# Formal Tropos for Web Service Compositions

ITC-irst, Università di Trento

**Abstract.** In order to manage the changes and to enable continuous adaptation of the business models to the market needs, the service-based distributed processes should be linked to the motivating business goals and requirements. In this work we present a framework for representing goals and requirements, and for integrating them with the process models. Formal analysis techniques are exploited within the framework to ensure the correctness of the strategical and process models, and to check the correspondence between them.

| | |
|---|---|
| Document Identifier | Deliverable D3.2 |
| Project | MIUR-FIRB project RBNE0195K5 “Knowledge Level Automated Software Engineering” |
| Version | v1.0 |
| Date | October 31, 2006 |
| State | Final |
| Distribution | Public |

**Acknowledgements.**

This document is part of a research project funded by the FIRB 2001 Programme of the “Ministero dell’I-struzione, dell’Università e della Ricerca” as project number RBNE0195K5.

The partners in this project are: Istituto Trentino di Cultura (Coordinator), Università degli Studi di Trento, Università degli Studi di Genova, Università degli Studi di Roma “La Sapienza”, DeltaDator S.p.A..

# Executive Summary

Service-oriented architectures and Web service infrastructure provide the ideal framework for interconnecting organizations and for defining distributed business applications. The possibility to exploit business process definition and execution languages is particularly relevant for capturing the process-oriented nature of these applications. However, business processes by themselves are not enough to manage the changes and to allow an organization to continuously adapt its business model to the typical needs of distributed applications. Business goals and business requirements need to be made explicit in the business process model in order to determine the implication of the business goals changes in the software system. A "strategic" description needs to be provided of the different actors in the business domain with their goals and with their mutual dependencies and expectations. Furthermore, a tight integration of the business requirements and of the business process specification is necessary. This permits requirements traceability, i.e., to see how the modifications of requirements or process specifications affect each other. To achieve this, automatic formal analysis and verification tools are necessary to verify the matching between the strategic and the process models and to check that this match is maintained when requirements or processes change.

In this document we propose a framework for representing strategies and goals of an organization in terms of business requirements. The framework allows to describe how an organizational strategy is operationalized into activities and implemented by Web services and business processes. Formal annotations are used at all levels to define constraints to business requirements and to business process models. Finally, this framework allows for the usage of formal analysis techniques, in particular Model Checking, to pinpoint problems and to identify possible solutions in this domain. We also present a prototype tool that aims at implementing the presented framework, and the underlying analysis techniques.

# Contents

# Chapter 1

# Introduction

The growing use of information technology and the high level of dynamics and interoperability introduced by the Internet provide new ways for interconnecting enterprises and customers that where not possible in the past. In particular, Web service technology is rapidly emerging as one of the most practical approaches for the integration of customers', vendors', and business partners' applications. The number of individual Web services made available by companies is continuously growing, but the real value to an organization comes only when these companies are able to connect services together. This forces transaction and integration costs to be driven down and makes the traditional static hierarchical structure less attractive both from a strategic and from an economical perspectives. Organizations should be considered in terms of business processes rather than functions, integrating activities in cross-functional and cross-enterprise chains [VCN$^+$01]. This integration is made more difficult by the fact that, while aiming to work together and to provide inter-enterprise business applications, usually companies do not want to uncover the internal structure of their business processes to each other. Moreover, they want to keep the ability to rethink and reorganize their business processes and to modify accordingly the implementation in order to adapt their strategy to changes and innovations.

Web services provide the basic technology for obtaining loosely coupled integration required by business processes. Indeed, business process description languages like Business Process Execution Language for Web Services (BPEL for short, [ACD$^+$03]) provide means to describe services without necessarily revealing the internal structure of the processes. However, in order to be effectively used for composing business processes, the technological infrastructure provided by Web services needs to be enriched with tools and methodologies that support the development of this composition and that permit to manage the changes and adaptations of business models and of business processes. This introduces the necessity of determining the implication of the business strategy and goals changes in the software system. To this purpose, business goals and business requirements need to be made explicit in the business process model. A "strategic" description needs to be provided of the different actors in the business domain with their goals and with their mutual dependencies and expectations. Furthermore, a tight integration of the business requirements and of the business process specification is necessary. This permits requirements traceability, i.e., to see how the modifications of requirements or process specifications affect each other.

One of the major prerequisites for an effective process integration is reliability. The process definition that is obtained from the business requirements model should be consistent with the requirements and with the goals it aims to achieve. And, whenever processes of different partners are composed into a new business application process, the composition should respect the goals and constraints of every participant. In other words, the business application should match the business requirements model. Automatic formal analysis and verification tools are necessary to verify this matching and to check that this match is maintained when requirements or processes change.

BPEL 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 to verify whether the composition of two or more processes satisfies general properties (such as deadlock freedom) or application-specific constraints (e.g., temporal sequences, limitations on resources). Different techniques have been already applied to the verification of business processes (see, e.g., [FUMK03, KHK$^+$03, Nak02, NM02]). 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.

In the document we describe a framework for the integration of business process models with the business requirements models (see also [PRB04, KPR04b, KPR04a] for details). The 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.

The design process starts from a description of the strategic goals and needs of an organization. These are refined and operationalized into tasks and activities, which are then transformed in turn into business processes and Web services. Formal annotations are used at all levels to define constraints to business requirements and to business process models. Our starting point is a modeling language, called *Tropos* [Yu97], whose objective is to capture the business requirements of the actors of a domain, their dependencies and expectations. We also exploit the formal counterpart of Tropos, the Formal Tropos language [FLM$^+$04], that provides rich notations for the definition of constraints and properties of the modeled domain, and that is amenable for formal verification. Both Tropos and Formal Tropos are extended to target the specific aspects of Web services technology. In particular, to deal with business processes, BPEL specifications are linked to the requirements models.

In order to carry out the verification and validation of the business process and business requirements models, we present a formal analysis approach based on model checking techniques. More precisely, we present the encoding of the Formal Tropos requirements models and BPEL processes in terms of formal specification languages, namely Promela [Hol03] and NuSMV [CCG$^+$02]. Promela is a formal specification language for distributed systems which offers communication and concurrency primitives inspired by process algebras. Moreover, its specifications can be verified using the SPIN [Hol03] model checker. Promela allows for a description of Web service processes which is similar to the one that can be written in languages like BPEL. The NuSMV language is the input language of the corresponding model checker, which is particularly suitable for the formal analysis of the logically based descriptions of the system, e.g., requirements specifications. In [FLM$^+$04] is shown how the requirements models can be efficiently verified using NuSMV model checker.

The verification of the formal business requirements and business process model is made by a prototype tool that performs translation of the models into the specification language of the corresponding model checker in order to carry out the verification tasks. The implementation of encoding of the Formal Tropos requirements in the Promela specification requires a definition of a novel procedure in order to tackle the state-space explosion problem. We present such a procedure that translates separately different constraints that appear in the Formal Tropos specification and to join the generated pieces into a special process aimed to check constraint satisfaction at each system step. In order to evaluate the efficiency of the presented approach, we have conducted a set of experiments on a small case study. The results of the experiments demonstrates that the approach is effective in revealing possible flaws both in the strategic and in the process models. They also suggest to apply the NuSMV-based reasoning for the validation of requirements models, while advertising the SPIN-based analysis for the verification of business processes against requirements models.

# Chapter 2

# Business Requirements and Business Processes

## 2.1 Business Processes

The description of the structure and behavior of the different business activities within an organization has been deeply investigated, for instance in the framework of business process modeling. However, the integration of business processes that are distributed among different organizations is still a challenging problem.

Web services are an emerging technology for building complex distributed systems focusing on interoperability, support for efficient integration of distributed processes, and uniform representation of applications. They allow companies to describe the external structure of their processes and how they can be invoked and composed. Web service support the interactions among the different partners by providing a model of synchronous or asynchronous exchange of messages. These exchanges of messages can be composed into longer business interactions by defining protocols constraining the behavior of all partners.

The terms orchestration and choreography are often used to refer to the two key aspects of process composition [Pel03]. In *orchestration*, the composition is considered from the perspective of one of the business parties. The focus is on the interaction that the business process under consideration performs with internal and external Web services in order to carry out its task. Orchestration is usually private to the business party, since it contains reserved information on the specific way a given process is carried out. *Choreography*, on the other hand, describes the interactions for a global, neutral perspective, in terms of valid conversations or protocols among the different parties. Choreography is usually public, since it defined the common rules that define a valid composition of the distributed business processes in the business domain.

Web services have developed different languages for orchestration and choreography

(BPEL, WSFL, WSCI, etc). Among them, BPEL [ACD$^+$03] is quickly emerging as the language of choice for the description of process interactions. BPEL provides core concepts for the definition of business process in an implementation-independent way. It allows both for the definition of internal business processes and for describing and publishing the external business protocol that defines the behavior of the interaction. Therefore, BPEL permits to describe both the orchestration and the choreography of a business domain with an uniform set of concepts and notations.

## 2.2 Business Requirements

While developing distributed business services, the designers usually focus on the "how", that is on the way a business process is implemented by means of standard business process description languages. However, these languages are not able to describe the business goals and strategies of an organization, its expectations over external services, and the links existing between these goals and expectations and the corresponding business processes.

By business requirements we mean all those aspects of the description of business process that are related to the strategy and the rationale of on organization (the "why" and the "what"), and that precede and motivate the definition of specific processes (the "how").

Similarly to what happens for business processes, also in business requirements we can distinguish an orchestration and a choreography perspective. In the *orchestration* perspective, the point of view of a particular business party is taken, its strategic goals are described, the definition of the business processes needed to achieve these goals is made evident along with the decision of what services to implement internally and what external services to exploit. The requirements from the *choreography* perspective, on the other hand, aim to describe the interaction opportunities that are present in the business domains. These opportunities are defined in terms of possible matches between the services required by certain actors and the services provided by other actors. Along with these intent/offer matches, the choreographical business requirements also define the business rules of the domain, namely the shared assumptions and constraints on the correct interactions inside the business domains. Orchestration is supposed to be private to the party, as it contains information on the business strategy that the party may not want to disclose to external organizations, while choreography is supposed to be publicly available to all participants. Correct business processes of a party should respect goals and constraints both of its own orchestration and of the shared choreography of the business domain.

### 2.2.1 A Case-Study

We consider a case-study in the field of public welfare, extracted from a larger domain concerning the local government of Trentino (Italy). In the case-study we consider a senior citizen that aims at being assisted, e.g., receiving services like transportation or meals at home. The assistance to citizens is provided by a Health-care Agency, that is run by the Local Government, and that aims at providing fair assistance to citizens. The Health-care Agency depends on external providers for the actual delivery of the requested services. The financial aspects of the Local Government are handled by a Bank that is in charge of paying the service providers and of asking the citizen to cover part of the costs of the used services. The interaction among the different parties is required to happen via Web services. The business goal of the citizen consists in receiving assistance, while the Health-care Agency is willing to provide assistance only to citizen that satisfy given eligibility criteria.

## 2.3 Business Requirements Modeling in Tropos

Tropos [CKM02] is a goal-driven, agent-oriented software development methodology that aims to cover all the phases of the development process starting from early requirements. It is founded on the premise that during early requirements 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. The methodology allows for the incremental refinement of the strategic models via goal/task decomposition and operationalization both at the informal and formal level.

In this work we show that (a suitable extension of) Tropos can be used to define business requirements and to integrate them with the corresponding business processes. We will use the health-care assistance case-study to illustrate the approach.

### 2.3.1 Tropos for Web Services

Figure 2.1 is a Tropos diagram that describe the *actors* (circles) participating to the case-study, and their strategic high-level *goal*s (the ovals attached to the actors). For instance, in the diagram we have the Citizen that aims at being assisted; the HealthcareAgency that aims at providing a fair assistance to the citizens; the ServiceProvider which goal is to provide the requested services; and the Bank which handles the government's finances.

Tropos allows for the description of the interactions among the different parties of the domain at the strategic level relying on the intent/offer matching mechanism represented in the diagram by means of *dependencies* (the ovals linked to two different actors).

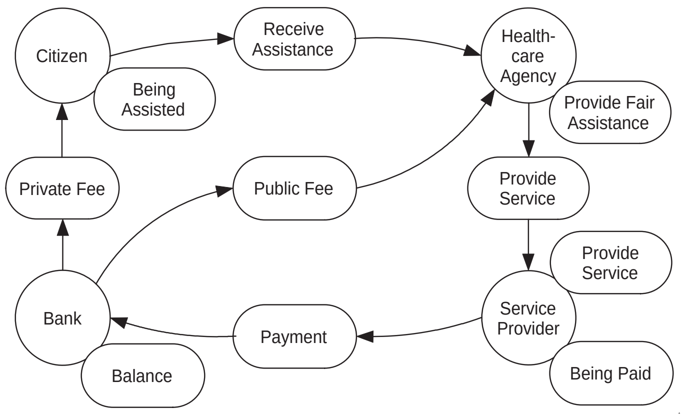

Figure 2.1: High level requirements model.

For instance, the Citizen depends on the Health-care Agency for being assisted, and this is formulated in the model with dependency ReceiveAssistance from Citizen to Health-careAgency. This diagram can be seen as a very high-level choreography representation of the requirements of our case-study.

Starting from this high-level view of the organizational or business system, Tropos proceeds with an incremental refinement process (see Figure 2.2). This refinement starts with a goal analysis, where the high level goals of one of the actors are refined into sub-goals and eventually operationalized into tasks. In our case-study, the goal analysis is very simple: the Citizen refines the goal BeingAssisted into the two tasks (hexagons) DoRequest and Pay, the goal ReceiveService, and the soft-goal (cloud) QualityService. The tasks are supposed to be implemented by software modules, while the goals that remain in the model after the goal analysis represent activities that are not carried out electronically (e.g., the assistance services are physically delivered to the citizen). Finally, soft-goals are used to describe non-functional requirements, with no clear-cut criteria as to when they are achieved (e.g., the citizen has some requirements on the quality of the services delivered to him).

The goal analysis phase is followed by a task refinement phase, where the high-level tasks are decomposed into sub-tasks. In Figure 2.2, task DoRequest is further refined into InitialRequest, ProvideInformation, WaitAnswer. In this simple case, the three sub-tasks are composed sequentially. Other forms of task decomposition are also possible, corresponding to the other typical ways of combining activities in activity diagrams (parallel composition, choice, iteration...).

The task decomposition procedure ends once we have identified all basic tasks that define the business process. As a last step in the definition of business requirements, we associate to the basic tasks the messages that describe the basic interactions among actors. For instance, task InitialRequest requires to send a message Request to the Health-careAgency. This message is received and processed by the task ReceiveRequest of the HealthcareAgency. The task AskAdditionalInfo of the HealthcareAgency is implemented

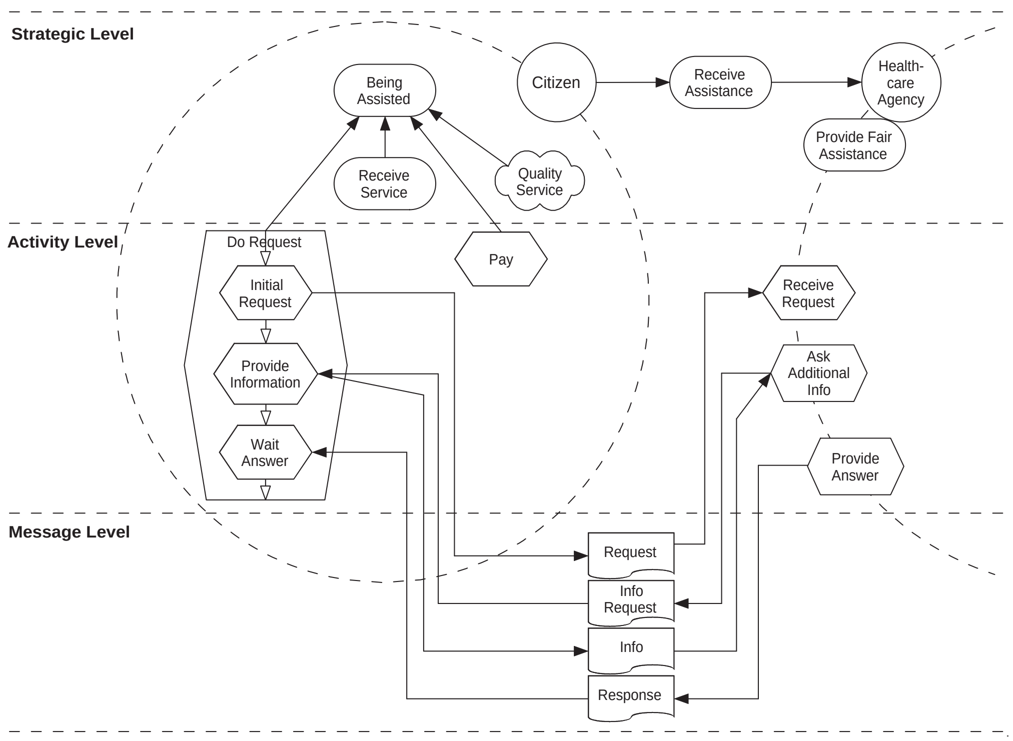

Figure 2.2: Requirements model refinement.

by sending a message InfoRequest to the Citizen which receives and processes it within task ProvideInformation and responds with an Info message. Once sufficient information has been gathered, the HealthcareAgency sends a Response message to the Citizen.

The steps of the refinement are represented in the Figure 2.2 as three levels: a strategic level, an activity level and a message level. All these levels are part of the requirements model, in the sense that they define different aspects of the requirements a valid implementation is supposed to respect.

We remark that Figure 2.2 represents the point of view of the Citizen, therefore it can be seen as an “orchestration” requirements diagram. Clearly, the “internal” refinement done for goal BeingAssisted has to take into account and to reflect the “contract” that governs the way the assistance is provided to the Citizen by the HealthcareAgency (and by the other actors of the diagram). Indeed, the diagram in Figure 2.2 shows also the links that exist between the Citizen and the HealthcareAgency, both at the goal level and at the message level. However, no description of the internal structuring of the HealthcareAgency is represented in the diagram, since this information is not available to the Citizen.

It is worth to be noticed that in this example we have adopted a top-down strategy for transforming high-level goals into interactions. Other strategies, e.g., a bottom-up approach from the messages to the goals, or a middle-out approach starting from the activities, are also possible in this framework.

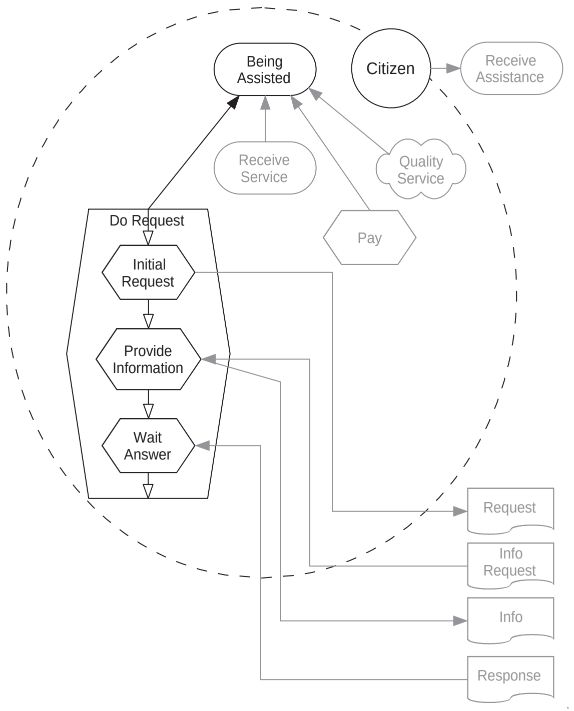

Figure 2.3: Citizen Tropos diagram.

### 2.3.2 Formal Business Requirements Specification

The Tropos methodology allows for extending the Tropos diagrams with formal annotations expressed in Formal Tropos (hereafter FT). The FT annotations specify the valid behaviors and the relations among the different actors, dependencies, goals, tasks and messages in the model. At the strategic level the FT annotations specify the conditions on goal creation and fulfillment, and assume/guarantee conditions on delegations. At the activity level, they define pre- and post-conditions on tasks and sub-tasks. Even more important, FT annotations allow to link these two levels and the underlying message level. The key advantage of FT with respect to other approaches is that it defines the dynamic aspects of a model and supports its formal verification already at the requirements level, without requiring an operationalization of the specification, e.g., into BPEL processes. A precise definition of FT and of its semantics can be found in [FLM$^+$04]. Here we present the most relevant aspects of the language based on the case-study. The excerpt of the Tropos diagram of the Citizen is represented in Figure 2.3, while the corresponding FT specification can be found in Figure 2.4.

FT gives a description of the different objects in the modeled domain, which is similar to a class declaration. In particular, a list of attributes is associated to each of these classes. Each attribute has a *sort* which can be either primitive (boolean, integer…) or can be a reference to other class. For instance, boolean attribute result of tasks DoRe-

**Goal Dependency** ReceiveAssistance **Mode maintain**  
**Depender** Citizen  
**Dependee** HealthcareAgency  
**Fulfillment condition** $\forall$ dr: DoRequest (  
(dr.**actor** = **depender** $\wedge$ **Fulfilled** (dr) $\wedge$ dr.result) $\rightarrow$  
**F** $\exists$ rs: ReceiveService (rs.**actor**=**depender** $\wedge$ **Fulfilled** (rs)))

**Task** DoRequest **Mode achieve**  
**Super** BeingAssisted  
**Actor** Citizen  
**Attribute** result:**boolean**  
**Fulfillment definition**  
$\exists$ wa:WaitAnswer(wa.**super** = **self** $\wedge$  
**Fulfilled** (wa) $\wedge$ (result $\leftrightarrow$ wa.result))

**Task** InitialRequest **Mode achieve**  
**Super** DoRequest  
**Actor** Citizen

**Task** ProvideInformation **Mode achieve**  
**Super** DoRequest  
**Actor** Citizen  
**Fulfillment definition**  
**G** ($\forall$ ir: InfoRequest(**Received** (ir) $\rightarrow$ $\exists$ i: Info(**Sent** (i)))

**Task** WaitAnswer **Mode achieve**  
**Super** DoRequest  
**Actor** Citizen  
**Attribute** result:**boolean**  
**Fulfillment definition**  
$\exists$ r:Response(**Received** (r) $\wedge$ (result $\leftrightarrow$ r.result))

Figure 2.4: Formal Tropos specification.

quest and WaitResponse, and the message Response determines whether the response to the request of the citizen has been positive or not. 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.

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 health-care agency has to be maintained over time. On the other hand, task DoRequest (as most of the tasks) is of type achieve, since the citizen aims at reaching a state where this task is terminated.

Behavioral aspects of the objects in the model and relations among them are annotated in the FT specifications as constraints. They allow to capture the conditions on the goals creation and fulfillment and pre- and post-conditions on tasks and sub-tasks. **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. **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. Moreover, special predicates can appear in the FT temporal logic formulas (i.e., **JustCreated** (t), **Fulfilled** (t), or **JustFulfilled** (t)). Additionally, for message classes **Received** (t) and **Sent** (t) predicates may be used.

In the FT specification of Figure 2.4, the definitions of task DoRequest and its sub-tasks model the life-cycle of the activity. For instance, the fulfillment definition of ProvideInformation specifies that this task aims to send an information message in reply of every incoming information request and it is fulfilled if all information requests have been answered. The DoRequest task is considered to be fulfilled whenever the response message is received (i.e., the WaitResponse task is fulfilled) and the value of the result attribute corresponds to the one contained in the message.

```
Possibility P1
/* It is possible to fulfill request */
 ∃ dr: DoRequest (Fulfilled (dr))

Assertion A1
/* Service is received only upon a positive response */
 ∀ c: Citizen (
    ∀ r: Response (Received (r) ∧ r.receiver = c → ¬ r.result) →
       ∀ rs: ReceiveService (rs.actor = c → ¬ Fulfilled (rs)))

Assertion A2
/* If some agency provides assistance and the citizen requests a service then the request should be fulfilled */
 ∀ dr: DoRequest (
    ∃ ra: ReceiveAssistance (ra.depender = dr.actor ∧ Fulfilled (ra) ∧
       ∀ r: Request (r.sender = dr.actor → r.receiver = ra.dependee)) →
    F Fulfilled (dr))
```

Figure 2.5: Formal Tropos Properties

Although the FT annotations are very expressive, in the typical cases only a limited amount of the expressive power of FT is actually used. For instance, pre-/post-conditions are typically propositional. However, in some cases it is useful to have the possibility of expressing more complex conditions. for instance, in the case of the post-condition of the ProvideInformation task we use temporal operator **G** to specify that this task can only be considered fulfilled once all information requests have been processed. This high-level condition is appropriate for the requirements model even if a specific mechanism will need to be used in the implementation to actually check when information request messages are ended (see Section 2.4). We remark that some temporal constraints are implicit in the semantics of FT and do not need to appear explicitly as annotations. 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 the order in which the sub-tasks of a given composed task are invoked is implicit in the FT semantics.

In an FT model, we can also specify properties that are desired to hold in the domain, so that they can be verified with respect to the model. We distinguish between **Assertion** properties which are desired to hold for all valid evolutions of the FT specification, and **Possibility** properties which should hold for at least one valid scenario. In Fig. 2.5 some properties for the case-study are reported.

For lack of space we refer to [tec03] for a complete description of the semantics of FT. Here we recall only that the valid behaviors of a model are those sequences of states that respect a set $C_i$ with $i \in I$ of temporal constraints. These constraints are obtained directly from the **Invariant**, **Creation**, and **Fulfillment** declarations that appear in the FT model. 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 an FT specification is translated to the verification of an LTL formula.

## 2.4 Integrating Business Requirements and Processes

The Web service business process specification may be easily derived from the business requirements model. The FT model already contains several pieces of information that can be exploited to generate a BPEL 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 BPEL specification. The description of the process model has to be completed by defining the body of the business process corresponding to the task. In our framework, this is achieved by associating to the task a business process defined in the BPEL language. For instance, the business process corresponding to the task of submitting a request is described by the BPEL specification in Figure 2.6.

Besides the result variable, which is already present in the formal requirements specification, the process contains 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.

In order to fulfill the requirement that all incoming information requests should be satisfied until an answer has been received by the health-care agency, a ⟨`while`⟩ loop is entered. Its body is repeated until variable waitResponse becomes false. The body of the loop consists of a ⟨`pick`⟩ instruction which suspends the execution of the process until a InfoRequest or a Response message is received. Every incoming information request, arrived with a InfoRequest message, is answered with a corresponding Info message. 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. When this message is received, the citizen does not expect any other information requests and the PorvideInformation task can be considered completed. Therefore, variable waitResponse is set to false, so that the ⟨`while`⟩ loop is left.

Some additional attributes, which are specific for FT, are added to the BPEL commands. These attributes are used to connect the evolution of the BPEL 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 BPEL code. For instance, sub-task InitialRequest is created during the initialization step and is fulfilled after the Request message has been sent (the BPEL command ⟨`empty`⟩ is used to place this fulfillment event in the right position of the

```
<sequence name="DoRequestBody">
  <assign name="Initialization"
    event="Create ir:InitialRequest(ir.super=self)">
    <copy>
      <from expression="true()"/>
      <to variable="waitResponse"/>
    </copy>
  </assign>
  <invoke operation="oRequest" inputVariable="vRequest"/>

  <empty name="PhaseSwitch"
    event="Fulfill ir:InitialRequest(ir.super=self) &
           Create pi:ProvideInformation(pi.super=self)"/>

  <while condition="getVariableData('waitResponse')">
    <pick name="WaitMessage">

      <onMessage operation="oInfoRequest"
                 variable="vInfoRequest">
        <reply operation="oInfo" variable="vInfo"/>
      </onMessage>

      <onMessage operation="oResponse" variable="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>
```

Figure 2.6: BPEL process for task **DoRequest** of actor **Citizen.**

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 2.6 binds the value of attribute result of the WaitAnswer sub-task to the value of variable result of the BPEL process.

# Chapter 3

# Encoding of Requirements and Process Models

The integrated business requirements and business process models allow for different forms of verification. In order to carry out these verification tasks, the BPEL and the FT specification should be translated into a format accepted by a formal analysis tool, such as model checker. Here we consider the translation procedure that considers encoding of the BPEL processes and FT specification in the languages accepted by NUSMV and SPIN model checkers.

## 3.1 Business Process as State Transition System

In our framework, we encode BPEL processes as *state transition systems* which describe dynamic systems that can be in one of their possible *states* (some of which are marked as *initial states*) and can evolve to new states as a result of performing some *actions*. We model process interactions as *external actions* defined on a set of operations (or message types) $M$. Following the standard approach in process algebras, external actions are distinguished in *input actions*, which represent the reception of message of type $\alpha \in M$, denoted as $\overleftarrow{\alpha}$, and *output actions*, which represent messages of type $\alpha \in M$ sent to external services, denoted as $\overrightarrow{\alpha}$. We also define a special action $\tau$, called *internal action*, which is used to represent evolutions of the system that do note involve interactions with the external services. A *transition relation* describes how the state can evolve on the basis of inputs, outputs, or of the internal action $\tau$.

**Definition 1** *A* state transition system *(STS) is a tuple $\langle \mathcal{S}, \mathcal{S}_0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \sigma \rangle$ where*

- *$\mathcal{S}$ is the finite set of states and $\mathcal{S}_0 \subseteq \mathcal{S}$ is the set of initial states;*
- *$\mathcal{I}$ is a finite set of input actions and $\mathcal{O}$ is a finite set of output actions;*

- $\mathcal{R} \subseteq \mathcal{S} \times (\mathcal{I} \cup \mathcal{O} \cup \{\tau\}) \times \mathcal{S}$ *is the transition relation;*

- $\sigma : \mathcal{S} \rightarrow 2^{\mathcal{AP}}$ *is a labelling functions that represents the variable valuations in the states of the process.*

Transitions of these state transition systems are defined according to the semantics of the BPEL constructs. Fairness conditions are added to the finite state machines 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 2.6, for instance, the only point where the process can be blocked forever is on the $\langle$`pick`$\rangle$ action, and only if no InfoRequest and Response messages are ever received.

We also remark that the assumption on the finiteness of the states set is required in order to enable the analysis techniques presented in this work.

## 3.2 Encoding Formal Tropos Specifications

The encoding of the FT models is performed by translating the class declarations (i.e., actors, goals, dependencies) and lifetime constraints (i.e., creation and fulfillment conditions) separately. Each class declaration is translated into a corresponding finite state automaton, while each constraint is translated into a corresponding LTL formula. While such encoding is natively supported by the NuSMV model checker (see [FLM$^+$04] for details), the corresponding representation in Promela language is not straightforward and requires a novel encoding procedure, which is presented below.

### 3.2.1 Formal Tropos Class Representation

Figure 3.1 contains the FT definition of task DoRequest and the corresponding Promela encoding. We associate to each FT class a Promela data type (DoRequestType in our case) and an array of elements on this data type (DoRequest in our case). The size of the array defines the maximum number of instances for the class. This bound is necessary to obtain a finite-state model and to apply model-checking techniques. In the example we allow for at most 2 instances of class DoRequest. Attributes of basic sorts (i.e., **boolean**) are translated into the corresponding Promela type. Attributes referring to FT classes (e.g., attribute actor) are declared as fields of type byte. This field is used as index in the array of the referenced class (e.g., array Citizen in the case of attribute actor).

Additional fields are introduced in the user-defined data types to model the life-cycle of the FT class instances. Attribute exists models class creation. Initially it is false and is set to true when the class is created. Attributes justcreated, fulfilled and justfulfilled are used to encode the **JustCreated**, **Fulfilled** and **JustFulfilled** predicates respectively.

```
Task DoRequest                        typedef DoRequestType{
  Actor Citizen                         byte actor;
  Super BeingAssisted                   byte super;
  Attribute result : Boolean            bool result;
                                        bool justcreated, exists;
                                        bool justfulfilled, fulfilled;
                                      }
                                      DoRequestType DoRequest[2];
```

Figure 3.1: DoRequest task representation in FT and Promela

The life-cycle of the class instances is encoded using Promela processes. Figure 3.2 depicts the process corresponding to the DoRequest task. Different instances of the process are used to model the behavior of the different class instances. The byte argument passed to the process defines the index of the specific class instance in array DoRequest.

Depending on the type of the FT class, the life-cycle of the instances consists of different phases, which is reflected in the corresponding process. In particular, **Actor** class instances initially are in a "NotExists" state. A possibility for an instance is to be never created thus terminating the corresponding process. Another possibility is that the instance eventually enters the "Exists" phase. The "NotExists" to "Exists" transition is only enabled if suitable instances exist for the attributes referring to other classes. In this case, the class attributes are initialized, in particular justcreated and exists are set to true. In the "Exists" phase the process nondeterministically modifies values of its non-constant attributes (if any). Additionally, **Actor** classes may nondeterministically create child goals, tasks and dependencies (e.g., instances of goal BeingAssisted in the case of Citizen). The child class attributes are initialized and the corresponding process is started.

Processes modeling the behavior of goals, tasks, and dependencies are quite different. They start already in state "Exists" since they are created "on demand" by their parent actors or goals. Similarly to actors, they also may create child instances when they are in the "Exists" phase (see e.g., the creation of subtask InitialRequest in Fig. 3.2). These instances may nondeterministically move to the "Fulfilled" phase, assigning true to attributes justfulfilled and fulfilled. The values of non-constant attributes can change also in the "Fulfilled" phase. If there are no such attributes the process terminates its execution. We remark that all operations done during a phase transition are performed in an atomic section, in order to respect the FT semantics.

A special routine system_step() is called whenever a process performs a step. This routine performs a set of tasks that need to be done whenever the system evolves. It is responsible to reset the attributes justcreated and justfulfilled of each FT class, so that these flags may be true only in the state that immediately follows the creation or the fulfillment of an instance. As we will see in the following, this routine is also used in the verification of the Promela specification.

```
proctype DoRequestProc(byte id) {
Exists:
   do :: atomic /* if the child subtask is not started yet,
                 assign relevant attributes and start it */
         {(!InitialRequest[0].exists)→ system_step();
          InitialRequest[0].super = id;
          InitialRequest[0].actor = DoRequest[id].actor;
          InitialRequest[0].exists = true;
          InitialRequest[0].justcreated = true;
          run InitialRequestProc(0);};
      . . . /* other child subtask may be started here */
      :: atomic /* Modify non-constant attributes */
         {system_step();
          if :: DoRequest[id].result = true;
             :: DoRequest[id].result = false;
          fi; /* The instance is fulfilled nondeterministically */
          if :: DoRequest[id].fulfilled = false;
             :: DoRequest[id].fulfilled = true;
                DoRequest[id].justfulfilled = true; goto Fulfilled;
          fi;}
   od;
Fulfilled:
   do :: atomic /* Modify non-constant attributes */
         {system_step();
          if :: DoRequest[id].result = true;
             :: DoRequest[id].result = false;
          fi;}
   od;
}
```

Figure 3.2: DoRequest task process definition

### 3.2.2 Representation of Logic Specifications

The logic specifications in FT are exploited to verify assertions and possibilities on the conditions defined by the constraints. The standard solution offered by SPIN for verifying assertions and possibilities consists of generating a “never claim” describing the automaton for the formula to verify (e.g., formula $\bigwedge_{i \in I} C_i \Rightarrow A$ for assertion $A$) and of asking SPIN to verify it. However, this solution turns out to be infeasible. Indeed, the large size of the global formula prevents the possibility of verifying the never claim. Also for rather simple specifications (for instance, a reduced FT specifications with only 5 classes and 3 simple constraints), the C file which is generated by SPIN and that contains the model checking code is so complex that `gcc` fails to compile it (a memory out is obtained even with 1GB available).

To overcome these problems we propose an alternative solution. We encode each

```
proctype constraint_verifier() {
  byte label[n] = 0; bool accepted[n] = false; byte next = 0;
  do :: constraints_done → break;
     :: else atomic
       {all_accepted = true; valid_step = false;
        ...  /* All constraints automata go here */
        valid_step = true; constraints_done = true;
        if :: accepted[next] →  /* Look for acceptance again */
              next_accepted = true; next = (next+1) % n;
           :: else
        fi;}
  od;
```

Figure 3.3: The constraint_verifier process

LTL formula defining a constraint in a separate automaton, and generate a new process constraint_verifier where all these automata are executed in parallel. This process is then added to the final specification. We enforce execution of all automata corresponding to constraints after each system step, and restrict the verification to the execution sequences where all the constraints holds.

The translation of each constraint into an automaton is done using either the built-in LTL2SPIN converter or external tools like LTL2BA [GO01]. These tools generate a Promela "never claim" whose body represents the automaton for the corresponding constraint. All these automata are joined in the body of the special process named constraint_verifier, which guarantees that all of them are executed in turn (see Fig. 3.3). Some modifications on the bodies are necessary before they can be joined. For instance, Fig. 3.4 shows the automaton generated by LTL2BA (on the left) and the modified automaton for constraint G($p \rightarrow$ F $q$), with $n$ being the index of the constraint. The **accept** labels of the automaton represent the accepting states of the automaton. Usually these labels are managed automatically by SPIN, in order to guarantee that acceptance states are visited infinitely often. In our case, acceptance states need to be managed explicitly. Therefore, all **accept** labels are renamed, and a special array accepted is used to store the information whether the automaton is visiting an accepting state. Moreover, a global variable all_accepted stores the information whether *all* automata are visiting an acceptance state. Finally, in order to preserve the position reached by every automaton after each step, this position is stored in a special array named label. A switch at the beginning of the modified body is used to restore the state of the automaton.

Due to the ad-hoc management of acceptance states, some effort is needed in order to restrict the verification to the valid executions. This is achieved if the following conditions are satisfied: *(i)* Whenever any process performs a step, every constraint automaton has to be executed. If some constraint automaton is blocked, the execution path should be excluded from the verification. *(ii)* On every infinite execution path all the constraints

```
                          if
                          :: label[$n$]==0 $\rightarrow$ goto C$n$_accept_init
                          :: label[$n$]==1 $\rightarrow$ goto C$n$_T0_S2
                          fi;
/* $G(p \rightarrow Fq)$ */    /* $G(p \rightarrow Fq)$ */
accept_init:              C$n$_accept_init:
  if                        if
  :: $(\neg p)||q \rightarrow$            :: $(\neg p)||q \rightarrow$ label[$n$] = 0;
    goto accept_init          accepted[$n$] = true;
  :: $(1) \rightarrow$                :: $(1) \rightarrow$ label[$n$] = 1;
    goto T0_S2                accepted[$n$] = false; all_accepted = false;
  fi;                       fi; goto C$n$_checked;
T0_S2:                    C$n$_T0_S2:
  if                        if
  :: $q \rightarrow$                  :: $q \rightarrow$ label[$n$] = 0;
    goto accept_init          accepted[$n$] = true;
  :: $(1) \rightarrow$                :: $(1) \rightarrow$ label[$n$] = 1;
    goto T0_S2                accepted[$n$] = false; all_accepted = false;
  fi;                       fi; goto C$n$_checked;
                          C$n$_checked:
```

Figure 3.4: Automaton generated by LTL2BA and its post-processed representation

automata should visit their acceptance states infinitely often. *(iii)* If the execution path is finite and all the processes have finished execution, all the constraint automata should be in their *acceptance* states thus satisfying fairness conditions.

In order to encode these aspects we introduce several global variables. We use the valid_step variable to verify that the execution is correct and not blocked, and to check that system steps and steps of the constraint automata are interleaved. The next_accepted variable is used to check that all the constraints automata eventually visit acceptance states. The all_accepted variable is set to true if all the constraints automata visit acceptance states simultaneously.

The requirements on the valid execution paths are captured by the following formula, stating that constraints automata are not blocked, that they visit acceptance states infinitely often, and that if the value of variable next_accepted remains true forever (which happens only if the execution path is finite) then variable all_accepted will stay true forever:

$$\begin{aligned}&\mathsf{G}(\mathsf{valid\_step} \wedge \mathsf{F}\ \mathsf{next\_accepted} \wedge \\ &\quad (\mathsf{G}\ \mathsf{next\_accepted} \rightarrow \mathsf{G}\ \mathsf{all\_accepted})).\end{aligned}$$

The values of boolean variables valid_step, next_accepted and all_accepted are defined in part in process constraint_verifier (see Fig. 3.3) and in part in function system_step (see Fig. 3.5), which is executed during each visible step of every process in the system (see, e.g., Fig. 3.2). Variable valid_step is initially true and keeps this value if every system step is followed by exactly one step of process constraint verifier. This behavior is obtained

```
inline system_step() {
  if :: constraints_done → constraints_done = false;
     :: else valid_step = false;
  fi;
  next_accepted = false;
  ... /* Reset justcreated and justfulfilled flags */
  DoRequest[0].justcreated = false; DoRequest[0].justfulfilled = false;
}
```

Figure 3.5: The system_step routine

through auxiliary variable constraints_done which is set to true every time process constraint verifier evolves, and is set to false every time the system evolves. If a system step is done when constraints_done is already false, then two consecutive system steps are detected, and valid_step is set to false (see Fig. 3.5). Analogously, if a constraint verification step is done when constraints_done is already true, then two consecutive constraint verifier steps are detected, and the constraint_verifier process is finished (see Fig. 3.3). Variable next_accepted is set to true if variable accepted[next] associated to the next constraint to be executed is true. In this case next is updated so that all the constraints are considered in turn. In the system_step routine its value is again reset to false so this variable can remain true forever only on finite paths. Analogously, if all the constraint automata have visited their acceptance states simultaneously, the value of the variable all_accepted is true.

In order to check assertions and possibilities in Promela, the formula to verify has to be adapted to take into account valid executions. The formula is then model-checked by transforming it into a never claim. For instance, formula

$$\begin{aligned}&\mathsf{G}(\mathsf{valid\_step} \wedge \mathsf{F}\ \mathsf{next\_accepted} \wedge \\ &\quad (\mathsf{G}\ \mathsf{next\_accepted} \rightarrow \mathsf{G}\ \mathsf{all\_accepted})) \Rightarrow A.\end{aligned}$$

is generated for assertion $A$. Indeed, this formula checks whether all valid executions satisfy the assertion. A possibility property $P$ can be verified by model-checking the formula:

$$\begin{aligned}&\mathsf{G}(\mathsf{valid\_step} \wedge \mathsf{F}\ \mathsf{next\_accepted} \wedge \\ &\quad (\mathsf{G}\ \mathsf{next\_accepted} \rightarrow \mathsf{G}\ \mathsf{all\_accepted})) \Rightarrow \neg P.\end{aligned}$$

If a counter-example is found for this property, it is indeed a witness execution that show that possibility $P$ holds.

# Chapter 4

# Formal Analysis

The framework here proposed enables for several formal verification activities over the business requirements and business processes models. First, it is possible to verify the requirements model, e.g. by checking its consistency, or by verifying it against properties describing behavior that the model is supposed (or not supposed) to exhibit. This permits to discover inconsistencies and errors in the earliest phases, before having an actual implementation. As far as business processes are concerned, further kinds of analysis can be thought of. For instance, we can check for absence of deadlocks or livelocks in the devised protocol among the different parties involved in the business process. That is we can verify that the described process never blocks or it is never stuck in a loop. We can also verify business processes against business requirements and strategic goal models, thus providing an evidence that the given process actually implements and fulfills the requirements. Finally, we can verify whether the BPEL protocol is consistent with the published one. Moreover, in the inter-enterprise applications, not only separate business processes should be analyzed but also process compositions, for instance represented with BPEL code. The implementation of these verification activities is still ongoing work.

In this section we present the T-Tool, a verification tool able to deal with FT specification, and we show how the functionalities it provides can be used to tackle some of the verification problems we envisaged.

## 4.1 The T-Tool Verification Tool

The T-Tool [FLM$^+$04] supports the kinds of formal analysis described previously. Here we highlight some of its functionalities and we refer the reader to [FLM$^+$04] for additional details. The T-Tool is based on finite-state model checking [CGP99]. Model checking allows for an automatic verification of a specification with the generation of (counter-)example traces to witness the validity (or invalidity) of the specification. A limit of finite state model checking is that it requires a model with a finite number of states. Thus an upper bound on the number of class instances has to be specified in order to perform model checking.

The T-Tool input is an FT specification along with parameters that specify the upper bounds for the FT class instances. On the basis of this input, the T-Tool builds a finite model that represents all possible behaviors that satisfy the constraints of the specification. The T-Tool then verifies whether this model exhibits the desired behaviors. The T-Tool allows for different verification functionalities including interactive animation of the specification, automated consistency checks, and validation of the specification against possibility and assertion properties. Through animation, the user can generate valid scenarios for the specification by interacting with the T-Tool and incrementally extending a partial evolution of the model. Animation allows for a better understanding of the specified business domain, as well as for the identification of trivial bugs and missing requirements that are often taken for granted. Consistency checks are standard checks to guarantee that the FT specification is not self-contradictory. Inconsistent specifications occur quite often due to complex interactions among constraints in the specification, and they are very difficult to detect without the support of automated analysis tools. Consistency checks are performed automatically by the T-Tool and are independent of the application domain. Assertions properties describe conditions that should hold for all valid evolutions of the specification, while possibility properties describe conditions that should hold for at least one valid evolution. The verification phase usually generates feedback on errors in the FT specification and hints on how to fix them.

The T-Tool performs the verification of an FT specification in two steps (see Figure 4.1). In the first step, the FT specification is translated into an Intermediate Language (IL) specification. In the second step, the IL specification is given as input to the verification engine that is responsible of the actual verification. The T-Tool provides the user with two different verification engines built on top of state-of-the-art model checkers, that is the NuSMV [CCG+02] symbolic model checker and the SPIN [Hol03] explicit state model checker.

## 4.2 Verification of Business Requirements

Business requirements model enables for different kinds of verification to be carried out. In particular, the business requirements model can be automatically verified for consistency. Among the different kinds of verification the T-Tool provides are noticeable the verification of the possibility to create different goals, and to fulfill them thus showing that the specification is not over-specified and different goals do not conflict with each other.

Besides this, our framework allows business analyst to specify queries on the model in the form of properties that the requirements model is supposed to satisfy. We distinguish between **Assertion** properties, which describe conditions that should hold for all valid

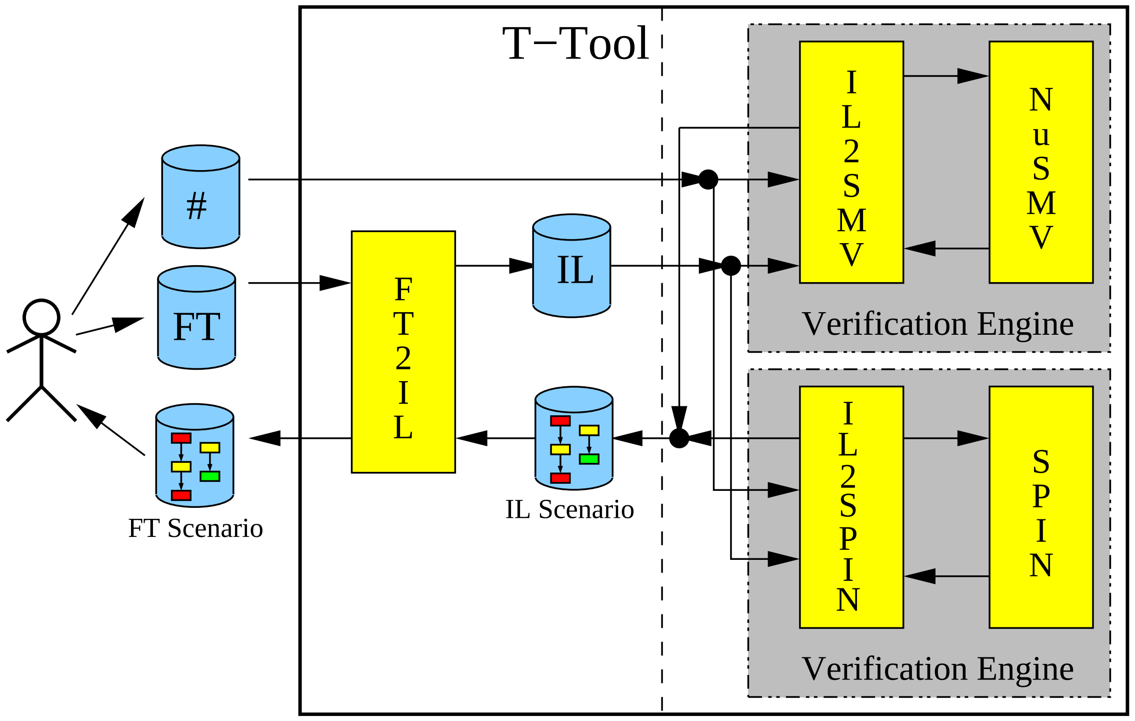

Figure 4.1: The T-Tool framework.

evolutions of the specification, and **Possibility** properties, which describe conditions that should hold for at least one valid evolution. Figure 2.5 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 allows 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 response to a request from the health-care agency has been received. Finally, assertion A2 requires that the task of doing a request is eventually fulfilled under the condition that there is a health-care agency that is bounded to provide assistance to the user and, the citizen sends the request to that particular health-care agency.

All the properties in Figure 2.5 are satisfied on the final version of the business 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 Receive-Assistance, and captured by the fulfillment constraints specified for this dependency as it can be seen in Figure 2.4.

It has to be remarked that, while verifying business requirements models, one cannot prove the business requirements specification is correct, since there is no reference model. However, the queries carried out on the business requirements specification can provide feedback on the captured behaviors and catch misunderstandings not trivial to be identified in an informal setting.

## 4.3 Verification of Business Processes

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. First, the given process can be checked for problems typical of a process specification, like the presence of deadlocks or livelocks. This is achieved by verification that the process specification will eventually complete on all its possible executions. For instance, in the case-study the citizen process can be blocked if it waits for the response from the health-care agency while the latter is not going to provide it for some reason (a deadlock). Or a health-care agency may request an additional information infinitely and the citizen process will stuck in this loop (a livelock). A further possibility consists of re-checking the formal queries defined for the requirements model (e.g., the properties in Figure 2.5) on the more detailed model. This is achieved by replacing a task of the FT specification (e.g task DoRequest) with the corresponding BPEL process and by checking again the queries. Another possibility is checking that the refined model satisfies the requirements described by the **Creation** 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 BPEL processes into the language of the verification engine chosen, i.e., into NuSMV or Spin finite state machines.

The verification applied to the BPEL process of Figure 2.6 pointed out some inconsistencies among the constraints of the requirements specification and the BPEL process definition. For instance, the fulfillment definition of the DoRequest task (see Figure 2.6, right part) requires that the value of the $\langle$`result`$\rangle$ variable in this task should be equivalent to the value of the corresponding variable in the WaitAnswer task. In the BPEL code (see Figure 2.6, left part) the value of this variable is copied directly from the Response message received. Thus, the corresponding variable of the WaitResponse task remains unchanged. This is shown in the counterexample represented in the Figure 4.2. The counterexample represents the message sequence chart of the execution of the model. The citizen creates the DoRequest task which generates the InitialRequest subtask where the request is sent to the health-care agency. In response, the latter creates tasks ReceiveRequest and EvaluateRequest. Then the subtask ProvideAnswer is created where the response message is created with positive result (result=1). The DoRequest process receives the message and created the subprocess WaitAnswer. The value of its variable result is initially false. Then the value of the variable result in the DoRequest process is assigned to value contained in the received message. Thus, the values of the variables result of the DoRequest task and WaitAnswer do not coincide. The constraint become true if we copy the content of the message to the variable result of the WaitAnswer task.

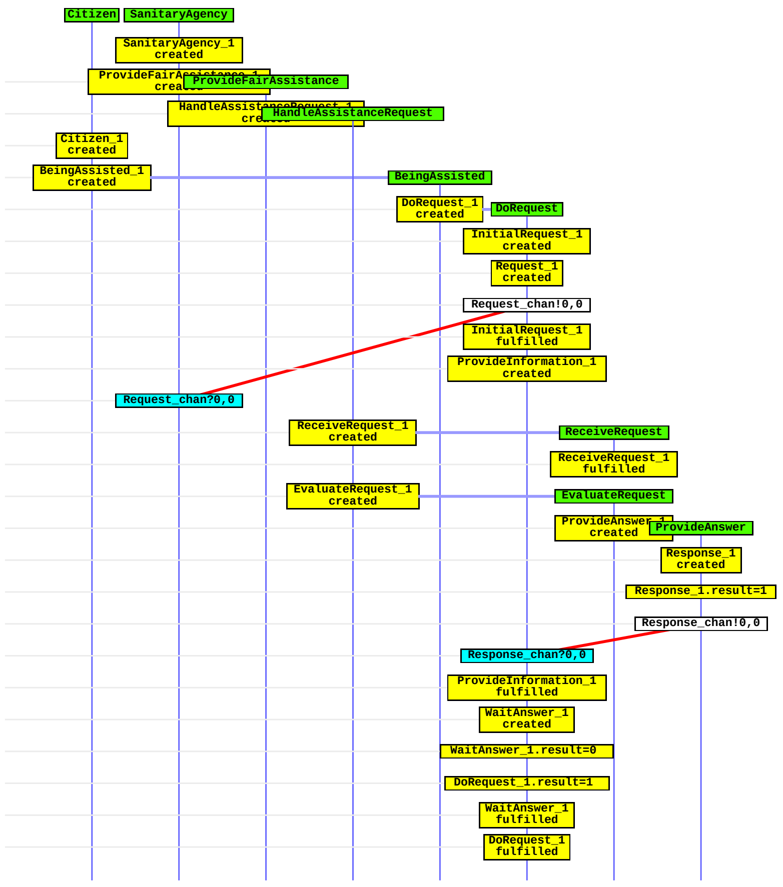

Figure 4.2: A counter-example generated by T-Tool.

As another example, if we modify the code of the process, e.g., by disallowing the reception of one of the two messages in the ⟨pick⟩ command, then the verification detects problems. If we disallow the reception of the InfoRequest message, the assertion A1 is violated. Indeed, if the health-care agency is requesting some information, the citizen is not able to answer to the request and the agency will not provide response to the citizen. Thus, it is impossible to fulfill the DoRequest. However, the ReceiveAssistance dependency still may be fulfilled, as it may be seen from the specification. In this case, the verification tool provides a counter-example similar to one represented in Figure 4.2. If we disallow the reception of the Response, even the possibility P1 becomes false. Indeed, if we do not receive the response, it is not possible to fulfill DoRequest.

The verification described so far deals mainly with the orchestration aspects of the Web service composition. However, other kinds of analysis may be applied in this framework. In particular, the choreography model of several participants may be verified against mutual expectations and requirements or even against global domain requirements on their composition.

Table 4.1: Logic specification translation

| | Direct Translation | | Compositional Translation | |
|---|---|---|---|---|
| | 1 instance | 1..2 instances | 1 instance | 1..2 instances |
| 1 constraint | 0,01sec | 0,01sec | 0,01sec | 0,01sec |
| 3 constraints | 0,03sec | 3,01sec | 0,03sec | 0,09sec |
| 5 constraints | 0,11sec | MO | 0,04sec | 0,14sec |
| 10 constraints | 10,65sec | SF | 0,04sec | 0,16sec |
| 30 constraints | MO | SF | 0,07sec | 0,20sec |
| 45 constraints | SF | SF | 0,15sec | 0,41sec |

## 4.4 Experimental Analysis

To experiment with the proposed approach, we applied the above verification tasks to the presented case study. We conducted three kinds of experiments. First, we evaluated the effectiveness of the compositional encoding of the logical specification w.r.t. the direct encoding based on one global LTL formula. Second, we tested the performance of SPIN on requirements verification tasks and compared it with NUSMV. Third, we tested the verification of requirements models against the processes implementing them. The experiments were executed on a bi-processor Pentium Xeon 2.4GHz, 2Gb of RAM, running Linux. All the tests have been executed within a time limit of 1 hour and memory limit of 2Gb. We mark the tests that failed to complete in the time limit as “TO”, and the test that exceed the memory limits as “MO”. In some cases, the never claim generation phase produced a segmentation fault. These cases are marked with “SF” in the tables.

### 4.4.1 Results of Logical Specification Translation

To compare the performance of the compositional logic specification translation we propose with the direct translation provided by SPIN, we considered a set of specifications of growing size. More precisely, the comparison has been performed on specifications with different numbers of constraints, and by allowing either up to 1 instance for each class (15 instances in total) or up to 2 instances for several classes (23 instances) [1]. Table 4.1 summarizes the results of the experiments with SPIN. It reports the time required for the translation of the FT specification into Promela. The results show that the compositional method outperforms the direct translation of the logic specification.

### 4.4.2 Results of Property Verification

To test the performance of SPIN and to compare it with NUSMV, we performed some verification experiments based on the assertions and possibilities of Fig. 2.5. To stress

[1]We recall that these upper bounds for the number of class instances are necessary to guarantee that the generated model is finite-state.

Table 4.2: Property verification results

| SPIN results | | | |
|---|---|---|---|
| | | **1 instance** | **1..2 instances** |
| **A1** | **HC4** | TO - 1284steps - 1382Mb | TO - 2857steps - 362Mb |
| | **BITSTATE** | Valid[(a)] - 21sec - 61Mb | TO - 3244steps - 1028Mb |
| | **3SPIN** | Valid[(b)] - 23sec - 69Mb | TO - 3207steps - 1178Mb |
| **A2** | **HC4** | TO - 1393steps - 1382Mb | TO - 2857steps - 362Mb |
| | **BITSTATE** | Invalid - 21sec - 52Mb | TO - 3244steps - 1058Mb |
| | **3SPIN** | Invalid - 24sec - 64Mb | TO - 3417steps - 1173Mb |
| **P1** | **HC4** | Valid - 27sec - 68Mb | TO - 2857steps - 362Mb |
| | **BITSTATE** | Valid - 14sec - 41Mb | TO - 3099steps - 956Mb |
| | **3SPIN** | Valid - 19sec - 56Mb | TO - 3312steps - 1143Mb |

Hash factors: [(a)] 1.97 – [(b)] 3.35

| NUSMV results | | | |
|---|---|---|---|
| | | **1 instance** | **1..2 instances** |
| **A1** | **BDD** | Valid - 9sec - 6,0Mb | TO - 235Mb |
| | **BMC** | Undec.[(*)] - 7sec - 20,4Mb | Undec.[(*)] - 106sec - 61,2Mb |
| **A2** | **BDD** | Invalid - 11sec - 6,9Mb | TO - 235Mb |
| | **BMC** | Invalid - 0,6sec - 3,8Mb | Invalid - 2sec - 11,3Mb |
| **P1** | **BDD** | Valid - 10sec - 5,8Mb | TO - 235Mb |
| | **BMC** | Valid[(**)] - 0,7sec - 5,3Mb | Valid[(**)] - 2sec - 16,0Mb |

[(*)] No counter-example found up to bound length 10
[(**)] Found example of length 4 satisfying P1

scalability, we have considered models of different sizes by allowing for different upper bounds to the number of instances for each FT class. We considered the case of 1 instance for each class and a intermediate “1..2” case, where we allow 2 instances for some classes. We compared different options of the SPIN model checker on the same problem. We also compared the results obtained with SPIN with those obtained using NUSMV. The Promela model for the FT specification contains 15 processes for the 1 instance case and 23 processes for the 1..2 instances case. With SPIN we experimented different verification options [Hol03], namely hash-compact verification (HC4), superstate verification (BITSTATE), and with the SPIN extension, namely “Triple SPIN” [DM04]. With NUSMV we experimented with the two model checking techniques provided by the tool, namely SAT-based bounded model checking [BCCZ99] (“BMC” in the tables), BDD-based model checking (“BDD”). We used a maximum length of 10 for the bounded model checking experiments[2]. The results of the verification are summarized in Table 4.2.

The verification provided the following results. Assertion 1 is true and assertion 2 is false. Possibility 1 is true and a corresponding witness trace is generated. Indeed, the dependency ReceiveAssistance is fulfilled whenever every assistance request accepted by the Health-care Agency is followed by receiving service. The dependency fulfilled also if there are no accepted requests. A counterexample demonstrates the fact that if the agency accepted the response and sent a message to the citizen the assertion is violated if the

[2]It worth noticing that the results provided by the BMC verification do not guarantee the validity of the formula, since counter-examples of length greater than the chosen maximum length (10 in our case) would not be detected. However, our experiments have shown that, in the model at hand, counter-examples usually have a length of 3 to 5. For this reason, a maximum length of 10 guarantees a high confidence in the result of the verification.

Table 4.3: Implementation verification results

| | | 1 instance | 1..2 instances |
|---|---|---|---|
| **A1** | **HC4** | TO - 516steps - 1442Mb | TO - 341steps - 1282Mb |
| | **BITSTATE** | Valid$^{(a)}$ - 32sec - 83Mb | Valid$^{(b)}$ - 169sec - 316Mb |
| | **3SPIN** | Valid$^{(c)}$ - 14sec - 35Mb | Valid$^{(d)}$ - 74sec - 171Mb |
| **A2** | **HC4** | Invalid - 125sec - 206Mb | TO - 341steps - 1162Mb |
| | **BITSTATE** | Invalid - 32sec - 71Mb | Invalid - 1285sec - 2003Mb |
| | **3SPIN** | Invalid - 15sec - 32Mb | MO - 673steps - 1141sec |
| **P1** | **HC4** | Valid - 2sec - 9,1Mb | TO - 341steps - 1282Mb |
| | **BITSTATE** | Valid - 3sec - 10,1Mb | Valid - 167sec - 306Mb |
| | **3SPIN** | Valid - 3sec - 12,0Mb | Valid - 59sec - 148Mb |
| **C** | **HC4** | Invalid - 2sec - 9,1Mb | TO - 341steps - 1282Mb |
| | **BITSTATE** | Invalid - 3sec - 11,4Mb | Invalid - 166sec - 306Mb |
| | **3SPIN** | Invalid - 3sec - 12,0Mb | Invalid - 62sec - 151Mb |

Hash factors: $^{(a)}$ 2.44 – $^{(b)}$ 1.66 – $^{(c)}$ 6.06 – $^{(d)}$ 1.61

message was not received by the citizen and the DoRequest task will never be fulfilled.

Comparing the results produced by SPIN and NUSMV, one can see that the BMC algorithms provided by NUSMV give overall the best results. In particular, this is the only technique that provides results for the 1..2 instances case. We remark that the translation of FT to the NUSMV language is highly engineered and optimized. Moreover, the NUSMV model checker has been extended to better handle models resulting from FT specifications. We expect to improve the performance of the SPIN back-end with similar optimization tasks.

### 4.4.3 Results of Implementation 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 Fig. 2.5 on the more detailed model (first three rows in Table 4.3). 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. The last row in Table 4.3 describes the results of the verification of the DoRequest task fulfillment definition.

# Chapter 5

# History of the Deliverable

The research work described in this deliverable had been carried out during first two years of the KLASE project. These activities were performed in several phases.

## 5.1 Initial Approach

The first step was to approach the problem of the modelling and analyzing business requirements for Web services and service-based processes, and how to manage their evolution. The solution proposed for these purposes was to extend the business process notations (in particular, BPEL) with the business requirements model [PRB04]. The resulting framework addresses the following issues:

- graphical definition of Web service requirements using Tropos modelling notations;
- formal representation of goals, requirements and constraints using Formal Tropos language;
- linking business process specification with requirements models using the Formal Tropos annotations;
- formal analysis of requirements and process models using NUSMV model checker.

## 5.2 Further Extensions

Several important weaknesses of the initial approach, both at the methodological and the technical level, led to some extensions investigated in the second year of this deliverable.

### 5.2.1 Extending Modelling Notations

While the Tropos language appears to be a good candidate for the modelling of strategic goal and requirements level, it lacks of notations suitable for the later phases of the design of distributed business processes. It does not provide a natural support for the process-like control flow concepts (e.g., sequential and parallel execution, conditional choice and loops), which are important for the definition and organization of task and activity models. Absence of these notations considerably complicates the design and specification of refined business models. The framework was extended as follows [KPR04b]:

- the business models of the service-based processes are given at three levels: strategic level, where the business goals and requirements are specified; activity level that defines the high-level process models and associated constraints; and message level, where the interactions are modelled and specified.
- the modelling notation is also extended with the ability to represent control-flow requirements between tasks in a process-oriented way.

### 5.2.2 Extending Analysis Capabilities

In order to improve the analysis capabilities we investigated the applicability of the alternative model checker tools, in particular SPIN model checker. This tool particularly aims at the analysis of distributed protocols and processes, thus being a natural candidate for the verification of Web service requirements models at later design phases, where the business processes are (partially) implemented. The resulting extensions are able to effectively address the following challenges:

- representation of the Formal Tropos requirements models in Promela-SPIN formats;
- representation of process models in Promela-SPIN formats;
- efficient algorithms for the analysis of interleaved requirements and processes models.

The methodological and practical results obtained on completion of this deliverable were further exploited in other research lines being held in the frames of the KLASE project, such as “Knowledge Level Methodology for Web Service Composition” (D1.3) or “Verification of Web Services” (D3.3).

# Bibliography

[ACD$^+$03] 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. *Business Process Execution Language for Web Services Version 1.1*, 2003.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proc. of the $5^{th}$ International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 1579 in Lecture Notes in Computer Science, pages 193–207, Amsterdam, The Netherlands, 1999. Springer.

[CCG$^+$02] 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. of Computer Aided Verification Conference*. Springer, 2002.

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CKM02] J. Castro, M. Kolp, and J. Mylopoulos. Towards requirements-driven information systems engineering: the Tropos project. *Information Systems*, 27(6):365–389, 2002.

[DM04] P. Dillinger and P. Manolios. Fast and Accurate Bitstate Verification for SPIN. In *Procs. of the $11^{th}$ Int. SPIN Workshop on Model Checking of Software*, 2004.

[FLM$^+$04] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in Tropos. *Requirements Engineering*, 9(2):132–150, 2004.

[FUMK03] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proc. of the 18$^t h$ International Conference on Automated Software Engineering (ASE'03)*, 2003.

[GO01] P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In *Procs. of Computer Aided Verification*, 2001.

[Hol03] G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.

[KHK$^+$03] J. Koehler, R. Hauser, S. Kapoor, F. Y. Wu, and S. Kumaran. A model-driven transformation method. In *Proc. of the Seventh International Enterprise Distributed Object Computing Conference (EDOC'03)*. IEEE Computer Society, 2003.

[KPR04a] Raman Kazhamiakin, Marco Pistore, and Marco Roveri. Formal Verification of Requirements using SPIN: A Case Study on Web Services. In *Proc. SEFM'04*, pages 406–415, 2004.

[KPR04b] Raman Kazhamiakin, Marco Pistore, and Marco Roveri. A Framework for Integrating Business Processes and Business Requirements. In *Proc. EDOC'04*, pages 9–20, 2004.

[Nak02] S. Nakajima. Model-checking verification for reliable web service. In *Proc. OOPSLA'02 Workshop on OOWS*, 2002.

[NM02] S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. WWW'02*, 2002.

[Pel03] Chris Peltz. Web Services Orchestration and Choreography. *Web Services Journal*, 2003.

[PRB04] M. Pistore, M. Roveri, and P. Busetta. Requirements-Driven Verification of Web Services. In *Proc. WS-FM'04*, *ENTCS*, 2004.

[tec03] The Formal Tropos language, 2003. Available from `http://dit.unitn.it/˜ft/doc/`.

[VCN$^+$01] A. Vasconcelos, A. Caetano, J. Neves, P. Sinogas, R. Mendes, and J. M. Tribolet. A framework for modeling strategy, business processes and information systems. In *5th International Enterprise Distributed Object Computing Conference (EDOC 2001)*. IEEE Computer Society, 2001.

[Yu97] E. Yu. Towards modeling and reasoning support for early requirements engineering. In *Proc. of the IEEE International Symposium on Requirement Engineering*, pages 226–235. IEEE Computer Society, 1997.