Requirements-Driven Verification of Web Services ⋆

Marco Pistore[1,2] and Marco Roveri[2] and Paolo Busetta[2]

[1] *DIT - University of Trento, Via Sommarive 14, I-38050 Trento, Italy*
pistore@dit.unitn.it

[2] *ITC-irst, Via Sommarive, 18, I-38050 Trento, Italy*
{roveri,busetta}@irst.itc.it

Abstract

We propose a requirements-driven approach to the design and verification of Web services. The proposed methodology starts from a requirements model, which defines a business domain at a "strategic" level, describing the participating actors, their mutual dependencies, goals, requirements, and expectations. This business requirements model is then refined into a business process model. In this refinement, definitions of the processes carried out by the actors of the domain are added to the model in the form of BPEL4WS code. We show how to exploit model checking techniques for the verification of the specification, both at the requirements and at the process level. At the requirements level, model checking is used to validate the specification against a set of queries specified by the designer; at the process level, it is used to verify if the BPEL4WS processes satisfy the constraints described in the requirements model.

1 Introduction

BPEL4WS [1] is quickly emerging as the language of choice for Web service composition. It provides a core of process description concepts that allow for the definition of business processes interactions. This core of concepts is used both for defining the internal *business processes* of a participant to a business interaction and for describing and publishing the external *business protocol* that defines the interaction behavior of a participant without revealing its internal behavior.

BPEL4WS opens up the possibility of applying a range of formal techniques to the verification of the behavior of Web services. For instance, it is possible to check the internal business process of a participant against the external business protocol that the participant is committed to provide; or, it is possible to verify whether the composition of two or more processes satisfies general properties (such

⋆ Research supported in part by the MIUR-FIRB Project RBNE0195K5 (ASTRO).

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

as deadlock freedom) or application-specific constraints (e.g., temporal sequences, limitations on resources). These kinds of verifications are particularly relevant in the distributed and highly dynamic world of Web services, where each partner can autonomously redefine business processes and interaction protocols. In the long term, we envision an environment where an agent executing one or more business processes can autonomously discover new types of services and extends its own processes accordingly. Before being integrated in the actor's processes, discovered resources must be verified against the agent's own requirements and constraints.

Different techniques have been already applied to the verification of business processes (see, e.g., [4,6,7,8]). However, current approaches do not address the issues of how to model the requirements that the BPEL4WS processes are supposed to satisfy, and of how to manage the evolution of processes and requirements. To this purpose, we propose to extend a BPEL4WS specification with a *business requirements model*. This provides a "strategic" description of the different actors in the business domain with their goals and needs and with their mutual dependencies and expectations, and provides the motivations behind business processes. The business requirements model drives the design of business processes and the verification that they achieve desired goals. It allows for the selection of partners and external services that satisfy the expected constraints. Also, it permits to trace changes in the requirements and in the processes. In the long term, it will give a semantic description to an autonomous agent of what it has to achieve and what may be provided by external partners, thus enabling dynamic composition of services.

This paper presents some preliminary results of our first steps towards the vision outlined above. Our starting point is a modeling language, called *Tropos* [10], whose objective is to capture the business requirements of the actors of a domain, their dependencies and expectations. The formal counterpart of Tropos, Formal Tropos [5] supports the definition of temporal constraints on the evolution of the modeled domain, and enables the application of a whole set of formal techniques to Tropos models. In this paper, we show how a business requirements model expressed in Tropos can be progressively refined into a business process model. In this refinement, BPEL4WS code is added to define the processes carried out by the actors of the domain. This BPEL4WS code is a procedural counterpart of the temporal constraints of the requirements model. We show how to apply model checking techniques for verifying that the refined process actually satisfies the original requirements.

This paper is structured as follows. In Section 2 we introduce the Tropos language; we show how to use it to model business requirements; and we describe how model checking techniques are applied to the validation of the requirements. In Section 3 we illustrate the refinement of the requirements model into a business process model; we also show how model checking techniques can be applied to verify whether the BPEL4WS processes satisfy the requirements. Section 4 ends the paper with concluding remarks and future work directions.

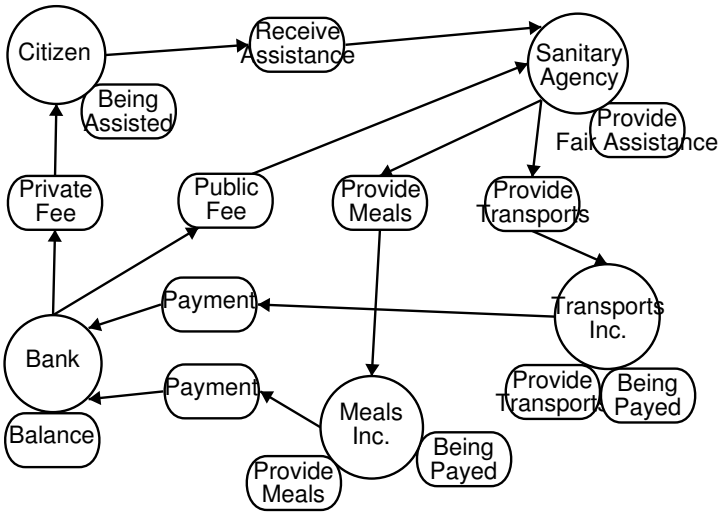

Fig. 1. High level business requirements model.

2 Business Requirements Modeling in Tropos

In this section we propose a language for describing business requirements in a Web Service framework. This language, called *Tropos* [10], provides graphical notations and a formal specification language that have been specifically designed for requirements. It has been adopted to model requirements of a variety of software and organization systems (see `http://www.troposproject.org/` for some examples).

2.1 Modeling Business Requirements: A Case-Study

The Tropos modeling language is founded on the premise that during the requirements analysis phase of the software development process it is important to understand and model the strategic aspects underlying the organizational setting within which the software system will eventually function. By understanding these strategic aspects, one can better identify the motivations for the software system and the role that it will play inside the organizational setting. For instance, in order to develop a software system that supports the elder citizens in receiving sanitary assistance from the public administration, we need first to understand the interdependencies among the citizens and the different actors in the public administration that underly the process of receiving assistance. In this paper we consider an extension of Tropos, which is called Tropos4WS, and which is suitable for integration with business process models.

We introduce Tropos4WS in the context of a case-study in the field of public welfare, extracted from a larger domain analysis concerning the local government of Trentino (Italy). Figure 1 is a Tropos diagram that provides a high-level description of the case-study domain. It represents the main *actors* and *goals* of the domain: the `Citizen` that aims at being assisted; the `SanitaryAgency` which aims at providing a fair assistance to the citizens; the `TransportsInc` which provides

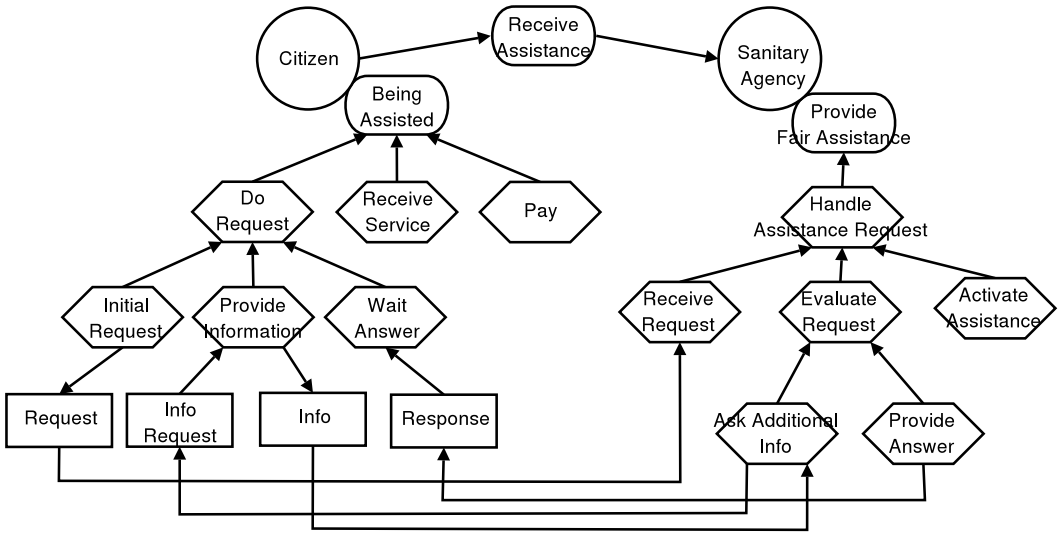

Fig. 2. Requirements model refinement.

transportation services; the `MealsInc` which delivers meals at home; and the `Bank` which handles the government's finances. The picture also describes the dependencies and expectations that exist among these actors. For instance, the citizen depends on the sanitary agency for being assisted, and this is formulated in the model with dependency `ReceiveAssistance` from `Citizen` to `SanitaryAgency`.

Starting from this high-level view of the organizational or business system, the Tropos methodology proceeds with an incremental refinement process (see Figure 2). Goals are decomposed into sub-goals, or operationalized into tasks, taking into account the dependencies existing among the different actors. For instance, the goals `BeingAssisted` and `ProvideFairAssistance` are refined in order to reflect the "contract" that governs the way the assistance is provided by the `SanitaryAgency` to the `Citizen`. More precisely, the `Citizen` refines the goal `BeingAssisted` into the three sub-task of `DoRequest`, `ReceiveService` and `Pay`. `DoRequest` is further refined into `InitialRequest`, `ProvideInformation`, `WaitAnswer`. On the other side, the `SanitaryAgency` refines the goal `ProvideFairAssistance` into the task `HandleAssistanceRequest`, which is further refined into `ReceiveRequest`, `EvaluateRequest` and `ActivateAssistance`.

The refinement procedure ends once we have identified all basic tasks that define the business process. To these basic tasks we associate messages that describe the basic interactions among actors. For instance, task `InitialRequest` requires to send a message `Request` to the `SanitaryAgency`. This message is received and processed by the `SanitaryAgency` task `ReceiveRequest`. The task `AskAdditionalInfo` requires to send a message `InfoRequest` to the `Citizen` which receives and processes it with task `ProvideInformation` and responds with an `Info` message. Once sufficient information has been gathered, the `SanitaryAgency` sends a `Response` message to the `Citizen`. Figure 2 shows the refinement for the interactions between `Citizen` and `SanitaryAgency`. Similar refinements need to be done also for the other tasks and interactions in the domain.

```
ENTITY AssistanceNeed
ENTITY Query

ACTOR Citizen
ACTOR SanitaryAgency

GOAL DEPENDENCY ReceiveAssistance
  Mode maintain
  Depender Citizen
  Dependee SanitaryAgency
  Creation condition EXISTS ba: BeingAssisted (ba.actor = depender)
  Invariant F EXISTS pfa: ProvideFairAssistance (pfa.actor = dependee & Fulfilled(pfa))
  Fulfillment condition FORALL dr: DoRequest (
                          (dr.actor = depender & Fulfilled(dr) & dr.result) ->
                             F EXISTS rs: ReceiveService (rs.actor = depender & Fulfilled(rs)))

TASK DoRequest
  Mode achieve
  Actor Citizen
  Super BeingAssisted
  Attribute constant need: AssistanceNeed
            result: boolean
  Invariant F EXISTS ir: InitialRequest (ir.super = self)
  Invariant EXISTS ir: InitialRequest (ir.super = self & Fulfilled(ir))
              -> F EXISTS pi: ProvideInformation (pi.super = self)
  Invariant EXISTS pi: ProvideInformation (pi.super = self & Fulfilled(pi))
              -> F EXISTS wa: WaitAnswer (wa.super = self)
  Invariant Fulfilled(self) -> EXISTS wa: WaitAnswer
              (wa.super = self & Fulfilled(wa) & (result <-> wa.result))
  Fulfillment definition EXISTS wa: WaitAnswer (wa.super = self & Fulfilled(wa))

TASK InitialRequest
  Mode achieve
  Actor Citizen
  Super FareRichiesta
  Invariant F EXISTS r: Request (r.sender = actor & r.need = super.need)
  Fulfillment definition EXISTS r: Request (r.sender = actor & r.need = super.need)

MESSAGE Request
  Sender Citizen
  Receiver SanitaryAgency
  Attribute constant need: AssistanceNeed
  Creation condition Exists dr: DoRequest (dr.actor = sender & dr.need = need)
```

Fig. 3. Formal Tropos specification.

2.2 *Formal Specification of Business Requirements*

The Tropos graphical models have a formal counterpart described in the Formal Tropos specification language. *Formal Tropos* (hereafter FT) has been designed to supplement Tropos models with a precise description of their dynamic aspects. In FT the focus is on the circumstances in which the goals and tasks arise, and on the conditions that lead to their fulfillment. In this way, the dynamic aspects of a requirements specification are introduced at the strategic level, without requiring an operationalization of the specification. A precise definition of FT and of its semantics can be found in [5]. Here we present the most relevant aspects of the language based on the case-study. An excerpt of its FT specification can be found in Figure 3.

An FT specification describes the relevant objects of a domain and the relationships among them. The description of each object is structured in two layers. The outer layer is similar to a class declaration and defines the structure of the instances together with their attributes. The inner layer expresses constraints on the lifetime of the objects, using a typed first-order linear-time temporal logic (hereafter LTL). Several instances of each element may exist during the evolution of the system. To distinguish among the different instances, a list of attributes is associated to each class. Each attribute has a *sort* which can be either primitive (boolean, integer...) or classes. For instance, boolean attribute `result` of task `DoRequest` determines

whether the response to the request of the citizen has been positive or not. Attribute `need` of goal `DoRequest` is used to distinguish between the different needs of the same citizen. `Entity` classes like `AssistanceNeed` are added to the FT specification to represent the "passive" elements of the domain that are used as attributes in other classes. An attribute may be marked as `constant`, which means that the value of the attribute does not change during the lifetime of the class instance.

Some special attributes are associated to each kind of class in the FT specification. Goals and tasks are associated to the corresponding actor with the special attribute `Actor`. Similarly, `Depender` and `Dependee` attributes of dependencies represent the two parties involved in a delegation relationship. Attribute `Super` for goals and tasks denotes the parent goal or task. For messages we use special attributes to characterize the actor instances corresponding to the sender (`Sender`) and to the receiver (`Receiver`). All these special attributes are constant by definition.

An important aspect of FT is its focus on the conditions for the *fulfillment* of goals and tasks. These are characterized by a `Mode`, which declares the modality of their fulfillment. The two most common modalities are `achieve` (which means that the actor expects to reach a state where, e.g., the goal has been fulfilled) and `maintain` (which means that the fulfillment condition has to be continuously maintained). For instance, dependency `ReceiveAssistance` is of type maintain, to capture the fact that this "contract" between citizen and sanitary agency has to be maintained over time. On the other hand, task `DoRequest` is of type achieve, since the citizen aims at reaching a state where this task is terminated.

The inner layer of an FT class declaration consists of constraints that describe the dynamic aspects of entities, actors, goals, and dependencies. In FT we distinguish among `Creation`, `Invariant`, and `Fulfillment` constraints. `Creation` constraints define conditions that should be satisfied when a new instance of a class is created. In the case of goals and tasks, the creation is interpreted as the moment when the associated actor begins to desire the goal or to perform the task. `Invariant` constraints define conditions on the life of all class instances. `Fulfillment` constraints should hold whenever a goal is achieved or a task is completed. Creation and fulfillment constraints are further distinguished as sufficient conditions (keyword `trigger`), necessary conditions (keyword `condition`), and necessary and sufficient conditions (keyword `definition`).

In FT, constraints are described with formulas in a typed first-order linear-time temporal logic. Besides the standard boolean and relational operators, the logic provides the quantifiers `Forall` and `Exists`, which range over all the instances of a given class, and a standard set of linear-time temporal operators. The latter include operator `X`, which defines a condition that has to hold in the next state of the evolution of the system, operator `F`, which defines a condition that has to hold eventually in the future, and operator `G`, which defines a condition that has to hold in all future states.

In the FT specification of Figure 3, the first three invariants of task `DoRequest` describe the expected evolution of the task and its relations with the sub-tasks. Namely, if the task `DoRequest` is started, then eventually sub-task

```
POSSIBILITY P1
  Exists dr: DoRequest (Fulfilled(dr))

ASSERTION A1
  Forall c: Citizen (
    Forall r: Response (r.receiver = c -> ! r.result) ->
      Forall rs: ReceiveService (rs.actor = c -> ! Fulfilled(rs)))

ASSERTION A2
  Forall dr: DoRequest (
   (Exists ra: ReceiveAssistance (ra.depender = dr.actor & Fulfilled(ra)
    & Forall r: Request (r.sender = dr.actor & r.need = dr.need -> r.receiver = ra.dependee)))
      -> (F Fulfilled(dr)))
```

Fig. 4. Validation properties on the requirements model.

`InitialRequest` is entered (1st invariant). After `InitialRequest` has ended, sub-task `ProvideInformation` is eventually entered (2nd invariant). And after also this sub-task has ended, `WaitAnswer` is eventually started (3rd invariant). The fourth invariant constrains the value of attribute `result` of the task to the value of the same attribute of sub-task `WaitAnswer` once this sub-task has ended. Finally, the `Fulfillment definition` tells us that the sub-task `WaitAnswer` has to complete before we can consider the `DoRequest` task fulfilled (necessary condition) and that, if `WaitAnswer` has completed, then `DoRequest` will eventually be fulfilled (sufficient condition).

We remark that some temporal constraints are implicit in the semantics of FT and do not need to appear explicitly in the class declarations. For instance, an implicit creation constraint for each sub-goal is that the parent goal has not yet been fulfilled — if the goal has been fulfilled there is no reason to create the sub-goal. Also, we do not allow two identical instances of the same goal for the same actor.

2.3 Business Requirements Validation

In FT it is possible to validate a requirements specification by allowing the designer to specify properties that the requirements model is supposed to satisfy. We distinguish between `Assertion` properties, which describe conditions that should hold for all valid evolutions of the specification, and `Possibility` properties, which describe conditions that should hold for at least one valid evolution.

Figure 4 reports an excerpt of desired properties for the considered case-study. Possibility P1 aims at guaranteeing that the set of constraints of the formal business requirements specification allow for the fulfillment of the task of doing a request in some scenario of the model. Assertion A1 requires that it is not possible for the citizen to fulfill its goal of receiving assistance services unless a positive answer to a request from the sanitary agency has been received. Finally, assertion A2 requires that the task of doing a request is eventually fulfilled along every scenario under the condition that: there is a sanitary agency that is bounded to provide assistance to the user (citizen's dependency `ReceiveAssistance`); and, the citizen sends the requests to that particular sanitary agency.

The verification of the FT business requirements model against the properties specified is performed with the T-Tool [5]. The T-Tool uses symbolic model checking techniques to perform the verification. It is based on the NuSMV [2]

state-of-the-art symbolic model checker. The T-Tool translates an FT specification into the input language of NuSMV, which is then asked to perform the actual verification. Since model checking requires a finite state model, for translation purposes, upper bounds need to be specified to the number of instances of the different classes that appear in the formal specification. Given these bounds, a finite state automaton is built. Its states describe valid configurations of class instances, according to the class signatures and attributes that appear in the formal specification. Its transitions define valid evolutions of these configurations according to some generic constraints that capture the semantics of FT, e.g., that constant attributes should not change over time, or that, once fulfilled, a goal stays fulfilled forever. The creation, invariant, and fulfillment constraints of the various classes are collected in a set $\{C_i \mid i \in I\}$ of temporal constraints. In this way, the valid behaviors of a model are those executions of the finite-state automaton that satisfy all temporal constraints C_i. Checking if assertion A is valid corresponds to checking whether the implication $\bigwedge_{i \in I} C_i \Rightarrow A$ holds in the model, i.e., if all valid scenarios also satisfy the assertion A. Checking if possibility P holds amount to check whether $\bigwedge_{i \in I} C_i \wedge P$ is satisfiable, i.e., if there is some scenario that satisfies the constraints and the property. In both cases, the verification of a property is translated to the verification of an LTL formula. In [5] we have shown how this verification can be performed efficiently using NuSMV.

All the properties in Figure 4 are true on the final version of the formal requirements model of the considered case-study. However, this result has required several revision steps, where both the model and the properties have been adjusted to capture the intended behaviors of the domain. For instance, assertion A2 had a crucial role in the process of precisely defining the mutual expectations incarnated by dependency `ReceiveAssistance`, and captured by the fulfillment constraints specified for this dependency as it can be seen in Figure 3.

3 From Business Requirements to Business Processes

3.1 Adding Process Specifications

In this section we show how to refine the business requirements model described in the previous section into a business process model. The key idea is to associate BPEL4WS code to the high-level tasks of the actors of the domain (e.g., task `DoRequest` of actor `Citizen`, or task `HandleAssistanceRequest` of the `SanitaryAgency`).

The formal business requirements model already contains several pieces of information that can be exploited to generate a BPEL4WS specification. For instance, it is possible to automatically generate the definition of messages, ports, and services for the business domains — these elements define the WSDL document associated to the BPEL4WS specification. The description of the process model has to be completed by defining the body of the business process corresponding to the task. In Tropos4WS, this is achieved by associating to the task a business process

```
<variables>
  <variable name="need" messageType="Need"/>
  <variable name="result" type="boolean"/>
  <variable name="vRequest" messageType="Request"/>
  <variable name="vInfoRequest" messageType="InfoRequest"/>
  <variable name="vInfo" messageType="Info"/>
  <variable name="vResponse" messageType="Response"/>
  <variable name="waitResponse" type="boolean"/></variables>

<sequence name="DoRequestBody">
  <assign name="Initialization" event="Create ir: InitialRequest (ir.super = self)">
    <copy><from expression="true()"/><to variable="waitResponse"/></copy>
    <copy><from variable="need"/><to variable="vRequest" part="need"/></copy></assign>
  <invoke name="SendRequest" operation="oRequest" inputVariable="vRequest"/>
  <empty name="PhaseSwitch"
    event="Fulfill ir: InitialRequest (ir.super = self) & Create pi: ProvideInformation (pi.super = self)"/>
  <while name="Cycle" condition="getVariableData('waitResponse')">
    <pick name="WaitMessage">
      <onMessage name="InfoRequest" operation="oInfoRequest" outputVariable="vInfoRequest">
        <sequence name="AnswerToInfoRequest">
          <assign name="PrepareInfo">
            <copy><from variable="vInfoRequest" part="query"/>
                  <to variable="vInfo" part="query"/></copy></assign>
          <invoke name="Info" operation="oInfo" inputVariable="vInfo"/>
        </sequence></onMessage>
      <onMessage name="Response" operation="oResponse" outputVariable="vResponse"
         event="Fulfill pi: ProvideInformation (pi.super = self) & Create wa: WaitAnswer (wa.super = self)">
        <assign name="LeaveLoop">
          <copy><from expression="false()"/><to variable="waitResponse"/></copy>
          <copy><from variable="vResponse" part="result"/><to variable="result"/></copy></assign></onMessage>
    </pick>
  </while>
  <empty name="DoRequestFulfilled" event="Fulfill wa: WaitAnswer (wa.super = self)"
     constraint="Forall wa: WaitAnswer (wa.super = self -> G (wa.result <-> self.result))"/>
</sequence>
```

Fig. 5. BPEL4WS process for task `DoRequest` of actor `Citizen`.

defined in the BPEL4WS language. For instance, the business process corresponding to the task of submitting a request is described by the BPEL4WS specification in Figure 5.

The process contains the variables `need` and `result`, which are already present in the formal requirements specification, and the additional variables `waitResponse`, `vRequest`, `vInfoRequest`, `vInfo`, and `vResponse`. The process behaves as follows. First, an initialization step is performed, during which the variable `waitResponse` is set to true, and the message `Request` is prepared by setting its `need` field. The `Request` message is sent in the following ⟨`invoke`⟩ command. A ⟨`while`⟩ loop is then entered, and its body is repeated until variable `waitResponse` becomes false. The body consists of a ⟨`pick`⟩ instruction which suspends the execution of the process until a `InfoRequest` or a `Response` message is received. If a `InfoRequest` message is received, a corresponding `Info` message is prepared and sent. The emitted `Info` message refers to the `query` contained in the received `InfoRequest` message. If a `Response` message is received, then the `result` variable of the process is set to reflect the `result` field of the received message. Moreover, the `waitResponse` variable is set to false, so that we can exit from the ⟨`while`⟩ loop.

Some additional attributes, which are specific of Tropos4WS, are added to the BPEL4WS commands. These attributes are used to connect the evolution of the BPEL4WS process with the evolution of the requirements model. The `event` attributes describe which sub-tasks of `DoRequest` are supposed to be created or fulfilled in the requirements model when a given point is reached in the BPEL4WS code. For instance, sub-task `InitialRequest` is created during the initialization step and is fulfilled after the `Request` message has been sent (the BPEL4WS com-

mand ⟨empty⟩ is used to place this fulfillment event in the right position of the process). The constraint attributes define additional constraints between the requirements layer and the process layer. They are typically used to define the values of the attributes of the sub-tasks. For instance, the constraint attribute of Figure 5 binds the value of attribute result of the WaitAnswer sub-task to the value of variable result of the BPEL4WS process.

3.2 Business Processes Verification

The definition of business processes, together with the bindings that link them to the corresponding tasks and messages in the formal requirements model, allow for different forms of verification. A first possibility consists of re-checking the formal queries that appear in Figure 4 on the more detailed model. Another possibility is checking that the refined model satisfies the requirements described by the Creation, Invariant, and Fulfillment constraints enforced in the requirements model for task DoRequest and its sub-tasks.

To support these kinds of verification, we have extended the T-Tool with a translation of BPEL4WS processes in NuSMV finite state machines. At the time of writing, this translation considers only a restricted subset of BPEL4WS, which covers all the constructs used in Figure 5, but does not include flow commands, event-, fault-, compensation-handlers, and correlation sets. In the translation, the current position in the execution of the BPEL4WS process is traced using a pc variable, ranging over the name attributes associated to the commands in the BPEL4WS code. Transitions between these states are defined according to the semantics of the BPEL4WS constructs. Fairness conditions are added to the finite state machine in order to guarantee that the process eventually progresses whenever the next action to be executed is not blocked. In the case of the process in Figure 5, for instance, the only point where the process can be blocked forever is on the ⟨pick⟩ action, and only if no InfoRequest and Response messages are received. The event and constraint attributes of the BPEL4WS commands are mapped into the set of temporal logic constraints that restrict the valid behaviors of the finite state machine.

By applying this approach to the verification of the BPEL4WS process of Figure 5 we obtain that all verification tasks are successful, and hence this process is a correct implementation of the requirements of task DoRequest. If we modify the code of the process, e.g., by disallowing the reception of one of the two message in the ⟨pick⟩ command, then the verification detects problems. If we disallow the reception of the InfoRequest message, for instance, assertion A2 turns out to be false. Indeed, if the sanitary agency is requesting some information, the citizen is not able to answer to the request and a deadlock in the process is reached. The counter-example of Figure 6 is generated in this case. If we disallow the reception of the Response, not only assertion A2, but also possibility P1 becomes false. Indeed, if we do not receive the response, it is not possible to fulfill DoRequest.

We remark that the approach described in this paper allows also for another kind of verification. Namely, in order to check that the process model is correct,

```
-- Assertion A2 is false as demonstrated by the
-- following execution sequence
-> State 3.1 <-
    Citizen_1.exists = 1
    AssistanceNeed_1.exists = 1
    BeingAssisted_1.exists = 1
    BeingAssisted_1.actor = Citizen_1
    BeingAssisted_1.fulfilled = 0
    DoRequest_1.exists = 1
    DoRequest_1.actor = Citizen_1
    DoRequest_1.fulfilled = 0
    DoRequest_1.waitResponse = 0
    DoRequest_1.answer = 0
    DoRequest_1.pc = Initialization
    DoRequest_1.need = AssistanceNeed_1
    DoRequest_1.super = BeingAssisted_1
-> State 3.2 <-
    SanitaryAgency_1.exists = 1
    InitialRequest_1.exists = 1
    InitialRequest_1.actor = Citizen_1
    InitialRequest_1.fulfilled = 0
    InitialRequest_1.super = DoRequest_1
    Request_1.exists = 1
    Request_1.need = AssistanceNeed_1
    Request_1.initiator = InitialRequest_1
    Request_1.sender = Citizen_1
    Request_1.receiver = SanitaryAgency_1
    DoRequest_1.waitResponse = 1
    DoRequest_1.pc = SendRequest
-> State 3.3 <-
    InitialRequest_1.fulfilled = 1
    ProvideInformation_1.exists = 1
    ProvideInformation_1.actor = Citizen_1
    ProvideInformation_1.super = DoRequest_1
    ProvideInformation_1.fulfilled = 0
    DoRequest_1.pc = PhaseSwitch
    Query_1.exists = 1
    ProvideFairAssistance_1.exists = 1
    ...
    ProvideFairAssistance_1.fulfilled = 1
    ...
    HandleAssistanceRequest_1.exists = 1
    ...
    HandleAssistanceRequest_1.fulfilled = 1
    ReceiveRequest_1.exists = 1
    ...
    ReceiveRequest_1.fulfilled = 1
    EvaluateRequest_1.exists = 1
    ...
    EvaluateRequest_1.fulfilled = 0
-> State 3.4 <-
    ReceiveAssistance_1.exists = 1
    ReceiveAssistance_1.dependee = SanitaryAgency_1
    ReceiveAssistance_1.depender = Citizen_1
    ReceiveAssistance_1.fulfilled = 0
    DoRequest_1.pc = Cycle
-> Loop starts here <-
-> State 3.5 <-
    InfoRequest_1.exists = 1
    InfoRequest_1.query = Query_1
    InfoRequest_1.ref = Request_1
    InfoRequest_1.sender = SanitaryAgency_1
    InfoRequest_1.receiver = Citizen_1
    ReceiveAssistance_1.fulfilled = 1
    DoRequest_1.pc = WaitMessage
-> Loop <-
```

Fig. 6. An example of counter-example generated by NuSMV.

one can show that it is equivalent to the requirements model according to a suitable behavioral equivalence. The NuSMV verification engine, however, does not support this kind of verification.

4 Future Work and Concluding Remarks

This paper has outlined a methodology for the design and verification of Web services as processes generated from business requirements models. The latter are expressed with a language, called Tropos, whose formal counterpart allows for the precise definition of goals and requirements of the actors of the domain. A set of formal techniques are used first to derive process skeletons in BPEL4WS, and then to verify that the refinements performed by the designer still satisfy the requirements.

A number of other approaches that use formal techniques for the definition and composition of Web services are being proposed in the literature (see [4,6,7,8] to cite a few). Distinguishing feature of the approach presented here is that we start from a higher-level, strategic domain model, where processes as such are represented at a very abstract level and other types of requirements – for instance, general business rules on resource usage or engagement with other partners – can be easily represented. This gives us more flexibility in composing processes, and let us perform a wider range of verifications than directly starting from a business process or from elementary service definitions.

The work presented here is our first step towards a long term vision where formal techniques are applied during the entire life cycle of services, from requirements analysis to execution. The objective is providing agents with sufficient se-

mantic knowledge and capabilities to discover and adapt to services and processes, and possibly provide feedback to designers for remodeling.

In the short term, we plan to experiment with model checking tools different from NuSMV. In particular, we plan to adopt verification tools that are based on π-calculus (e.g., [3,9]). This will allow for modeling BPEL4WS features (most notably, dynamic creation of new execution threads) that are difficult to model in NuSMV. Moving to tools based on π-calculus would also allow for the application of equivalence checking techniques to compare the business process model wrt the corresponding requirements model. The most serious obstacle in this direction is that the property specification languages currently available for the π-calculus are not adequate for expressing the Formal Tropos constraints.

In the longer term, we will investigate into improving the generation of BPEL4WS process skeletons in order to capture more details from the domain model, such as the type of long-term business transaction that is required. An improved BPEL4WS process should also enable an execution engine to relate faults and exceptions to specific goals or requirements of the domain model, in order to take appropriate action or provide feedback to the user.

References

[1] T. Andrews, F. Curbera, H. Dholakia, S. Systems, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services, version 1.1, 2003.

[2] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. CAV'02, LNCS 2004*. Springer Verlag, 2002.

[3] G. Ferrari, S. Gnesi, U. Montanari, and M. Pistore. A model checking verification environment for mobile processes. *ACM Transactions on Software Engineering and Methodology*, To appear.

[4] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proc. ASE'03*, 2003.

[5] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in Tropos. *Requirements Engineering*, 2003. To appear.

[6] J. Koehler, R. Hauser, S. Kapoor, F. Wu, and S. Kumara. A model-driven transformation method. In *Proc. EDOC'03*, 2003.

[7] S. Nakajima. Model-checking verification for reliable web service. In *Proc. OOPSLA'02 Workshop on OOWS*, 2002.

[8] S. Narayanan and S. Mcllraith. Simulation, verification and automated composition of web services. In *Proc. WWW'02*. ACM, 2002.

[9] B. Victor and F. Moller. The mobility workbench — a tool for the π-calculus. In *Proc. CAV'94, LNCS 818*. Springer Verlag, 1994.

[10] E. Yu. Towards modeling and reasoning support for early requirements engineering. In *Proc. RE'97*. IEEE Computer Society, 1997.