"BPEL-Mora": Lightweight Embeddable Extensible BPEL Engine

Thilina Gunarathne, Dinesh Premalal, Tharanga Wijethilake , Indika Kumara, and Anushka Kumar

Department of Computer Science and Engineering, University of Moratuwa, Sri Lanka
{thilina.gunarathne,dinesh.premalal,tharanga.wijeithilake, indika.kumara, anushka.kumar}@cse.mrt.ac.lk

Abstract. Web Services have become the de-facto standard for architecting and implementing business collaborations within and across organization boundaries. Web service composition refers to the creation of new (Web) services by combining the functionalities provided by existing ones. A process-oriented language for service composition has been proposed as WSBPEL. WSBPEL specification defines an XML based formal language and provides a general overview of the framework. However no design and implementation issues are described in it. Most of the available BPEL compliant process engines are heavy weight, complex and not extensible. This paper describes the design and implementation of an embeddable, scalable and extensible WSBPEL compliant process engine. This paper highlights the concepts and strategies that were followed during the design and implementation. Primary contribution of this paper is the design of stateless process model and the design of run time core engine using a multi-processor scheduler.

1 Introduction

Service Oriented Architecture (SOA) together with web services have become the de-facto standard for architecting and implementing business collaborations within and across organization boundaries. SOA takes a "software as a service" approach and exposes the functionality of software components as services. These isolated and opaque service components need to be able to collaborate in order to realize more complex functionality. There exists several web service based workflow models such as Web Services Business Processing Execution Language (WSBPEL) [1] in order to cater the above requirement.

WSBPEL is an XML based language that is intended to facilitate the building of more portable business process based on Web Service Description Language (WSDL)[2]. WSBPEL defines how multiple services can be composed together to create new services by combining the functionalities provided by those existing services in a coordinated way. Architecture of a workflow based application typically consists of two programming model abstraction layers denoted by the process model and the individual components. Web services architecture naturally provides the component layer abstraction while WSBPEL provides the process model.

Almost all the available WSBPEL compliant process engines are found to be complex and heavy weigh, while very few are extensible. Objective of BPEL-Mora was to design and implement a lightweight embeddable, extensible WSBPEL compliant engine. BPEL-Mora engine was designed to facilitate service composition, service orchestration, non service orchestrations as well as execution of client side workflows based on WSBPEL model. BPEL-Mora engine is designed to be embeddable in Apache Axis2 [3].

BPEL-Mora consists of four major modules. (1) Process Model (2) Kernel (3) Information model (4) Web service layer. Process Model is used to represent the business process inside the engine. Process model tree for a business process can be created either programmatically or by deploying a WSBPEL document. In order to maintain low memory foot prints we separated out the process model (Meta data about process) and run time state data of process instances. An Information model consisting of a context hierarchy was introduced to store the run time state data of the process instances. Scalability of the engine was achieved by introducing a kernel based on a multi processor, single queue, non pre emptive, priority based scheduler to execute the activities given in a process model. BPEL-Mora kernel was designed to minimize the resource requirement per process instance by avoiding thread proliferation. BPEL-Mora gives empowers user with the ability to write and add custom activities to the engine. This can be achieved very easily by using the by using the provided abstract classes for activities and complex activities. BPEL-Mora is integrated with the Apache Axis2 web services engines through an abstraction layer. Each and every process in BPEL-Mora is registered & exposed as a service through the web services engine. An interface similar to that of WSIF [4] is used for dynamic invocations based on WSDL bindings, with plans to migrate to WSIF later.

Rest of the paper is structured as follows. Section 2 and 3 reviews the background and related work for the subject of this paper. Section 4, 5, 6, 7 and 8 BPEL-Mora design and architecture is discussed deeply with respect to the four major architectural modules. Sections 9 and 10 conclude the discussion with evaluation, conclusion and future work.

2 Background

2.1 WSBPEL

Web Services Business Processing Execution Language– WSBPEL or BPEL for short is a Turing complete XML based programming language that is intended to build more portable business process based on WSDL. WSBPEL is created by merging two existing workflow languages, Microsoft's XLANG[5] and IBM's WSFL (Web Services Flow Language)[6]. Architecture of a workflow based applications typically consist of two layers of programming model abstractions denoted by the process model (also called orchestration layer) and by the individual components. Web services architecture natively provides an abstraction layer which separates out the implementations from the service definitions. This abstraction can be considered as the component layer of the workflow based applications. WSBPEL fits to the web services architecture as the orchestration layer or the process model for web services. BPEL was originally created by BEA, IBM, and Microsoft. Now it is undergoing standardization process at the OASIS consortium.

WSBPEL can be used to define two kinds of processes, namely executable processes and abstract processes. Abstract process is a protocol which specifies message exchange between different parties without revealing the internal behavior of them. Executable process specifies execution order of number of activities.

The building block or each element of a process is known as an activity. An activity can either be a primitive activity or a structured activity. Examples for primitive activities defined in WSBPEL are Invoke,Receive,Wait,Assign, etc. Structured activities are defined in WSBPEL in order to enable the presentation of complex structures by composing the primitive activities. Sequence, Switch, While, Flow are examples for structured activities.

2.2 Apache Axis2

Apache Axis2[3] is a complete re-design and a re-write of the widely used Apache Axis SOAP stack. Apache Axis2 is more efficient, more modular and more XML-oriented than the older version. Apache Axis2 is compliant with most of the new versions of core web services specifications and provides ws* support through its sub projects.

Apache Axis2 supports SOAP 1.1 [7] and SOAP 1.2 [8] and has integrated support for the REST style of Web services too. Hence, the same business logic implementation can offer both a WS-* style interface as well as a REST style interface simultaneously. Axis2 engine is based on a one way message processing model where the engine either perform send or receive functions with respect to a particular SOAP message. Apache Axis2 has the ability to support any Message Exchange Pattern. Axis2 has complete asynchronous messaging support ranging from API level asynchronous support to transport level asynchronous support.

Apache Axis2 is built on Apache Axiom, a new high performing, pull-based XML object model, which provides a simple API for SOAP and XML info-set. Axis2 engine contains a context hierarchy accessible to all services and handlers. All the run time state data are kept in this information model. Apache Axis2 further improves the popular handler architecture introduced by Axis 1.x by adding the concept of phases. In addition Axis2 introduce a concept called Message Receiver[9] which represent a service inside Axis2 and designated as the ultimate recipient of a particular SOAP message from the architecture point of view of the Axis2 engine.

Apache Axis2 is carefully designed to support the easy addition of plug-in "modules" that extend its functionality for features such as security and reliability. Apache Axis2 has a more improved versatile deployment model with support for hot deployment. This deployment model introduces a service archive format and a module archive format for easy deployment of services and modules.

3 Related Work

In this section we'll look at some of the other commercial and open source BPEL implementations along with some research literature.

The ActiveBPEL[10] engine is a widely used open source BPEL engine. It is designed to be deployed as a servlet in a standard servlet container such as Apache Tomcat. Apache Axis1.x[11] Web service engine is embedded internally in ActiveBPEL. ActiveBPEL is designed around the visitor pattern [12]. Active BPEL does not claim to provide a way to add custom activities in addition to BPEL activities.

Interesting study about the scalability of ActiveBPEL engine has been presented in an earlier study [13].According to that ActiveBPEL engine requires two OS threads for the creation of a new BPEL process instance. This shows that when the number of process instances increases in ActiveBPEL, the number of threads may go well beyond what most systems can handle, eventually making the workflow to be aborted. Also users may run in to deadlocks if they try to limit the size of the thread pool of the servlet container [13]. The above study concludes by deciding that the scalability of ActiveBPEL is limited only by the limited hardware resources, which will be not an acceptable remark for an embeddable engine.

IBM WebSphere Process Server (Version 6 as of 2006) is a proprietary BPEL compliant process server running on top of the WebSphere Application Server. WebShpere Process Server is a part of huge software with a wide range of functionality. WebSphere Process Server needs a minimum 1.3 GB (1350 MB) available disk space for installation, installer also requires approximately 600 MB of temporary space during installation and minimum 1 GB physical memory in Linux or Windows platforms [14] as the minimal system requirement.

PXE is another open source BPEL engine. It has many features such as microkernel architecture, pluggable persistency module, JMX-based administration, etc. [15]. There are many other open source and proprietary products like Microsoft BizTalk, Oracle's Business Process Manager, etc which supports BPEL. We wanted to challenge that belief and we wanted to explore new possibilities of BPEL type business processes. In our opinion this is due to the way people have looked at it.

4 Motivation & Our Approach

People tend to think about BPEL complaint business process engines as heavy weight, complex, resource hungry, expensive server side components which are meant to be used by high profile users. On par with the above mind set, all most all the available BPEL engines are found to be complex and heavy weight. But when having a closer look at most existing BPEL engines, we can see that most of them are tightly coupled with business process design modules and business process management modules making them heavy weight and complex. Some of them were built on top of older workflow models making it much worse.

In our opinion the above perception conceal some of the interesting use cases in which BPEL can be used. These use cases can range from service-enabling a device by embedding a BPEL compatible engine to running client side business processes along with custom activities. Our effort is to design and implement a lightweight, embeddable, easy to use BPEL compliant engine as oppose to the above perception. All most all the existing implementations embed web service engines inside the BPEL engine. As oppose to that we thought of developing a BPEL-Mora as a plug-in to an existing web service engine. Following are some of the use cases for such a engine.

Let's consider a simple BPEL use case where a user wants to expose a new web service by combining the functionalities provided by couple of simple web services in a coordinated way. With the currently available tools the user needs to have a bulky BPEL compliant engine installed in his server for this requirement. Our objective is to provide a simple yet powerful BPEL compliant engine as an add-on to a web service engine. Then the user will be able to perform his service composition inside his web service engine with the same simplicity and ease of deploying a web service, with no extra cost or effort. Also if we consider a scenario where a user needs to invoke several web services, then depending on the result he needs to invoke two other services and needs to get the a combined result. In simple words the user needs to do a mash-up. A lightweight BPEL library with a programming API is ideal for such a use case.

A client side application might have a requirement to interact with several web services to produce a result or to execute a workflow. This requirement can be easily & flexibly fulfilled by using a light weight BPEL runtime which can be embedded to the client application. This runtime will be more useful if the developers are given an option to create the process model programmatically using a simple API. Also the developers will become more creative and empowered if they can add custom activities to that run time. One example is a custom activity to take user input in a client side process by showing a dialog box. Another use case is that users need to do non-web service orchestrations at the server side by extending BPEL functionality. One such example would be to send e-mails as part of a business process.

4.2 Design principles

This section articulates some of the principles that have guided our efforts to design a BPEL engine that is light weight and embeddable.

Low Memory footprint. The BPEL-Mora engine should have a very low memory foot print in order to be embeddable. BPEL-Mora engine has deployed processes and instances of those processes running. A single deployed process can have several process instances of itself running. It can even be hundreds of instances per process. Reducing the increase of memory usage per new process instance is one of our main concerns. We achieve this by separating out the run time state data of the process instances and the metadata about the process. Process model representation represents only the process. Once the process is deployed its process model remains unchanged during the run time. All the run time state data are separated out to the context hierarchy which we call as the information model.

Scalability. A deployed business process may contain several numbers of parallel paths. WSBPEL does not impose any restriction for number of parallel paths a process can have. At the same time, there can be several instances of a process running at a given moment. If we take a given moment, there can be *l* number of processes deployed, there can be on average *m* number of parallel paths per deployed process and there can be on average *n* number of process instances per process running in the engine. *l***m***n* gives the total number of parallel paths of execution at that given moment. This *l***m***n* number can easily go up to hundreds. In a typical production environment it might well go up to thousands. This might give rise to thousands of threads if the OS or language threading libraries are used to create separate threads for each and every parallel path. As a solution to this we came up with a software emulated engine using a multi processor scheduler to execute the activities in process instances.

Extensibility. Our objective is to make BPEL-Mora an extensible workflow based service orchestration and composition engine with complete support for WSBPEL and with the ability to support many more. To make BPEL-Mora extensible BPEL-Mora should provide users with the ability to write and include their own activities. Visitor pattern [16] is popularly used in many BPEL engines to provide this extensibility. With the use of visitor pattern all the execution logic goes to one visitor class, making that class huge and unmanageable. Users need to be given access to modify this visitor class in order to add new activities and the user will be directly modifying the most important class of the engine.

Because of those defects, we wanted to avoid the visitor pattern to come up with much more modular, pluggable, component architecture. BPEL-Mora provides two abstract classes, one for simple activities and another one for complex activities, for the users to extend when writing their own custom activities. Users can use their custom written activities in process model by having the newly written activity classes in the class path.Users will be able to share of these custom activities among other users.

4.2 High level architecture

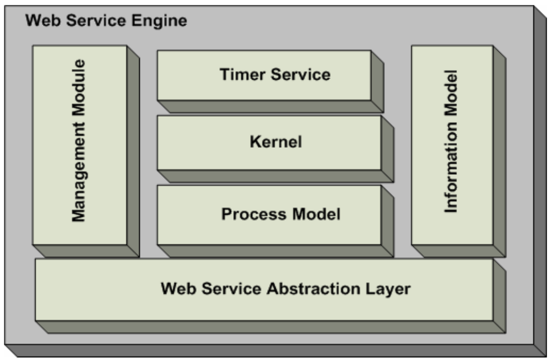

Fig. 1. High Level Architecture of BPEL-Mora

Process Model is used to represent workflows inside the engine. Any workflow that needs to be executed in the BPEL-Mora engine needs to be represented using an instance of an object model. Process model acts as the execution model of the workflow. Process model for a workflow can be created either programmatically or by deploying a BPEL document.

Information model consists of the context hierarchy, which stores the runtime state of the engine and processes in various levels.

Kernel with a multi processor scheduler is introduced to ensure the engine scale without proliferation of threads.

Web service layer consists mainly of BPEL Receive and Invoke activity implementations. BPEL-Mora is built on top of Apache Axis2 web services engine.

Management module provides the functionality to deploy BPEL processes and to do simple management tasks.

Timer service is used by the Wait & Pick WSBPEL activities and for deciding time outs in several queues like in the message buffer of Receive activity.

5 Information Model

Information model is designed to store the run time state of the engine. Four contexts have been introduced to store state data at various levels.

Engine Context holds the run time state data of the engine. This is the top element of the context hierarchy. Engine Context contains a map of all the deployed Process Contexts.

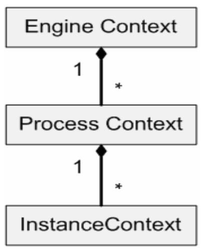

Fig. 2. Context Hierachy

Process Context holds the run time state data of a deployed process. A ProcessContext contains a map of all the top level InstanceContexts of that process. The number of InstanceContexts in this map equals to the number of instances of this process running in the engine.

Instance Context holds the run time state data of a single path of execution. It also holds a pointer called current activity which points to the activity being executing now or to the activity to be executed next. Each process instance running in the engine has a top level InstanceContext which represents that process instance. There can be a tree of InstanceContexts per process instance depending on the number of parallel paths in the process. When encountered a WSBPEL Flow activity BPELMora engine creates new InstanceContexts per each parallel path defined. Flow activity completes execution when all of its parallel paths are completed. On completion of the parallel paths, the original parent InstanceContext continues in the remaining execution path. This parent InstanceContext is used to store the state of links of the respective Flow activity while it's waiting for the completion of the parallel paths. More about handling links is discussed in the section 6.

Scope Context is used to store the data belonging to BPEL Scopes such as values of variables & correlations. InstanceContext always maintains a reference to the scope context of its current scope. WSBPEL uses lexical scoping. A new ScopeContext object is created whenever a ScopeActivity is encountered in an execution. This newly created ScopeContext is made a child of the existing ScopeContext giving rise to a ScopeContext hierarchy as shown in figure 3. BPEL-Mora uses this hierarchy as a search tree for variable & correlation values and fault handlers. A recursive look up of the ScopeContext hierarchy happens when a variable, correlation or a fault handler is referenced by an activity. BPEL-Mora first check whether it is defined in the current scope, if so looks up for that in the current ScopeContext. If it is not defined or found in that context, then the engine looks up for that in the hierarchy until the value of the is found. This scoping context hierarchy provides lexical scoping with the price of a performance penalty due to the recursive lookups. But we can ignore this performance penalty as negligible since scope hierarchies are shallow and simple in most of the BPEL documents.

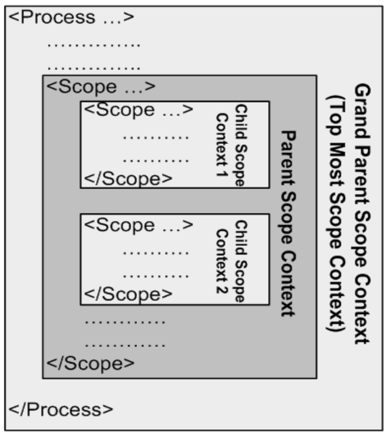

Fig. 3. Lexical Scoping System

6 Process Model

Purpose of the process model is to provide an object model representation of a deployed process capturing the meta-data from the WSBPEL document. We can also call the BPEL-Mora run time process model tree as an execution model. Objects in process model are designed to be run time stateless. Process model tree contains information about the process, but not about the process instances. Process model contains implementation classes corresponding to each and every activity specified in the WSBPEL specification. There are two main categories of activities specified in the WSBPEL specification namely simple activities and structured activities. As shown in figure 4, process model contains abstract classes to capture the common functionality needed for the above two activity categories.

Each and every class corresponding to an Activity contains an "execute()" method which contains the execution logic for that activity. This method takes in an InstanceContext object as the parameter. This InstanceContext object is used to store and retrieve all the state date regarding the process instance. According to the BPEL-Mora architecture implementations of "execute()" method needs to be re-entrant. On other words, the values of the local variables of that Activity object cannot be changed within this method. This gives the ability to share a copy of an activity object among different process instances without worrying about concurrency issues.

A need for return of control to the parent activity arises when implementing several workflow patterns [17] like "sequence", "parallel split" (BPEL Flow) and "while" using BPEL-Mora process model. The method "executeParent()"has been introduced to the "Activity" abstract class to cater to the above requirement. New powerful custom activities can be added to the engine by extending one of the above two abstract classes and using the "executeParent" method to return the control back to parent whenever needed. Users will also be able to share their custom written classes with other BPEL-Mora users.

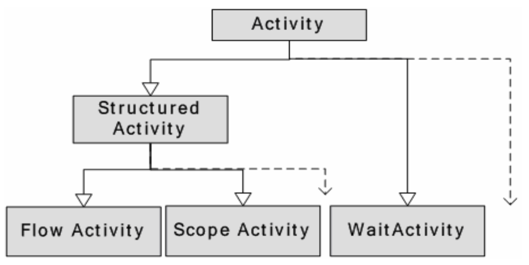

Fig. 4. Class hierarchy of the Process Model

Process model tree is created using a linked list approach. Child activity objects of a structured activity maintain parent-to-child, child-to-parent doubly linked relationship in first and last child with the parent activity object. A single child-to-parent link relationship is maintained in other children. All the siblings maintain a link to the next sibling connecting all the children of a structured activity.

6.1 Handling Links

A BPEL flow activity executes its immediate child activities concurrently giving rise to several parallel paths of execution. Flow activity enables expression of synchronization dependencies between activities that are running on different parallel paths. The link construct is used to express these synchronization dependencies. Links of a flow activity are separately defined inside the flow activity. Exactly one activity can declare to be the source and other one activity can declare to be the target of a link. We say X has a *synchronization dependency* on Y, if activity X is the target of a link that has activity Y as the source.

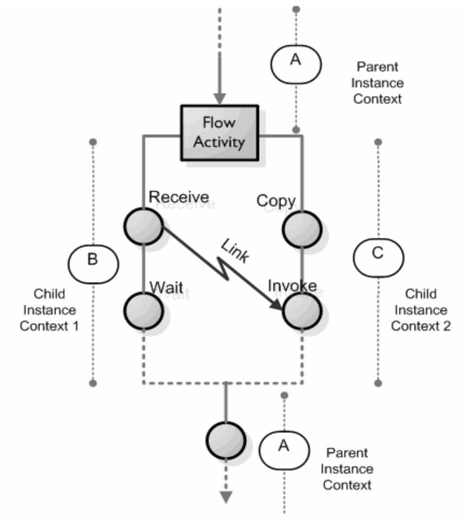

Fig. 5. Links

As discussed in section 4 when met with a flow BPEL activity, BPEL-Mora creates new child instance contexts for each parallel path, while the parent instance context waits till execution is complete in all the parallel paths. BPEL-Mora uses this parent instance context to store a list of “Link” objects whenever a Flow activity with defined links is executed. These “Link” objects can be in “true” state, “false” state or “not evaluated” state depending on the state of the source activity of that link. When the “Link” object status is “not evaluated” target activity has to wait till the “Link” state is evaluated. In BPEL-Mora implementation an Instance Context executing in a parallel path target Activity can register with a “Link” object in “not evaluated” state to be notified when the link is evaluated.

7 Kernel

A kernel with a multiprocessor scheduler is introduced to BPEL-Mora in order to ensure the engine scale without proliferation of threads. Following sections discuss about the scheduling of process instances and the life cycle of process instances.

7.1 Scheduling BPEL activities

A multi processor scheduler with a configurable number of processors is implemented in the BPEL-Mora kernel. In here normal java Threads in a thread pool were used to emulate the processors. Each worker thread in the thread pool simulates one processor. The decisions that we had to take with regards to the scheduler were (1) Unit of execution (2) Scheduling policy (3) Number of worker threads (4) Number of scheduling queues.

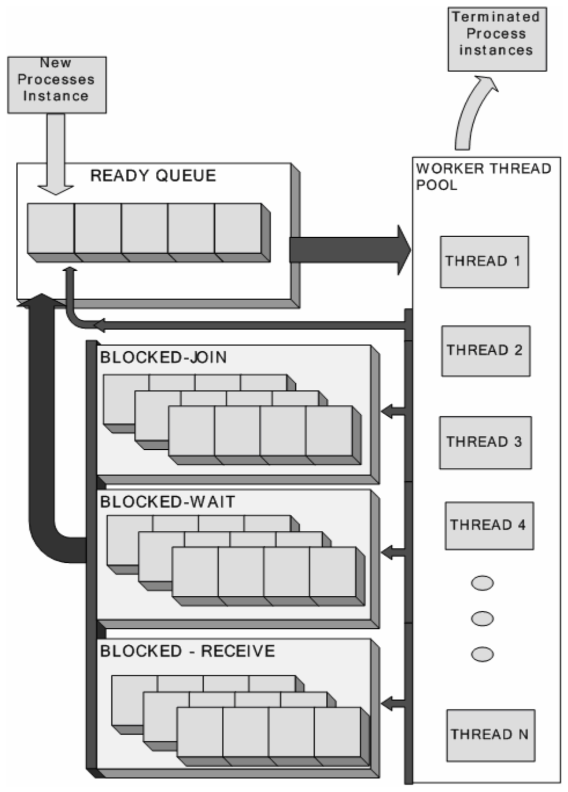

Fig. 6. BPEL-Mora Kernel. A worker thread picks an Instance Context object (represents a process instance) from the head of the scheduler queue and executes the current activity of the process instance. Then, depending on the resulting state the process instance is put into the relevant queue.

A single activity in a process model is chosen as the unit of execution for simplicity and clarity. All the currently supported activities including all the WSBPEL activities were designed to have a limited number of instructions per execution. All the current activities were carefully designed not to block the worker threads during the execution. Examples for this behavior are Receive, Invoke and Wait activity implementations. When a Receive or a Wait activity is executed, the Instance Context object belonging to the execution will be put in to the appropriate waiting queue freeing the worker thread. A call back object is used to store the Instance Context object in the case of Invoke activity.

With the above design an assumption can be made that “a BPEL-Mora activity will be executed in a predictable small bounded time period”. With this assumption, a non-pre-emptive priority based scheduling policy is used in the BPEL-Mora scheduler.

BPEL-Mora run time does not enforce any time constraint for the duration of execution of an activity. Priority for a process can be specified at the deployment time. Above assumption invalidates if users add custom activities that take longer times to execute.

Number of worker threads in the scheduler thread pool is made configurable to cater for the various underlying resource requirements. As an example, a server with parallel processors can gain advantage by increasing the number of threads while a single processor pc can harness the best by having a small number of threads like 5 threads.

Currently, BPEL-Mora scheduler uses a single scheduler queue assuming the context switch time is very small relative to the time taken for a unit of execution. Another option is to have a scheduler queue for each and every worker thread. An implementation like that can reduce the context switching time. On the other hand it'll unnecessarily increase the complexity of the scheduler due to the need to perform queue load balancing. The scheduler queue implementation needs to be blocking and thread safe. Hence, we have chosen a PriorityBlockingQueue[18] as our scheduling queue implementation.

7.2 Process instance life cycle

Process instances are created with the reception of a designated "startable" invocation message and are destroyed when the last activity of the process instance completes execution. Between those two we can define several more states with regards to the scheduler. Process instances may have parallel execution paths. These parallel execution paths can be in different states at a given time. Because of this it makes more sense to discuss about process instance life cycles with regard to a single execution path, which will be referred to as "single path of execution" here after. These single paths of execution are represented inside the engine by the instance context objects.

Four major states can be identified in a single path of execution of a process instance. They are (1) ready (2) running (3) blocked (4) terminated. All the paths of execution in the "ready" state are waiting in the scheduler's queue. New process instance entering the engine are initially in the "ready" state. A process instance is in the "running" state when it is executing inside the engine. A single path of execution terminates either when the process instance terminates or when it finishes executing the last activity in its execution path. A single path of execution represented by an instance context enters in to the blocked state in 3 ways.

(1)A single path of execution represented by an instance context enters into "Blocked-Join" state when it is waiting for a "link" to be evaluated. Instance context waits till the link gets evaluated. An instance context in "Blocked-Join" state moves to the "Ready" state upon successful evaluation of the "link".

(2) A single path of execution represented by an instance context enters into a "Blocked-Wait" state when a "wait" BPEL activity is executed. An instance context in "Blocked-Wait" state moves to the "Ready" state upon reaching of the given deadline or upon expiration of the specified time period.

(3) A single path of execution represented by an instance context enters in to a "Blocked-Receive" state when a "receive" BPEL activity is executed as well as a syn-

chronous Invoke activity is executed. An instance context in “Blocked-Receive” state moves to the “Ready” state upon receiving the expected message.

8 Interfacing with web service engine

BPEL-Mora is built on top of Apache Axis2. By the use of Axis2 BPEL-Mora takes lot of features for granted such as performance, ability to do REST style of web services, asynchronous support, WS* capabilities through Axis2 modules, etc. Interfacing with web service layer is done through the implementations of invoke and receive BPEL activities.

8.1 Providing web service operations

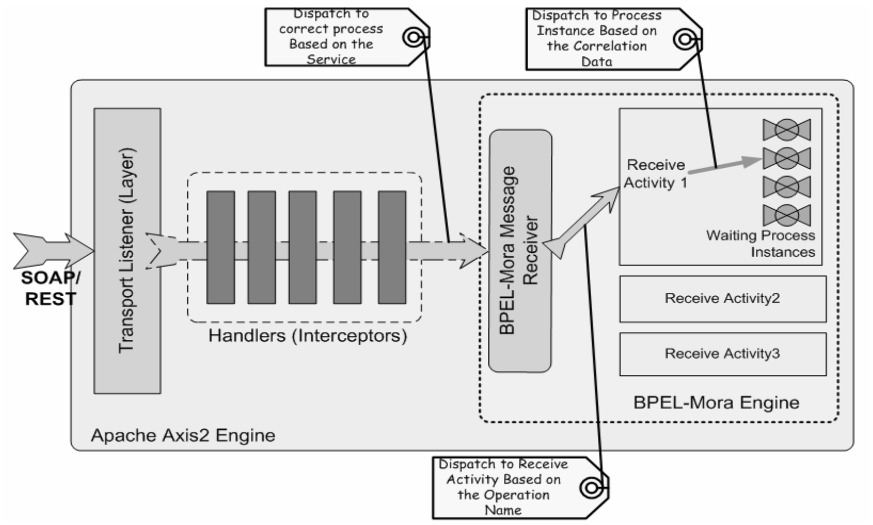
Dispatch to correct process Based on the Service
Dispatch to Process Instance Based on the Correlation Data
SOAP/ REST
Transport Listener (Layer)
Handlers (Interceptors)
BPEL-Mora Message Receiver
Receive Activity 1
Waiting Process Instances
Receive Activity2
Receive Activity3
BPEL-Mora Engine
Apache Axis2 Engine
Dispatch to Receive Activity Based on the Operation Name

Fig. 7. Apache Axis2 integration with the receive activity.

Each and every process in BPEL-Mora is registered as a service with the web service engine. Hence, we expect the web service engine to route the messages to the correct service, which in BPEL-Mora scenario is to route the message to the correct process. Web service layer in BPEL-Mora consists of a MessageReceiver implementation. MessageReceiver is an interface provided by Axis2, to use per service basis. Axis2 delivers the incoming messages to particular service to the specified MessageReceiver[9] implementation. BPEL-Mora MessageReceiver of a deployed process maintains a map of references to receive activity objects in a process against their operation names. BPEL-Mora uses this table to route messages to the correct receive activity object within the deployed process based on their operation name.

Each and every Receive Activity object maintains a queue of InstanceContext objects (process instances) blocked by waiting at that activity. Following sequence is followed when a message is received to a receive activity. (1) If the message receive activity object is designated as “startable”, then it’ll create a new instance of the process. (2)Otherwise the message is routed to the correct process instance in the queue based on the correlation data [1] in the message. (3) If a matching instance is not found in the queue, the message is buffered in a separate queue for a certain time period waiting for a matching process instance.

8.2 Invoking web service operations

Invoking of external web services is done through an interface similar to WSIF [4], This interface supports the creation of dynamic clients based on the WSDL binding.

The web service invocation interface is an implementation of Adapter Design pattern [19].Invocation model wraps Axis2 [3] client programming API and provides a Dynamic Invocation Interface (DII).BPEL-Mora web service invocation supports DII with or with out the WSDL. It is mandatory to provide the end point reference of the target service, in the case where the WSDL is not available,

Invoke Activity implementation handles all the request-response type invocations through the Axis2 client side non-blocking invocation mechanism. A special callback handler object containing the InstanceContext object corresponding to the process instance is used to receive the response.

9 Evaluation

Two tests were done to measure the performance of BPEL-Mora.

First test focuses on measuring the scalability of BPEL-Mora kernel and the scheduler. We used an embedded BPEL-Mora engine inside a test case and programmatically created and deployed the process. The process contained a flow activity inside a sequence activity. A custom activity which simply prints out its number and the execution count with a small time consuming logic was used as the children of the flow activity. This process was triggered programmatically.

Table 1. Scalability of the scheduler (figures are the average of 5 runs)

Number of children for the Flow activity	Time takes to execute (ms)	Avg. Time per 100 children (ms)
100	745	745
500	2064	412.8
1000	3443	344.3
2000	7130	356.5
5000	15701	314.12
10000	29642	296.42
15000	41578	277.18

These tests shows scalability of the engine and the fact that number of parallel paths and the overhead of creating InstanceContext object per each path do not affect the performance.

A second test was focused on the memory foot print of the engine. BPEL-Mora was deployed inside Apache Axis2 1.1 running inside Apache Tomcat 5.0.28 with jdk 1.4.2. A 25 kb BPEL document was deployed in BPEL-Mora. This process was designed to go in to Blocked-WAIT state as soon as the process instance was created.

Table 2. Memory usage (MB) Vs No. Of process instances

Process Instances	1	100	200	300	400	500	600	700
BPEL-Mora Mem. Usage	2.4	10.4	25	33	41.5	48.4	61.4	66
ActiveBPEL Mem. Usage	2.6	37.3	*Reached a Thread limitation*					

BPEL-Mora implementation followed a minimalist approach from day one. As a result of that the size of BPEL-Mora library remains less than 130kB. BPEL-Mora depends only on the Axis2 and its dependent libraries. Due to that adding WSBPEL capability to an existing Axis2 server can be done with the mere addition of 130 KB BPEL-Mora library. To embed BPEL-Mora in an application or to run it standalone requires the addition of Axis2 and dependent libraries, which are of size 2.8 MB.

10 Conclusion and future work

BPEL-Mora is a lightweight embeddable extensible WSBPEL compliant process engine. BPEL-Mora can be embedded into the web service engine to execute server side processes. BPEL-Mora has the capability to serve as a process run time to execute client side processes. In this paper, we presented the motivation behind our effort, discussed the architecture of BPEL-Mora engine and major design decisions we took in implementing BPEL-Mora. Affect of issues related to scalability, extensibility and memory foot print, to the embeddability of the engine was also addressed in this paper. Information model captures the run time state data of the process instances and manages the lexical scoping of variables. Architecture of the stateless object model was discussed focusing on extensibility and memory foot print. Architecture of the Runtime engine with its scheduler was discussed along with the various decisions we had to take during the implementation of the scheduler.

Providing full WSBPEL capability including fault handling and event based constructs together with improving the programming API to ease the programmatic creation of processes can be seen our immediate future objective. WSBPEL specification does not define how WS-Transactions [22, 23, 24] set of specifications can be used to provide transaction capability for WSBPEL processes. Adding transactions support for business processes using WS-Transactions family of specifications will be one of our future research goals.

Acknowledgements.
We thank our Project Supervisors Sanjiva Weerawarana & Sanath Jayasena for many insights and discussions. We thank our Project Coordinator Shantha Fernando and Vishaka Nanayakkara for their their support throughout the project.

References

1. OASIS WS-BPEL Technical Committee. Web Services Business Process Execution Language Version 2.0, Working Draft 01, December 2004. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.
2. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL), Version 1.1, March 2000. http://www.w3.org/TR/wsdl.
3. Web Services – Apache Axis2 , June 2006, http://ws.apache.org/axis2.
4. Web Services–Web Services Invocation Frame work (WSIF), June 2006 , http://ws.apache.org/wsif.
5. Thatte, S. , 'XLANG: Web Services for Business Process Design' ,Technical report, Microsoft, 2001.
6. Leymann, F. , 'Web Services Flow Language', Technical report, IBM, 2001.
7. D. Box et al, Simple Object Access Protocol (SOAP)1.1, May 2000 . http://www.w3.org/TR/SOAP.
8. M. Gudwin et al, Simple Object Access Protocol (SOAP)1.2, May 2000 . http://www.w3.org/TR/soap12-part1/
9. Apache Axis2, Architecture Guide . http://ws.apache.org/axis2/1_1/Axis2ArchitectureGuide.html
10. Active BPEL, June 2006, http://www.activebpel.org
11. Web Services – Apache Axis1.x , June 2006, http://ws.apache.org/axis.
12. ActiveBPEL Engine Architecture, July 2006 , http://www.activebpel.org/docs/architecture.html
13. Wolfgang Emmerich, Ben Butchart, Liang Chen and Bruno Wassermann, Grid Service Orchestration using the Business Process Execution Language (BPEL), October 2005. (pp. 28-30), http://sse.cs.ucl.ac.uk/omii-bpel/publications/bpel.pdf
14. WebSphere Process Server Version 6.0 System Requirements, July 2006. http://www-306.ibm.com/software/integration/wps/sysreqs/
15. FiveSight PXE, June 2006, http://pxe.fivesight.com/
16. J. Palsberg and C. B. Jay. The Essence of the Visitor Pattern. In *Proceedings of COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, pages 9{15, Vienna, Austria, August 1998. http://www.cs.ucla.edu/~palsberg/paper/compsac98.pdf.
17. Workflow patterns, June 2006, http://is.tm.tue.nl/research/patterns/patterns.htm
18. J2SE 5.0, Concurrency Utilities, June 2006 , http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html#concurrency
19. E. Gamma, R. Helm, R. Johnson, and J. Vlissides.*Design Patterns*. Addison-Wesley Pub Co, January 1995. ISBN 0201633612.
22. F. Curbera et al. Web Services Coordination (WS-Coordination),Version 1.0, August 2005.
23. F. Curbera et al. Web Services Atomic Transaction (WS-AtomicTransaction),Version 1.0, August 2005.
24. F. Curbera et al.Web Services Business Activity Framework (WS-Business Activity),Version1.0, August 2005.