervention which involves the change of BPEL processes obtained by the automatic transformation process.

References

[1] OMG Unified Modeling Language. UML Resources. [Online] 2010. [Cited: 10 26, 2010.] http://www.uml.org/.

[2] OMG Model Driven Architecture. [Online] 2010. [Cited: 10 26, 2010.] http://www.omg.org/mda/.

[3] OMG Service Oriented Architecture. [Online] 2010. [Cited: 10 26, 2010.] http://soa.omg.org/.

[4] OASIS BPEL. 2010. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

[5] Chenting Zhao, Zhenhua Duan, and Man Zhang. A Model-Driven Approach for Generating Business Processes and Process Interaction Semantics. 2009, Eigth IEEE/ACIS International Conference on Computer and Information Science.

[6] Birgit Korherr, and Beate List. Extending the UML 2 Activity Diagram with Business Process Goals and Performance Measures andthe Mapping to BPEL. SpringerLink, 2006, Vol. 4231/2006.

[7] Philip Mayer, Andreas Schroeder, Nora Koch. A Model-Driven Approach to Service Orchestration. 2008, IEEE International Conference on Services Computing.

[8] Chun Ouyang, Wil M.P. van der Aalst, Marlon Dumas, and Arthur H.M. ter Hofsteder. Translating BPMN to BPEL. 2006, International Conference on Web Services, pp. 285-292.

[9] Peter Houston, Microsoft Corporation. Selecting Between Synchronous and Asynchronous Alternatives. [Online] [Cited: 09 30, 2010.] http://wiki.daimi.au.dk/pca/_files/selectingbetweensynch.pdf.

[10] Dmitry Gorelik . Transformation to SOA. IBM. [Online] IBM. [Cited: 09 30, 2010.] http://www.ibm.com/developerworks/rational/library/08/0115_gorelik.

[11] H. N. Castejon, G. v. Bochmann and R. Braek, Realizability of Collaboration-based Service Specifications, Proc. Asia-Pacific Software Engineering Conference (APSEC), Nagoya, Japan, Nov. 2007.

[12] Arjan J. Mooij, Nicolae Goga and Judi Romijn Non-local Choice and Beyond Intricacies of MSC Choice Nodes. FASE, 2005, pp. 273–288 .

[13] Arjan Mooij, Jiud Romijn, and Wieger Wesselink. Realizability criteria for compositional MSC. Springer, 2006. Proc. of 11th Intl. Conf. on Algebraic Methodology and Software Technology. Vol. vol. 4019.

[14] WebSphere Integration Developer. IBM. [Online] [Cited: 10 29, 2010.] http://www-01.ibm.com/software/integration/wid/.

[15] Humberto Nicolás Castejón, Gregor v. Bochmann, and Rolv Bræk. On the realizability of collaborative services, Journal of Software and Systems Modeling, to be published.

[16] G. v. Bochmann, Deriving component designs from global requirements, Proc. Intern. Workshop on Model Based Architecting and Construction of Embedded Systems (ACES), Toulouse, Sept. 2008

[17] F. Laamarti, Derivation of component designs from a global specification, MSc thesis, University of Ottawa, 2010. See http://www.site.uottawa.ca/~bochmann/dsrg/PublicDocuments/Mastertheses/Laamarti%20-%20Derivation-of-Component-Designs-from- Global-Specifications.pdf