Transformation of UML Activity Diagrams into Business Process Execution Language

Nasser Mousa Faleh Mustafa

A thesis submitted to the Faculty of Graduate and Postdoctoral Studies

In partial fulfillment of the requirements

For the degree of Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Electrical Engineering and Computer Science

University of Ottawa, Ottawa, Ontario, Canada 201

Abstract

Researchers in software engineering proposed design method for distributed applications to construct a set of communicating system components from a global behavior. The joint behaviors of these components must precisely satisfy the specified global behavior. The next concern is to transform the constructed models of these components into executable business processes by ensuring the exchange of asynchronous messages among the generated business processes. The introduction of Service-Oriented Architecture (SOA) has helped to achieve this goal. SOA provides high flexibility in composing loosely-integrated services that can be used among business domains to carry out business transactions; this composition is known as service orchestration. Moreover, SOA supports Model Driven Architecture (MDA) such that services modeled as UML Activity Diagrams (AD) can be transformed into a set of Business Execution Language (BPEL) processes. Many researchers discussed the transformation of UML AD and the Business Process Modeling Notation (BPMN) into BPEL. However, they did not discuss the practical limitations that some of these transformations impose.

This thesis addresses the imitations of the transformation from UML AD to BPEL processes using the IBM Rational Software Architect (RSA). We showed here that the tool is unable to create the correct BPEL artifacts from UML AD components in certain cases, for instance when the behavior includes the alternative for receiving single or concurrent messages, a weak loop, or certain choice activities. Furthermore, we provided novel solutions to the transformations in these cases in order to facilitate the transformation from UML AD to BPEL.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AD	Activity Diagram
BPEL	Business Process Execution Language
BPEL4WS	Business Process Execution Language for Web Services
BPMN	Business Process Modeling Notation
HTTP	Hyper Text Transfer Protocol
ISO	International Organization for Standardization
MDA	Model Driven Architecture
OASIS	Organization for the Advancement of Structured Information Standards
OMG	Object Management Group
PIM	Platform-Independent Model
PSM	Platform-Specific Models
RSA	Rational Software Architect
SCA	Service Component Architecture
SCDL	Service Component Definition Language
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
WID	WebSphere Integration Developer
WPS	WebSphere Process Server
WS	Web Service
WSDL	Web Service Definition Language
XML	Extensible Markup Language
XSD	XML Schema Definition

Acknowledgment

This research would not have been possible without the support of many people. I would like to extend my heartfelt thanks to my supervisor Professor Gregor V. Bochmann for his guidance, patient and invaluable assistance through this research. My deep gratefulness to Dr. Liam Peyton who provides me with the necessary software for this project, also I would like to convey my thanks to the University of Ottawa for providing the facilities during my research study. Also, I would like to offer my thanks to Mr. Murali Pattathe from IBM-Canada for his help. Most especially, I would like to thank my beloved wife, my kids, and my family friends who gave me the courage to complete this work, and to God who made this possible.

Chapter One

1 Introduction

The natural complexity of software applications and the increasing demand for more software applications have urged researchers in software engineering to think about new tools that have consensus among software companies for simplifying the job of software engineers. Researchers proposed the Unified Modeling Language (UML) [1]. UML is considered an excellent graphical language that has the ability to define diagrams, generate automatic documentation, and allow software developers to communicate with a high level of precision. In addition, UML models can be imported, exported and transformed into other business models for further development. Moreover, it has been utilized in Model Driven Architecture (MDA) [2]. UML is a Platform-Independent Model (PIM) that can be transformed into Platform-Specific Models (PSM) such as BPEL [3].

1.1 Motivation

Modeling a set of communicating system components such that their joint behaviors precisely corresponds to a given the global behavior is challenging. A smart solution would be to model the behavior of each system component separately, then integrate the derived models and verify the solution. This solution seems theoretically simple, but in reality it is challenging because many coordination messages are involved. To track these messages the model should be integrated and executed. UML can help to define the behavior of the individual components, but it cannot be used to execute the behavior of these components. SOA and MDA provide an

execution environment. While SOA provides high flexibility in composing loosely-integrated services that can be used among business domains to carry out business transactions, MDA utilizes SOA to integrate the UML components then, these components can be transformed into a PSM model such as BPEL that can be executed.

The difficulty is relying on the transformation step, since the transformation does not always produce the right BPEL artifacts. UML is graph-oriented language, whereas, BPEL is mainly block-structured language, differences can be noticed during mapping between UML and BPEL artifacts.

1.2 Thesis Objectives

Automating the transformation from UML AD to BPEL is of great interest to our research. The objective of the thesis is to automate the transformation of a global system behaviour into a set of BPEL processes that realize the compositional service definition. Furthermore, the behavior resulting from executing the integrated BPEL processes must satisfy the global system behavior precisely, by including the exchange of asynchronous messages for the coordination the actions among the BPEL processes.

1.3 Thesis Scope

The thesis covers the following three facets:

The first aspect is utilizing the activity diagrams that are derived from the global system behavior using the derivation rules proposed by [4] and generated by [5] using the Eclipse tool.

The second aspect is modifying and transforming the activity diagrams into BPEL processes. This step is achieved by: First, import the derived AD of each component into the IBM Rational Software Architect (RSA) [6]. Second, modify the activity diagrams by specifying the message types and parameters,

conditions, and input variables. Third, create the structured diagrams and define the required and provided interfaces for each component. Finally, perform the transformation that generates the Web Service Definition Language (.WSDL) [7] file for each interface, and the (.BPEL) file for each AD.

The last aspect is deploying the generated interfaces and BPEL files into the IBM WebSphere Integration Developer (WID) [8] for further development and execution. This step involves modifications to some BPEL artifacts that are not supported by the RSA tool such as: (a) the UML choice that involves alternative message receptions, (b) the race conditions especially when there is a weak sequence, and (c) the alternative that involves the concurrent receptions of several messages. We proposed solutions to these problems such that a semi-automatic transformation from AD to BPEL can be achieved. Furthermore, combining our solution with the previous work that was performed by [4] [5] it is possible to have 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 component running in the IBM WebSphere environment.

1.4 Thesis Organization

The thesis will be structured as follows: Chapter 2 demonstrates the derivation of UML activity diagrams from collaborations. A review of the UML activity diagrams is explained, and a case study is introduced to demonstrate the derivation algorithm.

Chapter 3 introduces basic concepts and notations that are important as background for the thesis.

Chapter 4 portrays the IBM RSA and the automation of the transformation from AD to BPEL. A brief description is given about IBM RSA, and a case study is given to explain the transformation process.

Chapter 5 explains the IMB WID development tool, and shows the necessary modifications to the BPEL artifacts generated from the RSA tool.

Chapter 6 highlights the limitations of the transformation from Activity diagram to BPEL and explains our proposed solutions.

Finally, Chapter 7 provides a conclusion about this research and suggests related future work.

Chapter 2

2 The Derivation of Local UML AD from Global Collaborations

In Chapter 1 we mentioned the utilization of the activity diagrams that are derived from a global system behavior using the derivation rules proposed by [4] and generated by [5] using Eclipse tool . This chapter gives an overview of the UML activity diagrams. In addition it presents a case study that explains the derivation method.

2.1 Overview of UML Activity Diagrams

The Unified Modeling Language (UML) is a standard graphical language for modeling object-oriented software. It was developed in the mid-1990s as a collaborative effort by James Rumbaugh, Grady Booch, and Ivan Jacobson [9]. There are thirteen types of UML diagrams that are widely used in modeling systems behavior, structure, and interactions, in addition to modeling business processes. For the purpose of this research, we will concentrate on the UML activity diagrams.

The UML Activity diagrams are used to model the control and data flow from one activity to another in a business process or workflow. Activity diagrams and flowcharts are similar since both types show the flow between the actions in an activity; however, activity diagrams can also model concurrent and alternate flows. Activity diagrams encompass many notations such as control nodes, action node, and structured activities. The *Storage* activity diagram in Figure 1 illustrates the following elements:

Control nodes: An activity diagram may contain the following control nodes:

- **Initial node**: A fill-in circle represents the starting point of the diagram. This node is indicated by label number 1 in Figure 1.
- **Final node**: A filled circle with a border is the ending point. An activity diagram can have zero or more activity final nodes. Figure 1 label number 9 illustrates this node.
- **Control Flow**: The control flow is an arrow that connects activities and goes in one direction. Label 7 in Figure 1 represents a control flow.

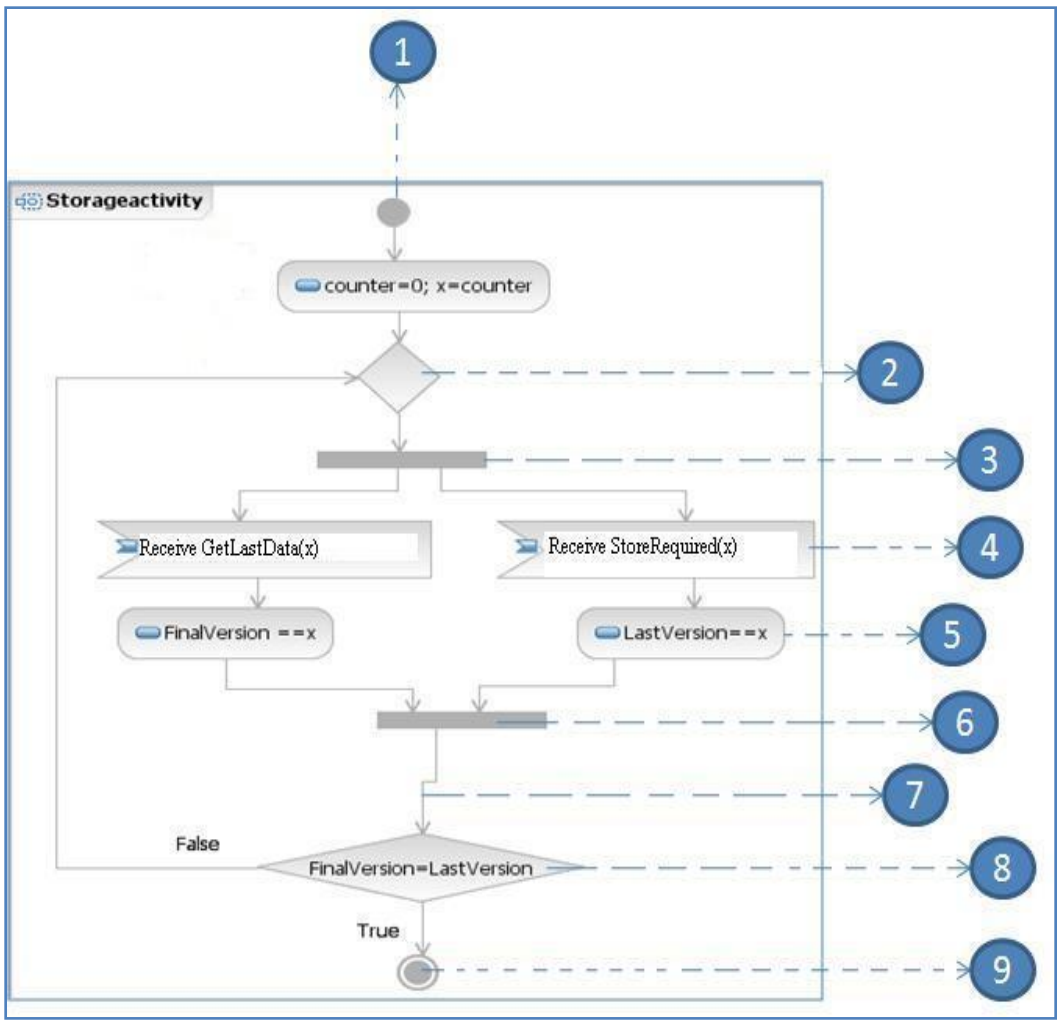


Figure 1: UML activity diagram.

- **Fork node**: A black bar with one flow going into it and several leaving. This denotes the beginning of parallel activities. Label 3 in Figure 1 illustrates a fork node.
- **Join node**: A black bar with several flows entering and one leaving it. All flows going into the join must complete before processing may continue. This denotes the end of parallel processing as shown by Label 4 in Figure 1.
- **Decision node**: A diamond with one flow entering and several leaving. The flows leaving include conditions. Label number 8 in Figure 1 shows an example of this type.

- **Merge node:** A diamond with many incoming flows and one leaving flow. The processing continues when one incoming flow reaches the merge node. This node is indicated by label 2 in Figure 1.

Action nodes: Actions are the fundamental elements in AD. The following action nodes are identified:

- **Opaque Action (Action):** This type is used as an action's placeholder. For example, the node in Figure 1 label 5 represents an opaque action node; it compares the variable *lastVersion* with *X* variable.

- **Call Operation (Invoke):** The Call operation is used to invoke a message type. Each call operation requires an input that specifies the process to which the request is sent, and in addition, message parameters of the invoked message type are required. A message type can have input and/or output message parameters. When a message type has only input message parameter, then an asynchronous call is made. When a message type has input and output message parameters, then a synchronous call with returned results is assumed.
- **Accept Call Actions (Receive):** Such actions represent the reception of a message of a particular message type. The called message type is assigned to the Accept Call action through triggers. The *Receive StoreRequired(x*) shown in Figure 1 (label 4) is an example of *AcceptCall* action.

Structured activities: a structured activity refers to a separate activity diagram that encompasses many basic activities.

2.2 Case Study: Client-Simulator-Storage Application

We have invented the *client-simulator-storage* application; it involves three collaborations between the *Client*, *Simulator*, and the *Storage components*. The collaboration in this context means an action that is performed by more than one component. The collaborations are the *initiate*, *simulation* and *detail* as shown in Figure 2. The *initiate* collaboration is started by the *Client* to initiate the *Storage*. During the *simulation*, the *Simulator* receives a defined parameter from the *Client*, performs the simulation, then sends the simulation result to the *Client* and the *Storage* concurrently as shown in the sequence diagram in Figure 3. In the *detail* collaboration, the *Client* requests the last result from the *Storage* and the *Storage* replies to this request.

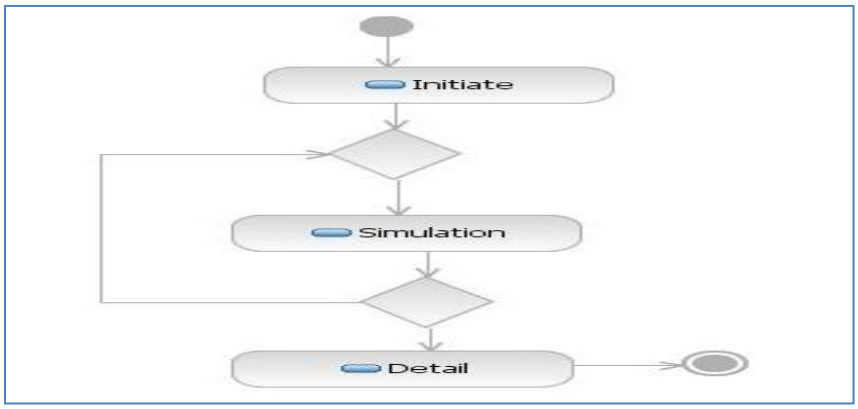


Figure 2: Client-Simulator-Storage Collaborations

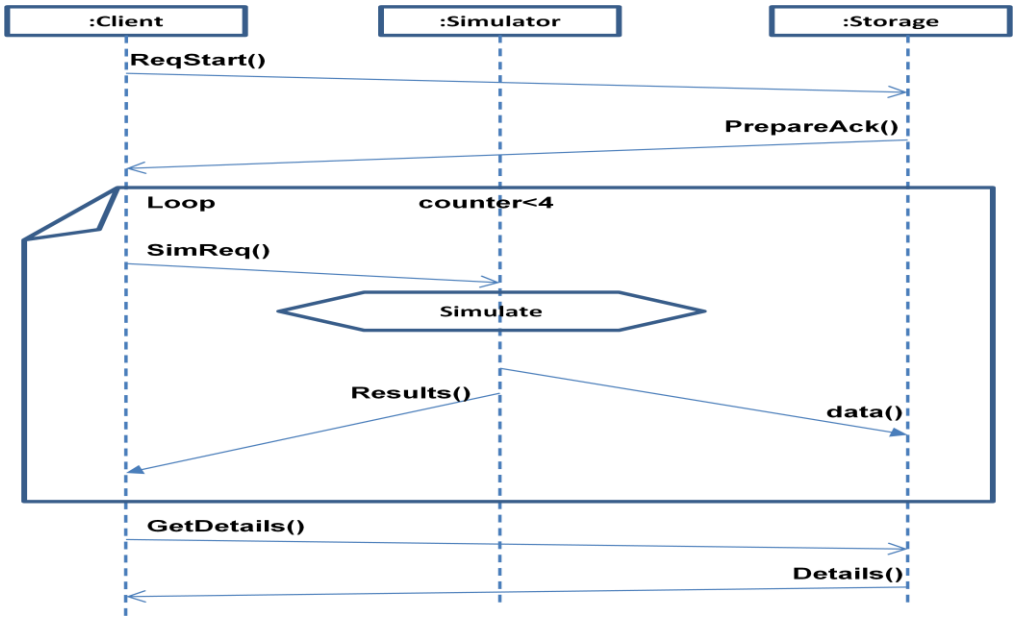


Figure 3: Client-Simulator-Storage Sequence diagram

The global collaboration activity diagram in Figure 4 illustrates the actions performed by each component. The *initiate* collaboration has the *ReqStart*, *PrepareAck*, and *ReceiveAck* local actions. The *ReqStart* and *ReceiveAck* are *Client* local actions that are responsible of initiating and receiving Acknowledgment from the *Storage*, while the *PrepareAck* is a *Storage* local action that prepares the acknowledgment message and sends it to the *Client*. The *Simulation* collaboration starts with a loop that contains local actions for the *Client*, *Simulator*, and *Storage* components. First, the *Client* starts with the *PrepareSimParam* action to prepare a parameter for simulation, next the *Simulator* performs the *Simulate* action, then the *CheckResults* and *StoreResult* are performed concurrently to check and store the result of the simulation. The *CheckResults* is a local action performed by the *Client*, while the *StoreResults* is performed by the *Storage*. The *detail* collaboration contains the *Client's* local action *ReqStorePerm* that requests the last simulated result from the *Storage*, and the *Storage's* local action

StorePermanent sends the last result to the *Client.* It is important to note that the small letters *s*, *p*, and *t* in Figure 4 stand for *starting*, *participating*, and *terminating* respectively, and the capital letters C, S, T represent the *Client*, *Simulator*, and *Storage* components, respectively. For instance, "s: C" means that the client is the component that starts the given action.

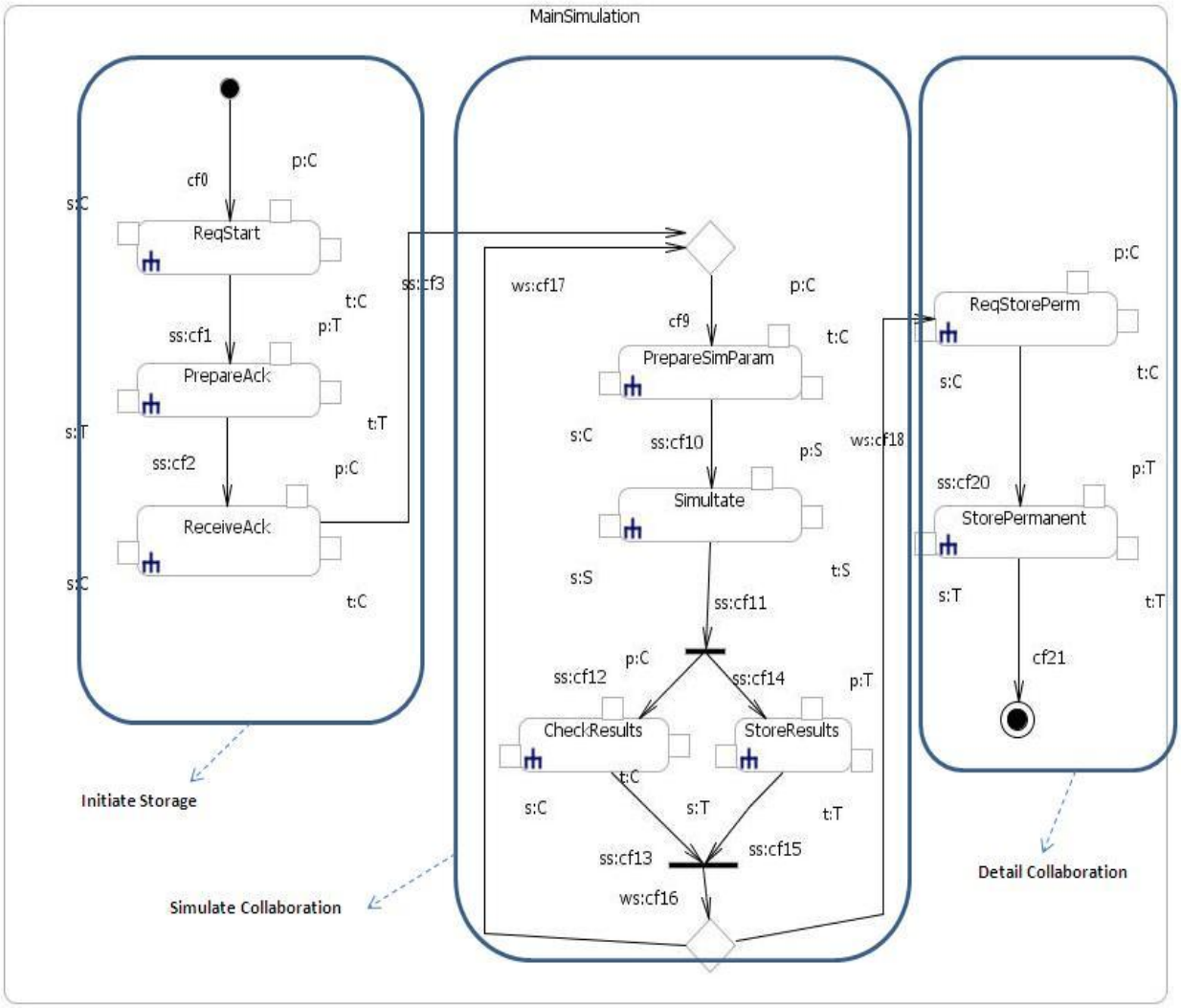


Figure 4: Client-Simulator-Storage global activity diagrams

2.2.1 Deriving the Individual Activity Diagrams from the Global Activity Diagram

To understand the behavior of the collaborative components, the local actions of each component is derived. A model transformation algorithm designed by [4] explains these derivation rules. The high-level description of a collaboration (as for instance shown in Figure 4) can be transformed into a distributed system design that contains a process for each component involved in the collaboration. Each process description defines the local behavior of the component in terms of local actions to be performed and messages to be sent to, and received from the other components. The transformation algorithm assures that the joint execution of these local component behaviors will give rise to a global system behavior that conforms to the high-level system specification. There are two main ideas that assure the correctness of this transformation algorithm: (1) the algorithm introduces certain so-called coordination messages that are exchanged between the different components in order to coordinate the order in which certain local actions are performed. (2) The messages received by a given component are first placed into a local message pool. When the local behavior of that component is ready to consume a specific message, it requests it from the message pool. Such a consumption request may specify one or several alternative message types, or message types with a specific parameter value. If no suitable message is in the buffer, the component's process will wait until one is received.

The implementation of the derivation algorithm to derive the behavior of each individual component from the global activity diagram was done by [5] using an Eclipse platform tool as illustrated in Figure 4. The first derivation step is to identify the collaborations. The second step is to define the starting, participating, and the terminating role(s) of each collaboration. The last step is to derive the coordination messages between the components. The automatically derived

local behavior of the *Client*, *Simulator*, and *Storage* components is shown in Figures 5, 6, and 7, respectively.

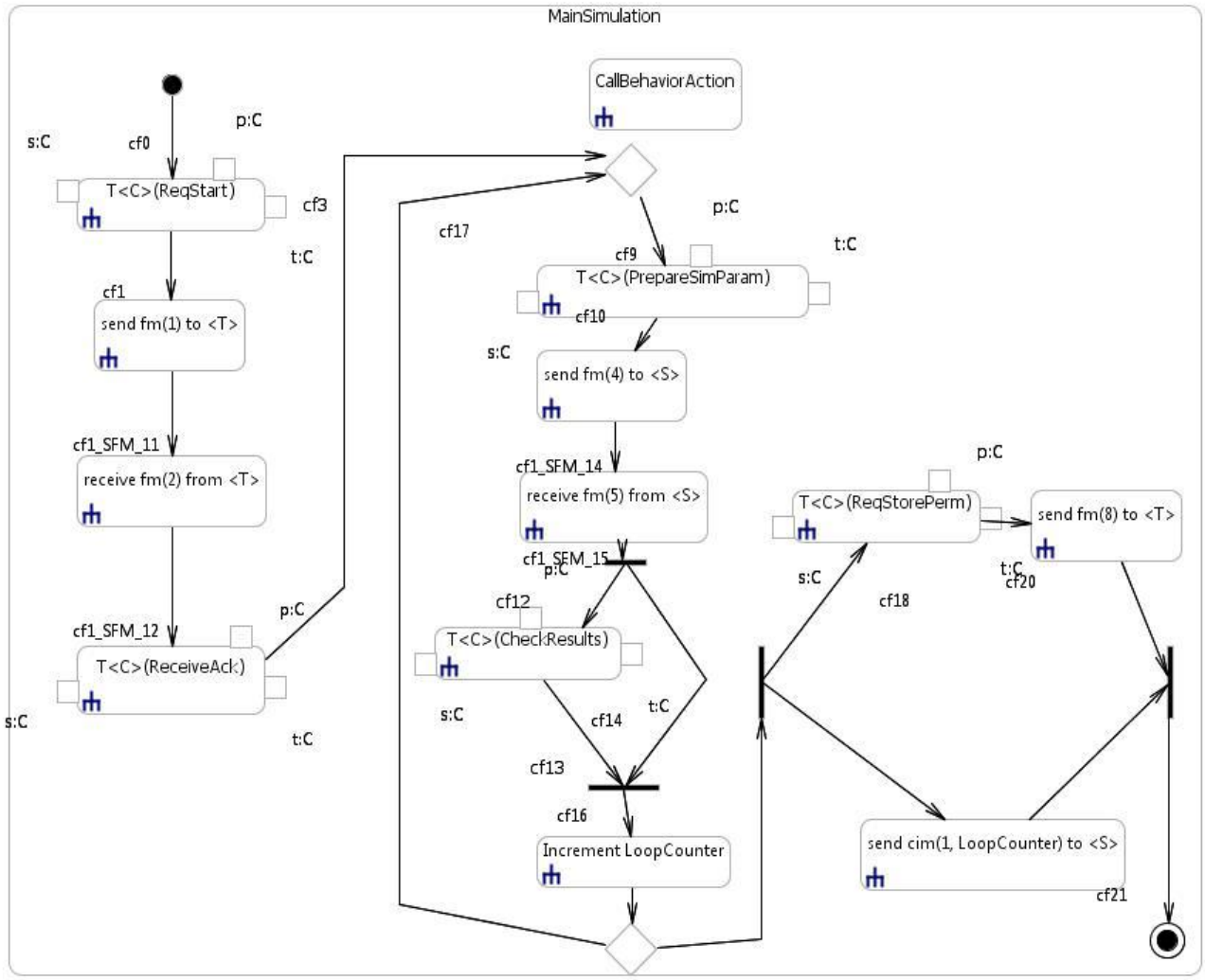


Figure 5: Client Activity Diagram

For example in Figure 5, the *ReqStart* local action is the same one that has been shown in the *initiation* collaboration in Figure 4. This action represents a preparation of the flow message fm (1). The *send fm (1) to<T>* represents a *Client* call (invocation) to the *Storage* activity. On the

other hand, the *Storage* activity in Figure 7 has a "*receive fm (1) from<c>* " local action that receives the *Client's* message. In addition, the *Storage* activity has the local action *prepareAck* that prepares an acknowledge message, called *fm (2),* and the "send *fm (2) to<C >* " local action represents a *Storage* invocation to the *Client*. The *Client* receives an *fm (2)* message through the "*receive fm (2) from<T>"* local action. It is important note that *fm (1)* and *fm (2)* are two different message types. Another example is the "*send fm (4) to<S>"* local action; it represents the *Client* invocation to the *Simulator* which sends asynchronously a parameter for simulation. This action involves two components, the *Client* and *Simulator*. The *Simulator* has the receive message *"fm (4) from<C>"* local action which receives an *fm (4)* message from the *Client*, as illustrated in Figure 6. Finally the concurrent actions *checkResults* and *storeResults* in Figure 4 appear separately in Figure 5 and 7; this is because *checkResult* is a local action performed by the *Client*, while the *storeResult* is a local action performed by the *Storage*. For this reason, the concurrent node of the *Simulator* activity in Figure 6 is empty.

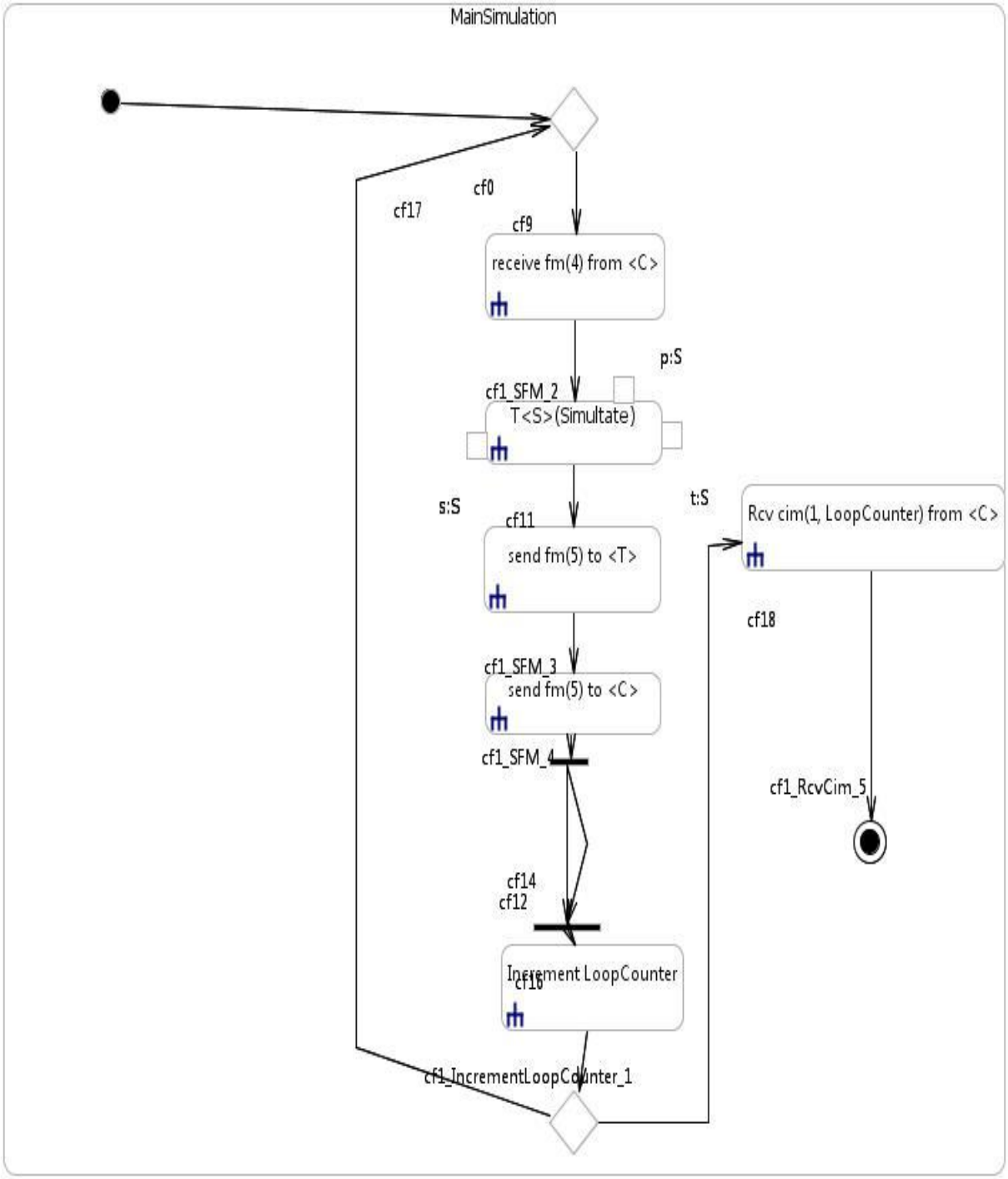


Figure 6: Simulator Activity Diagram

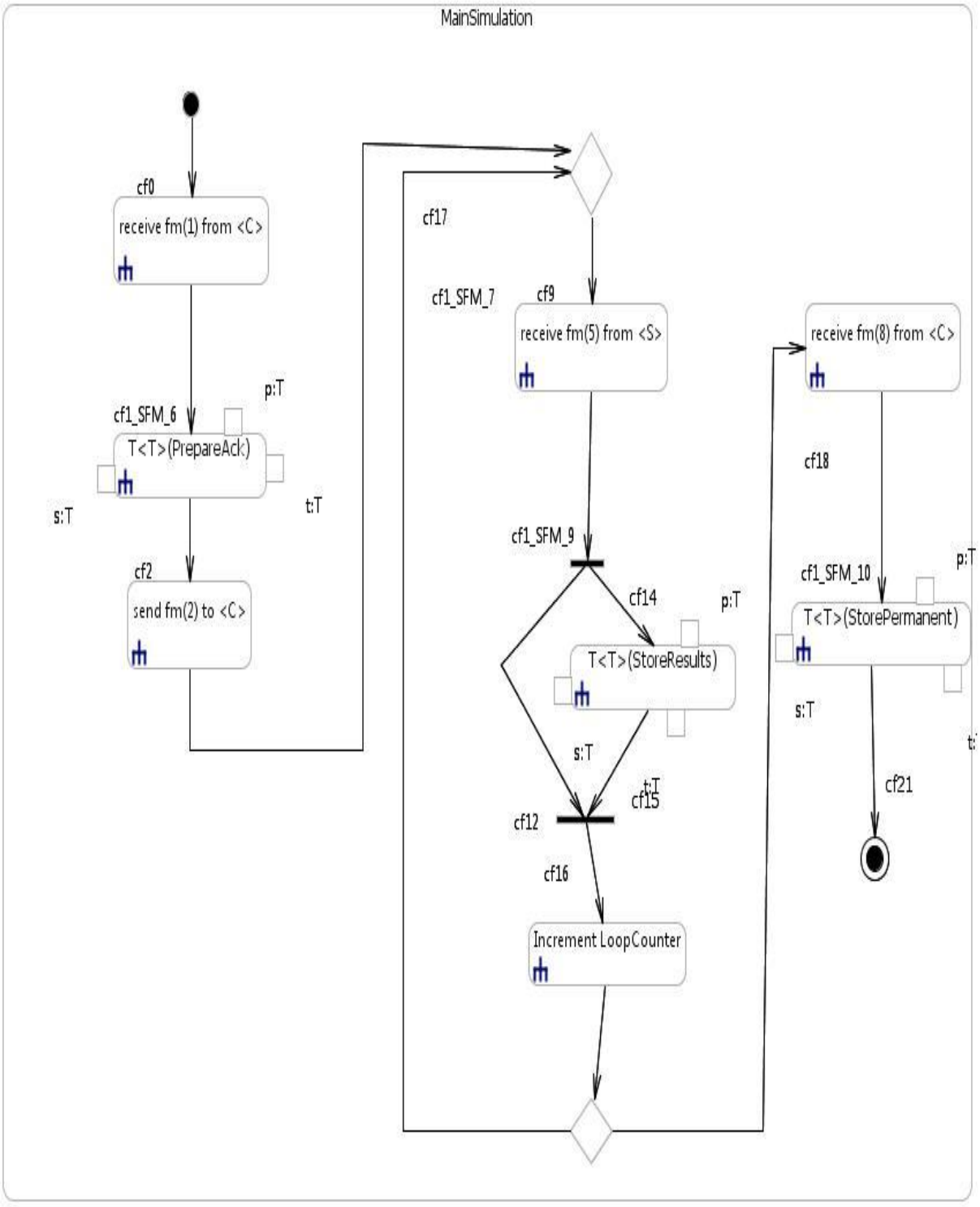


Figure 7: Storage Activity Diagram

Chapter 3

3 Basic Concepts and Notations

This chapter reviews some concepts that are important to understand the UML AD to BPEL transformation. First, it starts with an introduction to distributed systems. Second, it talks about Web Services (WS) and the relationship between WS and Service Oriented Architecture (SOA). Third, it describes business processes and business process modeling techniques. Finally, the chapter gives a review about BPEL technical specifications.

3.1 Distributed systems

Distributed systems architecture has emerged as many applications or entities need to interact with each other. Santoro articulated that distributed systems are characterized by the presence of network entities communicating with each other by means of messages, cooperating toward common tasks or the solution of a shared problem [17]. From this characterization, we can notice that any two components in a distributed system can communicate with one another. Furthermore, both components should use the same protocol to interact effectively. The emergence of Web Services has proposed a new way of communication between components. Each component is shown as a Web Service and has an interface that describes its functionality to other components and messages that are passed between components through these interfaces.

3.2 Web Services

A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format using the WSDL [9]. The interaction among these services is achieved through the use of the Simple Object Access Protocol (SOAP) to exchange messages. These messages are encoded in Extensible Markup Language (XML) and accessible by any application that has been designed to accept it. Furthermore, the messages can be transferred from one system to another via the Hyper Text Transfer Protocols (HTTP). Figure 8 illustrates assumed interactions among web services.

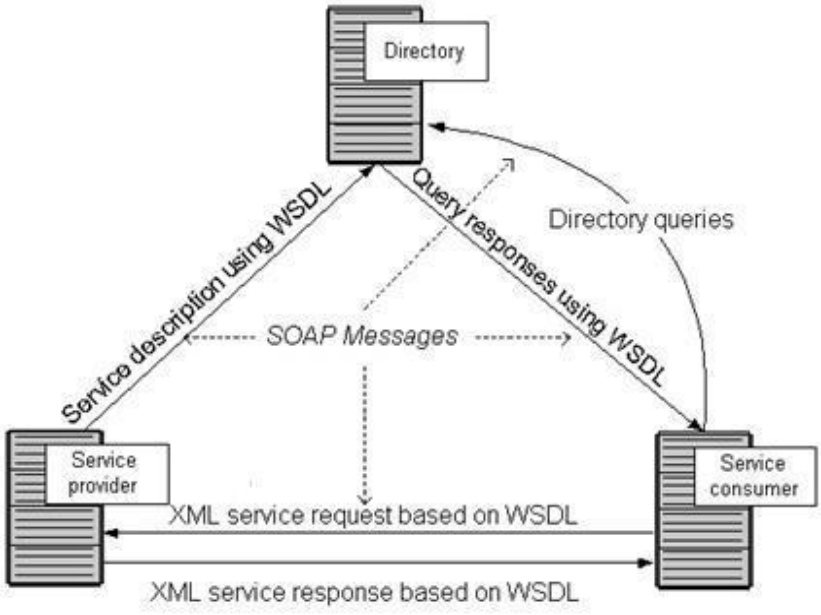


Figure 8: Web Services interactions (source [10])

3.2.1 Extensible Markup Language (XML)

Extensible Markup Language (XML) is a text-based language that describes a class of data objects called XML documents. XML is one of the most widely-used formats for sharing structured information among programs. To understand the structure of XML we should clarify the following concepts [11]:

- XML document (Data Object): an XML document has logical and physical structure. An XML document is composed of declarations, elements, comments, character references, and processing instructions. Its logical structure is composed of units, called entities, which may have attributes.
- XML namespaces: They are used to uniquely identify named elements and attributes in an XML document. In order to reduce the conflict between names a prefix can be used before the name. In XML, the default namespace is defined by using the *xmlns* attribute in the start tag of an element.
- Characters: a character is an atomic unit of text as specified by International Organization for Standardization (ISO/IEC 10646). Legal characters are tab, carriage return, line feed, and the legal graphic characters of Unicode or ISO/IEC 10646.
- Markup: A markup is a series of characters in XML document. It can be distinguished from text because it always begins either with the character < (in which case it ends with the character >) or the character **&** (in which case it ends with the character **;**). Examples of markup are start-tags, end-tags, empty-element tags, entity references, character references, and document type declarations.
- Character data: a text other than markup.

- Entities: They are part of XML document identified by a n*ame and have co*ntents. Each XML document has a unique entity called *document entity* which serves as the root of the entity tree and a starting-point for the XML processor.
- XML processor: It is a software module that is used to read XML documents and provide access to their content and structure.
- Parsed data: This term refers to the entities that are processed or read by the XML processor. A parsed entity contains a sequence of characters, called text. The unparsed entities are data other than XML text and not processed by the XML processor.

3.2.2 Asynchronous vs. Synchronous Message Communication

Messages exchange plays a vital role among participating services to achieve business functionalities and commit transactions. The participating components such as client and service can communicate with each other via synchronous or asynchronous messages. Using the synchronous message mode, the client sends the message to the service and waits for its response. In other words, the client has to wait in a blocking state for the service to finish servicing its requests. This type of communication is mandatory in some applications such as banking systems, where a branch blocks check cashing operation until an account balance is retrieved from a centralized database.[27]. In asynchronous message mode, sometimes called Fire-and-Forget, the client does not have to wait for a response from the service. For example, a client sends a change of billing address to a service. By employing this type of message communication, it is assumed the messaging infrastructure ensures delivery even if the service is temporarily offline, busy, or unobtainable.

3.2.3 Web Services Description Language (WDSL)

“The Web Services Description Language (WSDL) provides a model and an XML format for describing Web services. WSDL enables one to separate the description of the abstract functionality offered by a service from concrete details of a service description such as “how” and “where” that functionality is offered” [7]. This definition implies that a service description involves abstract and concrete descriptions. The abstract description defines the interfaces of the WS in terms of the messages it sends and receives. This is independent of the connection type, usually messages are associated with message types and described using XML schema. On the other hand, the concrete description specifies the *binding* format and the protocols used in this binding. In WSDL, binding describes how the service is bound to a messaging protocol, particularly the SOAP messaging protocol. A WSDL document includes these descriptions and contains the following elements:

- Type: a container that defines the data types used in the description of the service using XSD. XSD is used in XML to describe message formats to ensure a mutual understanding of the content between senders and receivers. For example, the XML data type "date" requires the format "YYYY-MM-DD". The code segment indicated by label 2 in Figure 9 illustrates an example of an XSD type.

- Message: represents an abstract, typed definition of the data being communicated. Label 3 of Figure 9 shows the *ReceiveFromManagerRequest* message example.
- Operation: a description of an action supported by the service. An example of an operation is shown in Figure 9, label 5.

- Port: refers to a single endpoint defined as a combination of a binding and a network address.
- Port type: represents an abstract set of operations supported by one or more endpoints. This is shown in Figure 9, label 4.
- Binding: it corresponds to a concrete protocol and data format specification for a particular port type.
- Service: a collection of related endpoints.

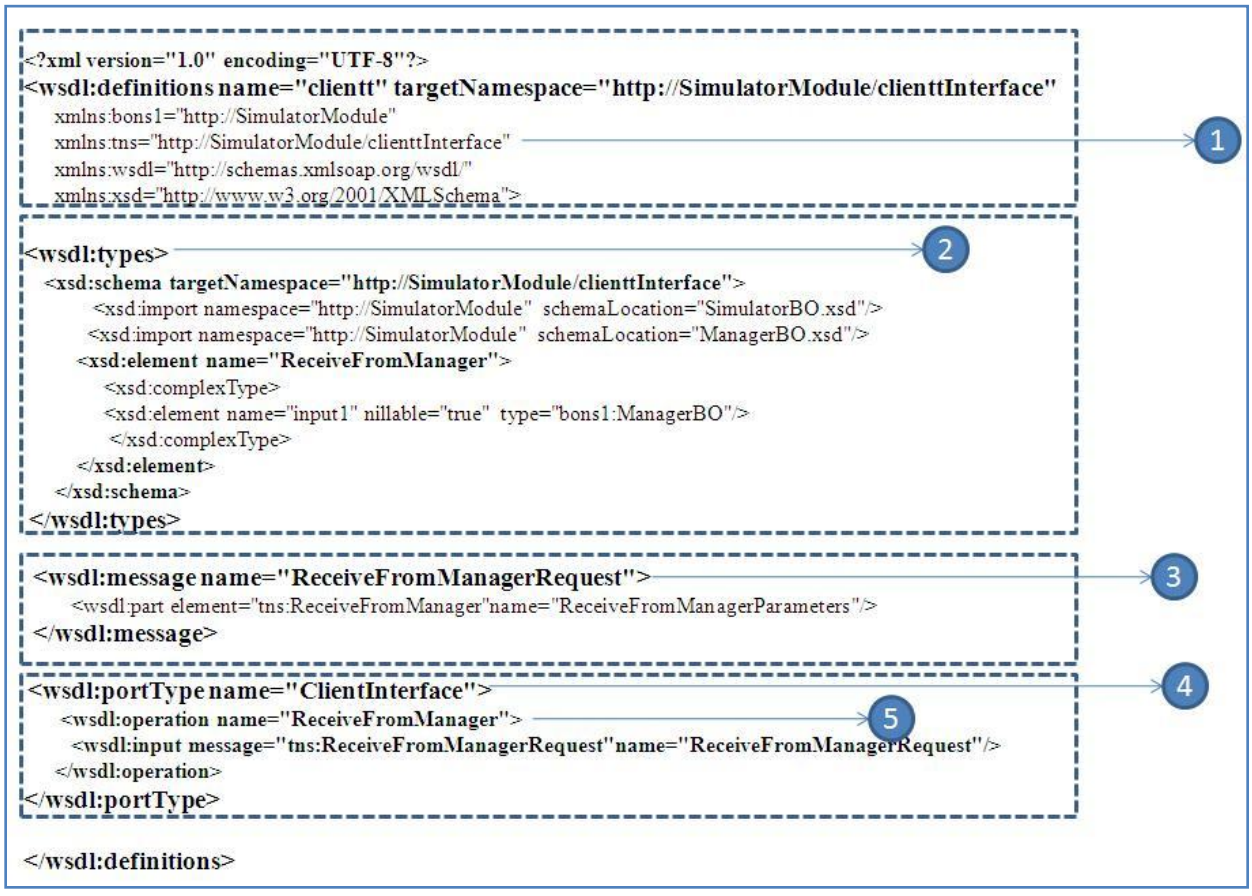


Figure 9: WSDL document

3.2.4 Simple Object Access Protocol (SOAP)

This protocol specifies the format of the XML messages, and how the message elements should be manipulated. A SOAP message has an envelope that identifies the XML object as a SOAP message, a header which contains header information, a body element that contains call and response information, and a fault element which contains errors and status information.

3.2.5 Universal Description, Discovery and Integration (UDDI)

It is a standard that describes how a business can register its service in a global registry, and search that registry for an interesting service.

3.3 Service Oriented Architecture (SOA)

SOA provides design architecture for loosely-integrated services that can be used among business domains to carry out business transactions. With the introduction of SOA, legacy software systems are made available for users as web services. These services form a distributed software system, and need not to be present on the same computer, the service could be (physically) located anywhere. Services in SOA represent the service provider, the service consumer, and the service registry. The Service provider is a software entity that implements a service specification, and can be dynamically located and invoked. The service consumer is another software entity that calls a service provider. Traditionally, this is termed a "client"; however, a service consumer can be an end-user application or another service. The service registry is another component that provides a look-up mechanism where service clients can find a service provider based on some criteria. For example, to locate a shipping service, consumers need to ask the directory service to find a list of service providers to ship a product [13].

3.4 Business Processes and Process Modeling

A business process is a collection of activities which collectively realize a business objective or policy goal [14]. A business process puts emphasis on how the work is done within an organization by showing the logical ordering of work activities, and defining the goal, inputs and outputs, and the start and end of a business process. The representation of a business process in a form that supports automated manipulation is called business process model. Many techniques have been used to model business processes, among the modern methods are Activity Diagrams (AD) of the Unified Modeling Language (UML) and the Business Process Modeling Notation (BPMN). Both techniques use graphical notations and have been developed by the Object Management Group (OMG) [15]. BPMN uses a Business Process Diagram, which is based on flowcharting techniques to model business processes that can be mapped into the Business Process Execution Language (BPEL), while activity diagrams use a similar notation which also can be transformed into BPEL for execution. The study of BPMN is beyond the scope of this thesis, so we will concentrate only on the use of UML activity diagrams in modeling business processes, and the transformation of activity diagrams to BPEL.

3.4.1 Business Process Execution Language (BPEL)

BPEL is an XML-based language used to define enterprise business processes within Web services. It is based on web services in the sense that each business process is assumed to be implemented as a web service. There are many extensions for web services specifications, among these is the Business Process Execution Language for Web Services (WS-BPEL or BPEL4WS or simply BPEL) [16]. The first version of BPEL was developed in August 2002 by BEA, IBM, and Microsoft. Since then, the majority of vendors have joined which has resulted in

several modifications and improvements and adoption of Version 1.1 in March 2003. In April 2003, BPEL was submitted to Organization for the Advancement of Structured Information Standards (OASIS). BPEL received wide support from major software vendors such as Oracle, Microsoft, IBM, HP, and Siebel [26].

3.4.2 BPEL Technical Specifications

BPEL popularity increased with the introduction of Service Oriented Architecture (SOA) which requires a simple language to describe the interactions among web services using branching and concurrency to form a business process (Abstract business process). BPEL also can be employed to write the exact specifications of how to execute such business processes (Executable business process). To understand BPEL processes, it is important to understand some BPEL concepts such as partner links, port type, process life cycle and BPEL activities. We have to emphasize that only a subset of BPEL specifications are discussed here, a detailed specification can be found on the OASIS website [16].

3.4.2.1 Partner Link

The <PartnerLink> defines the services that a business process will interact with through the use of a <partnerRole>. At the same time it specifies how other web services clients interact with the business process using<myRole>. A partner link should have a PartnerLinkType and a name, a PartnerLinkType could be assigned to more than one partner link. In the example in Figure 10, the *StorageInterface* is defined as an interface for the business process *Storage*, the *ClientInterfacePartner* and the *SimulatorInterfacePartner* are two partnerLinks that define the *Client* and *Simulator* respectively. The *Storage* process interacts with the *Client* and *Simulator* through theses partnerLinks.

```
<bpws:partnerLinks>
<bpws:partnerLink myRole="StorageInterfaceRole" name="StorageInterface"
        partnerLinkType="ns:StorageInterfacePLT"/>
<bpws:partnerLink name="ClientInterfacePartner" partnerLinkType="ns:ClientInterfacePLT"
        partnerRole="ClientInterfaceRole"/>
<bpws:partnerLink name="SimulatorInterfacePartner"
        partnerLinkType="ns:SimulatorInterfacePLT" partnerRole="SimulatorInterfaceRole"/>
</bpws:partnerLinks>
```

Figure 10: Partner Link elements

3.4.2.2 Port Type

The <*portType*> is the definition of an interface to the web service. Usually the interface includes the message types, inputs and output messages of a service. Inputs and outputs here represent SOAP messages that are received or sent and which are specified using the WSDL notations. Figure 11 shows a WSDL definition (*portType*) for the *Client* business process, the code segment shows the definition of the port type indicated by label 1, the operation(message type) definition in label 2, and the input message indicated by label 3. Note in this example there is no output because messages are sent asynchronously.

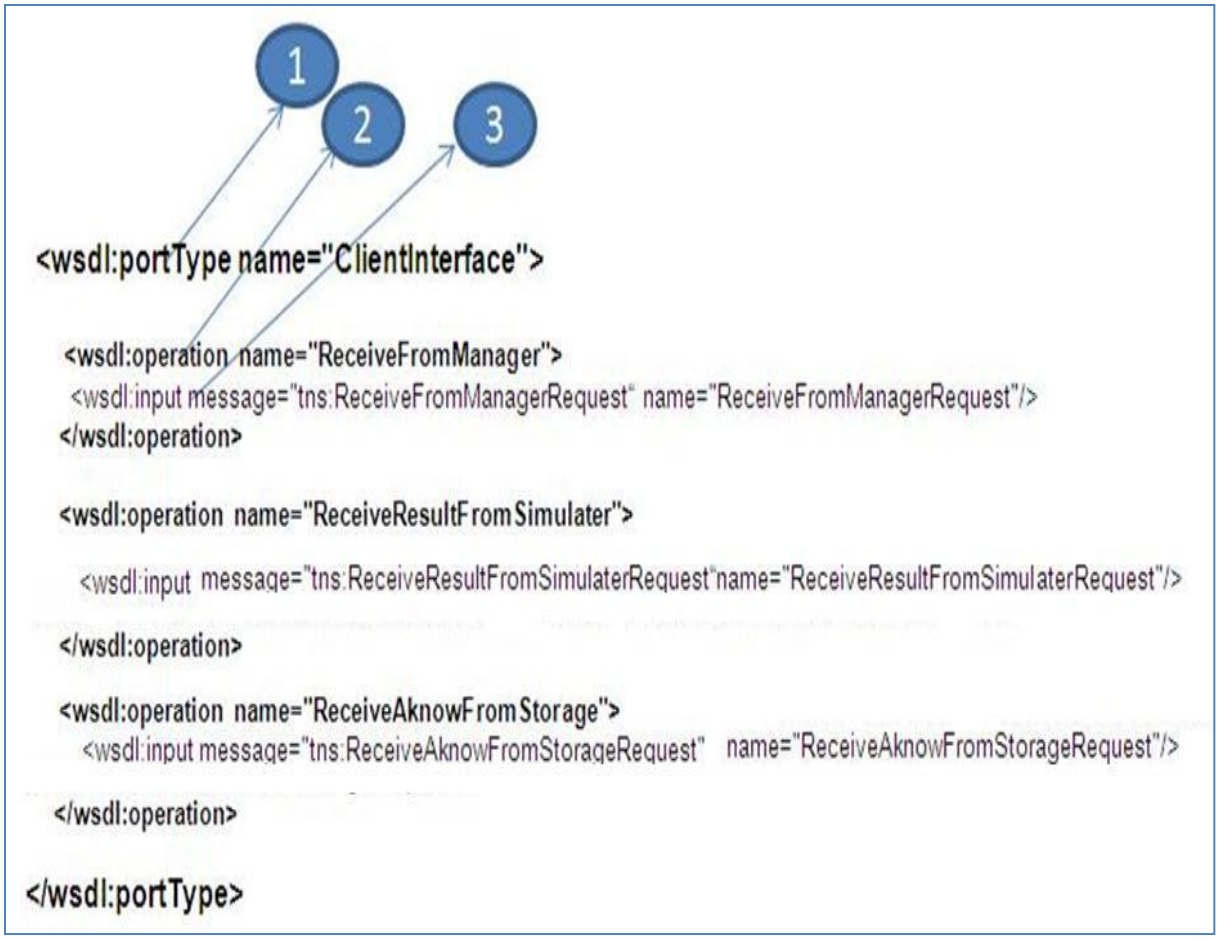


Figure 11: Port Type elements

3.4.2.3 Process Life Cycle

A BPEL process instance is created when the server receives the first message which will be consumed by a <*Receive*> or <*Pick*> activity. The <*Receive*> activity is able to create a process instance when the *createInstance* attribute is set to *yes*. If a process starts with a <*pick*> activity, then the *createInstance* of each <*onMessage*> element should be set to *yes.* To create more than one instance of the same process when several messages are received, a correlation set must be used.

3.4.2.4 Correlation Sets

A Correlation set is a set of variables whose values uniquely identify a process instance. A correlation set can be used in *Invoke, Receive, Pick,* and *Reply* BPEL activities. When a message is received by a BPEL process, that message must be delivered either to a new or an existing instance of that process. The task of determining the instance is what the correlation is all about [18]. It is mandatory to have a correlation set if a process contains more than one receive or pick activity. All receive activities and <*onMessage*> that belong to the same process instance must have the same correlation parameter value. Therefore, the BPEL execution engine checks any arrived message to see if the correlation parameter value matches an existing instance correlation parameter, if so, the message is dispatched to that process instance, otherwise the server creates a new process instance based on the correlation parameter considering the fact that it is the first message. A possible case of race condition might occur when a correlation set is used with different <*receive*> activities in a process. To illustrate the race condition, consider multiple <receive> activities correlated to the same process instance (i.e. having the same correlation parameter value). At runtime it is possible for a message that does not initiate the process instance to arrive before the one that should initiate the process instance; this is a case of a race condition. For example, in the sequence diagram in Figure 12 message B is allowed to create an instance of the *storage* process while message *C* cannot; if message C arrives at the *storage* before message B, a race condition occurs. Processes engines may employ different mechanisms to handle such race conditions [16].

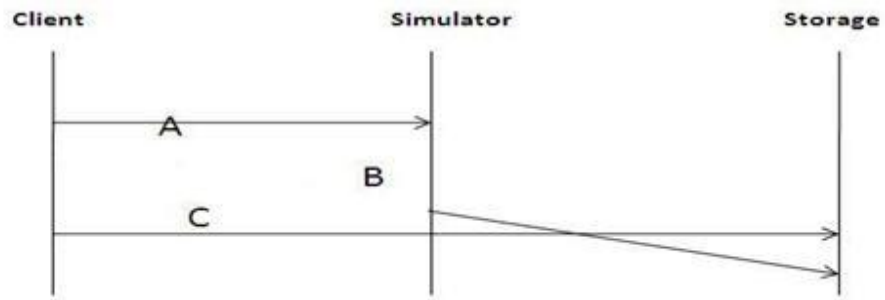


Figure 12: Possible race condition between messages B and C

3.4.3 BPEL Activities

WS-BPEL activities perform the process logic. Activities may be basic or structured. Basic activities express the fundamental steps of the process behavior, on the other hand, structured activities describe control-flow logic. Therefore, a structured activity might contain other basic and/or structured activities, recursively (19]. An explanation of some of these activities follows.

3.4.3.1 Receive Activity

Receive is one of the activities that offer interaction with the outside world. A business process offers services to its partners by <receive> and matching <reply> activities. A <*receive*> activity specifies a <*partnerLink*>, a <*portType*>, an invoked message type, and an input and output messages. Figure 13 illustrates a graphical view for the *ReceiveStartFromClient activity*, this activity receives the message parameter *start* on the port *StorageInterface* and stores it in the *storageactivity_Start* variable.

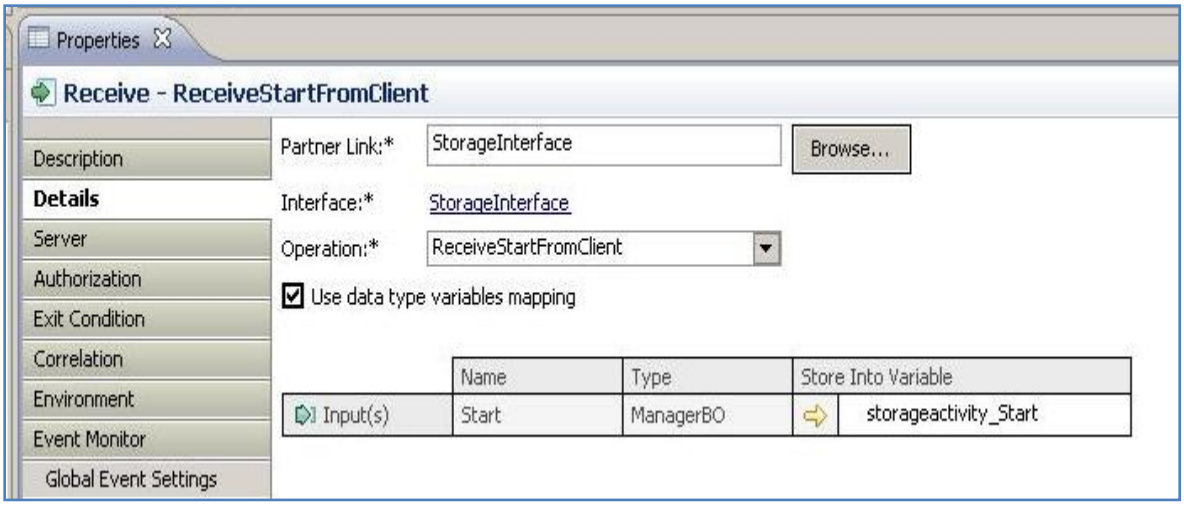


Figure 13: Receive activity view in the IBM WebSphere Integration Developer

The receive activity plays an important role in the life cycle of a business process. One of the ways to initiate a process is by setting the *createInstance* property of the first *receive* activity to *yes*. If multiple receive activities are allowed to initiate a process, then the *createInstance* property of each one is set to *yes*. In this case, the first activity that is called will initiate the process. A process should not contain two receive activities that have the same *partnerLink*, *portType*, *operation(message type)* and *correlationSet*. A BPEL source code of the <*receive*> activity of Figure 13 is shown in Figure 14. The figure shows the <*receive*> activity *ReceiveStartFromClient*. The *createInstance* of this activity is set to *yes* which indicates that the activity can start a process instance. Notice also that a correlation set is used here, this is an indication that this process contains more than one <*receive*> activity, the *initiate* property of the correlation set is set to *yes* which implies that this is the first <*receive*> activity in the process.

```
<bpws:receive createInstance="yes" name="ReceiveStartFromClient"    operation="ReceiveStartFromClient"
   partnerLink="StorageInterface"
   portType="ns0:StorageInterface"wpc:displayName="ReceiveStartFromClient"wpc:id="45"wpc:transactionalBehavior="commitAfter">
     <wpc:output>
       <wpc:parameter name="Start" variable="storageactivity_Start"/>
     </wpc:output>

     <bpws:correlations>
      <bpws:correlation initiate="yes" set="StorageCorrelationSet"/>
     </bpws:correlations>
</bpws:receive>
```

Figure 14: BPEL code of receive activity

3.4.3.2 Invoke Activity

A business process uses an <*invoke*> activity to invoke a one-way or a two-way (request-response) message type on a *portType* offered by a partner. In case of two-way message type, the <*invoke*> activity completes when the reply (response) is received. In case of one-way message type, the <*invoke*> activity completes when the requested message has been sent. The input and output messages must be specified within the *invoke* activity, while the *portType* specification is optional. Figure 15 depicts an <*invoke*> activity *SendStartMsgToStorage* that calls the message type *"ReceiveStartFromClient"* on a"*StorageInterface"* portType.

```
<bpws:invoke name="SendStartMsgToStorage" operation="ReceiveStartFromClient"
   partnerLink="StorageInterfacePartner" portType="ns2:StorageInterface" wpc:id="10">
     <wpc:input>
       <wpc:parameter name="Start" variable="clientactivity_Start"/>
     </wpc:input>
 </bpws:invoke>
```

Figure 15: BPEL code segment of Invoke activity

3.4.3.3 Switch Activity

The <*Switch*> activity represents conditional branching where the programmer can specify one or more *Case* statements. Each *Case* statement represents an activity branch. If no *Case* statement is satisfied, an *"otherwise"* statement is executed. The code segment in Figure 16 shows a <*switch*> activity that has a *case* and an *otherwise* branch, whenever the condition "*Final.intValue() == Last.intValue*()" is *true*, the case *"ContinueLoop = newjava.lang.Boolean(false)"* will be executed, otherwise, the "*ContinueLoop = new java.lang.Boolean(true)"* will be executed.

```
<bpws:switch name="Switch" wpc:displayName="Switch" wpc:id="57">
    <bpws:case wpc:id="58">
        <bpws:condition>
        <![CDATA[boolean __result__1 = Final.intValue() == Last.intValue()
          return __result__1;
          <wpc:displayName="ContinueLoop = false" wpc:id="63">
          <wpc:javaCode><![CDATA[ContinueLoop = newjava.lang.Boolean(false);]]>
          </wpc:javaCode>
    </bpws:case>
    <bpws:otherwise>
          < wpc:displayName="ContinueLoop = true" wpc:id="68">
          <wpc:javaCode><![CDATA[ContinueLoop = new java.lang.Boolean(true);]]>
          </wpc:javaCode>
    </bpws:otherwise>
</bpws:switch>
```

Figure 16: BPEL Switch Activity

3.4.3.4 While Activity

The <*while*> activity represents a loop where the body of the loop has one or more statements. Usually, the while loop contains a decision node that decides whether another execution of the body should proceed, or whether the loop terminates. If the condition in the decision node evaluates to *false,* the loop terminates and the statement following the body of the loop is executed. In BPEL, a Boolean variable is initialized to *true* before the <*while*> activity starts, when the while loop is executed, the condition is checked to decide whether to continue or terminate.

The code segment in Figure 17 exemplifies a loop node. Notice that the loop iterates as long as the condition "*ContinueLoop.booleanValue() != false"*. In this example, there is a switch with <*case*> and <*otherwise*> branches inside the loop.

```
<bpws:while name="LoopNode" wpc:id="10">
    <bpws:condition><![CDATA[boolean __result__1 = ContinueLoop.booleanValue() !=false;
    return __result__1;
    </bpws:condition>

   <bpws:switch name="Switch" wpc:displayName="Switch" wpc:id="57">
        <bpws:case wpc:id="58">
          ------------------------- Some code here

        </bpws:case>
        <bpws:otherwise>

         ------------------------- Some code here
        </bpws:otherwise>
   </bpws:switch>
</bpws:while>
```

Figure 17: BPEL While activity

3.4.3.5 Pick Activity

The <*pick*> activity is a structured activity that represents selective event processing where multiple message receptions are defined. The <*pick*> activity waits for the occurrence of exactly one event from the defined events, then executes the activity associated with that event. The selection of the activity depends on which event comes first. If a certain event is selected, the other ones will be stored in a queue but not consumed by the <*pick*> activity unless there is a loop that returns back to the <*pick*> activity.

Each <*pick*> must include at least one <*onMessage*> similar to a <*receive*> activity which waits for the reception of an inbound message. In addition, the <*pick*> activity may have <*onAlarm*> events that correspond to timer-based alarms. If a specified duration value has already been reached, then the <*onAlarm*> event is executed. If a <*pick*> activity is the starting activity in a process, a new instance of the business process is to be created upon the receipt of an <*onMessage*> event, the *createInstance* property of the <*pick*> must be set to *yes*. The example in Figure 18 shows a code segment for a <*pick*> activity that belongs to the *Storage* process. The code illustrates a <*pick*> with two alternatives <*onMessage*> actions the *ReceiveLastVersionFromClien*t and *ReceiveResultFromSimulater*. We notice that the <*pick*> activity does not start a process instance because the *createInstance* is set to *no*.

```
<bpws:pick createInstance="no" name="Pick" wpc:displayName="Pick" wpc:id="46"
          wpc:transactionalBehavior="participates">

    <bpws:onMessage operation="ReceiveLastVersionFromClient"partnerLink=
            "StorageInterface" portType = "ns0:StorageInterface">

    </bpws:onMessage>
    <bpws:onMessage operation="ReceiveResultFromSimulater" partnerLink=
            "StorageInterface" portType= "ns0:StorageInterface">

     </bpws:onMessage>

</bpws:pick>
```

Figure 18: BPEL Pick activity

Chapter 4

4 Transforming an AD into a BPEL Process

4.1 Literature Review

There has been a considerable amount of work on the transformation from UML or the Business Process Modeling Notation (BPMN) to BPEL. Most of the previous work considers that the generated BPEL process communicates with other Web Services through synchronous method invocations for the performance of certain actions. In contrast, we consider in this research the situation where several BPEL processes running in different servers collaborate by asynchronous message passing. Using .asynchronous messages allows concurrent message sending and reception,

BPEL is a kind of programming language, encoded in XML, intended for defining programs that coordinate the execution of Web Services that are provided on different servers, possibly by different organizations. As mentioned previously, 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 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 by (20] for generating BPEL processes from UML-2 system models. [21] 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. [28] introduces a UML-2 extension for SOA, called the UML4SOA profile, for modeling service orchestrations. The authors defined a 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 [22].

Our research focucses in transforming activity diagrams into BPEL processes using the IBM Rational Software Architect (RSA).

4.2 Development Tools for Web Services-RSA

IBM offers a wide range of software tools that can be used by software engineers during system development. These tasks include requirements gathering, system analysis, design and implementation. Table 1 summarizes these tools and the type of tasks that they can be used for.

Role	**Task**	**Tools**
Business Executive	Convey business goals and objectives	IBM Rational Request Pro
Business Analyst	Analyze business requirements	IBM WebSphere Business Modeler
Software Architect	Design the architecture of the solution	IBM Rational Software Architect
Web Services Developer	Implement the solution	IBM Rational Application Developer & IBM Integration Developer

Table 1: Development process Role, Task, and Tools source [23]

The IBM RSA is a modeling and development application made by IBM's Rational Software division. The application is built on Eclipse open-source software framework and supports the use of UML for creating portable applications and web services. RSA support model-to-code and code-to-model transformations. It has the capability to make forward transformations from UML to other languages such as Java, C#, C++, structured Query Language (SQL), and WSDL, and reverse transformations from Java and C++ to UML. The application has the ability to build a software architecture that supports common platforms and easy synchronization of models and code. Moreover, RSA can accelerate the implementation maintenance of service-oriented architecture (SOA) solutions such as a web service and BPEL processes. Many versions of RSA have been released, such as v7.0 in 2006, and the latest version v8.0 in August 2010 [24]

4.2.1 Creating models in RSA

RSA has the capability of creating many models types depending on the application requirements and user needs. We will illustrate here the use of RSA in creating UML models. RSA uses the UML models to describe a system at abstract levels. UML models in RSA show the visual representation of a system. Models can have elements, such as components, classes, packages, and one or more diagrams that show a specific perspective of a system. When the RSA explorer is started, it instructs the user to the type of project model he or she is going to create. The contents of a modeling project are organized into diagrams and models. This structure displays the logical containment of the UML model elements, regardless of where they are stored physically. The models are listed under the *Models* node, and the diagrams are listed under the

Diagrams node. RSA can create, import, or export many types of UML diagrams such as structure diagrams, use-case diagrams, class diagrams, and activity diagrams. The activity diagrams in a UML model are used to capture system behavior. RSA can be used to generate implementation code from models. Figure 19 shows a snapshot of RSA project explorer.

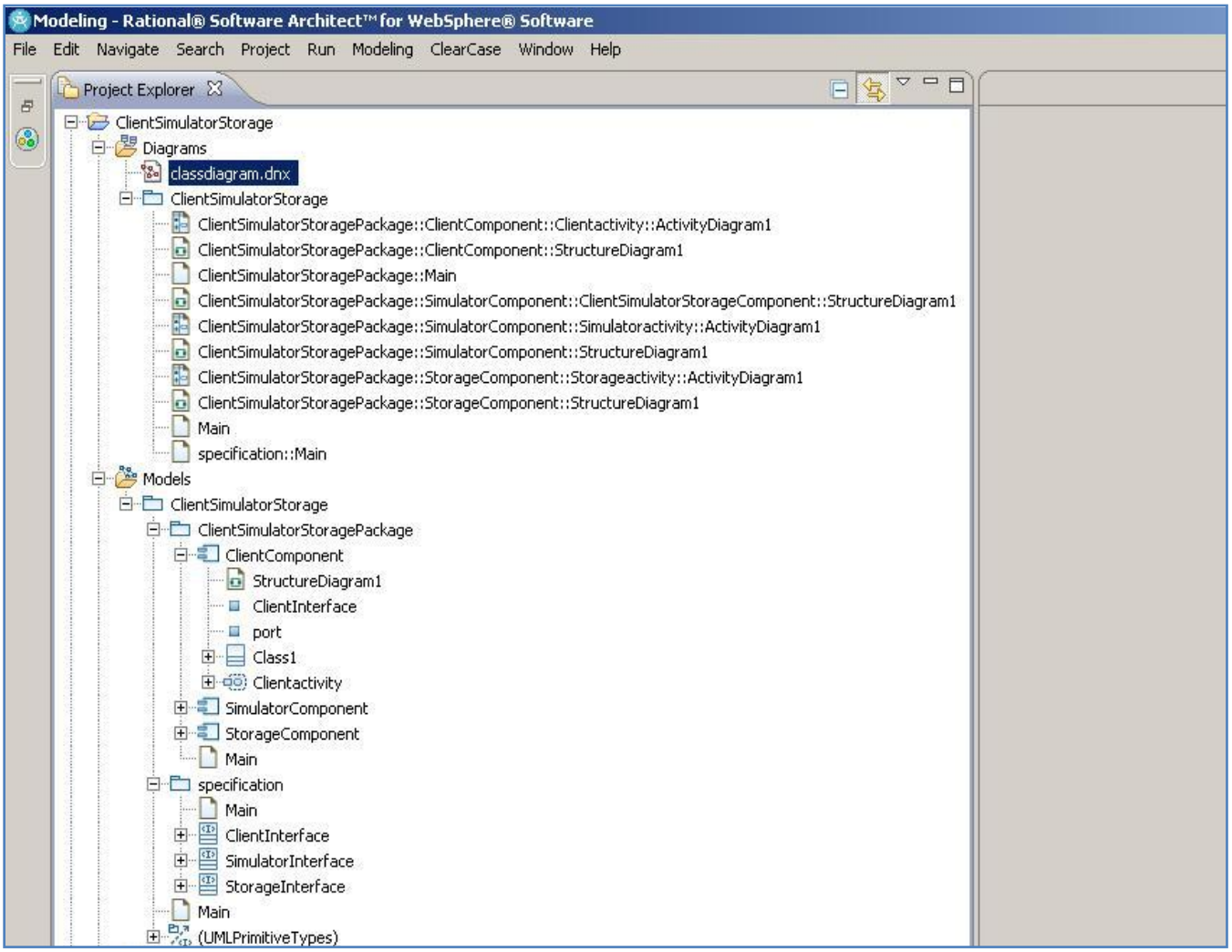


Figure 19: Project Explorer screen in IBM RSA

4.3 The Client-Simulator-Storage Case Study

The *client-simulator-storage* case study that was presented in Chapter 2 will be used here to show how to transform activity diagrams into BPEL processes. As described earlier, this case study contains three components that collaborate to define and simulate certain data, and store the simulation results. It describes the necessary modifications to the activity diagrams related to these components, and explains the necessary steps to perform the AD to BPEL transformation.

4.3.1 Additions to Activity diagrams

In order to automatically generate a BPEL process from an AD using the RSA tool, the AD input to the tool must have the right properties. The AD diagrams of Figures 5, 6, and 7 must be complemented with additional modeling elements before the automatic transformation can be performed. The followings are the additional required elements:

- **Input pin**: It represents an input parameter of an action. The input pin holds the input values that are received from other actions. For example, label 9 in Figure 20 indicates an input pin of type *Boolean.*
- **Output pin:** It represents an output parameter of an action. The output pin holds the output value that an action produces. Object flow edges deliver the output values to other actions. Label 14 in Figure 20 shows an output pin of type *Boolean.*
- **Loop validation variables**: Variables that define the number of loop iterations, and the conditions needed to continue or terminate the loop.

- **Activity Interface**: Defines the message types of an activity, each received message type should have a name, input message parameters. The type of each message parameter should also be specified.
- **Data types:** Specify the data types of all messages and variables needed to perform activity actions. These could be either primitive such as *integer* or complex data types such as *class type*.
- **Decision node conditions**: Represent expressions that evaluate to *true* or *false* and determine which branch of the decision node should be followed. Label 6 in Figure 20 shows the decision node that has *Final= = Last.*

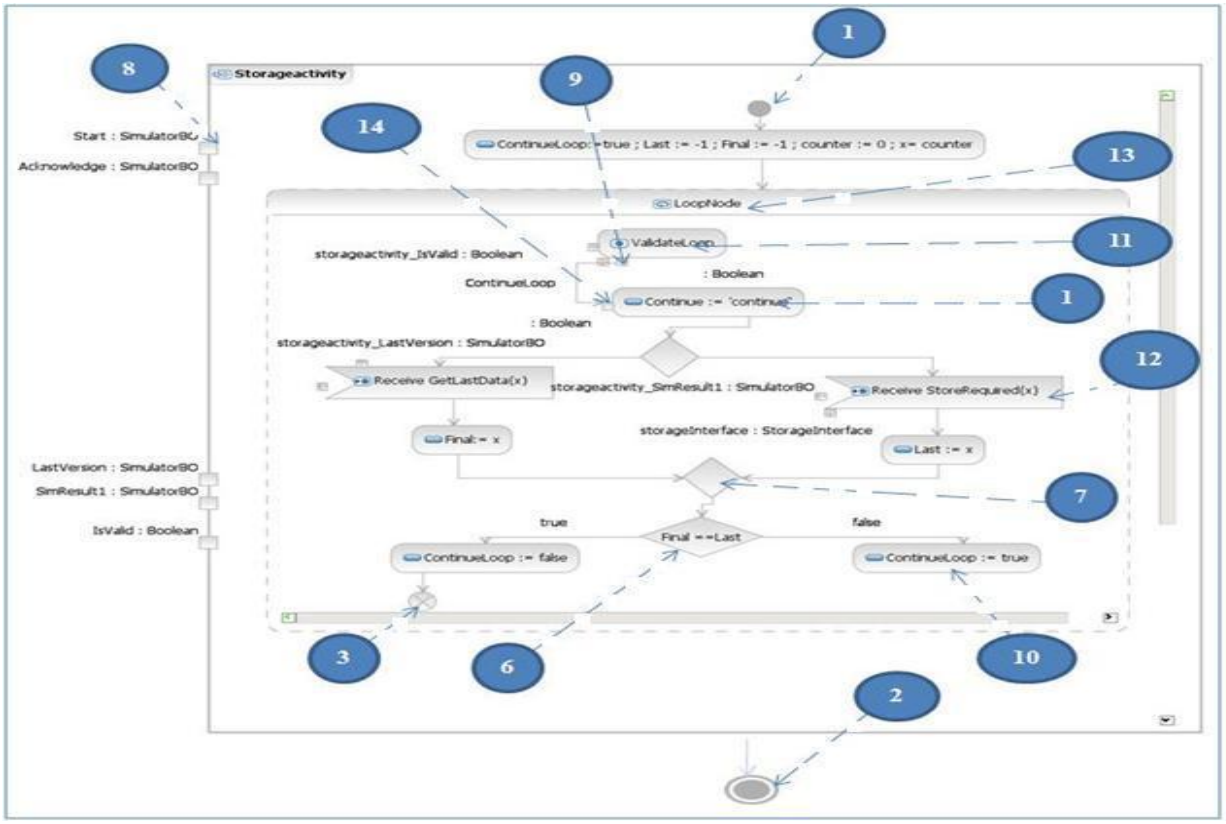


Figure 20: Storage Activity diagram

4.3.2 Modifications of Activity Diagrams in the RSA Tool

The Activity diagrams in Figures 4, 6, and 7 are produced by the Eclipse tool. The RSA tool could not recognize certain elements in these diagrams. The following elements are modified to adapt to the notations for activity diagrams used in the RSA tool.

1- **Receive action:** The receive action notation is changed in the RSA tool as shown by label 12 in Figure 20. Moreover, each receive action activity is linked to a message type that is specified by some activity interface.

2- **Loop:** The loop notation in the Eclipse tool cannot be recognized by the RSA tool. The loop in RSA looks like a structured activity that encompasses all actions to be performed within the loop body as indicated by label 13 in Figure 20. In addition, a validation action is added inside the loop to validate the loop iteration as indicated by label 11 of Figure 20.

3- **Activity Parameters:** In RSA, the variables that store input and output messages are represented by activity parameters. Each activity parameter is added to the border of the activity and represented by a name and type. For example, the *Start* message indicated by label 8 in Figure 20 represent an input message of type SimulatorBO class; this message has a value of type *string* and a correlation value of type *integer*.

4.3.3 Guidelines for the Transformation of Activity Diagrams into BPEL

Before performing the automatic AD-to-BPEL transformation, the following steps should be followed:

1- Services Identification

The *client-simulator-storage* application has three components: the *client*, *storage*, and the *simulator*. Each component has an interface that defines the set of message types received by that component. In addition, each message type has message parameters. We identified the following interfaces. They have to be defined using UML Class diagrams using the "interface" stereotype.

- **Client Interface:** defines the messages received by the *client* component. It has the *ReceiveResultFromSimulator(),ReceiveAcknowFromStorage(), ReceiveLastVersionFromStorge* and *ReceiveStartFromManager* message types.
- **Storage Interface:** defines the messages received by the *storage* component which include *ReceiveResultFromSimulater()* ,and *ReceiveStartFromClient()* message types.
- **Simulator Interface:** defines the set of messages received by the *simulator* component, it has only the *ReceiveParamFromClient()* message type.

The class diagram in Figure 21 illustrates these interfaces.

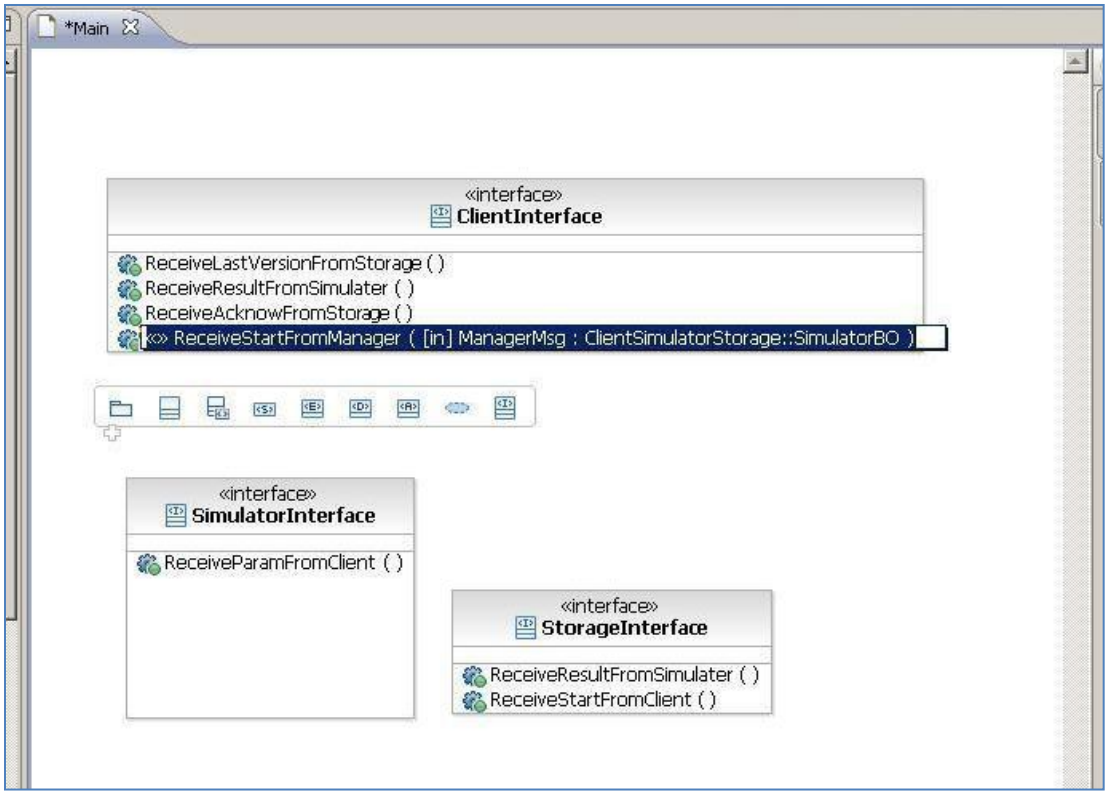


Figure 21: Client-Simulator-Storage Interfaces

2- Defining Parameters and Message types

Since the components in this application are sending messages to each other asynchronously, all messages are one-way messages (i.e. no output messages). We declared an object called *SimulatorBO*, it is used by all interfaces to define the data type of each message parameter as shown in Figure 22. For example, clicking on the *ReceiveStartFromManager* in the *client* interface shows the *ManagerMsg* parameter and its data type. All interfaces use the *SimulatorBO* class to define the data type of their message parameters. All parameters are input parameters since the messages are sent asynchronously.

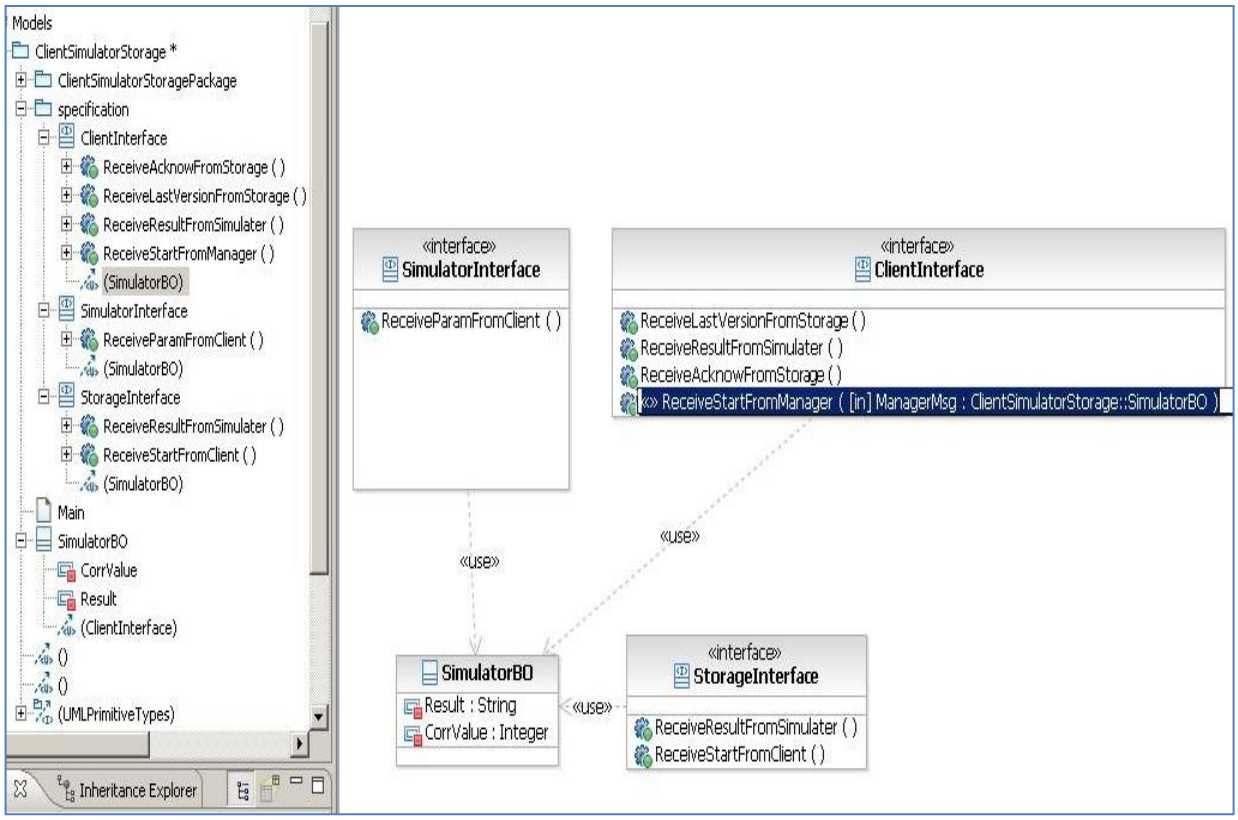


Figure 22: Message types and Parameters

3- Defining Local Variables

Local variables hold the data exchanged between services. It includes the variables that contain values of message parameters, or other variables that hold counter values or results of arithmetic operations. For example, the *simulatorBO* class has the *Result* variable of type *string* which holds a message value, and the *CorrValue* variable of type *Integer* holds the value of the correlation set parameter which identifies to which instance a given message belongs. The use of correlation sets will be discussed in Chapter 5.

4- Identifying Ports, Connectors, Provided and Required Interfaces

A composite structure diagram is used to show the interaction between different elements in a model. It depicts the internal structure of structured classifiers by using parts, ports, and connectors. A port defines the interaction point between a classifier instance and its environment. Ports are used to isolate the internal parts of an object from its environment. A connector is used to link a port and elements of a classifier, or to make communication between two instances [24].

The composite structure diagram also is used to show the interfaces of a service. An interface specifies the interaction that occurs in a port. In any port we can define required and provided interfaces. A provided interface describes the services that instances of a classifier offer to their clients, while a required interface specifies the services that a classifier needs to perform its functions and to fulfill its own obligations to its clients. The diagram in Figure 23 depicts the composite structure diagram for the *client* component. The diagram shows that the client component has the *clientinterface* as provided interface, and the *simulatorinterface* and *storageinterface* as required interfaces. The provided interface has a circle shape, while the required interface has a socket shape at the end.

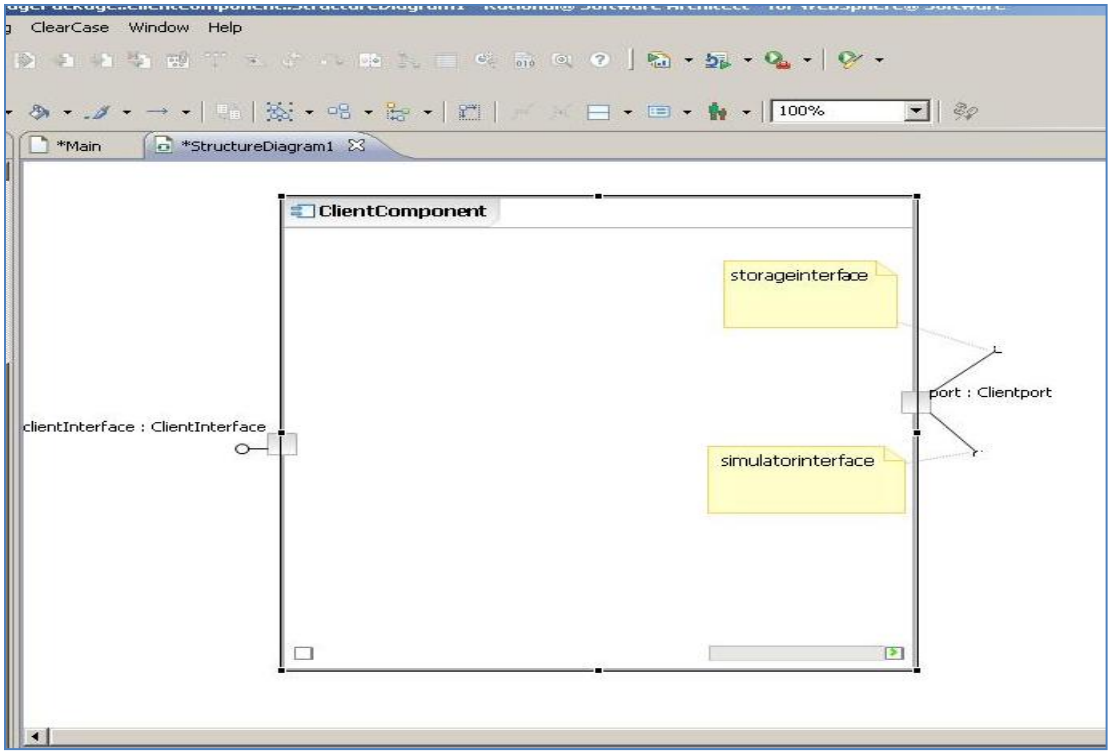


Figure 23: A composite structure diagram of the client component

The next composite structure diagram in Figure 24 shows the internal and external interactions needed by the *client*, *storage*, and the *simulator* components to communicate internally or externally. The diagram has an external port related to the *client* component for a certain component. Any interaction between the outside world and any components goes through these ports by using the links or the connectors as shown in Figure 24. Moreover, each component has two internal ports, one is used to provide an interaction point for the provided interface by the component, and the other port is used as an interaction point for the required interface needed by that component.

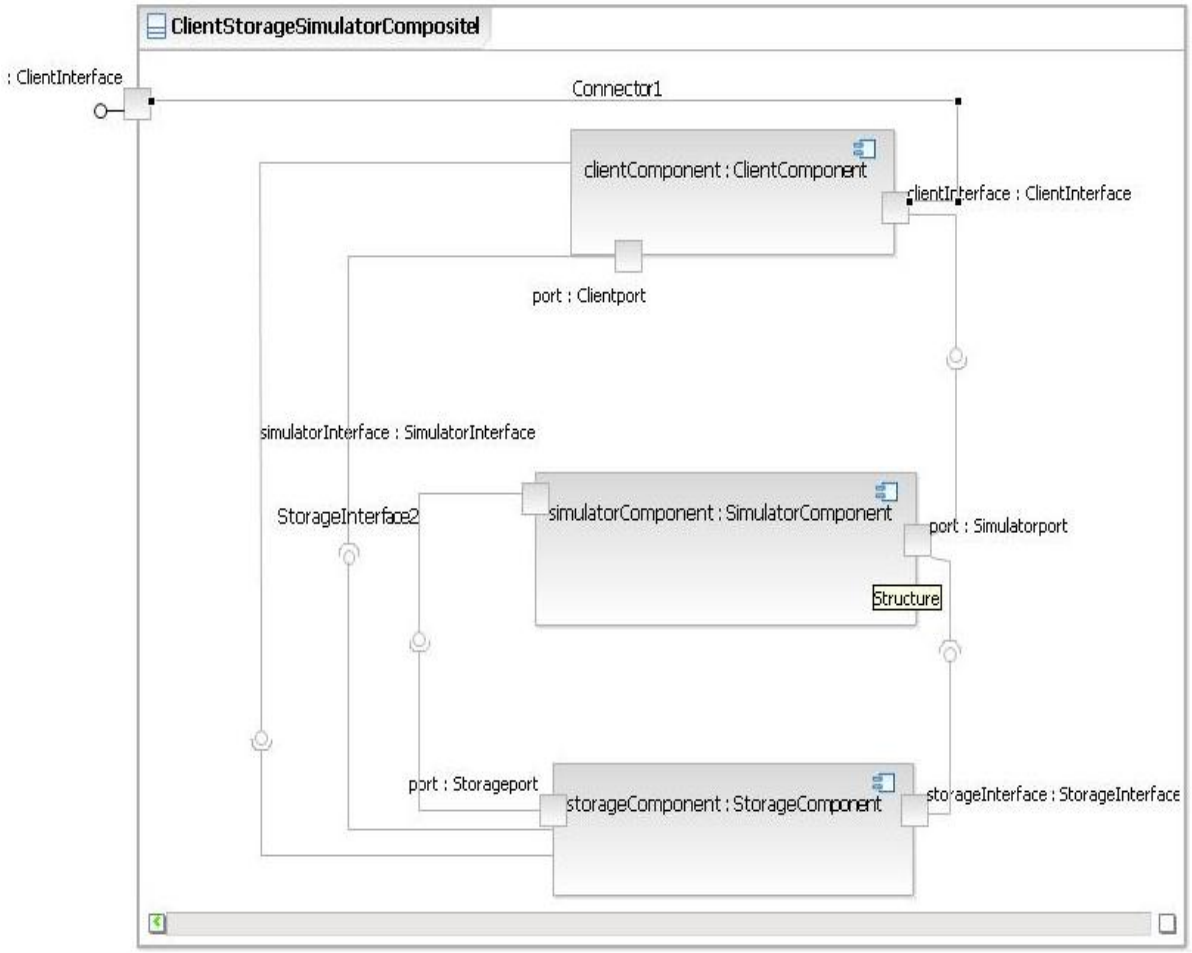


Figure 24: Composite Structure Diagram for the Client-Simulator-Storage Application

5- Configuring Transformations

After identifying the necessary information elements for the AD-to-BPEL transformation, the next phase is performing the model transformation using the RSA tool. A model transformation is an algorithm that takes a source model as input, and produces a new target model. Different transformations are supported by the RSA tool, and may convert UML models to code, convert code to UML models, or convert models to models at a different level of abstraction. A

transformation algorithm must be configured before executing the transformation. The configuration implies specifying the set of rules that define the relationship between the source and target of the transformation.

In order to configure a transformation, the following information must be specified:

- **Transformation type:** It specifies a particular transformation being applied, for example UML to SOA transformation.
- **Source:** It shows the name of the initial model that will be passed to the transformation algorithm.
- **Target container:** It identifies the project where the generated artifacts are stored.

The screenshot in Figure 25 shows part of these elements. As the figure shows, the name of the configuration is *Newconfiguration*, the target model of this configuration will be stored in the *ClientSimulatorStorage* module and the required transformation is UML to SOA.

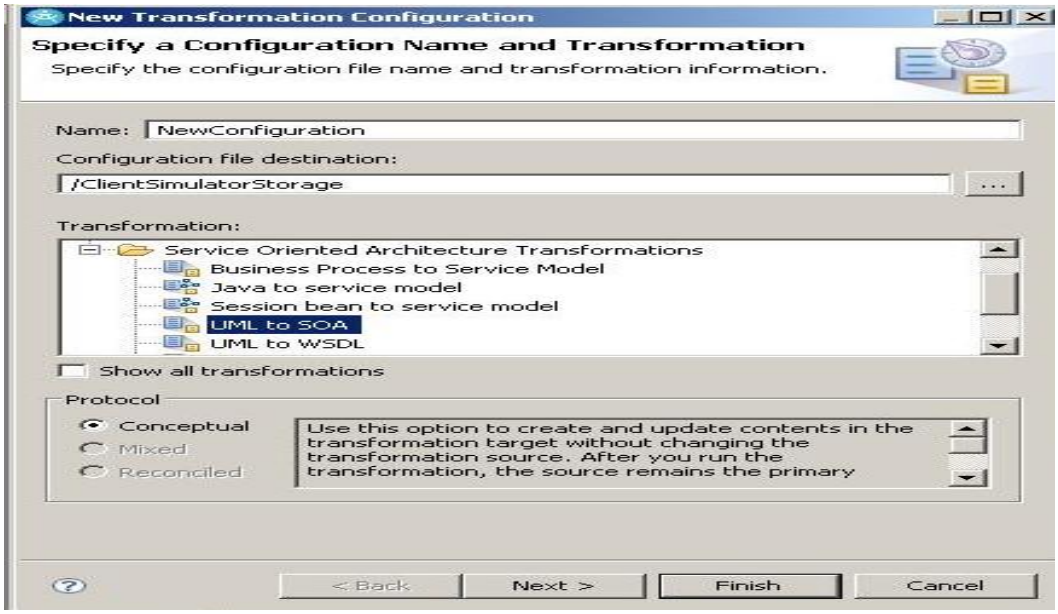


Figure 25: Configuring transformations in the IBM RSA tool

The next screenshot in Figure 26 shows the source and target transformations, the source model is a UML model which contains the activity diagrams of the client, storage, and simulator components. In addition, the source model includes the component diagram of each component. The target model *TargetSim* is the container project of all generated artifacts, this include all the artifacts with extension .BPEL, .WSDL, and, .XSD.

6- Running the Transformation Configuration

The BPEL artifacts are generated by running the transformation. This transformation invokes different transformation extensions to generate outputs for specific domains and runtime environments. The generated implementation model is a complete default realization of the service specifications that was explained in this chapter [24].

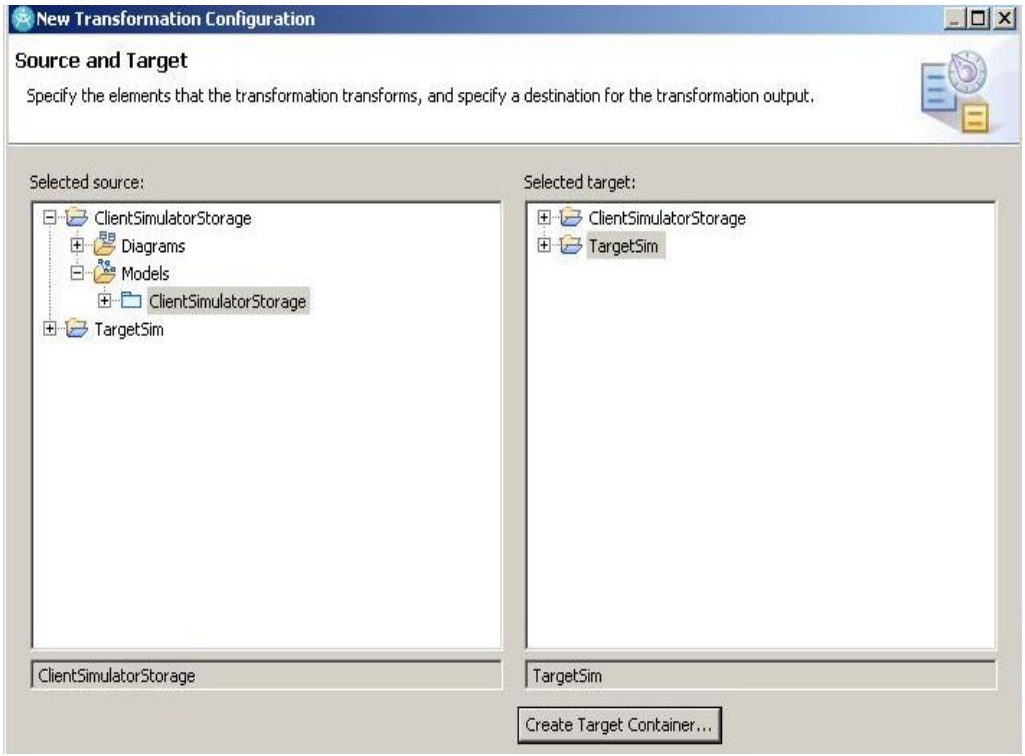


Figure 26: Specifying Source and Target Transformations

The output of the transformation depends on the transformation option that we specify in the transformation configuration. We have chosen the UMLto SOA transformation option; it is performed in two steps executed in sequence. First, the UML-to-SOA invokes the UML to SCDL transformation which is an IBM tool that is used to generate artifacts for service-oriented architecture (SOA). It creates Service Component Architecture (SCA) artifacts, and module and library projects. The SCA is a programming model developed by IBM for assembling diverse business components into reusable service components and applications following SOA. The generated module and library projects have the same structure as a WID project that enables developers to export them into the WID tool, then refine the implementation details of a business process and create more services.

After the UMLto SCDL transformation, the following transformations extensions generate output for specific domains and runtime environments:

- UML to WSDL generates Web Services Description Language (WSDL) interfaces.
- UML to XSD generates XML Schema Definition (XSD) data types.
- UML to BPEL creates a Business Process Execution Language (BPEL) process for each UML structured activity.

The screenshot in Figure 27 shows the output generated by running the configuration for the *client-simulator-storage* application. The *TargetSim* is the name of the project container, it contains four nodes; the first opened node *ClientSimulatorStorage* contains the generated WSDL interfaces and the .XSD file that represent the *simulatorBO* class.

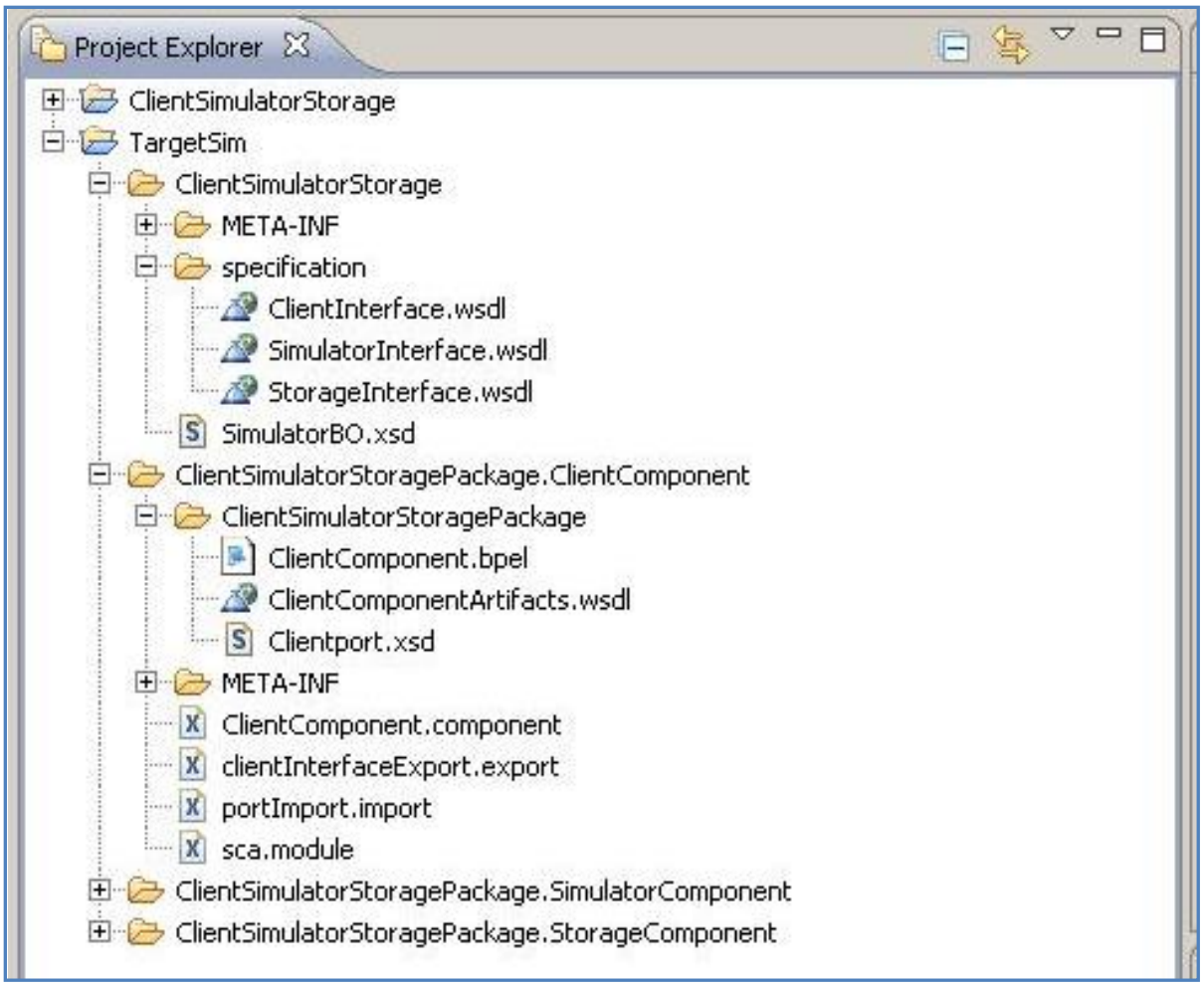


Figure 27: UMLtoSOA Generated Output in IBM RSA.

The second opened node (*ClientSimultorStoragePackage*) is related to the *client* component, it includes the generated BPEL *client* process, and the WSDL component artifacts which represent the required and provided interfaces, and the *Clientport* which is related to the *client* port in the composite structure diagram where data are exchanged between the *client* and the outside world, (see Figure 23 for more details). The last two closed nodes represent the generated output of the *simulator* and *storage* components, which have similar structure as the *client* component.

Chapter 5

5 Deploying, Modifying, and Executing BPEL Process

5.1 Introduction

The WebSphere Integration Developer (WID) is an IBM tool that is used to build and integrate business applications that fit into the Service-Oriented Architecture (SOA). It allows developers to wire components together independent of their implementation and without knowledge of low-level implementation details [25]. The IBM WID is integrated with the WebSphere Process Server (WPS) to provide a runtime environment for applications executed by the integration developer. The IBM WID is used here not to build applications from scratch, but rather it is used to import BPEL transformations from the IBM RSA tool. The imported applications can be further modified and executed. Two case studies are introduced in this chapter to clarify necessary modifications for the BPEL artifacts generated by RSA. Figure 28 shows a snapshot of project import screen in the IBM WID.

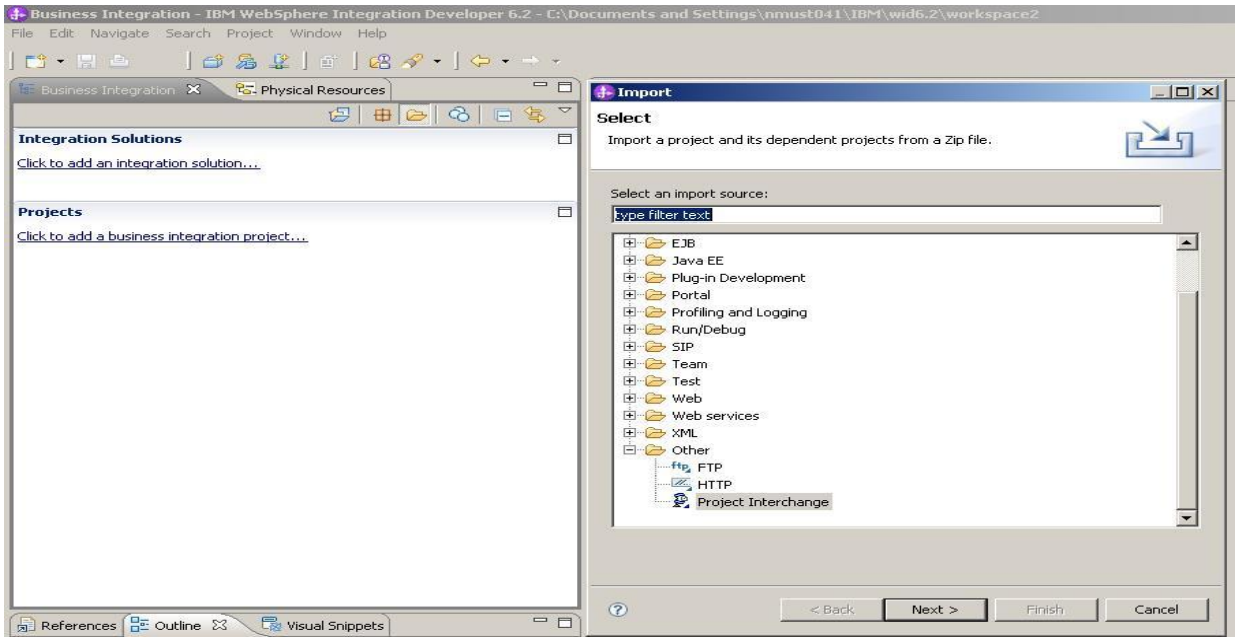


Figure 28: Importing Projects in WID

The left pane identifies which project to be imported into WID. The project interchange option that we have chosen represents an option to import a project generated by RSA tool. The screenshot in Figure 29 shows the RSA client-simulator-storage project imported to the WID.

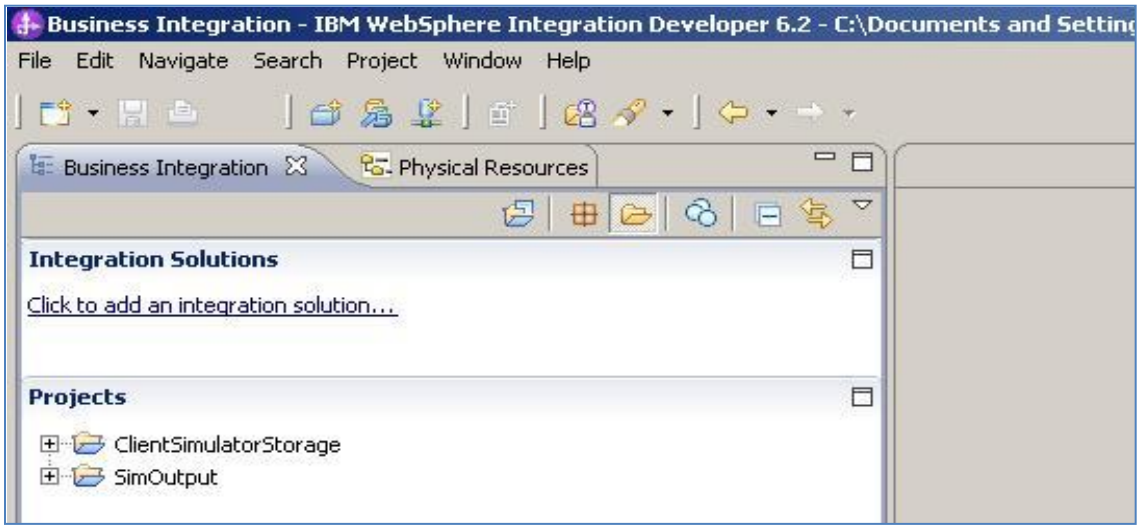


Figure 29: Importing the Client-Simulator-Storage in IBM WID

5.2 Case Study 1: The Client-Simulator-Storage

This application was explained in Section 2.2 and is used again in Section 4.2 to show how activity diagrams are transformed into BPEL processes. It is used here to clarify several issues. First, it introduces the necessary modifications to the BPEL artifacts to produce valid executable BPEL processes. Second, it shows how asynchronous messages are handled in weak loops to avoid race conditions.

5.2.1 BPEL Artifacts Modifications

This section describes three important modifications to the *Client-Simulator-Storage* application in the IBM WID.

1- Adding Correlation Sets

Correlations are used in runtime environments where there are multiple instances of the same process type. Correlation sets uniquely identify the process instance and are used to ensure that a certain business process sends the right message to the right partner instance. This issue can be solved by creating a correlation set parameter of a certain data type that identifies message/instance interactions. Correlation sets are required when more than one receive activity exists in a given process instance. Moreover, they can be used with Invoke, Reply, or Pick activities. For example, the graphical view of the BPEL *storage* process in the *client-simulator-storage* application has more than one receive message from the *client* and the *simulator* as shown in the example Figure 30.

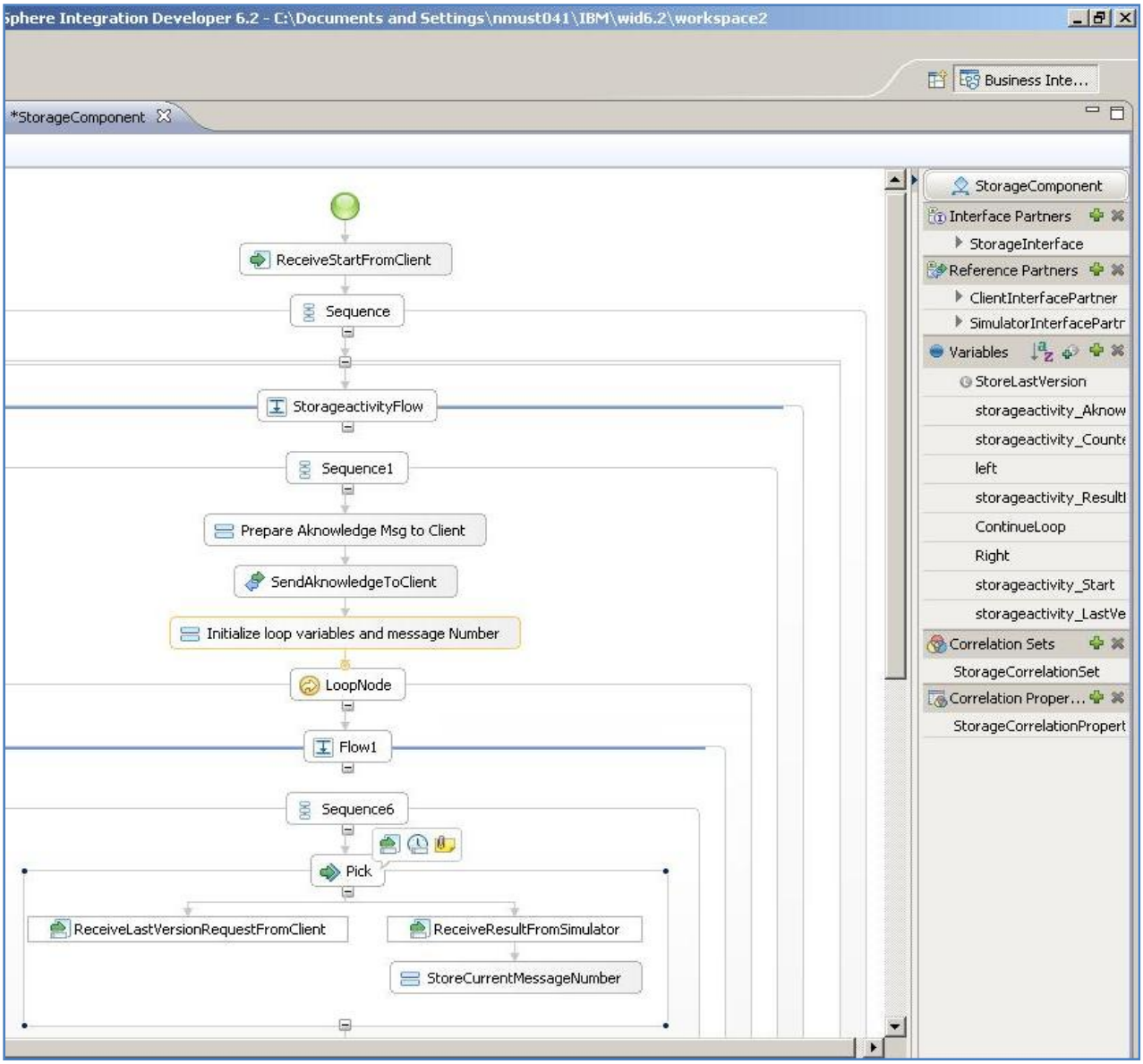


Figure 30: The BPEL Storage Process

We created a correlation set called *StorageCorrelationSet* with a *StorageCorrelationParameter* of type *int*. Any message received by the storage process should include a parameter that relates to this parameter and is of the same data type. For example the *ReceiveStartFromClient* Message in Figure 30 has a parameter called *CorrValue* of type *int* , the other messages have also

correlation parameters of type *int* as illustrated in Figure 31. The value of the correlation set parameter can be bounded to the message during runtime.

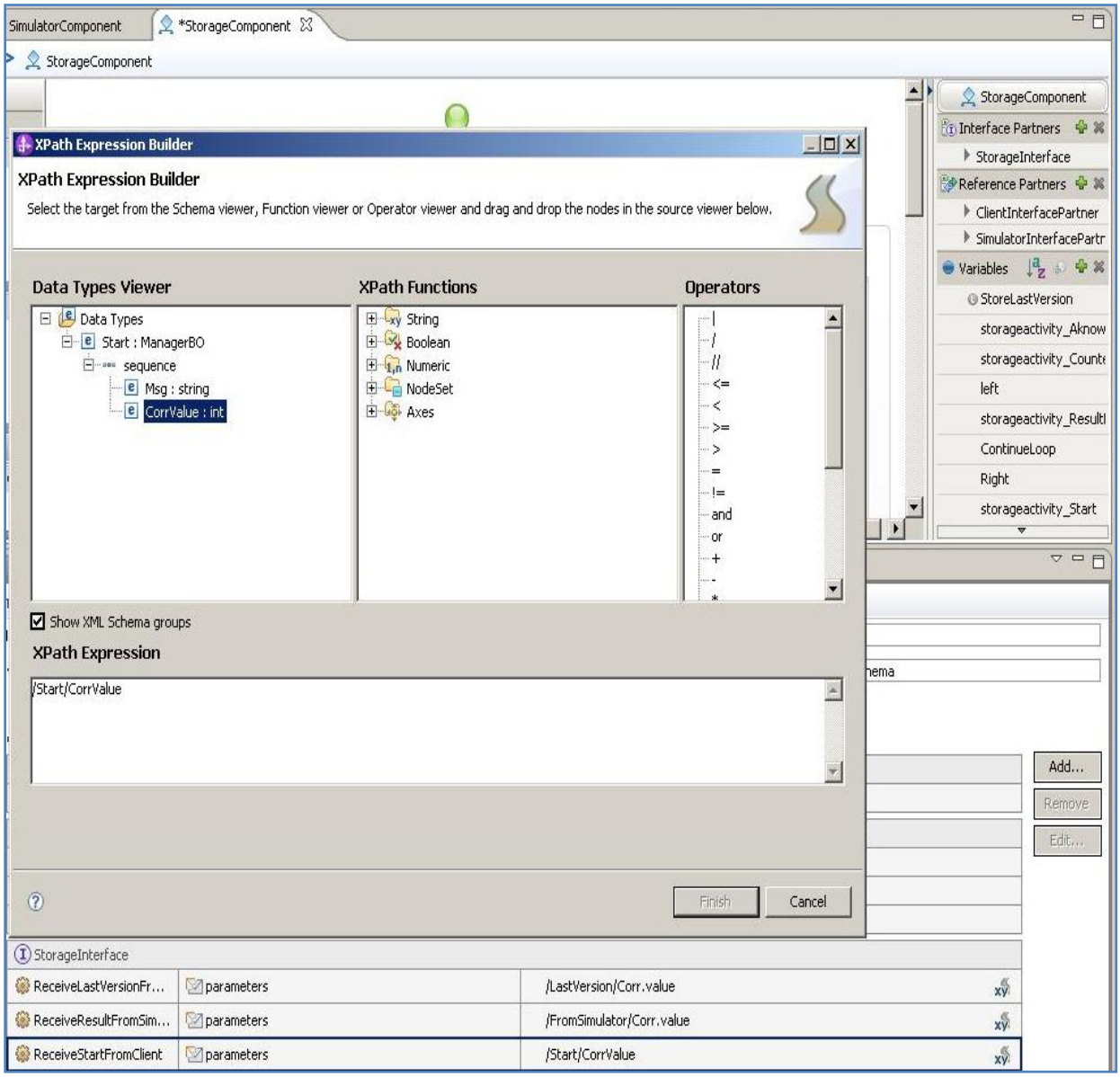


Figure 31: Adding Correlation Set Parameter to a Message

2- Static versus Dynamic Process Instantiation (adding a manager process)

In the *client-simulator-storage* application, we are assuming that the business processes are running in distributed systems. Whenever a BPEL process receives a first message from another process, a new process instance is created, and the process starts running. We therefore have the question of who starts the first process in the application? We have two solutions: (a) the processes instance is created when it received a message from an external operator (manager) - we called this *static instantiation*, or (b) the process is created when the first message arrives from a *client* process which is part of the application (no manager), we called this *dynamic instantiation*. In the *client-simulator-storage* application, we applied the static instantiation, whereas in the second case study (client-service) we applied the dynamic instantiation. The manager business process starts the *client* process since it is the first process in the application, then the other processes are started by the *client* as depicted by Figure 32. The manager process in this figure is invoking the client process by sending an initiation message. The initiation message has a correlation parameter value that identifies a new process instance.

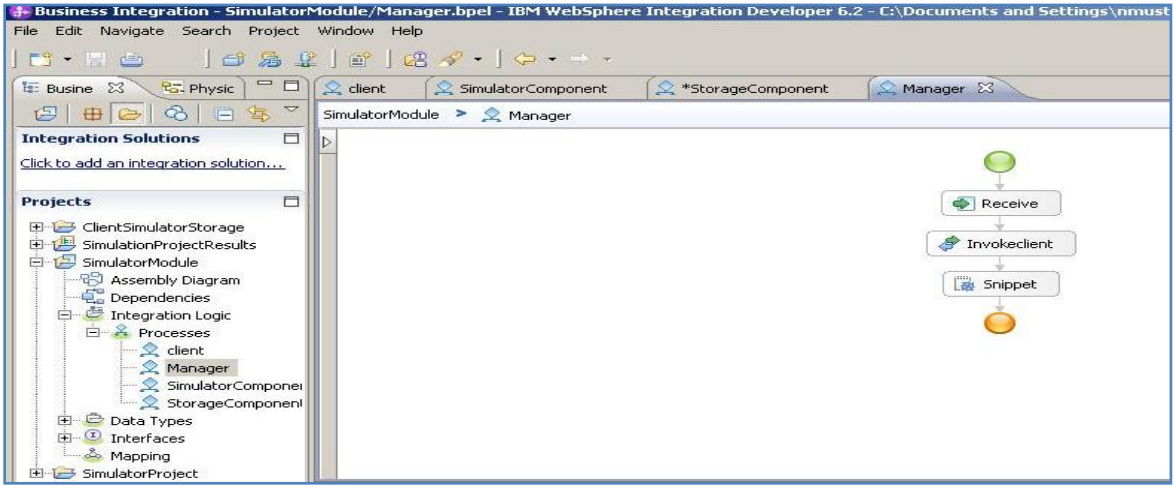


Figure 32: Static Instantiation Using Manager Process

3- Creating an Assembly Diagram

Each project created in the IBM WID should have an assembly diagram that wires all running processes or services without showing the detailed logic of each process. The provided and required interfaces for each process are shown in Figure 33. The diagram shows four BPEL processes, each one has a provided interface denoted by the circled letter I, and the cardinality 1..1 shown on each process represents a required interface for that process. For example, the *client* process has one provided interface (*ClientInterface*)and two required interfaces one related to the *storage (StorageInterface),* and the other related to the *simulator (SimulatorInterface)*. The diagram does not show any execution order except that the *manger* process is the starting one. Moreover, Figure 33 depicts the same logic of the structure diagram shown in Figure 23.

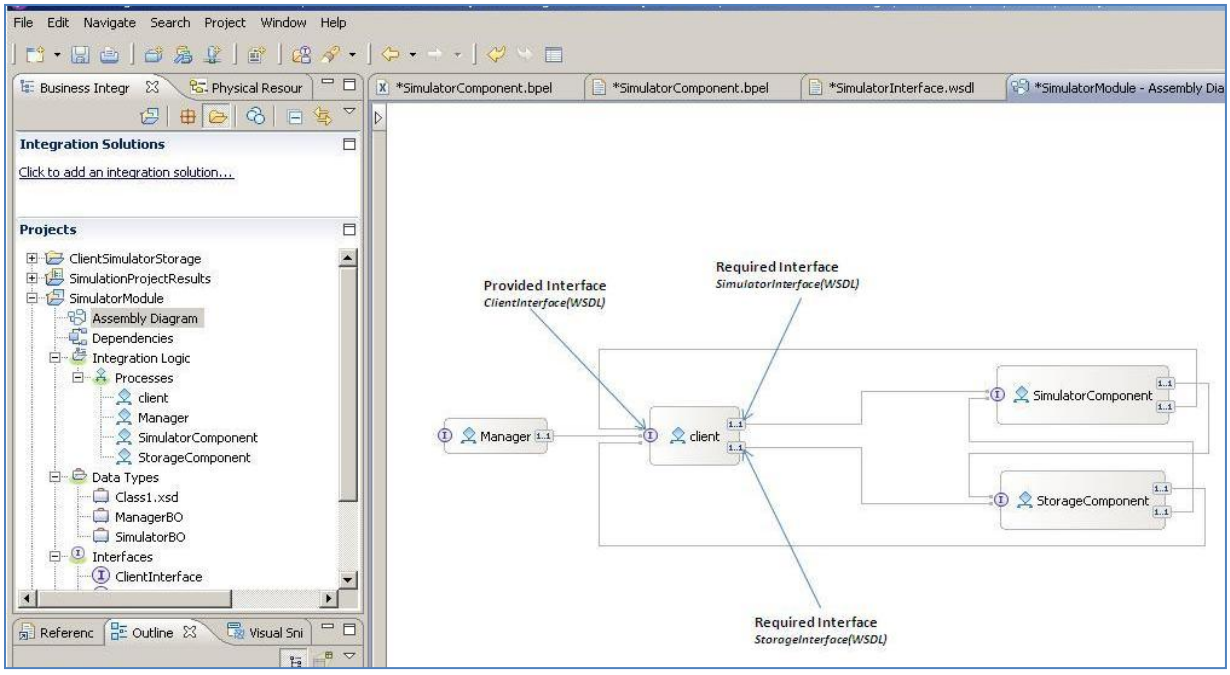


Figure 33: The Assembly Diagram of the Client-Simulator-Storage Application

5.3 Case Study 2: The Client-Service Application

This example assumes dynamic instantiation where a process instance is created when it receives the first message from a *client*. This application shows how to deals with transforming alternative message receptions modeled in UML AD into a valid BPEL artifact. In addition, it explains how to handle the case where one alternative involves the concurrent reception of several messages.

5.3.1 Application Description

The client-service application represents processing order collaboration between a *client* and a *service* components. The *client* fills an order and *sends* it to the service. Upon receiving the order, the *service* processes the order, if the order is not available, the service sends *NoStock* message to the *client* and closes the order, otherwise, the *service* sends concurrently to the *client* two messages; an *ItemFound* message, and an *Invoice* message after the invoice amount is calculated. The *client* component at this case either receives the *NoStock* message, or the *ItemFound* and the *Invoice* messages concurrently. Whenever the *client* receives the *Invoice* message, it prepares the payment and sends it to the *service,* after the *service* receives the payment it closes the order. At any time during this scenario, the *client* can fill another order and continue in a loop determined by a loop counter, if the loop counter exceeds its limit, the *client* stops and the application will terminate. A high level activity diagram that illustrates this application is shown in Figure 34, and a more detailed one is shown in Figure 35. The capital letters S, P, T mean Starting, Participating, and Terminating respectively, while the small letters c, s represent the *client* and *service* components.

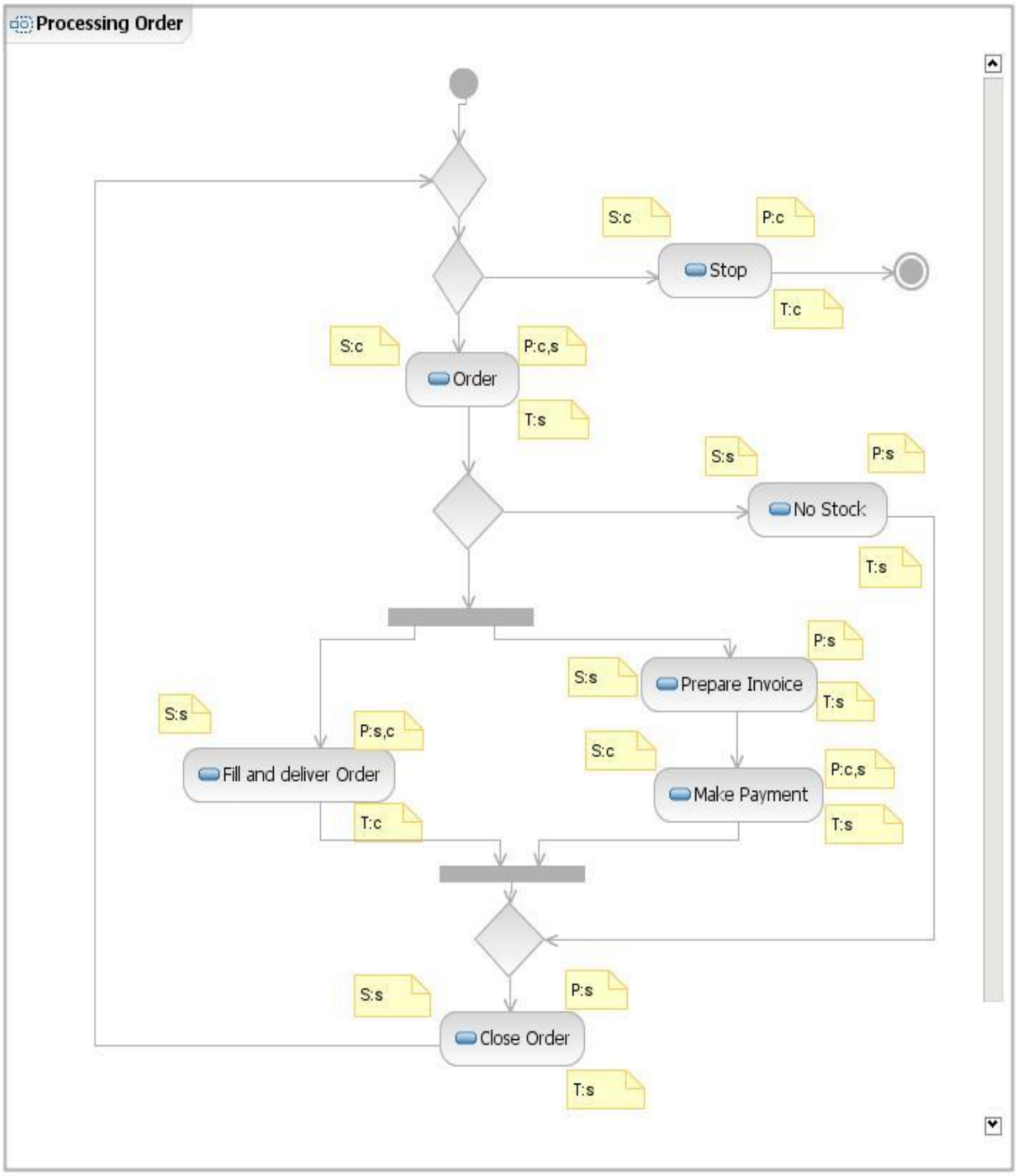


Figure 34: Processing Order Collaboration

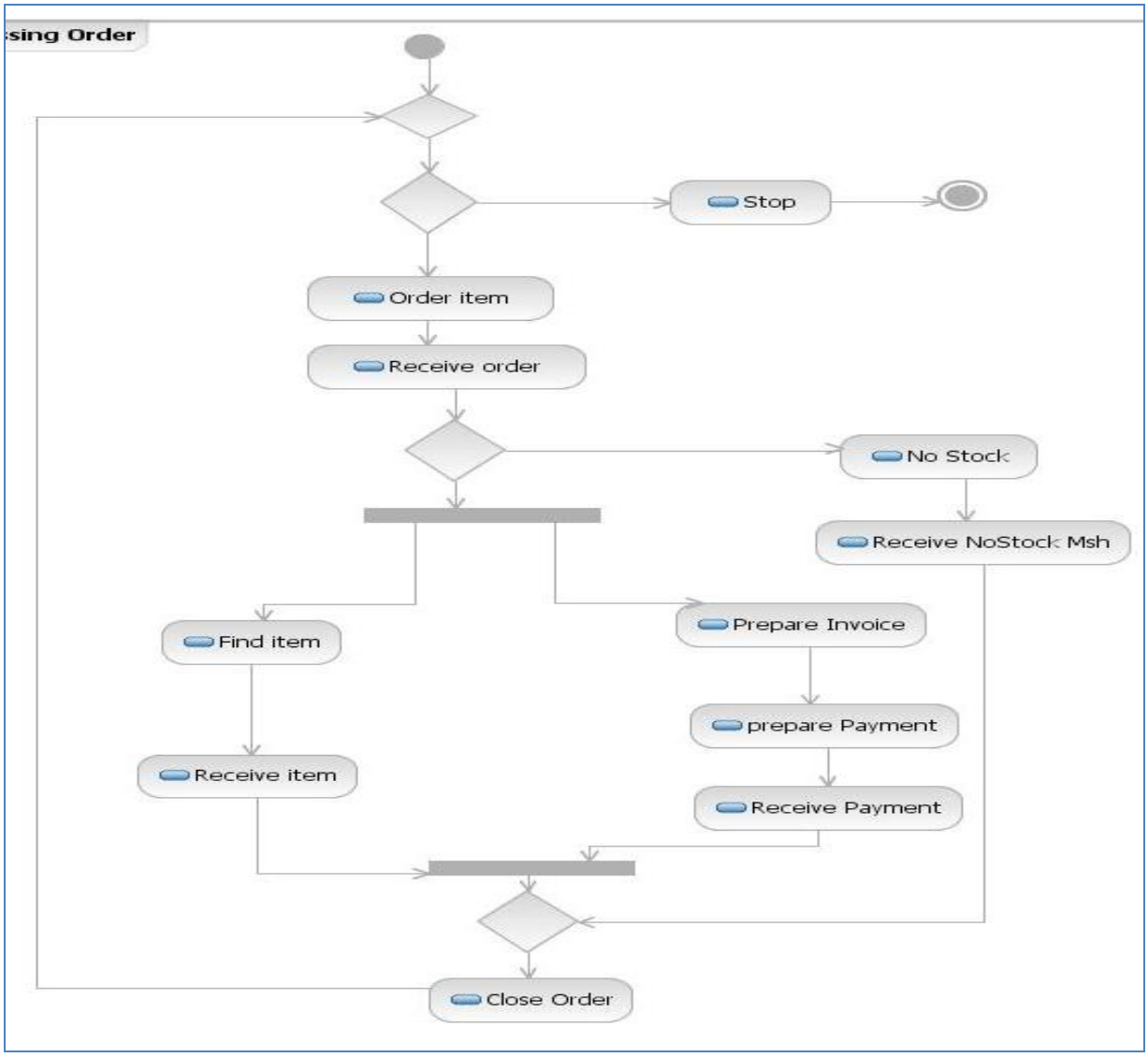


Figure 35: Detailed Processing Order Collaboration

5.3.2 Deriving the Client and Service Activities from System Global Collaboration

Applying the same technique that was introduced in Section 2.2.1, the *client* and *service* activity diagrams are derived. Each activity diagram contains the local actions performed by each component as illustrated in Figures 36 and 37. In these diagrams, it is important to note that the UML notation uses the diamond symbol to represent a merge node, as indicated by label 1, or a

choice node. Furthermore, the choice node may have different semantics: (a) the choice may depend on some local condition (e.g. a Boolean variable with value true or false), which we call "*choice with condition*", as illustrated by label 3 in Figure 36, or (b) the choice may be 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 call the latter case "*choice between messages*", as indicated by label 2 in Figures 34and 35.

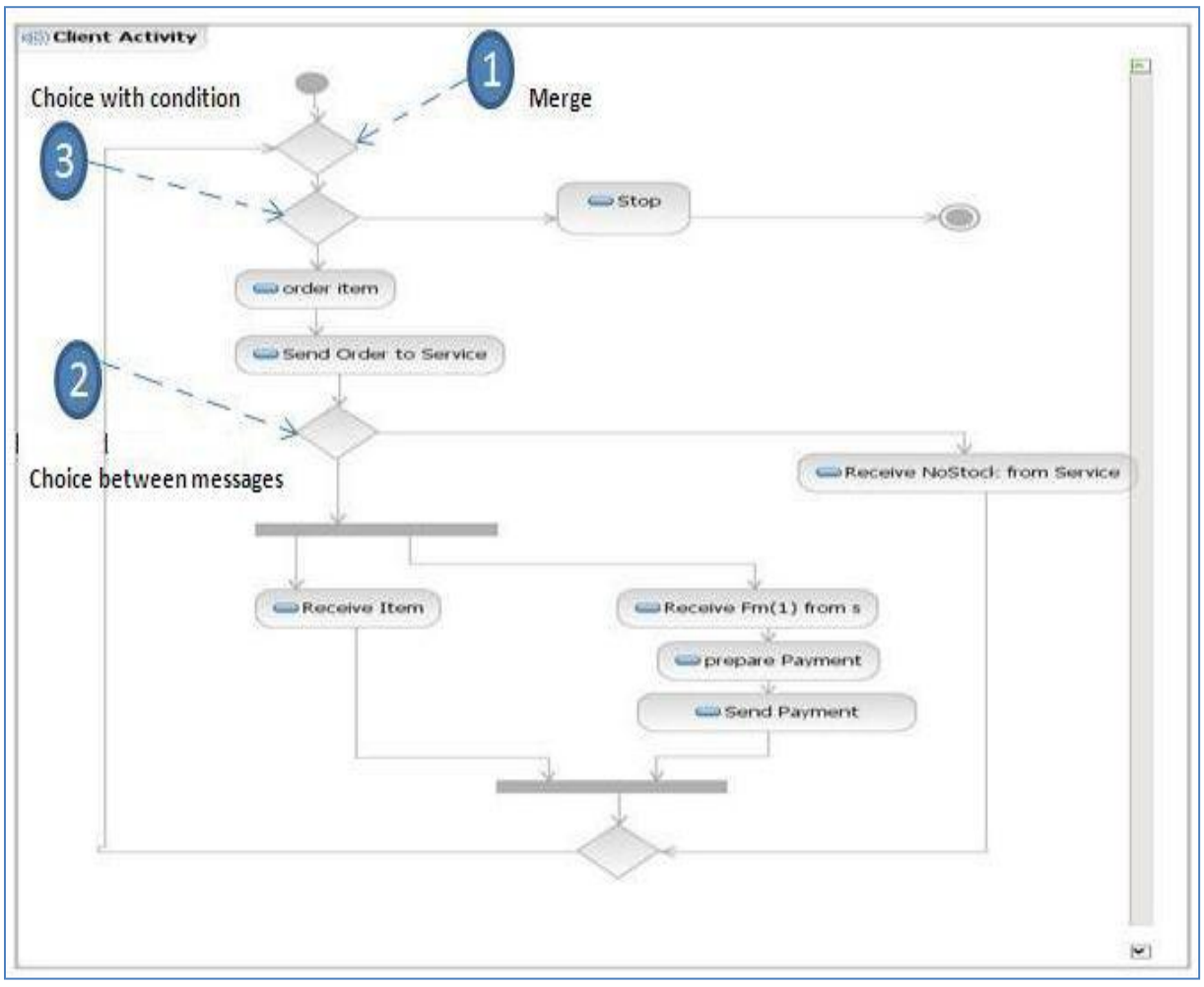


Figure 36: Client Activity Diagram

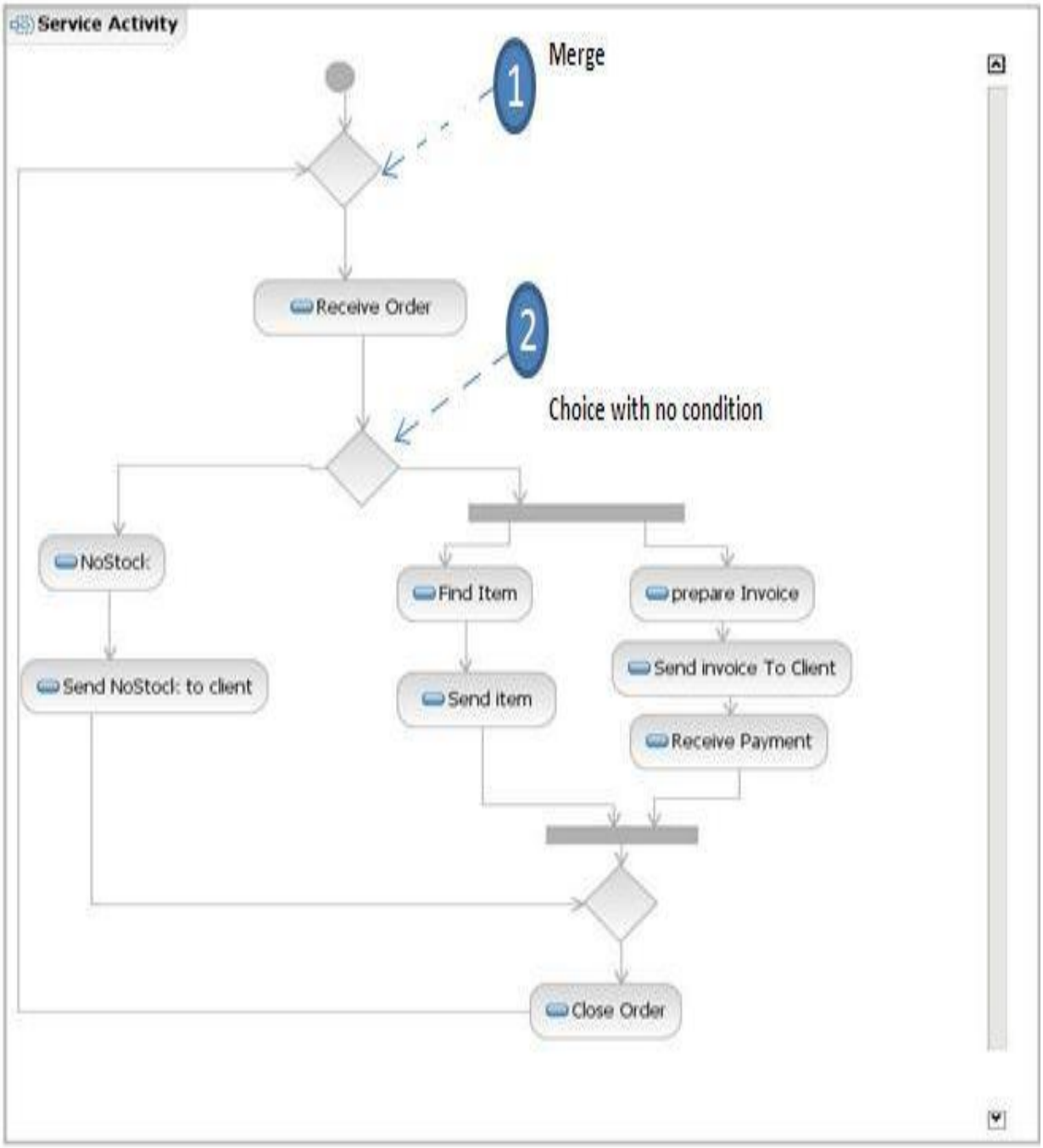


Figure 37: Service Activity Diagram

5.3.3 Modifying Activity Diagrams in RSA

Section 3.3 explained the required modifications to activity diagrams in RSA. Each action node should be specified, in addition each receive node should be correlated to a receive operation that identifies the input and output messages. Other local variables and conditions should be initialized. The diagram in Figure 38 illustrates the client activity diagram specifications in the IBM RSA tool. The tool sometimes uses different representations for nodes for example the loop node in Figure 36 has changed to look like the one in Figure 38. Additional action nodes are added before and after the loop node to initialize condition for starting the loop, another action is added inside the loop node to validate the loop iterations.

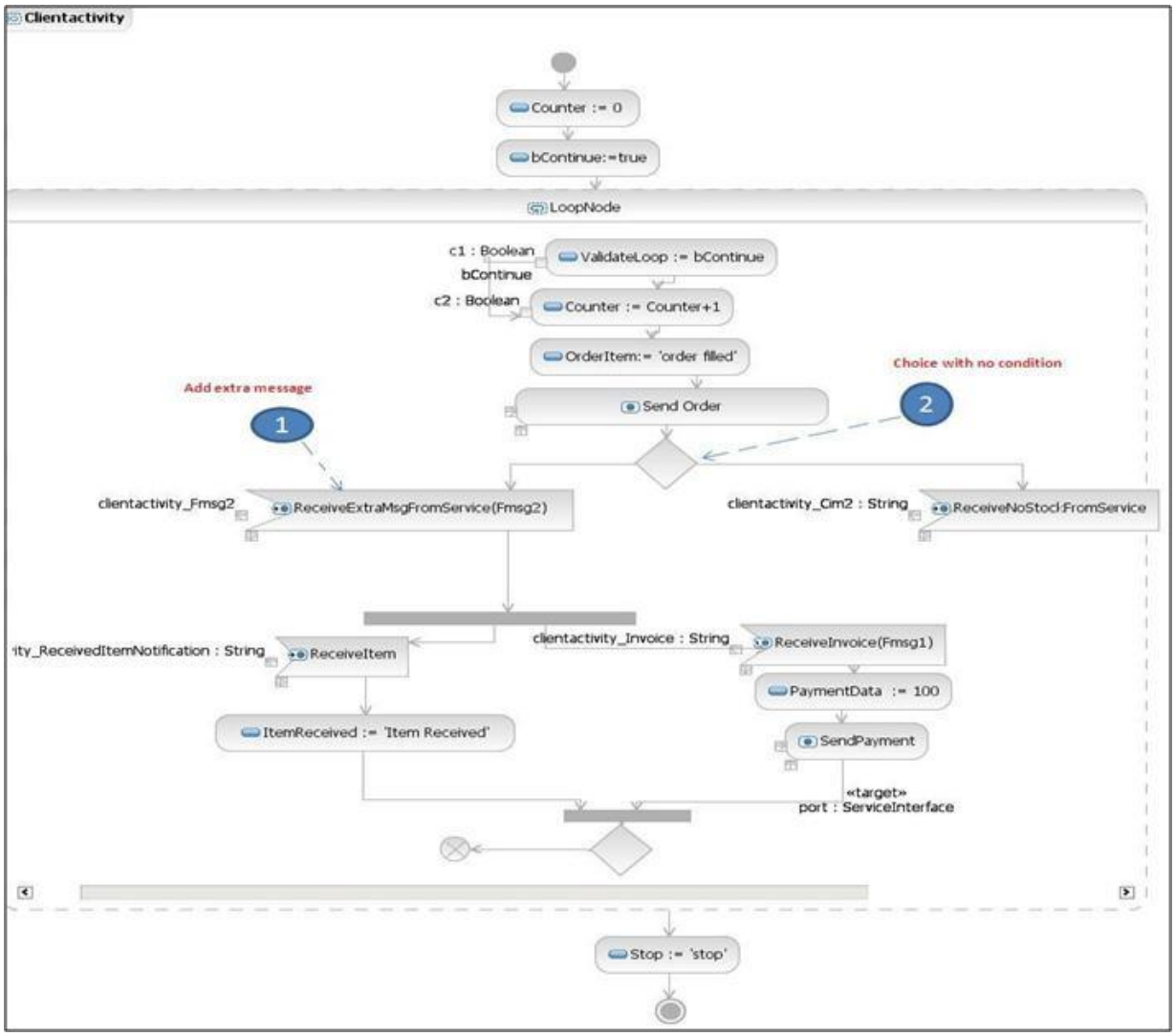


Figure 38: **Modifying Client Activity Diagram in RSA**

5.3.4 Transforming the client-service activity diagrams into BPEL

We applied here the same steps described in section 4.2 to generate BPEL artifacts from activity diagrams. RSA generates two BPEL transformations related to the *client* and *service* components, and two WSDL files related to the interfaces of these components. The

transformation output is imported into the IBM WID tool for further modification and execution. Figure 39 shows a generated *client* BPEL process which is equivalent to the activity diagram shown in Figure 38.

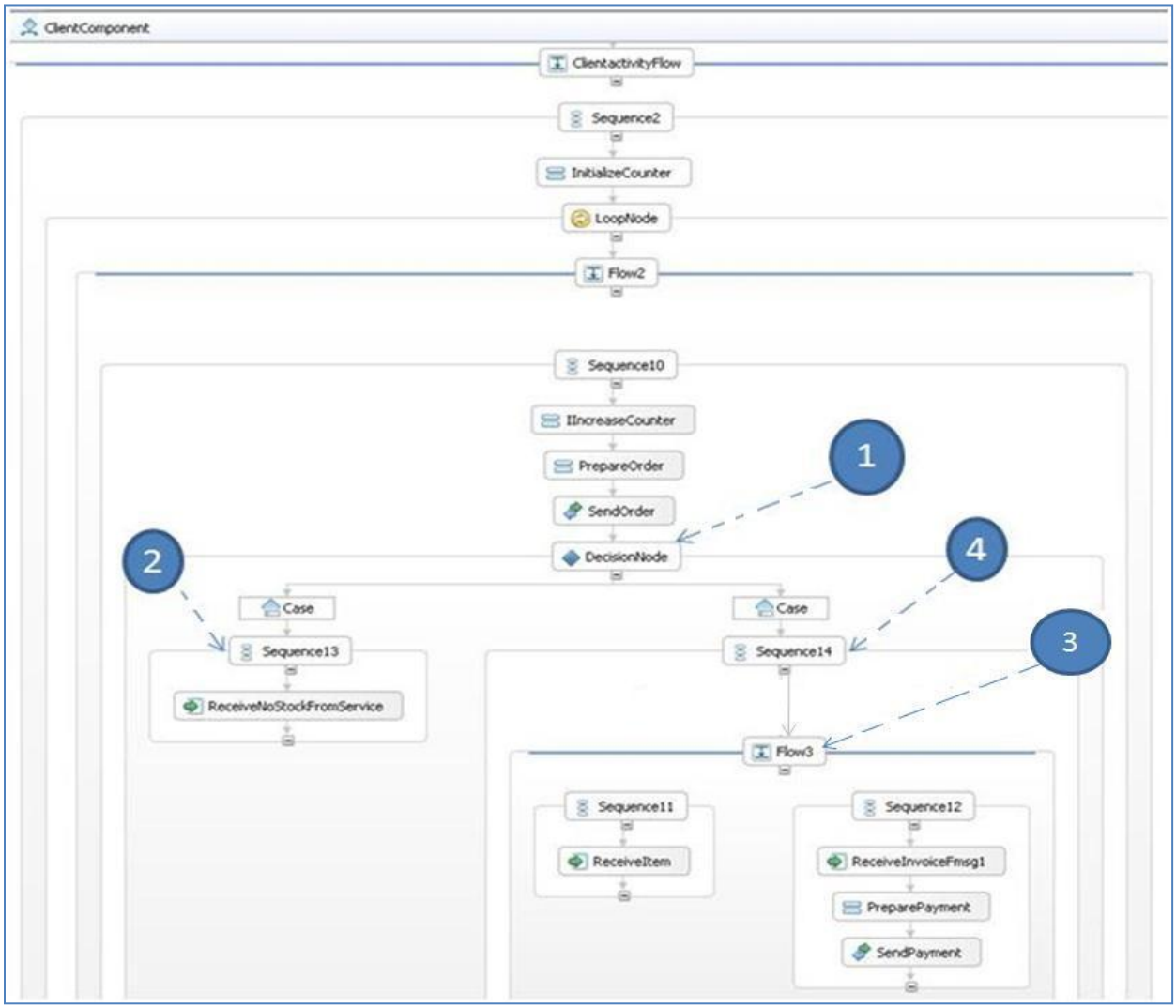


Figure 39: Client BPEL Diagram (partial) in WID before modification

5.3.5 Adding Correlation set and dynamic Instantiation

In this application, we applied the dynamic instantiation. First, we assumed that the *client* process is instantiated by an external operator. Moreover, we assumed that the *service* process instance is always started by the *client*. For each new order, an order number is included; this number represents the correlation set parameter value. In the service side, at each time the *service* receives a new order, a new service instance is created. The termination of the application is governed by the *client* process through the use of a loop counter.

5.4 Executing and Testing the BPEL Processes

We have executed the modified BPEL processes in the IBM WID using the WebSphere process server. The result of executing the *client-simulator-storage* application is shown in Figure 40. During the execution, we were testing the asynchronous messages passing between the collaborative components. The objectives of our test case are to ensure that the integrated behaviours of all participating components corresponds to the global system behavior. The following asynchronous messages were tracked to make sure that they are passed correctly to the intended BPEL processes.

1. The *manager* BPEL process is passing an initial message to the *client* BPEL process.
2. The *client* is sending a starting message to the *storage* process.
3. The *client* is receiving an acknowledge message from the *storage* process.
4. The *client* loop is performed four ; each time the client is defining a parameter and sends it as a message to the *simulator* process for simulation; at the same time, the *simulator* must send each time two concurrent messages to the *client* and *storage* processes.

5. The *client* and the *storage* processes are receiving the concurrent messages that are mentioned in point 4.
6. The *client* is sending a request message to the *storage* to get the final simulated message, and the *storage* process is receiving it.
7. The *storage* is sending the final simulated message to the client. This message must be sent only after the *storage* has received all simulation results from the *simulator*.

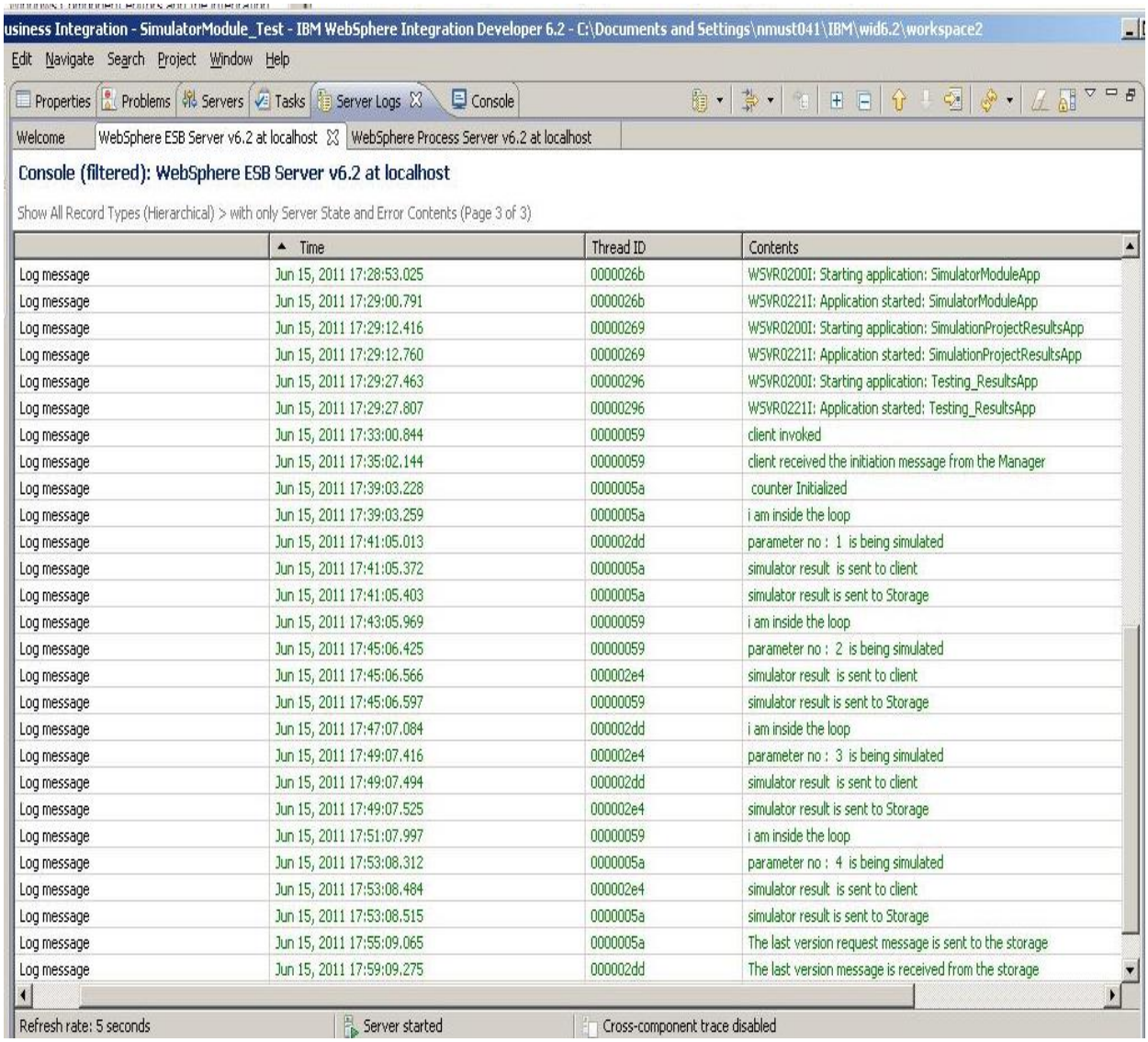


Figure 40: The Result of Executing the client-simulator-storage application

The results of testing the *client-simulator-storage* application are shown in Figures 41, 42, and 43. As Figure 41 shows, the application is started when the *manger* process receives an initiation message, then, the manger invoked the *client* process by sending an asynchronous message to it. The *client* also starts the *storage* process through an asynchronous message. We can also notice in this figure the initiation of the while Loop and the counter increment. Figures 42 and 43 are continuations for the same test case. in Figure 42, we can notice the *switch* which checks whether the storage has received the *final* simulation message from the simulator. In this case the loop will terminate, and the *storage* process sends the *final* message to the *client* and the application terminates afterward as shown in Figure 43.

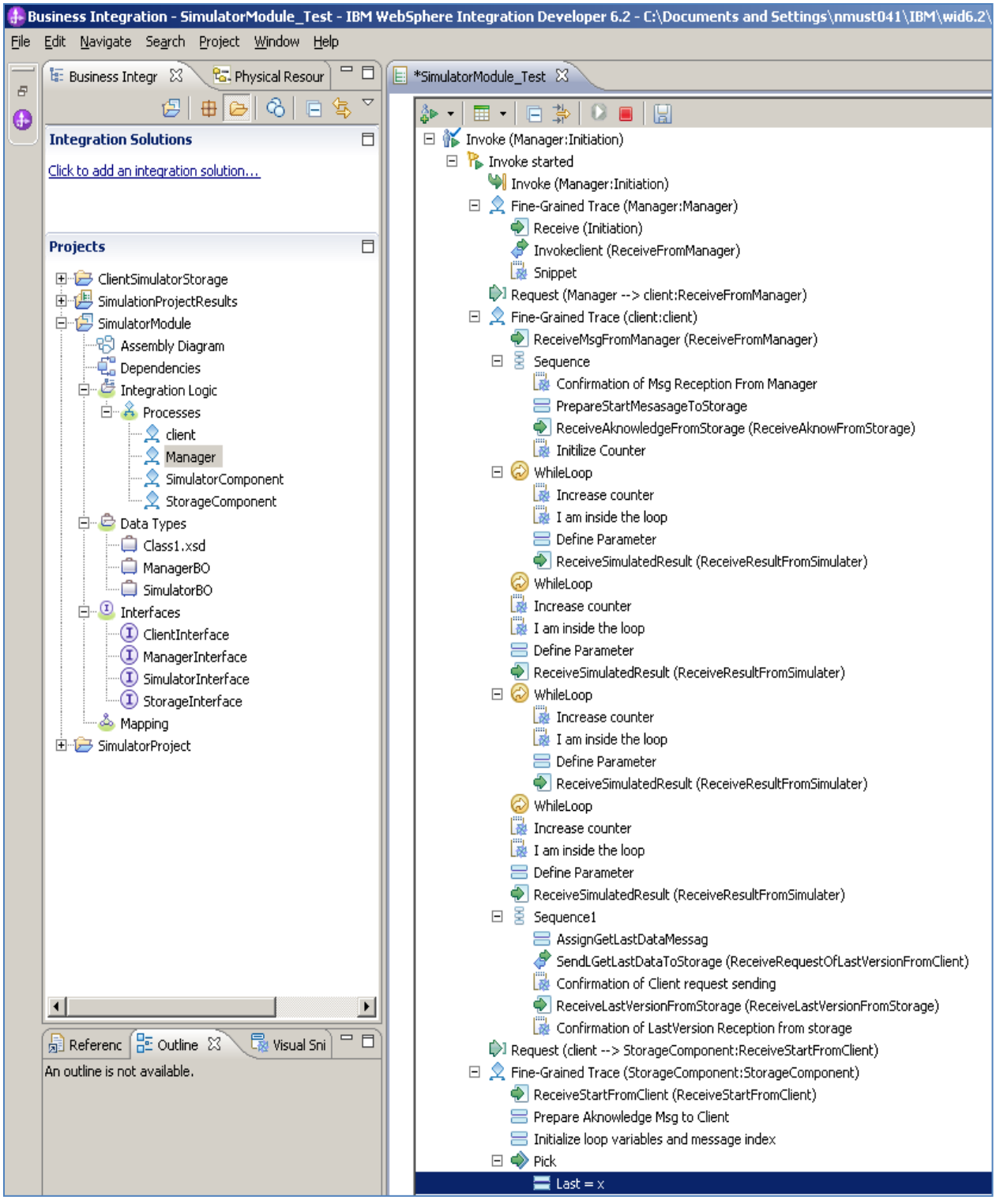


Figure 41: Testing the client-simulator-storage application

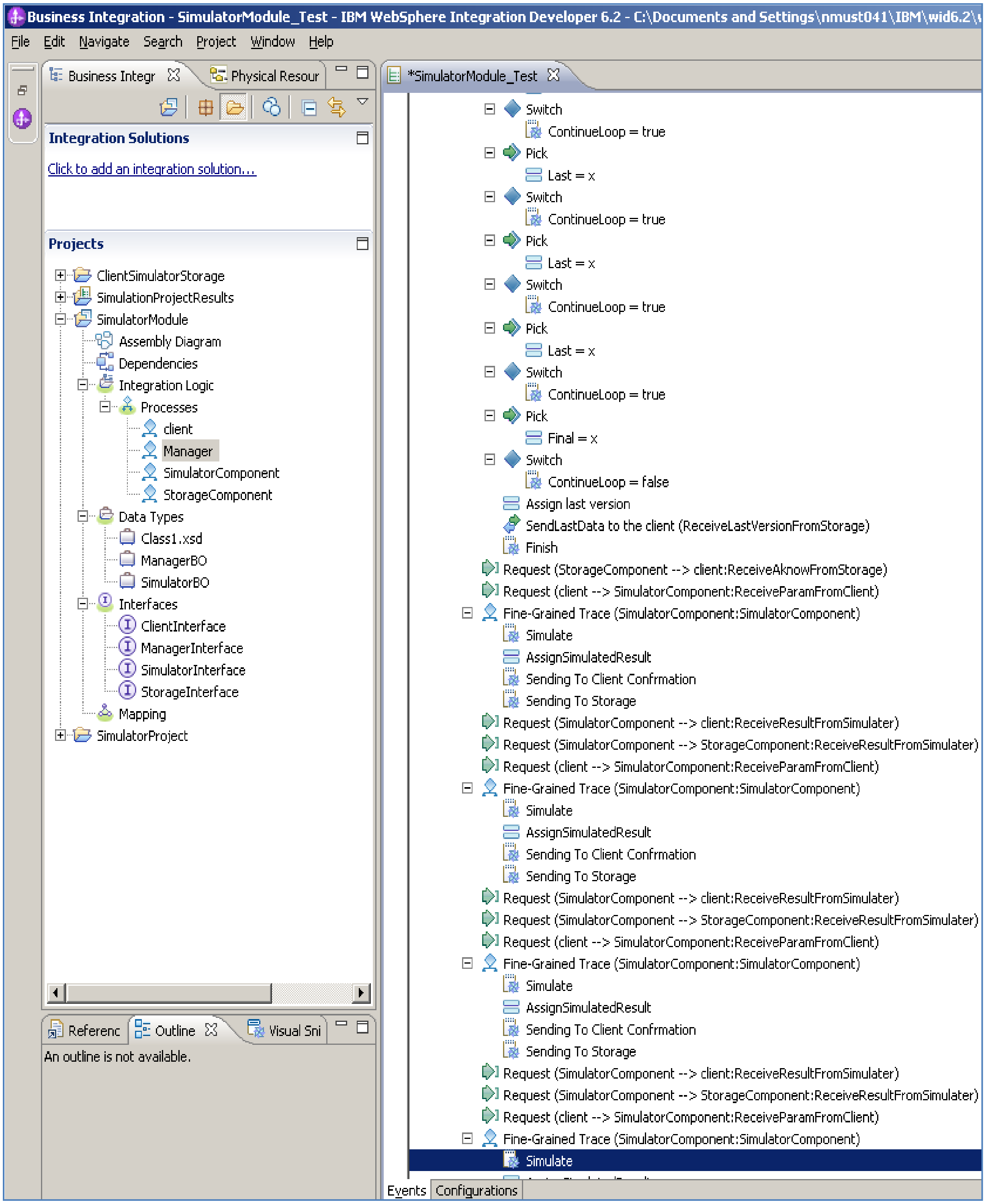


Figure 42: Testing the client-simulator-storage application - continued.

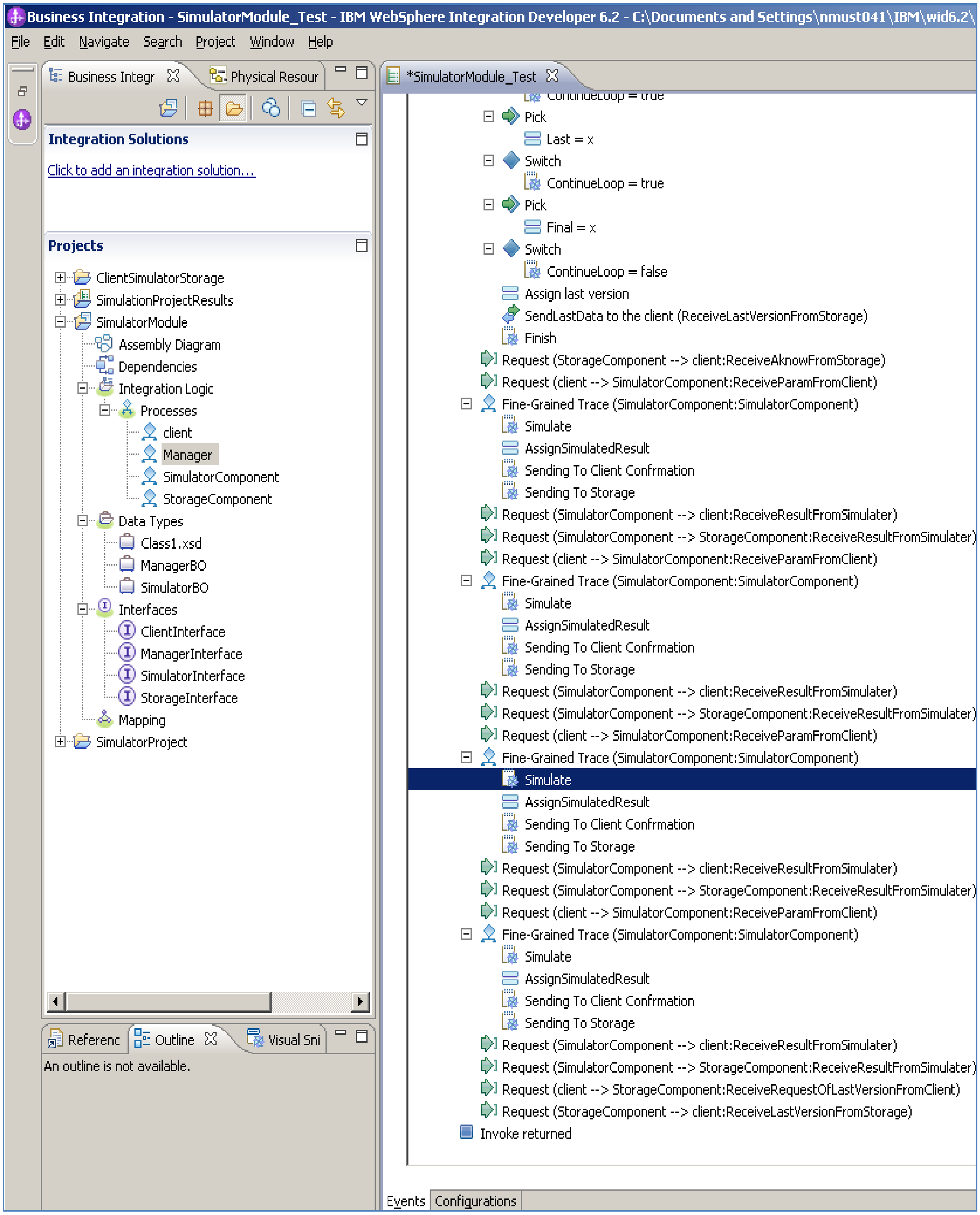


Figure 43: Testing the client-simulator-storage application - continued

5.4.1 Evaluation

Our test results prove the following:

1- All necessary asynchronous messages are passed correctly among the participating components.

2- The integrated behaviors of the *client*, *simulator*, and *storage* BPEL processes correponds to the global system behavior.

3- In terms of scalability, we have tested a maximum of three BPEL processes to prove our findings and the system shows a normal behavior without any crashes. We believe that our SOA model can be expanded to more than three processes without any problem.

4- In terms of performance, we have not measured the system performance since this is beyond our research objectives.

Chapter 6

6 Transformation Limitations and Solutions

6.1 Overview

The Service-Oriented Architecture (SOA) provided by the Web Services standards supports Model-Driven Development by allowing global business process models described in UML Activity Diagrams to be transformed into Web Services components specified by WSDL and/or BPEL. In this chapter we describe our experimentation with the transformation of UML-2 Activity diagrams into PBEL processes. Using examples coming from the derivation of local behaviors from global system requirement specifications, we found out that the IBM Rational tool does not support certain important asynchronous message exchange scenarios, and we describe here how the generated BPEL processes can be manually modified and corrected. We also noted that the translation tool does not correctly deal with the choice between two different sets of messages that could be received at a given point during the execution of the behavior. Moreover, we describe race conditions between different message reception and sending events.

6.2 Alternative Messages Reception

We consider here a simpler version of the client-service application. The *client* process in Figure 44 sends either message *A* or message *B* to the *service* component. The behavior of the *service* is shown in Figure 45**:** 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.

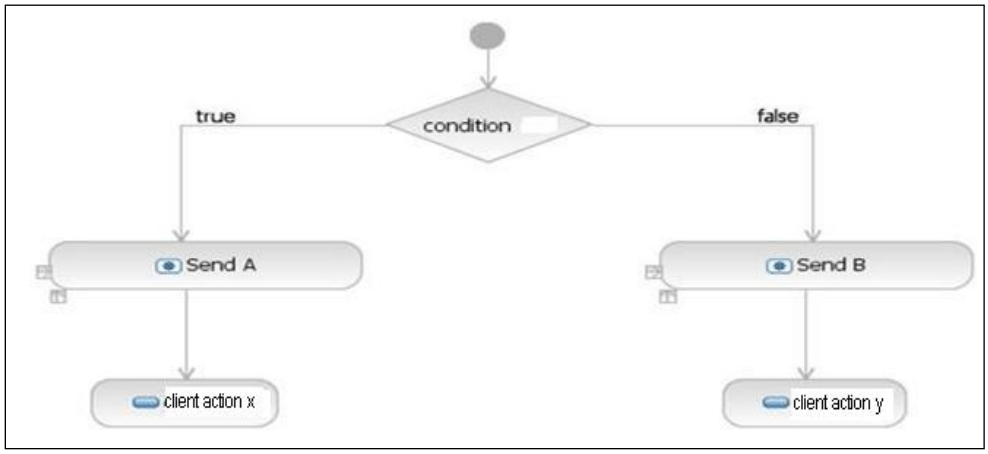


Figure 44: Activity Diagram - Client Behavior

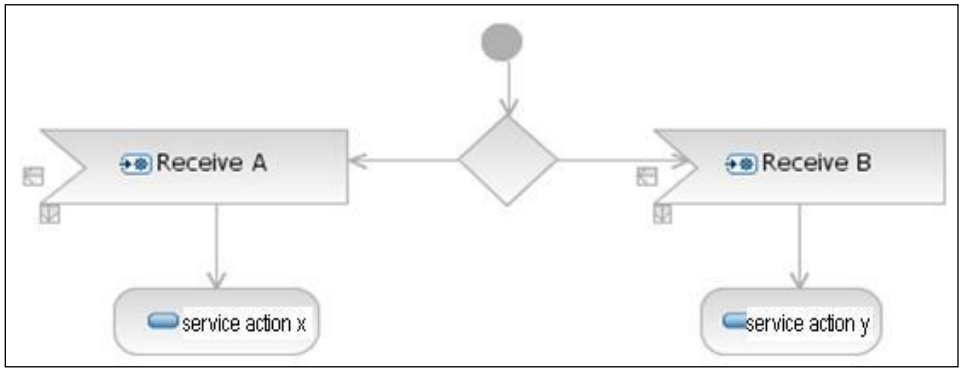


Figure 45: Activity Diagram - Service Behavior

The transformation function of the RSA tool did not distinguish these two cases. Both decision nodes, of the *client* and of the *service*, are 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 corresponds to the event that occurs first. Figure 46 shows the automatically obtained BPEL process for the *client* component, while Figure 47 shows the BPEL process for the *service* component after the manual modification.

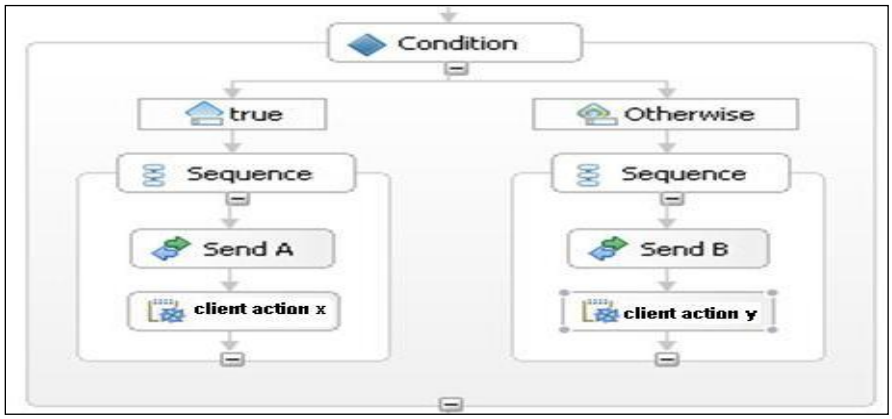


Figure 46: BPEL Process of Client Process with Decision Node

Figure 47: BPEL Process of Service Process with Alternative Message Receptions

6.3 Alternatives 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 a *client* process shown in Figure 38. The alternative indicated by label 2 includes the reception of two messages *ReceiveInvoice* and *ReceiveItem*. 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 transformed into a BPEL process with a <pick> node involving the three message types, since the <pick> node receives only a one message, never two. We propose in the following subsections two different solutions to this problem: (a) adding an extra message before the concurrent messages, and (b) converting the concurrency into alternatives.

A- Adding an Additional Message Before the Concurrent Receives

A proposed solution is to add an additional message transmission when the concurrent alternative is chosen. This leads to the Activity diagram of Figure 48, which involves an additional message transmission indicated by label 1, but has a simple overall structure. In case of receiving the extra message, the reception of the messages *ReceiveInvoice* and *ReceiveItem* will be performed concurrently. The transformation of Figure 48 into BPEL can be performed as described in Section 4.2 and is shown in Figure 49.

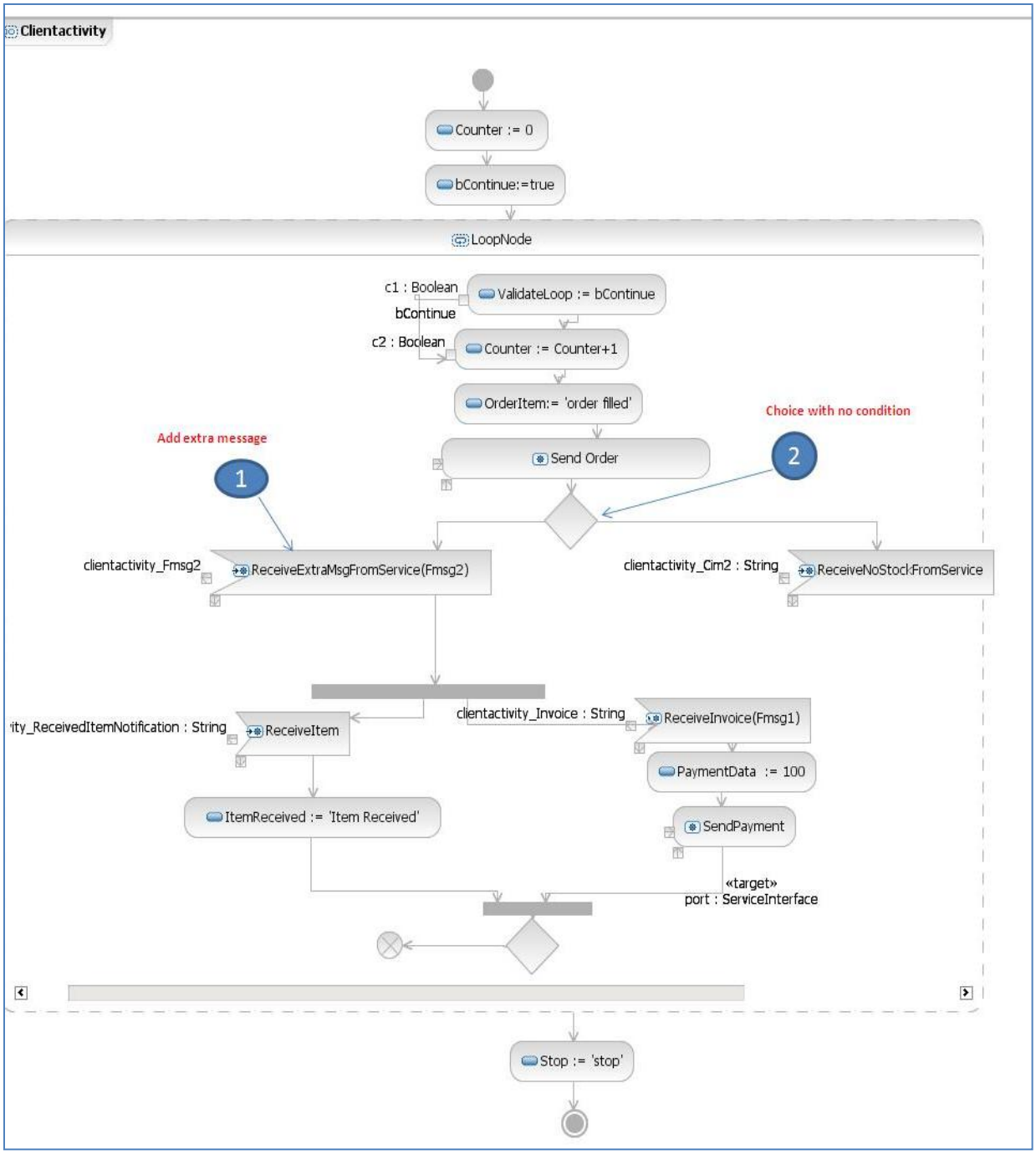


Figure 48: Client Activity Diagram Showing the Choice between a Single and a Concurrent Message Reception

Figure 49: Client BPEL Process with Additional Message Receive Showing the Choice between a Single and a Concurrent Message Reception

B- Converting the Concurrency to Alternatives

It is well-known that concurrency can be implemented by interleaving. Another simple version of the client activity is shown in Figure 50; 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 4.2. The three alternatives of Figure 50 are selected by a <pick> node which is manually inserted. The resulting BPEL process is shown in Figure 51.

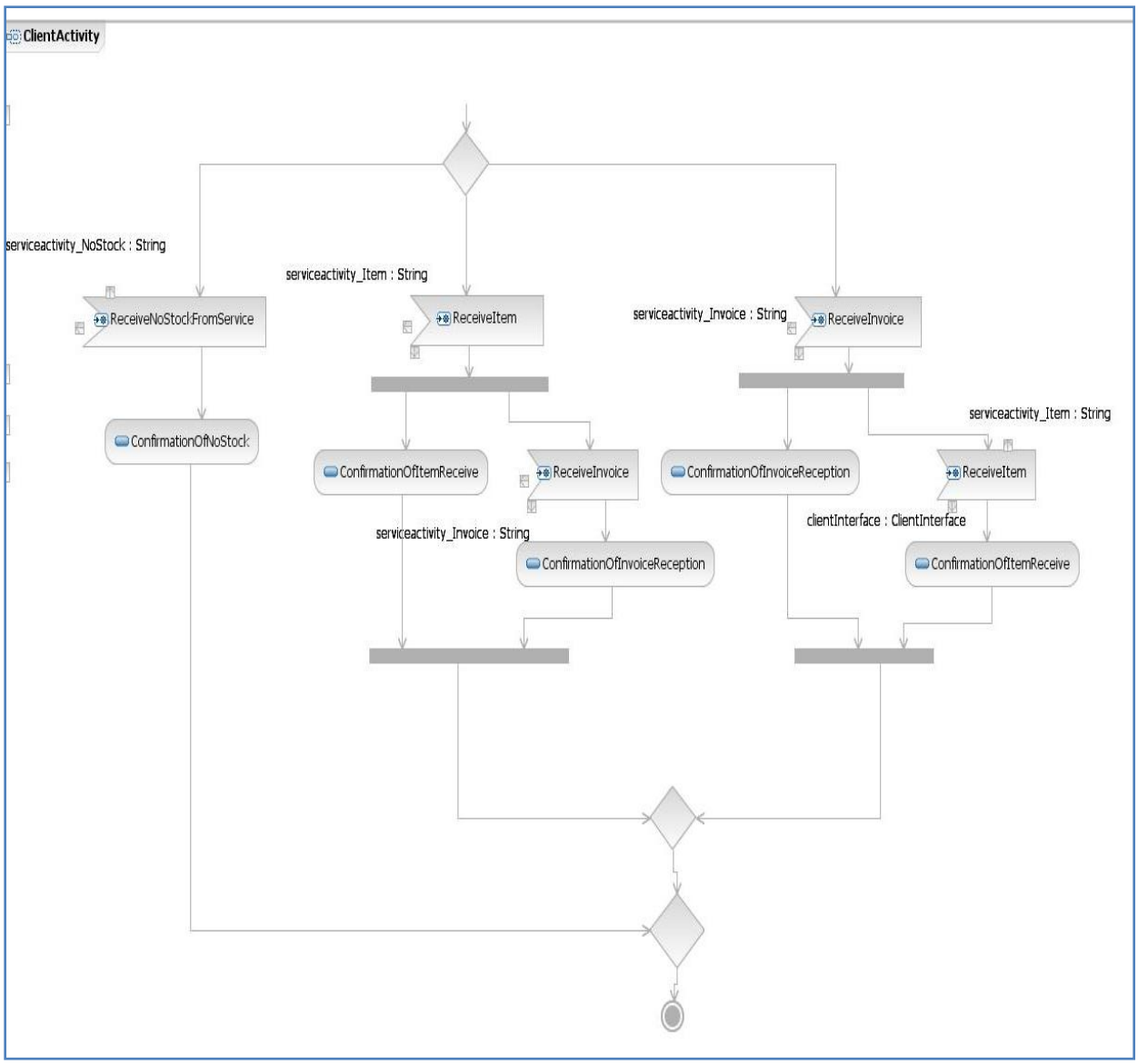


Figure 50: Client Activity Diagram-Simpler Version

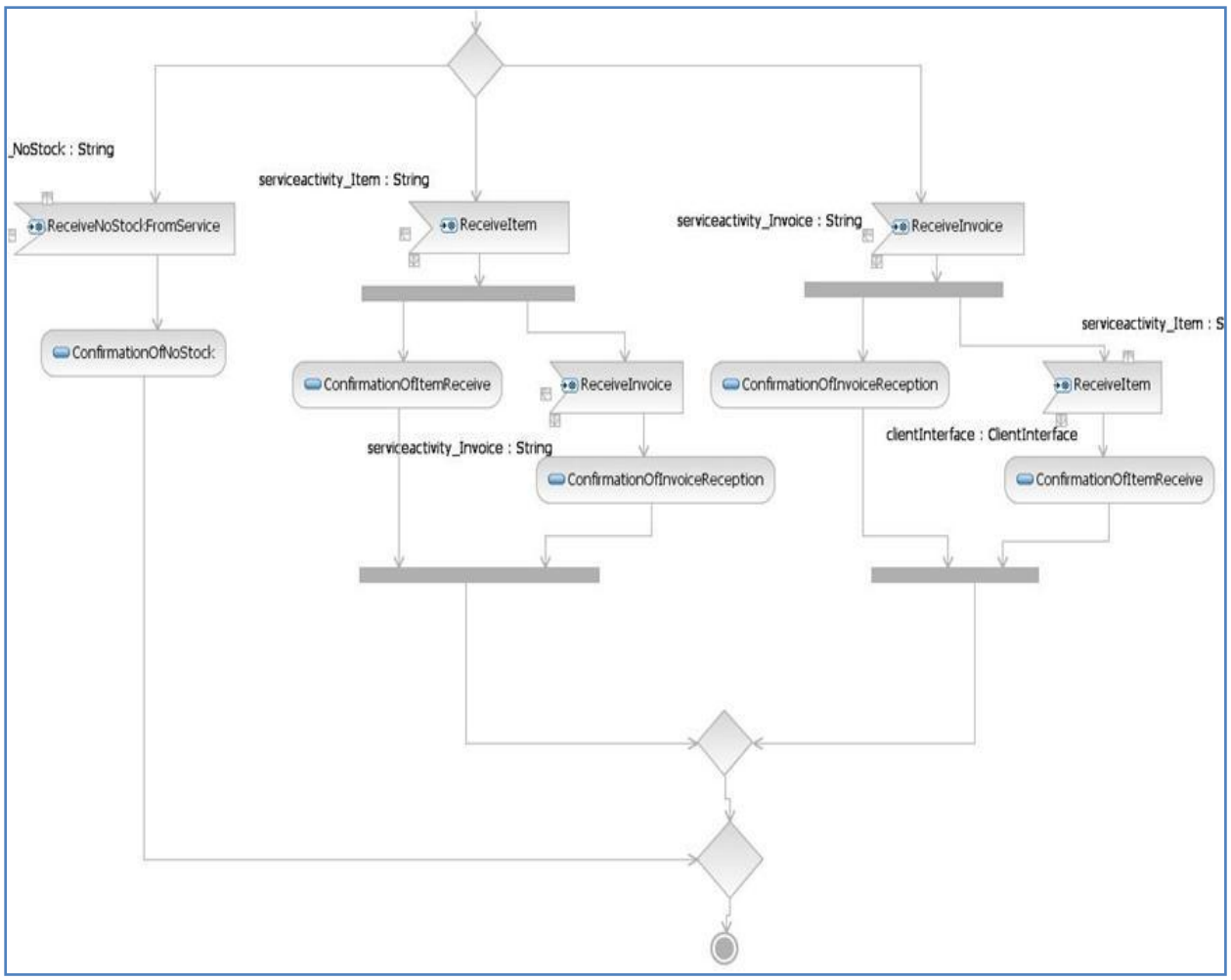


Figure 51: Client BPEL Process Obtained from Figure 50.

During the experimentation of the transformation from Activity Diagrams into BPEL using the IBM WID tool, we found that the tool does not support, within the message reception pool, the distinction of messages by their parameter values, that is, a BPEL process cannot wait for a message of type X with parameter value Y if a message of type X and parameter value Y' ≠ Y has already arrived. Because of this missing feature, we have introduced in Section 5.3.1 an additional complication in the translation process in order to deal with behaviors that include certain types of loops, such as loops with weak sequencing.

6.4 Race Conditions and Weak Loops

The diagram in Figure 2 shows a weak loop scenario. The figure illustrates the repeated execution of the *simulation* collaboration followed by the *details* collaboration. This application involves three parties: the *client*, the *simulator* and the *storage*. The role of the storage component 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). Our assumptions are (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 3.

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 this application, we note that a race condition may occur at the storage component 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 component will return the detailed results of the before-last simulation, which is an error. Moreover, note 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 3 times, as indicated in Figure 3, then the client should have a local counter variable that is incremented during each execution of the loop.

6.4.1 A Proposed Solution for Avoiding Race Conditions in Weak Loops

As mentioned earlier, race conditions can be handled by introducing a message reception pool at each component and letting the destination process determines 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. It is therefore necessary that the different message 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 on the values of certain message parameters.

A proposed solution for handling races is by including in all messages involved a sequence number which indicates how many times the loop has been executed [5]. Each component involved in the loop updates a 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 component will increment its counter before its value is included in the first *SimReq* message, and the *Simulator* component will forward the received parameter value in the parameters of the message sent. As a result, the *Storage* component 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* component. This is shown in Figure 52, where the left

diagram represents the behaviour of the *Client* component, while the right diagram represents the behaviour of the *Storage* component.

This procedure is included in the distributed system designs generated by the tool mentioned in Section 4.2 [4, 5]. 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.

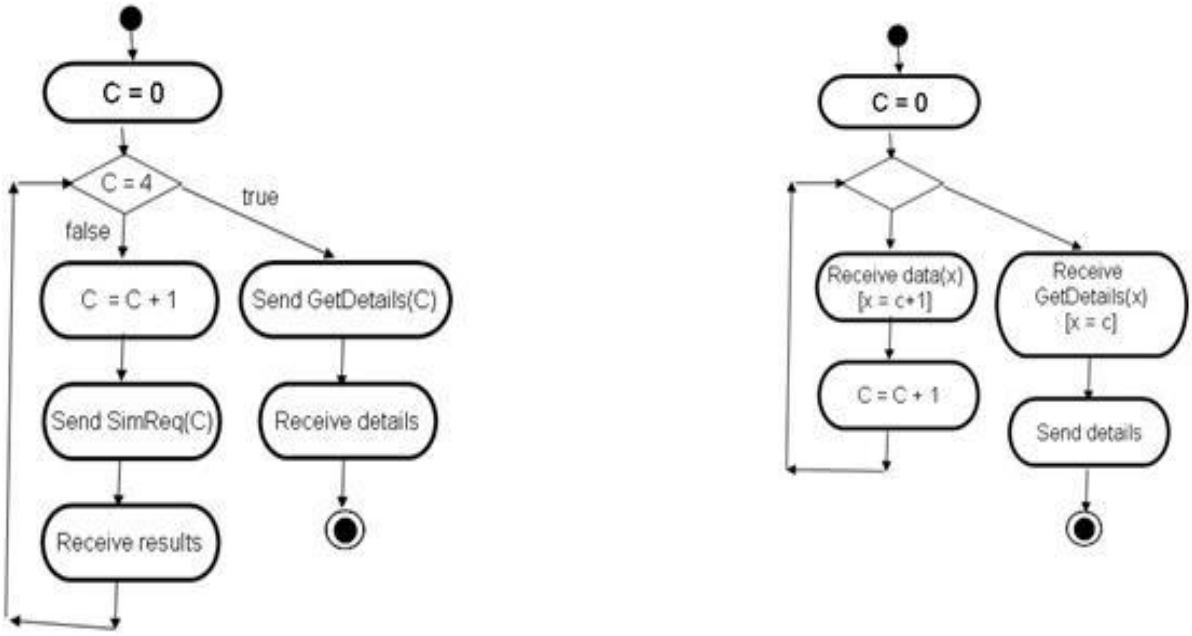
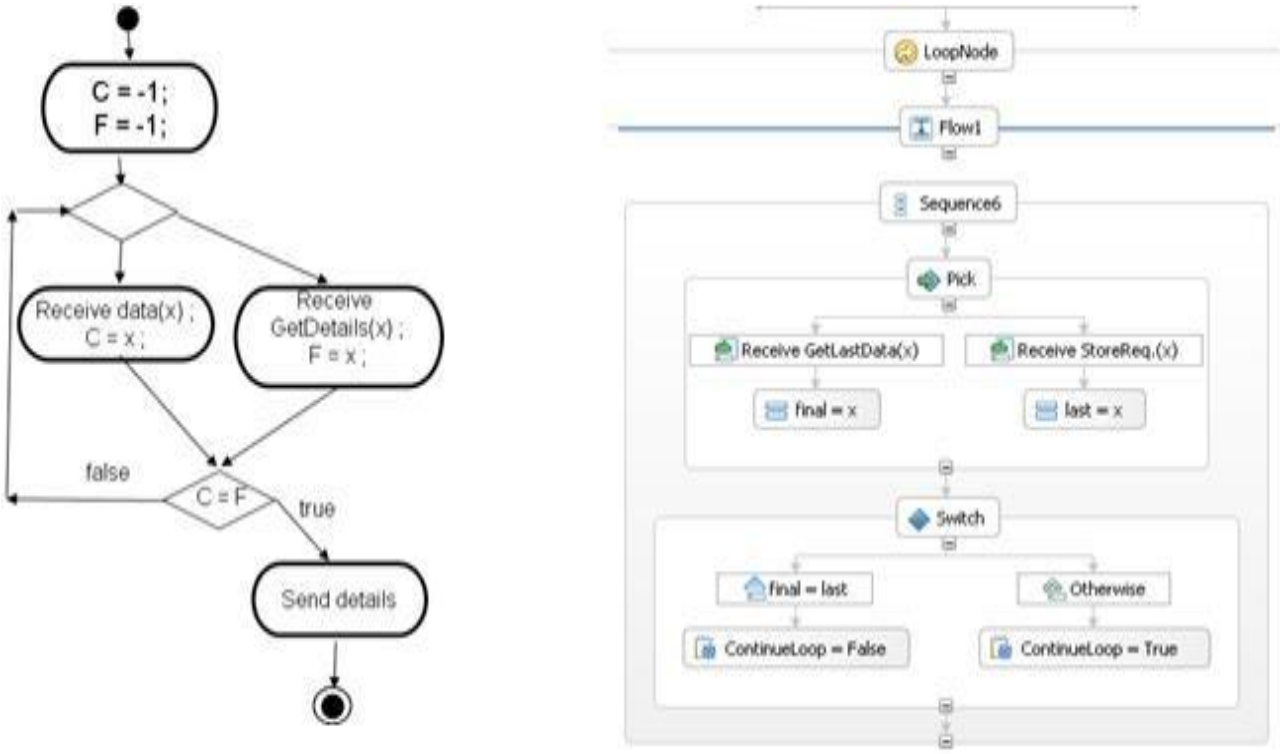


Figure 52: (a) Client Behavior **(b) Storage Behavior**

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 52(b), we propose the behavior pattern shown in Figure 53(a) which deals with the example of the *Storag*e component in our Client-Simulator-Storage example. The component 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. The BPEL behavior obtained for the *Storage* component is shown in Figure 53(b).



(a) Activity diagram **(b) BPEL process**

Figure 53: Behavior of Storage Component

Chapter 7

7 Conclusion and Future Work

This research shows that the behavior of system collaborations can be modeled into a set of communicating system components using the UML activity diagrams. Moreover, it proved that the resulted activity diagrams can be automatically transformed into BPEL processes, Moreover, the integrated processes behavior was realized by coordinating the actions among the BPEL processes through the exchange of asynchronous messages. The UML to BPEL transformation was validated by using two case studies that clarifies how asynchronous messages are coordinated among BPEL processes.

We have experimented with the IBM Rational Software Architect (RSA) which provides a translation from UML Activity Diagrams to one or several BPEL processes. The UML Activity Diagrams input was produced by an earlier research method for deriving the behaviour of several distributed system components from the specification of the global system behaviour using the Eclipse tool.

7.1 Contributions

We investigated the necessary steps and modifications to be performed throughout the transformation process. These steps include the following:

- Modifying the activity diagrams that are generated by the Eclipse tool after applying the derivation algorithm on the global system collaborations.

Providing additional modeling elements that are required for the AD-to-BPEL transformation.We have provided guidelines for performing these steps in order to automate the transformation of activity diagrams into BPEL processes. We have also described these steps in detail for two example applications.

We found that the IBM RSA tool does not support certain important asynchronous message exchange scenarios, and we described how the generated BPEL processes can be manually modified and corrected. Difficulties arise when a UML choice involves several alternative message receptions. In UML activity diagrams, the choice symbol is used to model either a choice that depends on a local condition, or a choice between two messages receptions. In the latter case, the message received first should determine that alternative to be executed, however, the generated BPEL process includes a decision node that depends on a condition. We solve this problem by modifying the generated BPEL code, replacing the BPEL decision node by a BPEL pick activity.

We also proposed a solution to some difficulties that arise in relation with race conditions, especially when there is a loop with weak sequencing. There are often situations of race conditions between the reception of messages from different parties or race conditions between the sending and receiving of messages. We have shown that race condition can be avoided by including a variable that distinguishes the parameter values of the messages that control the execution of a weak loop.

We have also provided a solution for a general UML AD to BPEL transformation for problems that are independent of the RSA tool, such as the case of alternatives that involve the concurrent reception of several messages. We proposed two solutions for this case: (a) adding an additional message when the concurrent alternative is chosen, which gives a simple overall structure, and

(b) converting the concurrency into alternatives. The latter solution suggests that each receive activity is modeled as one alternative followed by concurrent actions.

7.2 Future Work

Some work needs to be completed regarding the UML ADtoBPEL transformation. The IBM RSA tool does not support automatic transformation fully; it needs manual intervention as we have seen for the UML choice activity. It would be interesting to modify the RSA transformation algorithm.

Additional work on the IBM WID environment should consider the support of messages reception based on their parameter values, not only on their message type. This is important because in certain cases a process requires message consumption based on message parameter values, in addition to the distinction of message types.

Finally, the RAS tool could be modified to support the concurrent reception of multiple message types by a single receive activity. There are cases when more than one message type needs to be sent to the same receive activity.

8 References

1. Unified Modeling Langauge (UML). *Object Management Group (OMG).* [Online] 2010. [Cited: 10 13, 2010.] http://www.uml.org/.

2. Model Driven Architecture (MDA). *Object Management Group (OMG).* [Online] 2010. [Cited: 10 13, 2010.] http://www.omg.org/mda/.

3. OASIS Web Services Business Process Execution Language (WSBPEL). *OASIS.* [Online] 2010.[Cited:119,2010.]http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

4. **Bochmann, Gregor V.** *Deriving component designs from global requirements.* Proc. Intern. Workshop on Model Based Architecting and Construction of Embedded Systems (ACES), Toulouse ,2008.

5. **Laamarti, F.** *Derivation of component designs from a global specification.* Master Thesis; University of Ottawa - Canada, 2010.

6. Rational Software Architect. *IBM* [Online] [Cited: 10 13, 2010.] [Online]http://www.ibm.com/developerworks/rational/products/rsa/.

7. Web Services Description Language 1.1. *W3C.* [Online] [Cited: 10 13, 2010.] http://www.w3.org/TR/wsdl.

8. IBM WebSphere Integeration Developer. *IBM.* [Online] [Cited: 10 13, 2010.] http://www-01.ibm.com/software/integration/wid/.

9. **Timothy c. Lethbridge, Robert Laganiere.** *Object Oriented Softwrae Engineering.* Berkshire : McGraw-Hill, 2001.

10. Web Services Description Language (WSDL) Version 2.0 . *W3c.* [Online] 2007. http://www.w3.org/TR/wsdl20/#intro.

11. **W3C.** *Extensible Markup Language (XML) 1.0.* [Online] 11 2008. http://www.xml.com/axml/testaxml.htm.

12. **Ashok Iyengar, Vinod Jessani, Michele Chilanti.** *developerWorks® Series WebSphere Business Integration:* IBM Press, 2007.

13. **Duží, Marie.** *Business Processes Modeling.* Amsterdam : IOS press, 2007.

14. Web Services Business Process Execution Language Version 2.0. *OASIS.* [Online] 8 23, 2006. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html.

15. **Chun Ouyang, Wil M.P. van der Aalst, Dumas.** *Translating BPMN to BPEL.* International Conference on Web Services, pp. 285-292, 2006.

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

17. **Santoro, Nicola.** *Design and Analysis Of Distributed algorithms.*: A JOHN WILEY & SONS, INC., 2006.

18. **Rania Khalaf, William A. Nagy.** Business Process with BPEL4WS: Learning BPEL4WS, Part 6. *IBM.* [Online] [Cited: 11 3, 2010.] http://www.ibm.com/developerworks/webservices/library/ws-bpelcol6/.

19. **Ambühler, Thomas.** UML 2.0 Profile for WS-BPEL with Mapping to WS-BPEL. 10 27, 2005.

20. **W3C, Working Group.** Web Service Architecture. *W3C.* [Online] 2004. [Cited: 10 27, 2010.] http://www.w3.org/TR/ws-arch/#introduction.

21. **Barry, Douglas K.** *Web services and service-oriented architecture*. : Elsevier, 2003.

22. **Alan Brown, Simon Johnston, Kevin Kelly.** Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications. *Rational Software.* [Online] 2002. [Cited: 10 18, 2010.]

23. **Jim Amsden.** Modeling SOA. *IBM.* [Online] 10 2007. [Cited: 11 30, 2010.] http://www.ibm.com/developerworks/rational/library/07/1009_amsden/.

24. **IBM.** *IBM RSA documentaion.* [Online] [Cited: 10 13, 2010.]

25. **Alexandre Alves, BEA and others.** Web Services Business Process Execution Language Version 2.0. *Oasis.* [Online] [Cited: 10 28, 2010.] http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html#_Toc143402870.

26. **Sarang, Poornachandra, Juric, Matjaz and Mathew, Benny.** *Business Process Execution Language for Web Services.* BIRMINGHAM - MUMBAI : Packt Publishing, 2006.

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

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