2004 Society for Design and Process Science
Printed in the United States of America

FINITE STATE AUTOMATA AS CONCEPTUAL MODEL FOR *E*-SERVICES

Daniela Berardi,
Fabio De Rosa,
Luca De Santis,
Massimo Mecella
Dipartimento di Informatica e Sistemistica - Università di Roma "La Sapienza"
via Salaria 113, I-00198 Roma, Italy
E-mail: **{**berardi,derosa,lucads,mecella}@dis.uniroma1.it

Recently, a plethora of languages for modeling and specifying different facets of e-Services have been proposed, and some of them provide constructs for representing time. Time is needed in many contexts to correctly capture the dynamics of transactions and of composability between e-Services. However, to the best of our knowledge, all the proposed languages for representing e-Service behavior and temporal constraints lack both a clear semantics and an underlying conceptual model. In this paper, we propose a conceptual representation of e-Service behavior, taking time constraints into account, and a new XML-based language, namely WSTL (WEB SERVICE TRANSITION LANGUAGE), that integrates well with standard languages in order to completely specify e-Services. In particular, WSTL allows for specifying an e-Service starting from its conceptual representation, in a straightforward way.

Keywords: *e-Services, Timed e-Services, Web Service Transition Language (WSTL), conceptual model for e-Services, e-Service Behavior.*

1. Introduction

Since the last few years, we are witnessing a great change in business paradigms. Different companies are able to pool together their services, in order to offer more complex, value added products and services. Thanks to the spreading of network technologies and the Internet, that makes services easily accessible to a vast number of customers, companies are able to cooperate in very flexible ways, giving rise to the so called *virtual enterprises* (Georgakopoulos, 1999). Although it started in the business context, the scenario of inter-organization cooperation has spread out different contexts, e.g., *e*-Government (Elmagarmid *et al.*, 2001).

Inter-organization cooperation can be supported by Cooperative Information Systems (CISs) (De Michelis *et al.*, 1997). Many approaches have been proposed for the design and development of CISs: business process coordination and service-based systems (Dayal *et al.*, 2001), agent-based technologies and systems (Castro *et al.*, 2002), schema and data integration techniques (Rahm *et al.*, 2001), (Lenzerini, 2002). In particular, the former approach focuses on cooperation among different organizations that export services as semantically defined functionalities; cooperation is achieved by composing and integrating services over the Web. Such services, usually referred to as *e*-Services or Web Services, are available to users or other applications and allow them to gather data or to perform some tasks. In the following, we will refer to *Service Oriented Computing (SOC)* as a new emerging model for distributed computing enabling the construction of agile networks of collaborating business applications distributed within and across organizational boundaries[1].

Cooperation of *e*-Services poses many interesting challenges regarding, in particular, composability,

[1] The Service Oriented Computing net: `http://www.eusoc.net`

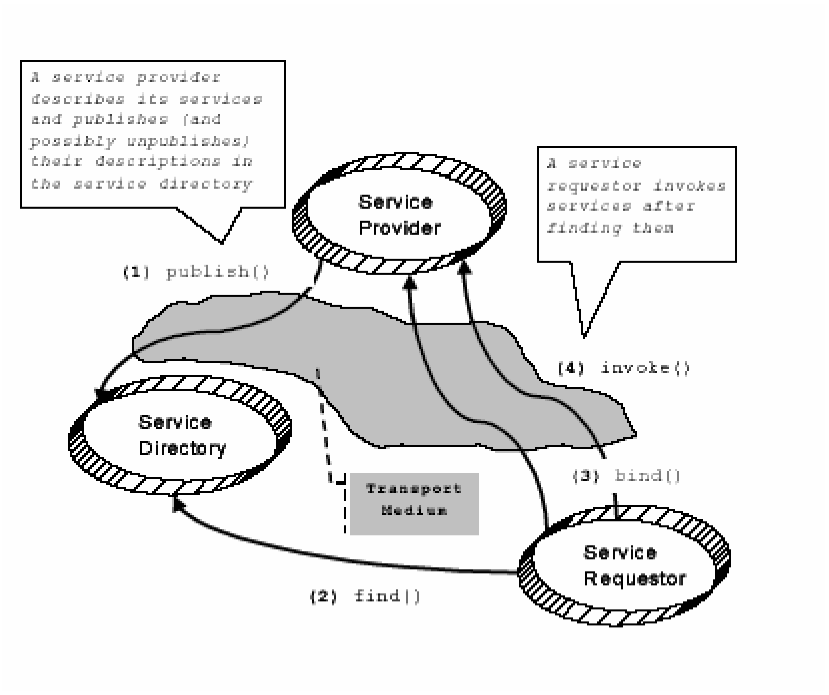
A service provider describes its services and publishes (and possibly unpublishes) their descriptions in the service directory
Service Provider
A service requestor invokes services after finding them
(1) publish()
(4) invoke()
Service Directory
Transport Medium
(3) bind()
Service Requestor
(2) find()

Figure 1. Service Oriented Architecture

synchronization, coordination, and correctness verification (Yang *et al.*, 2002). Indeed, in order to address such issues in an effective and correct way, the *dynamic behavior* of *e*-Services needs to be formally represented. There have been some preliminary efforts in this direction: (Mecella *et al.*, 2001) defines the notion of compatibility and dynamic substitution of *e*-Services, (Kuno *et al.*, 2001), (BEA *et al.*, 2002) propose standards to describe the interfaces and the conversations of *e*-Services.

Recently, some research efforts are concerned with adding a fundamental concept: time. Time is needed in many contexts to correctly capture the dynamics of transactions and of composability. Many *e*-Services available on the Internet are made up of sessions having an associated time-out, as those that allow a user to check his bank account, or to participate to an online auction. Analogously, we often face trade-offs involving time: when we want a fast, high-quality service, but no *e*-Service has both these features, we have to choose between speed and quality.

In this paper we propose a conceptual representation of *e*-Service behavior that takes time into account, and introduce an XML-based language that can be integrated with standard languages in order to completely specify *e*-Services. In particular, the proposed language allows for specifying an *e*-Service starting from its conceptual representation in a straightforward way.

2. Languages for *e*-Services

Recently, a plethora of languages for modeling and specifying different facets of service-oriented computing have been proposed; in this section, we discuss some of them, with respect to the concept of Service-oriented Architecture (SOA) (HP, 2001), in order to correctly compare all such works. The aim is to show that a *conceptual* method of representing *e*-Services is still lacking, and this constitutes the motivation of our work. A SOA is the minimal framework for *e*-Services consisting of some basic operations and roles (see Figure 1):

- **Service Provider:** it is the subject providing software applications for specific needs as services; *(i)* from a business perspective, this is the owner of the service (e.g., the subject which is possibly paid for its services), and *(ii)* from the architectural perspective, this is the platform the service is deployed onto. Available services are described by using a *service description language* and advertised through the `publish()` operation on a public available service directory.

- **Service Requestor:** it is the party that uses the services; *(i)* from a business perspective, this is the business requiring certain services to be fulfilled (e.g., the payer subject), and *(ii)* from an architectural perspective, this is the application invoking the service. A service requestor discovers the most suitable service in the directory through the `find()` operation, then it connects to the specific service provider through the `bind()` operation and finally it uses the chosen service (`invoke()` operation).
- **Service Directory:** it is the party providing a *repository* and/or a *registry* of service descriptions, where providers publish their services and requestors find services.

The transport medium is a parameter of a SOA, therefore such a framework is easily integrable on different technologies; specifically, an *e*-Service Architecture is one in which the transport medium is electronic, whereas in a Web Service Architecture the transport medium is the Web. All emerging proposals consider the last scenario, in which services "live" over the Web, on the basis of a common transport medium, consisting of Web technologies such as HTTP, SOAP and XML Protocol (W3C, 2002). They address some basic technological issues of a SOA, that is the definition *(i)* of the service directory, *(ii)* of the service description language, *(iii)* of possible interactions a service can be involved in (i.e., the conversations), and *(iv)* of how to compose and coordinate different services, to be assembled together in order to support complex processes (i.e., the orchestration).

In the UDDI initiative, the architecture of a distributed service directory is proposed (UDDI.org, 2001). In the context of the W3C, many service description languages are being proposed for specific purposes:

- Web Service Description Language (WSDL, (Ariba *et al*., 2001)) for describing services, specifically their static interfaces;
- Web Service Conversation Language (WSCL, (Kuno *et al.*, 2001)) for describing the conversations a service supports; the underlying model is the one of the activity diagrams;
- Web Service Choreography Interface (WSCI, (BEA *et al.*, 2002)) for describing the observable behavior of a service in terms of temporal and logical dependencies among the exchanged messages;
- Web Service Flow Language (WSFL, (Leymann, 2001)) for the design of composite Web Services starting from simple ones (composition of Web Services); the underlying model is the one of Petri Nets;
- XLANG (Satish, 2001) for both the specification of the behavior of services and their orchestration; in (Meredith, 2002), it has been pointed out that this language is complete, it owns the property of composability and that the underlying theoretical model is the one of process algebra;
- Business Process Execution Language for Web Services (BPEL4WS, (Curbera *et al.*, 2002)) with the aim of merging WSFL and XLANG.

With respect to the SOA, the languages WSDL, WSCL and WSCI concern the service provider, and are all service description languages: WSDL addresses only static interface specifications, whereas WSCL and WSCI consider also behavioral issues and time.

More in details, WSDL, analogously to an interface definition language (e.g., CORBA IDL), describes methods, ingoing and outgoing messages and data types used by the service. Moreover, it supplies a mechanism to locate the service (e.g., using a URI), a protocol to exchange messages and the concrete mapping between the abstract method definition and the real protocol and data format. WSDL defines what the *e*-Service does, not how it does it; moreover WSDL does not express the semantics of message exchange, neither their correct order.

WSCL models the conversation (sequence of exchanged messages) supported by an *e*-Service: it specifies the XML documents being exchanged, and the allowed sequence, in a fashion similar to an activity diagram. A WSCL document is composed of four main elements: *(i)* document type descriptions specifying the types (schemas) of XML documents the service can accept and send during the conversation, *(ii)* interactions modeling the actions of the conversation as document exchanges between two participants (i.e., the *e*-Service and its client), *(iii)* transitions specifying the order constraints between interactions, and *(iv)* conversations, listing all the interactions and transitions that make up the conversation.

WSCI can be considered as an evolution of WSCL: it describes the observable behavior of an *e*-Service and how operations can be choreographed in the context of message exchanges in which the *e*-Service participates. WSCI allows the description of the correct order of the exchanged messages and permits the definition of multiple behaviors for the same *e*-Service on the basis of the context in which is used. Furthermore, it provides methods to manage exceptional situations, such as timeouts and fault messages[2]. WSCI, finally, allows to define the abstract behavior of a process that involves more *e*-Services in term of interfaces and links between operations, thus giving a view of the process in terms of message exchanges; with such a respect, WSCI is also an orchestration language.

As far as the languages WSFL, XLANG and BPEL4WS, they are orchestration and coordination languages, aimed at specifying how multiple *e*-Services are coordinated and the state and the logic needed for such a coordination; they provide constructs for limited time modeling, for correlation of *e*-Service instances and for exception management. Orchestration languages will be not further addressed in this paper.

2.1 Conceptual Representation of *e*-Services

From the previous discussion, it stems that some proposals address the issue of service behavior representation, considering also time; but we argue that *(i)* a clear semantics of such languages is missing, and *(ii)* they are not suitable for service-oriented computing design at a *conceptual* level.

We argue that in service-oriented computing we need conceptual languages in order to represent *e*-Services from an *external* point of view; such an external point of view is the one to be considered when composing and orchestrating services. Moreover, we need also languages for *internal* specification, i.e., specifying the *e*-Service for designers and implementers. In this work we concentrate only on the former issue (i,.e., external conceptual representation), since, for the latter purpose, classic specification languages used in software engineering practice can be used.

We propose a conceptual way of representing *e*-Service behavior, including time, based on finite state automata, and a new language, namely WSTL[3] (WEB SERVICE TRANSITION LANGUAGE), that can be used to represent an *e*-Service specification at a "technological" level, obtained by a straightforward mapping from the *e*-Service specification at a conceptual level[4]. In this way, we are able to give a clear and formal semantics to WSTL constructs. WSTL integrates well with other languages: *e*-Service specifications are expressed in WSTL and stored into suitable registries/repositories, complementing standard WSDL specifications.

Figure 2 shows the proposed conceptual approach: a cooperative designer, which is interested in searching, reusing, composing, and orchestrating *e*-Services (which are all based on *e*-Service external facets), defines an *e*-Service conceptual specification. Such a specification is then translated into a technological representation, i.e., based on standard XML-based languages; such a representation consists both of a WSDL file that specifies the static interface of the *e*-Service, and of a WSTL file, representing the *e*-Service behavior. Such files are then published into service directories, and then used for future searches, compositions, orchestrations, etc. Starting from the *e*-Service conceptual specification, internal designers and implementers will produce software artifacts implementing the *e*-Service.

An initiative which is quite similar to the proposed conceptual approach is the one of the DAML-S Coalition (Ankolekar *et al.*, 2002). DAML-S is an ontology for describing Web Services that enables the definition of service content vocabulary in terms of objects and complex relationships between them, including class, subclass relations, cardinality restrictions, etc., including all XML typing information. More in details, the DAML-S Ontology comprises:

- `ServiceProfile`. This is like the yellow page entry for a service. It relates and builds upon the type

[2] Time modelling is delegated to the `delay` element for represent time interval and to the `onTimeout` event handler for timeout situations.
[3] The XML Schema is available at `http://www.dis.uniroma1.it/pub/mecella/projects/vispo/wstl/WSTL1.1.xsd`
[4] Note that the conceptual specification is independent from any technology.

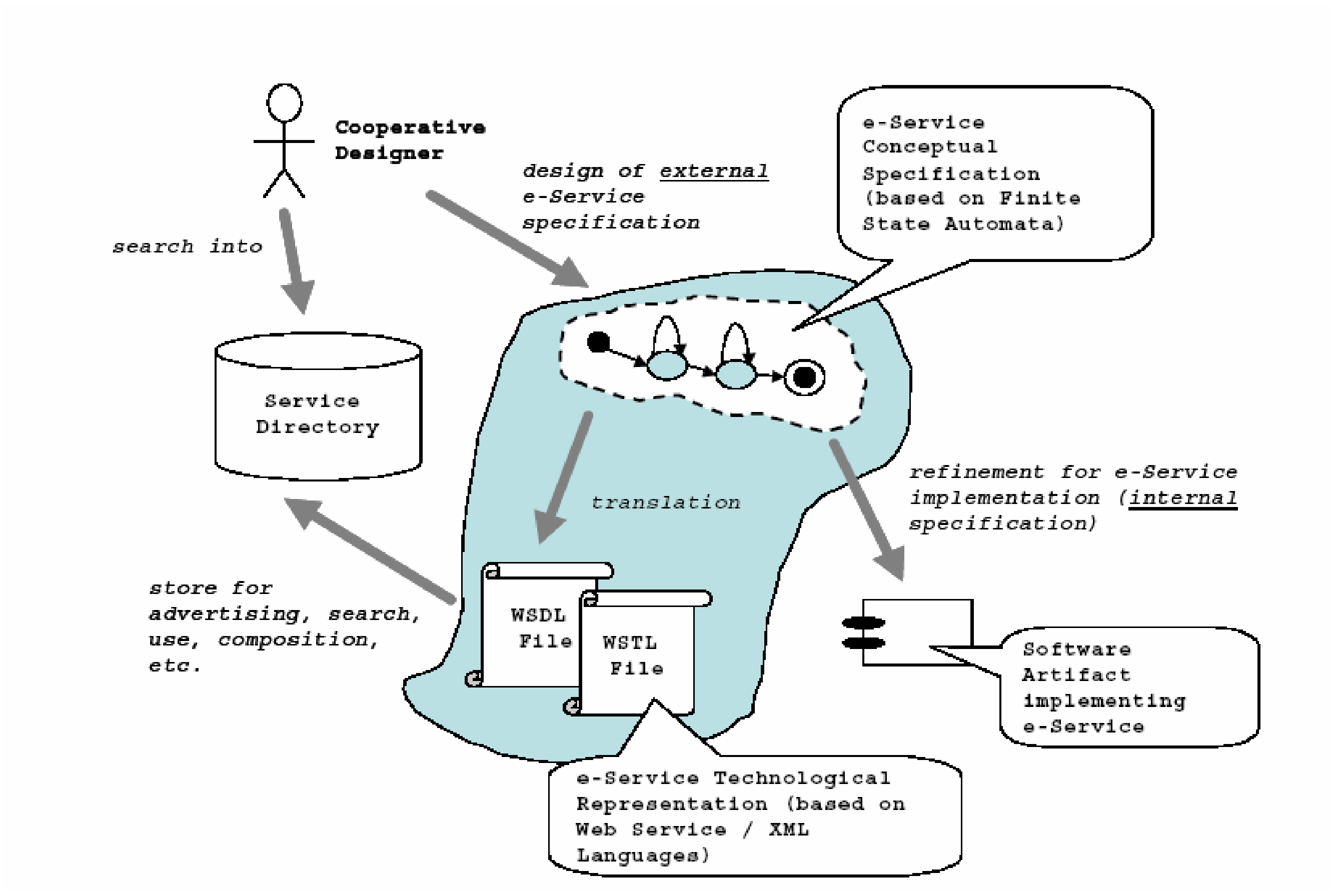
Cooperative Designer
search into
design of external e-Service specification
e-Service Conceptual Specification (based on Finite State Automata)
Service Directory
translation
refinement for e-Service implementation (internal specification)
store for advertising, search, use, composition, etc.
WSDL File
WSTL File
Software Artifact implementing e-Service
e-Service Technological Representation (based on Web Service / XML Languages)

Figure 2. Conceptual approach to e-Service specification

of content in UDDI, describing properties of a service necessary for automatic discovery, such as what the services offers, and its inputs, outputs, and its side-effects (preconditions and effects).

- `ServiceModel`. Describes the service's process model (the control flow and data-flow involved in using the service). It is designed to enable automated composition and execution of services.
- `ServiceGrounding`. Connects the process model description to communication-level protocols and message descriptions in WSDL.

The part of DAML-S more related to our work is the process model; its semantics has been defined both through a translation to (axiomatization in) first-order logic, as well as through a translation to an operational semantics using Petri Nets.

3. WSTL

WSTL is an XML-based description language able to represent the observable behavior of *e*-Services. WSTL does not address the implementation that actually drives the interactions; rather, it permits the description of what is observable from the point of view of the service users. Although simple, it enables the modeling of complex behaviors, such as loops and exceptions. WSTL extends the static interface of WSDL with elements which describe the correct sequence of the exchanged messages.

The root element of WSTL is the `Conversation` construct[5]; it describes the acceptable interactions of the *e*-Service in a message exchange scenario. Each WSTL conversation is characterized by a name and it can contain one or more `Transition` elements.

```
<Conversation name = string
              version = string
              description = string
        Transition+
</Conversation>
```

The `Transition` construct describes the behavior of the *e*-Service when a particular message is received or an internal event (e.g., a timeout) is triggered. It defines the successor state and the message that is returned. Details of behavior, such as WSDL messages (see below) and time constraints, are encapsulated in this element.

```
<Transition source = string
           target = string
           type = synchronous|asynchronous>
  (inputMessage|outputMessage|InputOutputMessage)
</Transition>
```

The `type` field allows for specifying whether the interaction is synchronous or asynchronous. Each `Transition` element contains exactly one of the following elements: `InputMessage`, `OutputMessage`, and `InputOutputMessage`.

```
<InputOutputMessage>
    input = message
    output = message
</InputOutputMessage>

<InputMessage>
    message
</InputMessage>

<OutputMessage>
    message
</OutputMessage>
```

The `InputMessage` element is used when a transition allows for an input message, without any output message. On the other hand, if the transition consists of an output message only, the `OutputMessage` element is used. Note that transitions containing either `InputMessage` or `OutputMessage` must be asynchronous. Finally, if the transition has both an input and an output message, `InputOutputMessage` elements are used.

```
<Message>
    name = string
    TimeConstraint?
</Message>
```

The various message elements previously described are based on WSTL `Message` elements. The value of field `name` must refer to a WSDL message type; as detailed in (Mecella *et al.*, 2002), in our approach the representation of an *e*-Service consists of a WSTL document plus a WSDL file containing message type definitions. The field `constraint` permits the association of a timeout descriptor with message, by means of the `TimeConstraint` element; this field is optional.

[5] The "?" symbols associated to an element means that the element can occur zero or one time; the "+" symbol states that the elements must occur at least once.

```
<TimeConstraint
    duration = string
    reference = string>
</TimeConstraint>
```

Finally, the optional field `reference` of the `TimeConstraint` element allows for specifying the event from which duration is calculated; if no `reference` field is specified, it is assumed that the duration is calculated from the beginning of the transition.

As a final remark, we would like to point out that WSTL allows the specification of *e*-Service conversations as computations of a finite state automaton, in which time constraints, as detailed in next section, can be added. Conversely WSCL, that is very similar from a syntactical point of view, does not allow the specification of time constraints with a clear semantics; additionally, it specifies conversations as activity diagrams.

4. Modeling e-Services

In this section, we propose a way to conceptually model *e*-Services, focusing on their dynamics, i.e., behavioral aspects, and we define the semantics of the language previously presented. In what follows, we consider an *e*-Service as a "black box" software application (delivered over the Web) *interacting* with a client (either a human or another *e*-Service) that wants to reach a goal. Each *interaction* consists of three steps:

1) the client invokes a *command* on the *e*-Service, by sending an input message;
2) the command triggers some internal computation, needed for the execution of the specific task associated with the command;
3) the client is possibly informed by an *output message* about the termination of the task and (possible) requested information.

Therefore, each interaction is described by the triple < *input command, internal computation, output message*>; however, by adopting a black box approach, we skip over internal computations and represent only the input/output behavior of *e*-Service from the client viewpoint, i.e., that have some effects towards the client: each interaction is therefore described by the pair <*input command, output message*> only.

At a given point, the input command that can be sent to the *e*-Service depends on the previous ones, and the client chooses the next command on the basis of the information returned by the previous output messages. At least in principle, interactions between an *e*-Service and its client may be unlimited, as in the case of an *e*-Service offering a continuous service, e.g., periodically delivering a product. According to these considerations, an *e*-Service can be fully modeled as an execution tree (Berardi *et al.*, 2003), whose branches represent all possible sequence of interactions between an *e*-Service and its client. However, since execution trees do not have a finite representation, we decide to resort to classes of *e*-Services that can be captured in a finite way.

We propose to represent an *e*-Service as a Finite State Automaton (FSA), since Finite State Automata (FSAs) are a widely known, simple but powerful formalism that allow us to capture a large class of *e*-Services. FSAs allow for modeling the behavior of a system as a sequence of transitions: in this way, we focus on the operational semantics of *e*-Services, i.e., we represent how they evolve and with which results, according to which command is invoked. In particular, we consider each *e*-Service interaction as a state transition, labeled with the pair <*input command, output message*>, where each state represents the history of executed sequences of interactions between the client and the *e*-Service.

Let eS be an e-Service. Its representation as FSA is a tuple of the form $S = \langle \mathrm{I}, \mathrm{O}, \Gamma, s_0, \delta, \mathrm{F} \rangle$ *where:*

- I *is a non-empty set of input commands, and* O *is a non-empty set of output messages;* $\mathrm{I} \times \mathrm{O}$ *is the alphabet of the FSA;*

- Γ *is a finite, non-empty set of states;*

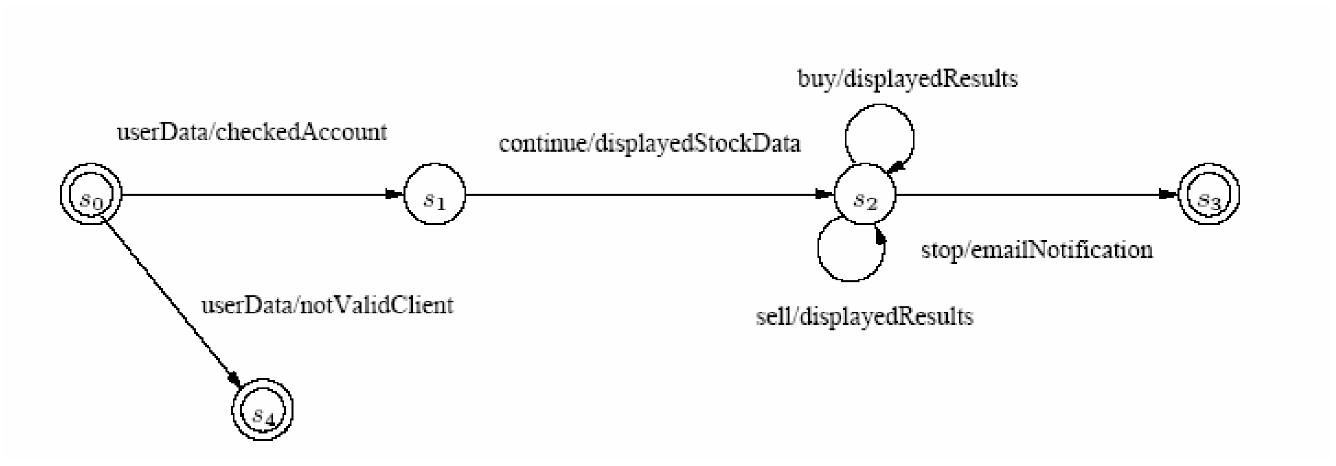

Figure 3. "Trading stocks" e-Service

- $s_0 \in \Gamma$ *is the initial state;*
- $F \subseteq \Gamma$ *is a non-empty set of final states, i.e., states where the client can terminate its interaction with the e-Service;*
- $\delta: \Gamma \times I \times O \rightarrow \Gamma$ *is the transition function, that represents the evolution of the FSA from a state to another one, for a given pair <input command, output message >*[6]*; the transition function can be undefined on the final states.*

We assume that, $s_0 \in F$ since a client may decide not to start the execution of an *e*-Service, even if he instantiated the *e*-Service. Indeed, an *e*-Service needs to be instantiated before it can be invoked, and its execution starts when the first input command is given (Mecella *et al.*, 2002).

A *configuration* is a pair $<s, i, o>$, where $s \in \Gamma, i \in I$ and $o \in O$. An *execution* of an *e*-Service is a finite, non empty sequence of configurations $<s_0, i_0, o_0>, \ldots <s_j, i_j, o_j>, <s_{j+1}, i_{j+1}, o_{j+1}>, \ldots, <s_n, i_n, o_n>$ where s_0 is the initial state, s_n is a final state, i.e., $s_n \in F$, and such that $\delta(s_j, i_j, o_j) = s_{j+1}$. The set of all executions forms the *behavior* of the *e*-Service.

Example 1: Let S_1 be an *e*-Service that allows a client to sell and to buy stocks online (see Figure 3). After instantiating the *e*-Service, the client inserts its userID and password (input command `userData`): the *e*-Service validates the client and checks the maximum amount that can be spent. This information is displayed to the client (output message `checkedAccount`). If, conversely, the client does not provide valid credentials, the *e*-Service will output a different output message `notValidClient`, thus terminating its interaction with the client. After visioning its available amount, the client invokes the `continue` command, in order to proceed to the stock trading; data on stocks are displayed (message `displayedStockData`) and the client can decide to sell (command `sell`) or buy (command `buy`) stocks: after each of these commands, information on the task result is displayed (message `displayedResults`). Finally, the client can decide to stop performing stock exchange transactions (command `stop`) and he is notified with an e-mail about his transactions (message `emailNotification`). This *e*-Service allows for many different executions, as for example:

- `userData/checkedAccount,continue/displayedStockData, sell/displayedResults,buy/displayedResults, stop/emailNotification`, in which a client operates a stock sale followed by a stock purchase;
- `userData/checkedAccount,continue/displayedStockData, stop/emailNotification`, in which the client uses the *e*-Service without any trading;

[6] State transitions are deterministic, since given a state, an input command and its corresponding output message, the successor state is univocally identified.

- `userData/notValidClient`, in which the client provides invalid credentials.

The client can terminate the interaction with the *e*-Service only in the two final states s_3 and s_4. □

Usually, *e*-Services similar to the one shown in Example 1, have additional behavior: in some scenarios, for instance, when an *e*-Service has not received any command from the client within a given instant, because of network overloading or because the client has exceeded the time limit for invoking the command, an error message is output to the client. In order to also capture such situations, absence of input commands or of output messages, as well as time constraints, need to be captured.

We introduce a new symbol ε that represents the lack of an input command or the absence of an output message: we refer to ε as the *empty* input command/output message. Note that an empty input command does not imply absence of internal computations (and therefore of the possible output message), since it may be the case that a computation starts with no triggering command, and it correctly returns an output message. We do not allow for the pair ε/ε, since no semantics can be associated to it from a user point of view. Analogously, each execution of an *e*-Service consists of at least one non-empty input command *or* one non-empty output message, otherwise no semantics can be associated to an execution. An *e*-Service with time constraints on transitions can be modeled by using Timed Finite State Automata (TFSAs). TFSAs have been discussed in many papers, e.g., (Henzinger *et al.,* 1994), (Alur *et al.*, 1994): starting from the discussion in these papers, we will slightly modify the previous definition in order to cope with time.

We make each transition depending also on the instants when the input command and the output message are invoked and returned, respectively. Additionally, we add two timing functions τ and σ: the former associates an input command with the *duration* of time within which this command needs to be executed by the client; the latter associates an output message with the *duration* of time within which the task triggered by the input command needs to be completed and the output message has to be returned by the *e*-Service; duration are considered starting from the instant in which the current state is entered. We assume that each *e*-Service has its own timing functions.

Let eS^T *be a timed e-Service. Its representation as TFSA is a tuple of the form* $S^T = \langle I, O, \Gamma, \varepsilon, s_0, F, T, \delta, \tau, \sigma \rangle$, *where:*

- I, O, Γ, s_0, F *are defined as for FSA;*
- T *is the set of timings, denoted by non- negative integers;*
- ε *denotes both the empty input command and the empty output message;*
- $\tau: \Gamma \times I \rightarrow T$ *is the starting timing function, defining the maximal duration of time by which an input command should be given in a specific state;*

  $\sigma: \Gamma \times O \rightarrow T$ *is the ending timing function, computing the maximal duration of time by which a computation is executed and the output message is returned, for a given state and a given output message. In particular,*

  - *on the final states having no outgoing transition,* τ *and* σ *are undefined;*

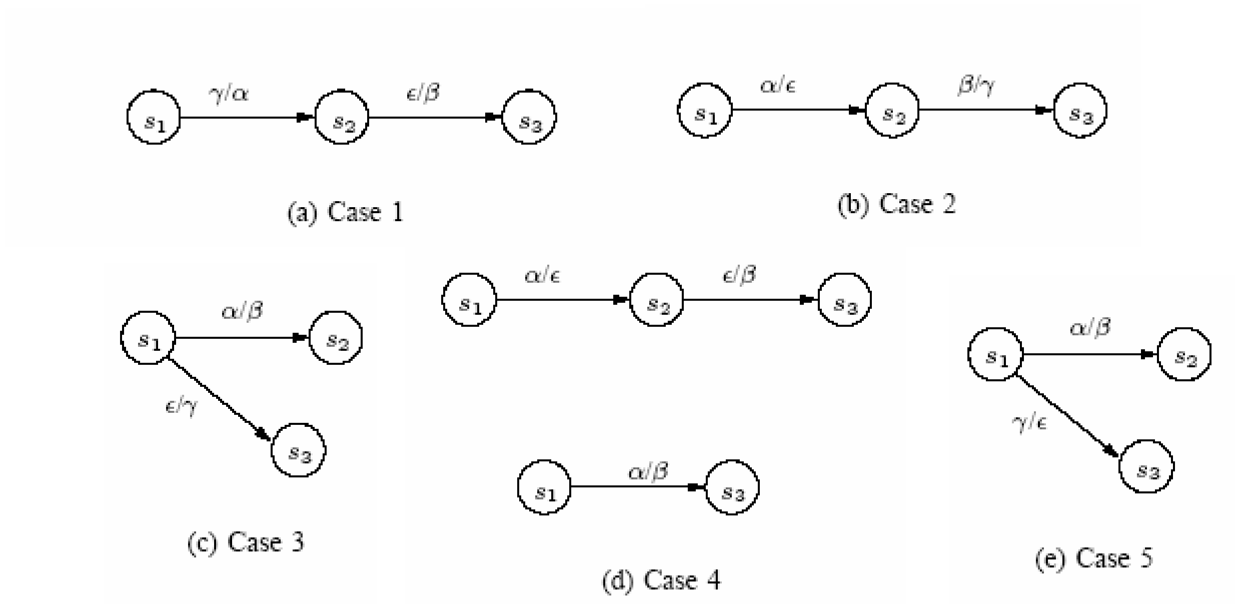
s_1 γ/α s_2 ϵ/β s_3
(a) Case 1
s_1 α/ϵ s_2 β/γ s_3
(b) Case 2
s_1 α/β s_2 ϵ/γ s_3
(c) Case 3
s_1 α/ϵ s_2 ϵ/β s_3
s_1 α/β s_3
(d) Case 4
s_1 α/β s_2 γ/ϵ s_3
(e) Case 5

Figure 4. Analysis of possible cases in which ε labels a timed transition

- *on the initial state, τ and σ are defined w.r.t.. the instant when the e-Service is instantiated*[7].

Note that since τ and σ impose timing constraints, transition by transition, we can choose not to impose any constraint on a transition. Additionally, τ and σ are undefined on ε, since it makes no sense defining a time constraint in a situation where either the input command or the output message is absent.

- $\delta: \Gamma \times [(I \cup \{\varepsilon\}) \times (O \cup \{\varepsilon\}) - (\{\varepsilon\}\times\{\varepsilon\})] \times T \times T \rightarrow \Gamma$ *is the transition function. In particular, within the expression $\delta(s_j, i, o, t^i, t^o)$, t^i and t^o have the following meaning:*
  - *if both i and o are different from ε, then t^o is the instant at which the output message o has been returned, and t^i is the instant at which the input command i has been invoked,*
  - *if i (resp. o) is ε, we set $t^i = t^o$, and t^i (resp. t^o) is intended as above. Finally, note that the transition function can be undefined on the final states.*

The notions of *configuration*, *execution* and *behavior* of an *e*-Service can be straightforwardly obtained from the analogous definitions previously given. In order to correctly capture timing on *e*-Services, several natural properties must be satisfied. In the following, we discuss them and we show how they can be imposed on δ, τ, and σ:

1) each transition cannot terminate before it has started, i.e., each output message cannot be returned before the corresponding input command;
2) each transition cannot start before the previous one has completed, i.e., each input command cannot be given before the previous output message has been returned (if the latter exists);
3) as far as δ is concerned, each input command and output message within each transition must be returned within the expected interval (when defined).

[7] We recall that an e-Service needs to be instantiated before it can be invoked (Mecella *et al.*, 2002).

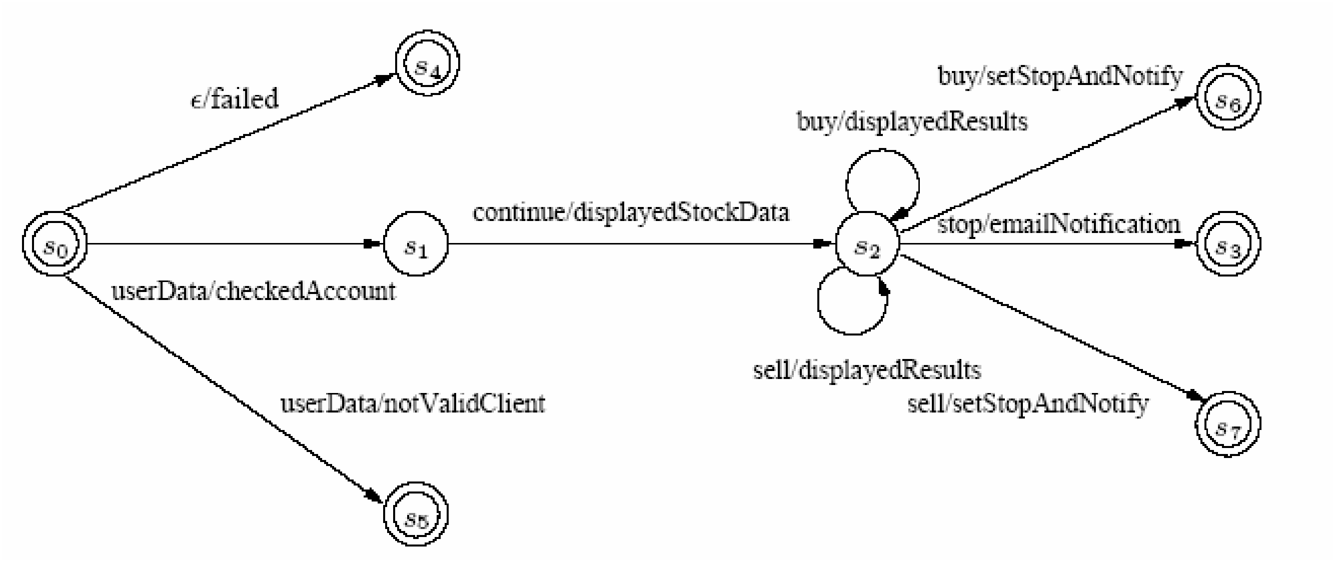

Figure 5. "Trading stocks" timed e-Service

Noting that Property 2 holds naturally from the definition of transition function, the other properties can be enforced in δ as follows: $\forall s_j, s_{j+1}, i, o, t^i, t^o$: $\delta(s_j, i, o, t^i, t^o) = s_{j+1}$ if $t^i \leq t^o \wedge t^i \leq \tau(s_j, i) \wedge t^o \leq \sigma(s_j, o)$. If either *i* or *o* are empty, or in general if τ and σ are not defined, this expression simplifies straightforwardly.

As far as τ and σ are concerned, Property 1 can be imposed by $\forall s_j, s_{j+1}, i, o, t^i, t^o$ *s.t.* $\delta(s_j, i, o, t^i, t^o) = s_{j+1}$: $\tau(s_j, i) \leq \sigma(s_j, o)$. Property 2 is guaranteed from the fact that both τ and σ are defined w.r.t. the current state, transition by transition. However, this situation holds when both the input command and the output message are not empty. If either one of them is ε, several cases may happen, as shown in Figure 4, where α and β denote input messages or output messages that are different from ε, whereas γ can also denote ε (when admissible). As far as the first transition of case 1 and second transition of case 2 is concerned, Property 1 is meaningless, since no timing constraints can be imposed on ε. Property 2 is guaranteed as in the general case. In the situation shown by case 3, Properties 1 and 2 express the fact that β cannot be returned before α is invoked: this is captured by $\tau(s_1, \alpha) \leq \sigma(s_1, \beta)$. The situation in the upper part of Figure 4(d) (case 4) can be reduced to the situation in the lower part, in which ε is not present. As far as case 5 is concerned, the presence of ε does not influence the choice of which transition is to be taken, since the timing constraints imposed by τ on the input commands may determine the choice. In cases like this we assume that $\tau(s_1, \alpha)$ is greater (or less) than $\tau(s_1, \gamma)$, implying that γ is different from α. A similar case can be identified w.r.t. σ and the output messages, leading to analogous conclusions.

The proposed formalization naturally allows for capturing additional time constraints, as those imposing that the interval between the instant when an input command is invoked and the instant when the corresponding output message is returned is less than a given duration. Before concluding the section, we apply our formalization to an example.

Example 2: Figure 5 shows the *e*-Service discussed in Example 1, extended with time constraints and *ε* transitions. In particular, this *e*-Service has the following behavior:

- if the command `userData` is not given within the expected interval, the output message `failed` is returned;

$$\begin{array}{ll}
\delta(s_0, userData, checkedAccount, t^i_1, t^o_1) = s_1 & \text{if } t^i_1 \leq t^o_1 \wedge t^i_1 \leq \tau(s_0, userData) \wedge t^o_1 \leq \sigma(s_0, checkedAccount) \\
\delta(s_0, \epsilon, failed, t^o_2, t^o_2) = s_4 & \text{if } t^o_2 \leq \sigma(s_0, failed) \\
\delta(s_0, userData, notValidClient, t^i_1, t^o_3) = s_5 & \text{if } t^i_1 \leq t^o_3 \wedge t^i_1 \leq \tau(s_0, userData) \wedge t^o_3 \leq \sigma(s_0, notValidClient) \\
\delta(s_1, continue, displayedStockData, t^i_4, t^o_4) = s_2 & \text{if } t^i_4 \leq t^o_4 \\
\delta(s_2, buy, displayedResults, t^i_5, t^o_5) = s_2 & \text{if } t^i_5 \leq t^o_5 \wedge t^o_5 \leq \sigma(s_2, displayedResults) + t^i_5 \\
\delta(s_2, buy, setStopAndNotify, t^i_5, t^o_6) = s_6 & \text{if } t^i_5 \leq t^o_6 \\
\delta(s_2, sell, displayedResults, t^i_7, t^o_7) = s_2 & \text{if } t^i_7 \leq t^o_7 \wedge t^o_7 \leq \sigma(s_2, displayedResults) + t^i_7 \\
\delta(s_2, sell, setStopAndNotify, t^i_7, t^o_8) = s_7 & \text{if } t^i_7 \leq t^o_8 \\
\delta(s_2, stop, emailNotification, t^i_9, t^o_9) = s_3 & \text{if } t^i_9 \leq t^o_9
\end{array}$$

Figure 6. Transition function for “trading stock” e-Service

- if the output message `displayedResult` associated to command `buy` is not returned before a given time, the message `setStopAndNotify` is returned, that stops the stock trading, notifies the client about this problem and reports to him about the stocks trade. The same happens for output message `displayedResult` associated to command `sell`.

Additionally, we impose several temporal constraints on transitions, defined by τ and σ, as shown in Figure 6. The transition function δis also defined in Figure 6. The constraints on τ and σ are:

$\tau(s_0, userData) \leq \sigma(s_0, checkedAccount)$
$\tau(s_0, userData) \leq \sigma(s_0, failed)$
$\tau(s_0, userData) \leq \sigma(s_0, notValidClient)$ □

5. Mapping

The FSA model of *e*-Services introduced in the previous section gives us the opportunity to exploit the results obtained in the literature about FSAs in order to prove important properties, such as the correctness of an *e*-Service, w.r.t. its intended behavior, in order to remove unexpected behaviors. This opportunity, however, would be useless without techniques for translating this abstract model into a concrete, standard-compliant representation. In this section we aim at illustrating how to obtain the specification of an *e*-Service in WSTL, starting from its conceptual representation as a FSA. In what follows, we illustrate the rules used to map an FSA into a WSTL document. The complete *e*-Service specification is then constituted by such a WSTL document plus a WSDL document containing message type definitions.

First, we associate a WSDL message type with each input command and with each output message. For example, for the input command i_j, we obtain the following message:

```
<message name ="i_j"
            message’s parts
<message>
```

Note that the structure of the message must be decided at implementation time, since it depends on the application. Then, we consider each transition with its associated constraints, and map them to a WSTL `Transition` element, in a very intuitive way. Suppose that the transition between s_i and s_j is captured as follows:

$\delta(s_i, i_k, o_k, t^{i0}, t^{o0}) = s_j$
$t^{i0} \leq t^{o0}$
$t^{i0} \leq \tau(s_j, i_k)$
$t^{o0} \leq \sigma(s_j, o_k)$

The current state s_i is mapped to the `source` field, and the successor state s_j is mapped to the `target` field. As far as representation of input commands and output messages are concerned, three different situations may happen:

1) $o_h = \varepsilon$ and $i_k \neq \varepsilon$: it is captured by an `InputMessage` element, having the field `name` equal to i_k. The `duration` field is set to $\tau(\cdot)$, if $\tau(\cdot)$ is defined. Finally, the `reference` field is used to define additional time constraints.
2) $i_k = \varepsilon$ and $o_h \neq \varepsilon$: it is represented by an `OutputMessage` element, having the field `name` equal to o_h. The values of the remaining fields are set using the same rules as before.
3) $i_k \neq \varepsilon$ and $o_h \neq \varepsilon$: it is captured by an `InputOutputMessage` element. The values of its fields are set using the same rules as before.

As an additional example, consider the situation when $i_k \neq \varepsilon$, $o_h \neq \varepsilon$,$\tau(\cdot)$ is defined, and $\sigma(\cdot)$ is not. Applying the above rules, we obtain the following `Transition` element.

```
<Transition source="s_i"
            target="s_j"
            type="synchronous">
    <InputOutputMessage>
          <input>
             <name>"i_k"</name>
            <constraint duration="tau(s_i,i_k)"/>
          </input>
          <output>
            <name>"o_h"</name>
          </output>
    </InputOutputMessage>
</Transition>
```

Finally, all the `Transition` elements are encapsulated into a `Conversation` element that represents the overall behavior of the *e*-Service. In what follows, we show the complete translation of the *e*-Service in Figure 5, as a WSTL document[8]. This file has been validated against the WSTL schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<Conversation name="trading stocks timed eService"
xmlns=".../WSTL/wstl11"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://.../WSTL/wstl11
http://.../WSTL/WSTL1.1.xsd">
  <Transition source="s0" target="s1" type="synchronous">
    <InputOutputMessage>
      <input>
        <name>"userData"</name>
        <constraint duration="00:00:25"/>
      </input>
      <output>
        <name>"checkedAccount"</name>
        <constraint duration="00:00:30"/>
      </output>
    </InputOutputMessage>
  </Transition>
  <Transition source="s0" target="s4" type="asynchronous">
    <OutputMessage>
      <name>"failed"</name>
      <constraint duration="00:01:00"/>
    </OutputMessage>
  </Transition>
  <Transition source="s0" target="s5" type="synchronous">
    <InputOutputMessage>
      <input>
        <name>"userData"</name>
        <constraint duration="00:00:25"/>
      </input>
      <output>
        <name>"notValidClient"</name>
        <constraint duration="00:00:30"/>
```

[8] For simplicity, we omit the WSDL type definitions.

```
      </output>
    </InputOutputMessage>
  </Transition>
  <Transition source="s1" target="s2" type="synchronous">
    <InputOutputMessage>
      <input>
        <name>"continue"</name>
      </input>
      <output>
        <name>"displayedStockData"</name>
      </output>
    </InputOutputMessage>
  </Transition>
  <Transition source="s2" target="s2" type="synchronous">
    <InputOutputMessage>
      <input>
      <name>"buy"</name>
      </input>
      <output>
        <name>"displayedResult"</name>
        <constraint duration="00:01:00" reference="buy"/>
      </output>
    </InputOutputMessage>
  </Transition>
  <Transition source="s2" target="s2" type="synchronous">
    <InputOutputMessage>
      <input>
        <name>"sell"</name>
      </input>
      <output>
        <name>"displayedResult"</name>
        <constraint duration="00:10:00" reference="sell"/>
      </output>
    </InputOutputMessage>
  </Transition>
  <Transition source="s2" target="s6" type="asynchronous">
    <InputOutputMessage>
      <input>
        <name>"buy"</name>
      </input>
      <output>
        <name>"setStopAndNotify"</name>
      </output>
    </InputOutputMessage>
  </Transition>
  <Transition source="s2" target="s7" type="asynchronous">
    <InputOutputMessage>
      <input>
        <name>"sell"</name>
      </input>
      <output>
        <name>"setStopAndNotify"</name>
      </output>
    </InputOutputMessage>
  </Transition>
  <Transition source="s2" target="s3" type="asynchronous">
    <InputOutputMessage>
      <input>
        <name>"stop"</name>
      </input>
      <output>
        <name>"emailNotification"</name>
      </output>
    </InputOutputMessage>
  </Transition>
</Conversation>
```

6. Conclusions and Future Work

In this paper we have presented a conceptual model for representing *e*-Service behavior and temporal constraints, based on FSAs. Indeed, FSAs allow us to capture a large class of *e*-Services and to formally verify important properties of *e*-Services (Weikum, 1999): these include correctness, safety (i.e., at each point in the execution of an *e*-Service certain logical invariants hold) and liveness (i.e., an *e*-Service is ensured to move towards a point where a goal can be reached). In future work we address such issues in the context of *e*-Service behavioral and temporal aspects. Additionally, we have introduced a new language, namely WSTL (WEB SERVICE TRANSITION LANGUAGE), that relies on such a conceptual model. The novelty of our approach is that WSTL provides constructs to which corresponding elements of FSAs can be straightforwardly mapped. At present this mapping is done manually, but we plan in the future to automate this step. In such a way, we have defined a precise semantics for our language and we allow for service-oriented computing at conceptual level. The issues of conceptual specification of service-oriented computing and translation to technologies have been considered in this paper for single *e*-Services. In the future, we will extend our work by considering them in the context of *e*-Service composition and orchestration.

7. Acknowledgements

This work has been partially supported by MIUR through the "Fondo Strategico 2000" Project *VISPO (Virtual-district Internet-based Service PlatfOrm)* (`http://cube-si.elet.polimi.it/vispo/index.htm`) and the "FIRB 2001" Project *MAIS (Multi-channel Adaptive Information Systems)*. The work of Massimo Mecella has been also partially supported by the European Commission under Contract No. IST-2001-35217, Project *EU-PUBLI.com (Facilitating Co-operation amongst European Public Administration Employees)* (`http://www.eu-publi.com/`).

8. References

Alur, R. and Dill, D. L., 1994, "A Theory of Timed Automata. Theoretical Computer Science", 126(2):183–235.

Ankolekar, A., Burstein, M., Hobbs, J., Lassila, O., Martin, D., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M. Payne, T., and Sycara, K., 2002, "DAML-S: Web Service Description for the Semantic Web." In Proceedings of the 1st International Semantic Web Conference (ISWC 2002).

Ariba, Microsoft, and IBM, 2001, "Web Services Description Language (WSDL) 1.1". W3C Note. Available on-line (link checked March, 1st 2003): http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

BEA, Intalio, SAP, and Sun, 2002, "Web Service Choreography Interface (WSCI) 1.0". W3C Document. Available on-line (link checked March, 1st 2003): http://www.w3.org/TR/wsci/.

Berardi, D., Calvanese, D., De Giacomo, G., and Mecella, M., 2003, "Reasoning about Actions for e-Service Composition". Technical Report 01-2003, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza".

Castro, J., Kolp, M., and Mylopoulos, J., 2002, "Towards Requirements-driven Information Systems Engineering: the Tropos Project". Information Systems, 27(6).

Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S., and Weerawarana, S, 2002, "Business Process Execution Language for Web Services (Version 1.0)". IBM Document. Available on-line (link checked March, 1st 2003): http://www.ibm.com/developerworks/library/ws-bpel/.

Dayal, U., Hsu, M., and Ladin, R., 2001, "Business Process Coordination: State of the Art, Trends and Open Issues". In Proceedings of the 27th Very Large Databases Conference (VLDB 2001).

De Michelis, G., Dubois, E., Jarke, M., Matthes, F., Mylopoulos, J., Papazoglou, M., Pohl, K., Schmidt, J., Woo, C., and Yu, E., 1997, "Cooperative Information Systems: A Manifesto". In Papazoglou, M. and Schlageter, G., editors, Cooperative Information Systems: Trends & Directions. Accademic-Press.

Elmagarmid, A. and McIver Jr, W., 2001, "The Ongoing March Towards Digital Government" .IEEE Computer, 34(2). Special Issue.

Georgakopoulos, D., editor, 1999, Proceedings of the 9th International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises (RIDEVE' 99).

Henzinger, T., Manna, Z., and Pnueli, A., 1994, "Temporal Proof Methodologies for Timed Transition Systems". Information and Computation, 112(2):273–337.

HP, 2001, "Web Services Concepts: a Technical Overview. HP Document." Available on-line (link checked March, 1st 2003): http://www.bluestone.com/downloads/pdf/web_services_tech_overview.pdf.

Kuno, H., Lemon, M., Karp, A., and Beringer, D. , 2001, "Conversations + Interfaces = Business Logic". In Proceedingsof the 2nd VLDB International Workshop on Technologies for e-Services (VLDB-TES 2001).

Lenzerini, M., 2002, “Data Integration: A Theoretical Perspective”. In Proceedings of the 21st ACM SIGACT-SIGMODSIGART Symposium on Principles of Database Systems (PODS 2002).

Leymann, F., 2001, “Web Service Flow Language (WSFL 1.0)”. IBM Document. Available on-line (link checked March, 1st 2003): http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

Mecella, M. and Pernici, B., 2002, “Building Flexible and Cooperative Applications Based on e-Services”. Technical Report 21-2002, Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”. (available on line at: http://www.dis.uniroma1.it/~mecella/publications/mp_techreport_212002.pdf).

Mecella, M., Pernici, B., and Craca, P., 2001, “Compatibility of e-Services in a Cooperative Multi-Platform Environment”. In Proceedings of the 2nd VLDB International Workshop on Technologies for e-Services (VLDB-TES 2001).

Meredith, G., 2002, “Contract and Types”. In Invited Talk at the 3rd VLDB International Workshop on Technologies for e-Services (VLDB-TES 2002).

Rahm, E. and Bernstein, P. , 2001, “A Survey of Approaches to Automatic Schema Matching”. VLDB Journal, 10(4).

Satish, T., 2001, “XLANG. Web Services for Business Process Design. Microsoft Document”. Available on-line (link checked March, 1st 2003): http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.hm.

UDDI.org, 2001, “UDDI Technical White Paper”. Available on-line (link checked March, 1st 2003): http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf.

W3C, 2002, “XML Protocol”. XML Protocol Web Page (link checked September, 1st 2002): http://www.w3.org/2000/xp/.

Weikum, G., 1999, “Towards Guaranteed Quality and Dependability of Information Services”. In Invited Talk at the 8th German Database Conference (Datenbanksysteme in B¨uro, Technik und Wissenschaft).

Yang, J., van den Heuvel, W., and Papazoglou, M., 2002, “Tackling the Challenges of Service Composition in e-Marketplaces”. In Proceedings of the 12th International Workshop on Research Issues on Data Engineering: Engineering E-Commerce/E-business Systems (RIDE-2EC 2002).