Monitoring and execution for contract compliance

Martin Sailer[1] and Michal Morciniec
E-Service Markets Department
HP Laboratories Bristol

HPL-2001-261 (R.1)
10th October, 2005*

electronic commerce; e-contract; business processes

Real world interactions between businesses are governed by contracts that define rights and obligations of parties. Electronic contract structures the information contained in textual contract so that it can be used to automate certain aspects of contracting process.
In this report we focus on the contract fulfilment stage of this process. A simple electronic contract structure expressed as an XML document is introduced that allows one to define obligations on contract parties to perform actions. The contract actions are implemented as public collaboration patterns that in turn are implemented by each of the party as business processes that lead to message based interactions.
We describe a system that can process an electronic contract and when an obligation becomes due determines the relevant collaboration pattern and a corresponding business process. It then monitors the execution of the process and messages transmitted to a business partner. The information collected by the system can later be used in contract compliance decision-making, i.e. determining whether an obligation has been carried out successfully.

---

[1] Martin Sailer, Institut für Informatik, Technische Universität München, München, Germany
[*] Internal Accession Date Only

© Copyright 2005 Hewlett-Packard Development Company, L.P. Approved for External Publication

1 Introduction 2
2 E-Contract and Collaborations 2
2.1 E-Contract Lifecycle 3
2.2 E-Contract 3
2.3 Collaboration 6
2.3.1 Collaboration implementation with business processes 7
2.3.2 E-Contract considerations for collaboration implementations 9
3 Requirements 10
4 Design of Contract Execution Framework 11
4.1 Creation of Adaptors 12
4.2 Creation of Bindings 12
4.3 Execution of Statement 13
5 Use of Process Manager 14
5.1 Main Concepts 14
5.2 Java API 16
5.3 Design of the Adaptor 18
5.4 Monitoring 19
5.4.1 Process Monitoring 19
5.4.2 Activity Monitoring 20
5.5 Simulation of Process execution 21
5.6 Visualization of Contract Execution 23
5.7 Recommendation and Desired Features 24
6 References 25

1 Introduction

The work described in this paper was done on a 3-month industrial placement sponsored by HP Labs Bristol. It has been realized by cooperation between the chair of the Technical University of Munich for network and systems management and HP Labs Bristol

The placement was conducted at the HP Labs Bristol in context of a practical work (*Systementwicklungsprojekt*), which is part of the *Hauptdiplom* (equivalent to Master Of Science).

2 E-Contract and Collaborations

Recently there has been renewed interest in modelling of business contracts in the academic computer science community [Milosevic 2001] as well as in the industry [Ibbotson J., Sachs M., 1999]. This is motivated by the fact that enterprises increasingly use the Internet for communication with their partners and would like to leverage this technology in order to gain efficiency in contracting processes.

Contracts are important in the context of loosely coupled structures [Marshall, 1999] like supply chains that involve independent entities. Because there is no central authority that coordinates activities of entities making up a supply chain, each entity is responsible to arrange a contract with their partner defining the collaboration in which they will engage.

In real life, contracts define rights and obligations of parties as well as conditions under which they arise and become discharged. The rights and obligations concern either states of the affairs or actions that should be carried out. Often contracts also specify secondary (reparation) obligations that come into force when a party does not carry out an obligation. The essence of contracts is the definition of commitment *states* that is imposed on contracting parties. These states come into force and become discharged as a result of actions that the parties carry out or as a result of an occurrence of an external event such as expiration of a deadline.

During the contract fulfilment, parties collaborate by exchanging information and carrying out actions that have been agreed. The collaboration only occurs because of the commitments defined within the contract. In normal circumstances parties aim to fulfil their responsibilities but it is perfectly admissible that a party will refuse to carry out an agreed action or refuse to maintain agreed state of affairs activating a secondary (reparation) obligation. This situation typically occurs if an unforeseen event takes place (e.g., import restrictions) forcing one to de-commit from the obligation.

So far, contracts have usually been treated merely as text documents. However, enriching contracts with well-structured information such as conditions under which a commitment becomes due or conditions under which it is deemed to be fulfilled allows for automation of certain aspects of the contracting process. Thus, the resulting e-contract can provide a high degree of consistency in contract performance and lead to improvement in contract management.

2.1 E-Contract Lifecycle

Conceptually, the lifecycle can be split into the three stages of contract drafting, formation, and execution [ECS 2000]. Figure 1 illustrates the contract lifecycle.

Contract drafting phase: Given the contract template model, the drafter role constructs an instance of the template. In this phase the contractual roles, abstract business interactions and contractual situations are specified. The template typically has a number of free variables that are agreed upon in the next phase.

Contract formation phase: Participants assume contract roles and negotiate the details of their responsibilities. The negotiable variables of the contract (deadlines, order of actions) become fixed and concrete business interactions are bound to the abstract ones defined in the template. The relationships between contract parties are created and captured in contract statements. The statements contain policy expressions that imply obligations and rights of parties.

Contract execution phase: Actual delivery of contract consideration takes place. Typically, this phase constitutes service or goods delivery, invoicing, bill calculation, presentment and payment. The interactions between the parties are monitored for their compliance to the terms agreed on in the contract.

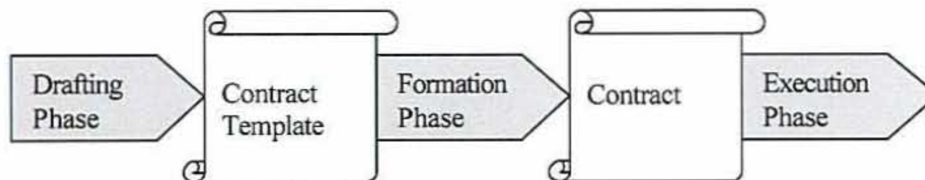


Figure 1 – Contract lifecycle

2.2 E-Contract

An electronic contract (e-contract) is a formalization of real life contracts (informal contract or contract without machine-readable format) expressed in natural language and typically structured into a set of clauses. General requirements for an e-contract model can be derived from a study of informal contracts. Our model is based on the on-going analysis and study of the *INCOTERMS*[1] international contract of sale [Ramberg 2000]. Our modelling approach in based on the *stit* (see to it that) statements that [Daskalopulu 2000] can be expressed more formally as modal logic. This role-based model allows for a specification of the commitments of different types as well as expression of delegation [Norman 2000] of these commitments.

The e-contract is a declarative specification captured in XML. Its syntax can be validated with a corresponding XML Schema. This specification can then be loaded

[1] Incoterms are international rules for the interpretation of the most commonly used terms in international trade

into a contract platform that provides contract management capabilities as well as means of contract fulfilment. Below a subset of the model that is relevant to the subject of this report is presented.

The property that is immediately apparent from studying contracts is that it is parameterised by contract roles. In *INCOTERMS*, the roles are *Buyer* and *Seller*. When the contract is signed, the entities assume the specific roles and obligations that correspond to them. Furthermore, each signed contract has a globally unique reference that can be used in contractual communication. This can be captured by the following XML and Schema fragment (Figure 2).

```
<contract contractId="223" xmlns="http://www.hp.com/contract"
xmlns:act="http://www.hp.com/action" xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-
instance" xsi:schemaLocation="http://www.hp.com/contract
C:\Contract\src\test\Contract.xsd">
	<header>
		<preamble>International Sale of Goods Contract</preamble>
		<role name="Buyer"/>
		<role name="Seller"/>
		<partyInfo roleNameRef="Seller">Hewlett-Packard</partyInfo>
		<partyInfo roleNameRef="Buyer">British Library</partyInfo>
	</header>[...]
</contract>
...
```

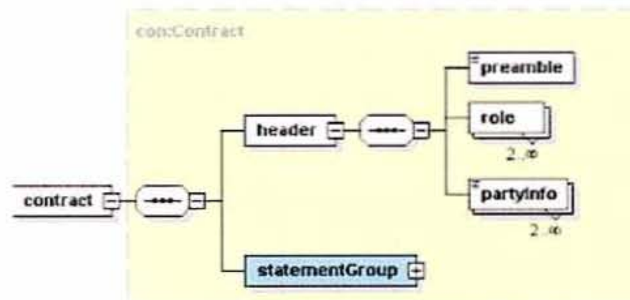
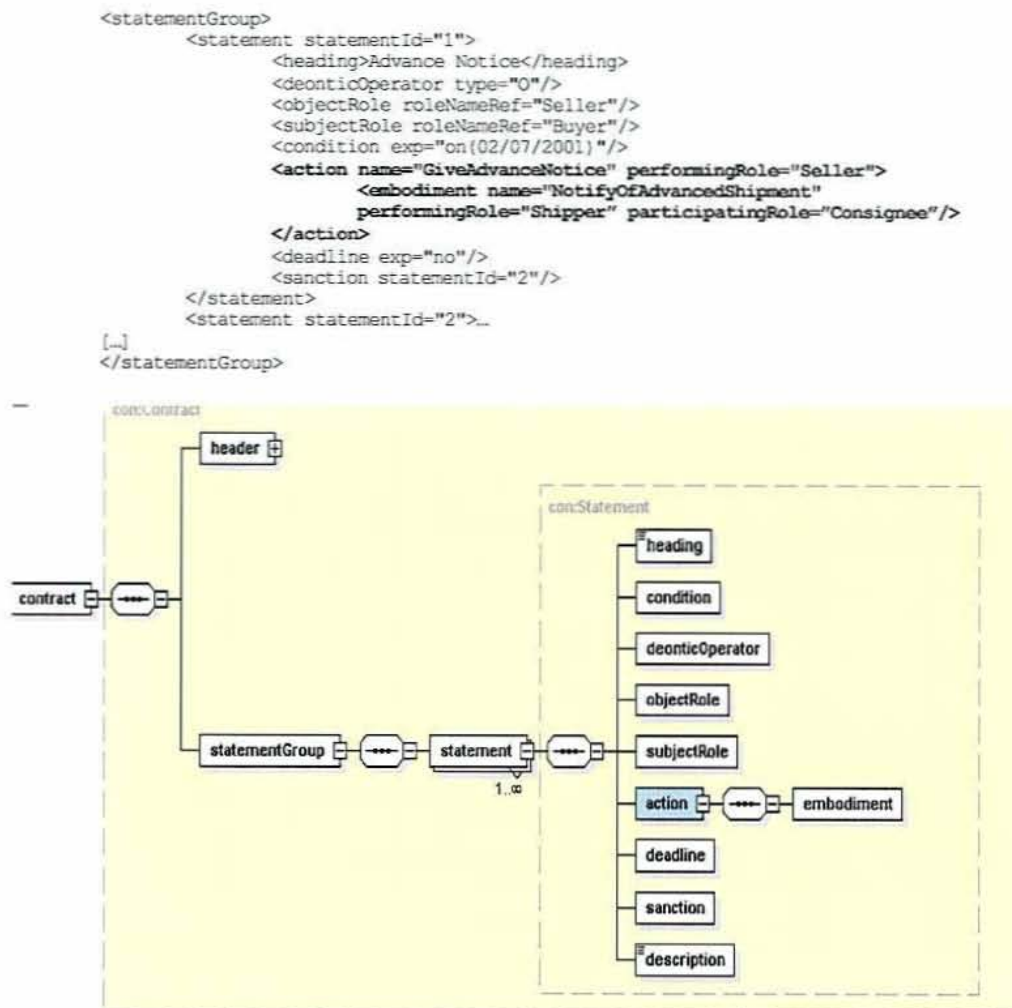


Figure 2 – reference to contract roles

The `<partyInfo>` element is inserted when the entities (Hewlett-Packard and British Library) sign the contract and it refers to a specific contract role. The party information contains further details such as data necessary for the electronic communication (e.g., e-mail address, host or IP address and port, etc.) but is not shown here.

The *INCOTERMS* contain statements of obligations that mostly concern actions that a role should carry out. One of the important obligations of the *Seller* is that he has to give sufficient advance notice of the delivery that the *Buyer* paid for. This is so that *Buyer* can arrange for means to unload the goods and transport them to his premises. In the e-contract the normative states are defined in the `<statement>` elements enclosed within the statement group. The model for a statement described detailed a condition upon which the normative state will be activated (here on the 2 July 2001). The type of the normative state is indicated by the `<deonticOperator>` element. The allowed types are obligation, permission and prohibition. The statements has two role elements `<objectRole>` and `<subjectRole>`, specifying that the holder of the obligation is the *Seller* and the beneficiary of it is *Buyer*. The `<deadline>` and `<sanction>` elements are optional and specify the deadline by which the normative state must be discharged and the secondary obligation that will be activated in case of no performing.



```
<statementGroup>
        <statement statementId="1">
                <heading>Advance Notice</heading>
                <deonticOperator type="O"/>
                <objectRole roleNameRef="Seller"/>
                <subjectRole roleNameRef="Buyer"/>
                <condition exp="on(02/07/2001)"/>
                <action name="GiveAdvanceNotice" performingRole="Seller">
                        <embodiment name="NotifyOfAdvancedShipment"
                        performingRole="Shipper" participatingRole="Consignee"/>
                </action>
                <deadline exp="no"/>
                <sanction statementId="2"/>
        </statement>
        <statement statementId="2">…
[…]
</statementGroup>
```


Figure 3 – statement of obligation

Finally, this normative state definition requires the *Seller* to perform contract action *GiveAdvanceNotice*. Furthermore, because the parties will communicate through the Internet, the contract action of giving advance notice will be carried out according to an agreed collaboration pattern *NotifyOfAdvancedShipment*. The entire statement defined above can be read as the following:

"*On 2 July 2001* the *Seller* shall be *obliged* to the *Buyer* to see to it that *Seller* carries out action *GiveAdvanceNotice* according to collaboration *NotifyOfAdvancedShipment*".

The e-contract represented by the XML document is loaded into the contract platform and processed. As a result of this processing a number of objects is created that are

pertinent to contract execution and monitoring. The obligation object registers with the calendar and will become pending on 2 July. The enterprise entity (human or automatic procedure) playing the Seller role will then decide whether to fulfil the obligation or not. If it agrees the collaboration pattern *NotifyOfAdvancedShipment* will be realised by an exchange of electronic messages.

The Execution and Monitoring framework is responsible for realizing this collaboration, monitoring it and notifying other components about its state so that when the collaboration completes the normative state of obligation can be discharged.

2.3 Collaboration

In the previous section, the idea of an electronic contract containing statements that define commitment states concerned with action performance has been introduced. It was mentioned that contracts are parameterised with roles allowing parties to bind to them at run-time. In a similar fashion, actions that form a part of the commitment state definition are parameterised by collaborations (recall the `<embodiment>` element).

In order for an enterprise to effectively collaborate with any other entity there must exist a shared set of collaboration templates. Furthermore, if collaborations are to be carried over the Internet, they must be mapped onto the underlying IT infrastructure resulting in standard message based protocols. A number of e-service initiatives in the industry such as RosettaNet [Rosetta] or ebXML [ebXML] aim to address this issue. These initiatives define public repositories that contain descriptions of collaborations captured in a specification language. Typically the specification language is a form of a collaboration diagram that details the roles taking part in the collaboration, the names of the activities, the sequencing constraints as well as the data (often represented as an XML document conforming to a given schema) passed between the activities. We have found that most of the INCOTERMS contract actions can be realized with the RosettaNet collaborations. Going back to the example from the previous section contract action *GiveAdvanceNotice* can be implemented with the collaboration *NotifyOfAdvancedShipment* (the actual name for this collaboration according to RosettaNet classification is PIP3B2).

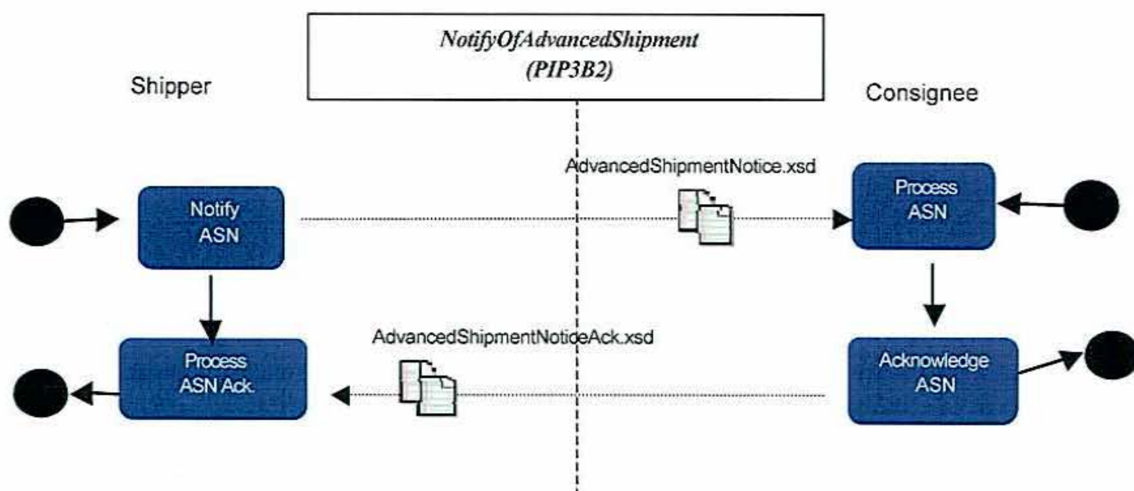


Figure 4 – RosettaNet PIP3B2

The collaboration specification visualized informally above has a formal description that can be serialized into XML. At the moment there is no industry wide standard for collaboration descriptions and there are competing proposals from open consortia such as RosettaNet or ebXML business process working group as well as IT vendors.

2.3.1 Collaboration implementation with business processes

The collaboration specification needs to be mapped onto a suitable implementation system. Workflow systems are a natural choice because the workflow description allows for role based task allocation and sequencing. Consequently, the collaboration specification can easily be transformed into process descriptions that can be enacted by the workflow system. Such automated transformation has been proposed for the HP Process Manager [Piccinelli 1999].

The result of such transformation is a set of process descriptions (one process for each collaboration role). The process implements a swim-lane of the collaboration specification as long as it meets the following criteria:

- It has work nodes that correspond to the activities in the collaboration specification;
- The work node has an attachment containing the document that is valid according to the required schema;
- The work node has a resource rule that will route the work the item to the work list served by the program that interfaces with the messaging system.

The work node in the process description meeting the above criteria can be called a public work node because it corresponds to the activity in the shared collaboration description.

When the task associated with the public work node is progressed relevant data is passed into the messaging application that formats a message and sends it to the collaborating party.

Figure 5 – Example transformation

An example set of processes that meet the transformation criteria is shown above. Notice that the process implementation of the collaboration specification can have work-nodes that do not correspond to activities in the collaboration description and are private to the enterprise. Alternatively, the process entirely consisting of public work-nodes and be executed as a sub-process. The containing process still meets the

transformation criteria i.e., it can be considered an implementation of the relevant part of the collaboration.

When the e-Contract is negotiated and a specific collaboration (such as *NotifyOfAdvancedShipment*) is proposed, contract parties determine the collaboration roles they want to play and make an <embodiment> entry in the XML representation of contract. Assuming that the collaboration messaging protocol is standard each party has to make sure that:

- they produce collaboration implementations for specific adaptors.
- the implementations are deployed into their IT systems.

These two steps can be accomplished using preferred vendor tools, but in the prototype, the tools for the HP ProcessManager are used. After collaboration implementation, a set of bindings between the collaboration for a given collaborative role and the corresponding collaboration implementation is produced. This is shown below as an XML file and a corresponding schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<actionBindings xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
xsi:schemaLocation="http://www.hp.com/ab
C:\Contract\src\test\ActionBinding.xsd">
        <binding name="NotifyOfAdvancedShipmentBinding" type="ProcessManager">
                <action name="NotifyOfAdvancedShipment" performingRole="Consignee">
                        <processImplementation processGroup="Shipper"
processName="NotifyOfAdvancedShipment.Consignee"/>
                </action>
        </binding>
        <binding name="NotifyOfAdvancedShipmentBinding" type="ProcessManager">
                <action name="NotifyOfAdvancedShipment" performingRole="Shipper">
                        <processImplementation processGroup="Shipper"
processName="NotifyOfAdvancedShipment.Shipper"/>
                </action>
        </binding>
</actionBindings>
```

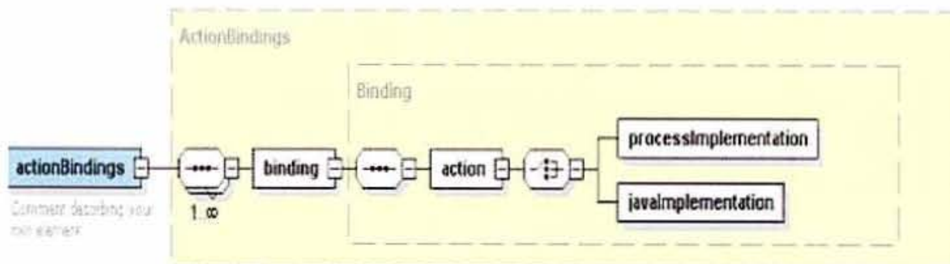


Figure 6 – process bindings captured in XML

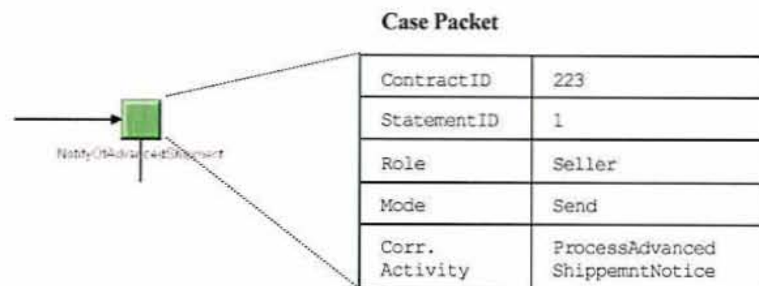
The <binding> element contains the binding name and indicates the type of the adaptor that will enact implementation of the collaboration. Because implementations in the prototype are processes, the type is *Process Manager*. The entry <action> specifies the collaboration part name and <processImplementation> contains the details of the collaboration implementation that the adaptor requires.

We have designed contract execution framework to be general and we left out a possibility for collaboration implementations and adaptors other than business process and the Process manager. Such implementations would have an adequate entry in the binding as indicated by the <javaImplementation>.

When a contract is setup for execution the XML Binding file is read, bindings relevant to the contract are loaded and the corresponding *Binding* object is made persistent. When the adaptor receives a request to execute a part of the collaboration, it uses the Binding object to identify corresponding implementation.

2.3.2 E-Contract considerations for collaboration implementations

In business, any collaborative interaction takes place within the context of the contract. For commodity goods, usually one contract is signed that is parameterised with respect to good type and -in conjunction with a trade confirmation document- is valid for collaboration concerned with many goods instances. Contracts for complex goods such as services are parameterised by service instances and therefore are valid for collaborations concerned with specific service instance. In our prototype, we focus on the second type of contracts.



Case Packet

ContractID	223
StatementID	1
Role	Seller
Mode	Send
Corr. Activity	ProcessAdvanced ShippemntNotice

Figure 7 – case packet of Work Node

As a given enterprise will typically have a number of contracts with the business partners within its IT infrastructure, there will be a number of instances of collaborations running. Therefore, some sort of contract context has to be passed into the collaboration instance so that the data received from the partners can be routed into appropriate instance. In our prototype, we implemented collaborations with business processes and therefore we pass the contract context (contractID, statementID and contract role that is fulfilled by the party) into process instances.

The contract context is available within each public work node of the process that implements the collaboration as shown in the figure above. Furthermore, we consider that each public work node (as defined in section 2.3.1) has a case packet variable *mode* that indicates whether the node is sending or receiving and a pointer *correspondingActivity* that referenced corresponding collaborative activity.

The resource rule associated with this node routes the task to a Worklist that is served by the messaging program upon its progression. The contract context allows messaging application to extract data from the work item. It then queries contract repository to obtain the electronic address of the contract party. Finally, it formats an appropriate message and sends it using an agreed protocol such as Mic rosoft *SOAP*.

3 Requirements

The ideas described in previous chapters are crucial to the design of the contract execution framework. Consequently, required features originating from those ideas arise. Since the work carried out in the placement covers certain parts of the framework, only the requirements and desired features belonging to that part are shown below:

Collaboration:
As mentioned in the previous section, the collaboration specification needs to be mapped onto an implementation system. Furthermore, certain actions such as sending and receiving documents between entities require to be handled by the implementation system. Since the HP Process Manager is the implementation system of choice for the prototype, means to enable collaboration between instances of it had to be found. Corresponding solutions are described in section 5.5.

Contract Compliance:
Contract execution can be considered as complying with the contract, if it meets the collaboration specification. In order to perform compliance check, each entity requires a monitoring component that collects appropriate data for the rules and tests that determine if an obligation has been fulfilled or not. While evaluation of the collected data leading to decisions is not carried out by the monitor, its task is crucial to the core part of the framework that deals with decision making and scheduling on a contract level. Issues related to monitoring are covered in section 5.4.

Visualization:
Having in mind that the prototype should allow visualizing the underlying ideas of execution framework a graphical user interface has been proposed to address this issue. After entering their party name, entities should be able to view their rights and obligations as well as information according to monitoring and execution (section 5.6).

4 Design of Contract Execution Framework

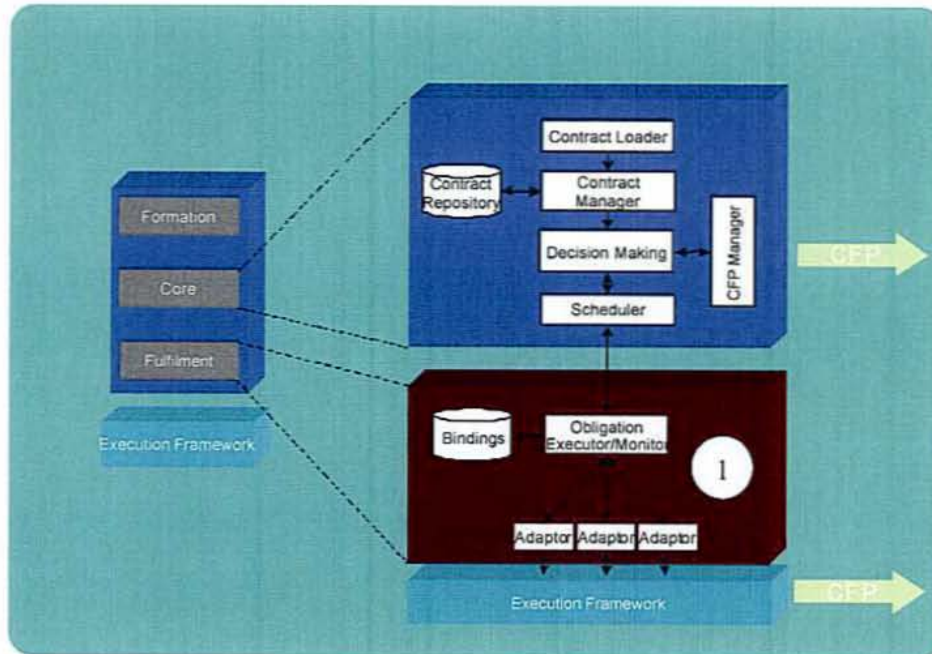


Figure 8 – framework overview

Conceptually the contract frameworks consists of following blocks:

- Formation of contract
- Core functionality dealing with decision making and scheduling on the contract level (the core block maintains state of commitments and decides if they need to be fulfilled)
- Fulfilment dealing with fulfilling contractual obligations

The main idea is that the contract *Statement* can be associated with a *Binding*. The association is done by the *Adaptor*, which is described detailed in Chapter 5.3. *Bindings* and *Adaptors* are specific to the InteractionController they interoperate. As HP ProcessManager is used for the prototype, *ProcessAdaptor* and *ProcessBinding* come to play.

The work described in this paper concerns the execution and monitoring of a contract, performed by the fulfilment part (shown in box 1 in Figure 8).
The fulfilment block (shown in Figure 9) current point of integration is the *Scheduler* that invokes a execute method on the *ExecutionManager* when a *Statement* is to be executed. Additionally the *Scheduler* exposes a notify method that can be called when the *Binding* associated with the *Statement* completes its execution.

The following sections describe briefly, how a *Statement* is associated and executed within the fulfilment part.

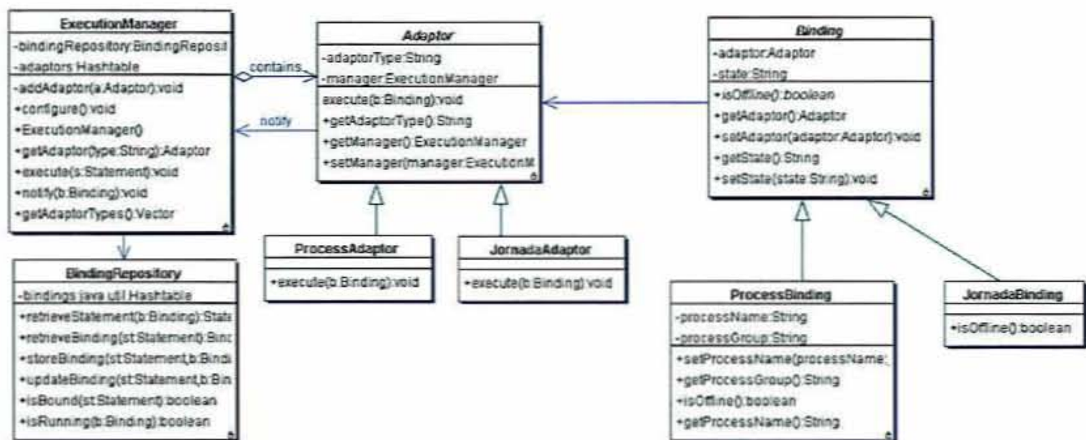


Figure 9 – fulfilment package class diagram

4.1 Creation of Adaptors

The *ExecutionManager* calls the configure method that creates instances of *Adaptors*. It also sets a reference to itself so that later on the *Adaptors* can notify it of *Binding* completion.

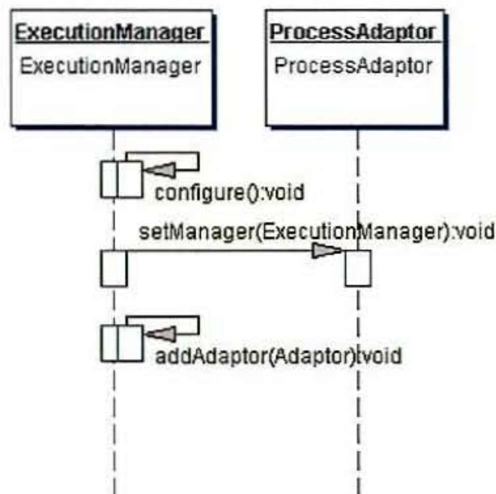


Figure 10 – creation of Adaptor

The Adaptors are added to the hash-table that can be referenced by the adaptor type (which is the class name of a specific Adaptor).

4.2 Creation of Bindings

The Binding to a contractual Statement can be created after Adaptors have been instantiated. Therefore, abstract Actions as specified in the contract have to be mapped onto concrete Actions. Each *Adaptor* can be queried for a list of possible

Bindings and performs the mapping. In case of *ProcessAdaptor*, *Bindings* are identified by ProcessName and ProcessGroup. The *Adaptor* carries out the task of resolving the action description into the action embodiment. When the contract is loaded and a contract object model is constructed each statement in the contract can be queried for the action declarations. These are passed to the *Adaptor* that resolves them into concrete embodiments represented by *Bindings*. *ProcessAdaptor* can resolve complex actions into processes and simple actions into tasks. This activity occurs during the preparation of contract for execution. When the *Binding* is created by the *Adaptor*, its state is set to "unbound" and when the association is made it becomes "bound and ready for execution".

4.3 *Execution of Statement*

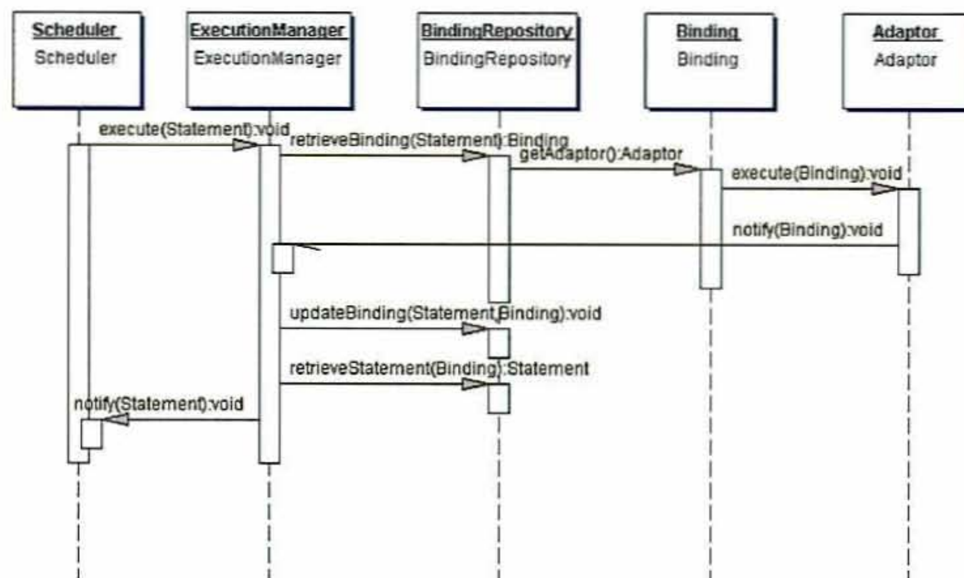


Figure 11 - Creation of Bindings for Contract in the set-up phase

Execution of a contract Statement is determined by the *Scheduler* component of the core. When the Statement is to be executed, the *ExecutionManager* is contacted and queries the *BindingRepository* for the *Binding* associated with the *Statement*. It then executes the Binding that its turn calls an appropriate Adaptor that executes it. Because the execution of contractual obligations is typically long running the notification about completion of the *Binding* to the *ExecutionManager* is asynchronous. The *ExecutionManager* updates the state of the *Binding* in the *Repository* and notifies the *Scheduler*.

When the bearer of commitment expressed by a *Statement* intends to fulfil it he will call the *execute* method on the *ExecutionManager* who will retrieve the appropriate *Binding* and pass it to the *Adaptor* for execution. The Binding knows about the Adaptor that can execute it and can delegate to it the execute method.

5 Use of Process Manager

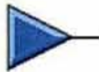
As mentioned earlier Hewlett-Packard ProcessManager was chosen as InteractionController for validation of contract framework design. Therefore *Monitor*, *ProcessAdaptor* and *ActivityMonitor* have to be able to perform interaction with the Process Manager. Process Manager provides three ways for programming-level access: Perl package, COM API, Java API. Since Java is the language of choice for the contract framework communication is done through Java API.
The Process Manager lacks functionalities that enable collaboration between multiple parties. Hence, ways had to be found to simulate collaboration as discussed in section 2. Section 5.5 shows how collaboration is achieved.

5.1 Main Concepts

This section describes briefly the required steps to create and run a process with the Hewlett-Packard Process Manager.

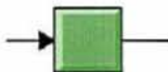
The application *Process Definer* offers functionalities for creation of Process Definitions. It provides a graphical representation of the process as a flow diagram, which consists of four different types of nodes:

Start Node



- Required to initiate the process
- Initiator (a person or an application) feeds some data into the process

Work Node



- Represents a specific item of work
- A service is specified that will be called to carry out the work, and the data will be pass to/from this service

Route Node

- Decision Point within the process
- AND/OR/NOT logic can be used
- Expressed as IF statement

Complete Node

- Used to show where a process comes to a logical end
- When the flow of the process reaches a complete node, it stops

Work Nodes carry out tasks that are determined by service descriptions. Services as used in the Process Manager require specification of input and output data items as well as *Resource Rules*. Actually, tasks are carried out by the *Resource Rule*, through whom an abstraction of executor is achieved. More concretely, Resource Rules specify Objects that are resolved into concrete person or application (e.g., through database lookup). Data items can contain basic variable types like String, Integer, Float, Boolean as well as complex types (e.g, word document) embodied by attachments. In addition, data items are determined by their scope, which is either global meaning accessibility within the whole process or local with respect to a specific node.
Data items can be defined in the graphical user interface of *Process Definer*, whereas Resource Rules require specification in a proprietary script language.
After creation, the process definition needs to be checked in the process repository. Thereby the process is stored in the repository under revision control. From then on, the process is available in the Process Manager and can be deployed in order to make it runnable. Figure 12 shows these steps.

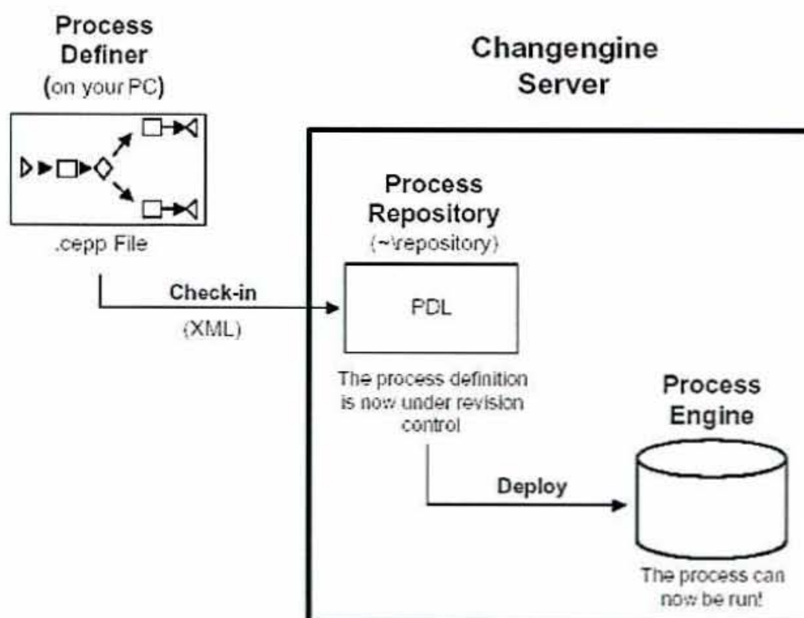


Figure 12 – check-in and deployment of process

5.2 Java API

Clients can communicate with the Process Manager in two different ways. The first way is to connect to the web server via TCP/IP (usually on port 80), which forwards requests to the Worklist Server through CGI. The disadvantage of this approach results from the dependency on the speed of the web server. Since standard access for clients to a web server is "connectionless" - meaning that the client establishes and re-establishes the connection with the server on each request – it is not suitable for bandwidth demanding applications.
The second way of communication uses the API Server that is part of the Worklist Server. It keeps the HTTP connection (port 9123) open so communication is not constantly slowed by reconnection times. Figure 13 illustrates these two connection schemes.

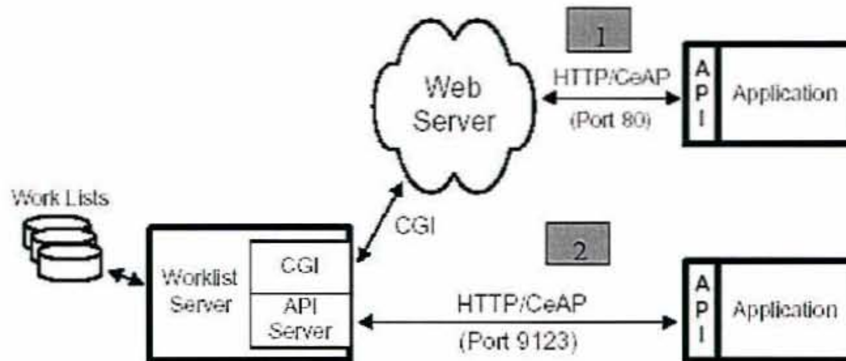


Figure 13 – API communication schemes

The Worklist Server maintains several Work Lists, which contain Work Items. Work Lists represent Resource Rules in a way that Work Nodes specified in the process can result in Work Items being dispatched between Work Lists. For instance, tasks that the Admin has to carry out appear in his Work List as shown in Figure 14.

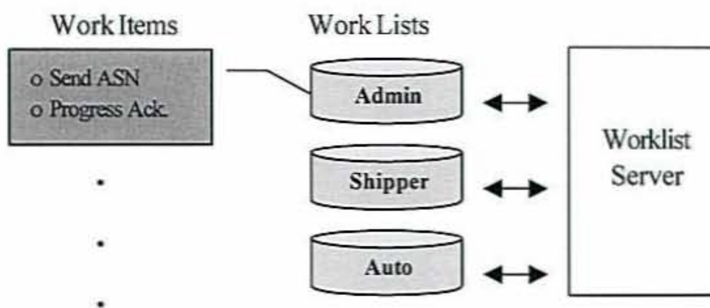


Figure 14 – Process Manager architecture

The structure of the Java API classes reflects the architecture of the Process Manager in maintaining the dependency between Work Lists and Work Items. Firstly, a session has to be established in order provide the context for the current logon. After that, Work Lists can be accessed and Work Items contained in them modified. Data Items of Work Items are treated as fields within Work Items. Only complex data items, encapsulated in Attachments, are represented in the Process Manager class model, which is shown in Figure 15.

Figure 15 – Java API overview

From a functional point of view Java API provides following features:

- Start Process
- Pass Data to Work Item
- Read Data from the Work Item

But Components of the framework that interact with the Process Manager require certain that cannot be accomplished with the Java API:

- History of execution
- Process Completion
- Temporal Information

Nevertheless, solutions have been found to achieve the desired functionality without using Java API. These solutions are detailed in chapter 5.4.

5.3 Design of the Adaptor

The *ProcessAdaptor* carries out the task of enabling interaction between the framework and the Process Manager. Thus, it is in control of all contract related data on the level of process instances/names/groups. This property led to the idea of integrating parts of the monitoring -concretely process monitoring- in the ProcessAdaptor. As a result, *Monitor* has only to deal with *Binding* and retrieves its actual state through interrogation of *ProcessAdaptor* that maintains the process level data. Concepts related to monitoring are discussed in section 5.4. An UML class diagram representation of the *ProcessAdaptor* is shown in Figure 16.
At initialisation, *ProcessAdaptor* establishes a session with the Process Manager, which is held open until destruction of ProcessAdaptor object. As described in chapter 4.3, when *ExecutionManager* intends to execute *Statement* it resolves the corresponding Binding and invokes the *execute* method of the appropriate Adaptor. In case of *ProcessAdaptor*, the Binding is cast to *ProcessBinding*, which contains process specific information like process group/name. This allows *ProcessAdaptor* to invoke the *startProcess(processGroup,processName)* method, which internally uses methods supplied by the Java API to start the process and to determine process instance ID. Furthermore, *ProcessAdaptor* passes contract context information to the process (i.e., Contract ID) that is stored in global variables and therefore available within the whole process. The process instance ID is stored in the appropriate *ProcessBinding* and later on required to perform process monitoring.

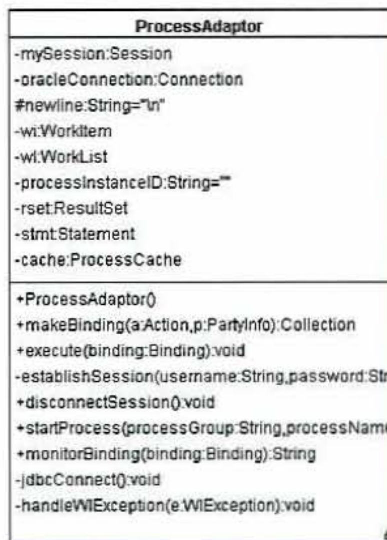


Figure 16 – ProcessAdaptor class diagram

In addition, *Adaptor* carries out the task of mapping the action description onto the action embodiment. When the contract is loaded and a contract object model is constructed each statement in the contract can be queried for the action declarations. These are passed to the *Adaptor* that resolves them into concrete embodiments represented by *Bindings*. More concretely, *ProcessAdaptor* resolves complex actions into processes and simple actions into tasks. This activity occurs during the preparation of contract for execution.

The resolution of an action depends on the contract party and so in addition to Action declaration the PartyInformation is passed as argument to the *makeBinding* method.

5.4 Monitoring

The contract defines states of obligation that are agreed between parties and should be adhered to. These states of obligation will be discharged when prescribed conditions hold. As collaborating parties are only in charge of contract fulfilment with respect to a party based projection of the contract, they desire to perform validation against contract interaction description. Therefore, each party requires to have monitor component that provides data for the rules and tests that determine if an obligation has been fulfilled or not (recall requirement *contract compliance*). Components such as *Decision Maker* that actually evaluate data provided by the *Monitor* are located in the core part of the contract framework. In order to feed these components with appropriate data *Monitor* needs to observe:

- State of process, which can be started, stopped, suspended or completed
- Tasks (Activities) and data items, especially when they have been sent out, -received

Although the web-based user interface of the Process Manager allows having a look at the current process state as well as temporal information about tasks, corresponding functionalities are not obtainable through the existing Java API. Therefore, custom ways had to be found to realize process and activity monitoring.

5.4.1 Process Monitoring

Process monitoring is based on the Process Manager's internal directly realisation of process logging. More concretely, process relevant data is obtained from an Oracle database in which the Process Manager stores logging information. Investigations of how the web-based graphical user interface builds pages containing process information have revealed the tables holding the required data.

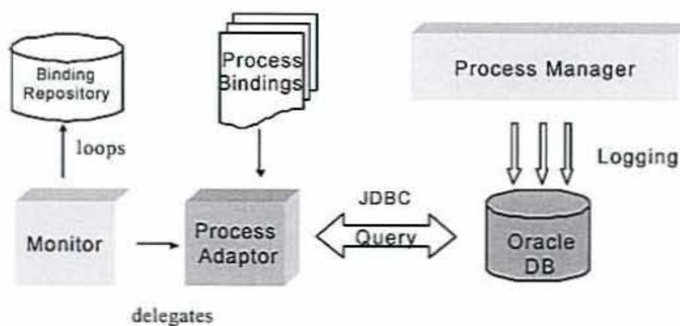


Figure 17 – process monitoring

As mentioned in the previous section, we interaction with the Process Manager is delegated to *ProcessAdaptor*. The *Monitor* queries *BindingRepository* seeking Bindings which state equals "executing". All matching *Bindings* are passed to

ProcessAdaptor, which retrieves process information (i.e., process instance ID) from ProcessBinding (typecast required) allowing it to construct appropriate queries against database. This monitoring routine is restarted regularly and is illustrated in Figure 17. Methods required for communicating with the Oracle Database are delivered by the Oracle JDBC driver.
If *Binding* completes, *Monitor* updates information in the *Repository*, where data is filed according to the contract context. Consequently, the completed *Binding* is not included in the next run of monitor thread.

5.4.2 Activity Monitoring

Ideally, the *Monitor* should be carrying out passive monitoring (i.e., the controller passes rele vant information to the monitor according to a monitoring description) but due to the limitations of the current Process Manager, active monitoring (by polling relevant work-lists in the Work-list Server) is used. While passive monitoring allows registering to certain events (i.e., Activity Y completed) that Interaction Controller fires, active monitoring requires knowledge about interaction description to determine Work Lists to poll. Figure 18 shows active monitoring with the Process Manager.

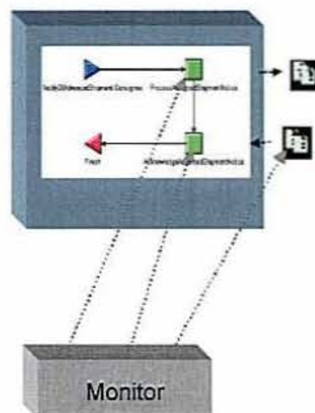


Figure 18 – active monitoring overview

In order to be able to perform activity (task) level monitoring we route the tasks that we want to monitor to the *Auto* work-list that is observed by the *ActivityMonitor* component. After gathering the monitoring information for the activity, the monitor progresses the task to allow process progression. Thus, access to data items is granted as well as exact temporal information. Furthermore, data ite ms relevant to discharge of the state of Obligation can be filed in a repository, which can be of use for components that determine if an obligation has been fulfilled or not. Figure 19 depicts active monitoring in context of the Process Manager.

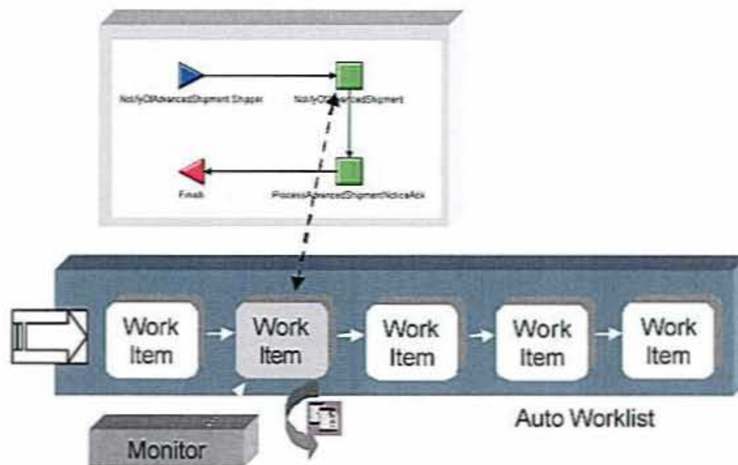


Figure 19 – active monitoring with Auto Worklist

5.5 Simulation of Process execution

Components being introduced in previous sections are shown in context of process execution. In order to simulate business interaction with the Process Manager, Interaction description expressing the collaborative workflow needs to be translated in process descriptions (recall requirement *collaboration*).

Figure 20 shows the process descriptions involving participants shipper (left side) and consignee.

Figure 20 – process representation of collaboration

As described in Figure 20, collaborative workflow contains documents send between parties. For instance, Shipper needs to give Advanced Shipment Notice seven days before actual shipment takes place. Ideally, the Process Managers of each party would handle communication among each other, but due to limitations of current version we had to find custom way. Seeing that setting up communication between different instances of the Process Manager is time consuming and would require technologies like *Java Messaging Service*, a single installation is used to achieve collaboration. Similar to the scheme introduced by active monitoring, tasks are routed to the *Auto*

Work List. *ActivityMonitor* is in control of Work Items in the *Auto* Work List, consequently we added methods to it that simulate the interaction. Behaviour of *ActivityMonitor* is subject to role it has been assigned, therefore two *ActivityMonitors* are required in the scenario.

Figure 21 shows workflow of an *ActivityMonitor* playing role shipper in flow diagram notation. Firstly, it evaluates case packets of a *Send* Work Items appearing in the *Auto* Work List (recall section 2.3.2). If ContractID, StatementID, and Role match with the values *ActivityMonitor* has been initialised, it enters *Send Mode* and tries to find the corresponding *Receive* Work Item (belongs to Consignee process) indicated by the *correspondingActivity* case packet variable. If described conditions are hold the *Attachment*containing *AdvancedShipmentNotice* is set in the *Receive* Work Item and the *Send* Work Items is progressed. This leads the *ActivityMonitor* to switch to *Receive* Mode awaiting the *Receive* Work Item belonging to the Shipper process to appear. Only if the *Receive* Work Item contains the appropriate *Attachment (AdvancedShipmentNoticeAck)* and case packet variables (ContractID, StatementID, and Role) it is progressed and in so doing the process finished. The *ActivityMonitor* playing role Consignee operates in a similar way, following description of the Consignee process. Consequently, *Receive* Mode is progressed previous to *Send* Mode.

As the result correct execution of collaborative process is guaranteed, although undergoing less flexibility.

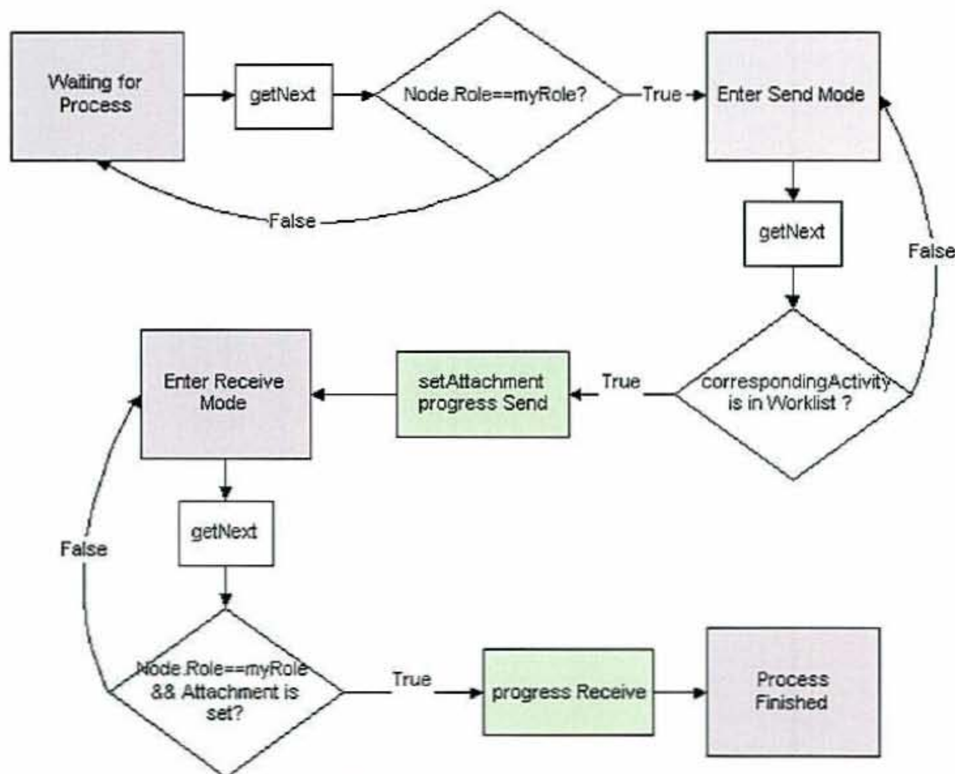


Figure 21 – workflow of activity monitor Shipper

5.6 Visualization of Contract Execution

In chapter 3, it has been mentioned that the prototype should allow visualisation of contract framework ideas. Therefore, we created a graphical user interface that allows visualizing simulation of process execution. Figure 22 shows a screenshot of the running GUI.

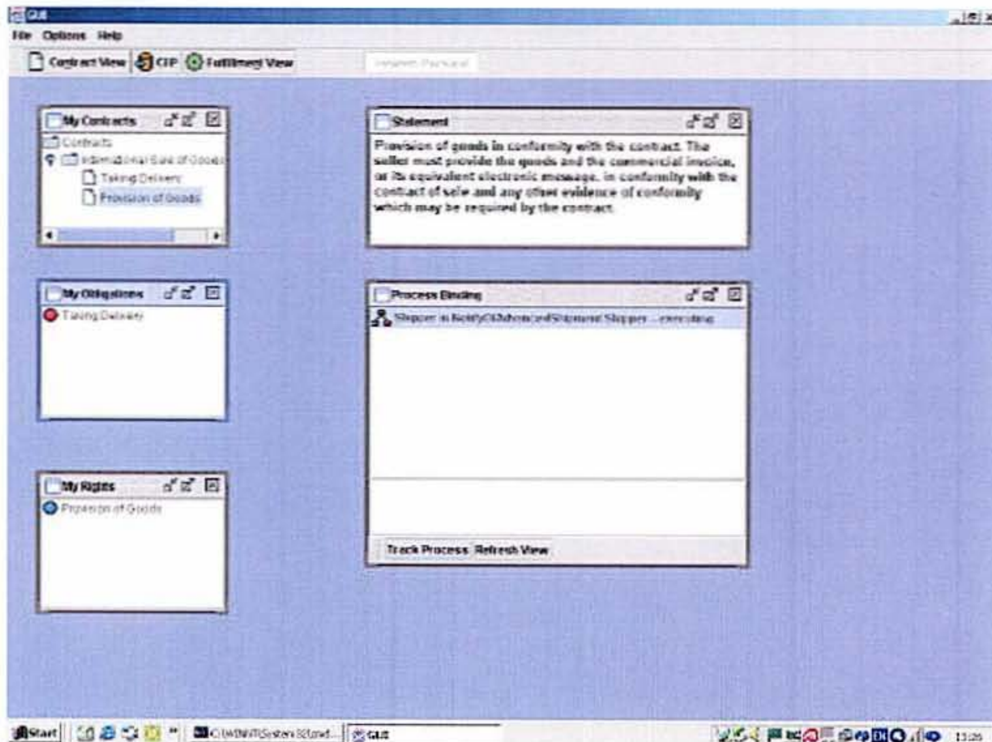


Figure 22 – screenshot of graphical user interface

For simulating execution of contract the GUI is started by each party, in our Scenario namely Hewlett-Packard and British-Library. A file chooser allows user to select the electronic contract expressed as an XML file he wants to execute. The Contract is loaded by the *ContractLoader* and object model associated with it is constructed. Based on the party information, windows are populated and statements are executed. Functionality of different windows and information they provide is shown below.

My Contracts

All loaded Contracts are shown in a tree view, where contract nodes contain statements associated with as sub nodes.

My Obligations

When user selects contract in *My Contracts* Window obligations of the actual party resulting from the contract are displayed.

My Rights
When user selects contract in *My Contracts* Window rights of the actual party resulting from the contract are displayed.

Statement
Textual representation of the statement as specified in the contract is given when user selects statement node in the *My Contracts* window.

Process Binding
Information about a ProcessBinding, associated with the statement that is selected in the *My Contracts* window, is given. Information contains Process Group, process name and current state of *ProcessBinding*. Additionally, by clicking the *Track Process* button an Internet page that shows current progression of process is opened in browser. The page is supplied by the Process Manager and shows actual progress of process execution by displaying completed Work Nodes in a different colour.

5.7 Recommendation and Desired Features

From the work with the HP Process Manager, we discovered desired features meeting needs of execution framework:

Dynamic processes deployment:
As mentioned in chapter 2.3.1, generation of processes implementing collaboration can be achieved by transformation tools. Given that, one would like to deploy the generated processes programmatically. Combining these two methods, an agreed collaboration could be automatically progressed to set of *ProcessBindings.*

Programmatic process management:
It is described in chapter 5.4.2 that passive monitoring is the preferred method but is not supported by current version of Process Manager. This could be realized by adding API methods that allow receiving events related to process/activity/case packet and variables lifecycle.

Intra Process Manager communication:
Execution of Contract involves actions like sending and receiving documents. This results in data exchange among Process Manager instances of participating entities. Ideally, Process Manager would provide a proprietary protocol handling communication. Because exchange is most likely carried over the Internet, secure communication needs to be supported.

6 References

[ECS 2000] Boulmakoul A., Bartolini C., Morciniec M., 2000, "Electronic Contract Specification", HP Labs Bristol

[Ibbotson J., Sachs M., 1999] Ibbotson J., Sachs M., 1999, "Electronic Trading Partner Agreement for E-Commerce", IBM Corporation.

[Marshall, 1999] Marshall C., "Enterprise Modelling with UML", Addison Wesley 1999.

[ebXML] Electronic Business XML initiative, http://www.ebXML.org/

[RosettaNet] RosettaNet Consortium, http://www.rosettanet.org

[Daskalopulu 2000], Daskalopulu A., 2000, "Modelling LegalContracts as Processes", *Legal Information Systems Applications*, 11th International Conference and Workshop on Database and Expert Systems Applications, IEEE C. S. Press, pp. 1074–1079

[Norman 2000] Norman T., Reed C., "Delegation and responsibility", in Proceedings of the Seventh International Workshop on Agent Theories, Architectures and Languages.

[Ramberg 2000] Ramberg J., 2000, "ICC Guide to Incoterms 2000", International Chamber of Commerce.

[Piccinelli 1999] Piccinelli G., 1999, "A process decomposition technique for distributed workflow management" in Proceedings of the 2nd IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS).

[SOAP 2000] World Wide Web Consortium, 2000, "SOAP Version 1.2", http://www.w3.org/TR/2001/WD-soap12-20010709/