Describing and Reasoning on Web Services using Process Algebra

Gwen Salaün, Lucas Bordeaux, Marco Schaerf

DIS - Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italia
Email: {salaun,bordeaux,schaerf}@dis.uniroma1.it

***Abstract*— We argue that essential facets of web services, and especially those useful to understand their interaction, can be described using process-algebraic notations. Web service description and execution languages such as BPEL are essentially *process description* languages; they are based on primitives for behaviour description and message exchange which can also be found in more abstract process algebras. One legitimate question is therefore whether the formal approach and the sophisticated tools introduced for process algebra can be used to improve the effectiveness and the reliability of web service development. Our investigations suggest a positive answer, and we claim that process algebras provide a very complete and satisfactory assistance to the whole process of web service development. We show on a case study that readily available tools based on process algebra are effective at verifying that web services conform their requirements and respect properties. We advocate their use both at the *design stage* and for *reverse engineering* issues. More prospectively, we discuss how they can be helpful to tackle *choreography* issues.**

I. Introduction

Developing web services (WSs) raises many software engineering issues, some of which are new and some of which have been recurrent problems already encountered in previous programming paradigms. Implementing WSs is error-prone, because of the complex interactions and message exchanges that have to be specified. E-commerce applications also need to match the requirements expressed by their users, and it is therefore needed that these requirements be stated accurately. A more unusual specificity that distinguishes them from more traditional software components is their being accessed through the internet. WSs are distributed, independent processes which communicate with each other through the exchange of messages, and the central question in WS engineering is therefore to make a number of processes work together to perform a given task.

Due to this specificity, it has already been argued by other authors (*e.g.,* [17]) that WSs and their interaction are best described using *process description languages*. Indeed, it is clear that most emerging standards for describing and composing WSs (for instance BPEL4WS [1] - hereafter shortened to BPEL) are actually such languages. Here we advocate the benefits of more abstract notations provided by process algebra (PA) [5], [18]. Being simple, abstract and formally defined, PA make it easier to formally *specify* the message exchange between WSs, and to *reason* on the specified systems. Advanced automated tools have been developed to support both of these tasks, and it is therefore natural to investigate their usefulness for WS development.

In our opinion, abstract service descriptions based on PA can be used at several levels:

- **During the design stage,** PA can be used to describe an abstract specification of the system to be developed (a graphical front-end similar to the one described in [21] can ease this task); this gives a preliminary, validated model to be used as a reference for implementation. A mapping from the algebra into the executable language is needed for the automatic generation of code skeletons.
- **For reverse engineering purposes,** a translation in the other direction is needed to extract an algebraic description from existing web services.

Verification tools can help in both cases, either on a service to be designed or on an existing one which is reverse-engineered. These tools can be used to check whether interacting services correspond to user needs, or to detect that they fail to ensure important properties, like the absence of deadlock.

An application of the reverse engineering aspect concerns *choreographic* issues. As made explicit by the W3C choreography working group [26] it is now accepted that, in a near future, the interface of WSs should evolve and that a description of their *observable behaviour* should be provided in addition to their sole WSDL interface. This description will be based on an XML-based standard like WSCI [25] or BPEL [1]. As a particular case of our reverse engineering proposal, extracting a process-algebraic descriptions of services would enable us to reason on choreographic problems.

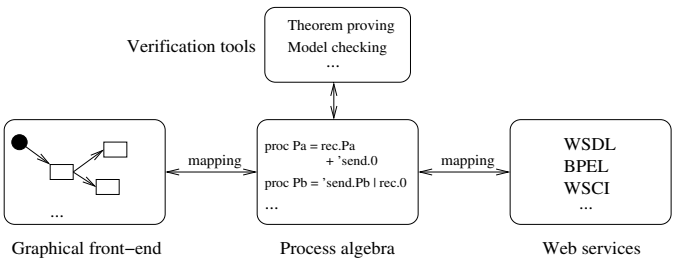

Fig. 1. Proposal overview

A global picture of the approach we envision for the use of PA in a WS setting is shown in Fig. 1. Central to our

Proceedings of the IEEE International Conference on Web Services (ICWS'04)
0-7695-2167-3/04 $ 20.00 IEEE

approach is the need for a sound, two-way mapping between the language used by the automated reasoning tools and the languages that the industrial standards will impose. The latter will typically be XML-based languages like BPEL or WSCI, while the tools need a more abstract, process-algebraic representation.

This paper is structured as follows. In Section II, we introduce PA and the way we use them to describe and reason on web services. Section III shows how a practical case study (a sanitary agency service) is described using the CCS PA, and how the CWB-NC tool is used to ensure some properties, like the correctness of a composition. Section IV gives some guidelines to map abstract descriptions from/into more concrete ones; we especially illustrate this point with CCS and BPEL. Related work is discussed in Section V and concluding remarks are drawn up in Section VI.

II. Process Algebra for Describing and Reasoning on Web Services

PAs [5] are formal description techniques to specify software systems, particularly those constituted of concurrent and communicating components. Numerous PAs have been proposed; well-known calculi are CCS [18], ACP [3], CSP [11] and all their extension like the π-calculus [22], LOTOS [6] or Timed CSP [24]. These languages share the same ingredients: simple contructs to describe dynamic behaviours, compositional modelling, operational (and/or axiomatic, denotational) semantics, behavioural reasoning via model checking and process equivalences.

A. What is a Process Algebra?

In this subsection, our purpose is to give a flavour of what a PA is made up of. The algebra chosen in this paper is CCS [18] whose set of operators is small yet sufficiently expressive for the presentation of our approach.

Defining processes in CCS requires to first agree on a set of *action names* which represent the messages used in the system, for instance $\{$*askProductInfo, buy, cancel*$\}$, or $\{a, b, c\}$. The basic *actions* in CCS are to *receive* a message (this is noted by simply writing its name) ot to *emit* a message (this is written by its name prefixed by the quote symbol, *e.g.,* $'a$).

Processes are constructed as follows. A process which is terminated is written 0: "do nothing". A process can execute a *sequence* of the form $a.P$, where a is an action and P a process: "execute a then P". A process can perform a *nondeterministic choice* ($P + Q$: "execute P or execute Q"). The coexistence of several processes $P_1, \ldots, P_n$ whose execution is interleaved is written $P_1 | \ldots | P_n$ ("run $P_1, \ldots, P_n$ in parallel"). The "|" symbol, called *parallel composition*, is therefore used to define a global process made up of several subprocesses. Last, the *restriction* operator, noted $P \backslash sm$ where P is a process and sm is a set of names, imposes that an emission of m ($m \in sm$) by one subprocess of P can occur only if another subprocess does a reception of the same message name (synchronization). In practice, it is used to declare all the names on which synchronizations occur in the whole system. Here is an example of CCS process:

$$proc\ S = ((b.a.0 + c.a.0) | 'a.'c.0) \backslash \{a\}$$

This process is the parallel composition of two simple behaviours, and it imposes that the sending of message a be synchronous due to the restriction. The first subprocess can evolve either through the sequence of input messages $b.a$ ("receive a then receive b") or through the sequence $c.a$. The second can perform the sequence of output messages $'a.'c$.

We additionally need a symbol τ, denoting hidden actions. It is basically needed because processes can be described at different levels of abstraction, and a τ action can be used by an abstract process to hide some actions performed by a more concrete equivalent process. Practically, a τ action occurs when two processes synchronize together on a concrete action.

CCS is formalised using axiomatic and operational semantics [18]. The operational semantics describes the possible evolutions of a process; more precisely, it defines a transition relation $P \xrightarrow{\alpha} P'$ meaning intuitively that "P can evolve to P' in one step through action α" (where α is the emission of a message, the reception of a message or a τ action). This relation is defined by the set of rules presented below[1], which give a precise meaning to each operator.

$$\frac{}{\alpha.F \xrightarrow{\alpha} F}\ SEQ \qquad \frac{F \xrightarrow{\alpha} F'}{F+G \xrightarrow{\alpha} F'}\ NDC$$

$$\frac{F \xrightarrow{\alpha} F' \quad \alpha \notin L}{F \backslash L \xrightarrow{\alpha} F' \backslash L}\ RES \qquad \frac{F \xrightarrow{\alpha} F'}{F|G \xrightarrow{\alpha} F'|G}\ PC1$$

$$\frac{F \xrightarrow{a} F' \quad G \xrightarrow{'a} G'}{F|G \xrightarrow{\tau} F'|G'}\ PC2$$

For instance, the SEQ rule states that a process $\alpha.F$ can evolve to a process F by performing α, and the NDC rule states that a process involving a choice can evolve following one of the processes of the choice. Let us illustrate how these rules can be used to figure out a possible trace starting from the behaviour S introduced before. Intuitively, the trace below is obtained by the application of the sequence on the b action (the other possible evolution would be to fire the c action), then both processes evolve through a synchronization on a (the visible result is a τ action), and finally $'c$ is performed to complete the behaviour.

$$S \xrightarrow{b} (a.0|'a.'c.0)\backslash\{a\} \xrightarrow{\tau} ('c.0)\backslash\{a\} \xrightarrow{'c} 0$$

B. Specifying Web Services as Processes

We already claimed that WSs are essentially processes and that any meaningful representation of services should take into account behavioural information (this need has already been pointed out by several people in the community [17], [4], [12], [1]). An essential reason behind this need is that the interaction between WSs is typically more complex than, for instance, simple *(Remote) Procedure Calls*. Knowing the signature of

[1] Both "|" and + are commutative, the symmetrical rules are omitted here.

the parameters expected by a RPC is essentially sufficient to use it (a WSDL file would in this case be sufficient). On the contrary, even very simple services like the basic hotel booking service shown on Fig. 2 involve more complex interactions, since the service can dialogue with the caller, for instance to ask for complementary information.

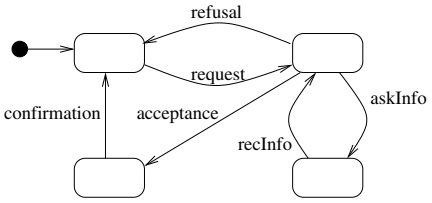

Fig. 2. A WS for a hotel

Clearly, it is not sufficient to know that this service receives three types of messages (booking requests, confirmation and extra information) to understand the way it interacts with other services, because its behaviour may follow different scenarii depending whether free rooms are available, whether the reservation form was filled correctly, etc. One needs to know that the booking request comes first, and that it can be followed by a number of requests for more information, by a refusal or by a confirmation.

PAs are an effective and unambiguous way to represent such behaviours. In fact, the graphical notation used in the previous example is easily translated into the following CCS expression (the following syntax is directly accepted by the CWB-NC tool [8] and is also used in Section III):

```
proc Hotel = request.InteractionLoop

proc InteractionLoop =
    'askInfo.recInfo.InteractionLoop
    + (  'refusal.Hotel
       +'acceptance.confirmation.Hotel))
```

The textual syntax is more adapted to proof-writing and formal reasoning, as well as to the description of large-scale systems. Graphical notations are anyway complementary and can be used by user-friendly front-ends [21]. We use them in this section for illustration purposes.

We note that the abstract descriptions we consider here are *internal* since we describe the body of WSs. External interfaces in their current acceptation (protocols and signatures of the messages received and emitted by the service, as specified in the WSDL files) can be easily extracted from these descriptions. However, the need for more expressive interfaces is now recognized, because of choreography issues. In a near future, processes could be viewed as interfaces abstracting away from part of the implementation details.

When describing software, the question of the level of details reflected by the abstract description always arises, and WSs are no exception. Behavioural aspects are unavoidable, they are the *strict minimum* of what needs be represented. In our opinion, there is usually little need to model implementation or networking issues (*e.g.,* server location, URL or URI) because these concepts are too low-level and do not add to the global understanding of a system. But in some cases, other features like data or time allow to have a more faithful representation of a service. In all cases where messages and action names are enough, simple algebras like CCS can be recommanded. Richer languages exist which can be used, for instance, to address the following needs:

• **Abstract data** can be manipulated by algebras like LOTOS [6]. They are used if there is a need to describe values of message parameters, local data (attached to a specific WS), or predicates guarding a piece of behaviour. In the case of the business trip, we can imagine a WS whose task is to book a hotel (among several ones) while respecting a limit price which guards the application of some of the actions (Fig. 3).

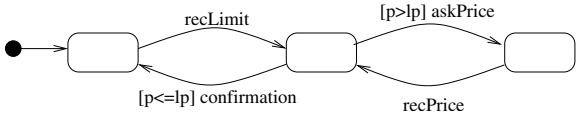

Fig. 3. Booking a hotel respecting a limit price

• **Timed PAs** like Timed CSP [24] allow the description of temporal constraints. As an illustration, it is easy to specify a waiting time for a specific event, and to verify afterwards that the concerned web service will never wait for a response for longer than the specified duration. This could be useful, for instance, for a hotel which does not wish to wait for more than one day for a confirmation (Fig. 4).

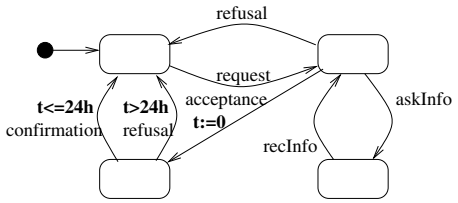

Fig. 4. A WS for a hotel with a confirmation delay

We last mention briefly that other PAs have been proposed for specifying many other aspects like stochastic behaviour, fault tolerance, mobility (particularly for dynamic evolution and reconfiguration), etc. Thereby, PA offers a great set of possible languages suitable to specify basic WSs, as well as more advanced ones involving different constructs depending on the application domain. The last reason, and not the least, to consider the use of PA is their well-foundedness, their formality, the experience acquired in this domain through some 25 research years, and therefore the existence of powerful reasoning tools issued from this theoretical and practical effort.

C. Automated Reasoning on Web Services

A major interest of using abstract languages grounded on a clear semantics is that automated tools can be used to check that a system matches its requirements and works properly. Specifically, these tools can help 1) checking that two processes are in some precise sense *equivalent* – one process is typically a very abstract one expressing the specification of the problem, while the other is closer to the implementation level; 2) checking that a process verifies desirable *properties* – *e.g.,* the property that the system will never reach some

unexpected state. Revealing that the composition of a number of existing services does not match an abstract specification of what is desired, or that it violates a property which is absolutely needed can be helpful to correct a design or to diagnose bugs in an existing service. We introduce the two general classes of techniques used for verifying processes; the case study we provide in Section III will illustrate their use.

1) Verifying Equivalences: Intuitively, two processes or services are considered to be equivalent if they are *indistinguishable* from the viewpoint of an external observer interacting with them. This notion has been formally defined in the PA community, and several notions of equivalence have been proposed. Here we briefly illustrate them in order for the reader to be aware of some of their subtleties relevant to the context of WSs; more on the subject can be found in [18].

A first approach is to consider two processes to be equivalent if the set of *traces* they can produce is the same (*trace-equivalence*). For instance, the possible executions of the processes $a.(b.0+c.0)$ and $(a.b.0+a.c.0)$ are shown in Fig. 5 part (A), where messages a, b and c can be respectively understood as requests for reservation, editing data and cancellation. Both of these two processes will have $a.b$ and $a.c$ as possible traces: they will either receive the messages a then b, or a then c.

Nevertheless, it is not fully satisfactory to consider these two processes equivalent since they exhibit the following subtle difference. After receiving message a, the first process will accept either message b or c. The second process behaves differently: on receiving message a, it will either choose to move to a state where it expects message b, or to a state where it expects message c. Depending on the choice it makes, it will not accept one of the messages whereas the first process leaves both possibilities open. The second process does not guarantee that a request for reservation (a) followed by *e.g.,* cancellation (c) will be handled correctly (c might not be received if the process has chosen the left-hand side branch). The notion of equivalence called *bisimulation* [18] is a refinement of trace equivalence which takes these differences into account.

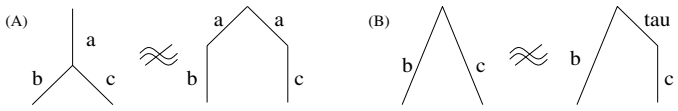

Fig. 5. Classical examples of processes not observationally equivalent. For instance, a represents a request for hotel reservation, b asks for the information regarding the booking, while c represents cancellation.

Further subtleties arise when one has a partial knowledge of the behaviour of a process. This may happen for two reasons: 1) during the design stage, where the specification which is being defined is necessarily abstract and incomplete; 2) when one finds or reuses an existing WS, and only an interface or a partial description hiding private details is available. This is expressed in CCS using the τ symbol, which states that hidden actions take place. τ actions must be taken into account when reasoning on the equivalence of two processes, as evidenced by Fig. 5 part (B). Both of the processes depicted here can receive b (edition of reservation data) or c (cancellation). But whereas the first one can receive any of the two, the second one can choose to first execute some unobservable actions which will lead it to a state where it can only receive message c. Once again it cannot be guaranteed that the second service will accept cancellation requests, and this depends on some decisions it takes secretly.

Weak[2] bisimulation (*a.k.a.* observational equivalence) is therefore widely acknowledged as the finest and most appropriate notion of process equivalence, and is implemented in tools like CWB-NC, which can automatically check that two algebraic expressions denote the same observational behaviour.

2) Verifying Properties: The properties of interest in concurrent systems typically involve reasoning on the possible scenarii that the system can go through. An established formalism for expressing such properties is given by *temporal logics*[3] like CTL$\star$ [14]. These logics present constructs allowing to state in a formal way that, for instance, all scenarii will respect some property at every step, or that some particular event will eventually happen, and so on.

An introduction to temporal logic goes beyond the aims of this paper, but it suffices to say that a number of classical properties typically appear as patterns in many applications. Reusing them diminishes the need to learn all subtleties of a new formalism. The most noticeable examples are:

- **Safety properties**, which state that an undesirable situation will never arise. For instance, the requirements can forbid that the system reserves a room without having received the credit information from the bank;
- **Liveness properties**, which state that some actions will always be followed by some reactions; a typical example is to check that every request for a room will be acknowledged.

The techniques used to check whether a system described in process-algebraic notations respects temporal logic properties are referred to as *model checking* methods [7].

III. Case Study: the Sanitary Agency

In this section, the goal is to illustrate how PA can be used to specify concrete WS problems and to reason about such interacting processes using existing verification tools. Therefore, this section should be viewed as disconnected of any development way (design approach or reverse engineering). We experiment our approach on a sanitary system. In this study, we chose CCS and the CWB-NC tool.

A. Informal Requirements

This case study is related to the field of public welfare and extracted from a larger domain analysis concerning the local government of Trentino (Italy). It was first dealt with in [23]. This software system aims at supporting elderly citizens in receiving sanitary assistance from the public administration. This problem involves several actors. First, we formalize the

[2]Another notion called *strong bisimulation* exists. It is nevertheless too restrictive in our context because it imposes a strict matching of the τ actions. Also note the notion of *congruence*, an observational equivalence which should be preferred when one wants processes to be equivalent *in any context*, *i.e.,* in all possible systems using them.

[3]This name should not give the impression that these logics introduce a quantitative notion of time, they deal with the future behaviour of a system.

service to be worked out as a citizen behaviour. A citizen posts a request, exchanges information with the agency, waits for a response, and if accepted receives a service and pays fees.

```
proc Citizen =
 'request.askInf.'provInf.
    ( refusal.Citizen
      + acceptance.
        ( provT.'paymentPrivateFee.Citizen
          + provM.'paymentPrivateFee.Citizen))
```

The goal of the next description is to figure out a right composition of WSs ensuring such a request. More precisely, the solution involving several processes (abstracting away from internal interactions) should be equivalent to the request modelled here as a citizen behaviour.

B. Description and Composition

The whole system is composed of services involved when elderly people apply for a sanitary assistance: the sanitary agency satisfying requests, the transportation service, the meal delivery, the bank managing funds. We start with a view of all the processes (boxes in Fig. 6) involved in this system as well as synchronizations between these entities. Synchronizations are denoted using lines joining the involved agents with actions and their direction (prefixed by a quote for emissions).

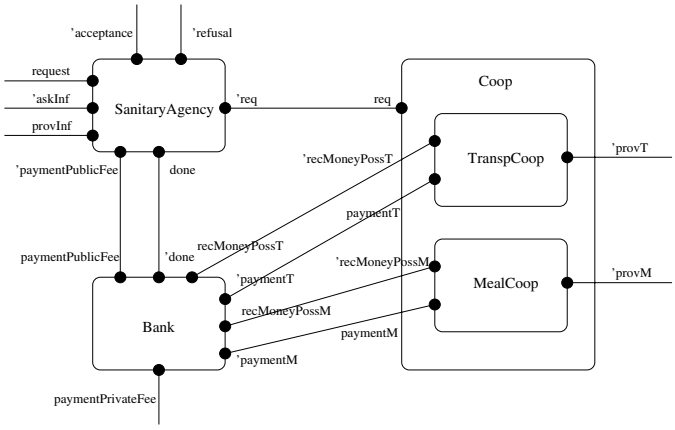

Fig. 6. Synchronizations in the sanitary agency

A sanitary agency has to manage requests submitted by elderly citizens. First, it asks some information to the citizen who has posted the request. Depending on that, it sends either a refusal and waits for a new request (as reflected by a recursive definition), or the request is accepted. In the latter case, a synchronization is performed with the cooperative controller (it controls in some way both cooperatives) to order the delivery of concrete (transportation or meal) services. Then, the agency pays some public fees to the bank and waits from the bank component for a signal indicating that the transaction[4] is completed.

```
proc SanitaryAgency =
  request.'askInf.provInf.
    ( 'refusal.SanitaryAgency
      +   ('acceptance.'req.'paymentPublicFee.
           done.SanitaryAgency ))
```

[4]A transaction could be defined as a complete execution of all our interacting WSs to solve a precise task.

Now, we introduce the behaviour of the cooperative. The transportation (resp. meal) cooperative provides its service, warns the bank to start the payment, and waits for the reception of its fees. A controller, called `Coop`, receives a notification of the agency and launches one of the possible services.

```
proc TranspCoop =
     'provT.'recMoneyPossT.paymentT.Coop

proc Coop =
     req.(TranspCoop+MealCoop)
```

The bank receives a firing message from the cooperative involved in the current request. The different payments are then performed: payment of public fees by the agency and of private fees by the citizen, payment of the cooperative by the bank. Note that the specified strategy for this service is to collect the money first and only then to pay for the service.

```
proc Bank =
  recMoneyPossT.(
        (paymentPrivateFee.paymentPublicFee.
         'paymentT.'done.Bank)
      + (paymentPublicFee.paymentPrivateFee.
         'paymentT.'done.Bank ))
  + recMoneyPossM.(   ...
```

In this experimentation, it is worth noting that CCS is enough to describe the main requirements of the case study while remaining at an abstract level. As mentioned previously and if needed, we could use a more expressive language. Otherwise, the binary and synchronous communication model turns out to be enough to specify communications in this example. In the current specification, the modelling of a citizen as a process could be criticized because his/her behaviour does not belong to the system but to the environment. Nevertheless, its representation is necessary in case of simulation (closed system). Finally, the solution at hand is sequential (treatment of one request after the other) and not concurrent (several requests could be posted and treated at the same time). This model is close to WS issues where a precise task has to be worked out by orchestrating different distributed instances of computational components (notion of transaction).

C. Reasoning

In this subsection, our goal is to show how reasoning tools (CWB-NC here) could help practitioners to reason on interacting WSs. The first steps (syntactical parsing when loading the file containing the processes definitions, finding deadlock, simulation) have enabled ourselves to raise some problems: typing error, missing synchronizations, incomplete behaviours, etc. Roughly speaking, these early checkings are useful to correct the coarse mistakes in design or writing. Each of this basic verification steps should be carried out again every time the specification is modified. We show below definitions needed to simulate an example of system. Herein, we illustrate how to build a closed system involving a single citizen. All the actions should appear in the restriction set.

```
set closedRestSet4S =
```

```
 { req, done, request, askInf, provInf, provM,
   provT, paymentT, paymentM, paymentPublicFee,
   paymentPrivateFee, recMoneyPossT,
   recMoneyPossM, refusal, acceptance }

proc ClosedSystem =
   (SanitaryAgency | Coop
    | Bank | Citizen) \ closedRestSet4S
```

Afterwards, the purpose is to ensure that our proposed implementation made up of three interacting components (the agency, the cooperative controller and the bank) is observational-equivalent to the initial task to be solved (expressed by the citizen behaviour). To prove such an equivalence, we first define the reverse behaviour of the request (that is the reverse of the citizen process). This is due to the CCS synchronization mechanism which holds on complementary actions. Therefore, the request is the complementary behaviour of the system, because one citizen interacts with the system on opposite messages.

```
proc RCitizen =
  request.'askInf.provInf.(
      'refusal.RCitizen
    + 'acceptance.(
          'provT.paymentPrivateFee.RCitizen
        + 'provM.paymentPrivateFee.RCitizen))
```

Hence, our aim is to prove using CWB-NC that both `RCitizen` and `System` are equivalent, that is `eq -S obseq RCitizen System`.

```
set restSet4S =
  { req, done, paymentPublicFee, paymentT,
    paymentM, recMoneyPossT, recMoneyPossM }

proc System =
   (SanitaryAgency | Coop | Bank) \ restSet4S
```

Several versions of the global system have been refined until the final version introduced in the previous subsection was obtained. Let us summarize our main errors to give an insight of the usefulness of automated equivalence checkings. Our first mistake was to not consider the notion of transaction. It is necessary to take into account that a task needs a sole execution of each process to be carried out. This kind of behaviour can be specified using non recursive processes, or by enhancing their behaviour so that they agree at the end of a transaction. The latter case was written down adding the `done` action in the `SanitaryAgency` and `Bank` definitions. The second error was due to a very general definition of the bank service. The initial behaviour defined different payments without more constraints (*e.g.,* it could accept several successive payments from a citizen at any moment). To ensure the equivalence, we added a synchronization between the cooperatives and the bank so that payments be performed only after the delivery of the service. At this level, trace equivalence is verified but not the observational one. Indeed, a prior specification proposed a double possible synchronizations on `reqT` or `reqM` to fire respectively the transportation or meal services. Then, a single synchronization between the sanitary agency and the cooperative controller leaded to the obtention of the observational equivalence (see Fig. 7 for a capture of the problem at hand). For this last specification, all the basic verification steps (simulation, no deadlocks, etc.) are satisfactory.

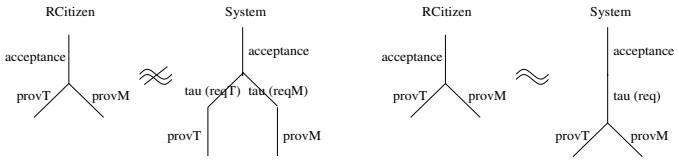

Fig. 7. Processes not observational-equivalent (left) and the solution (right)

The last reasoning we used was to prove some temporal properties of the system. Due to the simplicity of the case study, there are no critical properties to be satisfied. We illustrate below the CWB-NC automatic capability with a couple of properties also guaranteed in [23]. They are written out in CTL$\star$. The first one attests that a request can be done in one possible scenario. The second formula expresses that the firing of the output action `acceptance` is followed either by a meal delivery or by a transportation assistance.

```
prop can_request_ctl = EF <request>tt

prop MorTafterAcceptance =
   AG (  (not ['acceptance]tt) \/
         (AF (['provT]tt \/ ['provM]tt))  )
```

IV. Correspondence between Abstract Processes and Executable Applications

As mentioned before, the goal is to develop WSs from an abstract description or to extract a more abstract representation from an XML-based implementation. As far as the choreography issue is concerned, we remind that abstract processes would be seen as public interfaces which are abstract representations of black-box WSs. Ideally, we have to ensure a semantic-compatible equivalence between abstract specifications and executable implementations. However, this formal two-way mapping is difficult to guarantee (especially due to the absence of a well-admitted semantics for BPEL here). In this section, and as a first contribution in this direction, we give some guidelines for the mapping between CCS processes and BPEL code built on top of WSDL interfaces. This goal is reached by identifying similar concepts at both abstraction levels. We illustrate these links by showing pieces of the BPEL implementation for the sanitary agency system. The reader is referred to [1] for a description of BPEL.

First of all, some basic notions are defined in WSDL. The action/message notion in PA finds its equivalent in the `message` tag. WSDL operations refer to transmission/communication primitives; four basic primitives exist in WSDL: one-way (reception), notification (emission), request-response (two-way communication starting by a reception) and solicit-response (two-way communication starting by an emission). BPEL makes it possible to describe three of these constructs. A one-way `reply` (resp. `receive`) matches a CCS emission (resp. reception). Synchronizations between WSs are described through the `invoke` construct (request-response primitive).

Let us continue with the BPEL language. The process notion is shared between both description levels. An overall

activity is completed when the end of its behaviour is reached (no explicit construct unlike the termination denoted by 0 in CCS). Agent recursion (corresponding to the repetition of their behaviour) could be represented using a `while` activity. However, we emphasize that a specific abstract agent works out a precise task required by a more complex transaction, and consequently does not need this notion of recursivity at the implementation level. Such a call of a specific piece of code is often encoded as a creation of a new instance.

Sequence and nondeterministic choice have straightforward counterparts in CCS and BPEL (`sequence` and `pick` constructs in BPEL). In case of a deterministic choice described as a `switch` construct, we should use the same CCS choice operator. Let us remark that less (resp. more) details could appear at the abstract (resp. executable) level. As an example, see the condition expressed in the BPEL code below determining whether the request is accepted or not. Concurrent web services synchronizing themselves through message exchanges (particularly using the `invoke` operation) are declared as CCS parallel compositions. In a same WS, the `flow` activity enables one to implement concurrency and synchronization.

Now, we show an excerpt of the BPEL code corresponding to the sanitary agency WS as written out before in CCS. Below, the behaviour of the `SanitaryAgency` is briefly introduced (lots of XML details have been removed for lack of space). We can emphasize the reception of a request (initiating from a customer), the exchange of information, the refusal or acceptance of the service. All the WSDL and BPEL files needed for an executable description of our case study can be consulted there: `http://www.dis.uniroma1.it/~salaun/PA4WS`. Experimentations using the eclipse IDE and the collaxa plug-in for BPEL are still in progress.

```
<process name="SanitaryAgencyProcess"
...
   <sequence>
        <receive name="receiveRequest"
                    partnerLink="Citizen"
                    portType="cit:requestPT"
                    operation="requestO"
                    variable="request"
                    createInstance="yes">
        </receive>
        <invoke name="invokeCitizen"
                   ...
        </invoke>
     <switch>
     <case condition="cit:getContainerData(
           'provInfC', 'aCondition') = 'true'">
        <sequence>
           <reply name="replyAcceptance"
                     ...
           </reply>
           ...
        </sequence>
     </case>
     <otherwise>
        <reply name="replyRefusal"
                   ...
        </reply>
     </otherwise>
     </switch>
   </sequence>
</process>
```

We stress that some notions do not directly appear in BPEL. This is the case for the τ action and for the CCS restriction operator. However, the CCS notion of synchronization (especially using restriction and τ actions) finds its equivalent in WSDL communication primitives. Furthermore, we can imagine that the names needed in the CCS restriction set could be easily extracted from the WSDL files.

V. Related Work

In this paper, we advocate the use of PA to describe and compose WSs at an abstract level. As previously motivated, this abstract specification level can be connected to the executable level (system development or reverse engineering). The application layer is made up of all the existing standards. On top of the WSDL and SOAP well-known standards, different XML-based languages for WS orchestration/composition have already been proposed, like DAML-S [2]. An agreement is emerging to favour the BPEL execution language at this level. At a more abstract level, many of the WS abstract descriptions are semi-formal (no formal meaning) and are therefore error-prone and not supported by tools other than editors. Some more formal proposals have emerged, grounded for most of them on transition system models (labelled transition systems, Mealy automata, Petri nets, etc.) [4], [12], [20], [10], [13]. Our approach could be viewed as an alternative proposal. Compared to these proposals, PAs are adequate to describe web services, because they are formal, based on simple but expressive enough operators, and equipped with tools to support the design. Additionally, their constructs are adequate to specify composition due to their compositionality property.

Other related works are proposals dedicated to verifying WS description to ensure some properties. Some representative proposals following this idea are [20], [9], [19]. Summarizing these works, they use model checking to verify some properties of cooperating web services described using XML-based languages (DAML-S, WSFL, WSDL, BPEL, ConGolog). Accordingly, they abstract their representation and ensure some properties using ad-hoc (KarmaSIM) or existing (like SPIN, LTSA) tools. As far as the composition issue is concerned, different techniques have emerged which ensure a correct composition such as automatic composition [4], [16], planning [15], [13] or model deduction [20]. However, most of the existing proposals do not ensure this composition correctness [9], [12], [10]. Our contribution is the use of powerful proof theory accompanying PA in general, particularly the use of (weak) bisimulation is helpful for ensuring correct composition, *i.e.,* to readily verify equivalences between possible requests and composite WS. On a wider scale, usual system properties (deadlock, safety, liveness) can be also verified.

Last but not least are works connecting abstract representations of WSs with XML-based standards. As far as we know, a couple of works [9], [23] proposes some general guidelines to connect abstract constructs (state diagrams and Formal Tropos specifications[5]) and BPEL. So far, we cannot argue to achieve

[5]Formal Tropos mixes class structures and temporal formulae to specify early requirements.

a better result, but we already propose precise guidelines. In our opinion, a formal correspondence is essential at this level to preserve properties proved on the abstract specification, and to extract automatically a semantic-compatible abstract description from an executable implementation.

In comparison to these existing works, the strenght of our approach is to work out all these issues (description, composition, reasoning) at an abstract description level, based on the use of existing approaches and tools, while keeping a two-way connection with the application layer. Furthermore, we advocate a general approach which is not restricted to a single description technique or reasoning tool.

VI. Concluding Remarks

WSs are an emerging and promising area involving important technological challenges. Some of the main challenges are to correctly describe WSs, to compose them adequately and/or automatically, and to discover suitable WSs working out a given problem. In this paper, we propose a framework to develop and reason about WSs at an abstract level. The usefulness of such a representation is manifold: (i) designing an application based on WSs, (ii) applying reverse engineering to reason on deployed WSs, (iii) considering such abstract processes as WS public interfaces and therefore dealing with them for issues like composition. We advocate the use of PA as an abstract representation means to describe, compose and reason (simulation, property verification, correctness of composition) on WSs. In addition, links have been defined between abstract and concrete descriptions. We claim that our approach is general because we do not focus on a specific language, but on a whole family of formal description techniques. Furthermore, all the theoretical foundations underlying PA are adequate to describe and reason on services, especially to ensure correct composition as illustrated through the sanitary agency case study.

A first continuation of our current work is to experiment the use of process algebras involving more advanced constructs like LOTOS or the π-calculus. Indeed, it sounds realistic to represent exchanges of data between processes (for negotiation means as an example) or to need dynamic reconfiguration. Another direction is the automatic composition of WSs. In the current work, equivalence is only used to ensure a correct composition. A possible idea could be to complement the bisimulation theory to add automatic capabilities to the current framework. A last perspective is to intensively work on the formalisation of connections between abstract and concrete descriptions. Some ideas in this direction have been sketched and should be studied thoroughly.

Acknowledgements

This work is partially supported by Project ASTRO funded by the Italian Ministry for Research under the FIRB framework (funds for basic research).

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Specification: Business Process Execution Language for Web Services Version 1.1. 2003. Available at `http://www-106.ibm.com/developerworks/library/ws-bpel/`.

[2] DAML-S Coalition: A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. DAML-S: Web Service Description for the Semantic Web. In *Proc. of ISWC'02*, volume 2342 of *LNCS*, pages 348–363, Italy, 2002. Springer-Verlag.

[3] J. C. M. Baeten and W. P. Weijand. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1990.

[4] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of E-services That Export Their Behavior. In *Proc. of ICSOC'03*, volume 2910 of *LNCS*, pages 43–58, Italy, 2003. Springer-Verlag.

[5] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.

[6] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. In *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-Holland, 1989.

[7] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.

[8] R. Cleaveland, T. Li, and S. Sims. *The Concurrency Workbench of the New Century (Version 1.2)*. Department of Computer Science, North Carolina State University, 2000.

[9] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *Proc. of ASE'03*, pages 152–163, Canada, 2003. IEEE Computer Society Press.

[10] R. Hamadi and B. Benatallah. A Petri Net-based Model for Web Service Composition. In *Proc. of ADC'03*, volume 17 of *CRPIT*, Australia, 2003. Australian Computer Society.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1984.

[12] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: a Look Behind the Curtain. In *Proc. of PODS'03*, pages 1–14, USA, 2003. ACM Press.

[13] A. Lazovik, M. Aiello, and M. P. Papazoglou. Planning and Monitoring the Execution of Web Service Requests. In *Proc. of ICSOC'03*, volume 2910 of *LNCS*, pages 335–350, Italy, 2003. Springer-Verlag.

[14] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

[15] S. A. McIlraith and T. C. Son. Adapting Golog for Composition of Semantic Web Services. In *Proc. of KR'02*, pages 482–496, France, 2002. Morgan Kaufmann Publishers.

[16] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing Web Services on the Semantic Web. *The VLDB Journal*, 12(4):333–351, 2003.

[17] G. Meredith and S. Bjorg. Contracts and Types. *Communications of the ACM*, 46(10):41–47, 2003.

[18] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[19] S. Nakajima. Model-checking Verification for Reliable Web Service. In *Proc. of OOWS'02, satellite event of OOPSLA'02*, USA, 2002.

[20] S. Narayanan and S. McIlraith. Analysis and Simulation of Web Services. *Computer Networks*, 42(5):675–693, 2003.

[21] M. Y. Ng and M. Butler. Tool Support for Visualizing CSP in UML. In *Proc. of ICFEM'02*, volume 2495 of *LNCS*, pages 287–298, China, 2002. Springer-Verlag.

[22] J. Parrow. An Introduction to the π-Calculus. In *[5]*, chapter 8, pages 479–543. Elsevier, 2001.

[23] M. Pistore, M. Roveri, and P. Busetta. Requirements-Driven Verification of Web Services. In *Proc. of WS-FM'04*, Italy, 2004.

[24] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and Practice. In *Proc. of REX Workshop on Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 640–675, Germany, 1992. Springer.

[25] W3C. *Web Services Choreography Interface 1.0*. Available at `http:www.w3.org/TR/wsci`.

[26] W3C. *Web Services Choreography Requirements 1.0 (draft)*. Available at `http:www.w3.org/TR/ws-chor-reqs`.