Formal Verification of Requirements using SPIN: A Case Study on Web Services*

Raman Kazhamiakin Marco Pistore
DIT, University of Trento
Via Sommarive 14, I-38050, Trento, Italy
{*raman,pistore*}*@dit.unitn.it*

Marco Roveri
ITC-irst
Via Sommarive 18, I-38050, Trento, Italy
roveri@irst.itc.it

Abstract

In this paper we describe a novel approach for the formal specification and verification of distributed processes in a Web service framework. The formal specification is provided at two different levels of abstraction. The strategic level *describes the requirements of the Web service domain, in terms of the different actors participating to it, with their goals and needs and with their mutual dependencies and expectations. The* process level *shows how these requirements are operationalized into communicating processes running on the different Web servers. We model the strategic level exploiting* Formal Tropos *(FT), a language for the formal definition of the requirements of agent-oriented systems which is based on Linear Time Logic. We model the process level using* Promela*, a language designed to describe communicating concurrent processes and amenable to formal verification. We exploit the* SPIN *model checker to perform* V&V *tasks. At the strategic level, requirements are validated against queries formulated by the designer, while at the process level the Promela specifications are verified against the requirements. The implementation of these V&V tasks requires the definition of a novel procedure to encode the FT requirements in Promela. The experiments described in the paper show that the approach is effective in revealing possible flaws both in the strategic and in the process models.*

1. Introduction

Requirements engineering is the very first step of a system development process. It concerns the precise identification of stakeholders' needs about the system-to-be, and aims to map them to specifications of the required software behaviors. Errors and ambiguities in requirements have been widely recognized as one of the major sources of problems in software development. Despite its criticality, requirements are often described only in natural language and requirements analysis techniques are rarely applied.

The *Tropos* project (http://www.troposproject.org/) aims to develop a requirements driven software engineering methodology for agent-oriented, distributed systems. Tropos provides graphical notations for requirements models, based on concepts like actors, goals, and dependencies among actors. Moreover, it adopts a variety of tools and techniques to support the analysis of the requirements models. In particular, Tropos provides a formal specification language, called *Formal Tropos* (hereafter FT). FT supplements the graphical notations with a first-order Linear Temporal Logic (LTL) suitable for automated verification. In FT, the graphical notations allow for the description of the "structural" aspects of the requirements model, for instance in terms of the network of relationships and dependencies among actors. The temporal logic permits to represent also the "dynamic" aspects of the model, describing for instance when an actor's goal can be considered fulfilled, or how the network of dependencies among actors evolves over time. This logical specification consists of a set of LTL constraints "anchored" to the different structural components of the model. In previous works [7] we have shown how to use the model checking techniques provided by NUSMV [2] to perform different kinds of formal analysis, such as consistency checking, animation of the specification, and property verification. In our experience, this formal analysis allows for a more precise understanding of the requirements model, and can reveal gaps and inconsistencies in the requirements that are difficult to discover in an informal setting.

A problem left open by the previous investigations is how to extend the formal analysis of a FT specification to the later phases of the software engineering process, such as architectural and detailed design and implementation. In this paper we do a step in this direction by considering a specific application domain, namely Web services. This domain is particularly interesting, since it allows for some simplifying assumptions. First, the architectural de-

*This work has been supported in part by the FIRB-MIUR project RBNE0195K5 "Astro".

sign phase is simplified by the fact that Web services already assume a specific (service-oriented) software architecture. Second, in many cases the detailed design of the Web services is limited to the definition of the processes implementing them. This can be done using process description languages like BPEL4WS [3], an emerging standard for specifying and executing distributed Web services.

In the paper we show how a FT requirements specification of a Web service system can be stepwise refined into a detailed design by extending it with process descriptions. We also show how model checking techniques can be exploited to validate the requirements model, as well as to verify the processes against these requirements. More precisely, we exploit the Promela [9] language for the definition of the Web service processes. 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 [9] model checker. Promela allows for a description of Web service processes which is similar to the one that can be written in languages like BPEL4WS.

In order to verify FT models enriched with Promela processes, we have extended the verification tool for FT described in [7] with a new back-end based on SPIN. While translating FT into Promela, we model each FT component as a separate process. Following the standard approach in SPIN, the LTL constraints on the "dynamics" of the model are translated into Promela code, which is then composed with the other processes in order to carry out the verification tasks. In the translation we need to face the problem of the size of the LTL formulas that need to be translated. Indeed, these formulas are generated as the conjunction of the constraints associated to the different components of the FT model. While each constraint is a small LTL formula, the composed formula is huge and the standard approaches for LTL verification provided by SPIN do not work on it. The solution that we have adopted is to translate separately the different constraints that appear in the FT specification and to join the generated pieces into a special process aimed to check constraint satisfaction at each system step.

In the paper we describe some experiments we have conducted on a small case-study in the Web service domain. The experiments test the effectiveness of the approach, and compare the results with those obtained using NUSMV. The results show that SPIN is able to carry out most of the considered verification tasks, both at the requirements level and at the process level. NUSMV has a better performance and can handle larger verification tasks, but its applicability is limited to the verification of the requirements model.

The paper is structured as follows. In the section 2 we briefly discuss some aspects of Web service technology that are relevant for understanding the approach proposed in the paper. Section 3 introduces the Formal Tropos methodology on a case study. Section 4 describes how we encode an FT specification into SPIN, while in Section 5 we report the results of the experiments. Finally, Section 6 discusses related work, draws conclusions and outlines future work.

2. Web Services

Web services are an emerging technology for building complex distributed systems focusing on interoperability, support for efficient integration of distributed processes, and uniform applications representation. Different services, provided by different organizations, perform basic activities that, combined in suitable ways, allow for the definition of complex business processes. Using Web service technology, companies can describe and publish their services, together with the information on how they can be invoked and composed. Moreover, Web services support the interactions among the different partners by providing a model of synchronous or asynchronous exchange of messages. These messages exchanges can be composed into longer 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. In an *orchestration* the composed process is considered from the perspective of one of the business parties, while the *choreography* describes the interactions from a global, neutral perspective, in terms of valid conversations or protocols among the different parties. Web services have developed different languages for orchestration and choreography (BPEL4WS, WSFL, WSCI...). Among them, BPEL4WS [3] is quickly emerging as the language of choice for the description of process interactions. BPEL4WS 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 valid interactions. Therefore, BPEL4WS permits to describe both the orchestration and the choreography of a business domain with an uniform set of concepts and notations.

While aiming to work together and to provide inter-enterprise Web service-based applications, companies do not want to disclose their internal processes and try to keep the ability to reorganize their processes in order to adapt to the strategy changes. In order to manage these changes the requirements should be tightly integrated with the Web service processes. Moreover, in order to provide reliable applications, the formal verification of conformance of the business process with the corresponding requirements model is of vital importance. The high level of abstraction introduced by Web services and the related technologies for service composition enable these integration and analysis in an implementation- independent way.

In the following sections we will show how the require-

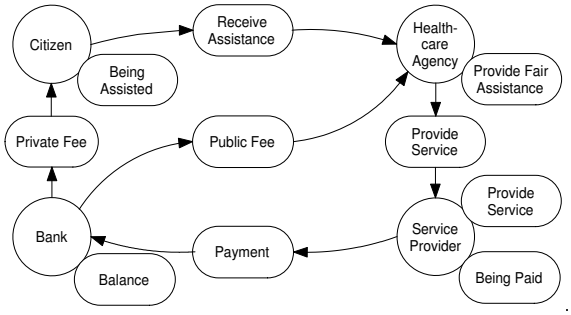


Figure 1. Tropos model of the case-study

ments models may be represented in the Tropos framework, how the process description may be integrated with the requirements model, and how formal analysis of these models is carried out.

3. Modeling Requirements and Processes

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 a senior citizen aims at being assisted, e.g., receiving services like transportation or meals at home. The assistance to citizens is managed by the Health-care Agency, that is run by the Local Government, and that aims at providing fair assistance to citizens. The Health-care Agency delegates to external service providers the actual delivery of the assistance 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 for the fee corresponding to the used service. The interaction among the different parties is required to happen via Web services and to be specified and implemented using BPEL4WS.

3.1. Specifying Requirements in Formal Tropos

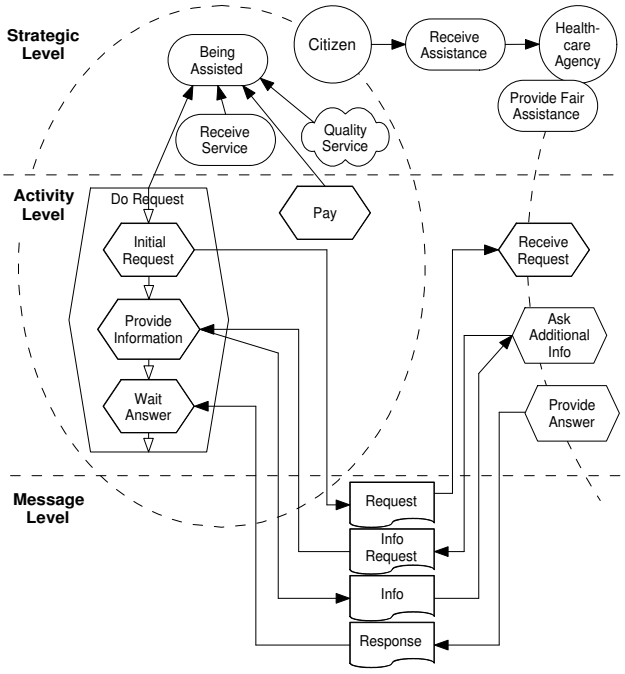
The **Tropos** modeling language provides graphical notations for modeling requirements [15]. It focuses on modeling and understanding the strategic aspects underlying the organizational framework within which the software system will eventually function. Thus it allows to better identify the motivations for the software system and the role that it will play inside the organizational framework. Figure 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 Citizen that aims at being assisted; HealthcareAgency that aims at providing a fair assistance to the citizens; ServiceProvider which goal is to provide a specific service; and 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 an intent/offer matching mechanism. This mechanism is represented in the diagram by means of *dependencies* (the ovals linked to two different actors). For instance, the Citizen depends on the Health-care Agency for being assisted, and this is formulated in the model with the ReceiveAssistance dependency.

Starting from this high-level view of the organizational or business system, Tropos proceeds with an incremental refinement. Goal analysis techniques are used to transforms the high level goals of one of the actors into sub-goals and eventually to operationalize them into tasks (see Fig. 2, left). In our case-study, the goal analysis is simple: the Citizen refines the goal BeingAssisted into the DoRequest and Pay tasks (hexagons), 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 delivered physically). Finally, soft-goals are used to describe non-functional conditions, with no clear-cut criteria as to when they are achieved (e.g., the citizen has some constraints on the quality of the delivered services).

The goal analysis phase is followed by a task refinement phase, where the high-level tasks are decomposed into subtasks. In Fig. 2, task DoRequest is further refined into InitialRequest, ProvideInformation, WaitAnswer. In this case, the three sub-tasks are composed sequentially, but other forms of task decomposition are also possible, e.g., parallel composition, choice, iteration... The task decomposition procedure ends once we have identified all basic tasks that define the 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 HealthcareAgency. This message is received and processed by the task ReceiveRequest of HealthcareAgency. The task AskAdditionalInfo is implemented 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.

These refinement steps are represented in Fig. 2 at 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 Fig. 2 represents the point of view of the Citizen. Only the interface of the HealthcareAgency is considered, and no description of the internal structuring of the HealthcareAgency is represented in the diagram, since this information is not available to the Citizen.

Formal Tropos 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 goal, ex-



Goal Dependency ReceiveAssistance **Mode maintain**
Depender Citizen **Dependee** HealthcareAgency
Fulfillment condition ∀ dr: DoRequest (
(dr.**actor** = **depender** ∧ **Fulfilled** (dr) ∧ dr.result) →
F ∃ rs: ReceiveService (rs.**actor** = **depender** ∧ **Fulfilled** (rs)))

Task DoRequest **Mode achieve**
Super BeingAssisted **Actor** Citizen
Attribute result : **boolean**
Fulfillment definition
∃ wa:WaitAnswer(wa.**super** = **self** ∧
Fulfilled (wa) ∧ (result ↔ wa.result))

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

Task ProvideInformation **Mode achieve**
Super DoRequest **Actor** Citizen
Fulfillment definition
G (∀ ir: InfoRequest(**Received** (ir) → ∃ i: Info(**Sent** (i)))

Task WaitAnswer **Mode achieve**
Super DoRequest **Actor** Citizen
Attribute result : **boolean**
Fulfillment definition
∃ r:Response(**Received** (r) ∧ (result ↔ r.result))

Figure 2. Formal Tropos specification

pectations, and dependencies 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 can be found in [7]. Here we present the most relevant aspects of the language based on the case-study. An excerpt of the FT annotations associated to the DoRequest task can be found in Fig. 2 (right).

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 Temporal logic (hereafter LTL). It has to be noticed that several instances of each element may exist during the evolution of the system. For instance, different DoRequest tasks may exists for different Citizen instances, or for different types of medical problems of the same Citizen.

Each object has an associated list of attributes. Each attribute has a *sort* (i.e., its type) which can be either primitive (boolean, integer. . .) or an object. For instance, attribute result of task DoRequest determines whether the response from the HealthcareAgency was successful or not. Attributes of primitive sort usually define the relevant state of an instance. Goals and tasks are associated to the corresponding actor with the special attribute **Actor** (e.g., task DoRequest). Similarly, **Depender** and **Dependee** attributes of dependencies represent the two parties involved in a delegation relationship. The **Super** attributes of goal/task elements represent a decomposition of the primary elements into sub-elements. In particular, InitialRequest, ProvideInformation and ReceiveResponse are subtasks of the DoRequest task, which is reflected in their **Super** attribute.

In FT the focus is on the conditions for the *fulfillment* of goals, tasks, and dependencies. These are characterized by a **Mode**, which declares the modality of their fulfillment. Examples of 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). The inner layer of an FT class declaration consists of constraints that describe the dynamic aspects of the domain elements. In FT we distinguish among **Invariant**, **Creation**, and **Fulfillment** constraints. **Invariant** constraints define conditions that should hold throughout the lifetime of all class instances. **Creation** and **Fulfillment** constraints define conditions on the two critical moments in the lifetime of class instances, namely their *creation* and *fulfillment*. **Creation** constraints can be associated with any class. Such constraints should be satisfied whenever an instance of the 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 can be associated only with goals, tasks, and dependencies. These constraints should hold whenever a goal is

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 3. Formal Tropos Properties

achieved, or a task is completed.

Constraints in FT are described with formulas in LTL. Besides the standard boolean and relational operators, the logic provides the quantifiers ∀ and ∃, which range over all the instances of a given class, and a set of *temporal operators*. In the examples in this paper we use operators **F**, which defines a condition that has to hold eventually in the future, and **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 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. 3 some properties for the case-study are reported.

For lack of space we refer to [6] 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. In [7] we have shown how to exploit NuSMV for this verification.

3.2. Process Representation in Formal Tropos

Once the requirements of a Web service domain have been specified and analyzed, we need to extend these requirements with the definition of the processes implementing them. Here we show how the FT requirements can be furthermore refined into executable code by means of

```
bool waitResponse;
atomic{
  CREATE ri: InitialRequest;
  ri.super = self;
  waitResponse = true};
atomic{
  CREATEMESSAGE vRequest: Request;
  Request_channel ! vRequest};
atomic{
  FULFILL ir: InitialRequest [ir.super == self];
  CREATE pi: ProvideInformation; pi.super = self};
do::atomic{ waitResponse ->
    if::InfoRequest_channel ? vInfoRequest;
        CREATEMESSAGE vInfo : Info;
        vInfo.reference = vInfoRequest;
        Info_channel ! vInfo;
      ::Response_channel ? vResponse;
        FULFILL pi: ProvideInformation [pi.super==self];
        CREATE wa: WaitAnswer; wa.super = self;
        waitResponse = false;
        self.result = vResponse.result;
    fi};
  ::else break;
od;
atomic{
  FULFILL wait: WaitAnswer [wait.super == self];
  FULFILL self};
```

Figure 4. DoRequest process specification

the Promela [9] language. In particular, Promela processes are used to describe the behavior of the services that will constitute the distributed application. We remark that in the Web service framework these processes would have been written in languages like BPEL4WS. Promela provides communication and control constructs for defining these processes that are very similar to the ones provided by BPEL4WS, therefore it is a very natural candidate for mapping BPEL4WS into a language suitable for formal verification. At the time of writing we are developing an automated translator from BPEL4WS specifications into Promela.

In order to link the operational model with the requirements model we have extended Promela with a set of macro commands. In particular, the CREATE and CREATEMESSAGE macros specify an instance creation event of a subtask or a message respectively. The FULFILL macro specifies a successful task completion event with optional guarding expression in square brackets.

Figure 4 represents the process specification of the task DoRequest. The process behaves as follows. First, initialization steps are performed. Within these steps the variable waitResponse is set to true, an instance of the InitialRequest task is created, the message Request is prepared and sent, the InitialRequest is completed and the ProvideInformation task is started. Hereafter, a loop is entered and its body is repeated until variable waitResponse becomes false. The body of the loop consists of a if 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. 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

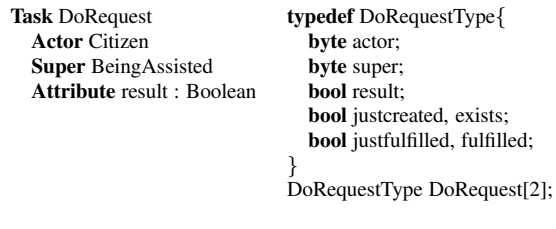


Figure 5. DoRequest task representation in FT and Promela

exit from the loop. Immediately after the loop, two fulfillment events are generated specifying that the WaitAnswer is completed as well as the DoRequest task itself. A process model specified in this way is automatically translated into pure Promela code. In particular, all the macro commands are replaced with the corresponding Promela code, where the necessary data modification is performed.

4. Encoding Formal Tropos into Promela

In this section we show how an FT specification can be translated into Promela. The translation manages separately the outer layer and the inner layer of the FT specification. The outer layer, consisting of FT classes and attributes, is mapped into Promela processes and data structures. The inner layer, consisting of the temporal constraints, assertions, and possibilities, could in theory be represented as a unique LTL specification and mapped into a "never claim". However, this approach does not work due to the huge size of the composed LTL specification. Therefore, we propose an alternative encoding that is compositional on the constraints.

4.1. Formal Tropos Class Representation

Figure 5 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.

```
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 6. DoRequest task process definition

The life-cycle of the class instances is encoded using Promela processes. Figure 6 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 de-

```
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 7. The constraint_verifier process

```
                                  if
                                  :: label[n]==0 → goto Cn_accept_init
                                  :: label[n]==1 → goto Cn_T0_S2
                                  fi;
/* G(p → Fq) */                  /* G(p → Fq) */
accept_init:                      Cn_accept_init:
  if                                if
  :: (¬p)||q →                      :: (¬p)||q → label[n] = 0;
     goto accept_init                  accepted[n] = true;
  :: (1) →                          :: (1) → label[n] = 1;
     goto T0_S2                        accepted[n] = false; all_accepted = false;
  fi;                               fi; goto Cn_checked;
T0_S2:                            Cn_T0_S2:
  if                                if
  :: q →                            :: q → label[n] = 0;
     goto accept_init                  accepted[n] = true;
  :: (1) →                          :: (1) → label[n] = 1;
     goto T0_S2                        accepted[n] = false; all_accepted = false;
  fi;                               fi; goto Cn_checked;
                                  Cn_checked:
```

Figure 8. Automaton generated by LTL2BA and its post-processed representation

pendencies 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. 6). 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.

4.2. 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 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 [8]. 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. 7). Some modifications on the bodies are necessary before they can be joined. For instance, Fig. 8 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

```
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 9. The system_step routine

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 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:

$$\mathsf{G}\big(\mathsf{valid\_step} \wedge \mathsf{F\ next\_accepted} \wedge (\mathsf{G\ next\_accepted} \rightarrow \mathsf{G\ all\_accepted})\big).$$

The values of boolean variables valid_step, next_accepted and all_accepted are defined in part in process constraint_verifier (see Fig. 7) and in part in function system_step (see Fig. 9), which is executed during each visible step of every process in the system (see, e.g., Fig. 6). 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 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. 9). 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. 7). 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

$$\mathsf{G}\big(\mathsf{valid\_step} \wedge \mathsf{F\ next\_accepted} \wedge (\mathsf{G\ next\_accepted} \rightarrow \mathsf{G\ all\_accepted})\big) \Rightarrow A.$$

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:

$$\mathsf{G}\big(\mathsf{valid\_step} \wedge \mathsf{F\ next\_accepted} \wedge (\mathsf{G\ next\_accepted} \rightarrow \mathsf{G\ all\_accepted})\big) \Rightarrow \neg P.$$

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

5. Experimental Analysis

To experiment with the proposed approach, we have implemented the FT to Promela translation described in the previous section (the tools used in the experiments can be found at http://dit.unitn.it/ ft/ws/). The verification of an FT specification can now be done choosing between two backends, the old one which exploits NuSMV and the new one using Spin.

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.

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

Table 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

class (15 instances in total) or up to 2 instances for several classes (23 instances)[1]. Table 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.

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. 3. To stress 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 [9], namely hash-compact verification (HC4), superstate verification (BITSTATE), and with the SPIN extension, namely "Triple SPIN" [4]. With NUSMV we experimented with the two model checking techniques provided by the tool, namely SAT-based bounded model checking [1] ("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 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 message was not received by the citizen and the DoRequest task will never be fulfilled.

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

[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 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

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.

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. 3 on the more detailed model (first three rows in Table 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 3 describes the results of the verification of the DoRequest task fulfillment definition. This constraint is violated for the following reasons. The fulfillment definition requires that the value of the result variable in this task should be equivalent to the value of the corresponding variable in the WaitAnswer task. In the process implementation code (see Fig. 4) the value of this variable is copied directly from the Response message received. Thus, the corresponding variable of the WaitResponse task remains unchanged.

Table 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

6. Related Work and Conclusions

In earlier work [7] we have proposed a framework for the specification and verification of early requirements based on symbolic model checking and NuSMV. In this paper we propose an alternative verification approach, based on explicit state model checking and SPIN. Moreover, we have extend the scope of the verification to include the design of distributed processes defined in Promela. The approach is based on a novel, compositional encoding of the LTL constraints that define the valid behaviors of the requirements model in the verification tasks. The experiments show that the approach is viable, even if the performance is currently a rather serious limit for its applicability.

In the paper, we have used a case study in the field of Web services for illustrating the framework and for experimenting the verification techniques. We refer to [14, 11] for a broader discussion on the relationships between the proposed framework and Web service technologies.

Formal analysis is often used to verify correctness of specifications, but, it is usually applied in later phases. The works that are most relevant to ours in the sense that they also focus on the verification of requirements models are Alcoa/Alloy [10], KAOS [12], and the work on "Topoi Diagrams" [13]. However, as far as we know, these formalisms have not been applied for Web Services. For a complete analysis of these works w.r.t. FT we refer the reader to [7].

There are several works on the verification of Web services. The closest to our approach is [5] that provides a framework for the verification of LTL properties on the protocols specified in BPEL4WS by translating them into Promela. Differently to our approach this work concentrates on the translation and verification of communications specified by the protocol, while in our approach we address also the problem of verifying models where protocols are interleaved with requirements.

There are several directions for further research. We are investigating optimization of the model generator by integrating advanced abstraction techniques that exploit, for instance, possible symmetries in the specification. This could lead to a better exploitation of the partial order reduction capabilities provided by SPIN. We are also interested in a deeper investigation of the compositional approach for the verification of complex LTL properties that we have outlined in this paper. Finally, we are interested in linking our approach to the Web service technologies, allowing for instance to specify the processes using industrial standards like the BPEL4WS language instead of SPIN.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Procs. of the* 5^{th} *Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 1999.

[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 *Procs. of Computer Aided Verification Conference*, 2002.

[3] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. *Business Process Execution Language For Web Services*. BEA Systems & IBM Copora-tion & Microsoft Corporation, 2002.

[4] 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.

[5] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of the Thirteenth International World Wide Web Conference (WWW'04)*, 2004. To appear.

[6] A. Fuxman, R. Kazhamiakin, M. Pistore, and M. Roveri. The Formal Tropos Language, 2003. Available from `http://dit.unitn.it/~ft/doc/`.

[7] 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.

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

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

[10] D. Jackson. Alloy: a Lightweight Object Modeling Notation. *ACM Transaction on Software Engineering Methodology*, 11(2):256–290, 2002.

[11] R. Kazhamiakin, M. Pistore, and M. Roveri. A Framework for Integrating Business Processes and Business Requirements. In *Proc. 8th Int. IEEE Enterprise Distributed Object Computing Conference (EDOC'04)*, 2004. To appear.

[12] E. Leiter. *Reasoning about Agents in Goal-oriented Requirements Engineering*. PhD thesis, Universite Catholique de Louvain, 2001.

[13] T. Menzies, J. Powell, and M. E. Houle. Fast Formal Analysis of Requirements via "Topoi Diagrams". In *Procs. of the* 23^{rd} *Int. Conference on Software Engineering*, 2001.

[14] M. Pistore, M. Roveri, and P. Busetta. Requirements-Driven Verification of Web Services. In *Procs. of 1st Int. Workshop on Web Services and Formal Methods (WSFM'04)*, 2004.

[15] E. Yu. Towards Modeling and Reasoning Support for Early Requirements Engineering. In *Procs. of the IEEE Int. Symposium on Requirement Engineering*, 1997.