# Deliverable 3:
# Negotiation protocols compatibility studies and adaptation.

**Abstract.**
In this deliverable we focus on the market place dynamics when agents join dynamically already existent market instances of a specific protocol. We address specially the problem of roles behavioural mismatch. We illustrate the problem by using the two negotiation mechanisms (single shot and iterative). The last part of this deliverable concerns the automatic adaptation as a solution to the behavioural mismatch, again the method is illustrated using an example of a single shot bidder that try to join an iterative environment.

| | |
|---|---|
| **Deliverable Id.:** | **3 and 4** |
| **Date:** | **8/12/2008** |
| **Classification:** | **Public/Private** |
| **Originating Partner:** | |
| **Author(s):** | **LIP 6 + IBISC** |
| **Relevant Work package:** | **WP 3** |
| **Relevant Task:** | **Task 3.1 et 3.2** |
| **Release Status:** | **Version 0** |
| **Project Co-ordinator:** | **Lip6** |
| **Partners:** | **IBISC** |

# I. Introduction

During the first two deliverables we have shown how using the SOA approach we can create an opening market place environment. The openness character of the proposal is due to the fact that the market environment supports the introduction of new negotiation protocol specification using CDL (choreography description language) and that the functional model does not suppose that the entities implementing roles are a priori known and know each other (dynamic discovery). In this deliverable we address the next challenge of the project proposal: the regularities of the markets or how we avoid the appearance of ad-hoc negotiation market places and restrict only market place that implements only already existent negotiation protocols. More precisely we will define what is a correct entity according to a role implementation and while we do not suppose that entities are developed from scratch for a role we will consider the possibility of wrapper synthesis to correct the mismatch between an implementation and a given role specification.

Three points will be addressed in this deliverable, first we focus on the notion of mismatch between a role specification and the entity that implements such role. We focus on the signature level and the behavioural level the semantic level is out of the scope nevertheless we will mention some on going works that deal with it. When the notion of incompatibility is clarified we present the notion of adaptation and we present it as a technique to correct the mismatch in a non intrusive way (no modification of the entity definition) and this by generating a wrapper that fits the entity to the required behaviours.

Last, we will present a method based on Petri Nets [5] for the formalisation and verification of properties on the protocols. This method allows programmers to guarantee that only protocols that respect a given set of properties will end successfully.

The deliverable is structured as follows: section one we recall the functional model of the negotiation market we introduce also some notations. Section 3 we presents the compatibility problem by the way of two scenarios the first when an iterative bidder tries to play in a single Shot protocol and the second when a single shot Bidder tries to play in an iterative protocol. Thus we divide the set of a roles implementations to two subsets, the compatible (regular one) ones and the incompatible ones. The section 4 we introduce the adaptation as a possible solution to make non incompatible implementation as compatible one we illustrate the adaptation mechanism on the second scenario and we generate an adapter. Section 5 presents the formalisation and verification method applied to a specific set of properties. We conclude in section 6 by some remarks and some perspectives of the project.

# II. The functional model: and the incompatibility problem

The WSCDL specification (see deliverable 2) describes a market protocol. The market protocol involves a set of role types. The protocol describes the accepted flow of message exchanges between the different role types from a global point of view. The compilation of WSCDL specification using the PI4SOA tool generates for each role its specification. A role specification is defined by the WSDL interface and the abstract BPEL specification. The WSDL description defines the set of operations that the considered role must provide: we call it the **Signature Interface.** The BPEL and the partners' declarations define the behaviour of the role in term of the accepted order of its operations invocation and also the partner ones invocation, we call this the **Behavioural Interface**. Note here that the composition of the different partners behaviours fits within the global specification (WSCDL one).

The functional model (presented in deliverable 1) acts in two different level; class level and instance level. In the class level, market mechanisms (protocols specification) are created (using WSCDL) and then compiled to a set of partners Roles specifications (WSDL + BPEL). In the instance level, a set of BPEL services instance are connected to realize a market mechanism instance. As claimed in deliverable 1 such web services are not supposed to be defined from scratch using the compilation steps specification. In fact existent web services can join the virtual market and play roles. Also even if a web service was developed from scratch to implement a specific role of a specific protocol would play an other role of an other protocol. The question raised in this deliverable is ; in open virtual market how can we control the validity of Web services desiring playing a role of a specific protocol. To answer this question we must first define what is is a correct execution of a protocol then we define what is a good (correct) candidate for a role to guarantee a correct execution.

During all the project we have worked on two types of protocols the first is a single shot and the second is an iterative shot. Both of them are defined on the same set of roles only the behaviours of some of roles are different. The roles are the SIS, the Bidder, The Market and the initiator (note here that this not exclude the fact that other roles can be added or removed). The SIS plays the roles of a registry of markets instances. This role is very specific and its implementation (or the Web service implementing this roles) must be controlled that's way we suppose here that this roles is not concerned by dynamic agent in the virtual market place. We can suppose that the market place contains implementation of this roles as Web services. The market role implements the core of the auction protocols (the rules of the game and also the decision mechanism of the winner) thus makes it also very specific in our context while an implementation of a market must be controlled not only in term of protocol but also in term of rules implementation: partiality, etc. So dynamic agent playing roles of market needs to be trusted which is not the aim of this work.

Now we will suppose that for each of the two protocol it exists a set of agents (here Web services ) that implements correctly the SIS and the markets roles. It remains the most dynamic part of market place which is the players represented here by Initiator and the Bidder. This two roles can be played by dynamic agents that appear and disappear in the market place. In this deliverable, for simplicity reasons, we will focus on one of the two roles which is the more complex one, the Bidder.

# III. Incompatibility problem

To illustrate the incompatibility we will try to simulate two cases ; the first case concerns an agent implementing the bidder role of iterative protocol that joins a single shot partnership (SIS, Initiator and Market), while the second case is about a single shot Bidder that joins an iterative partnership. In the sequel we will model each of this two cases by presenting the specification (generated using the PI4SOA tool, see deliverable 2) of each role and then illustrate the compatibility problem.

## 1. General consideration of the approach

When designing an auction protocol using choreography, the designer specifies an implicit behaviour of a given role according a global view (see deliverable one for the sketch on top-down *vs* bottom-up approach or scenario). After compilation, the resulted behaviour (of a specific role) specification represents a the behaviour type required by the partnership (the environment) in order to terminated the protocol (here the termination is intended to be from protocol engineering point of view i.e. no dead or live locks). Let **P** be a negotiation protocol defined by the composition of set of roles **R_1....R_n** (noted **P= C( R1.....Rn)** when **C** is composition operator (in our case it will be a synchronized parallel composition).
Let now **S be a service and R_i a role the question whether S_i is a correct implementation of R_i can be handled by the way of two approach :** the first one we can compare directly the behaviours specification of **S** and the role specification **R_i** according some equivalence relation (that conserve the termination) and the software engineering literature contains some of them (weak-bissimulation etc...). In [ ] we have shown that those relation may be restrictive (see [] for more detail on this topic). The second approach is to consider S as compatible to **R_i** if the composition of S is compatible to the environment of **R_i**. The environment specification of the role R_i is the composition of all the roles of the protocol but **R_i**, we note it E(**R_i)= C(R_j)** with **j ≠ i).** The compatibility relation considered here is the perfect client relation presented in []. In the sequel to introduce the incompatibility on an implementation of role we use the second approach applied on the two former scenarios.

## 2. Case1 : iterative bidder single shot partnership

We present here the case when an iterative implementation of the bidder is proposed to play the bidder in a single shot protocol. In the following, we ill present the single shot environment **SingeleShotEnv=C(SIS, Market, Initiator)** and the the iterative bidder role specification and we end this section by the composition of the hole partnership and we check for safety.

### The single shot Bidder environment specification:

The Single shot bidder environment is composed by a single shot Market, the SIS and the Initiator.
**The Market:**
The market is invoked for an instance creation by the initiator and then it registers the instance in the SIS and waits for Bidders registrations. When the biding session is open the market sends the session value to the registered Bidders (here we consider that the price is in the interval [0 ..3] for state number reason) then waits for the bids. if a bid value is less then the session value a bad bid message is sent. In the end of the session the bidders are informed by the result. Here is its FSP specification. Some convention are used by the tool to generate the model. Each operation **O** that belongs to a partner **P** and composed by an input message **M1**

and an output message **M2** is writen **P_wsdl_M1|M2_O.** When such message appears in the specification of the partner **P** it correspond to the receive of the invocation (when IN) and the reply (when OUT). If the message appears in the specification of a partner **P'** different from **P** it corresponds to the invocation**.**

MARKETPAR_RECEIVE110 = (market_wsdl_IN_instanciatemarketop -> END).
MARKETPAR_RECEIVE211 = (market_wsdl_IN_getmarketownerref -> END).
MARKETPAR_INVOKE312 = (sis_reg_wsdl_IN_registerop -> END).
MARKETPAR_INVOKE413 = (sis_reg_wsdl_IN_getbibsessionref -> END).
MARKETPAR_RECEIVE514 = (market_wsdl_IN_askforplay -> END).
MARKETPAR_REPLY615 = (market_wsdl_OUT_askforplay -> END).
MARKETPAR_INVOKE716 = (bidder_wsdl_IN_sendcurrentvalue[v:Price] ->
MARKETPAR_RECEIVE817[v]),
MARKETPAR_RECEIVE817[v:Price] = (market_wsdl_IN_biding[b:Price] -> if (b<v) then (bidder_wsdl_IN_badbidoutofgame -> bidder_wsdl_IN_notifyres -> END) else (bidder_wsdl_IN_notifyres -> END)).

*// the market model specification*

**MARKETPAR** = MARKETPAR_RECEIVE110 ; MARKETPAR_RECEIVE211 ; MARKETPAR_INVOKE312 ; MARKETPAR_INVOKE413 ; MARKETPAR_RECEIVE514 ; MARKETPAR_REPLY615 ; MARKETPAR_INVOKE716; END .

**The initiator :**
The Initiator partner is the role that creates a market instance (for a given good). It then gives the market its references in order to be contacted at the end of the bidding session for the result. Here is its FSP specification.

INITPART_INVOKE12 = (market_wsdl_IN_instanciatemarketop -> END).
INITPART_INVOKE23 = (market_wsdl_IN_getmarketownerref -> END).
**INITPART** = INITPART_INVOKE12 ; INITPART_INVOKE23; END.

**The SIS:**
the SIS Offer two independent services one for market registration and one for market localisation. When a market instance is initiated by the Initiator the market register the information about the instance. The SIS offer also an interface to the Bidders in order to locate during runtime instances of a given market protocol and/or goods. Here is its specification.

SISPARTICIPANT_RECEIVE15 = (sis_reg_wsdl_IN_registerop -> END).
SISPARTICIPANT_RECEIVE26 = (sis_reg_wsdl_IN_getbibsessionref -> END).
SISPARTICIPANT_RECEIVE37 = (sis_get_wsdl_IN_getmarket -> END).
SISPARTICIPANT_REPLY48 = (sis_get_wsdl_OUT_getmarket -> END).
SISPARTICIPANT_INVOKE59 = (bidder_wsdl_IN_getmarketref -> END).
**SISPARTICIPANT** = SISPARTICIPANT_RECEIVE15 ; SISPARTICIPANT_RECEIVE26 ; SISPARTICIPANT_RECEIVE37 ; SISPARTICIPANT_REPLY48 ; SISPARTICIPANT_INVOKE59; END.

*Figure 1: The Single Shot Environement models*

**The single Shot Bidder Environment.**

The Three pre-cited roles compose the environment of the bidder according to the single shot specification. In order to obtain what the partnership wait from a correct bidder partner we compose them in order to model the bidder environment. The single Shot Bidder Environement is obtained by a synchronous composition fo the SIS, the INITIATOR and the MARKET specifications. The communication between this roles are unobservable ones from the bidder Roles that's why we project the resulted composition on the bidder set of actions.

set BIDDER_ALPHA={sis_get_wsdl_IN_getmarket, sis_get_wsdl_OUT_getmarket, bidder_wsdl_IN_getmarketref, market_wsdl_IN_askforplay, market_wsdl_OUT_askforplay, bidder_wsdl_IN_sendcurrentvalue[v:Price], market_wsdl_IN_biding[v:Price], bidder_wsdl_IN_badbidoutofgame, bidder_wsdl_IN_notifyres}

||**SINGLESHOTMINUSBIDDER**=(SISPARTICIPANT || MARKETPAR || INITPART) @ BIDDER_ALPHA .

### The Iterative Bidder specification

The next step consist in making an iterative bidder playing a single shot negotiation protocol for that we will compose the environment resulted form the previous step within the iterative bidder role specification. The iterative bid behaviours is obtained by compiling the iterative protocol WSCDL specification. Here is the iterative bidder specification:

BIDDER_INVOKE17 = (sis_get_wsdl_IN_getmarket -> END).
BIDDER_INVOKE17_REPLY = (sis_get_wsdl_OUT_getmarket ->END).
BIDDER_INVOKE17_SEQ = BIDDER_INVOKE17; BIDDER_INVOKE17_REPLY; END.
BIDDER_RECEIVE28 = (bidder_wsdl_IN_getmarketref -> END).
BIDDER_INVOKE39 = (market_wsdl_IN_askforplay -> END).
BIDDER_INVOKE39_REPLY = (market_wsdl_OUT_askforplay ->END).
BIDDER_INVOKE39_SEQ = BIDDER_INVOKE39; BIDDER_INVOKE39_REPLY; END.
BIDDER_RECEIVE29 = (bidder_wsdl_IN_sendcurrentvalue[t:Price] ->END).
BEGINITBID= (bidder_wsdl_IN_sendcurrentvalue[t:Price] ->
market_wsdl_IN_biding[v:Price] -> (bidder_wsdl_IN_notifyres -> END |
bidder_wsdl_IN_badbidoutofgame -> BEGINITBID) | bidder_wsdl_IN_notifyres -> END ).
**BIDDER** = BIDDER_INVOKE17_SEQ ; BIDDER_RECEIVE28 ;
BIDDER_INVOKE39_SEQ ; BEGINITBID; END.

Its graphical representation can be found in the figure 2.

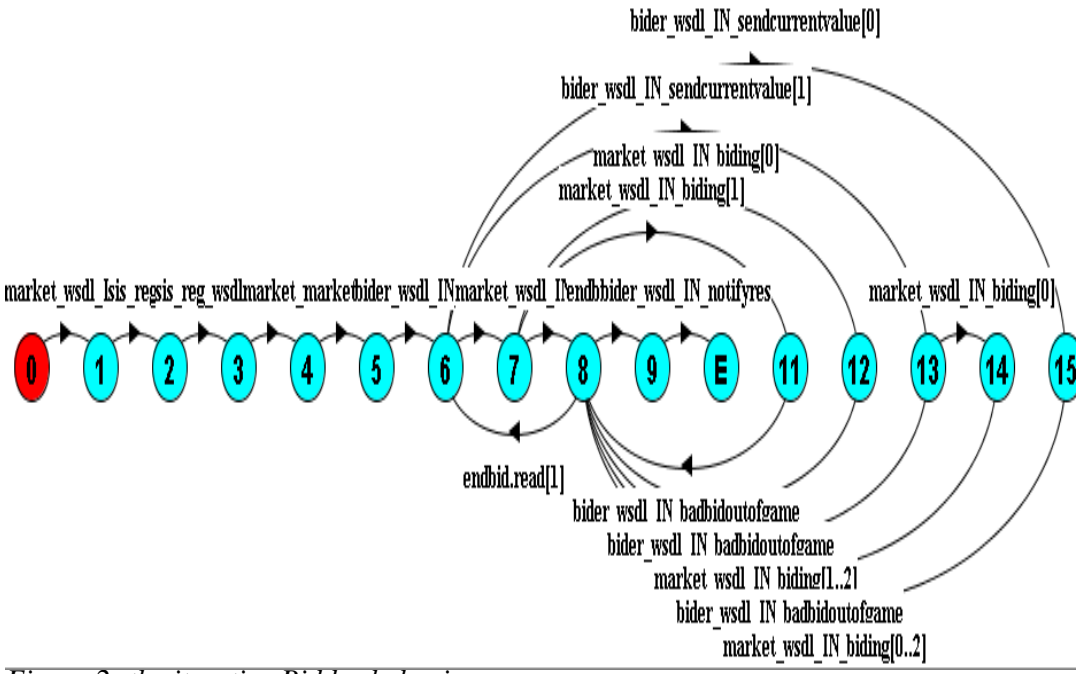



*Figure 2: the iterative Bidder behaviour*

It is obvious to see that the iterative bidder is a generalisation of the single shot behaviours. Its also easy to see that while the single shot bidder behaviours is included in the iterative bidder one then the iterative one can fulfil the Single shot bidder Environment constraints. The analysis of the composition of the iterative Bidder and the Single Shot Environment is deadlock free.

To conclude this section we have shown by this example that a given role can be played by a specification other then generated by the compilation of the hole protocol. We can imagine that other services (supposing a semantic mismatch between messages and operations) can plays a given role in a negotiation. We can also conclude that it also possible to check that compatibility just by composing the services behaviours specification within the target protocol environment. In [HAD04] we have addressed this problem about partner compatibility in a choreographed Web services Environment.

# 3. Case2 : A single shot bidder and iterative partnership

We continue to explore the compatibility problem by the inversing the situation of the previous section. Here we try to make a composition of a singleShot bidder within an iterative environment. As in the previous section we presents the behavioural specification of each role and then we show the result of the composition in thus scenario.

## The single shot Bidder environment specification:

The Single shot bidder environment is composed by a single shot Market, the SIS and the Initiator.

**The Market:**

The beginning on the market processes in the iterative protocol is similar to the single shot one. The main difference is that the biding session is repeated an undefined number of times. Each session the market fix the session values and send it to the set of bidders then they send their bid (the bid is also rejected if the sent is less than the session value). The biding ended when the market sends a result message. Here is the FSP specification of the market protocol.

```
MARKETPAR_RECEIVE110 = (market_wsdl_IN_instanciatemarketop -> END).
MARKETPAR_RECEIVE211 = (market_wsdl_IN_getmarketownerref -> END).
MARKETPAR_INVOKE312 = (sis_reg_wsdl_IN_registerop -> END).
MARKETPAR_INVOKE413 = (sis_reg_wsdl_IN_getbibsessionref -> END).
MARKETPAR_RECEIVE514 = (market_wsdl_IN_askforplay -> END).
MARKETPAR_REPLY615 = (market_wsdl_OUT_askforplay -> END).
MARKETPAR_INVOKE716 = (bidder_wsdl_IN_sendcurrentvalue -> END).
MARKETPAR_INVOKE1019 = (bidder_wsdl_IN_notifyres -> END).
BIDSESSION=ITERATION[1],
BEGINIT = (endbid.read[i:MARKETPAR_IntRange]->ITERATION[i]),
ITERATION[i:MARKETPAR_IntRange] = if (i==1) then
(bidder_wsdl_IN_sendcurrentvalue[v:Price] -> market_wsdl_IN_biding[t:Price] ->
TESTBID[v][t]) else END,
TESTBID[v:Price][t:Price]=if (t<v) then (bidder_wsdl_IN_badbidoutofgame -> BEGINIT)
else BEGINIT.
MARKETPAR = MARKETPAR_RECEIVE110 ; MARKETPAR_RECEIVE211 ;
MARKETPAR_INVOKE312 ; MARKETPAR_INVOKE413 ; MARKETPAR_RECEIVE514
; MARKETPAR_REPLY615 ; BIDSESSION ; MARKETPAR_INVOKE1019; END.
// Entry: SEQUENCE end ---------------------------
//||MARKETPAR_Instance = (SWITCH1BADBIDSWITCH1BADBIDBADBIDOUTCOME
|| MARKETPAR_SEQUENCE1).
```

**The INITIATOR and the SIS :**
this two roles have exactly the same behaviours as in th single shot protocol.

INITPART_INVOKE12 = (market_wsdl_IN_instanciatemarketop -> END).
INITPART_INVOKE23 = (market_wsdl_IN_getmarketownerref -> END).
**INITPART_SEQUENCE1** = INITPART_INVOKE12 ; INITPART_INVOKE23; END.
SISPARTICIPANT_RECEIVE15 = (sis_reg_wsdl_IN_registerop -> END).
SISPARTICIPANT_RECEIVE26 = (sis_reg_wsdl_IN_getbibsessionref -> END).
SISPARTICIPANT_RECEIVE37 = (sis_get_wsdl_IN_getmarket -> END).
SISPARTICIPANT_REPLY48 = (sis_get_wsdl_OUT_getmarket -> END).
SISPARTICIPANT_INVOKE59 = (bidder_wsdl_IN_getmarketref -> END).
**SISPARTICIPANT** = SISPARTICIPANT_RECEIVE15 ; SISPARTICIPANT_RECEIVE26 ; SISPARTICIPANT_RECEIVE37 ; SISPARTICIPANT_REPLY48 ; SISPARTICIPANT_INVOKE59; END.

**The iterative Bidder Environment**
Again we compose (using a synchronous product) the three pre-cited roles in order to define the iterative protocol environment of the bidder role. The Three pre-cited roles compose the environment of the bidder according to the single shot specification. As the two roles Initiator ans SIS are constants roles then the environment is changing according to the changes of the market behaviour. Here is the FSP specification of the iterative bidder Environment :

||**SINGLESHOTMINUSBIDDER**=(SISPARTICIPANT || MARKETPAR || INITPART)@ BIDDER_ALPHA.

The single Shot Bidder

BIDDER_INVOKE17 = (sis_get_wsdl_IN_getmarket -> END).
BIDDER_INVOKE17_REPLY = (sis_get_wsdl_OUT_getmarket ->END).
BIDDER_INVOKE17_SEQ = BIDDER_INVOKE17; BIDDER_INVOKE17_REPLY; END.
BIDDER_RECEIVE28 = (bidder_wsdl_IN_getmarketref -> END).
BIDDER_INVOKE39 = (market_wsdl_IN_askforplay -> END).
BIDDER_INVOKE39_REPLY = (market_wsdl_OUT_askforplay ->END).
BIDDER_INVOKE39_SEQ = BIDDER_INVOKE39; BIDDER_INVOKE39_REPLY; END.
BIDDER_RECEIVE410 = (bidder_wsdl_IN_sendcurrentvalue[v:Price] -> END).
BIDDER_INVOKE511 = (market_wsdl_IN_biding[t:Price] -> END ).
EXTSWITCH=(bidder_wsdl_IN_badbidoutofgame -> bidder_wsdl_IN_notifyres -> END | bidder_wsdl_IN_notifyres -> END).
BIDDER_RECEIVE713 = (bidder_wsdl_IN_notifyres -> END).
**BIDDER** = BIDDER_INVOKE17_SEQ ; BIDDER_RECEIVE28 ; BIDDER_INVOKE39_SEQ ; BIDDER_RECEIVE410 ; BIDDER_INVOKE511 ; EXTSWITCH ; END.

similar to the previous section we compose the single shot bidder with an iterative bidder Environment and then we check it for deadlock freeness. As expected the hole process fails because the environment ask the bidder to be capable to play more then one round while the single Shot can play only one. The process can deadlock when the market decides to play a second round. The single Shot bidder after sending its first bid expects either an exception or a result message and the Environment except that in addition to the exception and the result the

bidder can receives the next round session value message. The LTSA tools analysis capability can produce the trace that leads to the deadlocks and here is the result of the deadlock analysis produced by LTSA (Figure 3) :

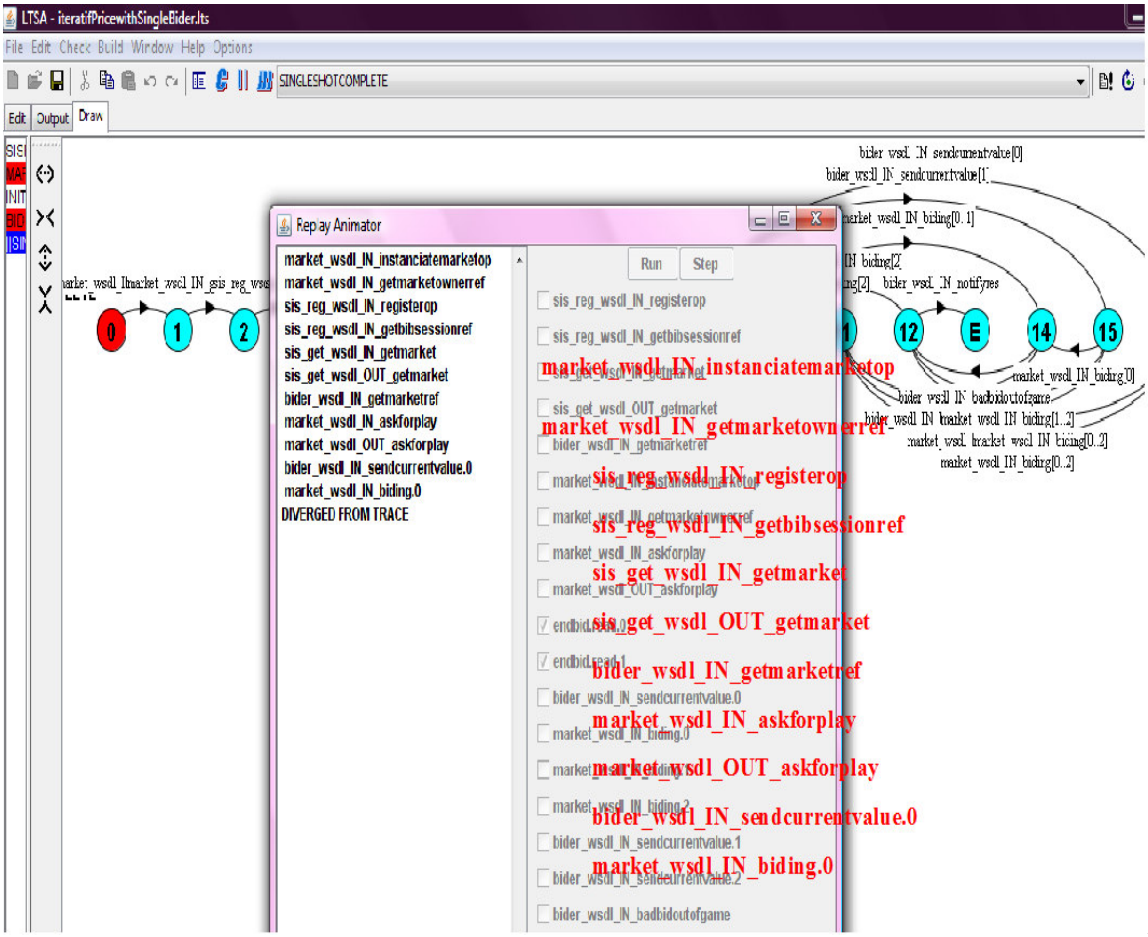



*Figure 3: The LTSA compatibility analysis result*

# IV. Toward role Adaptation

We have shown in the previous section based on example how implementation of a given role can be checked as compatible one or not against the roles environment. Two cases are presented; a case where the agent playing the role is compatible, an iterative bidder playing a single shot protocols, and a second one where it is not compatible, case of single shot bidder that plays an iterative protocol. In this section we focus on the the second case and we try to give some direction to answer to the question : When an agent is not compatible in which case we can add a wrapper, using adaptation, that make it compatible and what is the effect on the protocol properties? First we explain what is adaptation and then we illustrates a method of adaptation on the second case.

## 4. Introduction to model based adaptation

Component-Based Software Engineering (CBSE) aims at building systems by assembling pre-produced software components, which would jointly realize the desired functionality[3]. However, one of the main issues raised by this approach is that in practice we cannot expect that any given software component perfectly matches the needs of a system where it is trying to be reused, nor that the components being assembled fit perfectly one another. Reusing software often requires a certain degree of adaptation especially in presence of legacy code. To deal with these problems, *Software Adaptation* is a discipline, concerned with providing techniques to arrange already developed pieces of software in order to reuse them in new systems, eliminating the potential mismatches arising from their composition.

Software Adaptation promotes the use of *adaptors*, specific computational entities developed for guaranteeing that a set of mismatching components will interact correctly and that without modifying the code of the components, which is important due to their black-box nature. Commonly several levels of interoperability, and accordingly of interface description [2,4]: signature level, behavioural level (interaction protocols), service level (non-functional properties such as quality of service), and semantic level (functional specification of what the component actually does). At each one, mismatch may occur and have to be corrected.

In this project and as explained in the introduction we focus on the mismatch in the behaviours Level. The Figure 4 presents an over view of the adaptation processes at the behavioural level; the input of the problem is a set of agent (web services) behaviour specification (Abstract BPELs) and the abstract description of the constraints that must be respected to make the involves components work together. Such description is called an adaptation mapping or contract. The out put of the process is component that play the role of wrapper that control the evolution of each component in such manner that lead them all to a safe sates. Generally the mapping or the contract of behavioural adaptation is defined by a set of synchronisation vectors. The vector are given as input making adaptation methods as semi automatic.

In a recent work we have proposed an other approach where we suppose that message make a reference to a semantic structure and we tried to infer automatically the set of possible synchronisation vectors. Also the aim of the works is to generate a set of distributed adapter in stead of a centralized one.

In the sequel of the section we will briefly present the adaptation using semantics and then we apply it on the adaptation of the single shot Bidder in an iterative environment.

*Figure 4: Distributed Web services Adaptation [2]*

### Semantic structure:

A semantic structure that we note here I is a couple (U,R) where U is a set of units of sense (UoS), over which we range using u, and R □ $2^U \times U$ is a relation where (U, u) □ R denotes that given a set U of UoS, one can obtain u. The idea of the this structure is to define the composition relation between basic types and structured one (in case of Web services the structured types are the messages).

We suppose that each Web service we have a mapping relation between its message types to a set of unit of senses used to define the message. So for each message we can write it as a set of unit of sense.

### Services Behaviour Model

Each service can be modelled using a Semantic extension of Input output Transition Systems (SIOLTS). An SIOLTS is an automaton which action are split in sending and receiving messages actions (also called communicating automaton). Each state contains a configuration represented by a set of unit of sense. Given a state the configuration associated represents the minimum set of UoS known by the service at this stage.
An SIOLTSA is tuple <S, M, Δ, $s_0$, F,Mat> :

- S a finite set of states
- H: S--> $2^U$ the configuration that associates to each state a set of UoS considered as known at that state
- M= !M U ?M U {tau} // tau here represents the internal action
- Mat : M→$2^U$
- Δ ⊆ S × M × S with
  1. (s,!m,s') ∈ Δ iff Mat(m) ⊆ s
  2. (s,?m,s') ∈ Δ ⇒ Mat(m) ⊆ of s'
- $s_0$ is initial state
- F is a sub set of S representing final states.

Th transition relation must respect the following constraint : first, a message can be send form configuration iff that state contains the set of UoS to compose the message. When we receives a message the next state contains the set of UoS of the received message.

### Adaptation problem

Given a semantic structure and a set of services noted here by $Si=<S_i, M_i, \Delta_i, s_{0i}, F_i, Mat_i>$ we try to generates set of Adapter $A_i$ (SIOLTSs) in such way the composition of adapters and services is deadlock free.

**Adaptation Method :**

In we have presented in more detail the adapters synthesis method here we give the intuition of each step then we apply it on the example of case 2. Our method is based on four steps :

1. first we generate SIOLTS that models each partner
2. we compute a perfect client (or also the perfect Environment of each partner)
3. we compute a global adapter that compose the different environment
4. Two cases are possible :
   - either the global adapter is impossible and then we conclude that no possible adaptation
   - Or we construct the global adapter and then we distribute it.

## 5. Applying the Adaptation method on the Single Shot bidder in an iterative Bidder Environment

As shown in the section 3 the case two can dead lock because the iterative market can execute more that one bidding session while the single shot Bidder can play only one.
Here we apply our adaptation method to show how adaptation can correct the incompatibility of an agent and a role.

**Step1 : modelling the two services**
We have two partner the iterative bidder Environment and SingleShot Bidder. For clarity reason we present here a part the SIOLTS that concerns the adaptation, the bidding session. The involved message types are :

_sendcurrentvalue → (for short) sessionV
_biding → bid
_badbidoutofgame → badbid
_notifyres → result

we add a simple semantic structure that contains one Uos for each Message I=({p,b,e,r},∅) we match each message to a subset of UoS :

_sendcurrentvalue → {p} // a sesion value data
_biding → {b} // a biding value data
_badbidoutofgame → {e} // exception
_notifyres → {r} // result data

The Figure 5 represents the two partner behaviour and the direction of the message (sending (_in) and receiving(_out))

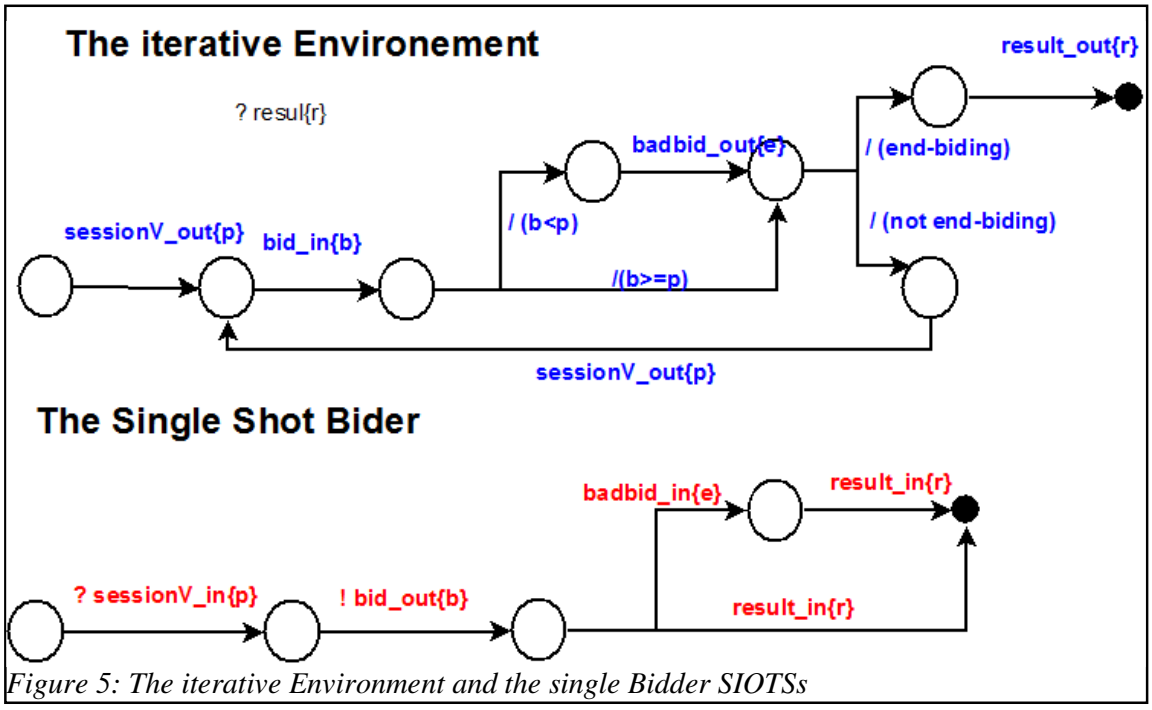



*Figure 5: The iterative Environment and the single Bidder SIOTSs*

**Step 2 : generation of the perfect Clients**

this step consist in designing the need of each partner to reach its final states. What we call the perfect Client. Note that the perfect Client of the iterative Bidder environment is the iterative Bidder it self (by construction of the compilation step). And note also that the perfect client of the single shot Bidder is its environment. In X we have presented an algorithm in order to synthesis a perfect client of a correct service. One can see such client as τ_reduction while constructing a mirror (by complementing the messages direction) .

The figure 6 represents the perfect client of the two involved partner.

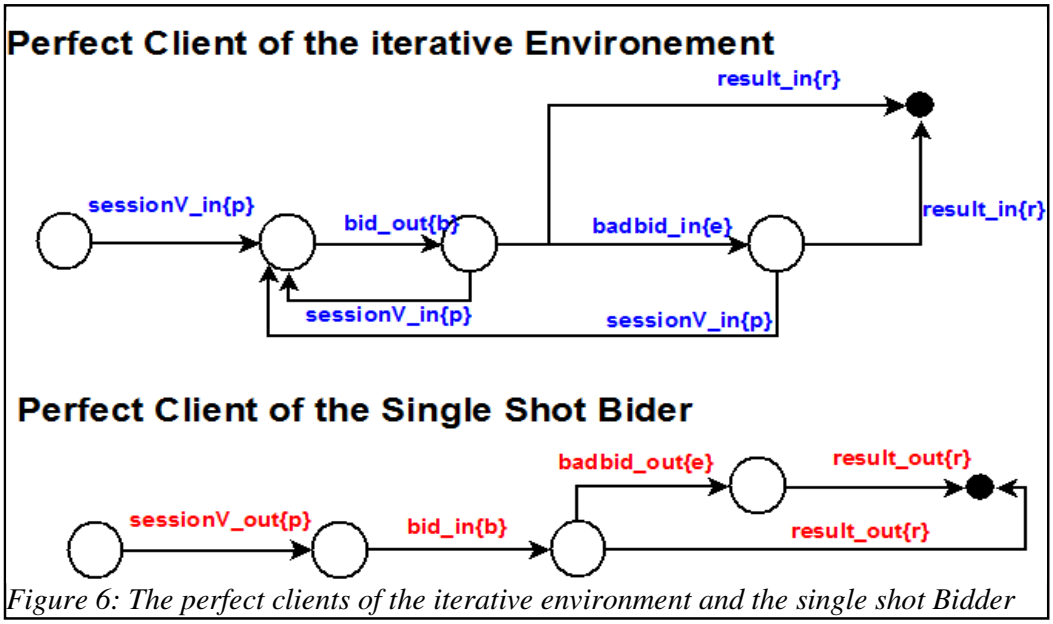



*Figure 6: The perfect clients of the iterative environment and the single shot Bidder*

## Step 3 : constructing the Global Adapter

the construction of the global adapter is obtained by an asynchronous product of each partner perfect client. The product is restricted by the constraint on the UoS : Receiving increase the configuration and sending is possible only when the configuration allow it. The idea is to evolve clients while it possible and search causality between receiving information and sending information. Note here that the perfect client actions direction are the opposite of the partner's ones (the receive is allowed because the partner can produced it).

We developed here few steps of the global Adapter construction and then we give directly the resulted IOLTS.

We begin with en empty configuration and the initial state represents the initial state of the two partners perfect clients. All the receive actions from a partner current state is possible leading to configuration augmented by the received message UoS. Only the output action which do not need any UoS are possible from the initial states. As classic in asynchronous product only one partner state evolves at each step (all combination must be explored). Each state is then a combination of partner states with a configuration. Each time we reach state already reached (same states same configuration then we redirect the arc to that state). In the Figure 7 we give the global adapter and in each state we give also the configuration. we keep the colour code used for perfect client (blue for the environment perfect client and red for the Bidder perfect client).

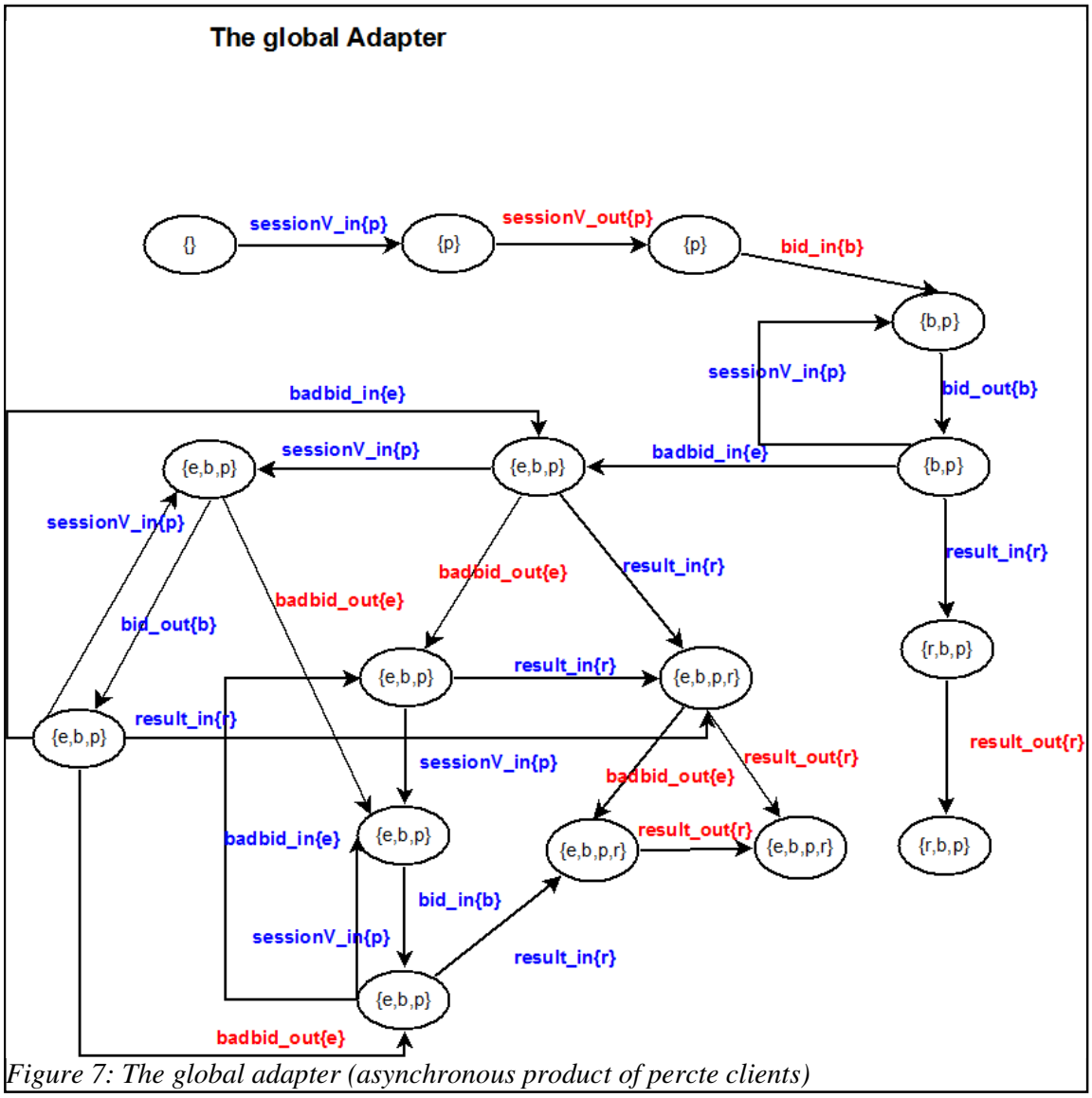



*Figure 7: The global adapter (asynchronous product of percte clients)*

## Step 4 : generating local adapter

The local adapter is an extension of the perfect clients into local adaptors
3 steps (for each perfect client):

- adding receptions of required UoS from other perfect clients
- adding emissions of UoS required by other perfect clients
- updating alphabets and operations

All this sub steps can be performed using (backward | forward) analysis of the global adaptor.

In our case we are face to bi-adaptation so without distributing the behaviour the global adapter can be used to adapt one the two partner to the other according a specific renaming of the actions. While we consider that environment must not be modified we suppose that the incoming role must adapt it self to the Environment, for that we have just to reinterpret the global adapter by reconsidering the meaning of the action as follow (i) the blue action are action directed to the environment (here on the part of the protocol that we consider to the market while only the market is involved here ) and the red actions are directed to the role to

be adapted, here the single shot bidder. We obtain then the local adapter to be placed (as wrapper) between the single Shot Services and the iterative environment.
In order to check the correctness of the adapter we can compose the bidder, the environment and the adapter and we check for deadlock freeness. Here after the FSP specification of the adapted part of the protocol. Note that we focus only on the part to be adapted (the biding session) also we prefix message by the owner of the message and the component concerned by e g **e.bid_out** is the sending of the bid action from the adapter to the Environment.

***/*the bidder bahviour */***

**BIDDER=**(sessionV_in->bid_out->(badbid_in->result_in->**END**|result_in->**END**)).

***/*the environement bahviour */***

**ENV**= (sessionV_out->**BIDSESSION**),
**BIDSESSION**= (bid_in->(badbid_out->(sessionV_out->**BIDSESSION** | result_out->**END**)|sessionV_out->**BIDSESSION**| result_out->**END**)).

***/*the adapter behaviour */***

**ADAPTER**=(e.sessionV_in->b.sessionV_out->b.bid_in->**AD1**),
**AD1**=(e.bid_out->**AD2**),
**AD2**=(e.result_in->b.result_out->**END**| e.badbid_in->**AD3** | e.sessionV_in->**AD1**),
**AD3**=(e.sessionV_in->**AD4**|b.badbid_out->**AD6**|e.result_in->**AD7**),
**AD4**=(e.bid_out->**AD5**|b.badbid_out->**AD8**),
**AD5**=(e.sessionV_in->**AD4** | e.badbid_in->**AD3** | e.result_in->**AD7** | b.badbid_out->**AD9**),
**AD6**=(e.result_in->**AD7** | e.sessionV_in->**AD8**),
**AD7**=(b.result_out->**END**| b.badbid_out->**AD10**),
**AD8**=(e.bid_out->**AD9**),
**AD9**=(e.result_in->**AD10**|e.sessionV_in->**AD8**|e.badbid_in->**AD6**),
**AD10**=(b.result_out->**END**).

***/* renaming actions in order to obtain synchronisation */***
||**ENV_S=ENV**/{sessionV/sessionV_out,bid/bid_in,badbid/badbid_out,result/result_out}.
||**BIDDER_S=BIDDER**/{sessionV/sessionV_in,bid/bid_out,badbid/badbid_in,result/result_in}.
||**ADAPTER_S=ADAPTER**/{e.sessionV/e.sessionV_in,e.bid/e.bid_out,e.badbid/e.badbid_in, e.result/e.result_in,b.sessionV/b.sessionV_out,b.bid/b.bid_in,b.badbid/b.badbid_out, b.result/b.result_out}.

***/* the composition of the singleshot bidder, the iterative Environement and the adapter. */***

||**ADPTER_BIDDER**=({b}::**BIDDER_S** || {e}::**ENV_S** || **ADAPTER_S**).

## interpretation of the result

The generated adapter is wrapper. During the first bid session it does nothing other then forwarding the singleShot Bidder messages to the environment and the environment message to the Bidder. After the first bid, two cases are possible : either the Market end the biding by sending the result or it decides to plays an additional biding session by sending the new session value. In the first case the adapter continue to play a role of bridge between the Bidder and the environment. In the second case and during the second biding session the adapter will receives the new session value (it will not be sent to the Bidder, not expected) and then plays by sending the bid value of the first session and so on. At the end it forwards the final result the single Bidder (and the exception if the new fixed value of the session increases over the bidder price).

# V. Formalisation and verification of properties

## 5.1. Introduction

The aim of this section is to describe a method for the formalisation and verification of properties on the protocols. This allows to guarantee that only protocols that respect the properties will end successfully.

This proposition relies on the automatically translation of the protocols into Petri Nets, based on the work by Amal El Fallah Seghrouchni and Hamza Mazouzi [6]. The expression of guards on transitions of the Petri Net allows to check the properties at runtime.

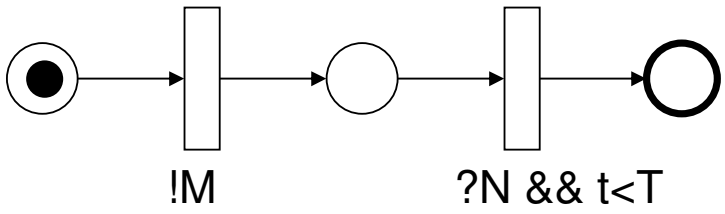
The next section presents the formal representation of protocols using Petri Nets. The third section presents the formal representation of properties that were discussed with OrangeLab for this project.

## 5.2. Translation of protocols into Petri Nets

Considering the protocols that were proposed in the deliverable 1, we can identify four main situations that correspond to four properties to be checked:

I. The general case: Agent A send a message M to agent B and it expects an answer N before a given timeout T.

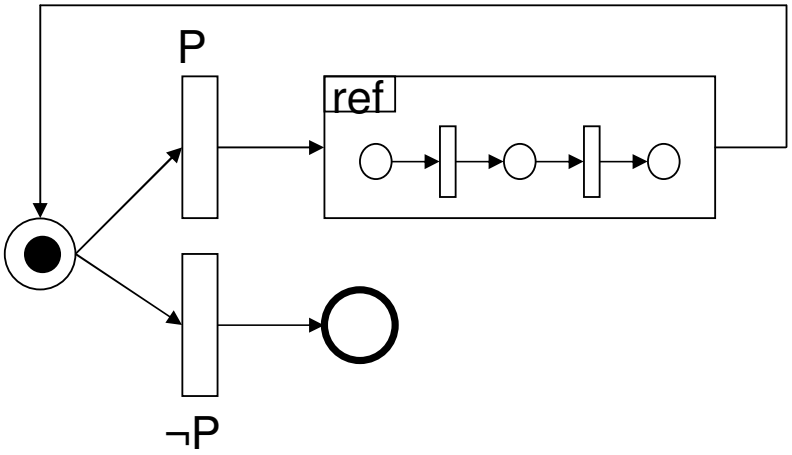
This situation can be represented by the following net:




This net ends only is the agent sents message M and receives an answer N before timeout T.

II. The iterative case (loop elements in AUML): a sub-protocol runs as long as a property P is verified.

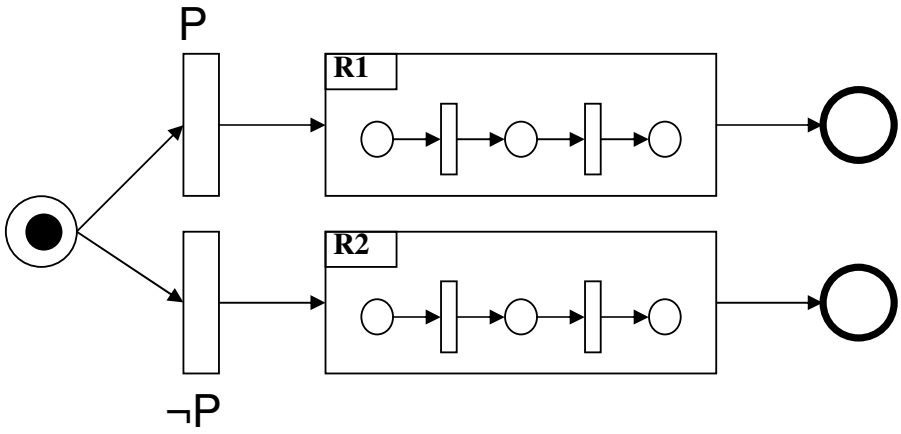
This situation can be represented by the following net:




This net ends only when P is false after the execution of the referred sub-net (identified by tag ref in the figure).

III. The alternative case (alt elements in AUML): the agent must choose between two sub-protocols, depending on a property P.

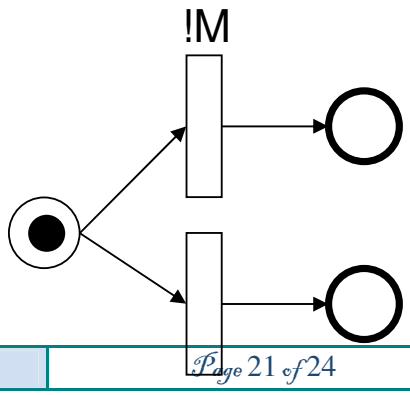
This situation can be represented by the following net:




This net will either trigger the sub-net R1 when P is true or R2 when P is false. The net ends only after the triggered sub-net ends.

IV. The optional case (option elements in AUML): a message M can be sent but it is not mandatory.

This situation can be represented by the following net:



Combining these four nets allows the programmer to model all protocols defined in deliverable 1 and to prove the termination of all these protocols.

In addition, a set of properties has been proposed by OrangeLab for verification.

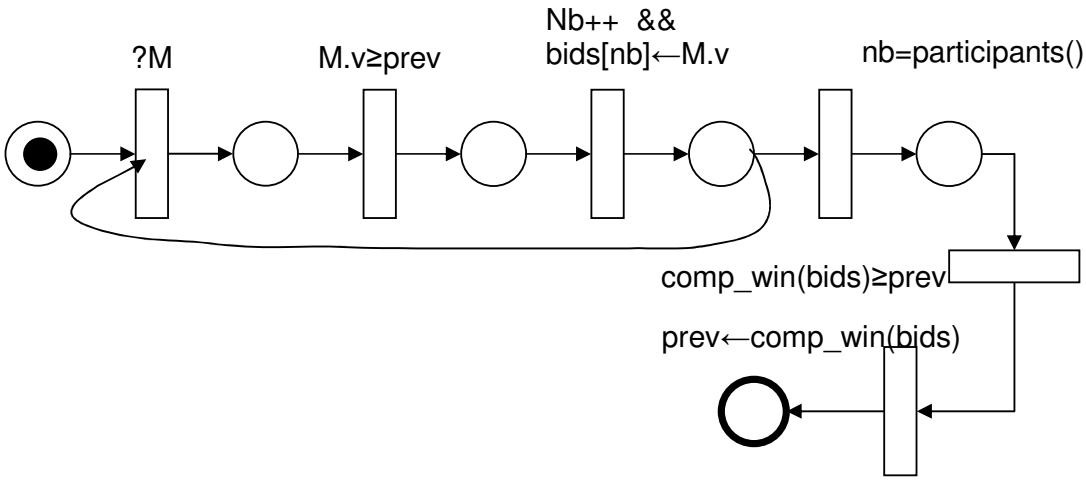
## 5.3. Verification of properties

Some properties can be verified using guarded-transition Petri Nets. Each property corresponds to specific conditions, associated to predefined variables and functions that can be used to compute numerical aspects of the agent's current state.

In the project, we considered the following two properties:

V. At the end of any round the Bid of a provisionally winning Buyer is always superior (or equal) to the last announced price.

This property corresponds to the following net:

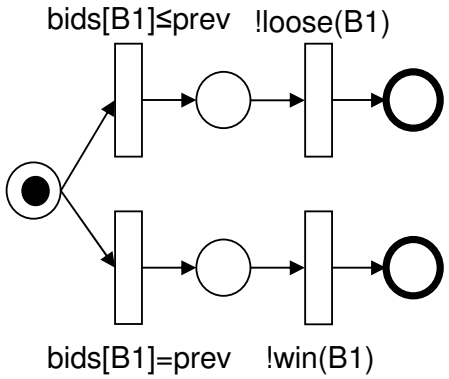



where:

- **M.v** represents the value given in message M;
- **prev** is a variable that stores the bid value of the previous turn;
- **nb** is a variable that stores the number of bids received;
- **bids** is a table that stores the received bids;
- **participants()** is a function that returns the number of participants;
- **comp_win()** is a function that takes a set of bids and returns a winner value;
- **tmp** is a temporary variable.

VI. If a Buyer B1 has been notified loosing, then there is no winning buyer with a valuation lesser than that of B1.

This property corresponds to the following net, that has to be merged with the previous one:




where:

- **prev** is a variable that stores the bid value of the previous turn;
- **bids[B1]** represents the value send by agent B1

In addition, a third property was discussed:

"The number of bidders of a given type (buyer/seller) is not greater than the maximum number of allowed registrations at the market (for each buyer/seller role)"

We think that such a property cannot be translated into a Petri net for verification, since it strongly depends on the implementation of the protocol and the method for counting bidders and sellers.

# VI. Conclusion

In this deliverable we addressed two important aspect toward dynamic and open market place. While we don't suppose that agent playing in market will be developed from scratch their roles we presented here the notion of compatibility checking of an agent behaviours against a role behaviour. In order to illustrate this point we have the idea two inverse the two bidder behaviours in the two protocols ; an iterative bidder that plays a single shot market and a single shot that plays an iterative market. We have shown that in the first case the iterative bidder is compatible with the single shot market place while in the next case the system can deadlock. Focusing on the next case we present the problem of adaptation ; if one agent is incompatible with a role is it possible to generate an adapter that make it compatible. For that we have presented the model based adaptation and then we have present an adaptation methods that generates automatically the adapter behaviours. The resulted adapter of a single shot bidder in an iterative environment is component that behaves as an iterative bidder with the same value of bid sent by the single shot agent.

References :

[1] S. Haddad, T. Melliti, P. Moreaux, and S. Rampacek. A dense time semantics for Web services specifications languages. In *Proc. of the 1st Int. Conf. on Information & Communication Technologies: from Theory to Applications (ICTTA'04)*, pages 647--648, Damascus, Syria, April 19-23 2004. IEEE France.

[2] Tarek Melliti, Pascal Poizat and Sonia Ben Mokhtar. Distributed Behavioural Adaptation for the Automatic Composition of Semantic Services. in *FASE'2008 - Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science (LNCS) 4961:146-162, Springer, 2008

[3] C. Canal, J. M. Murillo, and P. Poizat. Software Adaptation. L'Objet, 12(1):9–31, 2006. *Special Issue on Software Adaptation*.

[4] M. Autili, P. Inverardi, A. Navarra, and M. Tivoli. SYNTHESIS: a tool for automatically assembling correct and distributed component-based systems. In Proc. of *ICSE'*07.

[5] Carl Adam Petri and Wolfgang Reisig (2008) Petri net. *Scholarpedia*, 3(4):6477

[6] Mazouzi H., El Fallah Seghrouchni A., Haddad S. "Open protocol design for complex interaction in Multi-Agent Systems". In the proceedings of AAMAS' 02. ACM - Publisher. Bologna, Italy, July - 2002