Consistency Checking of Web Service Contracts

M. Emilia Cambronero
Department of Computer Science
University of Castilla-La Mancha
SPAIN
Email: emicp@info-ab.uclm.es

Joseph C. Okika
Department of Computer Science
Aalborg University, Aalborg
DENMARK
Email: ojc@cs.aau.dk

Anders P. Ravn
Department of Computer Science
Aalborg University, Aalborg
DENMARK
Email: apr@cs.aau.dk

***Abstract*—Behavioural properties are analyzed for web service contracts formulated in Business Process Execution Language (BPEL) and Choreography Description Language (CDL). The key result reported is an automated technique to check consistency between protocol aspects of the contracts. The contracts are abstracted to (timed) automata and from there a simulation is set up, which is checked using automated tools for analyzing networks of finite state processes. Here we use the Concurrency Work Bench. The proposed techniques are illustrated with a case study that include otherwise difficult to analyze fault handlers.**

Keywords:
Web Services contract, consistency, WS Choreography, WS Orchestration.

I. Introduction

Service Oriented Architecture (SOA) [1] reorganizes series of previously operational software applications and support infrastructure into an interconnected set of services, each accessible through standard interfaces and messaging protocols. It promotes services that are distributed, heterogeneous, autonomous and open in nature. SOA is particularly applicable when multiple applications running on varied technologies and platforms need to communicate with each other. With SOA, enterprises can mix and match services to perform business transactions with less programming effort. SOA is implemented with web service technology. Thus there is consensus today, that a web service is a programmable component that provides a service and is accessible over the Internet. They are based on standards like Simple Object Access Protocol (SOAP) [2], [3], [4], can be standalone, or linked together to provide enhanced functionality.

Businesses depend on web services, therefore their properties are of great importance, and informal checking and consensus approaches to when a service is good enough may not suffice. A business will only reluctantly use enterprise applications offered as open web services, because of the high risks involved in using untrusted services from unknown providers. Formal contracts defining the desired properties are therefore studied intensively today, because they are a way to manage the risks that come with the interaction among these inter-organizational services.

Traditionally, contracts in an object oriented setting consider only the functional aspect (pre-condition, post-condition, invariant) of an interface specification. A pre-condition is a constraint that must be satisfied before calling a method or

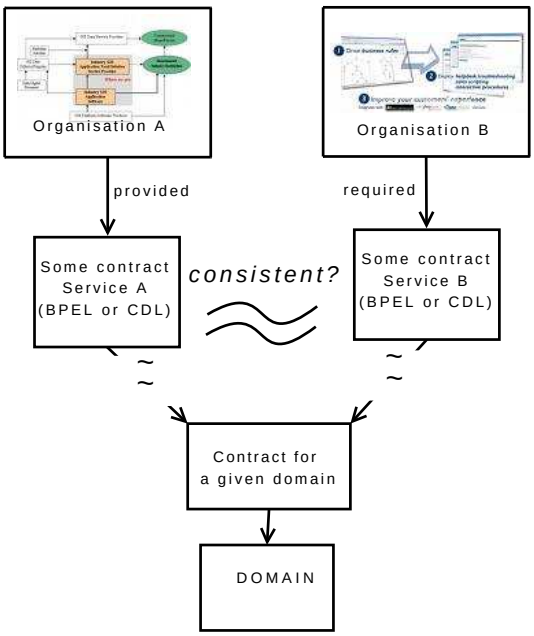


Fig. 1. Analysis of Web Service Contracts

operation; it checks for valid arguments. A post-condition is a corresponding property that is true when the call completes; it is the input-output relation. Finally, an invariant is a constraint on the state of an object; it must hold before and after any operation, and clearly after initialization of the object. These concepts, as popularized by Meyer's "Design by Contract" [5], are, however, just part of the properties exhibited by web services. Since web services are intrinsically distributed, they are by nature concurrent programs, and thus their overall functionality depends not only on correct implementation of the local functionality by sequential algorithms, but even more on the interplay between local functionality and global behavior (protocols and timing).

In this paper we focus on protocol or behavioural aspects of service contracts. There are several proposals for contract specification standards for web services, see e.g. [6] for an overview. Prominent among these standards are the Business Process Execution Language (WS-BPEL)[7] and Choreography Description Language (WS-CDL) [8]. BPEL offers a programming model for specifying the orchestration of web services whereas CDL specifies the choreography of

interacting services. However, when web service contract are specified using either BPEL or CDL, there is no assurance that they are consistent unless verified. Though there are efforts toward this form of analysis, there remain challenges in the area of automated approach to checking consistency in addition to other properties.

In previous work [9] we have demonstrated a viable solution to the problem of checking for functional and behavioural properties of individual services. This is done through translation of the specifications to timed automata followed by model checking for relevant properties. In [10] we considered the problem of consistency across specifications and identified a need to set up a correspondence between the individual automata. The novel contribution in this paper is to make such a consistency check practical by translating the automata to CCS, the input language for the Concurrency Work Bench. As demonstrated by a case study, this technique is applicable and gives a handle for automating yet another consistency check for web services.

Directly Related Work: Web Service contracts is attracting a lot of attention and several researchers propose various approaches and frameworks toward specification and analysis. For instance [11], [12], [13], [14] looks at it from a formal semantics viewpoint, whereas [15], [16] propose languages for specifying contracts. All these points to the fact that there is an important need for contracts to be specified and analyzed.

An earlier treatment of contracts in an object-oriented paradigm is Design by Contract [5]. Similar treatment concerning components is found in [17]. Here, the functional specification is achieved through assertions; which consists of preconditions, post-conditions and invariants. The framework in [18] takes a pragmatic approach at code level where the assertions are part of the language. We agree that these functional specifications are important in order to specify a formal agreement between a service provider and its clients. It expresses what a client should do before making a service request and what the provider will give as result of it.

Among the related work of Web Service contracts is [19]. It proposes to visualize contracts by graph transformation rules. Apart from expressing contracts in terms of pre- and post-conditions of operations together with invariants, they introduced the notions of provided and required contracts. With this, they use the provided contracts to create the test cases and test oracles whereas the required interfaces are used to drive the simulation. We like their treatment of functional specifications, but it needs to be supplemented with other aspects, and one may gain something by investigating model checking as a supplement to testing.

Quantitative aspect are researched in [20], [21], [22]. The Web Service Level Agreement (WSLA) framework [20] is targeted at defining and monitoring SLAs for Web Services. WSLA enables service customers and providers to unambiguously define the agreed performance characteristics and the way to evaluate and measure them. We want to mention here that WSLA complements Web Service Definition Language (WSDL) [23], [24], which is an XML grammar that describes the capabilities of Web services through its interface descriptions. WSLA is used to define a contract between service provider and service requester, but its treatment of functional behavior is limited.

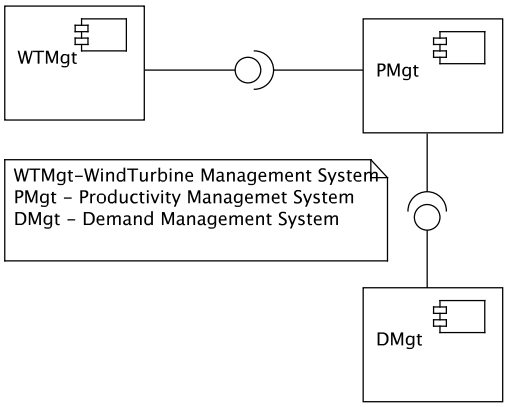


Fig. 2. Wind Turbine Management System Components

The above mentioned contributions focus on a single web service language, and either the functional or the behavioral side of a contract. We extend their perspective by considering the overall consistency of a service specified in languages covering more than one aspect. Furthermore we demonstrate how existing tools are adapted for such checks.

Overview: In Section II, we give a detailed presentation of Web Service contracts where the aspects of contracts are described. We introduce in this section, a case study of a Windmill Management System. Section III details the analysis of Web Service contracts. General consistency, satisfiability, and application specific issues are presented. A comparison with other approaches follows and finally, we conclude in Section V.

II. Web Service Contracts

To manage the risks that come with the interaction among several services, the service provider and a consumer must have a contract that specifies the details of the service. As mentioned before, it is important to note, however, that there are different aspects of contract in play when dealing with web services. First, there is the functional aspect which describes the functional properties, and second, there is the protocols aspect which specifies the behaviour as a sequence of messages, events, signals, etc. There is also the extra functional QoS (Quality of Service) requirements aspect. This is further illustrated following the example presented in the following subsection.

A. Example

We consider a Windmill Management System. The system monitors and controls wind turbines, and it has several components which are web services located in different places. We focus on three of these components, because it gives us

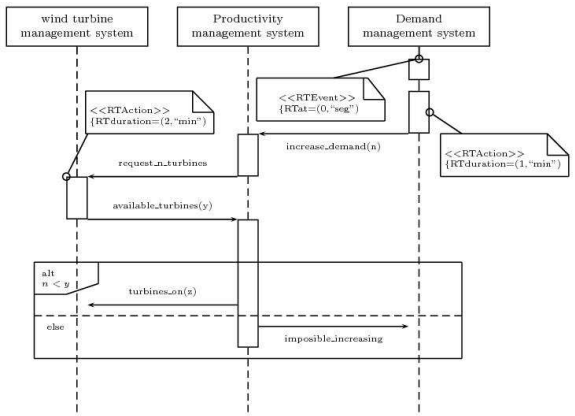


Fig. 3. Wind Turbine Management System Sequence Diagram

the scenario needed to specify a web service contract. The components are briefly described below and shown as an UML component diagram in Figure 2. The interaction between these services are illustrated using a RT-UML sequence diagram, shown in Figure 3. The informal requirements for the components are:

- Wind Turbine Management: sends a report to Productivity management every hour.
- Productivity Management: receives and analyzes the report from Wind Turbine Management.
- Demand Management: generates a report of power needs for Productivity Management.

We look at this example from two perspectives; WS-CDL and WS-BPEL. WS-CDL provides a definition of the information formats being exchanged by all participants. In other words, it specifies the protocols. WS-BPEL provides the message exchanges and functions as viewed by one participant. It describes the functionality of a single business process offered as a service by an enterprise.

B. Contract Aspects in WS-CDL

CDL offers a model for specifying a common understanding of message exchanges. This language describes the choreography of web services systems, that is, the relationships between the composite services in a peer-to-peer environment. It uses the WS definition language (WSDL) to define and locate common type definitions.

WS-CDL is a very verbose notation, therefor the key concepts of contracts in WS-CDL are summarized below, while a full description of the demand management system is found in appendix A.

Interface: In WS-CDL, each interface is associated with a particular role, where a `roleType` enumerates potential observable behaviors a participant can exhibit when interacting with other participants. The syntax is the following:

```
<roleType name="DemandRoleType">
      <description type="description"/>
      <behaviour name="DemandBehaviour"
           interface="WSDLDemandType" />
</roleType>
```

The `behaviour` element defines an optional interface attribute, which identifies a WSDL interface type.

Functional Specification: pre-conditions, post-conditions and invariants: In WS-CDL these elements are defined by means of *workunits*; which define the constraints that must be fulfilled for making progress and describe some activities within a choreography. The constraints are give by *XPath 2.0* expressions.

XPath 2.0 supports date and time variables, so we can use these variables in WS-CDL as well. Furthermore, XPath provides a number of functions to manage these datatype values.

```
<workunit name="demand increase detected"
      guard="cdl:equal(cdl:getVariable
         ('tns:DemandClock'),'','),'0:00')"
      block="true">
  <assign roleType="DemandRoleType">
    <copy name="calculateincrease"
          causeException="true">
      <source variable="true"/>
      <target variable=
       "cdl:getVariable('detectedincreaseDone','','')"/>
    </copy>
  </assign>
</workunit>
```

A *workunit*'s `guard` element establishes the condition, which has to be fulfilled to perform the workunit activities. This element allows us to define pre-conditions. Postconditions and invariants can be introduced by appending a workunit with the condition as a guard at the end of the normal workunit flow. In order to define a condition we use XPath and XML Schema expressions.

Protocol: A *sequence* of activities is modeled in WS-CDL using the ordering structure `sequence`, which contains a set of activities that can perform sequentially.

A *non-deterministic choice* is implemented in WS-CDL using the ordering structure `choice`. The WS-CDL standard says that when two or more activities are specified here, only one of these is selected and the other ones are disabled. It is assumed that the selection criteria for those activities are non-observable.

The following WS-CDL code corresponds to the fragment in which the productivity system sends a message to the turbine system for the turbines to be turned on or else it sends a message to the demand system to indicate that it is not possible to satisfy the new demand. As you can see, it is modeled in WS-CDL by a choice activity in which we have two activities, and only one of them can be finally executed.

```
<choice>
  <workunit name="alt_else1_if"
            guard="Available == true" block="true">

      <interaction name="TurbinesOn_interaction"
            operation="TurbinesOn"
            channelVariable=
                 "Productivity2WindTurbineChannel">
        <participate relationshipType=
                    "ProductivityWindTurbine"
             fromRole="ProductivityRoleType"
             toRole="WindTurbineRoleType"/>
        <exchange name="TurbinesOnExchange"
                  action="request"/>
      </interaction>
```

```
  </workunit>

  <workunit name="alt_else1_else"
          guard="Available != true" block="true">

      <interaction name="Imposible_interaction"
                operation="Imposible"
                channelVariable=
                      "Demand2ProductivityChannel">
          <participate relationshipType=
                           "ProductivityDemand"
                fromRole="ProductivityRoleType"
                toRole="DemandRoleType"/>
          <exchange name="ImposibleExchange"
                    action="request"/>
      </interaction>
  </workunit>
</choice>
```

An *external choice* is implemented in WS-CDL using the ordering structure *workunit*, since it allows us to establish conditions to execute the corresponding activity. For that purpose, we may use the guards of workunits, by including in a guard an expression related with the value of a variable.

In WS-CDL, we use the workunit `repeat` to implement repetition. A workunit that completes successfully must be considered again for matching (based on its guard condition), if its repetition condition evaluates to `true`.

```
<workunit name="alt_else1_if"
         guard="Available == true"
         repeat="false" block="true" >

   <interaction name="TurbinesOn_interaction"
         operation="TurbinesOn"
         channelVariable=
             "Productivity2WindTurbineChannel">
       <participate relationshipType=
                        "ProductivityWindTurbine"
          fromRole="ProductivityRoleType"
          toRole="WindTurbineRoleType"/>
       <exchange name="TurbinesOnExchange"
                    action="request"/>
   </interaction>
</workunit>
```

Timing: Lower bounds, upper bounds, explicit clocks, reset and stop operations are handled by XPath and XML Schema.

XPath 2.0 supports date and time variables, so we can also use these variables in WS-CDL. Actually, XPath provides a number of functions to manage these datatype values. These variables can be used in particular to delay the execution for a certain time, or to establish the instant at which some actions must be executed. For that purpose, we may use the guards of workunits, by including in a guard an expression related with the value of a time variable.

Specifically, we use the XPath and XML Schema notation to specify the time aspects as follows:

a) Explicit clocks: are introduced by `xs:time`.

b) Bounds: are specified inside a workunit guard. In fact, as we capture delays or instants of execution, the specific expressions allowed are those constructed using the operators `op:time-equal op:time-less-than` and `op:time-greater-than` of XPath 2.0. We can also use the `hasDeadlinePassed` operation, which is defined in the WS-CDL specification to manage timing.

c) Reset.: In WS-CDL we reset a clock using an `assign` activity, which creates or changes the variable defined by the target element using the expression defined by the source element (in the same role).

d) Stop.: In order to model that a clock is stopped, we can capture the value of the time, of this specific instant, in a clock variable and then, when we want to initiate the time again, we can use the clock variable to continue from this point. We use two `assign` activities to capture and change the time value.

e) Synchronization.: The `interaction` WS-CDL element defines how the parties in a web services are synchronized. An interaction activity involves two roletypes, and an exchange of information between them. Actually, in WS-CDL several exchanges of information are allowed in a single interaction, and they can be either `request` or `respond` types, and these actions can be synchronous or asynchronous, depending on the `align` attribute.

```
<interation name="The demand management system
           sends increase in power demand to
           the productivity system"
   operation= = "sendIncreasing"
   channelVariable="Demand2ProductivityC">
   <description type="description">
     Sending the necessary increase of demand
   </description>
   <participate
      relationshipType= "DemandProductivity"
      fromRole="DemandRoleType"
      toRole="ProductivityRoleType" />
   <exchange name= "CalculatedIncerasing"
     informationType="Increase_demandType"
     action="request">
   </exchange>
   <timeout
     time-to-complete= "cdl:minor(cdl:getVariable
       ('tns:Clock1','','1:00')">?
</interaction>
```

In the `time-to-complete` attribute the timeframe in which an interaction must complete is specified. Then, when this time expires (after the interaction was initiated) and the interaction has not completed, a timeout occurs and the interaction finishes abnormally, causing an exception block to be executed in the choreography. The optional attributes `fromRoleTypeRecordRef` and `toRoleTypeRecordRef` are XML-Schema lists of references to record elements that will take effect at both roleTypes of the interaction.

Faults: Choreographies may have one exception block, which consists of some (possibly guarded) *workunits*, but only one of them can be finally executed (the first one whose guard evaluates to true). When the exception block is executed, the choreography terminates abnormally, even if the default exception workunit has terminated correctly. Exceptions are the following:

f) Interaction failures: For instance, sending of a message failed.

g) Timeout errors: For instance, an interaction did not complete within the alloted time.

h) Application failures: These are for instance illegal expressions.

CDL in summary: Overall CDL is a coordination language which focuses on the communication between agents providing the services. It is therefore very appropriate to give it

a semantics by translation into a network of communicating processes.

C. Contract Aspects in WS-BPEL

BPEL is a programming language to specify the behavior of a participant in a choreography. It allows existing Web services to be orchestrated into composite services. Choreography is concerned with describing the message interchanges between participants.

WS-BPEL is verbose also, so we do not include full descriptions; but as for WS-CDL, we present the WS-BPEL contract aspects below:

Interface: In WS-BPEL, the services with which a business process interacts are modeled as `partnerLinks`. Each `partnerLink` is characterized by a `partnerLinkType`, which defines the roles played by each of the services in the conversation and specifies the `portType` provided by each service to receive messages within the context of the conversation. These `portTypes` are defined in the WSDL document, and each role specifies exactly one WSDL `portType`.

In order to utilize operations via a `partnerLink`, the binding and communication data, including *endpoint references (EPR)*, for the `partnerLink` must be available. The fundamental use of endpoint references is to serve as the mechanism for dynamic communication of port-specific data for services. An example fragment of a `partnerLink` is:

```
<partnerLinks>
<partnerLink name="productivity">
partnerLinkType="as:productivityDemandMSLT"
   myRole="DemandMS"
   partnerRole="productivity" />
</partnerLinks>
```

The endpoint references syntax is:

```
<service-ref reference-scheme="http://example.org">
  <foo:barEPR xmlns:foo="http://example.org">
   ... </foo:barEPR>
</service-ref>
```

Functional Specification: preconditions, postconditions and invariants: WS-BPEL uses several types of expressions to implement the functional part of a web service contract:

- Boolean expressions. These expressions can appear inside a transition, a join, a while, and an if condition.
- Deadline expressions. The WS-BPEL elements that use these expressions are until-expressions of onAlarm and wait.
- Duration expressions. These appear in the `for` expression of `onAlarm` and `wait`, and the `repeatEvery` expression of `onAlarm`.
- Unsigned Integer expressions, that include counter values `startCounterValue`, `finalCounterValue`; as well as branches in a `forEach`.
- General expressions inside assign activities.

Protocol: sequence, choice, and iteration:

- A sequence of activities is modeled by the `sequence` structured activity. It contains one or more activities that are performed sequentially, in the lexical order in which they appear.

  An example is the Productivity process which is given as a sequence as follows:

```
<sequence>
 <if
 bpel:getVariableProperty('x','time:level')==0>
   <then>
  <!-Process productivity (invoke) - ->
   <assign>
<copy>
<from partnerLink="productivityMS"
endpointReference="myRole" />
<to>&increaseData.productivityMSRef </to>
</copy>
 </assign>
 <invoke name="increaseDemand"
          partnerLink="productivity"
          portType="as:productivityPT"
          operation="process"
          inputVariable="increaseData">
   <correlations>
   <correlation set="increaseIdentification"
   </correlations>
 </invoke>
 </if>
</sequence>
```

- Choice. Both non-deterministic and external choice are expressed in WS-BPEL by means of `pick` activities, which waits for the occurrence of an event and then executes the activity associated with it. When several events occur simultaneously, an implementation dependent choice is made. Thus, in analysis, the choice must be modeled as non-deterministic.
- Conditional. WS-BPEL contains a conventional conditional statement as well.
- Iteration. WS-BPEL uses the `while` and `repeatUntil` activities, to model iteration.

```
<while>
   <condition>
      $numberWindTurbine < 10
   </condition>
   <scope>
     ...
   </scope>
</while>

<repeatUntil standard-attributes>
    standard-elements
    activity
    <condition expressionLanguage="anyURI"?>
       ... bool-expr ...
    </condition>
</repeatUntil>
```

Timing: Lower bounds, upper bounds, explicit clocks, reset and stop operations are specified as in WS-BPEL using XPath and XML Schema.

i) Explicit clocks, lower and upper bounds: They are defined using XML Scheme notations, as explained before.

j) Reset: In WS-BPEL we can reset the clock using an `assign` activity, which copies data from one variable to another.

```
<assign validate="yes|no"? standard-attributes>
   standard-elements
     (<copy keepSrcElementName="yes|no"?>
     from-spec
        to-spec
      </copy> |
      <extensibleAssign>
        ...assign-element-of-other-namespace...
      </extensibleAssign>) +
</assign>
```

k) Stop: In order to model that a clock is stopped in WS-BPEL we do as in WS-CDL.

l) Concurrency and Synchronizations: They are implemented in WS-BPEL using a `flow` activity, which provides concurrency and synchronization. A `flow` completes when all of the activities enclosed by it have completed.

```
<flow standard-attributes>
    standard-elements
    <links>?
        <link name="NCName">+
</links>
        activity+
</flow>
```

Faults: Business processes are usually of long duration. They can manipulate data in back-end databases and line-of-business applications. Error handling in this environment is both difficult and business critical. The overall business transaction can fail or be canceled after many transactions have been committed. In this cases, the partial work done must be undone or repaired as best as possible. Error handling in WS-BPEL processes therefore leverages the concept of compensation, that is, application-specific activities that attempt to reverse the effects of a previous activity that was carried out as part of a larger unit of work that is being abandoned. It thus provides the means for a forward error recovery.

Specifically, WS-BPEL provides constructs to declare fault handling and compensation.

m) Compensation handler: WS-BPEL allows scopes to delineate that part of the behavior that is meant to be reversible in an application-defined way by specifying a compensation handler. A `compensationHandler` is simply a wrapper for an activity that performs compensation.

```
<compensationHandler>
   activity
</compensationHandler>
```

It is invoked with `compensateScope`, when an explicit scope is compensated, or `compensate` when successfully completed inner scopes are compensated in reverse order. A compensation handler for a scope is available for invocation only when the scope completes successfully.

```
<compensateScope target="NCName"
                 standard-attributes>
   standard-elements
</compensateScope>

<compensate standard-attributes>
   standard-elements
</compensate>
```

Compensations may only be invoked in `catch`, `catchAll`, `compensationHandler` and `terminationHandler` activities, where termination handlers provide the ability for scopes to control the semantics of forced termination by disabling the scope's event handlers and terminating its primary activity and all running event handler instances.

n) Fault handling: In a business process it can be thought of as a mode switch from the normal processing in a scope. Fault handling in WS-BPEL is designed to implement backward error-recovery in that it aims to undo or repair the partial and unsuccessful work of a scope in which a fault has occurred. The completion of the activity of a fault handler, even when it does not rethrow the handled fault, is not considered successful completion of the attached scope. Compensation is not enabled for a scope that has had an associated fault handler invoked.

Explicit fault handlers attached to a scope provide a way to define a set of custom fault-handling activities, defined by `catch` and `catchAll` constructs. Each `catch` construct is defined to intercept a specific kind of fault, defined by a fault `QName`. If the fault name is missing, then the catch will intercept all faults with the same type of fault data. A `catchAll` clause can be added to catch any fault not caught by a more specific fault handler.

```
<faultHandlers>
   <catch faultName="QName"?
      faultVariable="BPELVariableName"?
      ( faultMessageType="QName" | faultElement="QName" )?>*
      activity
   </catch>

   <catchAll>?
      activity
   </catchAll>
</faultHandlers>
```

There are various sources of faults in WS-BPEL. A fault response to an `invoke` activity is one source of faults, where the fault name and data are based on the definition of the fault in the WSDL operation. A `throw` activity is another source, with explicitly given name and/or data. WS-BPEL defines several standard faults with their names, and there may be other platform-specific faults such as communication failures.

BPEL summary: BPEL is essentially a programming language. However it has some features that are specially tailored to make it easier to build robust systems that can recover from a variety of faults. It includes features for expressing internal concurrent activities; they should however be used with care, because it is not always easy to comprenhed the interaction with compensations and fault handlers.

III. Analyzing Web Service Contract

Having described all the elements of specifications, we now present the translation to automata. In order to perform this translation, we note that WS-CDL and WS-BPEL are XML based languages for describing Web Services. The timed automata formalism we use is UppAal [25]; and it is represented by another XML document, thus, the translation has been developed with XSLT [26], XML Style sheets Language for Transformation, which is a language for transforming XML documents into other XML documents.

Figure 4 shows how the translation works: we have created some XSL style sheets, where we use XSLT instructions to extract the information from the WS-CDL document, and then the UppAal document is automatically generated. This document can be opened with the UppAal tool, and thus, we can use the model-checker of UppAal to verify some properties of interest. The tool can also run simulations of the model. We have also created some XSL style sheets to perform the same translation for WS-BPEL documents.

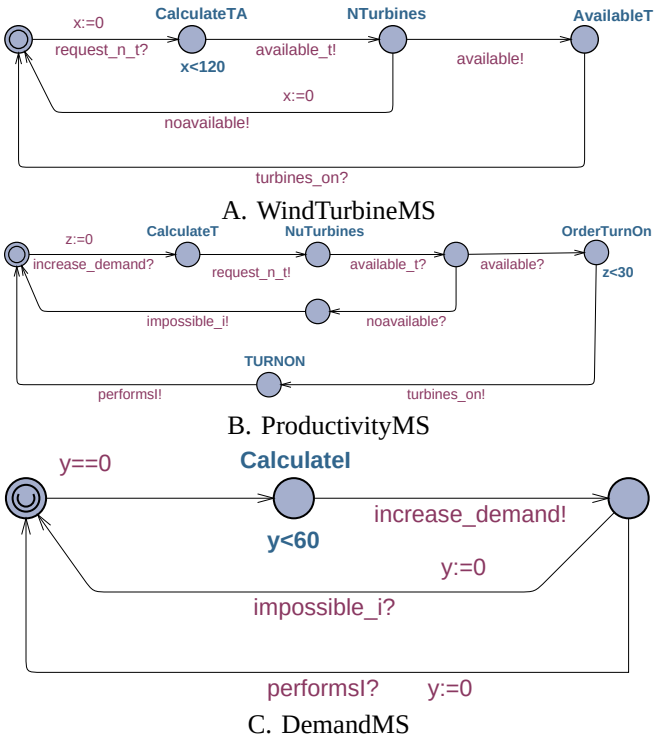


Fig. 4. Wind Mill Management System modeled in UppAal

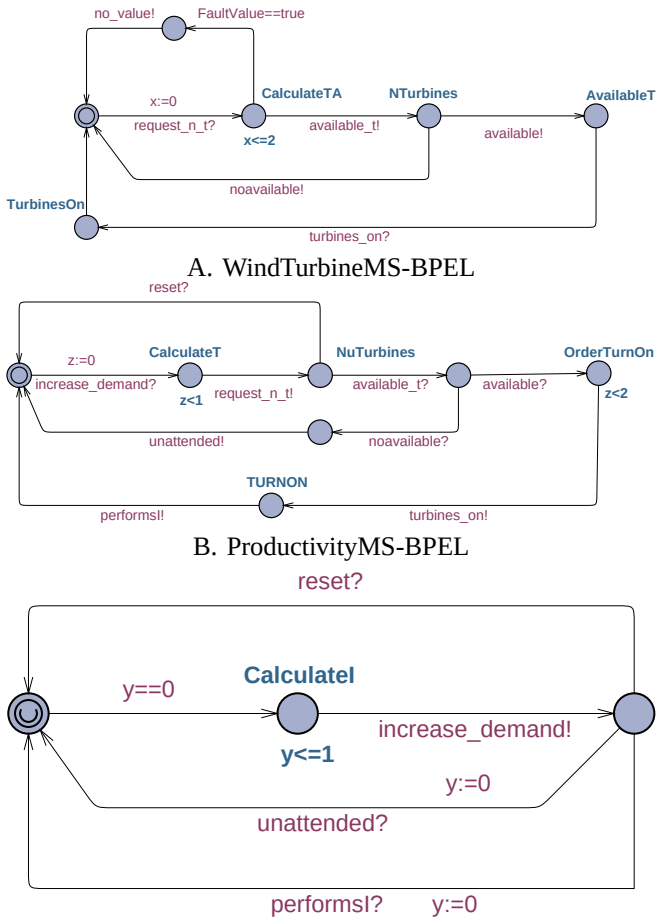


Fig. 5. Wind Mill Management System modeled in UppAal - from BPEL

For the two aspects we can check the following.

General Properties: We check the absence of deadlock for the CDL and for the BPEL; thus we check that the system is able to progress from start to termination; in UppAal this is easily formulated:

$A[]not\ deadlock$

This property holds for both systems.

The system should also be useful. If there are enough available turbines to fulfill the increase of demand, then the Productivity Management system shall send the command to turn on some of them to the Wind Turbine management system. This is formulated as the invariant that says that for all computations (A) and for all states ([]), the two automata locations coincide:

$A[]\ WindTurbineMS.AvailableT \rightarrow ProductivityMS.OrderTurnOn$

This example prpoerty holds as well.

Meeting the demand: Here we check for a BPEL property that the methods can be executed satisfying the contracts or generating the exceptions. For instance, when the demand system sends a message to the productivity system, because it detects an increase in the power demand (the message $increase\_demand$). Also, the Wind Turbine Management system always sends the number of available turbines on Productivity Management system's demand. This is represented in UppAal as follows:

$A[]\ ProductivityMS.NuTurbines \rightarrow WindTurbineMS.CalculateTA$

which holds as well.

Model checking summary: The form of checking that has been shown above is really exhaustive testing. Analysis of what properties to check depends on a systematic inspection of both requirements and the design by some review process, for instance Software Reviews, Code Inspections, and other proactive management processes whose purpose is to eliminate or to find and remove errors in product design as early as possible.

IV. Consistency Checking - Simulation

To check whether the two individually derived models are consistent, we use the concept of (bi-)simulation. A (bi-)simulation is an equivalence relation between state transition systems, associating systems which behave in the same way in the sense that one system simulates the other and vice-versa. The automata generated from the two contract aspects specification systems (WS-CDL, WS-BPEL) turn out to be bi-similar in the following aspects:

- they both accept the same operation sequence; since the WS-CDL specified the protocols, while WS-BPEL contains the operation names but with more information.
- they also accept the same message sequence. Thus, the state that receives the message (e.g. $increase\_demand$

in the example in Figure 4) is followed by a state that sends the message ($request\_n\_t$) inn both automata. The automaton from WS-BPEL may contain some internal states.

We use another model checking tool CWB-NC to check the consistency. We first map the contract captured by both BPEL and CDL to CCS [27], one of the the design languages for CWB, which has communication similar to UppAal; actually, UppAal was developed by people who had prior experience with CCS and the Concurrency Workbench. With the analogous roots, we have not found it useful to spend much time on whether this simple mapping preserves the semantics; it is fairly obvious that it does. More languages such as timed actions version of CCS, CSP, basic lotos, etc are supported as well in the CWB tool; it performs model checking, preorder checking and equivalence checking. As mentioned above, we focus on equivalence checking which allows to identify the behaviourally/observationally equivalent states in a system.

One may ask, why CWB is not used throughout the analysis, since it includes model checking. The answer lies in the lack of state variables; CWB can model the communication structure only, whereas UppAal supports state variables with bounded domains as well as clocks.

Translation from Uppaal to CWB CCS (CDL): We translate the contract specification models in UppAal to a process algebra CCS to allow us to check consistency. The Wind Mill management system consists of 3 processes as shown below:

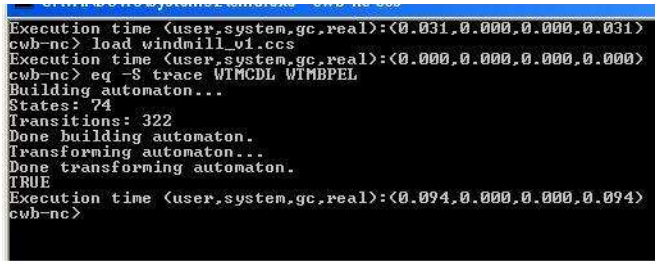
```
proc WTMCDL = (WMC | DMC | PMC)\
{request_n_t, available_t,
 noavailable, available,
 increase_demand, unattended,
 performsI}
```

Processes WMC, DMC, and PMC correspond to Windturbine management system, demand management system and productivity management system respectively as modeled in Figure 5. The three processes communicate through synchronization events. For instance, $request\_n\_t$ in Windturbine management and productivity management.

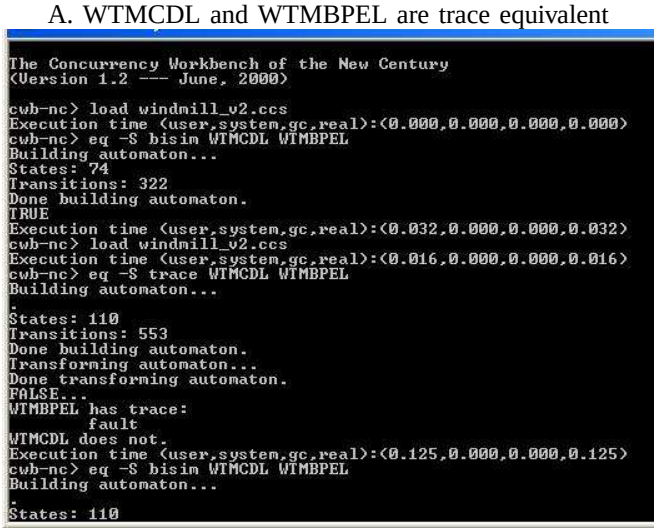
Translation from Uppaal to CWB CCS (BPEL): Similar to the translation of CDL, we translate the contract specification models in UppAal to a process algebra CCS. However, we have more processes from the BPEL contract specifications. These additional processes are fault handlers, compensation handlers and event handlers; but we focus on a fault handler. One can easily add other processes without violating consistency, since they are abstracted away when checking against CDL. In this case, the Wind Mill management system consists of 4 processes as shown below:

```
proc WTMBPEL = (WMC | DMC | PMC | FH)\
{fault, reset
 request_n_t, available_t,
 noavailable, available,
 increase_demand, unattended, performsI}
```

Processes WMC, DMC, and PMC correspond to windturbine management system, demand management system and productivity management system respectively as modeled in Figure 5. The three processes communicate through synchronization events. For instance, *request_n_t* in windturbine management and productivity management.



A. WTMCDL and WTMBPEL are trace equivalent



B. WTMCDL and WTMBPEL are trace bisimilar but not with fault handling

Fig. 6. Consistency Checking using CWB-NC

The simulation results: Figure 6 shows the result of bisimilarity check between CDL and BPEL. The first check, `eq -S bisim WTMCDL WTMBPEL` checks that they are bisimilar. The system has 74 states and 322 transitions. The CWB-NC reports that the processes are bisimilar as well as trace equivalent as shown in Figure 6 A. Recall that the fault handling events are hidden. Hence the bisimilarity. However, when the fault handler is made part of the system, the CWB-NC reports as expected that they are not trace equivalent. The lower part of Figure 6 B shows this result of checking that the two processes are trace equivalent. It shows that the result is $FALSE$ with an additional information that WTMBPEL has trace: $fault$ while WTMCDL does not. Therefore we note that CDL can only be consistent with an abstract version of BPEL where fault handlers are hidden.

V. Comparison with other approaches

Several model checking approaches has been employed to provide some form of analysis. An illustrative example which is well-explained is [28]. It deals with specification in only BPEL where both the abstract model and executable model are specified. The approach is based on Petri nets where a communication graph is generated representing the

process's external visible behaviour. It verifies the simulation between concrete and abstract behaviour by comparing the corresponding communication graphs.

Abouzaid and Mullins [29] propose a BPEL-based semantics for a new specification language based on the π-calculus, which will serve as a reverse mapping to the π-calculus based semantics introduced by Lucchi and Mazzara [30]. The mapping in this work is implemented in a tool integrating the toolkit HAL and generating BPEL code from a specification given in the BP-calculus. Unlike in our approach, this work covers the verification of BPEL specifications through the mappings while the consistency of the new language and the generated BPEL code is yet to be considered. As a future work, the authors plan to investigate a two way mapping. We expect that our approach will be useful in this setting by taking care of the consistency part of their approach.

In [31] the authors have presented an approach different from model checking: a state propagation approach. It uses preconditions and postconditions, and computes weakest execution states. The authors argue that descriptions of preconditions and postconditions are easier and more intuitive compared to linear temporal logic formulae for example. However, similar to the above mentioned approaches, only one language is considered. In this case, consistency checking of Web service function invocations using OWL-S metadata descriptions.

Compared to our approach, the final goal is similar: that is checking of consistency. However, there are some differences in the approach. First, our approach considers more than one language. This is because CDL has a more detailed capture of abstract processes compared to the BPEL abstract processes. Further, BPEL is a programming language to specify the behavior of a participant in a choreography whereas choreography is concerned with describing the message interchanges between participants. In addition, a choreography definition can be used at design time by a participant to verify that its internal processes will enable it to participate appropriately in the choreography. With this, certain properties of individual services can be verified as well as verifying the consistency between the protocols in both BPEL and CDL. This can also be extended with some domain specific languages.

VI. Conclusion

We have presented an approach for the analysis of web service contracts which uses model checking as its prime tool. The analysis is kept manageable by separating contract aspects and analyzing them individually. The price we pay for this aspect oriented analysis is a check for consistency between the individually derived models. However, this check by setting up a bi-simulation between automata can perhaps be automated, because the configurations of the two automata are systematically related through naming conventions and similarities in the WS-CDL and WS-BPEL constructs. The ideas are illustrated with an example specification of a Wind Turbine Management System which consists of three major components (with their services).

In the current contribution, we demonstrate the approach using timed automata as used in the UppAal tool [25], but in other contexts [32] we have experimented with using JML [33] for the functional aspects. We have not touched on verification of timing aspects, although this work was initiated in [9]. Thus the use of UppAal is to some extent a practical decision. We feel that it is well justified for the kinds of analyses that we discuss, because they are concerned with checking the properties of the service as such. For checking implementation conformance, it may not be ideal, and a translation to JML may be much more useful, in particular since Java may be an underlying implementation language, and JML is a formal specification language tailored to Java. Its basic use is thus the formal specification of the behavior of Java program modules. This direction is, however, not the main line of our investigation. The immediate work facing us is to streamline the tool fragments developed for these experiments, and in particular to make true the claim that the bi-simulation can be integrated in a more automated analysis process. It is well known that model checking has its limits, and investigations are also being done of theorem proving approaches [34] which may be more suitable for full implementation conformance checking.

Acknowledgment

The second author is funded by the Nordunet3 Project "Contract-Oriented Software Development for Internet Services".

References

[1] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[2] Y. Lafon and N. Mitra, "SOAP Version 1.2 Part 0: Primer (Second Edition)," W3C, W3C Recommendation, Apr. 2007, http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[3] S. Seely, *SOAP: Cross Platform Web Service Development Using XML*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001, foreword By-Kent Sharkey.

[4] A. Karmarkar, M. Gudgin, M. Hadley, Y. Lafon, J.-J. Moreau, H. F. Nielsen, and N. Mendelsohn, "SOAP version 1.2 part 1: Messaging framework (second edition)," W3C, W3C Recommendation, Apr. 2007, http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[5] B. Meyer, *Object-oriented software construction (2nd ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[6] J. C. Okika and A. P. Ravn, "Classification of SOA Contract Specification Languages." in *Proceedings of The IEEE International Conference on Web Services (ICWS)*, Sep. 2008, to appear.

[7] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, *BPEL4WS, Business Process Execution Language for Web Services Version 1.1*, IBM, 2003. [Online]. Available: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf

[8] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon, "Web services choreography description language version 1.0," W3C, W3C Working Draft, Dec. 2004, http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/.

[9] G. Diaz, J. J. Pardo, M. E. Cambronero, V. Valero, and F. Cuartero, "Verification of Web Services with Timed Automata," in *Proceedings of First International Workshop on Automated Specification and Verification of Web Sites*, vol. 157. Springer Verlags Electronics Notes in Theoretical Computer Science series, 2005, pp. 19–34.

[10] E. Cambronero, J. C. Okika, and A. P. Ravn, “Analyzing Web Service Contracts - An Aspect Oriented Approach.” in *Proceedings of the International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM’2007)*. IEEE Computer Society Press, November 2007, pp. 149 – 154.

[11] G. Castagna, N. Gesbert, and L. Padovani, “A theory of contracts for web services,” in *PLAN-X ’07, 5th ACM-SIGPLAN Workshop on Programming Language Technologies for XML*, jan 2007.

[12] S. Carpineti, G. Castagna, C. Laneve, and L. Padovani, “A formal account of contracts for Web Services,” in *WS-FM, 3rd Int.l Workshop on Web Services and Formal Methods*, ser. LNCS, no. 4184. Springer, 2006, pp. 148–162.

[13] H. Davulcu, M. Kifer, and I. V. Ramakrishnan, “CTR-S: A Logic for Specifying Contracts in Semantic Web Services,” in *Proceedings of WWW2004*, May 2004, pp. 144–153.

[14] G. Pu, X. Zhao, S. Wang, and Z. Qiu, “Towards the semantics and verification of bpel4ws,” *Electr. Notes Theor. Comput. Sci.*, vol. 151, no. 2, pp. 33–52, 2006.

[15] D. Reeves, B. Grosof, M. Wellman, and H. Chan, “Toward a declarative language for negotiating executable contracts,” in *In Proc. AAAI-99*, 1999.

[16] C. Prisacariu and G. Schneider, “A formal language for electronic contracts,” in *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS’07)*, ser. Lecture Notes in Computer Science, M. Bonsangue and E. B. Johnsen, Eds., vol. 4468. Springer, June 2007, pp. 174–189.

[17] A. Beugnard, J.-M. Jezequel, N. Plouzeau, and D. Watkins, “Making Components Contract Aware,” *Computer*, vol. 32, no. 7, pp. 38–45, 1999.

[18] B. Meyer, *Eiffel: the language*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992.

[19] R. Heckel and M. Lohmann, “Towards contract-based testing of web services,” 2004.

[20] A. Keller and H. Ludwig, “The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services,” *Journal of Network and Systems Management*, vol. 11, no. 1, pp. 57–81, March 2003.

[21] “Web Services Agreement Specification (WS-Agreement),” https://forge.gridforum.org/projects/graap-wg/document/WS-AgreementSpecification/en/7, 2004.

[22] “Web Services Architecture,” W3C Working Group Note, www.w3.org/TR/ws-arch/, Feb 2004.

[23] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web Services Description Language (WSDL) 1.1*, 1st ed., W3C, March 2001, URL: http://www.w3c.org/TR/wsdl.

[24] D. Booth and C. K. Liu, “Web services description language (WSDL) version 2.0 part 0: Primer,” W3C, Candidate Recommendation, March 2006.

[25] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, “UPPAAL in 1995,” in *Tools and Algorithms for Construction and Analysis of Systems*, 1996, pp. 431–434. [Online]. Available: citeseer.ist.psu.edu/article/bengtsson96uppaal.html

[26] J. Clark, “XSL Transformations (XSLT) Version 1.0,” W3C, Tech. Rep. REC-xml-19980210, 1998, http://www.w3.org/TR/xslt. [Online]. Available: citeseer.nj.nec.com/bray98extensible.html

[27] R. Milner, *Communication and Concurrency*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[28] A. Martens, “Consistency between executable and abstract processes,” in *EEE ’05: Proceedings of the 2005 IEEE International Conference on e- Technology, e-Commerce and e-Service (EEE’05) on e-Technology, e-Commerce and e-Service*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 60–67.

[29] F. Abouzaid and J. Mullins, “A calculus for generation, verification and refinement of bpel specifications,” *Electron. Notes Theor. Comput. Sci.*, vol. 200, no. 3, pp. 43–65, 2008.

[30] R. Lucchi and M. Mazzara, “A pi-calculus based semantics for ws-bpel,” *J. Log. Algebr. Program.*, vol. 70, no. 1, pp. 96–118, 2007.

[31] T. Kaizu, T. Noro, and T. Tokuda, “A state propagation method for consistency checking of web service function invocations in web applications,” in *ICWE ’06: Workshop proceedings of the sixth international conference on Web engineering*. New York, NY, USA: ACM, 2006, p. 18.

[32] GI-Dagstuhl, “Modelling contest: Common component modelling example (cocome).” [Online]. Available: http://agrausch.informatik.uni-kl.de/CoCoME

[33] J. Leavens, “JML’s rich, inherited specification for behavioural subtypes,” in *Proc. 8th International Conference on Formal Engineering Methods (ICFEM06)*, ser. LNCS, vol. 4260. Springer, 2006.

[34] P. Giambiagi, O. Owe, A. P. Ravn, and G. Schneider, “Language-based support for service oriented architectures: Future directions,” in *ICSOFT (1)*, J. Filipe, B. Shishkov, and M. Helfert, Eds. Setúbal, Portugal: INSTICC Press, September 2006, pp. 339–344.

APPENDIX A: WS-CDL DESCRIPTION OF THE DEMAND MANAGEMENT SYSTEM

```
<?xml version="1.0" encoding="UTF-8"?>
<package author="SCTR Group" name="" version="1.0">

<token name="WindTurbineRef" informationType="StringType"/>
<token name="ProductivityRef" informationType="StringType"/>
<token name="DemandRef" informationType="StringType"/>

<roleType name="WindTurbineRoleType">
<description type="description"/>
<behaviour name="WindTurbineBehaviour"/>
</roleType>

<roleType name="ProductivityRoleType">
<description type="description"/>
<behaviour name="ProductivityBehaviour"/>
</roleType>

<roleType name="DemandRoleType">
<description type="description"/>
<behaviour name="DemandBehaviour"/>
</roleType>

<relationship name="DemandProductivity">
<role type="DemandRoleType"/>
<role type="ProductivityRoleType"/>
</relationship>

<relationship name="ProductivityWindTurbine">
<role type="ProductivityRoleType"/>
<role type="WindTurbineRoleType"/>
</relationship>

<channelType name="Demand2ProductivityChannelType">
<role type="ProductivityRoleType"/>
<reference>
<token name="ProductivityRef"/>
</reference>
</channelType>

<channelType name="Productivity2WindTurbineChannelType">
<role type="WindTurbineRoleType"/>
<reference>
<token name="WindTurbineRef"/>
</reference>
</channelType>

<choreography>
<relationship type="DemandProductivity"/>
<relationship type="ProductivityWindTurbine"/>

<variableDefinitions>
<variable name="Demand2ProductivityChannel"
  channelType="Demand2ProductivityChannelType"/>
<variable name="Productivity2WindTurbineChannel"
  channelType="Productivity2WindTurbineChannelType"/>

<variable name="Available" informationType="xsd:boolean"
  roleTypes="Productivity"/>
<variable name="WindTurbineClock"
  informationType="tns:Clock" roleTypes="WindTurbine"/>
<variable name="DemandClock" informationType="tns:Clock"
  roleTypes="Demand"/>
<variable name="ProductivityClock"
  informationType="tns:Clock" roleTypes="Productivity"/>
<variable name="detectedincreaseDone"
   informationType="tns:boolean" roleTypes="Demand"/>
</variableDefinitions>

<assign roleType="Productivity">
```

```
<copy name="Available_assign">
<source expression="true"/>
<target variable="Available"/>
</copy>
</assign>

<assign roleType="Demand">
<copy name="detectedincrease">
<source expression="false"/>
<target variable="detectedincreaseDone"/>
</copy>
</assign>

<sequence>
  <workunit name="demand increase detected"
    guard="cdl:equal(
       cdl:getVariable('tns:DemandClock'),
          '','',),'0:00')" block="true">
  <assign roleType="DemandRoleType">
  <copy name="calculateincrease"
        causeException="true">
  <source variable="true"/>
  <target variable=
     "cdl:getVariable('detectedincreaseDone',
        '','')"/>
  </copy>
  </assign>
</workunit>

  <interaction name="Demand management system"
    operation="sendIncreasing"
    channelVariable="Demand2ProductivityChannel">
  <participate relationshipType="DemandProductivity"
    fromRole="DemandRoleType"
    toRole="ProductivityRoleType"/>
  <exchange name="CalculatedIncreasing"
    action="request"/>
  <timeout time-to-complete= "cdl:minor(
      cdl:getVariable('tns:DemandClock',
          '','',),'0:01')"/>
  </interaction>

  <interaction name="RequestTurbines_interaction"
    operation="RequestTurbines"
    channelVariable="Productivity2WindTurbineChannel">
  <participate
    relationshipType="ProductivityWindTurbine"
    fromRole="ProductivityRoleType"
    toRole="WindTurbineRoleType"/>
  <exchange name="RequestTurbinesExchange"
    action="request"/>
  <timeout time-to-complete= "cdl:minor(
      cdl:getVariable('tns:ProductivityClock','',''),
      '0:02')"/>
  </interaction>

  <interaction name="AvailableTurbines_interaction"
    operation="AvailableTurbines"
    channelVariable="Productivity2WindTurbineChannel">
  <participate
    relationshipType="WindTurbineProductivity"
    fromRole="WindTurbineRoleType"
    toRole="ProductivityRoleType"/>
  <exchange name="AvailableTurbinesExchange"
    action="request"/>
  </interaction>

  <choice>
    <workunit name="alt_else1_if"
      guard="Available == true" block="true">
      <interaction name="TurbinesOn_interaction"
        operation="TurbinesOn"
        channelVariable="Productivity2WindTurbineChannel">
      <participate
        relationshipType="ProductivityWindTurbine"
        fromRole="ProductivityRoleType"
        toRole="WindTurbineRoleType"/>
      <exchange name="TurbinesOnExchange"
        action="request"/>
      </interaction>
    </workunit>
    <workunit name="alt_else1_else"
      guard="Available != true" block="true">
      <interaction name="Imposible_interaction"
        operation="Impossible"
        channelVariable="Demand2ProductivityChannel">
      <participate relationshipType="ProductivityDemand"
        fromRole="ProductivityRoleType"
        toRole="DemandRoleType"/>
      <exchange name="ImposibleExchange"
        action="request"/>
      </interaction>
    </workunit>
  </choice>
</sequence>
</choreography>
</package>
```

Appendix B: CCS Description of the Wind Mill Management system in CDL and BPEL

```
*********************************************
*  This models the Wind Mill Management System
*
*  CDL system is consistent with abstract BPEL
*
*********************************************

****  CDL Specification Description  ***********

proc WTMCDL = (WMC | DMC | PMC)\
{request_n_t, available_t,
noavailable, available,
increase_demand, unattended, performsI}
*********************************************

proc WMC =
request_n_t.'available_t.('noavailable.WMC
+ 'available.WMC)

proc PMC =
increase_demand.'request_n_t.available_t.
(available.'performsI.PMC
+ noavailable.'unattended.PMC)

proc DMC =
increase_demand.(unattended.DMC + performsI.DMC)

************************ BPEL  ******************

proc WTMBPEL = (WMC | DMC | PMC | FH)\{fault, reset
request_n_t, available_t,
noavailable, available,
increase_demand, unattended, performsI}
*************************************************

proc FH = fault.'reset.FH

proc WMB =
request_n_t.('novalue.WMB + 'available_t.
('noavailable.WMB + 'available.turbines_on.WMB))

proc PMB =
increase_demand.'request_n_t.
('reset.PMB + (available_t.
(available.'turbines_on.'performsI.PMB
+ noavailable.'unattended.PMB)))

proc DMB =
increase_demand.('reset.DMB +
(unattended.DMB + performsI.DMB))
```