Model Aware Execution of Composite Web Services

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Karolina Zurowska

Permission to Use

Abstract

In the Service Oriented Architecture (SOA) services are computational elements that are published, discovered, consumed and aggregated across platform and organizational borders. The most commonly used technology to achieve SOA are Web Services (WSs). This is due to standardization process (WSDL, SOAP, UDDI standards) and a wide range of available infrastructure and tools. A very interesting aspect of WSs is their composeability. WSs can be easily aggregated into complex workflows, called Composite Web Services (CWSs). These compositions of services enable further reuse and in this way new, even more complex, systems are built.

Although there are many languages to specify or implement workflows, in the service-oriented systems BPEL (Business Process Execution Language) is widely accepted. With this language WSs are orchestrated and then executed with specialized engines (like ActiveBPEL). While being very popular, BPEL has certain limitations in monitoring and optimizing executions of CWSs. It is very hard with this language to adapt CWSs to changes in the performance of used WSs, and also to select the optimal way to execute a CWS.

To overcome the limitations of BPEL, I present a model-aware approach to execute CWSs. To achieve the model awareness the Coloured Petri Nets (CPN) formalism is considered as the basis of the execution of CWSs. This is different than other works in using formal methods in CWSs, which are restricted to purposes like verification or checking of correctness. Here the formal and unambiguous notation of the CPN is used to model, analyze, execute and monitor CWSs. Furthermore this approach to execute CWSs, which is based on the CPN formalism, is implemented in the model-aware middleware. It is also demonstrated how the middleware improves the performance and reliability of CWSs.

Contents

List of Tables

List of Figures

List of Abbreviations

BPEL	Business Process Execution Language
CPN	Coloured Petri Nets
CWS	Composite Web Service
HTTP	Hypertext Transfer Protocol
QoS	Quality of Service
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
WS	Web Service
WSDL	Web Service Description Language

Chapter 1

Introduction to Web Services technology

The Service Oriented Architecture (SOA) is a design and integration approach that uses services as computational elements. The basic principle that underlies SOA is presented in Figure 1.1. According to this, an abstract description of a service, understood as software functionality, is published in a register (discovery facility). There it can be found by a requestor that wants to use it. The requestor, after finding the description, binds to the service, so it obtains enough information to connect over a network to it and consume it [44].

The most widely accepted technology that adapts SOA is the Web Services (WSs) technology. The notion of a WS refers to systems that are built from several networked modules, however it also refers to a set of standards, which supports an implementation of such applications [48]. In the first approach a WS is defined as an interface that groups operations accessible over the network, and this access is possible with standard XML messages [22]. In the second approach a WS is a software operation described with the WSDL (Web Service Description Language [41]), which is invoked over a network using SOAP (Simple Object Access Protocol [40]). These operations can be also published and discovered with UDDI (Universal Description, Discovery and Integration [29]).

The main advantage of WSs is their interoperability, which means that they allow communication and cooperation between software components implemented in different programming languages or deployed on different platforms. WSs achieve this by using the already mentioned set of standards (e.g. WSDL, SOAP) as well as by relying on XML-based artifacts for describing, publishing and invoking activities. Choosing XML as the underlying language is an important element of ensuring the interoperability, because XML is machine and platform independent.

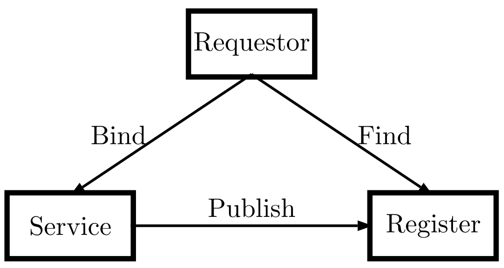


Figure 1.1: The concept of SOA.

1.1 Introduction to Composite Web Services

Besides interoperability the other important attribute of WSs is their composability (aggregation). WSs can be easily aggregated (orchestrated or choreographed) to work together, in order to provide more complex functionalities. In most cases the goal of such composition is to model a business process or workflow, like supply-production chains or planning services. There are two methods to compose WSs: orchestration and choreography [32]. Orchestration is an executable business process that interacts with other WSs, and is controlled by one party. Choreography is more collaborative, and it allows involved parties to define their role in interactions, and it tracks sequences of exchanged messages. For both types of modeling business processes the BPEL (Business Process Execution Language [28]) specification is mostly accepted. Other specifications that serve the same purpose are Business Process Management Language (BPML) together with Web Services Choreography Interface (WSCI) [32]. Besides the above there are other languages that allow specification of more abstract processes or workflows, because this field is researched since 70s [14].

Although there exist many possibilities, the most common way to implement a composition of WSs, a Composite Web Service (CWS), is to specify it in BPEL [28]. BPEL is an XML-based language that uses elements called partner links to refer to external WSs. In BPEL an activity is a computational unit, and it is either basic or structured. The basic activities are: invoke, receive and reply operations to interact with WSs, assignment operators to modify variables, waiting and an empty activity. The second type of activities (structured) contains: sequence, switch, while, flow and pick (awaits for specified events). All BPEL activities are enclosed in a scope, for which there can be defined fault and compensation handlers to deal with special situations. In Listing 1.1 there is an example of the process definition in BPEL.

Listing 1.1: An example of BPEL process.

```
<bpel:process
    xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
    xmlns:flight="http://localhost:8080/axis/services/FindFlight"
    xmlns:vp="http://tempuri.org/vactionplanner"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    name="BPELExample"  suppressJoinFailure="yes"
    targetNamespace="http://BPELExample">

  <bpel:partnerLinks>
     <bpel:partnerLink  myRole="planner"  name="VacationPlannerLinkType"
        partnerLinkType="vp:VacationPlannerLinkType"/>
     <bpel:partnerLink
        name="FlightFinderLinkType"
        partnerLinkType="flight:FlightFinderLinkType"
        partnerRole="flight_finder"/>
  </bpel:partnerLinks>

  <bpel:variables>
    <bpel:variable messageType="vp:planVacationRequest"
      name="planVacationRequest"/>
    <bpel:variable messageType="vp:planVacationResponse"
      name="planVacationResponse"/>
    <bpel:variable messageType="flight:findFlightRequest"
      name="findFlightRequest"/>
    <bpel:variable messageType="flight:findFlightResponse"
      name="findFlightResponse"/>
  </bpel:variables>

  <bpel:sequence>
    <bpel:receive createInstance="yes"
      operation="planVacation"
      partnerLink="VacationPlannerLinkType"
      portType="vp:vactionPlannerPortType"
      variable="planVacationRequest"/>
    <bpel:assign>
       <bpel:copy>
          <bpel:from part="destination" variable="planVacationRequest"/>
          <bpel:to part="destination" variable="findFlightRequest"/>
       </bpel:copy>
    </bpel:assign>
   <bpel:invoke inputVariable="findFlightRequest"
      operation="findFlight"
      outputVariable="findFlightResponse"
      partnerLink="FlightFinderLinkType"
      portType="flight:FindFlight"/>
   <bpel:assign>
       <bpel:copy>
          <bpel:from part="findFlightReturn" variable="findFlightResponse"/>
          <bpel:to part="planVacationReturn" variable="planVacationResponse"/>
       </bpel:copy>
    </bpel:assign>
    <bpel:reply operation="planVacation"
      partnerLink="VacationPlannerLinkType"
      portType="vp:vactionPlannerPortType"
      variable="planVacationResponse"/>
  </bpel:sequence>
</bpel:process>
```

The process in Listing 1.1 calls an external WS, which returns a flight number for the given destination name. In its BPEL implementation two partner links (for the CWS and the external WS) and a set of variables are declared. The sequence activity specifies the behavior of the process.

First it receives the message, with its input parameter, and copies this parameter to the variable that is an input of the external WS. The response from this WS is then assigned to the response message from the CWS, which is returned in the reply element. This process definition requires the WSDL description of the external WSs. An example of such a description is presented in Listing 1.2.

Listing 1.2: An example of WSDL description.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
     targetNamespace="http://localhost:8080/axis/services/FindFlight"
     xmlns:apachesoap="http://xml.apache.org/xml-soap"
     xmlns:impl="http://localhost:8080/axis/services/FindFlight"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <wsdl:message name="findFlightResponse">
      <wsdl:part name="findFlightReturn" type="xsd:string"/>
   </wsdl:message>

   <wsdl:message name="findFlightRequest">
      <wsdl:part name="destination" type="xsd:string"/>
   </wsdl:message>

   <wsdl:portType name="FindFlight">
      <wsdl:operation name="findFlight" parameterOrder="destination">
         <wsdl:input message="impl:findFlightRequest" name="findFlightRequest"/>
         <wsdl:output message="impl:findFlightResponse" name="findFlightResponse"/>
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:binding name="FindFlightSoapBinding" type="impl:FindFlight">
      <wsdlsoap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="findFlight">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="findFlightRequest">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://services" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="findFlightResponse">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://localhost:8080/axis/services/FindFlight"
            use="encoded"/>
         </wsdl:output>
```

```
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="FindFlightService">
    <wsdl:port binding="impl:FindFlightSoapBinding" name="FindFlight">
      <wsdlsoap:address
           location="http://localhost:8080/axis/services/FindFlight"/>
    </wsdl:port>
  </wsdl:service>

  <plnk:partnerLinkType
   xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
      name="FlightFinderLinkType">
   <plnk:role name="flight_finder">
      <plnk:portType name="impl:FindFlight"/>
   </plnk:role>
  </plnk:partnerLinkType>

</wsdl:definitions>
```

The WSDL description from Listing 1.2 defines 2 types of messages: request and response. These messages are then used as an input and output of the operation "FindFlight". All operations offered by the WS are gathered in an element "portType", which in this case contains only the "FindFlight" operation. Elements "binding" and "service" specify the details of the deployment of this service: how parameters should be sent and what is the address. The process in BPEL (Listing 1.1) along with WSDL descriptions of external WSs (Listing 1.2) are then deployed in the BPEL engine (for example Active BPEL [1]) and can be consumed by clients.

This thesis presents the alternative approach to compose and execute CWSs, which uses one of the formal methods, namely the Coloured Petri Nets (CPN). The remainder of the thesis is organized as follows. In Chapter 2 the motivation for this reasearch is presented. Chapter 3 gives the introduction to the CPN formalism and the related work in the area of using formal methods in the context of CWSsc, as well as other approaches to compositions of services. This is followed by the description of how the CPN is used to model, analyze, execute and plan the execution of CWSs. Chapter 5 shows how, based on the above concept, the model-aware middleware is implemented, and then, Chapter 6, presents the evealuation of this solution. The conclusions and future work are in Chapter 7.

Chapter 2

Problem definition

Composite Web Services (CWSs) support complex and flexible systems by combining services exposed by different providers. Figure 2.1 presents how such a composition can be viewed. "CWS A", since it is also a WS, responds to a message "request". It uses a set of external WSs (WS 1 to WS n), and then produces a "response" message. Of course, it must be specified how the external WSs are invoked, what the order of the performed operations is, and how the result is generated. To define those details the BPEL language [28] may be used, and replace the "CWS A" box with the appropriate XML-based specification. It is then executed using one of the BPEL engines (for example ActiveBPEL [1]). This solution is sufficient to implement and execute CWSs.

However the BPEL-based scenario is insufficient if we want to:

(1) verify the correctness of the CWS implementation (is it free of deadlocks? can it produce a correct result?),

(2) compare different paths of successful execution,

(3) analyze which external WSs are essential to successfully execute a CWS and which are not,

(4) observe the state of an execution (which operations are done already, which are left).

To reach these goals a CWS must be represented in another way than with BPEL. It should allow to capture and prove some properties of CWSs in a definite way. One of the solutions is to use formal methods. A formal method is a tool or a notation that introduces a mathematically grounded semantics, in order to specify a software system unambiguously and to prove its properties [19].

In the context of CWSs several formal methods are used, most commonly:

- state transitions models that can capture protocols for conversations between WSs [5],
- finite state guarded automata with queues for incoming messages [17],
- process algebras [16],
- Petri Nets [20] and hierarchical Coloured Petri Nets [45].

In the works presented above formal methods are used to verify and prove the correctness of compositions (point (1) in the previous list). Hence the general question is: *Can we use formal*

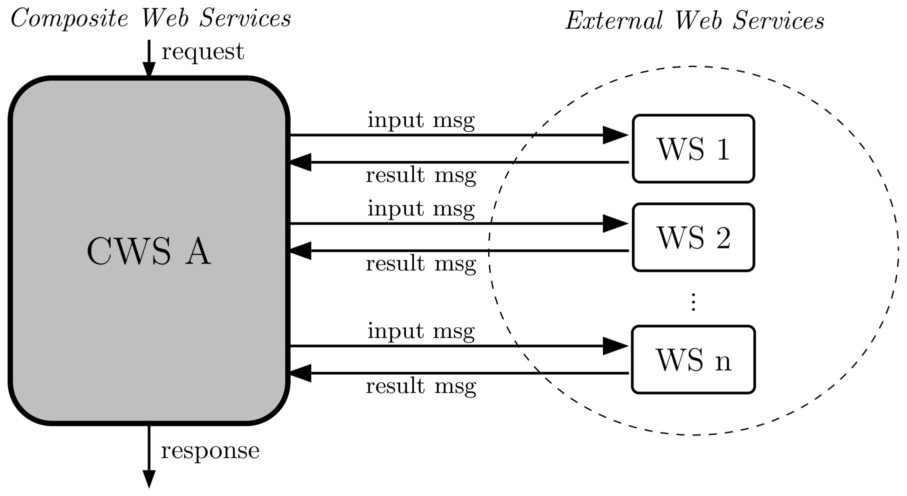


Figure 2.1: General approach to Composite Web Services.

methods to construct the model of a CWS and use it as the source of knowledge about the CWS itself and its executions (points (2) - (4))?

To answer this question we must first decide which formalism to use to represent and to model a CWS. In this work the Coloured Petri Nets (CPN) language was chosen, since it is suitable for design, specification and the simulation of systems [23]. The language has features especially useful in the CWS area of interest: the possible hierarchical structure of nets, the notion of colors that can deal with data and dual graphical and algebraic representation. So if CWSs are modeled with the CPN, such models are then the sources for knowledge about the compositions. This poses another question: *What kind of information we can get/infer with the CPN model of a CWS?*

First, we are able to analyze all possible paths of a CWS execution. This task is performed with one of the CPN analysis methods, namely state space analysis [24]. For each CPN model we construct a graph (called an occurrence graph) that represents all possible states of the net and events that move it from one state to another. In this way we know which states can result in successful execution of a CWS, and whether there is only one possible path to do it, or there are more paths.

Second, we know which interactions with external WSs are essential to successfully execute a CWS, and which are optional. This knowledge is also inferred from an occurrence graph constructed for the CWS model. It is done by means of reachability analysis. So it is known whether the state of successful execution is reachable from states that represent failures of interactions with external WSs. The notion of essential and optional interactions is significant, because it allows avoiding optional interactions with components that do not work properly, hence can be slow.

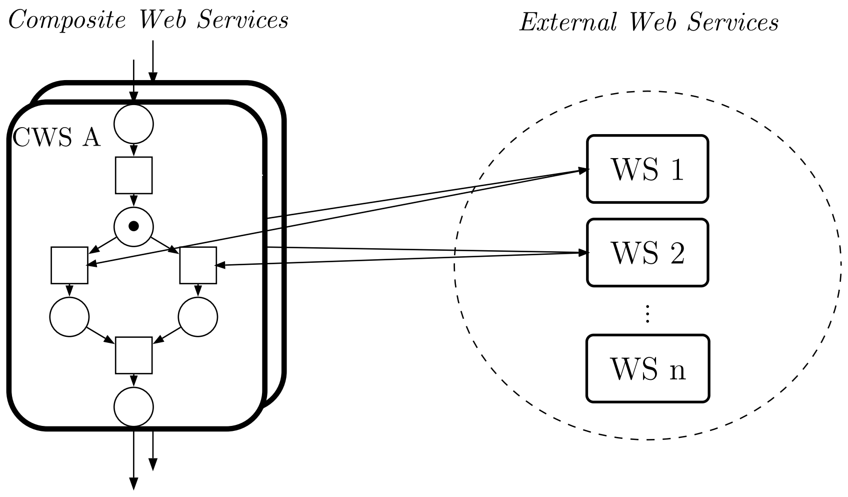


Figure 2.2: General approach to a CWS modeled with Petri Nets.

Third, we know in which state each execution of a CWS is. It means that operations that this CWS must perform are known. More specifically it is also known with which external WSs the CWS is going to interact. In turn future calls to external WSs are predicted and they can be optimized, for example to avoid overloading their servers.

Figure 2.2 presents the above approach to composing CWSs. For the sake of brevity the CWS is represented with ordinary Petri Nets, and inscriptions for messages are omitted. The Petri Nets model of "CWS A" has replaced the gray box from Figure 2.1. Therefore it is known in which order services are called, and what is the current state of the execution (marked as a black token). In this straightforward example it can be also determined, without state space analysis, that there are two possible paths of execution, and that calls to the external WSs are essential. Nevertheless for more complex CWSs these conclusions should be inferred in an unambiguous way from the occurrence graphs. The next question is then: *How the knowledge from the model of a CWS is used to make CWS more efficient and reliable?*

There are different possible ways to improve efficiency and reliability with the model of CWS. Some of them include:

(1) *dynamic service selection* - the model of a CWS enables a dynamic binding of instances of external WSs, based on previous and possible future interactions with other WSs,

(2) *omitting optional and faulty external WS* - services which are optional and which are faulty can be omitted, without affecting the overall CWS execution,

(3) *choosing a path of execution* - if there is more than one path to successfully execute a CWS, the selection of the optimal one is supported with the knowledge about previous calls to WSs and future states of all executed CWSs.

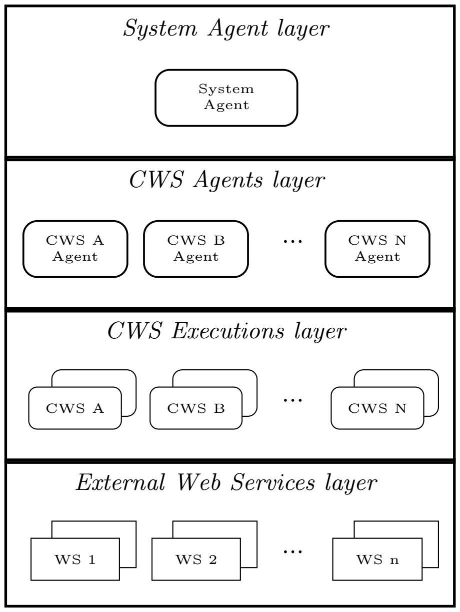


Figure 2.3: General architecture of model-aware CWS.

To facilitate the improvements stated above, an abstract architecture with appropriate capabilities is presented in Figure 2.3. It consists of several layers. At the bottom there are gathered external WSs - for each of them there are one or more instances. These services are used by a CWS during its execution, and at any time there might be several executions of the CWS. Each CWS has an agent that is responsible for reasoning about the current and future state of the CWS execution. At the top of the architecture there is the system agent that reasons about all executed CWSs and external WSs.

The abstract architecture presented in Figure 2.3 must be defined in more detail. So there are the following problems to solve:

(i) how CWSs should be modeled with the CPN?

(ii) how to use features of the CPN to analyze CWSs and to monitor their executions?

(iii) how reasoning capabilities of agents should be defined to allow improvements?

Finally the presented model-aware CWS middleware should be evaluated whether it improves the performance of CWSs.

Generally the issues that are analyzed in this work are: *Can we use the CPN formalism, not only to prove the formal properties of CWSs, but also as the basis of improving the execution of CWSs? Can we make this execution not only aware of its model but also aware of other executions of CWSs and external WSs? Is such awareness beneficial for CWSs executions?*

Chapter 3

Related work

This work considers how using the model of a Composite Web Service (CWS) at runtime can improve its execution. To represent CWSs the Coloured Petri Nets (CPN) language was chosen. The CPN formalism has properties, like the ability to model hierarchies or to represent data, which are important in the context of executing CWSs. In the following sections these and the other features of the CPN are presented. There is also shown how in other works in CWSs area of interest Petri Nets and the CPN are used.

The main purpose of using the model of CWSs is to improve their performance by making them more adaptive to the changing environment. The approach presented in this work is not the only one, there are also other solutions in this area. They are presented later in two aspects: the dynamic service selection and the dynamic changes in the process definition.

3.1 Coloured Petri Nets

The CPN (which represent high level Petri Nets) is a formalism used as a language for designing, modeling and implementing systems [23]. It has a graphical representation and, associated with it, an algebraic one. The CPN language is also a formal method, so besides the representation it has an unambiguous semantic. In the following sections the general definition of the CPN and analysis methods are presented.

3.1.1 Definition of Coloured Petri Nets

The CPN, as well as other types of Petri Nets, are both state and action oriented. It means that they can represent states (with entities called *places*) of a system, and also actions (with entities called *transitions*). Additionally the entities of the same type cannot be connected, so places are connected only with transitions and the opposite. The state and action dualism enables the support for the principle of locality, which means that the behavior of an action (transition) depends only on the action itself and its input and output objects [18].

A place in the CPN is characterized by its name and associated type (also called color set) [26]. The name has no formal meaning, but it should be chosen to support readability of the net. The

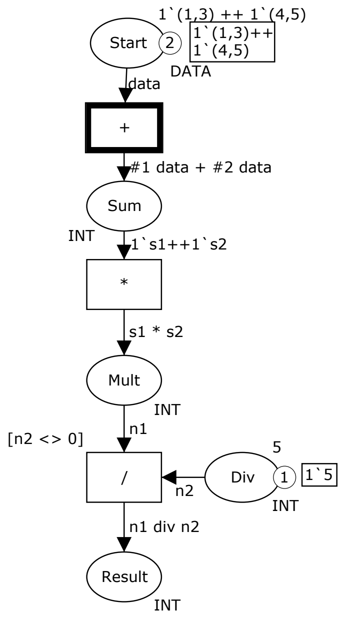


Figure 3.1: The example of the CPN - initial marking M_0.

type is more important, since it determines the kind of data, which are stored in a place. While executing the net, places contain a number of *tokens*, each token is an element from the type of an appropriate place. In a place several tokens may have the same value, because they are multi-sets over the type. A multi-set is a set, which can have duplicates of the elements. To indicate how many duplicates there are a coefficient for each element is specified. For example $1`A + 3`B$ is a multi-set over $\{A, B, C\}$, with one A, three B and zero C. The distribution of tokens in all places identifies a state of CPN, and is called a *marking*.

A transition in the CPN is responsible for moving tokens between places [26]. The specification of how many and which ones are moved is in an *arc expression* for arcs connected with a transition. Each arc expression is evaluated to a multi-set over the type of the place connected to the arc, and this multi-set is either removed (input places) or added (output places) to a place. In an arc expression it is possible to have variables, thus to determine which tokens are moved, the variables must be bound to values. If all all variables have a binding, the transition can be *enabled* or not in this binding. A transition is enabled if in all its input places (that are connected with it by incoming arcs), there are tokens required after evaluation of arc expressions. If a transition is enabled it is possible that it occurs (it is *fired*), and then tokens are removed from input places and added to output places of the transition.

Figure 3.1 shows a net that illustrates the above concepts, which calculates the expression $\frac{(a+b)(d+e)}{f}$. It consists of five places (graphically represented as ovals): four of them: Sum, $Mult$, Div, $Result$ have the type integer (INT) and the last one, $Start$, has the type that is the product of 2 integers ($DATA$). There are three transitions (graphically represented as rectangles): $+$, $*$ and

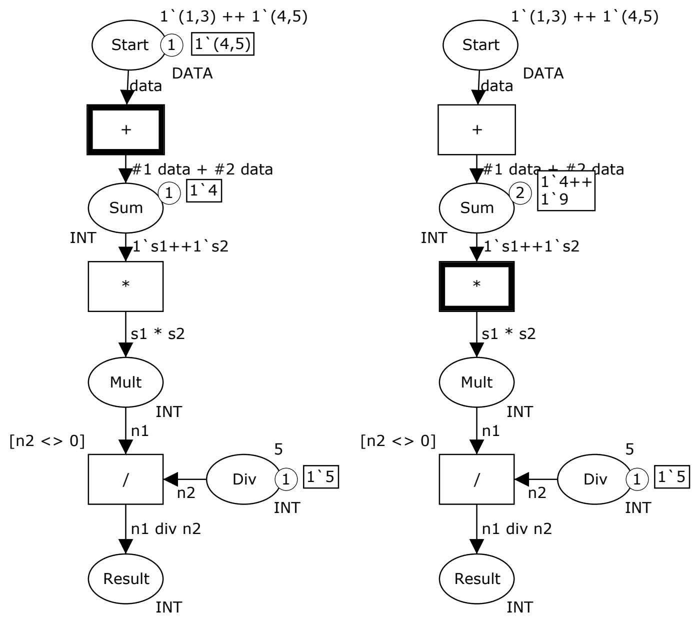


Figure 3.2: The example of the CPN - markings M_1 and M_2.

/, which model necessary operations. There are also variables in the net: *data* of $DATA$ type, and $s1, s2, n1, n2$ of INT type, all of them are in the arcs expressions. The state of the net is specified by the marking called initial (denoted as M_0). In this marking there are two tokens $1`(1,3)$ and $1`(4,5)$ in the place *Start* and one token $1`5$ in the place *Div* (tokens are graphically represented as circles next to a place they belong to). In the initial marking the + transition is enabled with 2 possible bindings: $< data = (1,3) >$ or $< data = (4,5) >$. This is, because after evaluating the incoming arc expression (which is *data*) with those bindings, there is the required token in *Start*, which is the transition's input place.

The pairs $(+, < data = (1,3) >)$ and $(+, < data = (4,5) >)$ are called *binding elements*, and both are enabled in M_0. If the transition + occurs with $< data = (1,3) >$, it results in the marking M_1 shown in the left part of Figure 3.2. The occurrence of the + transition means that the token with value $(1,3)$ is removed from the *Start* place, since it is the token value obtained after evaluating the incoming arc inscription (*data*) with the binding $< data = (1,3) >$. The outgoing arc expression for the + transition is $\#1\ data + \#2\ data$, and it must be evaluated in the current binding. This expression means to take the first element of the *data* variable and add it to the second one, so for the binding $< data = (1,3) >$ it is the token with value 4. This token appears in the place *Sum*.

In the marking M_1 only the binding element $(+, < data = (4,5) >)$ is enabled. The $*$ transition is not enabled, because there is only one token in the *Sum* place, and the incoming arc expression for it requires two tokens (one with a value of the variable $s1$ and one with a value of $s2$). However if the + transition with the binding $< data = (4,5) >$ occurs, it results in the marking M_2 shown

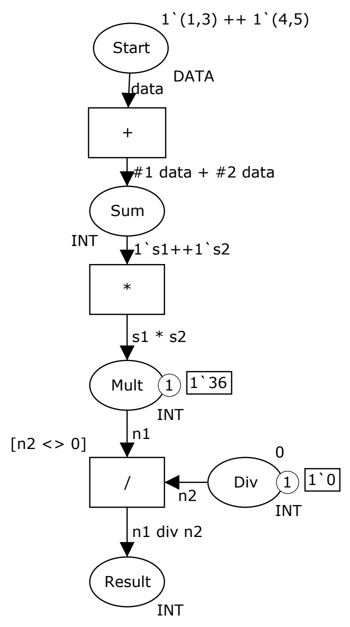


Figure 3.3: The example of CPN and the usage of guards - markings M_3.

in the right part of Figure 3.2. In this marking for the binding $< s1 = 4, s2 = 9 >$ the $*$ transition is enabled, because its input place *Sum* contains appropriate tokens (1‘4 + +1‘9).

Markings M_0, M_1 and M_2 form the *finite occurrence sequence* [26], for which we can use the following notation:

$$M_0\ [Y_1>\ M_1\ [Y_2>\ M_2 \text{ where: } Y_1 = \{(+, < data = (1,3) >)\} \text{ and } Y_2 = \{(+, < data = (4,5) >)\}$$

The presence of this sequence means that markings M_1 and M_2 are reachable from M_0. In general a marking M'' is reachable from M' if there is a finite occurrence sequence between M' and M'' [25]. The set of all markings which are reachable from M' is noted as $[M'>$. In the example $\{M_1, M_2\} \subset [M_0>$. Because M_1, M_2 are reachable from the initial marking M_0, they are also called simply *reachable* markings.

The possible enabling of a transition may be altered by adding a *guard* [23]. A guard is a boolean expression that reduces accepted bindings to the ones that evaluate the guard expression to true. In Figure 3.3 there is the net similar to the previous example, however this time the token in the place *Div* has a value 0 (not 5 as previously). For the marking M_3 the only binding in which the / transition could be enabled is $< n1 = 36, n2 = 0 >$. But according to the guard for the transition ($n2 \neq 0$) the variable $n2$ must be different than 0. In this case the binding element $(/, < n1 = 36, n2 = 0 >)$ is not enabled.

All the previous concepts, besides the graphical representation, have also a formal definition. It is presented in Definition 3.1.1 [23].

Definition 3.1.1. A non-hierarchical CPN is a tuple: CPN $= (\Sigma, P, T, A, N, C, G, E, I)$, which satisfies the following conditions:

(i) Σ - a finite set of types (color sets).

(ii) P - a finite set of places.

(iii) T - a finite set of transitions.

(iv) A - a finite set of arcs, where sets A,P,T are pairwise disjoint.

(v) N - a node function: $N \in [A \rightarrow (P \times T \cup T \times P)]$.

(vi) C - a color function: $C \in [P \rightarrow \Sigma]$.

(vii) G - a guard function: $G \in [T \rightarrow \mathit{Expression}]$, such that

$$\forall t \in T : [\mathit{Type}(G(t)) = \mathbb{B} \wedge \mathit{Type}(\mathit{Var}(G(t))) \subseteq \Sigma],$$

where Type maps an expression or a variable to its type, and Var maps an expression to its variables.

(viii) E - an arc expression function : $E \in [A \rightarrow \mathit{Expression}]$, such that

$$\forall a \in A : [\mathit{Type}(E(a)) = C(p(a))_{MS} \wedge \mathit{Type}(\mathit{Var}(E(a))) \subseteq \Sigma],$$

where a function p maps an arc to its place.

(ix) I - an initialization function : $I \in [P \rightarrow \mathit{Expression}]$, such that

$$\forall p \in P : [\mathit{Type}(I(p)) = C(p)_{MS}].$$

Definition 3.1.1 is the basis to define formally the behavior of the CPN: markings, enablings and occurrence of transitions.

3.1.2 Analysis of Coloured Petri Nets

One way to analyze the net is to execute it by firing transitions with bindings, as was shown for the example net in Figures 3.1, 3.2 and 3.3. However such a method is not enough to prove the correctness of what the CPN represents. The proof of such correctness can be made by checking properties of a net. This in turn requires construction of a state space, which in the CPN is represented with an occurrence graph. It is a graph, which has a node for each marking and an arc for a binding element. Several properties of the net can be inferred from an occurrence graph: boundedness, home and liveness properties [23].
Boundedness properties tell how many tokens there can be in a place (Definition 3.1.2) [25].

Table 3.1: Properties of the net in Figure 3.1.

Boundedness properties:

Place	Integer bounds	Multi-set bounds
Start	2	$1`(1,3) + +1`(4,5)$
Sum	2	$1`4 + +1`9$
Mult	1	$1`36$
Div	1	$1`5$
Result	1	$1`7$

Home and liveness properties:

Place	Home marking	Dead marking
Start	empty	empty
Sum	empty	empty
Mult	empty	empty
Div	empty	empty
Result	$1`7$	$1`7$

Definition 3.1.2. For a place $p \in P$, an integer $n \in \mathbb{N}$ and a multi-set $m \in C(p)_{MS}$:

- n is an integer bound for p iff $\forall M \in [M_0\!>: \; (|M(p)| \leq n)$.

- m is a multi-set bound for p iff $\forall M \in [M_0\!>: \; M(p) \leq m$.

Home properties tell about a marking or a set of markings which are reachable from all markings, so you can always return to them (Definition 3.1.3) [25].

Definition 3.1.3. For a marking $M \in \mathbb{M}$ and a set of markings $X \subseteq \mathbb{M}$:

- M is a home marking iff $\forall M' \in [M_0\!>: \; M \in [M'\!>$.

- X is a home space iff $\forall M' \in [M_0\!>: \; X \cap [M'\!> \neq \emptyset$

Liveness properties tell whether binding elements remain active (Definition 3.1.4) [25].

Definition 3.1.4. For a $M \in \mathbb{M}$ and a set of binding elements $X \subseteq BE$:

- M is dead iff $\forall x \in BE \; : \; \neg M[x\!>$

- X is dead in M iff $\forall M' \in [M\!> \forall x \in X \; : \; \neg M'[x\!>$

- X is live in M iff $\forall M' \in [M_0\!> \exists M'' \in [M'\!> \exists x \in X \; : \; M''[x\!>$

Table 3.1 presents the above properties for the net in Figure 3.1. The integer bounds are 2 for *Start* and *Sum* places and 1 for the other places, the multi-set bounds are different for all places and represent partial results of the expression calculations. The home marking and the dead marking are the same, with the token only in the place *Result*, so we can be sure that the net will terminate only after all calculations are finished.

Occurrence graphs are a very useful analysis method to check many other properties, like: is there a path to reach a particular marking from another one, which path is the shortest one, which transitions occur in that path.

The disadvantage of these graphs is a problem known as the state explosion [24]: even for relatively small nets the graph may become very complex and its analysis may become intractable. There are 2 mechanisms to reduce the size of an occurrence graph: symmetry and equivalence classes [24]. Both of them share the same idea, which is to define a representative marking and a binding element and gather all similar markings and bindings elements with the representative entity. Hence we can still analyze the graph and observe the properties, but without the details.

3.2 Petri Nets and Composite Web Services

Petri Nets are used in CWSs mostly as a modeling language to provide means to verify the composition. The main research areas are: transforming a definition of a process in BPEL to Petri Nets, modeling compositions and analyzing performance and reliability of CWSs.

The transformation between the definition of a CWS in BPEL to Petri Nets is proposed by Hinz et al. [21] and by Ouyang et al. [30]. In the first approach each activity in BPEL is translated into an appropriate Petri Net. The activities form patterns in a Petri Net with interfaces that allow joining them. Ouyang et al. use mappings for translations, and model extensively control links (used for implementing control dependencies in parallel executions). Both works present also implementations of parsers to generate Petri Nets. Hinz at al. use the parser BPEL2PN that generates files for the LoLA model checker [34] to produce an occurrence graph and to check properties of the net. Ouyang et al. propose the BPEL2PNML parser, which produces files that are input for the WofBEPL tool to analyze the net (for example to find unreachable activities). In the work by Hinz et al. there is more emphasis to capture the control flow of the process (they do not model messages), whereas in the approach by Ouyang et al. there is very comprehensive consideration of all BPEL aspects.

The verification of the BPEL process is also proposed by Yang et al. [45]. In this approach the CPN language is used, and rules are provided to transform the process definition. This is then the basis to check the properties like liveness, and to infer whether the process is correct. Yang et al. do not propose any implementation to verify a model of a CWS in CPN, and this is done manually.

Mecalla et al. [27] and Zhang et al. [47] propose Petri Nets as a modeling language for CWSs. In the first research the framework is used to manage a distributed orchestration of WSs. To do it, the authors present the Orchestration Net, which consists of places of different types (orchestration places, input and output message places). This kind of net can be executed on orchestration engines, thus the control is decentralized. In the second research Zhang et al. present WS-Net language, which is based on the CPN but incorporates object-oriented concepts. A composition of services consists of components described as nets, which can communicate via XML messages with the SOAP [40] standard. Components are in a three-layer specification: an interface net (declares services to be provided), an interconnection net (declares services to be acquired from other components) and an interoperation net (describes the functionality of each component).

Petri Nets models are also used to predict the reliability or the performance of CWSs. Zhong et al. [49] propose stochastic Petri Nets to measure the reliability of CWSs. Stochastic Petri Nets are Petri Nets, which have two types of transitions: timed and immediate. The first type has firing rates, and the second firing probabilities. In the approach of Zhong et al. after generating a Petri Net model from BPEL it is annotated with rates and times, and then it is solved. It means that the reliability of the whole CWS is estimated. Stochastic Petri Nets are also used by Tan et al. [35], however they propose the simplification rules to reduce a state space to explore. In this way, although the results are approximate, they are still accurate.

3.3 Adaptive Composite Web Services

WSs are remotely deployed components, so their behavior is not static over time, and additionally they are vulnerable to all failures encountered in distributed systems [11]. In turn CWSs include those characteristics and to avoid propagation of them, they must be able to deal with the rapidly changing environment. The approaches to overcome this problem can be divided into two categories. The first one is based on the dynamic selection of used components, so if a performance of a WS decreases, then another one that is capable of delivering similar functionality is selected. In the second category the definition of the whole CWS is adapted. In the following sections these approaches are studied.

3.3.1 Service selection

In the area of the dynamic service selection lots of work has been done. The differences between propositions are based on used assumptions like: whether it is possible to discover new services during the execution, what does the similarity or equivalence mean, and what attributes are used to score available services and chose the best one. In the following several different solutions in this area are presented.

Benatallah et al. [7] propose the middleware called Self-Serv. It enables dynamic service selection within a service container, which gathers substitutable services. Service containers allow advanced management of services, by definition of membership modes, like an explicit mode (set of WSs

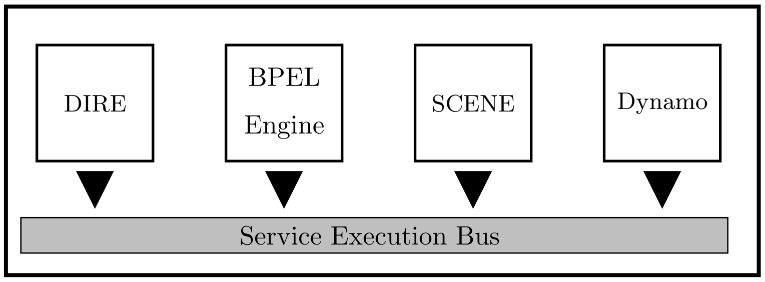


Figure 3.4: Overview of the adaptive framework, where: DIRE - distributed registry, SCENE - service execution environment, Dynamo - dynamic monitoring.

is defined during definition) or a query mode (members are defined in a form of a query to the registry). The criteria used to select the best service are multiattributed with weights assigned to each attribute, they constitute the score for each service. The selection of a service is performed at runtime: a service with the highest score is chosen. In the Self-Serv environment the model of CWSs, which is specified with state-charts, is used as the execution framework [6]. To do this there are defined state coordinators for each state, and they are responsible for notifications about states completions and managing pre- and postconditions of states. The control flow in the CWS is thus based on transition between states.

A framework for CWSs that is adaptive, yet uses WSs standards (WSDL, BPEL, UDDI) was implemented by Baresi et al. [4]. In their work they use a BPEL process specification, which is enhanced with rules that enable the dynamic service discovery or binding at runtime. The choice of a service is based on the criteria defined by a user at the definition time of the process. To allow a more robust discovery and binding of services Baresi et al. propose distributed registries. The framework can also change services as a response to the events gathered during the monitoring. The abstract description of these concepts is presented in Figure 3.4. There are 4 contributors to the execution of a CWS: a process defined in BPEL, in which some of the invocations may be abstract (binding is then performed at runtime), the registry of services that can be distributed between providers of services (DIRE), the execution environment (SCENE) with rules that allow the dynamic discovery and binding, and the monitoring facilities (Dynamo), which produce events to reconfigure the process.

De Antonellis et al. [13] propose a "matchmaking" approach for a dynamic selection of external WSs. They distinguish between abstract and concrete services, which refer respectively to external services used during the definition of workflows and used during the execution. Each abstract service introduces a compatibility class, and then each concrete services associate itself with one or more compatibility classes. Concrete services are analyzed and their similarity to the abstract class is evaluated. The evaluation of a WS is based on the degree of affinity of names for input and output parameters, and for operations in the service. The affinity is computed using the terminological

Table 3.2: The overview of solutions used for dynamic service selection in CWSs.

The framework and authors	**The basis for the discovery of a set of candidate WS**	**The basis for the selection of a WS to execute**	**The definition of a CWS (used model/notation)**
Self-Serv by Benatallah et al. [6, 7]	Service containers with membership modes: one is a query mode to query UDDI registry.	Multiattribute utility function (weighted sum of parameters).	State-charts executed in Self-Serv.
Framework by Baresi et al. [4]	Discovery preferences specified for each WS call.	Selection preferences (boolean expressions) based on properties (like cost, time).	BPEL enhanced with rules to dynamically discover, select and bind a WS.
Framework by De Antonellis et al. [13]	Membership to a compatibility class.	The terminological relationship for operations and parameters of a service with an appropriate abstract service.	A Petri Net based model.
Meteor-S by Verma et al. [39]	Functional and non-functional semantic description.	Satisfying the quantitative and qualitative process constraints.	BPEL specification of an abstract process.

relationship between them. In this work to support the usage of a wider range of concrete services, a mapping between parameters is defined, and in some cases also wrappers (if the number or types of parameters are different). De Antonellis design orchestrated composition with the Petri Nets formalism based on the work of Mecella et al. [27].

The semantic definition of a WS, enhanced with non-functional properties, is used to improve discovery of services in the METEOR-S framework [39]. Verma et al. use a BPEL abstract process description, and concrete services are bound at runtime. In order to find the best services they use constraints on properties for services required by a CWS, and reason which services do not violate them. In the METEOR-S a mechanism is implemented, which is based on ontologies, to translate input and output parameters between different services. In case of a failure of an external WS, the service can be replaced, but it means that all dependent services might be also replaced (if they violate constraints).

Table 3.2 presents the overview of the above propositions. The choice of the discovery method influences the actual selection of a WS to execute. In the above works the selection and discovery criteria are specified at the development time by the user.

3.3.2 Process definition

The other way to improve the reliability or performance of CWSs is to make a process definition adaptive. Some of the propositions in this area are presented below.

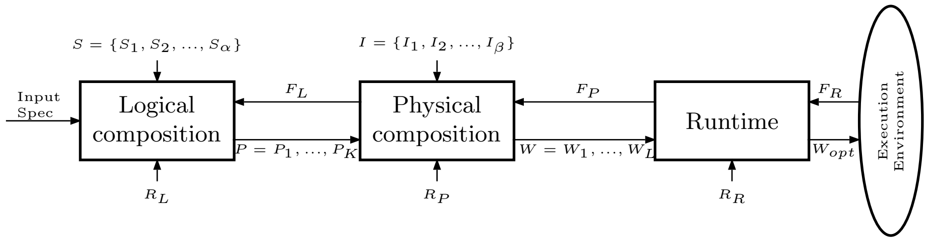


Figure 3.5: Overview of the A-WSCE framework.

Dahlem et al. [12], in order to improve the availability of CWSs, propose to implement the mediation layer. This layer is responsible for the selection of the best CWS according to their current state. In their work they provide a proxy, and then at runtime the composition that provides the required QoS for used services is chosen. Although this solution is used to select an appropriate alternative of composition that is based on evaluation of external WSs, it provides different versions of the process definition in BPEL. The alternatives are defined at the development time and they differ in the set of used services and their providers.

A more comprehensive proposition to the dynamic choice of a process is presented by Chafle et al. [9]. In this work there are two stages of composition, which refer to functional and non-functional requirements of the desired CWS. The abstract overview of this solution is shown in Figure 3.5. First there is the logical composition stage. In this stage there are chosen types of WSs from the set S and for them with given specification K different workflow templates (abstract workflows) are constructed. In the second stage, physical composition, L executable workflows are implemented, which use services instances from the set I. At runtime the framework selects the optimal executable workflow from the W set. This choice is supported by three rank functions: R_L - assigns a score to each abstract workflow, R_P - scores executable workflows using QoS values for them and R_R - supports the choice of W_{OPT} based on the current QoS values for services instances. In order to maintain only the set of optimal workflows there are feedback functions: F_R - carries measured QoS values for used services, F_P - for aggregated QoS values for a period of time, and F_L - for information about service types. This solution is further extended by an additional information Value of Change [10], which analyzes whether the potential gain in QoS parameters is worth measuring. In that way it is possible to avoid reconfiguring workflows.

Erradi et al. [15] present middleware for self-adaptation of CWSs, which is based on policies. A policy is a condition or an event that can trigger predefined actions, and it is specified with the extension of the WS-Policy [42] standard. As envisioned by its authors, this approach should be used to deal with special cases in a CWS execution. Manageable and Adaptive Service Composition (MASC) framework supports the static and dynamic customization of a process instance. The static customization is performed before an execution of the CWS and is response for runtime environment events. The dynamic customization, during the CWS execution, is started when the monitoring system raises an event. Erradi et al. also propose, complementary to the MASC, dynamic corrections of CWSs, which are used to prevent failures of CWS.

In the eFlow system proposed by Casati et al. [8] adaptation requirements are also defined by a user. In this proposition there is a clear distinction between the service definition and the service selection. So a process may use services, which are discovered and bound according to provided rules during an execution. There are also special types of services, called multiservices and generic nodes, which may actually be bound to many different physical services. In order to support services, with different input and output parameters, appropriate mapping between process variables and service variables are defined. In the eFlow it is possible to alter the definition of the flow with two mechanisms: ad-hoc and bulk changes. The first one allows an authorized user to change the currently executed processes by adding and deleting activities. The latter mechanism is used to alter many processes, for which the specified condition is true.

Yu et al. [46] give a solution for finding WSs in case of unavailability of one or more of them. The solution is based on service classes that contain individual services with multiple QoS levels. First there is an abstract definition of a business process, which specifies the flow of service classes. From this specification a graph with QoS constraints is constructed - nodes represent levels of QoS for all required services. During the execution of a CWS, which is the optimal path in the graph, some nodes may become unavailable. For this situation authors present two algorithms for finding the shortest path: to find a backup path and to find a replacement path (to reconfigure the process). So the CWSs may adapt if used WSs become unavailable.

Table 3.3 presents the overview of the described frameworks. Again most adaptations must be configured at the time of development, and they are used to mostly to cope with failures of external WS.

3.4 Summary

The CPN language offers unambiguous semantics and at the same time a graphical representation. The language is based on sound mathematical foundations and can be analyzed using different methods, for example state space analysis. In the context of modeling CWSs, the CPN is useful because it enables:

- efficient data representation (with types of places and varibles),
- verification of properties of a CWS like absence of deadlocks or different paths of execution.

The above advantages of the CPN (and of Petri Nets in general) are already explored in the area of WSs and their compositions. As shown [21, 30, 45] there are solutions to translate a BPEL

Table 3.3: The overview of solutions in adaptations of a process definition.

The framework and authors	**The definition of a CWS (used model)**	**The goal of adaptations**	**Possible changes in a process**	**Causes/reasons for adaptations**
Framework by Dahlem et al. [12]	BPEL processes.	Improving the availability of a CWS.	Different service providers in versions of a process.	Not available external WS
Adaptive WSComposition and Execution by Chafle et al. [9]	State charts	Optimizing QoS of a workflow.	Different service types and service instances.	QoS changes of external WS and of workflow and instances failures.
MASC (Manageable and Adaptive Service Composition) by Erradi et al. [15]	XAML - Extensible Applications Markup Language.	Managing special cases of a process by introducing policies.	Removal, addition, replacement of an activity in a process.	Events defined in a policy.
eFlow by Casati et al. [8]	Graphs, which define order and execution of nodes.	Dealing with dynamic business environment.	Add or remove services or process nodes.	Changing providers or requirements of a process.
Framework by Yu et al. [46]	Graph-like abstract process definitions.	Improvements in the reliability of a CWS.	Replace a WS or change a path of execution.	Failure or not available external WS.

process definition into Petri Nets, and how to verify the correctness of a CWS. Moreover there are also implementations to make such translations automatic, and tools to verify the required properties. The other [27, 47, 49, 35] research focuses more on the modeling aspects of Petri Nets. If a model of a CWS exists, there are considered features like the expected reliability or response time of the CWS.

In the area of adapting CWSs to the dynamic environment the problem of selecting the optimal external WS is well researched [7, 4, 13, 39]. The approaches to tackle it vary in the way services are discovered and selected. Some of them use only key-based search (with UDDI or other registries), whereas some enhance a search with the semantic description of WSs. Either way it is important to take into account differences in the interface of WSs, and how parameters are mapped between the different instances of WSs.

The other solutions [12, 9, 10, 15, 8, 46] propose adaptations in the definition of a CWS. It is done at two levels: the goal of the first level is to improve availability or performance of a CWS, and at the second level to respond to changes in business requirements. In the first level CWSs become autonomic (as for the principles for autonomic computing [31]), whereas in the second level changes are managed manually by defining rules or policies.

The goal of this research is to make CWSs reactive to changes in the environment. However it is achieved with a model of a CWS, so the problem considered here is how during execuctions a CWS can use possibilities present in its definition.

Chapter 4
Modeling, analyzing, executing and planning Composite Web Services with Coloured Petri Nets

The Coloured Petri Nets (CPN) language is used to model, analyze, execute and plan the execution of Composite Web Services (CWSs). In this chapter the theoretical aspects of these issues are presented. First the approach to model CWSs is described, and then how the model can be analyzed. Finally it is shown how CWSs are executed with the CPN, and how the model can be used to plan the execution.

4.1 Modeling Composite Web Services

To use the CPN language to model CWSs the operations specific to them must be defined with the CPN semantics. First the approach to model interactions with external WSs is presented, and then it is proposed how to represent other operations.

4.1.1 Interactions with external Web Services

During its execution a CWS interacts with external WSs, or more precisely, according to WSDL [41], with their operations. In general it means that the CWS sends input data and gets in turn the required information. Thus we can represent such an interaction as a function $result = externalWS(query)$. In the CPN this function is modeled as a transition like in the example in Figure 4.1. There are variables of type Integer as an input and output, and they model the query and the result of a call to a WS respectively. However this approach is not enough, because it omits modeling:

- values of parameters in input and output messages for a WS(as defined in its WSDL description [41]),
- different types of output from an interaction (like faults in a WSDL description [41] or failures),

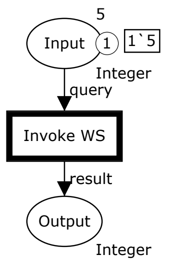
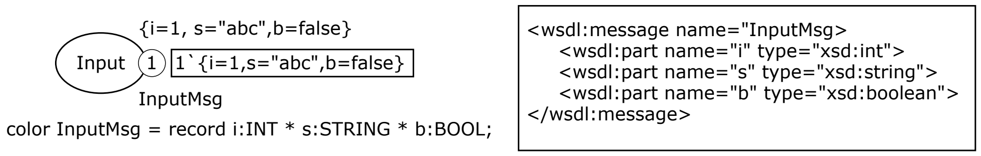


Figure 4.1: The example of an interaction with an external WS as a function.



```
color InputMsg = record i:INT * s:STRING * b:BOOL;
```

```
<wsdl:message name="InputMsg>
    <wsdl:part name="i" type="xsd:int">
    <wsdl:part name="s" type="xsd:string">
    <wsdl:part name="b" type="xsd:boolean">
</wsdl:message>
```

Figure 4.2: A color set in the CPN and an appropriate WSDL description of a message.

- different modes of interactions with an external WS (synchronous and asynchronous).

The above aspects must be taken into account to model interactions with external WSs.

One of the features of the CPN is the possibility to model hierarchies of nets [23]. In the context of modeling CWSs, the hierarchies allow defining details of interactions with other WSs in a separate net. So each call to an external WS is represented with a separate net. A net is then included in the other by substituting an appropriate transition and its surrounding arcs. So a CWS is modeled with a set of nets, called pages [23], whereas in each page there is a non-hierarchical CPN. Hence for each CWS there is a main page with a general model of a CWS and additional pages for each call to an external WS.

In a WSDL description of an operation in a WS there are input and output messages. They consist of parts, which represent parameters with types. To model these parameters in the CPN, a record type as a color set is used. It enables mapping of names and values, as defined in a WSDL description of input and output messages. In this way each part of an input or output message is represented with an appropriate field in a record. An example of how a WSDL description of a message is mapped to a color set is shown in Figure 4.2. The figure also presents an example of an initial value (a token with a color) for the declared color set. Color sets of the records type can also use other record types as fields, so there is the possibility to model more complex and compound types of messages. The mapping between types in XML Schema [43] and basic color sets, like integers, is straightforward, so it is omitted here.

An interaction with an external WS returns a response message: either a message with a result or a fault message (defining fault messages for operations is optional) [41]. Moreover, because a CWS interacts with remotely deployed components, it is possible that there is no response at

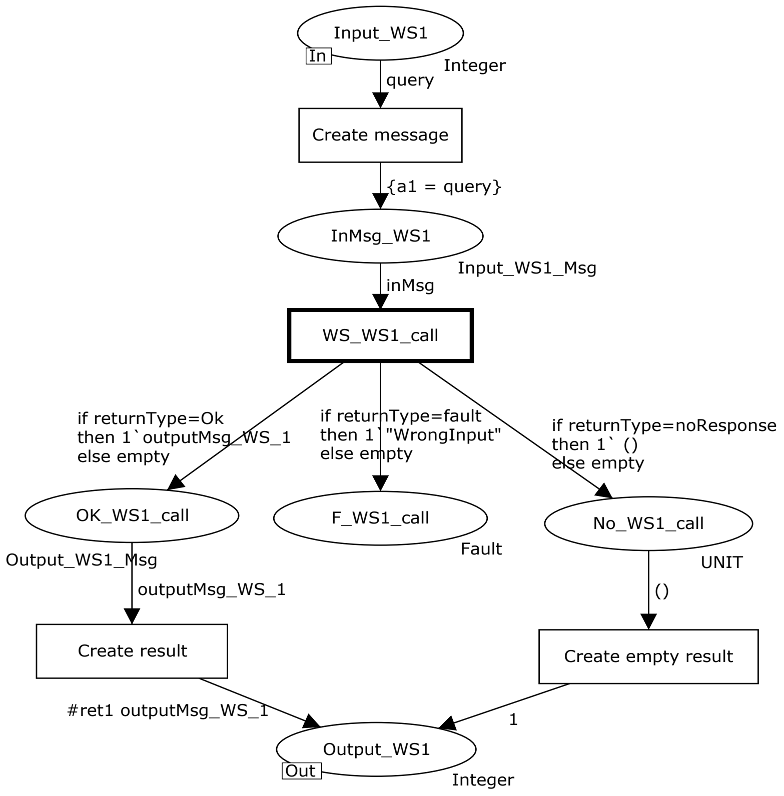


Figure 4.3: An example of a net, which specifies details of an interaction with an external WS.

all (omission failures [11]). All these possibilities must be taken into account when modeling an interaction. So there are 3 possible types of output from an external WS:

- a response message with a result as specified in a WSDL description,
- an optional fault message also specified in the description,
- no response or a fault message other than specified - it indicates a failure of a WS.

Each of these types of output is modeled with an output place from a transition that represents a call. An output transition from a place that represents a correct response, models retrieving a value of this response. It is also possible that there are output transitions from the other places, and they model routines that deal with faults or failures. If in a WSDL description of a WS there are no fault messages, then there are only 2 places: for correct and no response type of output.

Figure 4.3 presents the net that models an interaction with an external WS. Places with inscriptions ”In” and ”Out” refer to port nodes, and they are also in the main page of a CWS that invokes this WS. The CWS in the example uses an Integer value for the input and output, thus a conversion from and to the format of messages must be made. The input message has only 1 parameter that is modeled by a record type with only one field $a1$. The transition *Create message* creates new

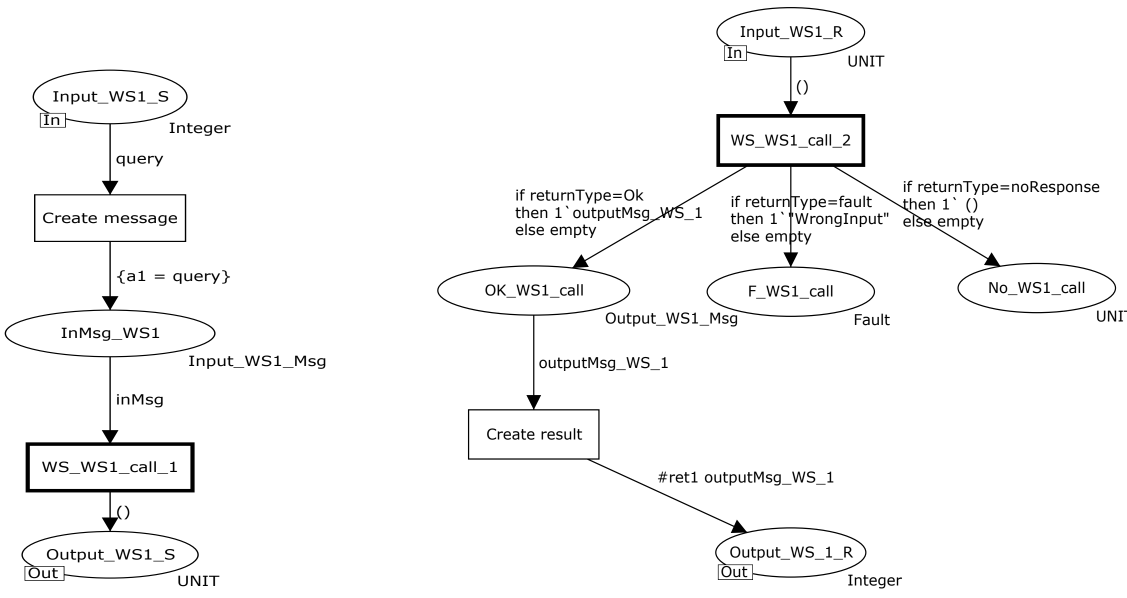


Figure 4.4: Examples of nets, which represent an asynchronous type of an interaction with WS. On the left there is a send operation, and on the right a receive operation.

variable of that record type, and assigns to it a value the *query* variable. Then a WS is invoked and three types of possible output are shown, with a variable *returnType* to represent them. After the call with, the output required by the CWS is created. This is either the contents of the return message (the *Create result*) transition), or value 1 if there is no response (the *Create empty result* transition). If a fault from the description of this WS is a response, nothing is specified afterward, which means that there is no routine to deal with it.

The interaction presented in Figure 4.3 assumes that the call is synchronous. However, it is also possible to invoke a WS in an asynchronous mode: first send a message and then request a response. Moreover it is also possible that only a message is sent, without requesting a response, thus an external WS is only notified. To model this kind of interaction there two additional operations defined: send and receive. The first one is to create a message and send it to a WS, the second is to get a response and transform its contents to a format required by a CWS. In the latter operation all possible types of output (correct response, declared fault message if any, no response) must be considered. An example of two nets that represent an asynchronous call are shown in Figure 4.4. They are similar to the interaction from Figure 4.3, however there is no routine to deal with the no response type of output.

The approach and examples presented in this section show how to model interactions with external WSs using the CPN. There are 3 types of such interactions: invoke, send and receive. In the CPN they are modeled as transitions, thus there are 3 subsets of all transitions: $T_{invokeWS}$, T_{sendWS} and $T_{receiveWS}$. They are defined as follows.

Definition 4.1.1. A transition t represents an invoke operation if:

$$\begin{aligned} t \in T_{invokeWS} \text{ iff } \quad & (t \in T) \ \wedge \ (size(In(t)) = 1) \ \wedge \ (size(Out(t)) >= 2) \ \wedge \\ & (\exists p \in In(t) \, : \, C(p) \approx inMsg) \ \wedge \ (\exists p1 \in Out(t) \, : \, C(p1) \approx outMsg) \ \wedge \\ & (\exists p2 \in Out(t) \, : \, C(p2) = UNIT) \end{aligned}$$

where:

- T is a set of all transitions in a net,
- In and Out are functions that map a node to its input and output nodes, respectively,
- $size$ is a size of a set,
- C maps a place into its color set,
- $\approx$ maps WSDL messages into record types,
- $inMsg$ and $outMsg$ represent accordingly all input and all output messages defined in a WSDL description of a WS.

Definition 4.1.1 says that a transition, which models an invoke operation, has one input place with the color set that is mapped from a WSDL input message. It also has at least two output places: one with the color set that is mapped from a WSDL output message and one with the unit color set (it represents ”no response” type of output). The size of the set of output places can be bigger than 2, because there can be fault messages in the WSDL description, each of which is modeled as an output place.

Definition 4.1.2. A transition t represents a send operation if:

$$\begin{aligned} t \in T_{sendWS} \text{ iff } \quad & (t \in T) \ \wedge \ (size(In(t)) = 1) \ \wedge \ (size(Out(t)) = 1) \ \wedge \\ & (\exists p \in In(t) \, : \, C(p) \approx inMsg) \ \wedge \ (\exists p1 \in Out(t) \, : \, C(p1) = UNIT) \end{aligned}$$

where the notation is as in Definition 4.1.1.

Definition 4.1.2 differs from Definition 4.1.1 in the output type, since for the send operation it is only the unit color set.

Definition 4.1.3.

$$\begin{aligned} t \in T_{receiveWS} \text{ iff } \quad & (t \in T) \ \wedge \ (size(In(t)) = 1) \ \wedge \ (size(Out(t)) >= 2) \ \wedge \\ & (\exists p \in In(t) \, : \, C(p) = UNIT) \ \wedge \ (\exists p1 \in Out(t) \, : \, C(p1) \approx outMsg) \ \wedge \\ & (\exists p2 \in Out(t) \, : \, C(p2) = UNIT) \end{aligned}$$

where the notation is as in Definition 4.1.1.

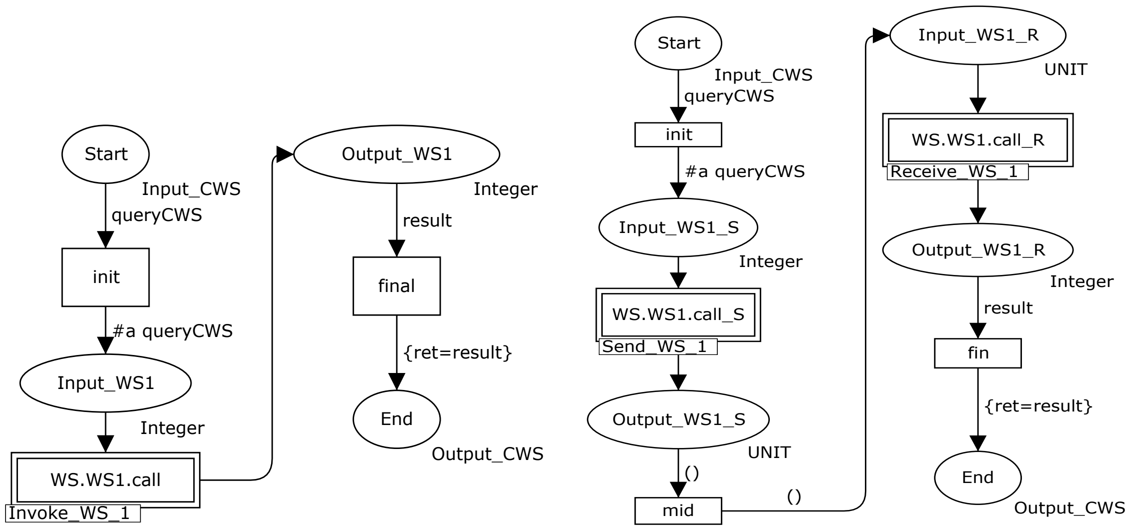


Figure 4.5: Examples of main pages of CPN. On the left there is modeled a CWS with invoke operation and on the right there is a CWS with send and receive operations.

The difference between Definition 4.1.3 and Definition 4.1.1 is that in an input type of the receive there is the unit color set. The set of all interactions for CWS is:

$$T_{WS} = T_{invokeWS} \cup T_{sendWS} \cup T_{receiveWS}$$

4.1.2 Other operations

Details of interactions with external WSs are modeled on separate pages, and they are then a part of a more abstract specification of a CWS, which is modeled on a main page. Two examples of such pages are presented in Figure 4.5. They are main pages of CWSs which model only interactions presented in Figure 4.3 and Figure 4.4, and they call external WSs in synchronous and asynchronous mode, accordingly. The transitions that model interactions with external WSs (with inscriptions *Invoke_WS_1*, *Send_WS_1* and *Receive_WS_1*) are substituted with the previously shown nets. The hierarchy of pages makes it possible to model a CWS in the abstract way on the main page and to deal with details on additional nets. In turn the general model of a CWS, which is on the main page, operates only on variables and colors required by it without transformations to messages types (for the net from Figure 4.5 it is color set Integer and variables *query* and *result*).

Interactions with external WSs are not the only operations that a CWS can contain. There are operations on data, various control operations (for example if statements, while loops) or others. All of these can be represented in the CPN. The examples of different possible structures can be found in workflow patterns [37]. It is also possible that a CWS and the CPN model represent complex workflows, the details of this approach can be found in [38]

To make an analysis easier the following requirements for nets, which represent CWSs, are

Table 4.1: Size of occurrence graphs according to the size of the color set for an output from an external WS.

	CWS 1 : Synchronous call			CWS 2 : Asynchronous call		
	size = 5	size = 10	size = 100	size = 5	size = 10	size = 100
Number of nodes	20	35	305	22	37	307
Number of arcs	28	53	503	29	54	504

specified:

- the place that represents the beginning of a CWS should not contain input arcs, and the color of this place determines the type of input parameters to this CWS,
- the place that represents the end of a CWS should not contain output arcs, and the color of this place determines the type of output from this CWS,
- transitions that represent the interactions with external WSs have 1 input and 1 output arc on the main page.

4.2 Analyzing Composite Web Services

State space analysis is a powerful tool, which enables proving unambiguously the properties of a net. However, occurrence graphs may become very big. Table 4.1 presents an example of this phenomenon. There are numbers of nodes and arcs for occurrence graphs constructed for the nets presented in the previous section (the CWS 1 is in Figures 4.3 and the left part of 4.5, the CWS 2 is in Figures 4.4 and the right part of 4.5). For both nets the initial marking is a token $1`\{a = 4\}$ in the place *Start*. The data demonstrates that the occurrence graph becomes big, even though the represented CWS are straightforward.

In the CPN that models a CWS the size of an occurrence graph is strongly dependent on the number of possible results, which external WSs may return. So if the number of results is higher, then the occurrence graph has more nodes and arcs, since all values must be represented. This relation is also presented in Table 4.1, the bigger the size of a color set for the output variable, the bigger the graph. Additionally it can be observed in Figure 4.6 with the full occurrence graph for the previous example of CWS 1 (in Figures 4.3 and the left part of 4.5). Labels of the nodes contain the number of a marking and a name of a non-empty place with tokens. For example the node number 7 represents marking, in which only Ok_WS1 place has one token with the value $\{ret1 = 4\}$. The first 3 nodes in the graph represent generating a message from input parameters. Then according to the binding of $returnValue$ there are: the node 5 if a fault is returned, the node 4 if there is no response or the nodes 6 to 10 if there is a correct return value from the $WS1$. It is

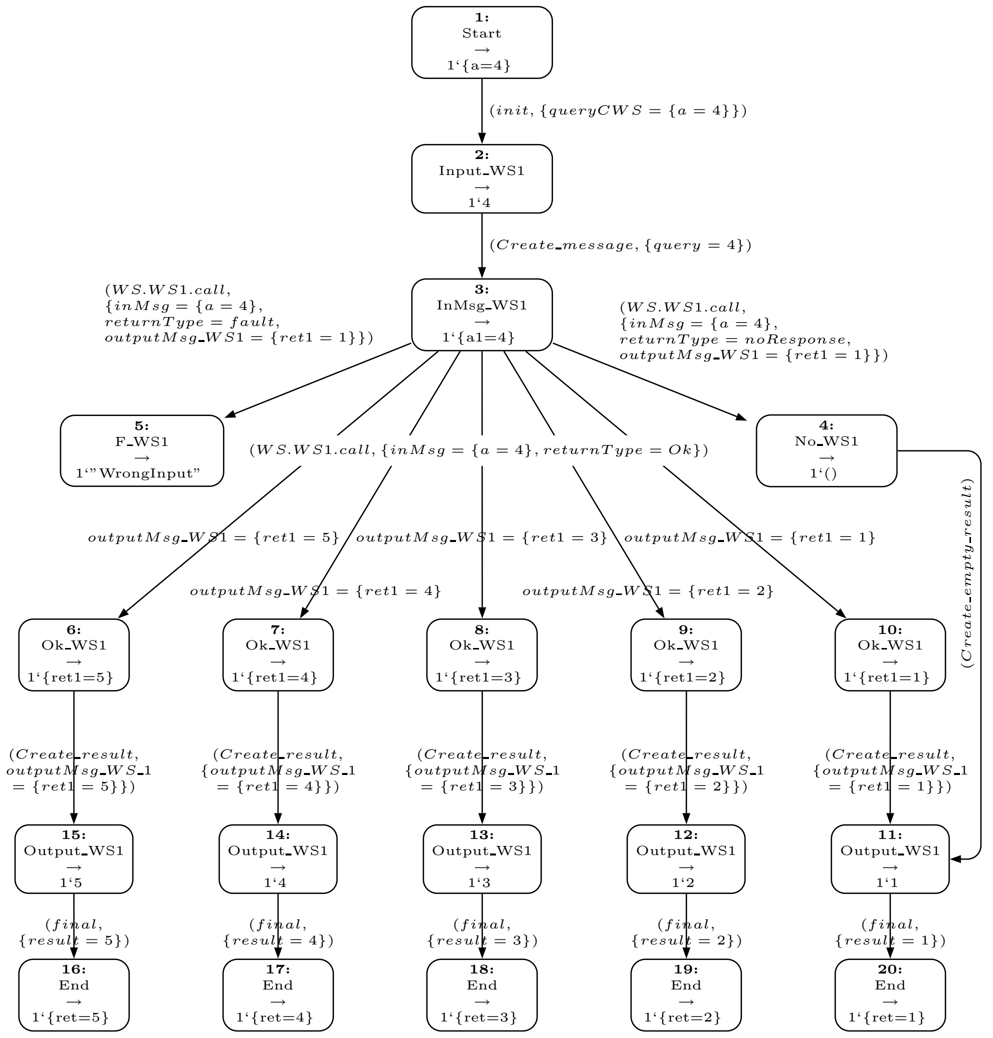

1: Start → 1‘{a=4}
(init, {queryCWS = {a = 4}})
2: Input_WS1 → 1‘4
(Create_message, {query = 4})
3: InMsg_WS1 → 1‘{a1=4}
(WS.WS1.call, {inMsg = {a = 4}, returnType = fault, outputMsg_WS1 = {ret1 = 1}})
(WS.WS1.call, {inMsg = {a = 4}, returnType = noResponse, outputMsg_WS1 = {ret1 = 1}})
5: F_WS1 → 1‘"WrongInput"
4: No_WS1 → 1‘()
(WS.WS1.call, {inMsg = {a = 4}, returnType = Ok})
outputMsg_WS1 = {ret1 = 5}
outputMsg_WS1 = {ret1 = 3}
outputMsg_WS1 = {ret1 = 1}
outputMsg_WS1 = {ret1 = 4}
outputMsg_WS1 = {ret1 = 2}
(Create_empty_result)
6: Ok_WS1 → 1‘{ret1=5}
7: Ok_WS1 → 1‘{ret1=4}
8: Ok_WS1 → 1‘{ret1=3}
9: Ok_WS1 → 1‘{ret1=2}
10: Ok_WS1 → 1‘{ret1=1}
(Create_result, outputMsg_WS_1 = {ret1 = 5}})
(Create_result, {outputMsg_WS_1 = {ret1 = 4}})
(Create_result, {outputMsg_WS_1 = {ret1 = 3}})
(Create_result, {outputMsg_WS_1 = {ret1 = 2}})
(Create_result, {outputMsg_WS_1 = {ret1 = 1}})
15: Output_WS1 → 1‘5
14: Output_WS1 → 1‘4
13: Output_WS1 → 1‘3
12: Output_WS1 → 1‘2
11: Output_WS1 → 1‘1
(final, {result = 5})
(final, {result = 4})
(final, {result = 3})
(final, {result = 2})
(final, {result = 1})
16: End → 1‘{ret=5}
17: End → 1‘{ret=4}
18: End → 1‘{ret=3}
19: End → 1‘{ret=2}
20: End → 1‘{ret=1}

Figure 4.6: The full occurrence graph for the CWS 1 with the size=5 of the color set for an output of the external WS.

important to note that nodes of this last type, which represent all possible correct results and their successors, make the graph big. Moreover the differences between those paths are only in values of tokens and bindings for variables.

The above conclusion is the basis for the method used to reduce the number of nodes and arcs in occurrence graphs for CWSs. So if correct results from a call to an external WS do not affect further states, then only one path is explored and stored. More formally we can define an equivalence specification for markings (nodes in a graph) and bindings elements (arcs in a graph) as in Definition 4.2.1.

Definition 4.2.1. For the CPN model of CWS, given two markings $M_1, M_2 \in \mathbb{M}$ and an equivalence relation $\approx_{WS}$ on the results from WS we have:

$$M_1 \approx_M M_2 \Rightarrow \forall_{p\in P} : M_1(p) = M_2(p) \vee (|M_1(p)| = |M_2(p)| = 1 \wedge M_1(p) \approx_{WS} M_2(p))$$

Similarly, given two bindings elements $BE_1, BE_2 \in \mathbb{BE}$ and an equivalence relation $\approx_{WS}$ on the results from WS we have:

$$\begin{aligned} BE_1 \approx_M BE_2 \quad \Rightarrow \quad & \text{tran}(BE_1) = \text{tran}(BE_2) \ \wedge \\ & \forall_{v_1\in \text{var}(BE_1)} \exists_{v_2 \in \text{var}(BE_2)} : \text{Type}(v_1) = \text{Type}(v_2) \wedge v_1 \approx_{WS} v_2 \end{aligned}$$

According to Definition 4.2.1 two markings are equal if they differ only for places with one token and the difference is specified by the equivalence for results from WS. Two binding elements are equal if they differ for the same transition and for variables with the same types and the difference is specified by the mentioned equivalence.

In the graph in Figure 4.6 we can define one class of the results from the call to $WS1$. It means that all tokens $\{\{ret1 = 1\}, \{ret1 = 2\}, \{ret1 = 3\}, \{ret1 = 4\}, \{ret1 = 5\}\}$ are equal, as well as $\{1, 2, 3, 4, 5\}$ and $\{\{ret = 1\}, \{ret = 2\}, \{ret = 3\}, \{ret = 4\}, \{ret = 5\}\}$. Also all bindings for variables *outputMsg_WS_1* and *result* are the same. The occurrence graph with equivalence classes (also called the OE-graph) for the CWS 1, is presented in Figure 4.7. The graph has only 8 nodes and 8 arcs, and these numbers do not change even if the size the color set for an output from external WS increases. The biggest advantage is that only one path for results from a call to external WS is explored, hence this graph requires less space to store and less time to construct.

The definition of an equivalence relation $\approx_{WS}$ depends on an interaction with an external WS, so it is specific for each WS and for each CWS. In this work it is assumed that this relation is given, although it can be inferred automatically. In this case the whole graph must be constructed and then after each call to external WS symmetries are extracted. This solution requires the exploration of the whole graph, thus it requires more resources.

4.3 Executing Composite Web Services

To observe the dynamic behavior of the net it can be executed. During an execution enabled binding elements are monitored and transitions are fired, which result in consecutive markings. Each execution of a net is also a path in an occurrence graph. However, in order to execute a CWS in the CPN, the activities specific for CWSs must be determined. These specific activities are: responding to requests, interacting with external WS and producing an output from a CWS.

Since a CWS is also a WS, it responds to requests that contain input parameters embedded in an XML message. The types of parameters of that message are mapped in the same way as it is for interactions with external WSs. For each CWS the type of its *Start* place determines, what type

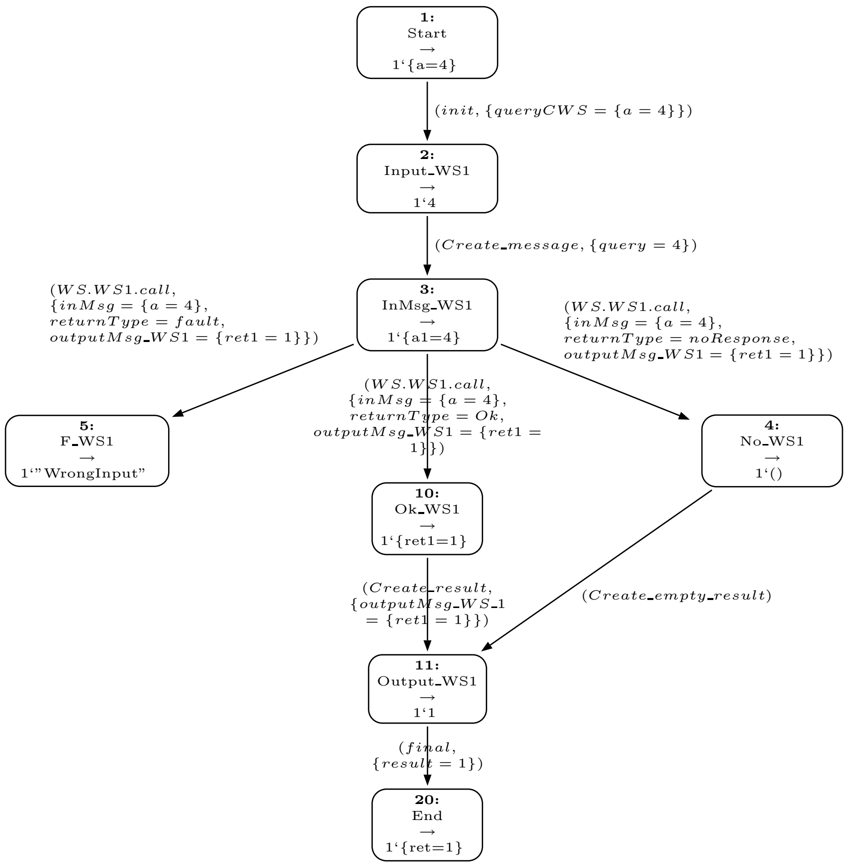

1: Start → 1'{a=4}
(init, {queryCWS = {a = 4}})
2: Input_WS1 → 1'4
(Create_message, {query = 4})
3: InMsg_WS1 → 1'{a1=4}
(WS.WS1.call, {inMsg = {a = 4}, returnType = fault, outputMsg_WS1 = {ret1 = 1}})
(WS.WS1.call, {inMsg = {a = 4}, returnType = noResponse, outputMsg_WS1 = {ret1 = 1}})
(WS.WS1.call, {inMsg = {a = 4}, returnType = Ok, outputMsg_WS1 = {ret1 = 1}})
5: F_WS1 → 1'"WrongInput"
4: No_WS1 → 1'()
10: Ok_WS1 → 1'{ret1=1}
(Create_result, {outputMsg_WS_1 = {ret1 = 1}})
(Create_empty_result)
11: Output_WS1 → 1'1
(final, {result = 1})
20: End → 1'{ret=1}

Figure 4.7: The OE-graph for the CWS 1.

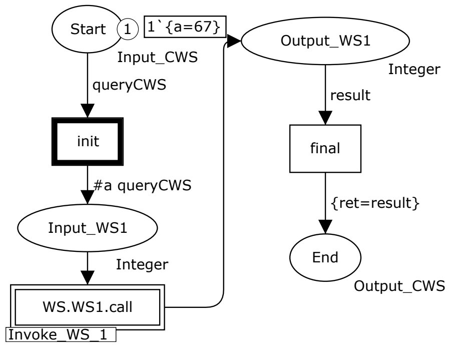
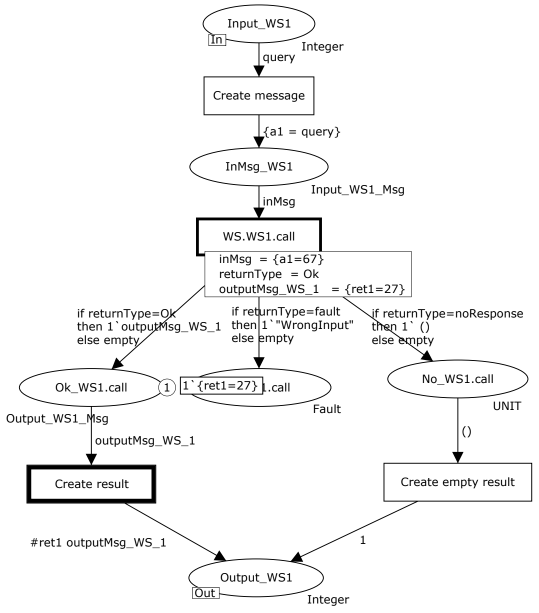


Figure 4.8: The example of receiving a request to the CWS1.

of input parameters it may accept. The presence of a token in the *Start* place means that there is a request to this CWS, and the value of the token specifies value of parameters of the request. For example in Figure 4.8 there is CWS 1 (from Figures 4.3 and the left part of 4.5) with a token 1'$\{a = 67\}$ in the *Start* place. It means that there is a request for CWS 1, with a value 67 for its only parameter a, which is of type Integer.

The next activity specific for CWSs is a call to an external WS. As it was presented previously, a



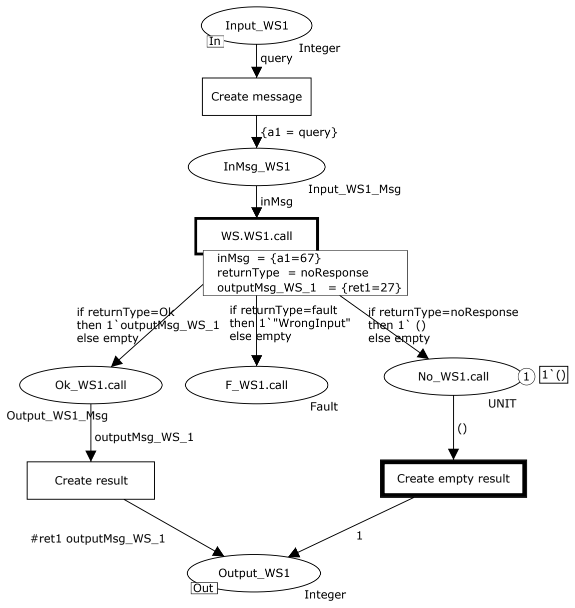


Figure 4.9: The example of 2 results of an interaction with an external WS: on the left for the binding $< resultType = Ok, outputMsg\_WS\_1 = \{ret1 = 27\} >$, on the right for $< resultType = noResponse, outputMsg\_WS\_1 = \{ret1 = 27\} >$.

call to an external WS is modeled on a separate page and consists of three phases: transformation to a color that models input parameters, an actual call and transformation from output parameters. The result from an interaction in the CPN model is represented by two values: a return type and an output message. In the model these values are determined manually. For example in Figure 4.9 there are 2 results from the interaction with the external WS for binding $< resultType = Ok, outputMsg\_WS\_1 = \{ret1 = 27\} >$ and $< resultType = noResponse, outputMsg\_WS\_1 = \{ret1 = 27\} >$. Because $outputMsg\_WS\_1$ is a free variable for this transition, it must be also bound for the second case, although it is not used. The detailed description of how WSs are called during an execution of the CPN, and how the above results are produced, is presented later.

Finally if all operations are finished, a CWS should return a result. In the context of the CPN the end of execution can be either successful or not. The successful execution is equal to a marking with only one token and this token is in the *End* place; the unsuccessful execution is equal to a marking from which the successful one is not reachable. The value of the token in the *End* place is also a value returned by the CWS, and the color of this place indicates the type of returned message (the same as for interactions with external WS). In Figure 4.10 the example of a successfully executed CWS with the result of type Integer and the value 27 is presented.

Each execution of a CWS is represented as a path in an occurrence graph, and is a finite sequence of markings. There are 3 types of parameters that might change between executions of the same CWS:

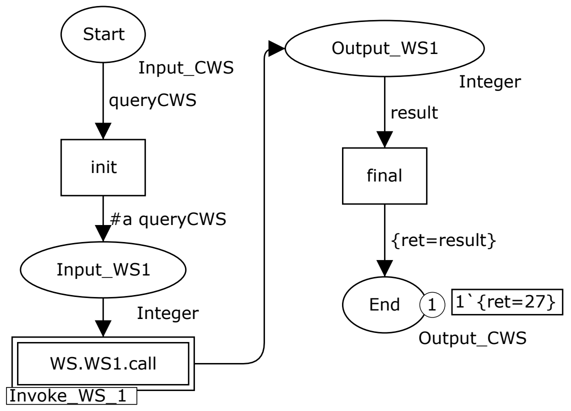


Figure 4.10: The example of a successfully finished CWS.

- the marking of a *Start* place,
- the output (type and value) from interactions with external WSs,
- other free variables that are not bound by arc expressions.

The above parameters determines the CWS execution.

4.4 Planning execution of Composite Web Services

Planning the execution of a CWS means selecting the path in an occurrence graph, in which the successful marking is reached in the shortest time. So in order to plan it is required that the current state of execution is known and time to reach other states is possible to infer. Moreover, it must be possible to determine, when interactions with external WSs are going to take place.

To plan executions of CWSs the concept of timed CPN is used [24]. In the net there is a global clock, which represents model time. Additionally each token can carry a time value, which indicates the earliest model time when this token can be used. So a binding element must be color enabled (as it is for a non-timed CPN) and also must be ready: the time value of a token to remove must be less or equal to model time. A transition may create a time stamp for its output tokens, which models that the transition needs to be performed the defined number of time units.

For the example from the previous section (from Figures 4.3 and the left part of 4.5) in Figure 4.11 there is the timed version of the main page. It represents the CWS with a request, which is currently processed. The request is modeled by a token in the $Input\_WS1$ place. The token besides its value carries also a time stamp that equals 0. Transitions have additional inscriptions which start with "@+", they model the time required to perform appropriate operations. For this net an occurrence graph can be constructed. As well as for the ordinary CPN, the equivalence classes can reduce the number of nodes and arcs. Figure 4.12 shows the first nodes in such a

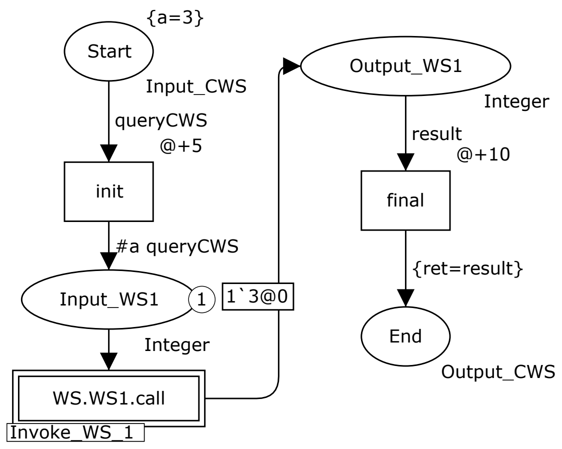


Figure 4.11: The timed version of the CWS 1 with a token that represents the currently executed request.

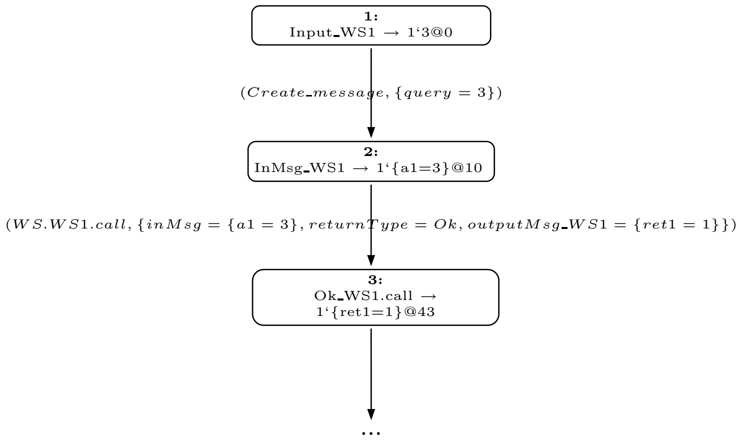

1:
Input_WS1 → 1`3@0
(Create_message, {query = 3})
2:
InMsg_WS1 → 1`{a1=3}@10
(WS.WS1.call, {inMsg = {a1 = 3}, returnType = Ok, outputMsg_WS1 = {ret1 = 1}})
3:
Ok_WS1.call →
1`{ret1=1}@43
...

Figure 4.12: The part of the OE-graph for the timed CPN model of CWS 1.

graph. First there is the marking already shown in Figure 4.11, in which a binding element with the *Create_message* transition is enabled. After firing this transition the token has a time stamp with a value 10, which is a delay introduced by the transition. So the model time must be set to 10, in order to use this token. Firing the next transition, a call to $WS1$, adds another delay to the token. This is repeated until there are no enabled transitions. The marking reached in this situation can represent a successful execution. If there are more than one such markings, then the one in which a token has the lowest value of time stamp is chosen. A path to it is the optimal execution and is a plan of an execution of this CWS. It is also important to note that a marking number 2 in the graph means that there is going to be a request to $WS1$ in 10 time units. So the future load on this external WS is anticipated.

4.5 Summary

Using the CPN language in the context of CWSs requires considering issues specific for compositions, like interacting with external WSs or representing input and output parameters for a CWS. It also requires the decision which level of abstraction is the most appropriate. Here there is presented a proposition, which models CWSs and their interactions on the level of the contents of exchanged messages. Even though the details of communication protocols (SOAP, HTTP) are not included, the proposed granularity is enough to model and execute CWSs; it also allows planning and predicting the future states of the CWS execution.

The following definitions summarize the approach presented in this chapter:

Definition 4.5.1. A CWS model (CWSM) is a tuple
$CWS = (CPN, OEG_1, ..., OEG_k, \approx_1, ..., \approx_n)$, where:

- CPN is the CPN, as defined previously, with an empty initial marking,
- $OEG_1, ..., OEG_k$ is a set of OE-graphs constructed for initial markings,
- $\approx_1, ..., \approx_n$ is a set of equivalence relations for the output from external WSs used in a CWS.

Definition 4.5.2. A CWS execution (CWSE) is a tuple
$CWS = (CPN, IN, OEG, Path)$, where:

- CPN is the CPN, as defined previously,
- IN is a set of input parameters (the initial marking),
- OEG is an OE-graph for an initial marking IN,

Definition 4.5.3. A CWS plan is (CWSPlan) is a tuple
$CWS = (CPN, DEL, OEG_{timed})$, where:

- CPN is the CPN, as defined previously,
- DEL is a delay function for transitions $DEL \in [T \to TIME]$,
- OEG_{timed} is an OE-graph constructed for specified delays.
- $Path$ is a list of binding elements, which should be executed.

Chapter 5

Architecture and implementation of model-aware middleware

In the previous chapter the use of the Coloured Petri Nets (CPN) in modeling and executing Composite Web Services (CWSs) was presented. This chapter shows how this approach is incorporated in the model-aware middleware. The architecture of the middleware is based on the layered architecture shown in Figure 2.3. Its more definite specification, by means of the components diagram, is given in Figure 5.1. Two registries (the External WS Registry and CWS Registry) are responsible for storing data provided by a user: either a WSDL description of an external WS or a CPN model of a CWS. Both registries are used by the CWS Execution Engine to execute a CWS. The execution is initialized and monitored by the CWS Agents. Additionally each execution of a CWS generates events, which are stored in a database (the Data component). The data gathered there is the basis for the Reasoning Mechanism to reason about the state of WSs, this knowledge is used by the CWS and System Agents to manage all executions of CWSs.

In the following sections first the use cases of the middleware are presented. Then all of the components are described in more detail.

5.1 Use cases for the middleware

The most important functionalities of the model-aware middleware are: constructing and storing models of CWSs and responding to requests to these services. Both of them require the cooperation

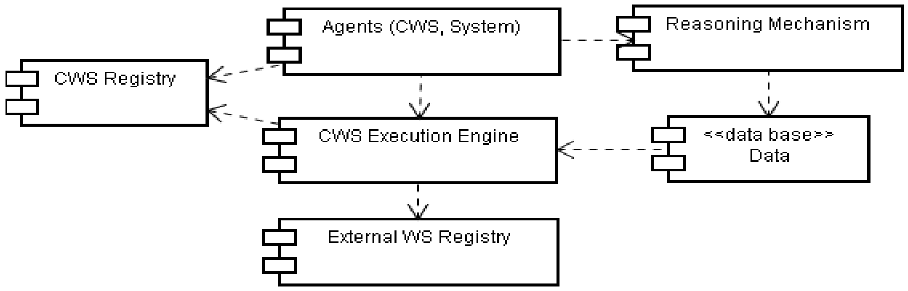


Figure 5.1: The components of the model-aware middleware for CWS.

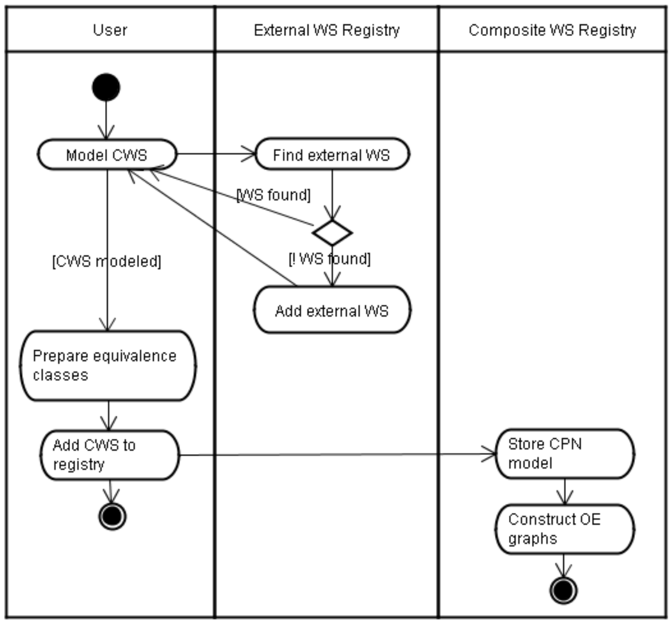


Figure 5.2: Activities in the process of managing models CWS.

of several components presented in Figure 5.1. The following sections describe that cooperation.

5.1.1 Managing CWSs

Managing CWSs consists of modeling them with the CPN, and then storing them, in a way that allows their executions. Figure 5.2 presents the most important activities in that process. Management of CWSs involves both registries: the External WS Registry and the CWS Registry. The first one exposes external WSs, so it is possible to use them to model interactions. The latter is used to store CWS models and to prepare OE graphs required during execution.

5.1.2 Executing CWSs

An execution of a CWS is a process triggered by a request to this CWS. First the path to execute a CWS is chosen, and then it is monitored during the execution. The whole process is shown in Figure 5.3. It starts with a request to the appropriate CWS Agent, which prepares a plan based on the current state of WSs, and checks predicted loads on external WSs with the Reasoning Mechanism. The plan is executed, and at the same time data are gathered and stored. If the execution differs from the plan, a new plan for the remaining operations is prepared. The Reasoning Mechanism analyzes data in the Database, and reasons about states of all external WSs. In case of any changes in those states, the CWS Agent prepares a new plan. After the execution is finished the response is returned to the requestor.

All components of the model-aware middleware are described in the following sections.

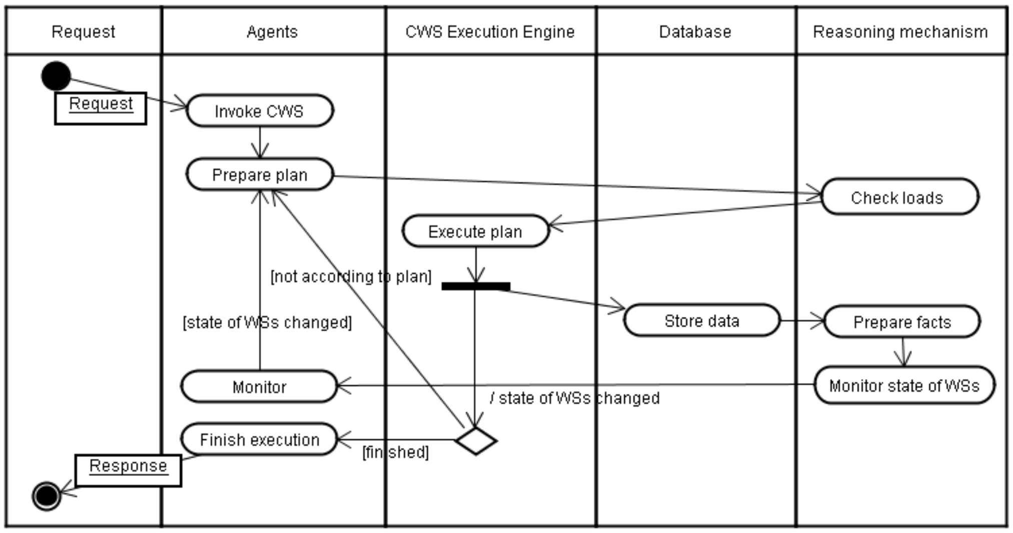


Figure 5.3: Activities in the process of executing CWS.

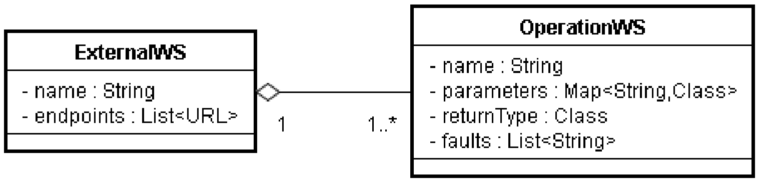


Figure 5.4: The class diagram for an external WS stored in the External WS registry.

5.2 External WS Registry

The component External WS Registry stores data about external WSs used during executions of CWSs. The main functionalities that this component realizes are:

- adding and removing WSs and their operations,
- adding and removing endpoints for WSs (instances of external WSs),
- synchronously and asynchronously calling operations of WSs.

To add an external WS to the registry a user provides a WSDL description of this WS, which is parsed to objects. Figure 5.4 presents the class diagram for a WS stored in the registry. Each WS has its unique name, a list of offered operations, as well as a list of endpoints. The endpoints represent instances of the external WS, and are used to call this external WS. The list of endpoints can be dynamically managed by adding or removing elements.

There are three activities offered by the External WS Registry to interact with WSs: invoke, send and receive. The first one is for a synchronous type of interaction and the last two are for

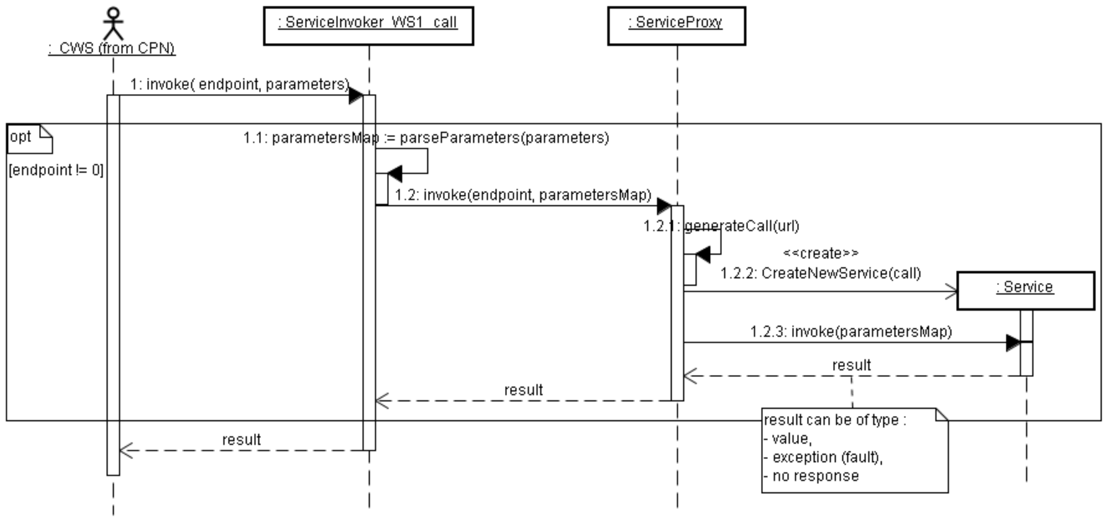

: CWS (from CPN)
: ServiceInvoker_WS1_call
: ServiceProxy
1: invoke(endpoint, parameters)
opt
[endpoint != 0]
1.1: parametersMap := parseParameters(parameters)
1.2: invoke(endpoint, parametersMap)
1.2.1: generateCall(url)
<<create>>
1.2.2: CreateNewService(call)
: Service
1.2.3: invoke(parametersMap)
result
result
result
result can be of type :
- value,
- exception (fault),
- no response

Figure 5.5: The sequence diagram for the invoke activity.

the asynchronous one. Their arguments and return values are gathered in Table 5.1. In those activities the name of an external WS and its operation are not arguments, because each operation is implemented as a separate class (they are generated when a WS is added to the registry). In Figure 5.5 there is a sequence diagram for the invoke activity (send and receive activities are similar so sequence diagrams for them are omitted here). A request comes from a CWS, which calls the invoke method on a ServiceInvoker object for the required operation, since it is the object visible during the execution of the CPN. ServiceInvoker uses the proxy, and calls its invoke method. The proxy creates the actual call to the WS, and returns the result to ServiceInvoker. If an endpoint equals 0, then this external WS is not called, and no response type of output is returned.

5.3 CWS Registry

The component CWS Registry is responsible for storing data about CWSs, and making them available for the CWS Execution Engine. To add a CWS to the registry a user must provide: a file with a CPN model and equivalence classes for each interaction with external WSs and for a CWS itself. In Figure 5.6 there is the class diagram for a CWS as stored in the registry. Each CWS is

Table 5.1: Arguments and return values of invoke, send and receive activities.

Operation	Arguments	Return value
invoke	endpoint, parameters	result of a call to WS
send	endpoint, parameters	nothing
receive	endpoint	result of a call to WS

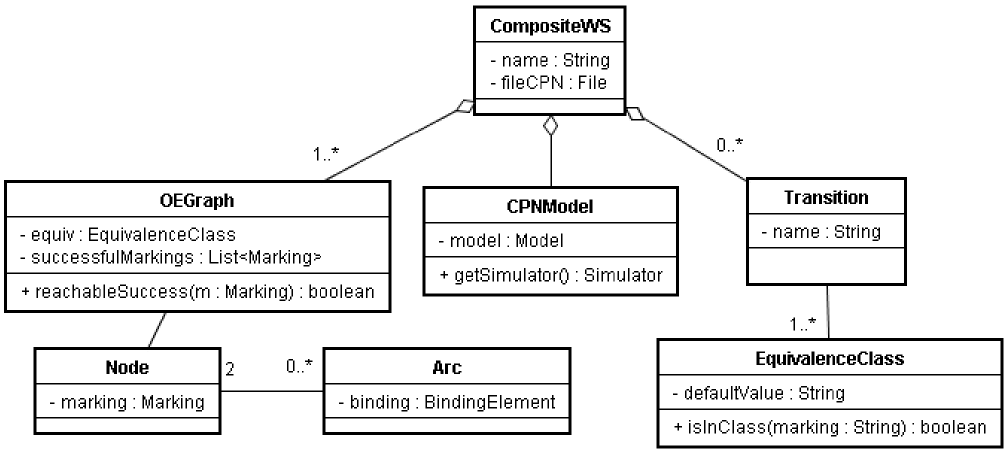

CompositeWS
- name : String
- fileCPN : File
1..*
0..*
OEGraph
- equiv : EquivalenceClass
- successfulMarkings : List<Marking>
+ reachableSuccess(m : Marking) : boolean
CPNModel
- model : Model
+ getSimulator() : Simulator
Transition
- name : String
1..*
Node
- marking : Marking
2
0..*
Arc
- binding : BindingElement
EquivalenceClass
- defaultValue : String
+ isInClass(marking : String) : boolean

Figure 5.6: The class diagram for a composite WS as stored in the CWS Registry.

identified by its name, and it has a reference to the file with its CPN model. After loading a model from a file, the CWS has a reference to it, which is required to create a simulation environment for the CWS. The CWS contains also a set of transitions for calls to external WSs, each with one or more equivalence classes. An equivalence class has a default value for an output variable from a call, and inclusion conditions. These equivalence classes are used to construct a set of OE-graphs for the CWS along with a successful marking for each class. Listing 5.1 shows the algorithm (in Java-like syntax):

Listing 5.1: The algorithm for constructing a set of OE-graphs for a CWS.

```
Map constructOEGraphs(CompositeWS cws){
  Map<EquivClass, OEGraph> graphMap;
  Set<EquivalenceClass> equivCWS = cws.getEquivClasses("Start");
  CPNSimulator simulator = cws.getCPNModel().getSimulator();
  for (EquivalenceClass equiv : equivCWS) {
    Place startPlace = cws.getCPNModel().getStartPlace();
    Marking initialMarking = prepareMarking(startPlace, equiv.getDefaultValue());
    simulator.setMarking(startPlace, initialMarking);
    OEGraph graph = createGraph(inititalMarking, simulator);
    graphMap.put(equiv, graph);
  }
  return graphMap;
}
OEGraph createGraph(Marking initMarking, Simulator simulator) {
  OEGraph graph = new OEGraph();
  graph.addNode(initMarking);
  Set nodeSet = new Set(initMarking);
  while (!nodeSet.isEmpty()){
      Marking currentMarking = nodeSet.next();
      simulator.setMarking(currentMarking);
      List enabledTransitions = simulator.getEnablings();
      for (Transition transition : enabledTransitions){
        if (transition.getType().equals(WSTransition)) {
          String outputVar = transition.getOutputVariable();
          for (EquivalenceClass eqClass : cws.getEquivalenceClasses(transition)) {
            for (Binding binding : simulator.getEnabledBindings(transition)) {
              binding.set(outputVar, eqClass.getDefaultValue());
              BindingElement bElement = new BindingElement(transition, binding);
              for (Place outputPlace : transition.getOutputPlaces(){
                Marking newMarking = new Marking(currentMarking);
                newMarking.set(transition.getInputPlace(),"empty");
                newMarking.set(outputPlace, prepareMarking(outputPlace, eqClass));
                graph.addNode(newMarking);
                graph.addArc(currentMarking, bElement, newMarking);
                nodeSet.add(newMarking);
              }
            }
          }
        } else {
          for (Binding binding : simulator.getEnabledBindings(transition)) {
            BindingElemenent bElement = new BindingElement(transition,binding)
            Marking newMarking = simulator.fire(bindingElement);
            graph.addNode(newMarking);
            graph.addArc(currentMarking,bindingElement,newMarking)
            nodeSet.add(newMarking);
          }
        }
      }
    }
  return graph;
}
```

An OE-graph is constructed for each equivalence class for a CWS. The default value for equivalence class is converted into the initial marking, which is the first node in the graph. Then for each new marking the list of enabled transitions is determined. If an enabled transition represents a call to an external WS, its equivalence classes are used, and a token is removed from its input place and added to all output places. If a transition represents other operation it is fired. Each generated marking is added to a list of nodes that are still waiting to be explored. If the list is empty the graph is constructed and added to a map of graphs for this CWS.

From the OE-graphs it is possible to infer which calls to external WSs are optional. This happens if for the marking that represents the no response type of output it is possible to reach the marking for a successful execution. This property is used during an execution of a CWS.

5.4 CWS Execution Engine

The component CWS Execution Engine is responsible for executing CWSs and for generating events stored by the Data component. It receives requests from a CWS Agent and it uses data from the CWS Registry. The conceptual phases of executing CWSs with the CPN were presented in Section 4.3; here they are presented from the implementation point of view. The execution of the CPN model is supported with BRITNeY Suite [36], as a plugin to the tool.

Parameters of each execution of a CWS are:

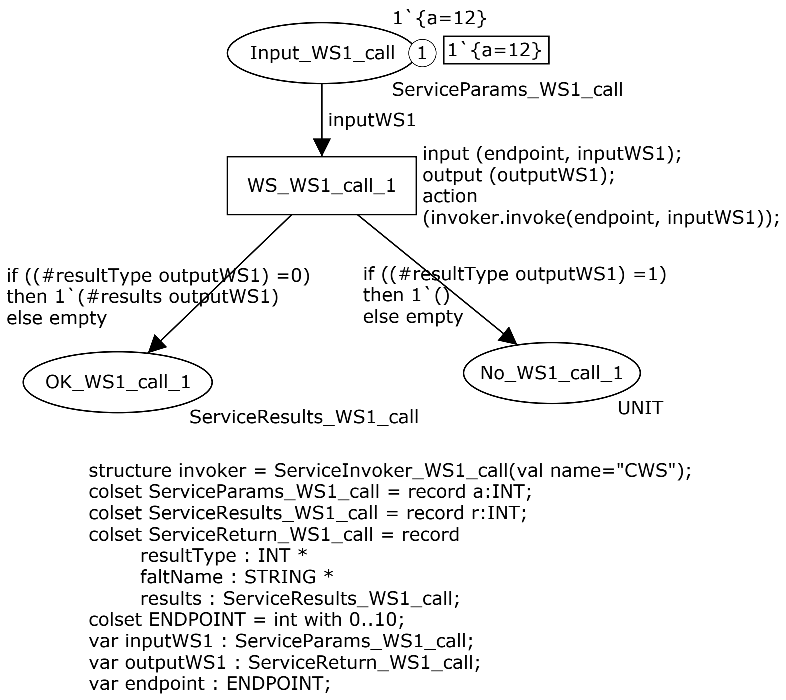
- the initial value for *Start* place (the initial marking),
- the OE graph constructed for an appropriate equivalence class of the CWS,
- the list of binding elements, which is a plan of the execution (it is a path in the OE-graph).

The Execution Engine retrieves a simulator for the CWS, which is used during execution of the CPN model. The algorithm used to execute a CWS is in Listing 5.2:

Listing 5.2: The algorithm for a CWS execution.

```
List<BindingElement> path;
void execute(CPNSimulator simulator, OEgraph graph){
  boolean end = false;
  while (!end){
    BindingElement generalBE = path.getFirst();
    List enabledBE = simulator.getEnabledBindings();
    BindingElement actualBE = findSimilar(generalBE,enabledBE);
    simulator.fireBindingElement(actualBE);
    Marking currentMarking = simulator.getMarking();
    Marking graphMarking = transformToGraphMarking(currentMarking);
    if (actualBE.getTransition().getType().equals(TransitionType.WS)) {
      path = updateBEList(graphMarking);
    }
    end = !graph.reachableSuccess(graphMarking) || graphMarking.isEndMarking();
  }
}
```

Because the OEgraph does not represent exact values for variables representing an output from calls to external WSs (these are default values from an equivalence class), the list of binding elements received from the CWS agent is also not exact. Hence during the execution of the CWS the most similar binding is searched for. The similarity is evaluated in the set of enabled bindings, first using the name of the transition and then each variable is checked, whether there is binding with the same value. The path to execute, which is the list of binding elements, is not complete, because the CWS Agent is uncertain about the actual output from each interaction with an external WS.



```
structure invoker = ServiceInvoker_WS1_call(val name="CWS");
colset ServiceParams_WS1_call = record a:INT;
colset ServiceResults_WS1_call = record r:INT;
colset ServiceReturn_WS1_call = record
        resultType : INT *
        faltName : STRING *
        results : ServiceResults_WS1_call;
colset ENDPOINT = int with 0..10;
var inputWS1 : ServiceParams_WS1_call;
var outputWS1 : ServiceReturn_WS1_call;
var endpoint : ENDPOINT;
```

Figure 5.7: The transition with the implementation of a call to an external WS and the declaration of colors and variables.

So after firing the transition that represents a call, the CWS execution engine maps the output to the appropriate equivalence class and informs the CWS Agent (method `updateBEList()`), which in turn prepares and sends the next list of binding elements to execute.

In order to execute CPN models of CWSs, there must be a mechanism to call external WSs from this model. So if the transition, which is responsible for a call to the external WS is fired, then one of the operations (invoke, send, receive) from the External WS Registry is performed. This is implemented as a plugin to BRITNeY Suite [36], which makes the ServiceInvoker classes (each for an operation from an external WS) visible in the simulation. This class has methods: invoke, send and receive, they are used during the call to an external WS. The example of an invoke transition and its surrounding places is shown in Figure 5.7. To fire this transition there is required binding only for variables defined in the region input ($inputWS1$, *endpoint*), so for variables representing an input and an endpoint. A binding for the variable $outputWS$ is the result of the call to the method `invoke()`, which in turn calls the appropriate WS. The result available in the CPN consists of 3 fields: the type of a response, the value of result if an operation was successful and the name of a fault if the fault is returned.

During an execution of a CWS (after firing each transition) the engine generates events. They contain in general:

- the name of the CWS,

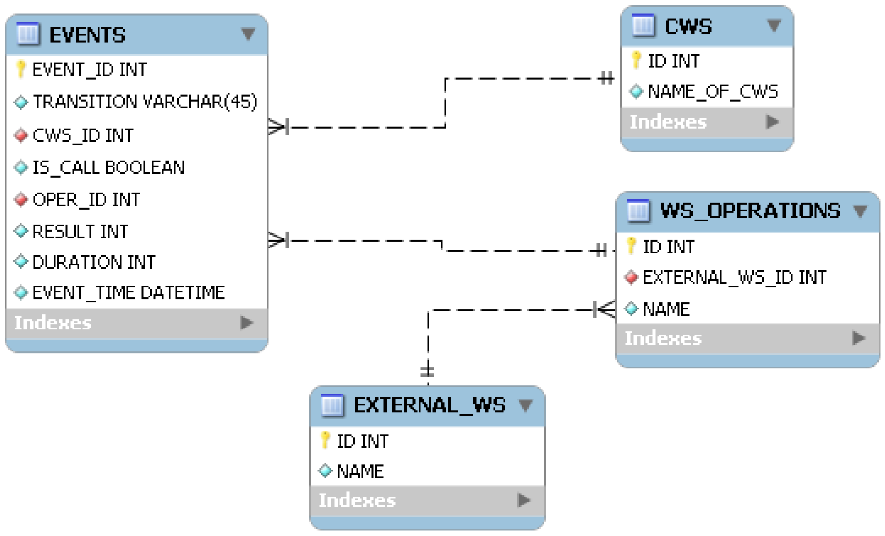


Figure 5.8: The design of a database to store events from the execution of CWSs.

- the name of the transition,
- if the transition is a call to an external WS: name of the WS, name of the operation, endpoint and type of result,
- time required to perform it.

5.5 Data

The Data component stores events generated by the CWS Execution Engine. The contents of the events was presented in the previous section, and in Figure 5.8 is the design of a database that can store these events. Table EVENTS has columns for the name of CWS (in a separate table CWS) and the name of a transition. Optionally an event may be a call to an external WS, and in such a case the additional table is used, which determines an operation (table WS_OPERATIONS) and an external WS (table EXTERNAL_WS).

5.6 Agents

The Agents component is responsible for managing executions of CWSs. There are two types of agents: one for the system and one for each CWS. In general the CWS Agents monitor the state of their CWS, and inform the System Agent about the predicted load for each used external WS. The System Agent monitors the state of all external WSs and prepares the best load distribution among all instances.

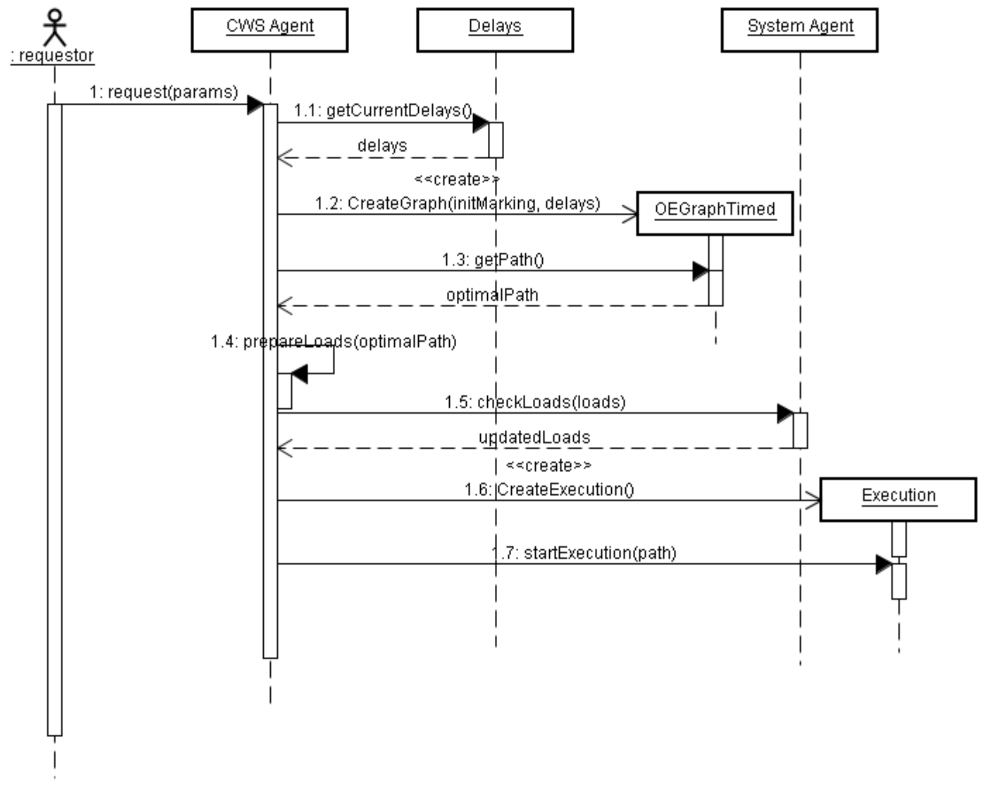


Figure 5.9: The sequence diagram of preparing a plan of an execution of a CWS.

5.6.1 CWS Agents

Each CWS has its own agent, which monitors and selects the optimal path to execute it. To achieve this, CWS Agents plan each execution and then monitor whether the execution is as planned. The plan is prepared using the timed version of a CPN model (as presented in Section 4.4) and the current delays for all transitions and the state of all endpoints (instances) for external WSs. The System Agent informs CWS Agents about any changes in delays or in states.

A CWS Agent prepares a plan of the execution as a list of binding elements that should be fired by the CWS Execution Engine. The plan is prepared when there is a new request to the CWS, or if the execution differs from the plan, or if state of an external WS required by the CWS changes. The initial plan is prepared using parameters of a request to a CWS and contains all binding elements required to successfully execute the CWS, if this is possible. The execution may be different than planned, because the output from an external WS is not as predicted in the plan. In this case a plan is prepared starting from the state the execution is currently in. Finally a state of an external WS can change (for example it may become unavailable), then if this WS is used in remaining execution of the CWS, the new plan has to be prepared.

Figure 5.9 shows the activities of a CWS Agent during planning an execution of a CWS. First a CWS Agent gets current delays and states for transitions and external WSs. These delays and initial marking (obtained from values of input parameters) are used to prepare a timed occurrence graph. The path in this graph is a plan of the execution. It is the basis to determine loads, which are sent to the System Agent. The System Agent responds with the same or updated list of loads (for example it changes an endpoint), and the changes are incorporated into the plan. The execution of a CWS starts with the path to the first call to an external WS, as presented in the previous section. These activities are done every time the CWS Agent must prepare a plan; however, for subsequent plans the initial marking to construct a timed graph is the current marking.

Each plan of the execution is a list of binding elements and is a path in a timed graph, for which a successful end marking is reached in the smallest amount of time. The algorithm to prepare the timed occurrence graph and to choose the optimal path in it, which becomes a plan, is presented in Listing 5.3.

Listing 5.3: The algorithm for planning.

```
Graph graph = new Graph();
Simulator simulator;
Delays delays;
List<MarkingTimed> endMarkings;

void createGraph(MarkingTimed startMarking, long modelTime){
    List<BindingElement> enabledBE = simulator.getEnabledBE();
    if (enabledBE.isEmpty())
      endMarkings.add(simulator.getMarking());
    for (BindingElement be : enabledBE){
      if (be.getTransition().isCall()){
        List<Endpoint> endpoints = getBestEndpoints(delays, be.getTransition());
        for (endpoint : endpoints)
          for (EquivClass equivClass : be.getTransition().getClasses()){
            MarkingTimed endMarking;
            if (endpoint.isValid()){
              endMarking = prepareMarking(equivClass, modelTime);
            } else {
              endMarking = prepareNoResponse(modelTime);
            }
            simulator.setMarking(endMaring);
            be.getBinding().setEndpoint(endpoint);

            graph.addNode(endMarking);
            graph.addArc(startMarking, endMarking, be);
            createGraph(endMarking, modelTime + endpoint.getDelay());
          }
        }
      } else {
        simulator.fire(be);
        MarkingTimed endMarking = simulator.getMarking();
        endMarking.setTime(modelTime);

        graph.addNode(endMarking);
        graph.addArc(startMarking, endMarking, be);
        createGraph(endMarking, modelTime + delays.getDelay(be.getTransition()));
      }
    }

    List<BindingElement> getPath(MarkingTimed initial){
      List<BindingElement> path;

      MarkingTimed shortest = null;
      for (MarkingTimed marking : endMarkings){
        if (marking.isSuccessful() && marking.getTime() < shortest.getTime())
          shortest = marking;
      }
      if (shortest != null)
        path = graph.findPath(initial, shortest);
      return path;
    }
}
```

First all possible binding elements are determined. If it is a call, an endpoint (instance) of an external WS with the shortest response time is chosen and an appropriate marking is created for each possible equivalence classes. Two endpoints are considered only if the working endpoint is overloaded, then it is checked whether a call to an external WS can be omitted. If there is not any valid endpoint, the no response type of output is generated. In case of other transitions, a simulator simply fires them. During a construction of a graph, there are stored end markings, so the ones for which no transitions are enabled, which are used to select the optimal path. First the earliest end marking is selected if it represents a successful execution and its time value is the smallest. Then a path between initial marking and the selected end marking is a plan of the execution of this CWS.

5.6.2 System Agent

The main functionality of the System Agent is to manage the load on all external WSs and to monitor their current state and performance. It uses the Reasoning Mechanism component and also communicates with all CWS Agents to gather their load requirements.

Each CWS agent sends the load requirements to the system agent. The load is represented as shown in the class diagram in Figure 5.10. It is identified by its CWS and and the used external WS. After receiving the load from each CWS Agent, the System Agent aggregates it and with the

Reasoning Mechanism analyzes whether it is possible to avoid overloading external WSs.

The System Agent for each instance (endpoint) of an external WS maintains its current state. A state of an instance consists of:

- a status - there are three types of status: normal, overloaded and not responding.
- response time - as an average of last calls to this instance.

In the normal status response times for operations may change. If this change is sudden and big an external WS moves to the overloaded status for a predefined amount of time, in which it does not receive any requests. If an instance does not respond at all, this instance is marked as not responding and its operations are not used anymore. The Reasoning Mechanism changes states of external WSs, and it is presented in the following section.

5.7 Reasoning Mechanism

The Reasoning Mechanism is responsible for analyzing data from the Data component. Based on the data it produces conclusions about time delays for transitions and states of instances of external WSs. The Reasoning Mechanism also checks whether it is possible to handle the load without overloading.

The Reasoning Mechanism uses the Jess rule engine [33] and data gathered during executions of CWSs. There are 2 sets of rules, and they are presented in Appendix B. The first group analyzes states of external WSs and time delays for transitions, the second analyzes whether the load on external WSs is distributed in the optimal way.

To reason about current states and delays the Reasoning Mechanism uses data from a database, transformed to facts required by the rule engine. In general there are 3 types of facts: for a delay of an operation from an external WS, for a status of an external WS and for a delay of a transition other than a call. Additionally for each of the above there are facts for current values (as opposed to the last ones from a database), they are created when the new fact for a transition or an external WS is added. Adjusting a delay for a transition is done if its current value differs from the last one. For each operation of external WSs there is stored an average response time (from a database) and a normal response time. The latter one is changed only if there is more definite difference in

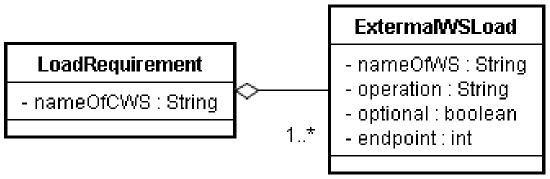


Figure 5.10: The class diagram for a load requirement.

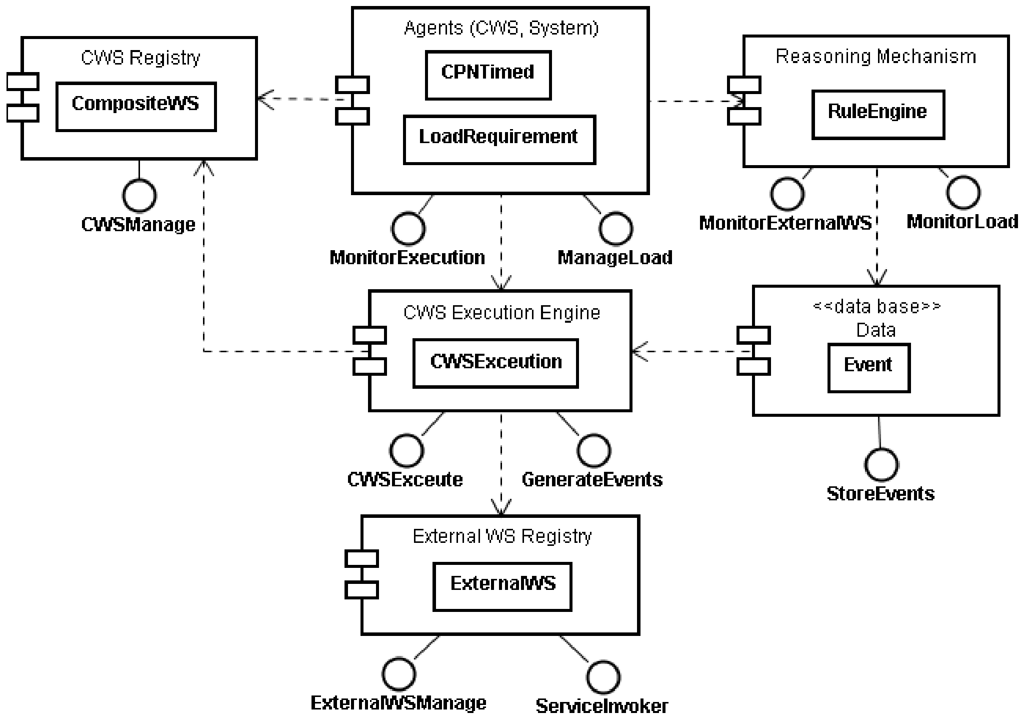


Figure 5.11: The refined version of the component diagram for the model-aware middleware architecture.

an average. There are 3 types of a status: normal, overloaded and not responding. The status is changed to not responding if such an event is received from a database, and this status is final. Moving to the overloaded status is temporary, and is done if the last response time is much bigger than normal, and the average is increasing.

The second set of rules is responsible for analyzing future loads on endpoints for external WSs. The main goal of this is to avoid overloading them. When a status of an external WS is changed to overloaded, then the last interarrival time is stored, and it indicates a possibly overloaded situation. So if there are 2 CWSs that call this endpoint with interarrival less then stored, then one of them is moved to another endpoint. If the change is not possible, because there are no other working endpoints or it can cause overloading on the new endpoint, then a call is omitted if it is optional. If it is compulsory the call is till performed.

5.8 Summary

In the above sections the framework for the model-aware middleware for CWSs was presented. It uses all conceptual aspects that deal with modeling, executing and planning CWSs with the CPN. However, here they were presented from the implementation point of view.

To summarize the above architecture the components diagram from the beginning of the chapter (Figure 5.1) is presented again in Figure 5.11. It is enhanced with interfaces offered by each component and the most important data that the component uses. The External WS Registry manages information about the used services and allows executing them. The CWS Registry manages data for each CWS, and also makes them available to the CWS Execution Engine component. The CWS Execution Engine executes CWS and generates events gathered in the database. The database stores them, and the Reasoning Mechanism component transforms them into facts. The Agents monitors executions of all CWSs and they manage loads on instances of external WSs.

Chapter 6
Evaluation

The main goal of the evaluation of the model-aware middleware is to prove the feasibility of its implementation and to prove that it can make the execution of Composite Web Services (CWSs) more efficient in terms of the performance and reliability. Another objective is to analyze how the reasoning capabilities of the middleware affect the performance, so what is the cost of reasoning. In order to achieve the above goals several stages of the evaluation were performed. First the middleware was tested whether it provides the required functionalities. Then it was used in the experiments to determine how efficient the middleware is in optimizing CWSs, and how the complexity of CWSs influences the performance. The evaluation process was divided into the following phases:

1. *Phase 1 - testing the middleware.* In this phase the middleware was tested using the test cases described below.

2. *Phase 2 - experiments - optimizing the execution of CWSs.* In this phase the measurements for different settings of CWSs and external WSs were taken and analyzed.

3. *Phase 3 - experiments - analyzing the impact of complexity of CWSs.* In this phase the experiments with the increasing complexity of CWSs were performed.

The above phases are presented in more detail in the following sections. First there are described test cases, then the general settings and data definitions for all experiments. Finally there are shown the results of those experiments.

6.1 Testing model-aware middleware

In this phase all components of the model-aware middleware were tested.

The External WS Registry is responsible for managing external WSs, as well as calling them. Table 6.1 gathers the test cases for this component with achieved correct results.

The next component, the CWS Registry, is responsible for managing CWSs. This component loads models of CWSs and their equivalence classes, then it generates a set of OE-graphs. The test

Table 6.1: The test cases for the External WS Registry.

The description of a test case	**The preconditions**	**Correct results**
Adding a WS ("WS1") from a WSDL file.	the registry is empty	"WS1" service in the registry
Adding a new endpoint for a WS ("WS1").	the registry contains "WS1" with 1 endpoint	"WS1" has 2 endpoints
Removing a WS ("WS1").	the registry contains "WS1"	the registry does not contain "WS1"
Removing an endpoint for a WS ("WS1").	the registry contains "WS1" with 2 endpoints	"WS1" has 1 endpoint
Invoking an operation of a WS ("oper1" in "WS1"), returns a correct response.	the registry contains "WS1" with "oper1", "WS1" is available	"WS1" is called, the return value is of type "Ok"
Invoking an operation of a WS ("oper1" in "WS1"), no response.	the registry contains "WS1" with "oper1", "WS1" is not available	the "WS1" is called, the return value is of type "NoResponse"
Invoking an operation of a WS ("oper1" in "WS1" with a possible fault), returns a fault response.	the registry contains "WS1" with "oper1", "WS1" is available	the "WS1" is called, the return value is of type "Fault"
Sending a message to an operation of a WS ("oper1" in "WS1")	the registry contains "WS1" with "oper1", "WS1" is available	the "WS1" is called
Receiving a response from an operation of a WS ("oper1" in "WS1"), returns a correct response	the registry contains "WS1" with "oper1", "WS1" has received send message	the response is received, the return value is of type "Ok"

cases for this component are in Table 6.2 (the correctness of constructed OE-graphs was checked manually).

The CWS Execution Engine component allows the execution of CWSs gathered in the CWS Registry. It uses a list of bindings to fire transitions. Table 6.3 presents the test cases used to test this component.

The Data component is responsible for storing events in a database. The test cases for this component are described in Table 6.4

The Agents component is responsible for starting and monitoring execution, as well as for gathering the load on all external WSs. It reacts to events from the CWS requestor, the CWS Execution Engine and it uses the Reasoning Mechanism. In Table 6.5 are gathered test cases for this component, divided into two groups for CWS Agents and for the System Agent.

The Reasoning Mechanism retrieves data gathered in a database, transforms them into facts and uses the Jess rule engine [33] to analyze them and produce conclusions. The test cases for generating facts are in Table 6.6, and the rules are presented in Appendix B.

Table 6.2: The test cases for the CWS Registry.

The description of a test case	**The preconditions**	**Correct results**
Adding a CWS (”CWSA”) from a file with the CPN model: 1 invoke operation, 1 equivalence class for a CWS and 1 for a call.	the registry is empty	”CWSA” in the registry, 1 OE-graph is constructed (1 type of a correct result from a call to WS)
Adding a CWS (”CWSA”) from a file with the CPN model: 1 invoke operation, 2 equivalence classes for a CWS and 1 for a call.	the registry is empty	”CWSA” in the registry, 2 OE-graphs are constructed (1 type of a correct result from a call to WS)
Adding a CWS (”CWSA”) from a file with the CPN model: 2 invoke operations, 1 equivalence class for a CWS and 1 for calls.	the registry is empty	”CWSA” in the registry, 1 OE-graph is constructed (1 type of a correct result from each call to WS)
Adding a CWS (”CWSA”) from a file with the CPN model: 2 invoke operations, 1 branch (alternative path), 2 equivalence classes for a CWS and 2 for calls.	the registry is empty	”CWSA” in the registry, 2 OE-graphs are constructed (2 types of a correct result from each call to WS)

Table 6.3: The test cases for the CWS Execution Engine.

The description of a test case	**The preconditions**	**Correct results**
Executing a CWS (”CWSA”) - a successful execution.	the registry contains ”CWSA”	for each fired transition in ”CWSA” events are generated, the result is of type ”Ok”
Executing a CWS with a compulsory call to a WS, which stops responding (”CWSA” with ”WS1”)- failure execution.	the registry contains ”CWSA”, ”WS1” is not available	for each fired transition in ”CWSA” events are generated, after a call an empty path is received, the result is of type ”Failure”
Executing a CWS with an optional call to a WS, which stops responding (”CWSA” with ”WS1”).	the registry contains ”CWSA”, ”WS1” is not available	for each fired transition in ”CWSA” events are generated, after a call the new path is received, the result is of type ”Ok”

Table 6.4: The test cases for the Data.

The description of a test case	**The preconditions**	**Correct results**
Incoming event from CWS Execution Engine (a transition different then a WS call).	the database is empty	the row in table EVENTS with appropriate data
Incoming event from CWS Execution Engine (a transition for a WS call).	the database is empty	the row in table EVENTS with appropriate data

Table 6.5: The test cases for the Agents.

The description of a test case	**The preconditions**	**Correct results**
CWS Agents		
Incoming request for a CWS (”CWSA”).	no other executions of ”CWSA”	OEGraphTimed created, a path chosen, load generated and accepted, execution of ”CWSA” started with bindings to the first call
Incoming event from the Execution Engine: transition for a call fired, marking as planned (no replanning required).	there are executions of ”CWSA”	a path to the next call sent to the Execution Engine
Incoming event from the Execution Engine: transition for a call fired, marking different than planned.	there are executions of ”CWSA”	OEGraphTimed created, a path chosen, load generated and accepted, execution of ”CWSA” resumed with bindings to the next call
Incoming event from the System Agent: a delay changed for an external WS (”WS1”) used in ”CWSA”.	there are executions of ”CWSA”	executions informed to prepare new plans (replanning required) in the next update from the Execution Engine
Incoming event from the System Agent: a load distribution for ”CWSA” changed.	there are executions of ”CWSA”	executions informed to change endpoints
System Agent		
Incoming event from the CWS Agent: new load from ”CWSA” (load accepted).	there are loads from other CWS Agents	aggregated load sent to the Reasoning Mechanism, received the same
Incoming event from the CWS Agent: new load from ”CWSA” (load changed).	there are loads from other CWS Agents	aggregated load sent to the Reasoning Mechanism, received changed, all appropriate CWS Agents informed
Incoming event from the Reasoning Mechanism: delay for a transition changed	there is a CWS Agent for a CWS with the changed transition	the appropriate CWS Agent informed
Incoming event from the Reasoning mechanism: status of an external WS changed	there are CWS Agents	all CWS Agents informed
Incoming event from the Reasoning mechanism: response time for an operation from an external WS changed	there are CWS Agents	all CWS Agents informed

Table 6.6: The test cases for the Reasoning Mechanism.

The description of a test case	The preconditions	Correct results
New event in a database (a transition different then a WS call).	the list of facts is empty, there is an event in a database	a fact added to the rule engine
New event in a database(a transition for a WS call).	the list of facts is empty, there is an event in a database	2 facts added to the rule engine: one for status of a WS, one for a response time

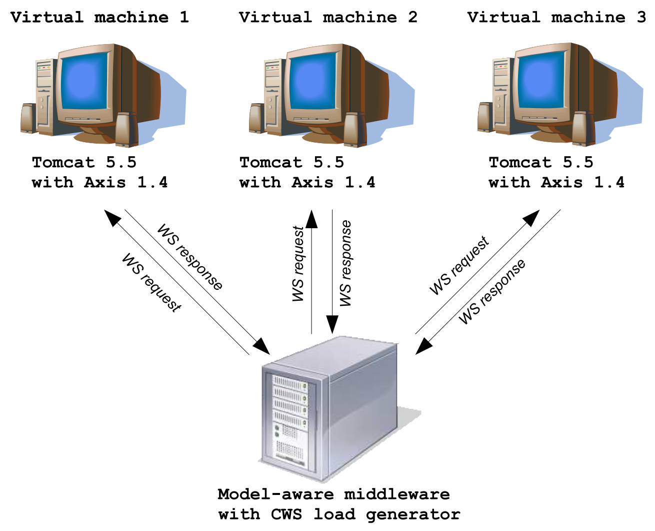


Figure 6.1: The infrastructure used in experiments.

6.2 General settings for experiments

6.2.1 Infrastructure

Figure 6.1 presents the infrastructure used in experiments. The middleware and the generator of load to CWSs is on one machine, as a set of plugins to BRITNeY tool [36]. Each instance of an external WS is on a separate machine (a virtual machine), and all of them use Axis 1.4 [2], deployed on Tomcat 5.5 server [3].

6.2.2 Definition of data used in experiments

The experiment phase of the evaluation is performed to gather and analyze data for different settings of CWSs and external WSs. To quantify the results there are defined: the set of independent (changing) variables, as well as the set of dependent (observed) variables.

The independent variables are of three types:

1. *CWS complexity*, which consists of (for each CWS):

   - n - a number of all calls to external WSs within a CWS,
   - $n_{optional}$ - a number of optional calls to external WSs within a CWS,
   - m - a number of possible alternative paths in the definition of a CWS;

2. *CWS load settings*, which consists of:

   - k - a number of requests,
   - $t_1, t_2, ..., t_k$ - interarrival times;

3. *External WS parameters*, which consists of:

   - N - a number of external WSs in a registry,
   - i_{WSk} for $k = 1, ..., N$ - numbers of instances (endpoints) for each external WS,
   - $S_{WSk,i}$ for $k = 1, ..., N$ and $i = 1, ..., i_{WS_k}$ - a service time of an instance as a function of the number of requests to WS_k,
   - $A_{WSk,i}$ for $k = 1, ..., N$ and $i = 1, ..., i_{WS_k}$ - a reliability of an instance as a function of the number of requests to WS_k, measured as a number of successful calls divided by a number of all calls.

The dependent variables are of 2 types:

1. *Response times*, which consists of:

   - $rt_1, ..., rt_k$ - response time for consecutive requests sent to a CWS,
   - T_{CWS} average response time, defined as:

$$RT_{CWS} = \frac{\sum_{i=1}^{k} rt_i}{k};$$

2. *Reliability*, which consists of:

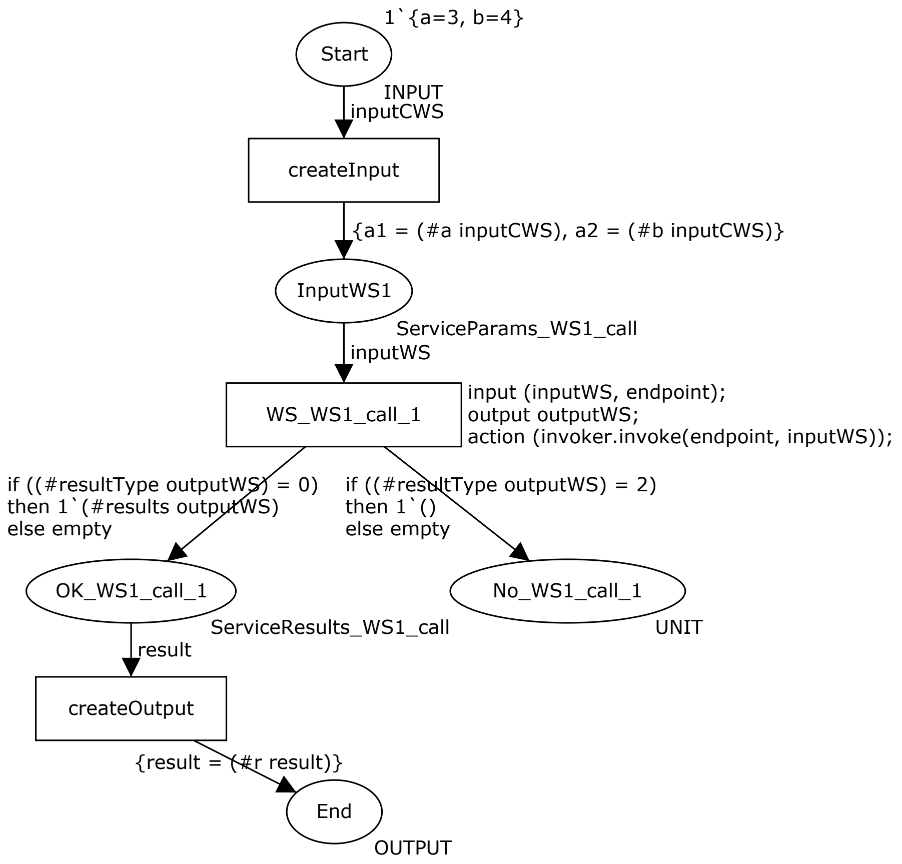
   - $r_1, ..., r_k$ - reliability for consecutive requests sent to a CWS, defined as:

$$r_n = \frac{n_{success}}{n},$$

     where $n_success$ is number of correct results from a CWS,
   - R_{CWS} - overall reliability (which is r_k for k requests).

6.2.3 Definition of CWSs used in experiments

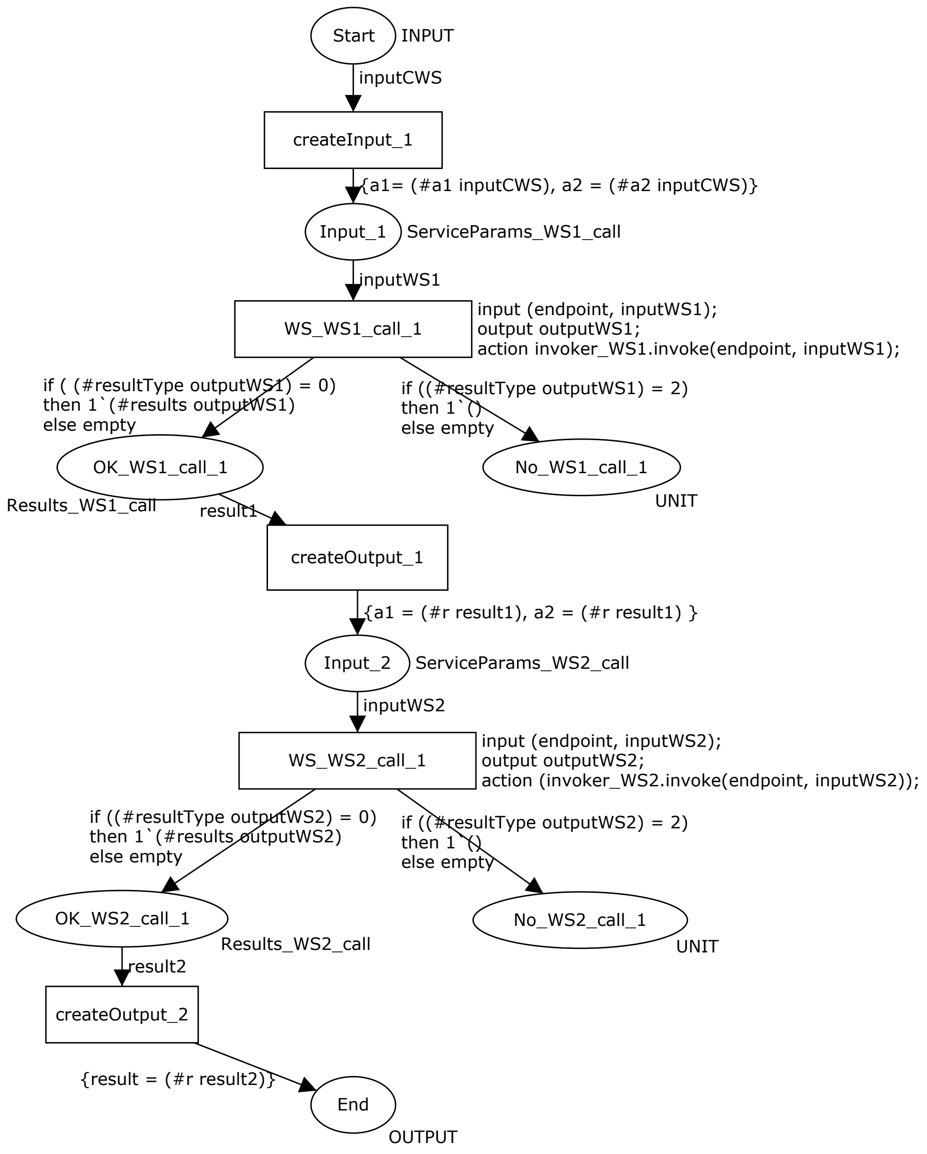
Figures 6.2, 6.3, 6.4 and 6.5 present general models of CWSs used in the experiments. The actual CPN models are different only in names of external WSs and color sets for their input and output (they are non-hierarchical nets, since the BRITNeY tool [36] does not support hierarchies).



```
structure invoker = ServiceInvoker_WS1_call(val name="CWS");
colset INPUT = record a:INT * b : INT;
colset OUTPUT = record result:INT;
colset ENDPOINT = int with 0..9;
colset ServiceParams_WS1_call = record a1 : INT * a2 : INT;
colset ServiceResults_WS1_call = record r : INT;
colset ServiceReturn_WS1_call = record
          resultType : INT *
          faultName : STRING *
          results : ServiceResults_WS1_call;

var inputCWS : INPUT;
var endpoint : ENDPOINT;
var inputWS : ServiceParams_WS1_call;
var outputWS : ServiceReturn_WS1_call;
var result : ServiceResults_WS1_call;
```

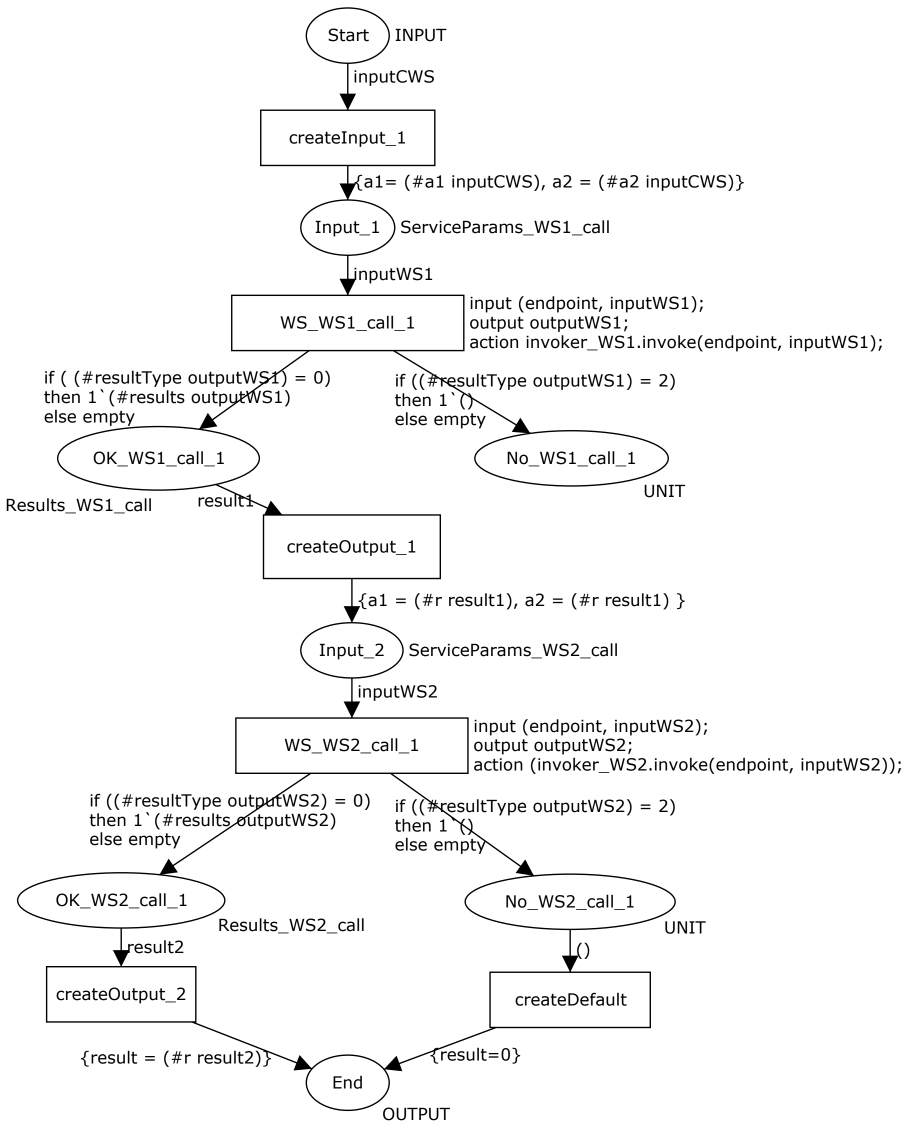
Figure 6.2: The CPN model of a CWS with 1 compulsory call to an external WS ($n = 1$, $n_{optional} = 0$ and $m = 1$).


Start
INPUT
inputCWS
createInput_1
{a1= (#a1 inputCWS), a2 = (#a2 inputCWS)}
Input_1
ServiceParams_WS1_call
inputWS1
WS_WS1_call_1
input (endpoint, inputWS1);
output outputWS1;
action invoker_WS1.invoke(endpoint, inputWS1);
if ((#resultType outputWS1) = 0)
then 1`(#results outputWS1)
else empty
if ((#resultType outputWS1) = 2)
then 1`()
else empty
OK_WS1_call_1
Results_WS1_call
No_WS1_call_1
UNIT
result1
createOutput_1
{a1 = (#r result1), a2 = (#r result1) }
Input_2
ServiceParams_WS2_call
inputWS2
WS_WS2_call_1
input (endpoint, inputWS2);
output outputWS2;
action (invoker_WS2.invoke(endpoint, inputWS2));
if ((#resultType outputWS2) = 0)
then 1`(#results outputWS2)
else empty
if ((#resultType outputWS2) = 2)
then 1`()
else empty
OK_WS2_call_1
Results_WS2_call
No_WS2_call_1
UNIT
result2
createOutput_2
{result = (#r result2)}
End
OUTPUT

```
colset INPUT = record a1 : INT * a2 :INT;
colset OUTPUT = record result : INT;
colset ENDPOINT = int with 0..9;
var inputCWS : INPUT;
var endpoint : ENDPOINT;
```

```
for k=1,2 :
   structure invoker_WSk = ServiceInvoker_WSk_call(val name="CWS");
   colset ServiceParams_WSk_call = record a1:INT * a2 : INT;
   colset Results_WSk_call = record r : INT;
   colset ServiceReturn_WSk_call = record
            resultType : INT *
            faultName:STRING *
            results : Results_WSk_call;
   var inputWSk : ServiceParams_WSk_call;
   var outputWSk : ServiceReturn_WSk_call;
   var resultk : Results_WSk_call;
```

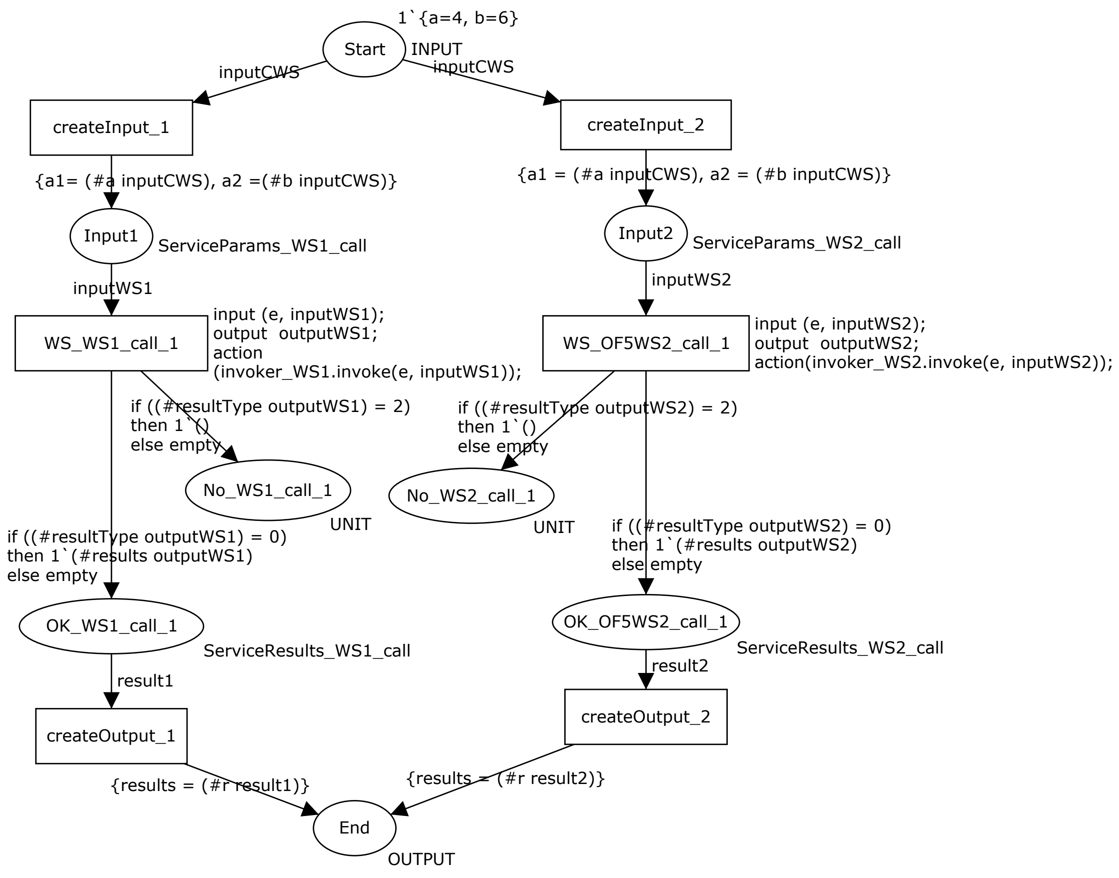
Figure 6.3: The CPN model of a CWS with 2 compulsory calls to external WSs ($n = 2$, $n_{optional} = 0$ and $m = 1$).



```
colset INPUT = record a1 : INT * a2 :INT;
colset OUTPUT = record result : INT;
colset ENDPOINT = int with 0..9;
var inputCWS : INPUT;
var endpoint : ENDPOINT;
```

```
for k=1,2 :
   structure invoker_WSk = ServiceInvoker_WSk_call(val name="CWS");
   colset ServiceParams_WSk_call = record a1:INT * a2 : INT;
   colset Results_WSk_call = record r : INT;
   colset ServiceReturn_WSk_call = record
            resultType : INT *
            faultName:STRING *
            results : Results_WSk_call;
   var inputWSk : ServiceParams_WSk_call;
   var outputWSk : ServiceReturn_WSk_call;
   var resultk : Results_WSk_call;
```

Figure 6.4: The CPN model of a CWS with 2 calls to external WSs with the second one optional ($n = 2$, $n_{optional} = 1$ and $m = 1$).



```
colset INPUT = record a1 : INT * a2 :INT;
colset OUTPUT = record result : INT;
colset ENDPOINT = int with 0..9;
var inputCWS : INPUT;
var e : ENDPOINT;
```

```
for k=1,2 :
   structure invoker_WSk = ServiceInvoker_WSk_call(val name="CWS");
   colset ServiceParams_WSk_call = record a1:INT * a2 : INT;
   colset Results_WSk_call = record r : INT;
   colset ServiceReturn_WSk_call = record
             resultType : INT *
             faultName:STRING *
             results : Results_WSk_call;
   var inputWSk : ServiceParams_WSk_call;
   var outputWSk : ServiceReturn_WSk_call;
   var resultk : Results_WSk_call;
```

Figure 6.5: The CPN model of a CWS with 2 alternative paths with calls to external WSs ($n = 2$, $n_{optional} = 0$ and $m = 2$).

6.3 Results of experiments

6.3.1 Results of experiments in optimizing executions of CWSs

In optimization experiments the performance of the middleware is compared with two other propositions:

- the BPEL-like type of execution - for all executions always the first instance (endpoint) of an external WS is chosen,
- the QoS-like type of execution - before each execution of a CWS an instance with the shortest response time is chosen, or if all instances are not working a WS is not called.

Both of these types of executions use the CPN models, but the planning capabilities of the middleware are not used.

All experiments are performed for $k = 100$ executions and 3 different load types:

- constant light load - with $\Delta_t = 2000$ ms,
- constant heavy load - with $\Delta_t = 1000$ ms,
- exponential distribution of interarrival times with $\lambda = 0.5$ requests/s.

The experiments are divided into 3 groups: overcoming failures, choosing an optimal path and dynamic service selection.

Overcoming failures

Overcoming failures of external WSs means that faulty instances of external WSs are not used anymore. In case of optional calls it means that even though external WSs are not called, a CWS can be still executed successfully. The settings of experiments for this part are presented in Table 6.7.

In the experiment OF-1 the CWS calls one external WS, which has 2 instances and the first instance stops responding. Figure 6.6 presents the response times of this CWS for different load types. For the model aware execution response times are the highest, if the first instance of $WS1$ is responding, since this type of execution requires additional reasoning. After the 60th request (which is the 50th call to the first instance of $WS1$, since it takes several calls to test both instances), the middleware detects that this instance is not working, and selects the second and slower one. The same holds for the QoS-like type of execution. For the BPEL-like type the response times after the 50th request (the second instance is not called at all) are high, because it still calls the not working instance of $WS1$ and it takes certain amount of time. For the heavier load response times are a little bit higher, and for the exponential load they are more fluctuating. What is characteristic for

Table 6.7: The settings for experiments for the capability of the middleware: overcoming failures.

Experiment	**Description**	**Setup - CWS Complexity**	**Setup - External WS parameters**
Experiment OF-1	Switching between instances of an external WS - the first one stops responding.	CWS1 (Fig. 6.2): $n = 1$, $n_{optional} = 0$, $m = 1$	$N = 1$, $i_{WS1} = 2$, $S_{WS1,1} = 200$ms, $A_{WS1,1} = 100\%$ for 50 calls then decreasing, $S_{WS1,2} = 500$ ms, $A_{WS1,2} = 100\%$.
Experiment OF-2	Switching between instances of an external WS - the first one stops responding (several CWSs call the same external WS).	CWS1, CWS2, CWS3 (Fig. 6.2): $n = 1$, $n_{optional} = 0$, $m = 1$	$N = 1$, $i_{WS1} = 3$, $S_{WS1,1} = 200$ms, $A_{WS1,1} = 100\%$ for 100 calls then decreasing, $S_{WS1,2} = 500$ms, $A_{WS1,2} = 100\%$ $S_{WS1,3} = 500$ms, $A_{WS1,3} = 100\%$.
Experiment OF-3	Omitting optional calls to an external WS, which stops responding ($WS2$ in CWS1).	CWS1 (Fig. 6.4): $n = 2$, $n_{optional} = 1$, $m = 1$	$N = 2$, $i_{WS1} = 1, i_{WS2} = 1$, $S_{WS1,1} = 200$ms, $A_{WS1,1} = 100\%$, $S_{WS2,1} = 200$ms, $A_{WS2,1} = 100\%$ for 50 calls then decreasing.
Experiment OF-4	Omitting optional calls to an external WS, which stops responding (several CWSs, all of them have $WS2$ as optional).	CWS1, CWS2, CWS3 (Fig. 6.4): $n = 2$, $n_{optional} = 1$, $m = 1$	$N = 2$, $i_{WS1} = 3, i_{WS2} = 3$, $S_{WS1,1} = 200$ms, $A_{WS1,1} = 100\%$, $S_{WS1,2} = 200$ms, $A_{WS1,2} = 100\%$, $S_{WS1,3} = 200$ms,$A_{WS1,3} = 100\%$, $S_{WS2,1} = 200$ms, $A_{WS2,1} = 100\%$ for 50 calls then decreasing, $S_{WS2,2} = 200$ms, $A_{WS2,2} = 100\%$ for 50 calls then decreasing, $S_{WS2,3} = 200$ms, $A_{WS2,3} = 100\%$ for 50 calls then decreasing.
Experiment OF-5	Omitting a path with an external WS that stops responding ($WS1$).	CWS1 (Fig. 6.5): $n = 1$, $n_{optional} = 0$, $m = 2$	$N = 1$, $i_{WS1} = 1, i_{WS2} = 1$, $S_{WS1,1} = 100$ms, $A_{WS1,1} = 100\%$ for 20 calls then decreasing, $S_{WS2,1} = 500$ms, $A_{WS2,1} = 100\%$.

this experiment, and also for all others presented in this section, is that for several first requests to a CWS, response times are very high. This happens because at the beginning it takes time to start the servers, and also to initialize the middleware. Figure 6.7 presents the reliability for all types of load for the above settings. They are very similar for all load types: for the model aware and QoS-like executions the reliability slightly decreases after detecting the not working instance of $WS1$, and for the BPEL-like the reliability decreases continuously after this point. Table 6.8 gathers average response times and the overall reliability for the experiment OF-1. The highest average response times and the lowest reliability is for the BPEL-like type of execution. For the other types the selection of the not working instance is omitted so they have better average response times. The model aware execution is worse than the QoS-like in terms of response time, because it requires additional reasoning.

The experiment OF-2 is done for the same CPN model of a CWS as in the experiment OF-1, however it is loaded 3 times, so there are 3 CWSs in the middleware (each of them creates its own set of graphs, chooses the optimal paths, and so on). The average response times of 3 CWSs for all types of execution and load are shown in Figure 6.8. The results are similar to the ones in Figure 6.6, but the change happens earlier. This is due to the fact that the first instance of $WS1$ stops responding after 100 calls, and there are 3 CWSs. Therefore after the 40th request in the model aware and QoS-like type of execution only the working instances are selected, and in the BPEL-like after the 33th request the same, not working one is still used. Also in this case response times for the exponential load are more variable. The reliability is again almost the same for all types of loads, so Figure 6.9 presents only the one for the constant light load. Besides the earlier change, reliabilities are similar as in Figure 6.7. Table 6.9 gathers the average response times and the overall reliability for this experiment. The average response times are the highest for the BPEL-like execution, and the smallest for the QoS-like. After comparing those results with the previous experiment (in Table 6.8), it is important to note that they are bigger (about 20%,) for the constant light load only. However, response times increase after the 33th request, not after the 50th as in the experiment OF-1, so it is hard to conclude whether introducing additional CWSs to the middleware does affect their response times.

In the experiment OF-3 a CWS interacts in a sequence with 2 external WSs: $WS1$ and $WS2$. The call to $WS2$ is optional, so when it stops responding the CWS can be still successfully executed. The consecutive response times for this CWS are presented in Figure 6.10. The results show that after the 50th request, if $WS2$ is not responding, the response times decrease for the model aware and QoS-like type of execution. This is because calls to $WS2$ are omitted, and it takes less time than the actual call. In the BPEL-like execution requests to $WS2$ are still sent, which causes additional delays that affect response times of the CWS. Response times in the exponential type of load are less constant, compared to the others, which is similar to the results of previous experiments. In these settings the reliability of all types of loads is 1, since the not working external WS is optional. The average response times and the overall reliabilities are in Table 6.10. The worst performance is in the BPEL-like type of execution, the best in the QoS-like one. Also there are slightly higher response times for heavier loads for the model aware and QoS-like executions.

The experiment OF-4 is performed for the same model of a CWS as in OF-3, but it is used for 3 CWSs. Moreover there are 3 instances of $WS1$ and $WS2$, and all instances of $WS2$ stops responding. Figure 6.11 shows response times for different types of load as average for 3 CWSs. In the model aware and QoS-like type of execution there are "peaks" in response times around the 25th, 38th and 54th requests. This is when 3 instances of $WS2$, one after another, stop responding. After the last "peak" the response times decrease, since $WS2$ is not called anymore. For the BPEL-like type, because only the first instance is used and the CWSs always interact with it, after the 17th request response times are higher. For the exponential type of load "peaks" are not that distinctive as in other types, and response times vary substantially. Table 6.11 shows the average response times and the overall reliability, which again is 1. As in the previous experiment the worst is the BPEL-like type of execution and the best the QoS-like. These results are substantially higher (the difference is about 20% for all averages) than in the experiment OF-3 with only one CWS. In both settings the response times decrease after the similar number of requests, thus in this case having more CWSs in the middleware results in the slower executions.

In the model of a CWS in the experiment OF-5 there are 2 alternatives: one consisting of an interaction with $WS1$ and one with $WS2$. After 20 calls $WS1$ stops responding. The response times for requests to this CWS are shown in Figure 6.12. For the model aware type of execution there is only a minimal increase after the 20th request, and this represents a selection of a path with the working but slower external WS. For the BPEL-like and QoS-like types of execution the selection of the path is random, so the variability in response times is more significant. In case of the BPEL-like type, after the 40th request the not working $WS1$ is still used and then the response times of the CWS are very high. For the QoS-like type these interactions are omitted, and then response times are small. Figure 6.13 presents the reliability of the CWS (only for the constant light load, for the other types of load the results are very similar). In this experiment the model-aware type of execution has much higher reliability after the 40th request. This happens, because only for the model-aware type it is possible to predict that selection of a path with the faulty $WS1$ causes the failure of the whole execution. In other types of execution the not working path is still randomly selected, and for these requests, the reliability decreases. Table 6.12 gathers the average response times and the overall reliability for these settings. The average of response times for the model-aware executions is much higher for all loads than for the other types. But in this experiment the important improvement is in the reliability, which for the model aware type is almost 1, and for the other types is around 0.7.

Table 6.8: Average response times and overall reliabilities in the experiment OF-1.

Execution type	**Constant light load**	**Constant heavy load**	**Exponential load**
Response times in ms			
Model aware	972	1101	1146
BPEL-like	1241	1351	1366
QoS-like	581	857	699
Overall reliability			
Model aware	0.98	0.96	0.98
BPEL-like	0.51	0.50	0.50
QoS-like	0.98	0.97	0.98

Table 6.9: Average response times and overall reliabilities in the experiment OF-2.

Execution type	**Constant light load**	**Constant heavy load**	**Exponential load**
Response times in ms			
Model aware	1162	1242	1264
BPEL-like	1635	1653	1652
QoS-like	725	853	897
Overall reliability			
Model aware	0.99	0.99	0.99
BPEL-like	0.33	0.33	0.33
QoS-like	0.99	0.99	0.99

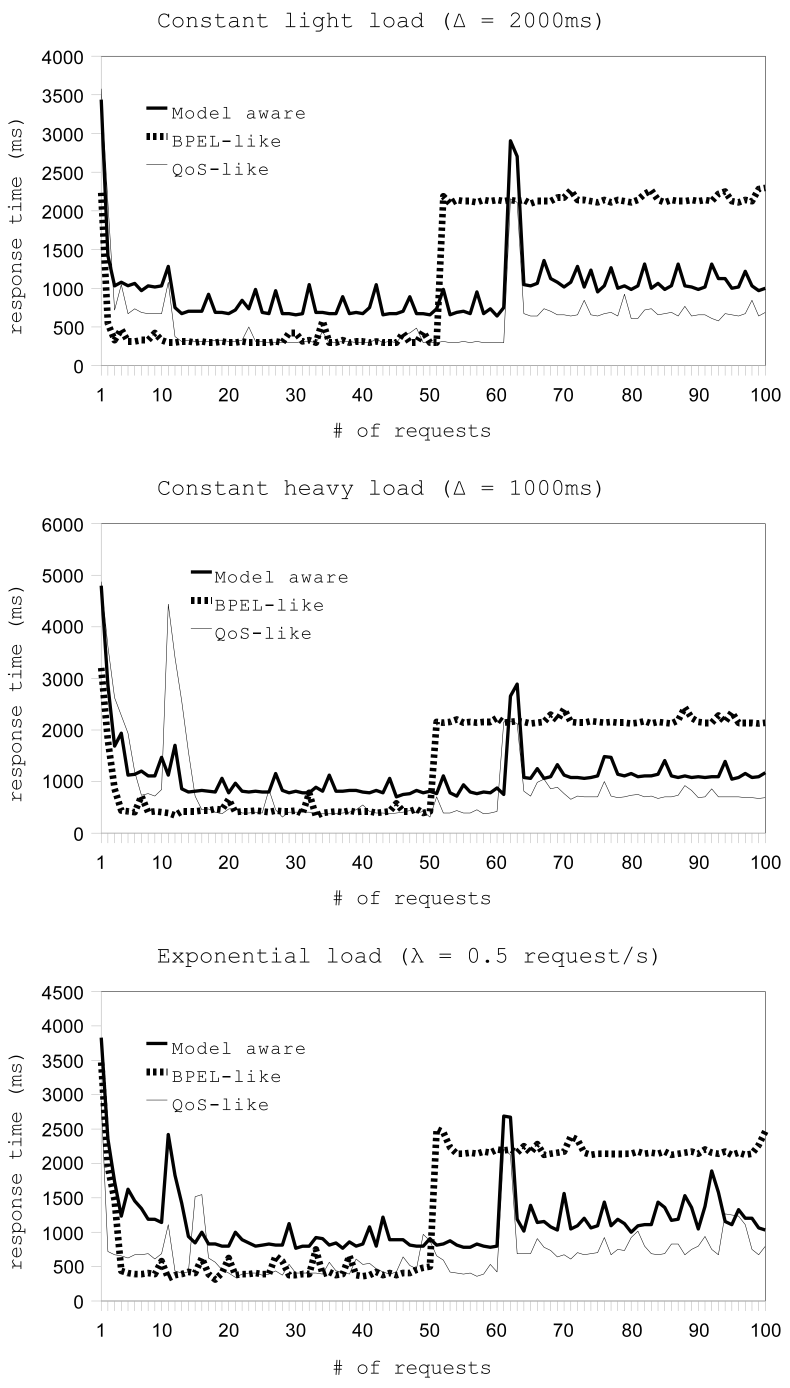


Figure 6.6: Response times for consecutive requests in the experiment OF-1.

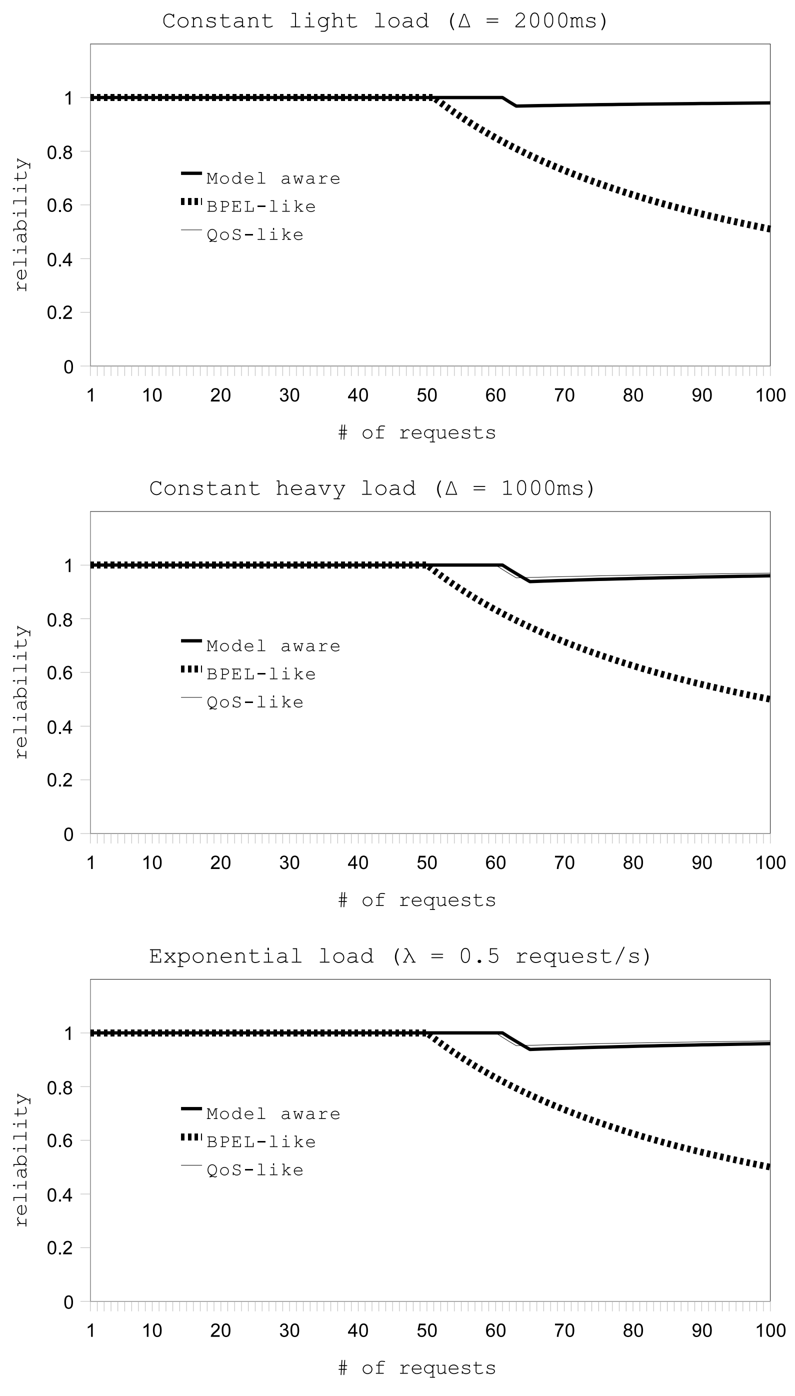
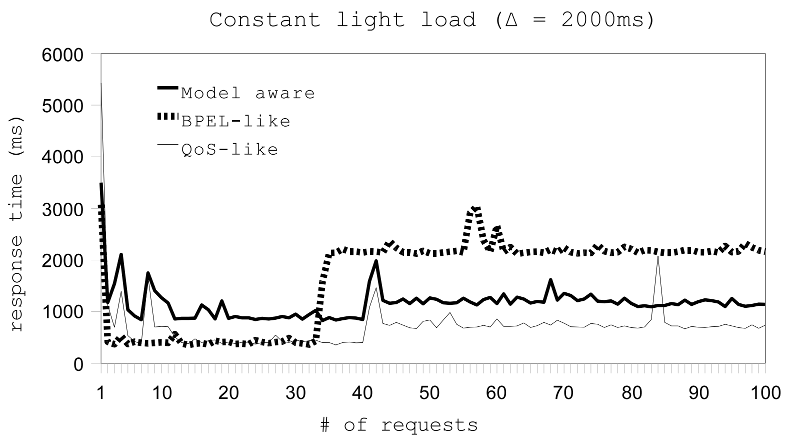
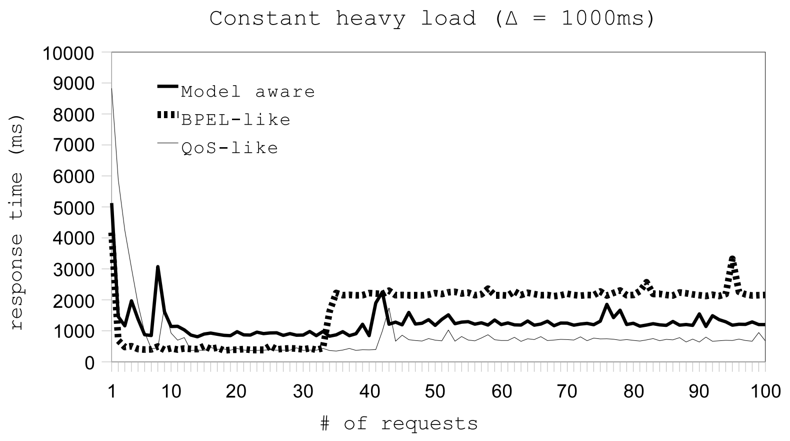


Figure 6.7: Reliability for consecutive requests in the experiment OF-1.





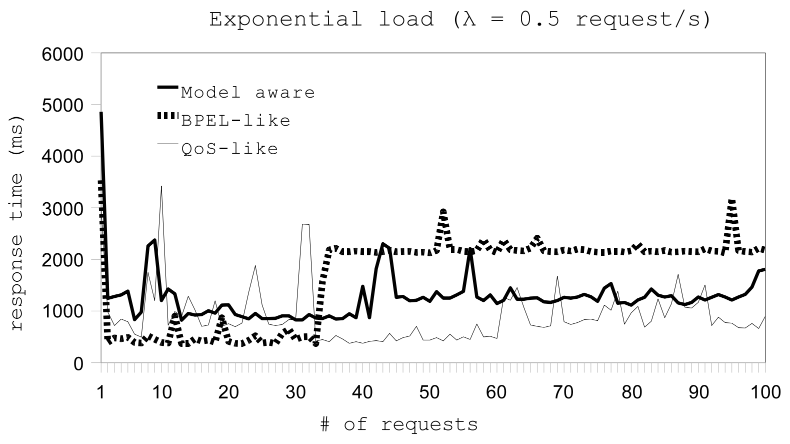


Figure 6.8: Response times for consecutive requests in the experiment OF-2 (average for 3 CWSs).

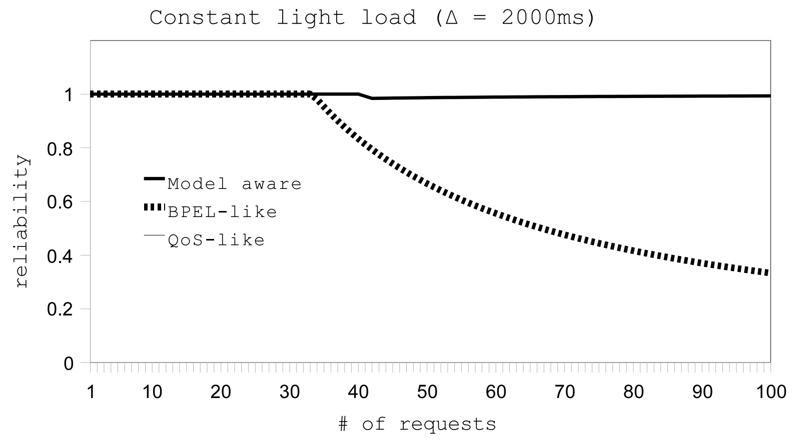


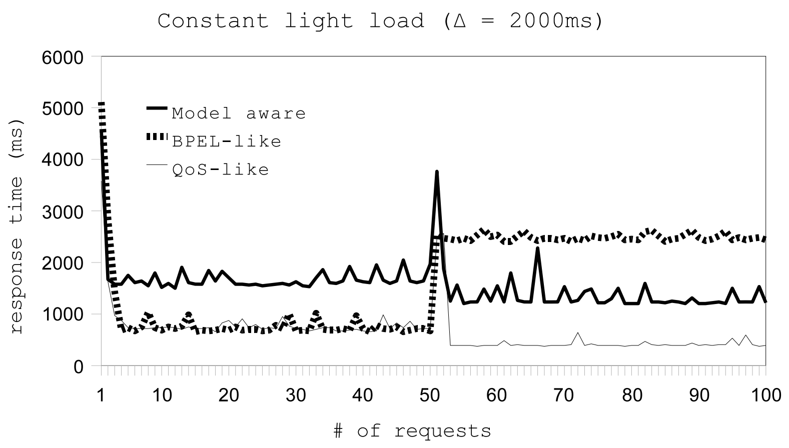
Figure 6.9: Reliability for consecutive requests in the experiment OF-2 (similar for the constant heavy and exponential load).

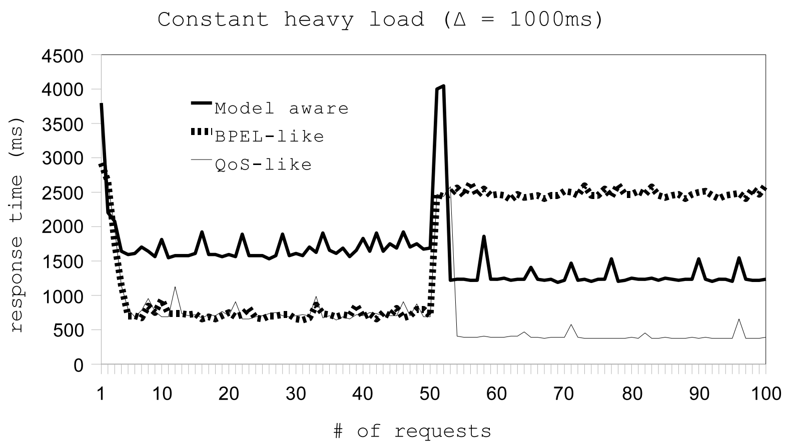
Table 6.10: Average response times and overall reliabilities in the experiment OF-3.

Execution type	**Constant light load**	**Constant heavy load**	**Exponential load**
Response times in ms			
Model aware	1551	1552	1598
BPEL-like	1682	1657	1719
QoS-like	659	697	766
Overall reliability			
Model aware	1	1	1
BPEL-like	1	1	1
QoS-like	1	1	1

Table 6.11: Average response times and overall reliabilities in the experiment OF-4.

Execution type	**Constant light load**	**Constant heavy load**	**Exponential load**
Response times in ms			
Model aware	1904	1991	1966
BPEL-like	2358	2261	2382
QoS-like	798	976	1146
Overall reliability			
Model aware	1	1	1
BPEL-like	1	1	1
QoS-like	1	1	1





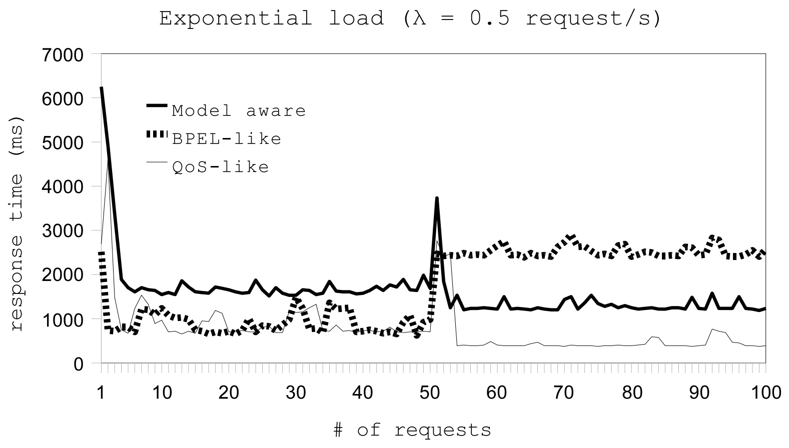
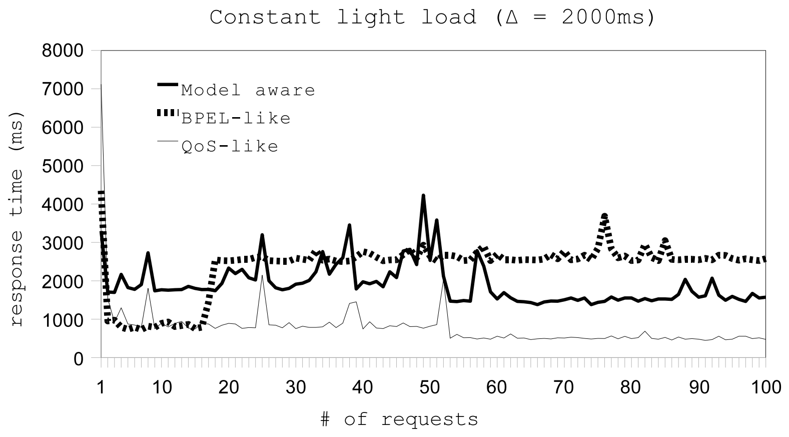
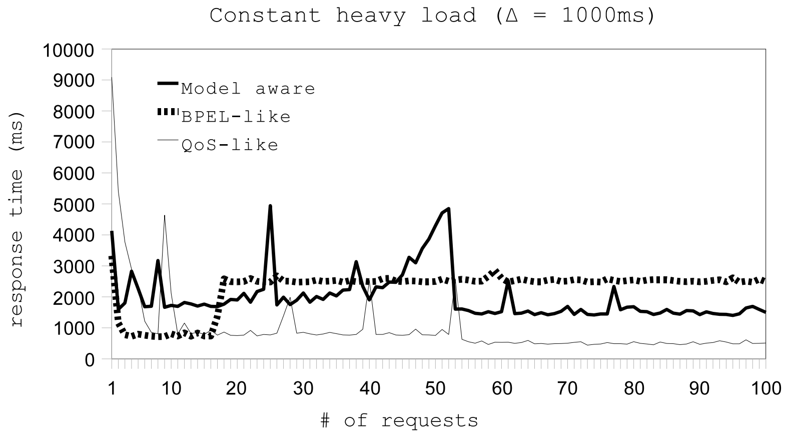


Figure 6.10: Response times for consecutive requests in the experiment OF-3.





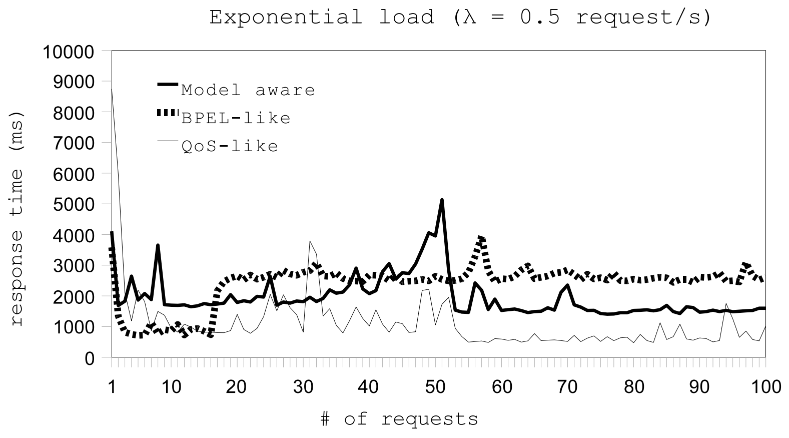


Figure 6.11: Response times for consecutive requests in the experiment OF-4 (average for 3 CWSs).

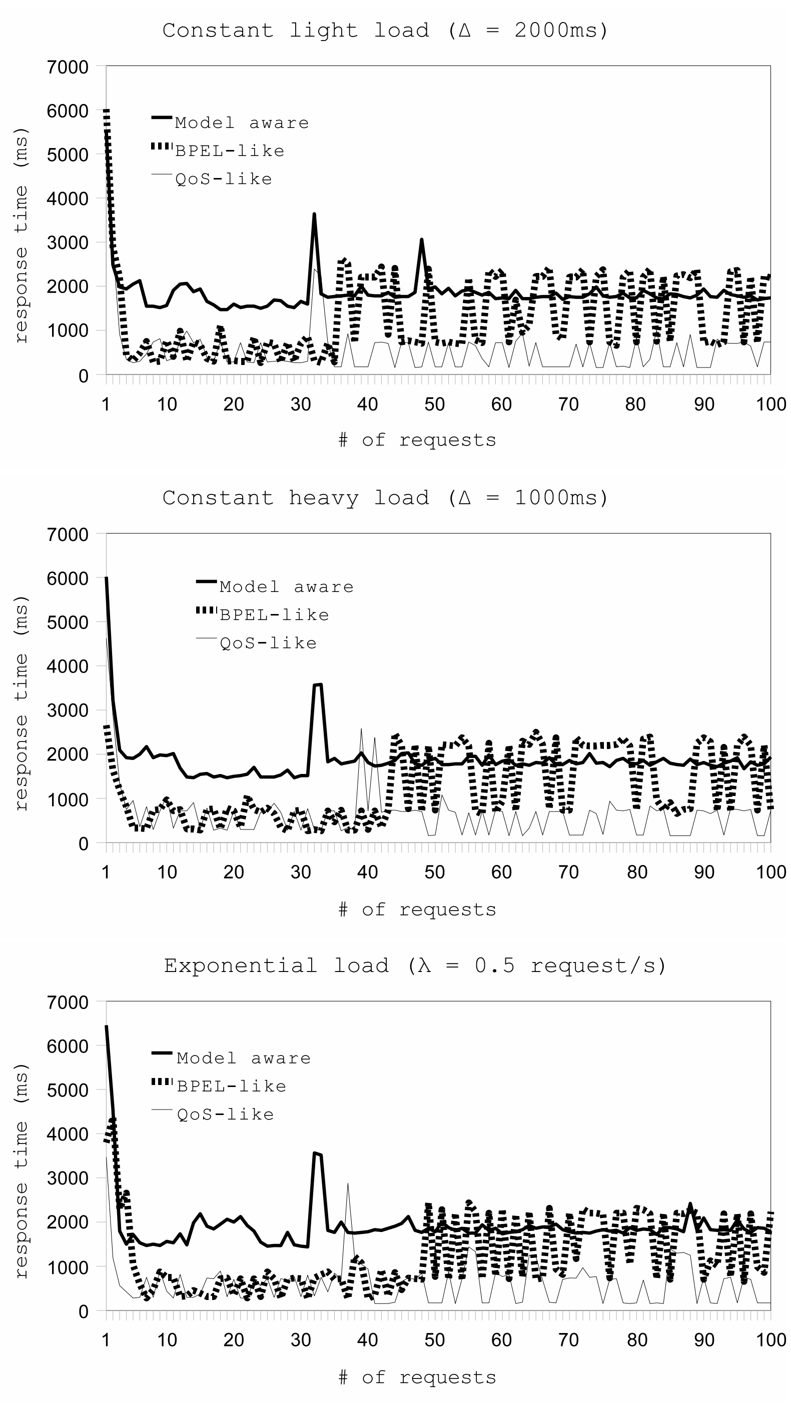


Figure 6.12: Response times for consecutive requests in the experiment OF-5.

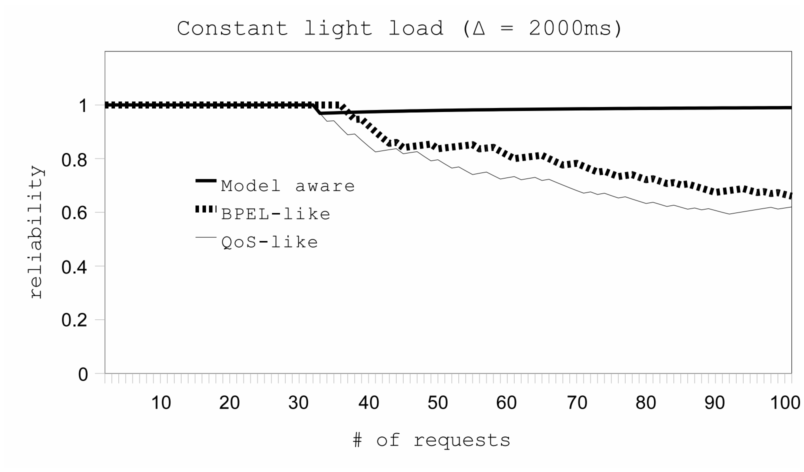


Figure 6.13: Reliability for consecutive requests in the experiment OF-5.

Table 6.12: Average response times and overall reliabilities in the experiment OF-5.

Execution type	**Constant light load**	**Constant heavy load**	**Exponential load**
Response times in ms			
Model aware	1845	1872	1909
BPEL-like	1300	1236	1236
QoS-like	560	646	624
Overall reliability			
Model aware	0.99	0.98	0.98
BPEL-like	0.66	0.65	0.71
QoS-like	0.62	0.76	0.71

Selecting optimal path

If in a CWS there are alternative paths to successfully execute it, then the optimal one should be selected. The criteria of this selection are based on the current state of all external WSs and transitions. For all experiments in this group only response times are measured, because the reliability is always 1. Table 6.13 presents the settings for experiments in selecting the optimal path.

In the experiment BP-1 there are 2 possible alternatives to execute a CWS. In the first one a CWS interacts with $WS1$ and in the second with $WS2$, and service times of $WS2$ is smaller. The subsequent response times for requests to this CWS and for different load types are shown in Figure 6.14. The model aware executions have higher response times, because they require additional reasoning. However, the decision which path to select is based on the data gathered in the previous executions, so after 10 calls to each of external WSs, the path with $WS2$ is used. For the BPEL-like and QoS-like execution the selection is random, that is why their response times

Table 6.13: The experiments for the capability of the middleware: selecting the optimal path.

Experiment	Description	Setup - CWS Complexity	Setup - External WS parameters
Experiment BP-1	Selecting the path with an external WS with smaller response time ($WS2$).	CWS1 (Fig. 6.5): $n = 2$, $n_{optional} = 0$, $m = 2$	$N = 2$, $i_{WS1} = 1$, $i_{WS2} = 1$, $S_{WS1,1} = 500$ms, $A_{WS1,1} = 100\%$, $S_{WS2,1} = 100$ms, $A_{WS2,1} = 100\%$.
Experiment BP-2	Selecting the path with an external WS with smaller response time ($WS2$, for several CWSs).	CWS1, CWS2, CWS3 (Fig. 6.5): $n = 2$, $n_{optional} = 0$, $m = 2$	$N = 2$, $i_{WS1} = 1$, $i_{WS2} = 1$, $S_{WS1,1} = 500$ms, $A_{WS1,1} = 100\%$, $S_{WS2,1} = 100$ms, $A_{WS2,1} = 100\%$.
Experiment BP-3	Changing the optimal path to the path with an external WS with smaller response time (first it is WS2, then WS1).	CWS1 (Fig. 6.5): $n = 2$, $n_{optional} = 0$, $m = 2$	$N = 2$, $i_{WS1} = 1$, $i_{WS2} = 1$, $S_{WS1,1} = 300$ms for 20 calls then 100ms, $A_{WS1,1} = 100\%$, $S_{WS2,1} = 100$ms for 20 calls then 500ms, $A_{WS2,1} = 100\%$.

are on 2 levels, each for a possible paths. The average response times are shown in Table 6.14. As expected the average response times for all types of load are substantially higher for the model aware type of execution. For the above settings of the experiment the average response times are similar for all types of loads.

The experiment BP-2 is similar to the previous one, but 3 CWSs are used. Figure 6.15 shows response times for consecutive requests (it is an average from all 3 CWSs). These results are similar to the ones in the experiment BP-1 with only 1 CWS. The difference is that in the model-aware execution the faster alternative (the path with $WS2$) is selected after a fewer number of requests, because all 3 CWSs interact with it. For the QoS-like and BPEL-like type it is harder to distinguish 2 levels of response times for both alternatives, because Figure 6.15 presents averages for all 3 CWSs (so in the same request 1 CWS can select the first alternative, and another CWS the second). The averages for all requests are shown in Table 6.15. These results are very similar to the ones in Table 6.14. In case of those 2 experiments (BP-1 and BP-2) there is no significant difference between 1 or 3 CWSs in the middleware.

The model of a CWS for the experiment BP-3 is the same as in the previous experiments in this group. However, the response times for external WSs are changing. First the shorter service time has $WS2$, then after 20 calls to this WS, it increases, and the better is $WS1$. The response times for consecutive requests in this experiment are presented in Figure 6.16. For the model aware type of execution after 10 calls to each of WSs, the path with $WS2$ is selected for the next 10 requests. Then after the 30th request this path becomes worse, because service times of $WS2$ increase, so the path with $WS1$ is used. After the 40th request the services times of $WS1$ are even smaller, this can be also observed in the performance of this CWS (especially in the constant heavy type of load). For the QoS-like and BPEL-like type of execution, although the response times are smaller, their variability is much higher, especially after the 40th request. Table 6.10 shows the average response times for these settings. These results demonstrate again that the model aware type of execution is slower. In a case of this CWS the average response times are almost the same for the light and heavy constant load, and slightly higher for the exponential.

Table 6.14: Average response times in the experiment BP-1.

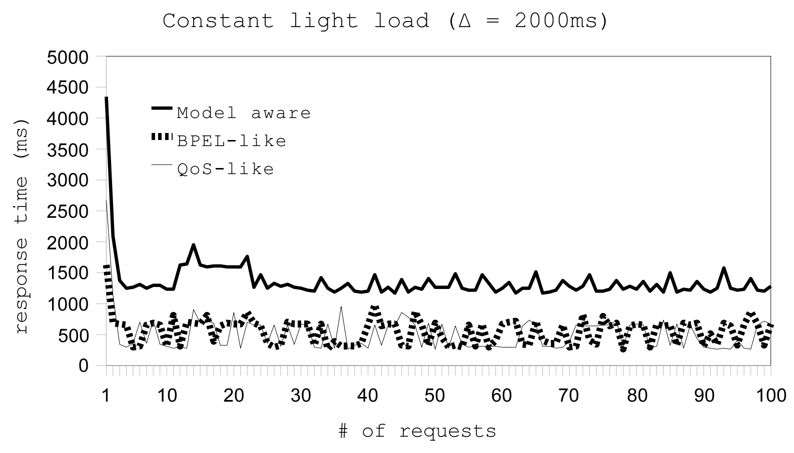
Execution type	Constant light load	Constant heavy load	Exponential load
Model aware	1359	1330	1355
BPEL-like	555	576	602
QoS-like	511	526	595

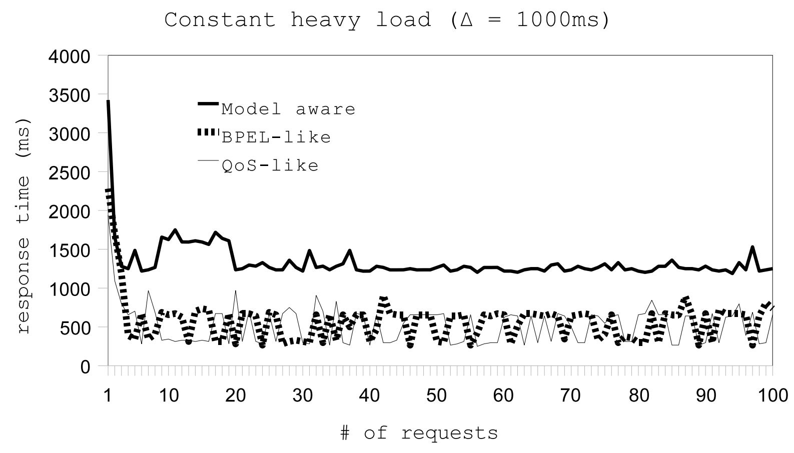
Table 6.15: Average response times in the experiment BP-2.

Execution type	Constant light load	Constant heavy load	Exponential load
Model aware	1313	1303	1327
BPEL-like	520	526	613
QoS-like	544	528	609

Table 6.16: Average response times in the experiment BP-3.

Execution type	Constant light load	Constant heavy load	Exponential load
Model aware	1350	1332	1365
BPEL-like	448	520	574
QoS-like	543	536	592





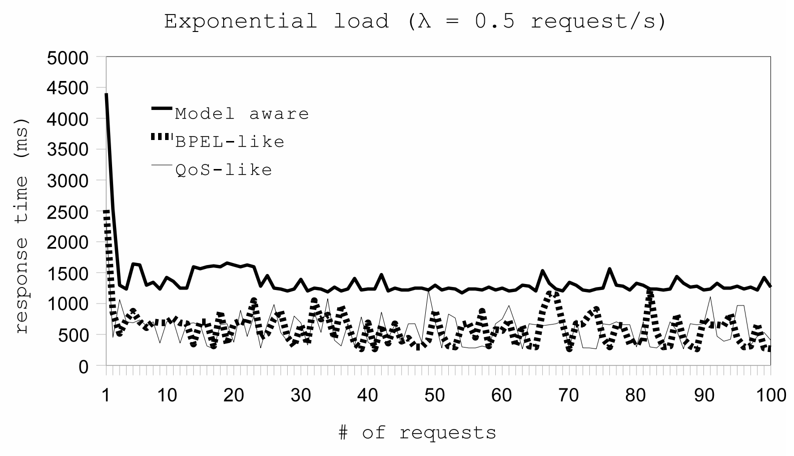


Figure 6.14: Response times for consecutive requests in the experiment BP-1.

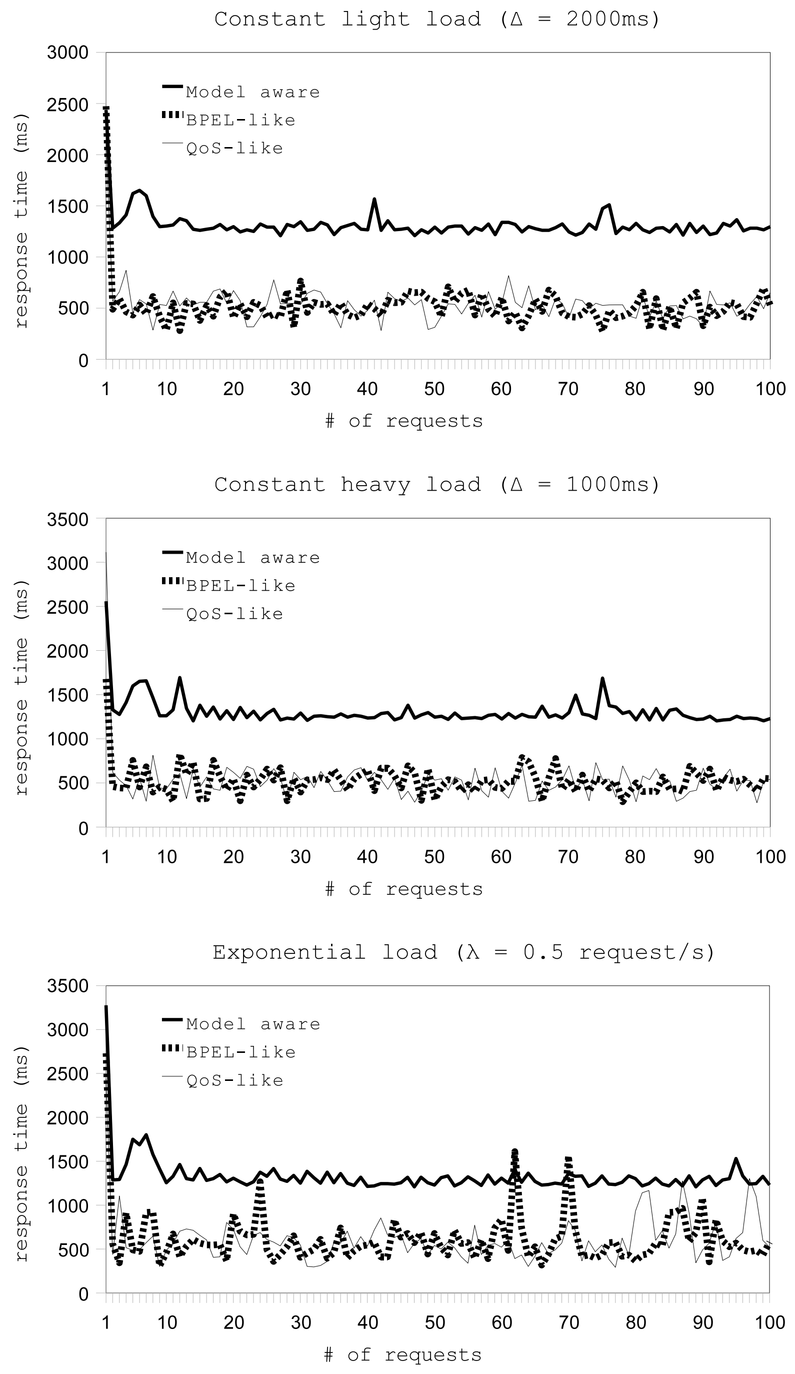
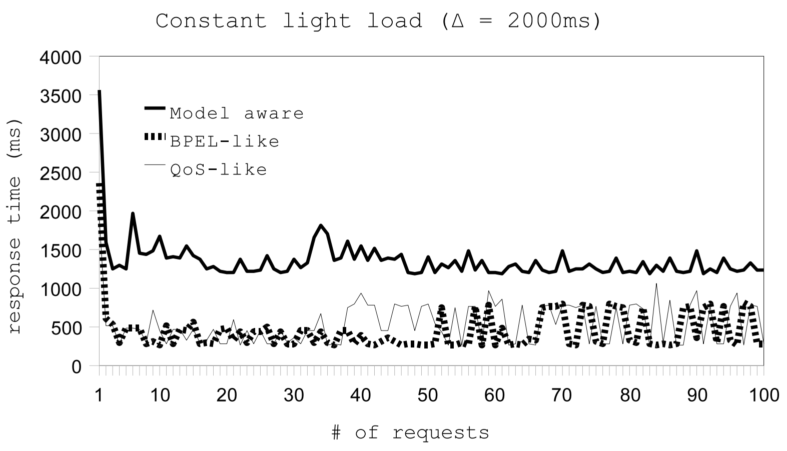
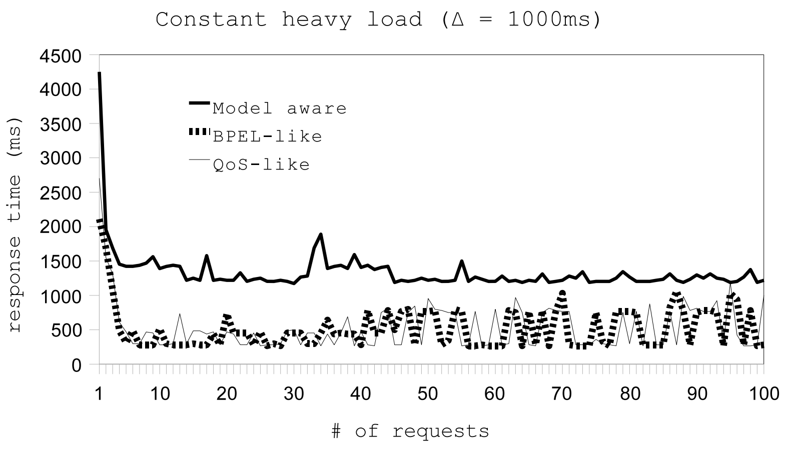


Figure 6.15: Response times for consecutive requests in the experiment BP-2 (average for 3 CWSs).





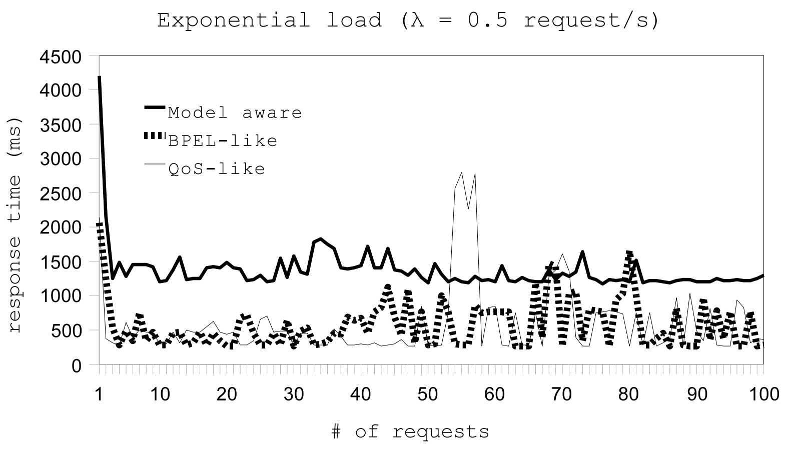


Figure 6.16: Response times for consecutive requests in the experiment BP-3.

Table 6.17: The experiments for the capability of the middleware: dynamic service selection.

Experiment	**Description**	**Setup - CWS Complexity**	**Setup - External WS parameters**
Experiment DSS-1	Dynamic distribution of load between 2 instances of WS (for 3 CWSs).	CWS1, CWS2, CWS3 (Fig. 6.2): $n = 1$, $n_{optional} = 0$, $m = 1$	$N = 1$, $i_{WS1} = 2$, $S_{WS1,1}$ = 100ms for 60 calls then 800ms, $A_{WS1,1}$=100%, $S_{WS1,2}$= 500ms for 60 calls then 100ms, $A_{WS1,2}$ = 100%.
Experiment DSS-2	Switching the distribution of load between 2 instances of WS (for 3 CWSs).	CWS1, CWS2, CWS3 (Fig. 6.2): $n = 1$, $n_{optional} = 0$, $m = 1$	$N = 1$, $i_{WS1} = 2$, $S_{WS1,1}$ = 300ms for 120 calls then 800ms, $A_{WS1,1}$ = 100%, $S_{WS1,2}$ = 100ms for 120 calls, then 500ms for next 20 calls, then 100ms, $A_{WS1,2}$ = 100%.
Experiment DSS-3	Distribution of load between 2 instances of WS and additional path (for 3 CWSs).	CWS1, CWS2 (Fig. 6.2): $n = 1$, $n_{optional} = 0$, $m = 1$, CWS3 (Fig. 6.5): $n = 2$, $n_{optional} = 0$, $m = 2$	$N = 2$, $i_{WS1} = 2$, $i_{WS2} = 1$, $S_{WS1,1}$ = 200 ms for 60 calls then 500 ms, $A_{WS1,1}$ = 100%, $S_{WS1,2}$ = 200 ms for 60 calls then 500 ms, $A_{WS1,2}$ = 100%, $S_{WS2,1}$ = 300ms, $A_{WS2,1}$ = 100%.
Experiment DSS-4	Distribution of load between 2 instances of WS and omitting optional (for 3 CWSs).	CWS1, CWS2 (Fig. 6.3): $n = 2$, $n_{optional} = 0$, $m = 1$, CWS3 (Fig. 6.4): $n = 2$, $n_{optional} = 1$, $m = 1$	$N = 1$, $i_{WS1} = 1$,$i_{WS2} = 1$, $S_{WS1,1}$ = 200ms $A_{WS1,1}$ = 100%, $S_{WS2,1}$ = 100ms for 60 calls then 800ms, $A_{WS2,1}$ = 100%.

Dynamic service selection

In the model-aware middleware it is possible to dynamically select the optimal instance of an external WS. The selection criteria are a state of all instances (status and the expected response time). Moreover the model-aware middleware detects also that instance is or may become overloaded. To react to this event there can be chosen another instance, or a call to this WS can be omitted, if it is optional. In Table 6.17 there are described settings of experiments that show this capability of the middleware.

In the experiment DSS-1 there are 3 CWSs, which interact with $WS1$ that has 2 instances. The service time of the first instance increases substantially and that of the second one decreases. The response times, as the average for all CWSs, for subsequent requests are shown in Figure 6.17. For the model aware and QoS-like type of execution at the beginning the first instance is selected, because it has lower response time. Then after 20 requests (there are 3 CWSs and the change in the service time of $WS1$ is after 60 calls) the second instance is chosen, for which the response time decreases, it can be observed after the 45th request. For the BPEL-like executions the first instance is always selected, so after the 45th request it has a worse performance. In the exponential type of load the BPEL-like response times vary greatly. This is because the service time of the first instance of $WS1$ is very high, so server becomes overloaded in case of small interrarival times, which are possible in the exponential load. So these delays are due to the low capabilities of server with $WS1$. Table 6.18 presents the overall averages for this experiment. The average for the model aware type of execution is almost twice as big as for the QoS-like, but still lower than for the BPEL-like. For these settings the average response times of the model aware type are constant for all types of load, but they increase for the other types of executions and the heavier or exponential load types.

In the experiment DSS-2, as for the previous one, there are 3 CWSs, which call the same external WS with 2 instances. In these settings the service time of the first instance increases after the first 120 calls (so after 40 calls from each CWS), and of the second increases also after 120 calls, but it is high only for the next several calls, then it decreases to the initial small value. The response times for all requests as averages of all CWSs is shown in Figure 6.18. For the model aware and QoS-like type of execution there are 4 relatively stable phases. The first one is before the 45th request when the second instance is called. The second one is until the 85th request in which the first instance is selected. For the last two phases again the second instance is used, at the beginning with higher service times (3rd phase) and after the 90th request with lower ones (4th phase). The phases are the most distinctive in the constant heavy load, and less for the light. For the exponential load there is an abrupt increase in the model-aware execution at the 65th request, so the switch to the second instance is done earlier. In the BPEL-like execution only the first instance is used so there are only 2 phases, and after the 40th request the response times increase. As for the previous experiment, also for this one in the exponential type of load the server with the first instance of $WS1$ becomes overloaded, and response times in the BPEL-like type vary substantially. The overall averages for all executions is in Table 6.19. They are the worst for the model aware type of execution, except for the exponential load, for which the highest average is in the BPEL-like type. In the model-aware type the averages are stable for all possible loads, for the other types they increase for the constant heavy load and the exponential one.

In the experiment DSS-3 there are 3 CWSs: the first two (CWS1 and CWS2) interact only with $WS1$, and the third one (CWS3) has two alternative paths and can either call $WS1$ or $WS2$. Service times for both instances of $WS1$ increase after 20 requests to CWSs, and are constant for $WS2$. The response times for consecutive requests for CWS3 are shown in Figure 6.19. In the model aware type of execution they are high, but quite stable for all calls. After the 40th request there is a change in selection of a path to the path with $WS2$, and the response times become slightly bigger. In the other types of executions after the 40th request the variability of response times increases. This is because paths to execute are chosen randomly, so also the path with $WS1$, which has worse performance, is used. In the exponential type of execution, due to high service times of the first instance of $WS1$, in the BPEL-like executions the server becomes overloaded. In Table 6.19 there are overall response times for all types of loads and executions. The overall average is higher for the model aware executions, but for all types of load values are similar. For the BPEL-like and QoS-like executions average response times grows with heavier load type and with the exponential one.

As in the previous experiment the experiment DSS-4 contains 3 CWSs. They all interact with $WS1$ and $WS2$, but in two of them (CWS1, CWS2) a call to $WS2$ is compulsory, whereas in the third CWS (CWS3) the call is optional. After 60 calls to $WS2$ its service time increases rapidly. The response times for CWS3 for different types of load are shown in Figure 6.20. For all types of loads response times for the model aware type of execution are constant, because after the 20th request the $WS2$ is considered as overloaded, and this call is omitted to avoid overloading. For the QoS-like and BPEL-like types this external WS is used in every execution. In the case of light load it does not affect response times, but for other types of loads it causes the server to respond very slowly. So the response times of this CWS either increase profoundly in the heavy constant type of load or vary significantly in the exponential load. In Table 6.21 there are overall averages for this CWS. For the light constant load the overall average is the highest in the model-aware execution. However, for the other types of load, since interactions with the overloaded server are avoided, the model-aware execution has the best performance. The difference is especially big for the heavy constant type of load.

Table 6.18: Average response times in the experiment DSS-1.

Execution type	**Constant light load**	**Constant heavy load**	**Exponential load**
Model aware	750	749	761
BPEL-like	831	867	1245
QoS-like	372	358	477

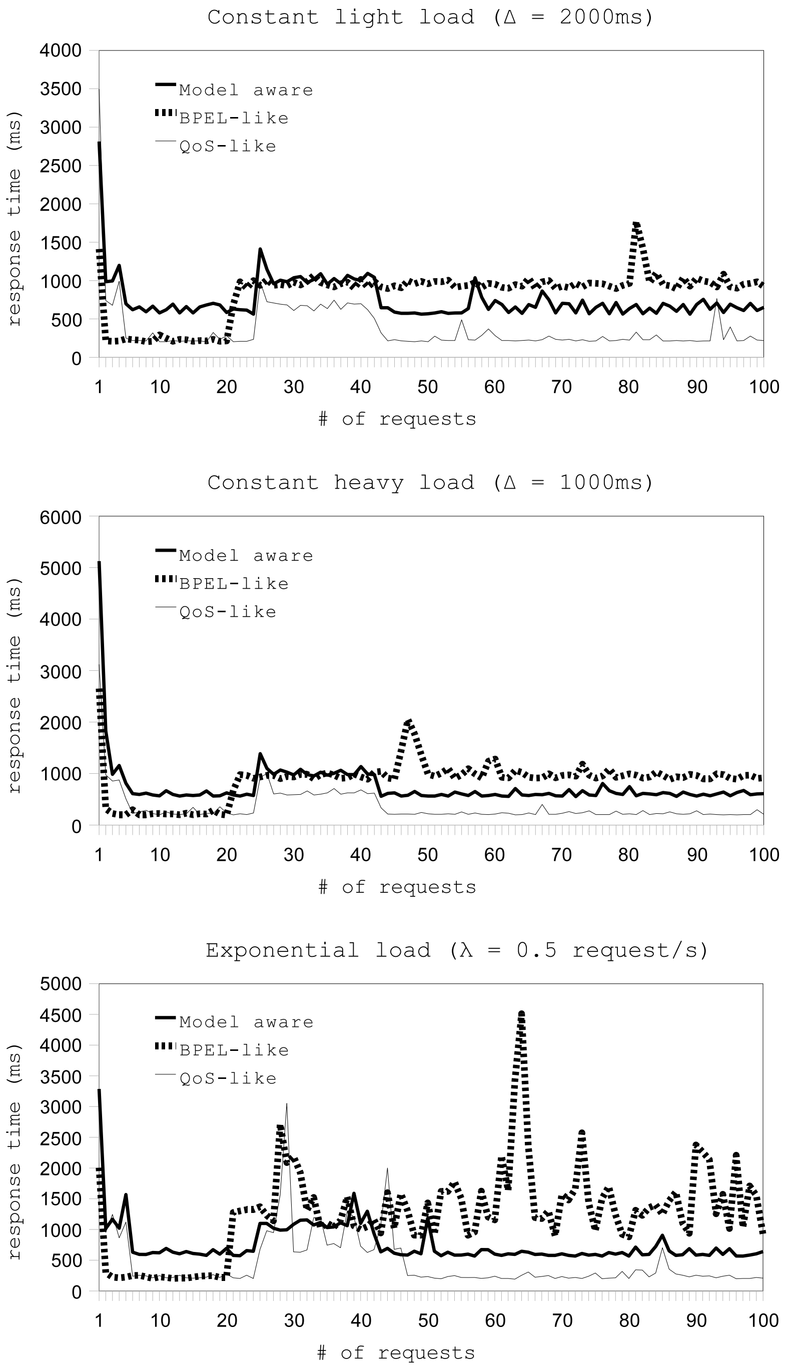


Figure 6.17: Response times for consecutive requests in the experiment DSS-1 (average for all CWSs).

Table 6.19: Average response times in the experiment DSS-2.

Execution type	Constant light load	Constant heavy load	Exponential load
Model aware	759	726	704
BPEL-like	609	624	957
QoS-like	327	337	392

Table 6.20: Average response times in the experiment DSS-3.

Execution type	Constant light load	Constant heavy load	Exponential load
Model aware	1493	1554	1520
BPEL-like	549	600	833
QoS-like	530	555	760

Table 6.21: Average response times in the experiment DSS-4.

Execution type	Constant light load	Constant heavy load	Exponential load
Model aware	1787	1847	1932
BPEL-like	1032	6653	2120
QoS-like	1078	6203	2195

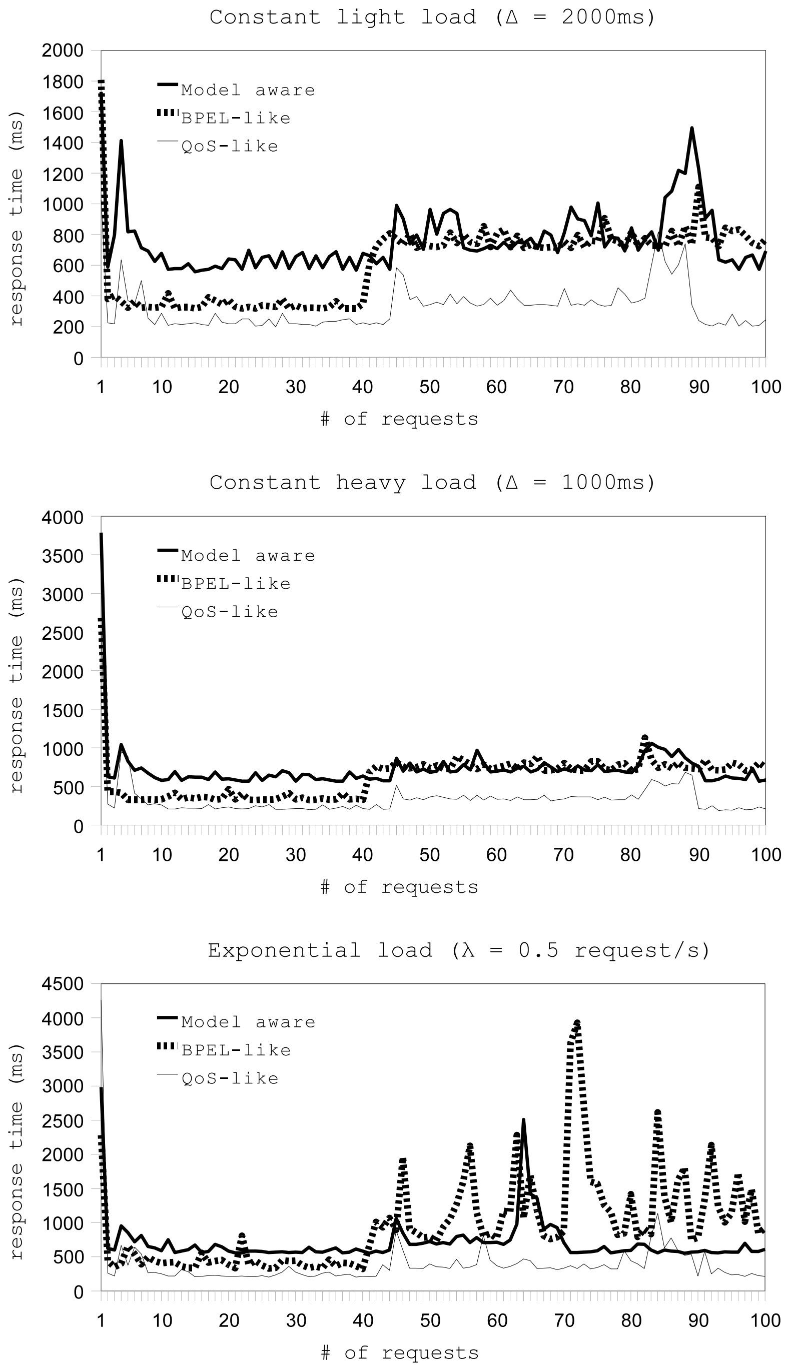
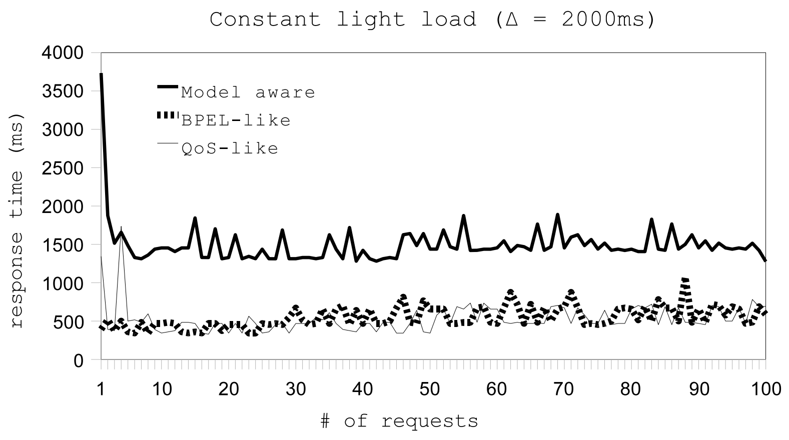
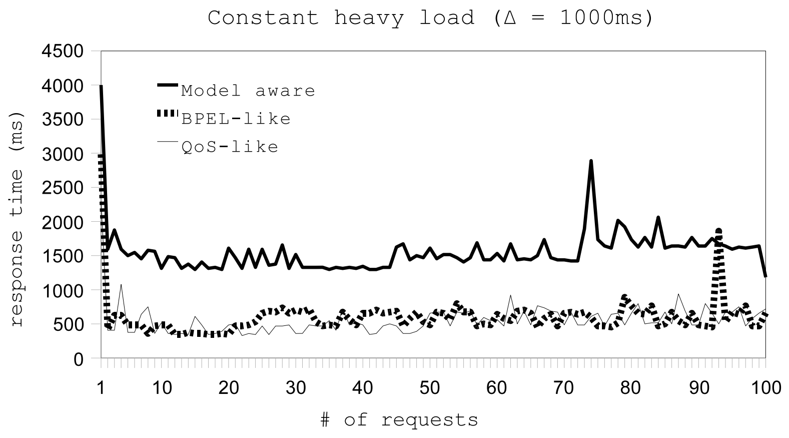


Figure 6.18: Response times for consecutive requests in the experiment DSS-2 (average for all CWSs).





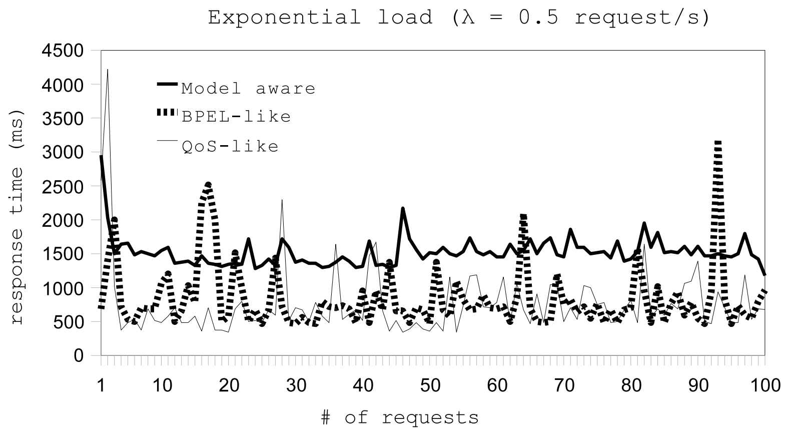
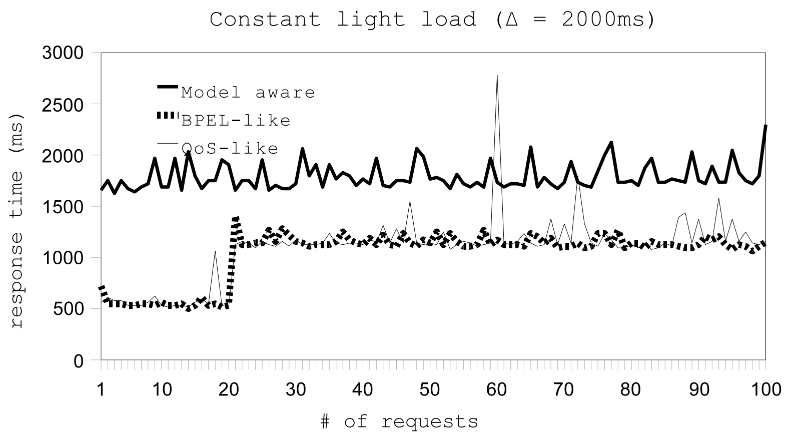
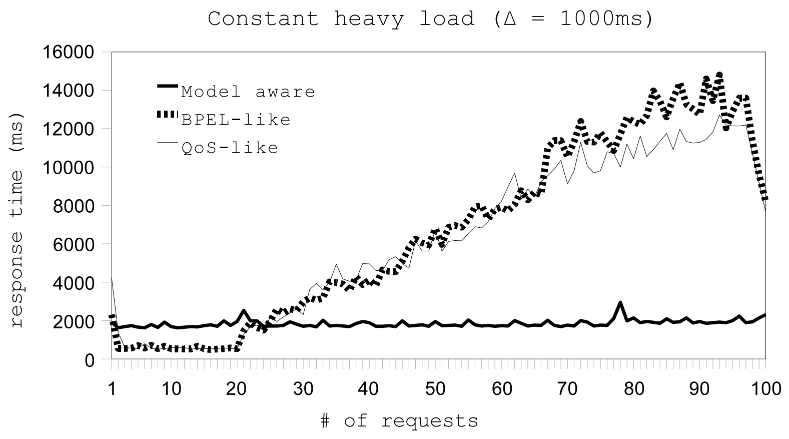


Figure 6.19: Response times for consecutive requests in the experiment DSS-3 (for CWS3).





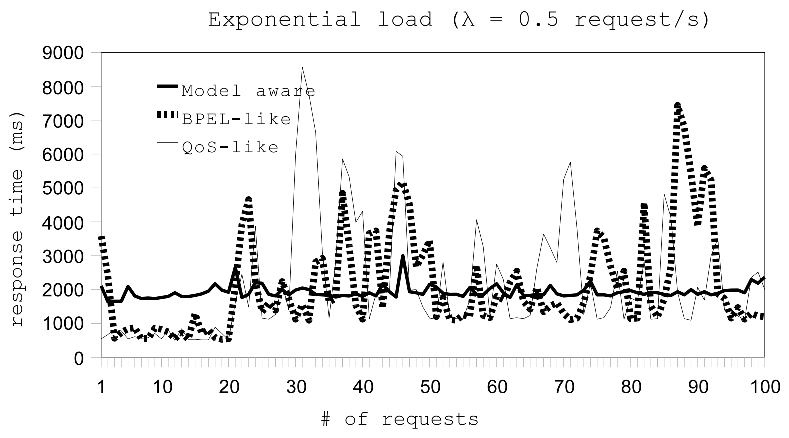


Figure 6.20: Response times for consecutive requests in the experiment DSS-4 (for CWS3).

Table 6.22: The general description of experiments for the complexity analysis.

Experiment	Description	Setup - general
Experiment BA-1	Increasing number of CWSs.	$N_{CWS} = 3$ or 4 or 5 , all CWSs (Fig. 6.2): $n = 1$, $n_{optional} = 0$, $m = 1$, $S_{WS1,1} =$ 200ms, $A_{WS1,1} = 100\%$.
Experiment BA-2	Increasing a number of calls and optional calls in CWS.	$N_{CWS} = 1$, CWS1 (similar to Fig. 6.4): $n = 4$ or 5 or 6, $n_{optional} = 4$ or 5 or 6, $m = 1$, $S_{WS1,1} = S_{WS2,1} = ... = S_{WS6,1} =$100ms, $A_{WS1,1} = A_{WS2,1} = ... = A_{WS6,1} =$100%.
Experiment BA-3	Increasing a number of alternatives (with a call to an external WS in each branch) in CWS.	$N_{CWS} = 1$, CWS1 (similar to Fig. 6.5): $n = 4$ or 5 or 6, $n_{optional} = 4$ or 5 or 6, $m = 4$ or 5 or 6, $S_{WS1,1} = S_{WS2,1} = ... = S_{WS6,1}$=200ms, $A_{WS1,1} = A_{WS2,1} = ... = A_{WS6,1} =$100%.

6.3.2 Results of experiments in analyzing complexity of CWSs

The experiments in this phase are performed to test the impact of increasing complexity of CWSs. They are done for two scenarios: base and reasoning. The base scenario is the BPEL-like type of execution from the previous section, and the reasoning is the model-aware type. All experiments are performed for $k = 100$ executions and 6 different load types:

- constant light load - with $\Delta_t = 2000$ ms,
- constant heavy load - with $\Delta_t = 1000$ ms,
- constant very heavy load - with $\Delta_t = 500$ ms,
- exponential light distribution of interarrival times with $\lambda = 0.5$ requests/s.
- exponential heavy distribution of interarrival times with $\lambda = 1$ requests/s.
- exponential very heavy distribution of interarrival times with $\lambda = 2$ requests/s.

The other settings for these experiments are presented in Table 6.22.

In the experiment BA-1 a number of CWSs in the middleware increases. The average response times for all CWS and load types are shown in Figure 6.21. Although in the reasoning case (the model aware executions) there are differences in response for increasing number of CWSs, they are not significant, and they do not exceed 100 ms. For the base case (the BPEL-like executions) this trend is not present at all. These results demonstrate however that the reasoning affects the performance of CWSs, since almost for each case they are two times slower than the base cases.

Nevertheless the advantage of the reasoning is that results are stable even for heavier loads. In the base case response times are a little bit higher for more extensive loads.

The next experiment takes into account the increasing number of calls to external WSs, and these calls are optional. Figure 6.22 presents the average response times for these settings. For almost all types of loads the average response time increases with each added call to an external WS. This is due to additional reasoning caused by each interaction, because it makes the construction of a timed OE-graph (the base to select an optimal path to execute) more complex. For the base case (the BPEL-like executions) there cannot be observed any significant correlation in response times and number of calls to external WSs. These results also show how expensive is reasoning in terms of performance, and this cost increases with each extra interaction. As for the previous experiment, also here the performance of CWSs in the reasoning case is stable for all types of loads, and for the base case it increases with heavier types of load.

In the experiment BA-3 the increasing number of alternatives in a CWS is considered. The response times for this experiment are in Figure 6.23. In the reasoning case the response times are proportional to the number of alternatives. This relationship is caused by increasing complexity of construction of a timed OE-graph. So if there are more alternatives in the CPN model of CWSs the additional paths must be first created and then considered. Based on the above results, this element of CWSs is the most expensive, since differences are very big. It means that adding new alternatives to the definition of a CWS causes the biggest loss in their performance. In the base case all response times are quite stable, since the choice of the path is random. The increasing number of alternatives has no significant effect on the response times in both scenarios in lighter or heavier loads, and they are very similar. Only for the very heavy exponential load the performance is slightly worse, which is caused by the temporary overloading of servers.

The results of the above experiments demonstrate that the additional call to an external WS and an additional path in a CWS increase substantially response times. This is due the increasing size of timed occurrence graph which is constructed for the CWS. Besides this the results also show that the difference between the model aware execution and the execution without reasoning is significant. Therefore, in order to make the reasoning beneficial, it is necessary that in a CWS better executions are possible. For example according to response times in Figure 6.23 the optimal path should be 5 to 10 seconds faster then the others, since this is the cost of reasoning in this case. If the possible gain is less than that the model aware middleware is not required. Nevertheless the Phase 2 of the evaluation showed that in some scenarios the model-aware execution improves the reliability of CWSs. So if the priority is to make CWSs reliable, then the cost of reasoning in terms of response times is acceptable.

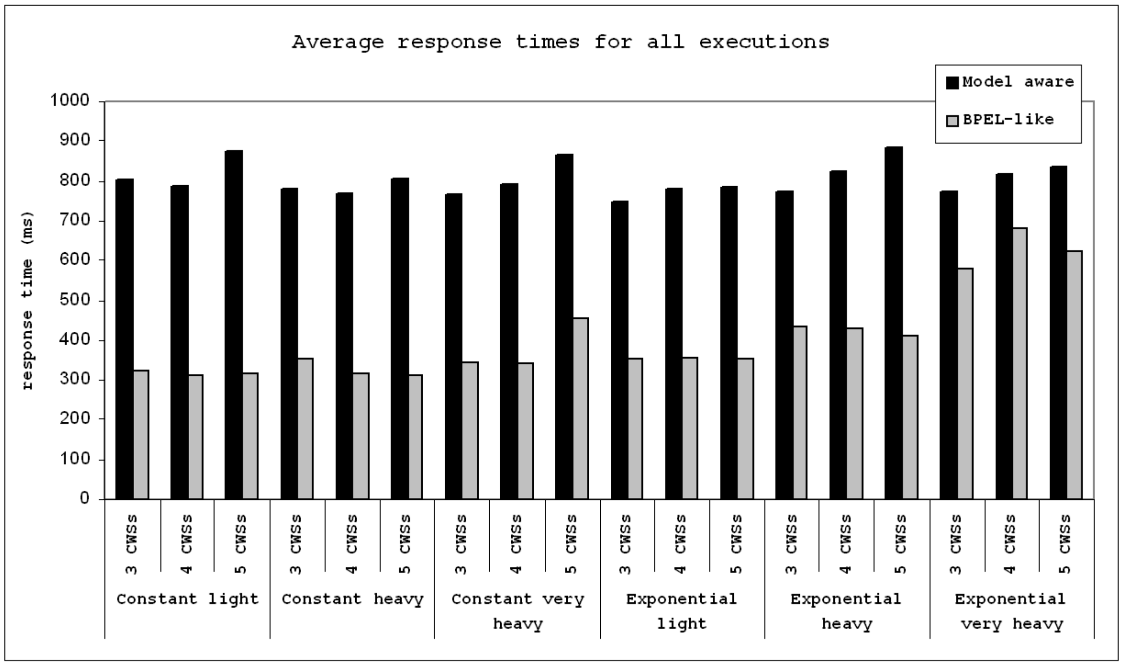


Figure 6.21: The average response times in the experiment BA-1.

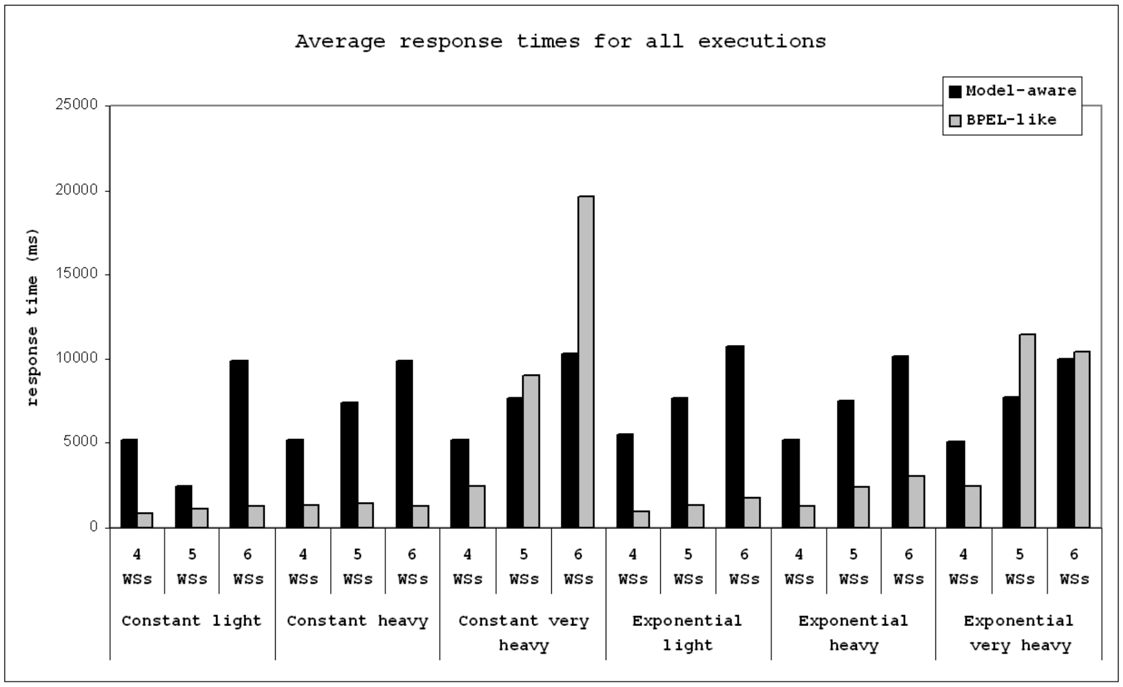


Figure 6.22: The average response times in the experiment BA-2.

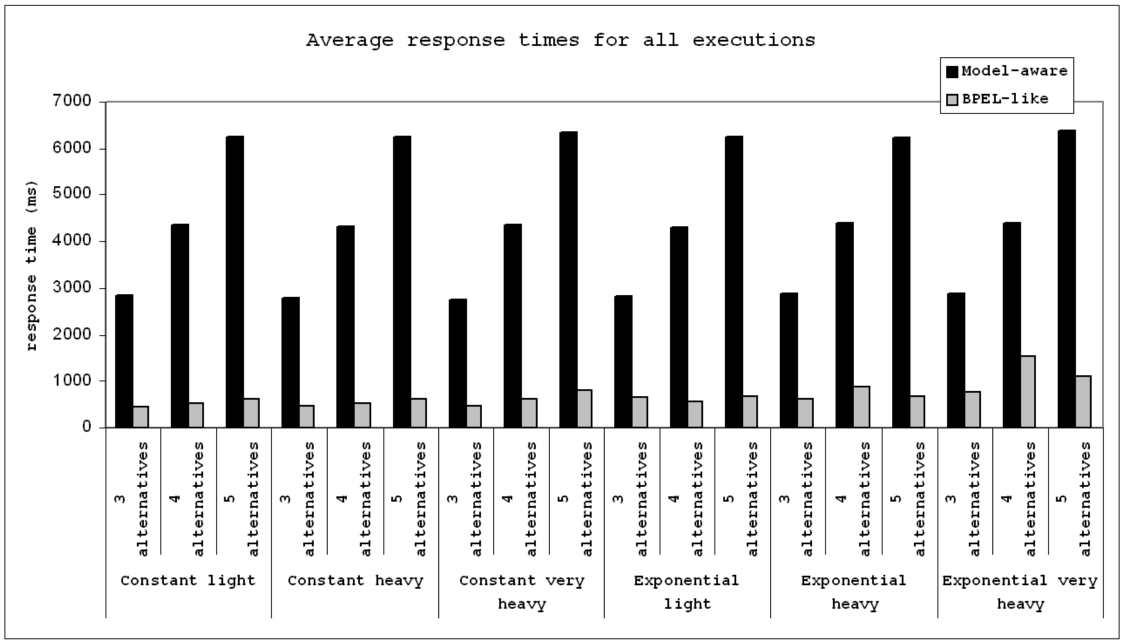


Figure 6.23: The average response times in the experiment BA-3.

6.4 Summary

The first phase of the evaluation was testing the middleware. In this phase the detailed functionalities of the middleware were demonstrated with the test cases performed for each component. The External WS Registry is able to store and manage external WSs and the Composite WS Registry the CPN models of CWSs. With the Execution Engine it is possible to execute CWSs and to generate events during executions. These events are then successfully transformed to facts and are basis to reason in the Reasoning Mechanism. The CWS Agents and System Agent responds to events from the Execution Engine and the Reasoning Mechanism, and can monitor and optimize execution of CWSs. The above represent the most important processes in the middleware. Thus, in this phase it was shown that the implementation of the model-aware middleware based on the CPN model is feasible.

The second phase of the evaluation was to use the middleware in optimizing the execution of CWSs. It was demonstrated that the middleware can efficiently react to the sudden unavailability of external WSs, and can perform much better than a static approach like BPEL. With the middleware it is possible also to select the optimal path and to change it dynamically, and in that way to provide more stable performance. Finally due to the ability to detect overloading it is possible to alter an execution or to omit optional calls to external WSs. In this way improvements in performance of CWSs can be significant.

The model-aware middleware is the most efficient in optimization over other approaches in two scenarios. The first scenario is to detect alternative paths in the model of a CWS, not only the better instances of external WSs to use. So the result of executing these alternative paths can be anticipated, whether it can lead to the successful execution of a CWS or what is the possible impact on the performance. The second scenario is to avoid overloading on used instances of external WSs. The middleware detects such situations, and then alters executions either by selecting another instance or path, or by omitting calls.

In the third phase of the evaluation an analysis was performed as to how the complexity of CWSs influences their performance. It was shown that the most important factor is the size or complexity of timed-OE graphs constructed for CWSs. The bigger they are, the more time reasoning requires and the slower CWSs are. The factors that make the graphs big are: the number of used external WSs and the number of possible alternatives to execute a CWS. Thus, improvements in the performance and the reliability in executing CWSs are possible, but can result in additional resources necessary to reason.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The Service Oriented Architecture (SOA) is an approach to build distributed applications in which services are computational elements. They can be published, discovered or consumed in a platform and organization independent manner. The most widely used technology to implement this paradigm is the Web Services technology. This is mainly due to a standardization process which resulted in standards like SOAP, WSDL, UDDI or HTTP, and a wide range of available tools (e.g. Axis). Besides the interoperability the SOA also supports composeability, which means that services can be easily aggregated into Composite Web Services (CWSs).

However, the emerging languages (e.g. BPEL) that allow implementation and execution of CWSs are limited in the monitoring and optimizing capabilities. One of the main reasons for this limitation is that BPEL or other languages do not give insight into the model of the composition. This lack of knowledge makes the truly dynamic behavior of CWSs very hard to achieve. In turn the performance of CWSs cannot be improved by analysis of their models and selecting the optimal way to execute them.

To overcome the above problems this thesis presents the model-aware approach to execute CWSs. In this approach the Coloured Petri Nets (CPN) language is the basis to model, monitor and reason about CWSs. In this way each execution of a CWS is supported with the knowledge of its model, and this enables efficient monitoring of states of execution. The model-aware approach also means that future states of the execution can be planned. Using the CPN as an unambiguous specification of a CWS can then improve its performance and reliability.

In the thesis the two aspects of the model-aware execution were explored: the theoretical one and the implementation. From the theoretical point of view, first it was presented how to model CWSs, so how to represent in the CPN interactions with WSs and other operations. Then the aspects of analysing and reducing the complexity of occurrence graphs for CWSs were considered. Finally it was proposed how to execute and plan the execution of CWSs. As for the implementation the architecture of the model-aware middleware was given. It deals with the issues of storing information about instances of external WSs and the CPN models of CWSs. Additionally the middleware enbles the execution of those models, in which interactions with other WSs are embedded. Based on the data gathered during executions, the rule engine reasons about states of WSs and possibilities to distribute load. With this knowledge each execution of a CWS is planned and monitored with timed occurrence graphs.

The evaluation of the model-aware middleware for CWSs demonstrates its ability to improve the performance of CWSs. The improvements are the most significant in cases when there are alternative paths of execution in the definition of a CWS. It is the feature that makes this middleware different than other optimization approaches, like the ones based on the QoS metric. Also, because it is possible to predict future loads on instances of WSs, the middleware can detect and avoid overloading those instances. Of course managing models, planning and reasoning require resources and processing power. It was shown that increasing complexity of the definition of a CWS influences substantially its performance.

Based on the above, the most valuable contributions of the thesis are:

- validating the approach to composing WSs using a formal method, like the CPN,
- proving the feasibility of the middleware implementation based on the CPN language, which can execute the CPN models of CWSs,
- the middleware for CWSs that plans and monitors states of their executions,
- analysis of the advantages and costs of using models of CWSs at runtime.

7.2 Application areas

The basic idea that underlies the model aware execution of CWSs is to improve their performance by analyze and exploring possibilities that are present in their definition. This goal is achieved with representing a CWS in the CPN formalism and then, based on the model, with reasoning about the optimal way to execute it. However, since the reasoning requires resources, this solution is not suitable in all cases. Also the aim of introducing the model awareness is not to replaces BPEL, rather to enhance it, because BPEL is still the most common way to implement CWSs.

The model aware execution of CWSs is the most efficient in CWSs, which are complex or which interact with external WSs that have changing performance. Therefore the reasoning is not necessary, and it only introduces the overhead, in CWSs which have one path to execute, and which cooperate with stable external WSs. In such cases it is enough to use the execution which is only based on the selection of the best instance of a WS. For more complex and long running CWSs the cost of model awareness is more beneficial. Especially if during the execution of a CWS there is the choice between two paths that can run for a long time, then the knowledge whether

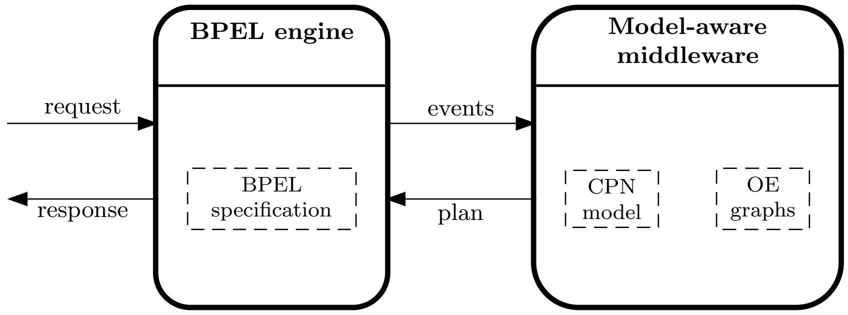


Figure 7.1: The abstract architecture of the BPEL engine with the model-aware middleware

all external WSs in those paths are available or which ones are better is very valuable. With the method presented in this thesis, it is possible to guide complex workflows during their execution.

The model-aware middleware aims to enhance the already existing language to implement CWSs, which is BPEL. There are already works, which investigates how to translate the process in BPEL into Petri Nets formalism (for example in: [21], [30], [45]). Therefore this thesis focuses more on the runtime aspects of using the CPN. In this way it is assumed that the translation from BPEL int the CPN model is feasible, and is not considered. So CWSs, before loading them to the middleware, are transformed from BPEL. After this, during the execution, only a model of a CWS is used.

Another possibility of using the model-aware middleware is to allow the execution of CWSs in one of BPEL engines (e.g. ActiveBEPL [1]) and monitor it with the model. Figure 7.1 presents this approach. The model-aware middleware is transparent for requestors, and works as a part of the BPEL engine. All the planning and reasoning still takes place in the middleware, however it uses events thrown by the engine. In this architecture it is required to customize the engine in order to allow efficient monitoring with the CPN model by analysis of events thrown during execution. Additionaly the BPEL engine accept plans generated by the middleware. So the execution can be changed to select the optimal instances of external WSs and the optimal paths. The current engines does not have such abilities, so this is one of the possible future works.

7.3 Future work

From the theoretical point of view the first possible future work is to formally define a transition that represents a call to an external WS. In the approach presented in the thesis this kind of transition is no different than others. However, it has its specific behavior, independent from the external WS it interacts with. So it is important to formally introduce this new type of transition into the CPN model, and specify how it can be enabled and what results it produces. In this way it can be determined how it influences the construction of ordinary or timed occurrence graphs.

The other theoretical improvement is to make not only one execution of a CWS adaptable, but the whole model. So if there are paths in the model that are never used, it can be beneficial to remove them from the CPN model of a CWS. There are, however, important issues to consider like: what are the conditions to make such a change, should it be temporary or permanent, if it is temporary how to store information about the unused paths. All of the above relate to the problem of runtime changes of the model, and requires work to make them general.

Finally in the theoretical approach presented in the thesis all executions of CWSs are of one type. So another problem to consider is how to execute different types of requests. For example there are requests for which response time has the highest priority, but others can wait longer, but instead require that all external WSs are called. The issue here is how to include such information in a model of a CWS, and then choose the appropriate type of the execution.

For the implementation the most important improvement is to allow dynamic discovery of services. For example to search for external WSs during execution, if the performance of existing ones decreases. In this way even more efficient adaptations can be achieved. But there are certain difficulties with this capability. One of them is how the search should be performed or how to make new instances of external WSs interchangeable with existing ones. There can be adapters, but then the issue is how to use them during the execution of the CPN models.

The other possible work is how to improve the performance of planning. It was shown how each additional path to consider in the timed OE graph influences the response times of CWSs. This is because during the construction of graphs very slow simulators are used. So the possible work is to implement a new, more efficient engine to reason about CWSs and to plan their execution.

In the current implementation there are only considered omission failures returned by an external WS. The important problem is then how to include other types of failures possible in distributed systems [11]. This would require theoretical work as well, since in the definition of a transition there are defined only "no response" failures. Another question is how these two approaches, theoretical and implementation, can be combined to provide a general tolerance for failures in the definition of a CWS. This is a very important aspect of CWSs, since services are components remotely deployed and their failures are independent from CWSs and are likely to occur.

References

[1] ActiveEndpoints. *ActiveBPEL - open source BPEL engine.* http://www.active-endpoints.com/active-bpel-engine-overview.htm.

[2] Apache. *Axis v 1.4.* http://ws.apache.org/axis/.

[3] The Apache Software Foundation. *Apache Tomcat 5.5.* http://tomcat.apache.org/.

[4] Luciano Baresi, Elisabetta Di Nitto, Carlo Ghezzi, and Sam Guinea. A framework for the deployment of adaptable web service compositions. *Service Oriented Computing and Applications*, 1:75 – 91, 2007.

[5] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Analysis and management of Web service protocols. In *Conceptual Modeling - ER 2004. 23rd International Conference on Conceptual Modeling. Proceedings, 8-12 Nov. 2004*, pages 524–41, Shanghai, China, 2004. Springer-Verlag.

[6] Boualem Benatallah, Marlon Dumas, Quan Z. Sheng, and Anne H.H. Ngu. Declarative composition and peer-to-peer provisioning of dynamic Web services. In *Proceedings 18th International Conference on Data Engineering*, pages 297 – 308, San Jose, CA, USA, 2002.

[7] Boualem Benatallah, Quan Z. Sheng, and Marlon Dumas. The Self-Serv environment for web services composition. *IEEE Internet Computing*, 7:40 – 8, 2003.

[8] Fabio Casati and Ming-Chien Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26:143 – 63, 2001.

[9] Girish Chafle, Koustuv Dasgupta, Arun Kumar, Sumit Mittal, and Biplav Srivastava. Adaptation in Web Service Composition and Execution. In *IEEE International Conference on Web Services (ICWS '06)*, pages 549–557, 2006.

[10] Girish Chafle, Prashant Doshi, John Harney, Sumit Mittal, and Biplav Srivastava. Improved Adaptation of Web Service Compositions Using Value of Changed Information. In *IEEE International Conference on Web Services (ICWS'07)*, pages 784–791, Salt Lake City, UT, USA, 2007.

[11] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 02 1991.

[12] Dominik Dahlem, Lotte Nickel, Sacha Jan, Bartosz Biskupski, Jim Dowling, and Rene Meier. Towards Improving the Availability of Service Compositions. In *Digital EcoSystems and Technologies Conference, 2007. DEST '07. Inaugural IEEE-IES*, pages 67–70, 2007.

[13] Valeria De Antonellis, Michele Melchiori, Luca De Santis, Masima Mecella, Enrico Mussi, Barbara Pernici, and Pierlugi Plebani. A layered architecture for flexible Web service invocation. *Software - Practice and Experience*, 36:191 – 223, 2006.

[14] Marlon Dumas, Will M.P. van der Aalst, and Arthur H.M. ter Hofstede. *Process-Aware Information Systems.* Wiley, 2005.

[15] Abdelkarim Erradi, Piyush Maheshwari, and Vladimir Tosic. Policy-driven middleware for self-adaptation of Web services compositions. In *Middleware 2006. ACM/IFIP/USENIX 7th International Middleware Conference*, pages 62 – 80, Melbourne, Vic., Australia, 2006.

[16] Andrea Ferrara. Web services: a process algebra approach. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM Press.

[17] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting BPEL web services. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM Press.

[18] Calude Girault and Rudiger Valk. *Petri Nets for systems engineering*. Springer-Verlag Berlin, Heidelberg, New York, 2003.

[19] Henry Habrias and Marc Frappier. *Software Specification Methods*. International Scientific and Technical Encyclopedia, 2006.

[20] Rachid Hamadi and Boualem Benatallah. A Petri net-based model for web service composition. In *ADC '03: Proceedings of the fourteenth Australasian database conference*, pages 191–200, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[21] Sebastain Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to Petri nets. In *Business Process Management. 3rd International Conference, BPM 2005. Proceedings (Lecture Notes in Computer Science Vol. 3649)*, pages 220 – 235, Nancy, France, 2005.

[22] IBM. *Definition of Web Service*. http://www-128.ibm.com/ developerworks/ webservices/ newto/websvc.html.

[23] Kurt Jensen. *Coloured Petri nets :basic concepts, analysis methods, and practical use, v.1*, volume 1. Springer-Verlag, Berlin, 1992.

[24] Kurt Jensen. *Coloured Petri nets :basic concepts, analysis methods, and practical use, v.2*, volume 2. Springer-Verlag, Berlin, 1992.

[25] Kurt Jensen. An introduction to the theoretical aspects of coloured Petri nets. *Lecture Notes in Computer Science; A Decade of Concurrency*, 803:230–272, 1993.

[26] Kurt Jensen. An introduction to the practical use of coloured Petri nets. *Lecture on Petri Nets II: Applications. Advances in Petri Nets*, 2:237 – 92, 1998.

[27] Massimo Mecella, Francesco Parisi Presicce, and Barbara Pernici. Modeling e-service orchestration through Petri nets. In *Technologies for E-Services. Third International Workshop, TES 2002. Proceedings (Lecture Notes in Computer Science vol. 2444)*, pages 38 – 47, Hong Kong, China, 2002.

[28] Microsoft, BEA, IBM. *Business process execution language BPEL v.1.1.* http://www-128.ibm.com/ developerworks/ library/specification/ws-bpel/.

[29] OASIS. *Universal Description Discovery And Integration (UDDI).* http://uddi.org/pubs/.

[30] Chun Ouyang, Eric Verbeek, Will M.P. van der Aalst, Stephen Breutel, Marlon Dumas, and Arthur H.M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162 – 98, 2007.

[31] M. Parashar and S. Hariri. Autonomic computing: an overview. In *Unconventional Programming Paradigms. International Workshop UPP 2004. Revised Selected and Invited Papers (Lecture Notes in Computer Science Vol. 3566)*, pages 257 – 269, Le Mont Saint Michel, France, 2004.

[32] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 10/ 2003.

[33] Sandia National Laboratories. *Jess, the Rule Engine for the Java Platform.* http://herzberg.ca.sandia.gov/ jess/ docs/70/ index.html.

[34] Karsten Schmidt. LoLA - A Low Level Analyser. In *Application and Theory of Petri Nets 2000: 21st International Conference, ICATPN 2000, Proceedings (Lecture Notes in Computer Science vol.1825)*, 2000.

[35] Zhangxi Tan, Chuang Lin, Hao Yin, Ye Hong, and Guangxi Zhu. Approximate performance analysis of Web services flow using stochastic Petri net. In *Grid and Cooperative Computing - GCC 2004. Third International Conference Proceedings.*, pages 193 – 200, Wuhan, China, 2004.

[36] University of Aarhus, Denmark. *BRITNeY Suite, Basic Real-time Interactive Tool for Net-based animation.* http://wiki.daimi.au.dk/ britney/.

[37] Will M.P. Van Der Aalst, A.H.M. Ter Hofstede, Bartek Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5 – 51, 2003/07.

[38] Will M.P. Van Der Aalst and Kees M. Van Hee. *Workflow management :models, methods, and systems.* MIT Press, Cambridge, Mass., 2004.

[39] Kunal Verma, Karthik Gomadam, Amit Sheth, John Miller, and Zixin Wu. The METEOR-S approach for configuring and executing dynamic Web Processes. Technical report, LSDIS Lab, University of Georgia, Athens, Georgia, 2005.

[40] W3C. *Simple object access protocol (SOAP) 1.2.* http://www.w3.org/ TR/ soap12-part1/.

[41] W3C. *Web Services Description Language (WSDL), v. 1.1.* http://www.w3.org/ TR/ wsdl.

[42] W3C. *Web Services Policy 1.2 - Framework (WS-Policy).* http://www.w3.org/ Submission/ WS-Policy/.

[43] W3C. *XML Schema - W3C Recommendation.* http://www.w3.org/ TR/ xmlschema-2.

[44] S. Weerawarana. *Web Services Platform Architecture SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More.* Upper Saddle River, NJ: Prentice Hall PTR,, 2005.

[45] YanPing Yang, QingPing Tan, and Yong Xiao. Verifying web services composition based on hierarchical colored petri nets. In *IHIS '05: Proceedings of the first international workshop on Interoperability of heterogeneous information systems*, pages 47–54, New York, NY, USA, 2005. ACM Press.

[46] Tao Yu and Kwei-Jay Lin. Adaptive algorithms for finding replacement services in autonomic distributed business processes. In *ISADS 2005. 2005 International Symposium on Autonomous Decentralized Systems*, pages 427 – 34, Chengdu, Jiuzhaigou, China, 2005.

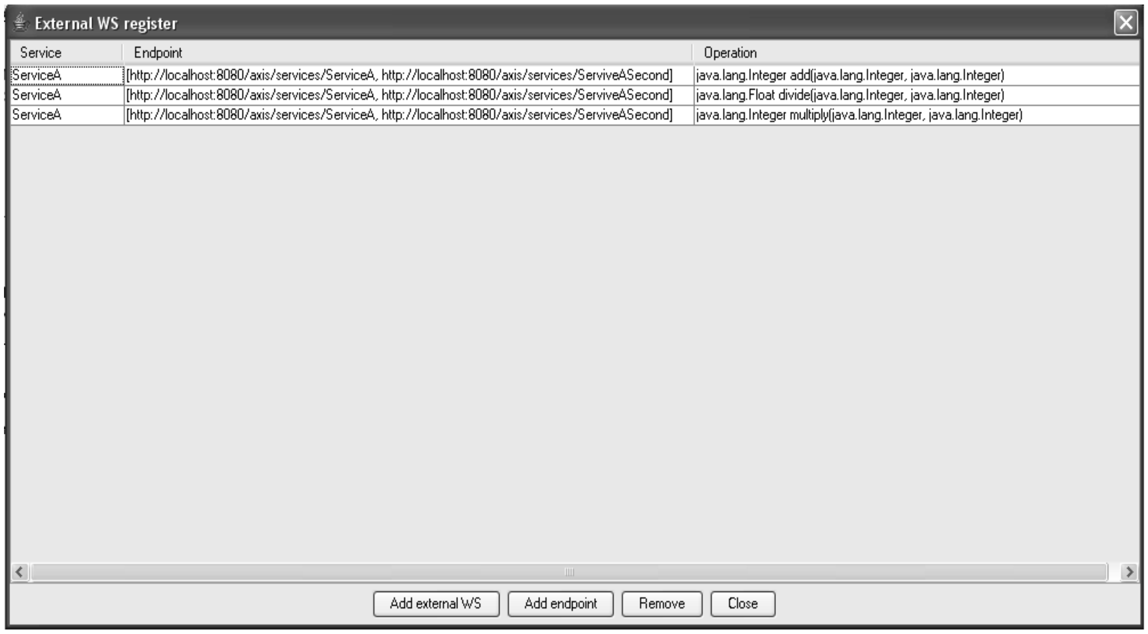
[47] Jia Zhang, Carl K. Chang, Chung Jen-Yao, and Seong W. Kim. WS-Net: a Petri-net based specification model for Web services. In *Proceedings. IEEE International Conference on Web Services (ICWS'04)*, pages 420 – 427, San Diego, CA, USA, 2004.

[48] Liang-Jie Zhang. Challenges and opportunities for Web services research. *International Journal of Web Services Research*, 1(1):vii–xiii, 2004/01 2004.

[49] Duhang Zhong and Zhichang Qi. A petri net based approach for reliability prediction of Web services. In *OTM Workshop 2006 (Lecture Notes in Computer Science vol. 4277)*, pages 116 – 125, Montpellier, France, 2006.

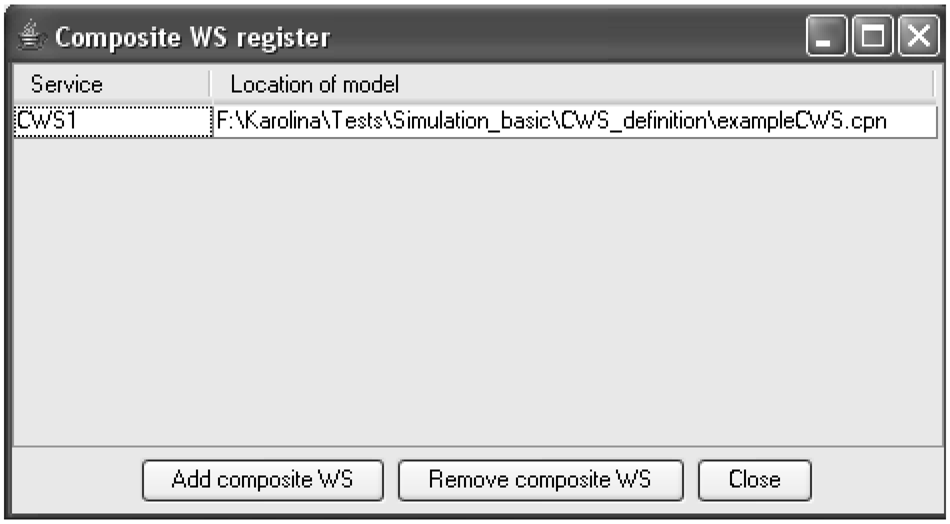
Appendix A

The model-aware middleware for CWSs



Service	Endpoint	Operation
ServiceA	[http://localhost:8080/axis/services/ServiceA, http://localhost:8080/axis/services/ServiveASecond]	java.lang.Integer add(java.lang.Integer, java.lang.Integer)
ServiceA	[http://localhost:8080/axis/services/ServiceA, http://localhost:8080/axis/services/ServiveASecond]	java.lang.Float divide(java.lang.Integer, java.lang.Integer)
ServiceA	[http://localhost:8080/axis/services/ServiceA, http://localhost:8080/axis/services/ServiveASecond]	java.lang.Integer multiply(java.lang.Integer, java.lang.Integer)

Figure A.1: The External WS Registry window.



Service	Location of model
CWS1	F:\Karolina\Tests\Simulation_basic\CWS_definition\exampleCWS.cpn

Figure A.2: The CWS Registry window.

Appendix B

Rules in the Reasoning Mechanism

B.1 Reasoning about state of endpoints of external WSs

```
(deftemplate Transition (declare (from-class Transition)))
(deftemplate TransitionCurrent (declare (from-class TransitionCurrent)))
(deftemplate ExternalWSStatus (declare (from-class ExternalWSStatus)))
(deftemplate ExternalWSStatusCurrent
(declare (from-class ExternalWSStatusCurrent)))
(deftemplate OperationWSTime (declare (from-class OperationWSTime)))
(deftemplate OperationWSTimeCurrent
(declare (from-class OperationWSTimeCurrent)))

(deftemplate Notifier (declare (from-class Notifier)))

(reset)

; prepare notifier in a working memory
(bind ?notifier (new Notifier))
(add ?notifier)

;rules for maintaining states of WSs and delays of transitions

;delay for transitions
(defrule transition_new "Creating a new current transition data"
    ?fact <- (Transition (cws ?cws) (transitionName ?transition)
     (timeAvg ?avg) (timeLast ?last))
    (not (TransitionCurrent (cws ?c & :(= ?c ?cws))
     (transitionName ?t & :(= ?t ?transition))))
=>
    (add (new TransitionCurrent ?cws ?transition ?avg ?last))
    (?notifier changeTransition ?cws ?transition ?avg)
    (retract ?fact)
)

(defrule transition_changed_zero "Changing time for transition - zero avg"
    ?fact <- (Transition (cws ?cws) (transitionName ?transition)
     (timeAvg ?avg) (timeLast ?last))
    ?current <- (TransitionCurrent (cws ?cws) (transitionName ?transition)
     (timeAvg ?t & :(<> ?t ?avg) & :(= ?t 0)))
=>
    (modify ?current (timeAvg ?avg))
    (?notifier changeTransition ?cws ?transition ?avg)
    (retract ?fact)
)

(defrule transition_changed_non_zero "Changing time for transition-non zero avg"
    ?fact <- (Transition (cws ?cws) (transitionName ?transition)
     (timeAvg ?avg) (timeLast ?last) )
    ?current <- (TransitionCurrent (cws ?cws) (transitionName ?transition)
     (timeAvg ?t & :(<> ?t 0) & :(> (/ (abs(- ?avg ?t)) ?t) 0.1)))
=>
    (modify ?current (timeAvg ?avg))
    (?notifier changeTransition ?cws ?transition ?avg)
    (retract ?fact)
    (?notifier info "Changing time for transition - non zero avg" )
)

(defrule remove_transition_times "Removing unused transition times "
(declare (salience -100))
    ?fact <- (Transition)
=>
    (retract ?fact)
)

; maintaining status of external WSs
(defrule status_WS_new_no_endpoint "Creating a new current status"
    ?fact <-(ExternalWSStatus (serviceName ?serviceName) (endpoint ?endpoint)
     (status ?status))
    (not (ExternalWSStatusCurrent (serviceName ?serviceName)))
    (not (ExternalWSStatusCurrent (serviceName ?n & :(= ?n ?serviceName))
     (endpoint ?e & :(= ?e ?endpoint))))
=>
    (add (new ExternalWSStatusCurrent ?serviceName ?endpoint ?status ))
    (add (new ExternalWSStatusCurrent ?serviceName 0 2))
    (?notifier changeStatusWS  ?serviceName ?endpoint ?status)
    (retract ?fact)
)

(defrule status_WS_new_endpoint "Creating a status for the next endpoint"
    ?fact <-(ExternalWSStatus (serviceName ?serviceName) (endpoint ?endpoint)
     (status ?status))
    (not (ExternalWSStatusCurrent (serviceName ?n & :(= ?n ?serviceName))
     (endpoint ?e & :(= ?e ?endpoint))))
=>
    (add (new ExternalWSStatusCurrent ?serviceName ?endpoint ?status ) )
    (?notifier changeStatusWS  ?serviceName ?endpoint ?status)
    (retract ?fact)
)

(defrule status_WS_changed_nw "Changing a current status to not working"
(declare (salience 100))
    ?fact <- (ExternalWSStatus (serviceName ?svcName)(endpoint ?endpoint)
     (status ?status & :(= ?status 2)))
    ?current <- (ExternalWSStatusCurrent (serviceName ?svcName)
     (endpoint ?endpoint) (status ?st & :(<> ?st ?status)))
=>
    (modify ?current (status ?status))
    (?notifier changeStatusWS ?svcName ?endpoint ?status)
    (retract ?fact)
)

(defrule status_WS_changed_ol "Changing a current status to overloaded"
(declare (salience 100))
    ?currentStatus <- (ExternalWSStatusCurrent (serviceName ?svcName)
     (endpoint ?endpoint) (status ?status & :(= ?status 0)))
    ?currentTime <- (OperationWSTimeCurrent (serviceName ?svcName)
     (operationName ?opName) (endpoint ?endpoint) (normalRT ?nrt)
     (interTime ?it))
    ?fact <- (OperationWSTime (serviceName ?svcName) (operationName ?opName)
     (endpoint ?endpoint) (trend ?t &:(= ?t 1))
     (avgTime ?avg &:(> ?avg (* ?nrt 1.5)))
     (lastTime ?lt & :(> ?lt (* ?nrt 3))))
=>
    (modify ?currentStatus (status 1) (lastOverloadTime
     (call System currentTimeMillis)))
    (modify ?currentTime (interTimeOverload ?it))
    (?notifier changeStatusWS ?svcName ?endpoint 1)
)

(defrule status_WS_changed_no "Changing a current status to normal"
(declare (salience 100))
    ?currentStatus <- (ExternalWSStatusCurrent (serviceName ?svcName)
     (endpoint ?endpoint) (status ?status & :(= ?status 1))
      (lastOverloadTime ?time & :
       (> (-(call System currentTimeMillis) ?time) 5000)))
    ?c <- (accumulate (bind ?list (new java.util.ArrayList))
         (?list add ?time)
         ?list
         ?time <- (OperationWSTimeCurrent (serviceName ?svcName)
         (endpoint ?endpoint))
        )
=>
    (modify ?currentStatus (status 0) (lastOverloadTime -1))
    (?notifier changeStatusWS ?svcName ?endpoint ?status)
    (?notifier changeTimeWSAllOperations(?list))
)

(defrule remove_status_unused "Removing unused statuses"
(declare (salience -100))
?fact <- (OperationWSTime)
=>
    (retract ?fact)
)


; maintaining times for operations for external WS
(defrule time_WS_new "Creating a new current ws time"
?fact <- (OperationWSTime (serviceName ?svcName) (operationName ?opName)
(endpoint ?endpoint & :(<> ?endpoint 0)) (lastTime ?last)
(avgTime ?avg &:(<> ?avg 0)) (trend ?trend))
    (not (OperationWSTimeCurrent (serviceName ?svcName) (operationName ?opName)
     (endpoint ?endpoint) ))
=>
    (add (new OperationWSTimeCurrent ?svcName ?opName ?endpoint ?last
     ?avg ?trend ?avg))
    (?notifier changeTimeWS ?svcName ?opName ?endpoint ?avg)
    (retract ?fact)
)

(defrule changig_nrt "Changing normal response time"
(ExternalWSStatusCurrent (serviceName ?svcName)(endpoint ?endpoint)
(status ?status & :(= ?status 0)))
     ?currentTime <- (OperationWSTimeCurrent (serviceName ?svcName)
      (operationName ?opName) (endpoint ?endpoint)(avgTime ?avg & :(<> ?avg 0))
             (normalRT ?nrt & :(<> ?nrt 0) & :
             (> (/ (abs(- ?nrt ?avg)) ?nrt) 0.5)))
=>
    (modify ?currentTime (normalRT ?avg))
    (?notifier changeTimeWS ?svcName ?opName ?endpoint ?avg)
)

(defrule changing_it "Changing interTime - first " (declare (salience 100))
    (OperationWSTime (serviceName ?svcName) (operationName ?opName)
     (endpoint ?endpoint) (interTime ?it &  :(> ?it 0)))
    (ExternalWSStatusCurrent (serviceName ?svcName) (endpoint ?endpoint)
     (status ?s & :(= ?s 0)))
    ?current <- (OperationWSTimeCurrent (serviceName ?svcName)
     (operationName ?opName) (endpoint ?endpoint)
     (interTime ?itCurrent & :(= ?itCurrent -1) ))
=>
    (modify ?current (interTime ?it))
)

(defrule changing_it_next "Changing interTime - next " (declare (salience 100))
    (OperationWSTime (serviceName ?svcName) (operationName ?opName)
    (endpoint ?endpoint) (interTime ?it &  :(> ?it 0)))
    (ExternalWSStatusCurrent (serviceName ?svcName) (endpoint ?endpoint)
     (status ?s & :(= ?s 0)))
    ?current <- (OperationWSTimeCurrent (serviceName ?svcName)
     (operationName ?opName) (endpoint ?endpoint)
     (interTime ?itCurrent & :(> (abs (- ?it ?itCurrent)) 300) & :
     (> ?itCurrent 0)  & :(> (/ (abs (- ?itCurrent ?it)) ?itCurrent) 1) ))
=>
    (modify ?current (interTime ?it))
    (?notifier info "Changing interTime - next")
)

(defrule time_WS_changed_zero "Changing time for WS - zero avg"
    ?fact <- (OperationWSTime (serviceName ?svcName) (operationName ?opName)
     (endpoint ?endpoint) (avgTime ?avg))
    (ExternalWSStatusCurrent (serviceName ?svcName) (endpoint ?endpoint)
     (status ?status & :(= ?status 0)))
    ?current <- (OperationWSTimeCurrent (serviceName ?svcName)
     (operationName ?opName) (endpoint ?endpoint)
     (avgTime ?t & :(<> ?t ?avg) & :(= ?t 0) ))
=>
    (modify ?current (avgTime ?avg))
    (retract ?fact)
)
```

```
(defrule time_WS_changed_non_zero "Changing time for WS - non zero avg"
    ?fact <- (OperationWSTime (serviceName ?svcName) (operationName ?opName)
     (endpoint ?endpoint) (avgTime ?avg))
    (ExternalWSStatusCurrent (serviceName ?svcName) (endpoint ?endpoint)
     (status ?status & :(= ?status 0)))
    ?current <- (OperationWSTimeCurrent (serviceName ?svcName)
     (operationName ?opName) (endpoint ?endpoint)
     (avgTime ?t & :(<> ?t 0) & :(> (/ (abs(- ?avg ?t)) ?t) 0.1)))
=>
    (modify ?current (avgTime ?avg))
    (retract ?fact)
    (?notifier info "Changing time for WS - non zero avg" )
)

(defrule remove_time_WS_unused "Removing time ws" (declare (salience -150))
    ?fact <- (OperationWSTime)
=>
    (retract ?fact)
)

}
```

B.2 Reasoning about loads on external WSs

```
(import plugin.reasoning.model.*)

(deftemplate ExternalWSStatusCurrent
(declare (from-class ExternalWSStatusCurrent)))
(deftemplate OperationWSTimeCurrent (
declare (from-class OperationWSTimeCurrent)))
(deftemplate LoadInfo (declare (from-class LoadInfo)))
(deftemplate Notifier (declare (from-class Notifier)))

(reset);

;Checks whether load is only to external WSs with normal status
(defrule change_to_existing_normal "Moving to normal" (declare (salience 100))
?loadInfo <- (LoadInfo (serviceName ?svcName) (endpoint ?endpoint))
(ExternalWSStatusCurrent (serviceName ?svcName) (endpoint ?endpoint)
(status ?s & :(<> ?s 0)))
    (ExternalWSStatusCurrent (serviceName ?svcName) (endpoint ?newEndpoint)
     (status ?status & :(= ?status 0)))
=>
    (modify ?loadInfo (endpoint ?newEndpoint) (changed true))
)

;Checks whether load can be omitted if there is no normal WS
(defrule change_to_optional "Omitting optional" (declare (salience 100))
  ?loadInfo <- (LoadInfo (serviceName ?svcName) (endpoint ?endpoint & :
   (<> ?endpoint 0))
(optionalCall ?op & :(= ?op true)))
    (not (ExternalWSStatusCurrent (serviceName ?svcName) (
     status ?s & :(= ?s 0))))
=>
    (modify ?loadInfo (endpoint 0) (changed true))
)

;Checks whether compulsory load can be changed to overloaded WS
(defrule change_to_compulsory_overload "Using overlaoded"
(declare (salience 100))
    ?loadInfo <- (LoadInfo (serviceName ?svcName) (endpoint ?endpoint & :
     (<> ?endpoint 0))
     (optionalCall ?op & :(= ?op false)))
    (not (ExternalWSStatusCurrent (serviceName ?svcName)
     (status ?s & :(= ?s 0))))
    (ExternalWSStatusCurrent (serviceName ?svcName) (endpoint ?newEndpoint)
     (status ?st & :(= ?st 1)))
=>
    (modify ?loadInfo (endpoint ?newEndpoint) (changed true))
)

;Moves load from possibly overloaded external WS not causing overload on other
(defrule change_endpoint_to_not_overloaded "Moving from possible overload"
(declare (salience 0))
    ?loadInfoFirst <- (LoadInfo (serviceName ?svcName) (operationName ?opName)
     (endpoint ?endpoint) (time ?t1))
    ?loadInfoSecond <- (LoadInfo (serviceName ?svcName) (operationName ?opName)
     (endpoint ?endpoint) (time ?t2 & :(> ?t2 ?t1)))
    (ExternalWSStatusCurrent (serviceName ?svcName) (endpoint ?endpoint)
     (status ?st & :(= ?st 0)))
    (OperationWSTimeCurrent (serviceName ?svcName) (operationName ?opName) (
     endpoint ?endpoint) (interTimeOverload ?max & :(< (- ?t2 ?t1) ?max) ))
    (ExternalWSStatusCurrent (serviceName ?svcName) (endpoint ?newEndpoint)
     (status ?s & :(= ?s 0)))
    (OperationWSTimeCurrent (serviceName ?svcName) (operationName ?opName)
     (endpoint ?newEndpoint) (interTimeOverload ?newMax) )
    (not (LoadInfo (serviceName ?svcName) (operationName ?opName)
     (endpoint ?newEndpoint)
     (time ?t3 & :(< (abs (- ?t2 ?t3)) ?newMax ))))
=>
    (modify ?loadInfoSecond (endpoint ?newEndpoint) (changed true))
)

;Omits load for optional calls from possibly overlaod external WS -
;if cannot move to other endpoints
(defrule change_optional_to_not_call "Omitting optional" (declare (salience 0))
    ?loadInfoFirst <- (LoadInfo (serviceName ?svcName) (operationName ?opName)
     (endpoint ?endpoint) (time ?t1) (optionalCall ?oc & :(= ?oc true)))
    ?loadInfoSecond <- (LoadInfo (serviceName ?svcName) (operationName ?opName)
     (endpoint ?endpoint) (time ?t2 & :(> ?t2 ?t1)))
    (ExternalWSStatusCurrent (serviceName ?svcName) (endpoint ?endpoint)
     (status ?st & :(= ?st 0)))
    (OperationWSTimeCurrent (serviceName ?svcName) (operationName ?opName)
     (endpoint ?endpoint) (interTimeOverload ?max & :(< (- ?t2 ?t1) ?max) ))
    (not (and (ExternalWSStatusCurrent (serviceName ?svcName)
     (endpoint ?newEndpoint)
     (status ?s & :(= ?s 0)))
    (OperationWSTimeCurrent (serviceName ?svcName) (operationName ?opName)
     (endpoint ?newEndpoint) (interTimeOverload ?newMax) )
    (not(LoadInfo (serviceName ?svcName) (operationName ?opName)
     (endpoint ?newEndpoint) (time ?t3 & :(< (abs (- ?t2 ?t3)) ?newMax ))))))
=>
    (modify ?loadInfoSecond (endpoint 0) (changed true))
)
```