Requirements Engineering manuscript No.
(will be inserted by the editor)

Specifying and Analyzing Early Requirements in Tropos

Ariel Fuxman[1], Lin Liu[1], John Mylopoulos[1,2], Marco Pistore[2,3], Marco Roveri[3], Paolo Traverso[3]

[1] *Department of Computer Science, University of Toronto, 40 St. George St. M5S 2E4, Toronto, Canada*
[2] *Department of Information and Communication Technology, University of Trento, Via Sommarive 14, I-38050, Trento, Italy*
[3] *ITC-irst, Via Sommarive 18, I-38050, Trento, Italy*
{*afuxman,liu,jm*}*@cs.toronto.edu* *pistore@dit.unitn.it* {*roveri,traverso*}*@irst.itc.it*

Received: / Revised version:

Abstract. We present a framework that supports the formal verification of early requirements specifications. The framework is based on *Formal Tropos*, a specification language that adopts primitive concepts for modeling early requirements (such as actor, goal, and strategic dependency), along with a rich temporal specification language. We show how existing formal analysis techniques, and in particular *model checking*, can be adapted for the automatic verification of Formal Tropos specifications. These techniques have been implemented in a tool, called the *T-Tool*, that maps Formal Tropos specifications into a language that can be handled by the NUSMV model checker. Finally, we evaluate our methodology on a course-exam management case study. Our experiments show that formal analysis reveals gaps and inconsistencies in early requirements specifications that are by no means trivial to discover without the help of formal analysis tools.

Keywords: Early Requirements Specifications, Formal Methods, Model Checking

1 Introduction

Early requirements engineering is the phase of the software development process that models and analyzes the operational environment where a software system will eventually function [23]. In order to analyze such environment, it is necessary to understand the objectives, business processes and interdependencies of different stakeholders. Although errors and misunderstandings at this stage are both frequent and costly, early requirements engineering is usually done informally (if at all). In this work, we present a formal framework that adapts results from the Requirements Engineering and Formal Methods communities to facilitate the precise modeling and analysis of early requirements.

Formal methods have been successfully applied to the verification and certification of software systems. In several industrial fields, formal methods are becoming integral components of standards [5]. However, the application of formal methods to early requirements is by no means trivial. Most formal techniques have been designed to work (and have been mainly applied) in later phases of software development, e.g., at the architectural and design level. As a result, there is a mismatch between the concepts used for early requirements specifications (such as goal and actor) and the constructs of formal specification languages such as Z [22], SCR [13], TRIO [12,20].

Our framework supports the automatic verification of early requirements specified in a formal modeling language. This framework is part of a wider on-going project called *Tropos*, whose aim is to develop an agent-oriented software engineering methodology, starting from early requirements. The methodology is to be supported by a variety of analysis tools based on formal methods. In this paper, we focus on the application of model checking techniques to early requirements specifications.

To allow for formal analysis, we introduce a formal specification language called *Formal Tropos* (hereafter FT). The language offers all the primitive concepts of *i** [23] (such as actors, goals, and dependencies among actors), but supplements them with a rich temporal specification language inspired by KAOS [17].

We have extended an existing formal verification technique, model checking [9], to support the automated analysis of FT specifications. We have also implemented this extension in a tool, called the T-Tool, which is based on the state-of-the-art symbolic model checker NUSMV [8]. The T-Tool translates automatically an FT specification into an Intermediate Language (hereafter IL) specification that could potentially link FT with different verification engines. The IL representation is then automatically translated into NUSMV,

Correspondence and offprint requests to: Marco Roveri, ITC-irst, Via Sommarive 18, I-38050 Trento – Italy. E-mail: `roveri@irst.itc.it`

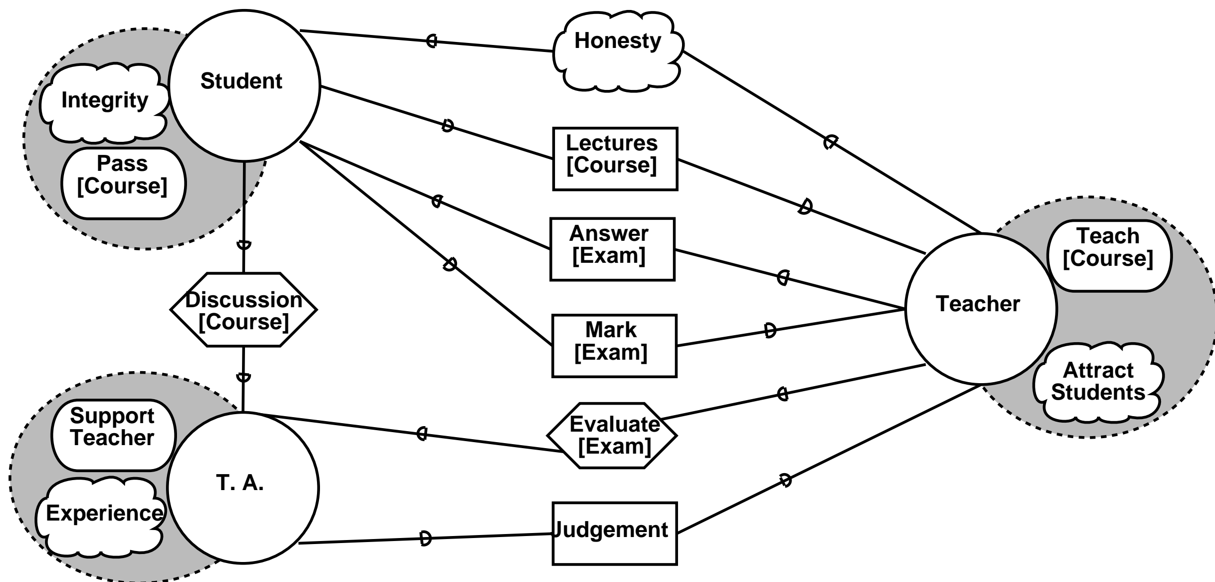


Fig. 1. High-level *i** model of the course-exam management case study.

which can then perform different kinds of formal analysis, such as consistency checking, animation of the specification, and property verification.

On the methodological side, we define some heuristic techniques for rewriting an *i** diagram into a corresponding FT specification. We also offer guidelines on how to use the T-Tool effectively for formal analysis, e.g., by suggesting what model checking technique to use when a particular formal property is to be validated. Finally, we report the results of a series of experiments that we conducted in order to evaluate the scope and scalability of the approach.

The paper is structured as follows. Section 2 introduces a case study and shows how to build an FT specification from an *i** model. Section 3 uses the case study to illustrate how FT can be used for the incremental refinement of a specification. Section 4 describes the T-Tool, focusing on its functionalities, architecture, and usage guidelines. Section 5 presents the experiments we carried out. Section 6 discusses related work, draws conclusions, and outlines future work.

2 From *i** to Formal Tropos – A case study

In this section we use a course-exam management case study to describe how an FT specification can be obtained from an *i** model.

*2.1 Strategic modeling with i**

The *i** modeling language [23] has been designed for the description of early requirements. It is founded on the premise that during this phase it is important to understand and model the social setting within which the system-to-be will eventually function. We use the notation of *i** as a starting point for our methodology, since it provides an informal graphical description of the organizational setting, which is later described in a formal language that is more suitable for automatic analysis.

The *i** framework offers three categories of concepts, drawn from goal- and agent-oriented languages: actors, intentional elements, and intentional links. An *actor* is an active entity that carries out actions to achieve its goals. Figure 1 depicts a high-level *i** diagram for the course exam management case study, with its three main actors: **Student**, **Teacher**, and teaching assistant (**T.A.**).

Intentional elements in *i** include *goals*, *softgoals*, *tasks*, and *resources*, and can either be internal to an actor, or define dependency relationships between actors. A *goal* (rounded rectangle) is a condition or state of affairs in the world that the actor would like to achieve. For example, a student's objective to pass a course is modeled as goal **Pass[Course]** in Figure 1. A *softgoal* (irregular curvilinear shape) is typically a non-functional condition, with no clear-cut criteria as to when it is achieved. For instance, the fact that a teacher expects the students to be honest is modeled with the softgoal **Honesty**. A *task* (hexagon) specifies a particular course of action that produces a desired effect. In our example, the element **Evaluate[Exam]** is represented as a task. A *resource* (rectangle) is a physical or information entity. For instance, the student waits for the lectures of the course (**Lectures[Course]**) and for a mark for an exam (**Mark[Exam]**), while the teacher waits for an answer to an exam (**Answer[Exam]**). In Figure 1 a boundary delimits intentional elements that are internal to each actor. Intentional elements outside the boundaries correspond to goals, softgoals, tasks, and resources whose responsibility is delegated from one actor to another. For instance, the student depends on the teacher for the marking of the exams, so the resource **Mark[Exam]** is modeled as a

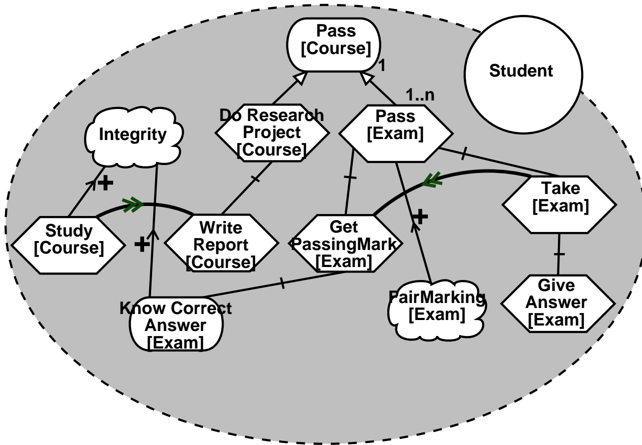


Fig. 2. High-level *i** model focusing on the **Student**.

dependency from the student to the teacher. In the diagrams, *dependency* links (—D—) are used to represent these inter-actor relationships.

Figure 2 zooms into one of the actors of this organizational setting, the student. The figure shows how the high-level intentional elements of the student are refined and operationalized. In *i**, these refinements and relationships among intentional elements are represented with intentional links, which include *means-ends*, *decomposition*, and *contribution* links. Each element connected to a goal by a *means-ends* link (—▷) is an alternative way to achieve the goal. For instance, in order to pass a course (**Pass[Course]**), a student can pass all the exams of the course (**Pass[Exam]**), or can do a research project for the course (**DoResearchProject[Course]**). *Decomposition* links (—+) define a refinement for a task. For instance, if a student wants to pass an exam (**Pass[Exam]**), she needs to attend the exams (**Take[Exam]**), and get a passing mark (**GetPassingMark[Exam]**). A *contribution* link (—>—) describes the impact that an element has on another. This can be negative (-) or positive (+) (e.g., **FairMarking[Exam]** has a positive impact on **Pass[Exam]**).

In Figure 2 we also use some elements that are not present in the original *i** definition, but turn out to be useful in subsequent phases of our methodology. These extensions have been described in [11]. *Prior-to* links (—↠) describe the temporal order of intentional elements. For example, a student can only write a report after studying for the course, and can only get a passing mark after she actually takes the exam. We also use *cardinality constraints* (the numbers labeling the links) to define the number of instances of a certain element that can exist in the system. For instance, for each **Pass[Course]** goal there must be at least one **Pass[Exam]** subgoal. Links without a number suggest one-to-one connections.

Figure 3 extends Figure 2 with the inner description of the teacher and of the dependencies existing between the teacher and the student. In the paper, we will concentrate on the subset of the course-exam management case study that is described in the *i** diagram in Figure 3 and we will not consider the teaching assistant.

2.2 *Formal Tropos specifications*

FT has been designed to supplement *i** models with a precise description of their dynamic aspects. In FT, we focus not only on the intentional elements themselves, but also on the circumstances in which they 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. With an FT specification, one can ask questions such as: Can we construct valid operational scenarios based on the model? Is it possible to fulfill the goals of the actors? Do the decomposition links and the prior-to constraints induce a meaningful temporal order? Do the dependencies represent a valid synergy or synchronization between actors?

The grammar of the FT language is given in Figure 4. 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 it defines the structure of the instances together with their attributes. The inner layer expresses constraints on the lifetime of the objects, given in a typed first-order linear-time temporal logic. An FT specification is completed by a set of global properties that express properties on the domain as a whole.

2.2.1 The outer layer

Figure 5 is an excerpt of the outer layer of the FT specification of the course-exam management case study. In the transformation of an *i** diagram into an FT specification, actors and intentional elements are mapped into corresponding "classes" in the outer layer of FT. Moreover, "entities" (e.g., **Course** and **Exam**) are added to represent the non-intentional elements of the domain.

Many instances of a class may exist during the evolution of the system. For example, different **Pass[Course]** dependencies may exist for different **Student** instances, or for different courses taken by the same student. Each class has an associated list of attributes. Each attribute has a *sort* (i.e., its type) and one or more optional *facets*. Sorts can be either primitive (Boolean, integer. . .) or classes. Attributes of primitive sorts usually define the relevant state of an instance. For example, Boolean attribute **passed** of resource dependency **Mark** determines whether the mark is passing or not. Attributes of non-primitive sorts define references to other instances in the domain. For example, attribute **exam** of goal **Pass[Exam]** is a reference to the specific exam to be passed, and attribute **pass_course** is a reference to a **Pass[Course]** instance that motivates the student to pass the exam. Similarly, dependency **Mark** refers to the exam that has to be marked (attribute **exam**) and to **GetPassingMark[Exam]** goal of the student that motivates the expectation of having a mark (attribute **gpm**).

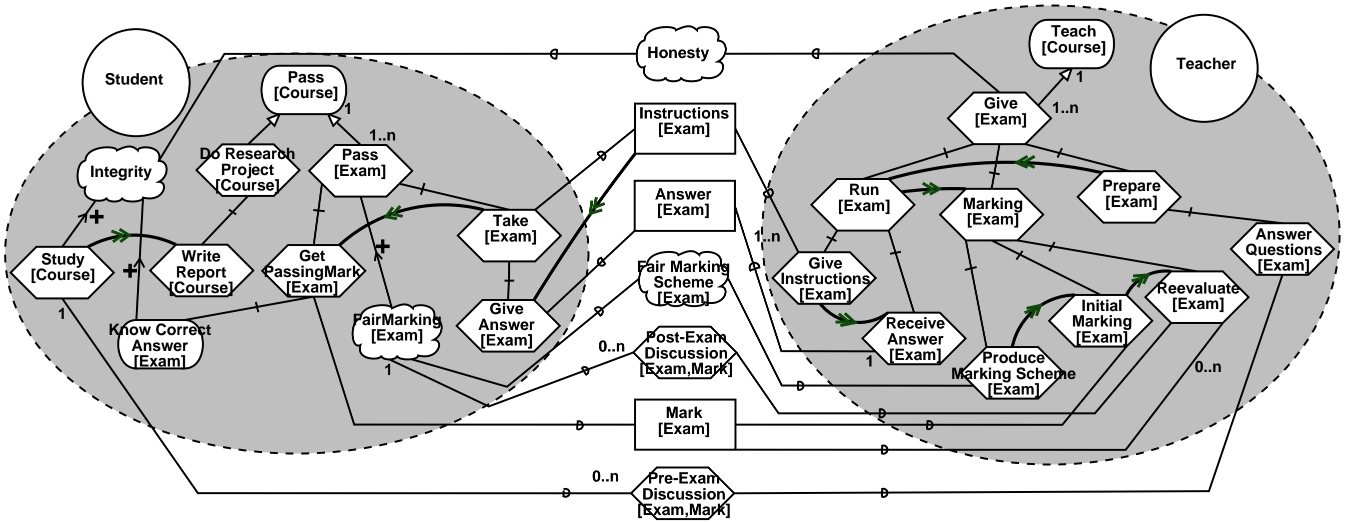


Fig. 3. Annotated *i** model of the course-exam management case study.

/ The outer layer */*

specification := (*entity* | *actor* | *int-element* | *dependency* | *global-properties*)*

entity := Entity *name* [*attributes*] [*creation-properties*] [*invar-properties*]

actor := Actor *name* [*attributes*] [*creation-properties*] [*invar-properties*]

int-element := *type name mode* Actor *name* [*attributes*] [*creation-properties*] [*invar-properties*] [*fulfill-properties*]

dependency := *type* Dependency *name mode* Depender *name* Dependee *name* [*attributes*] [*creation-properties*] [*invar-properties*] [*fulfill-properties*]

type := (Goal | Softgoal | Task | Resource)

mode := Mode (achieve | maintain | achieve&maintain | avoid)

/ Attributes */*

attributes := Attribute *attribute*$^+$

attribute := *facets name* : *sort*

facets := [constant] [optional] . . .

sort := *name* | integer | boolean | . . .

/ The inner layer */*

creation-properties := Creation *creation-property*$^+$

creation-property := *property-category event-category temporal-formula*

invar-properties := Invariant *invar-property*$^+$

invar-property := *property-category temporal-formula*

fulfill-properties := Fulfillment *fulfill-property*$^+$

fulfill-property := *property-category event-category temporal-formula*

property-category := [constraint | assertion | possibility]

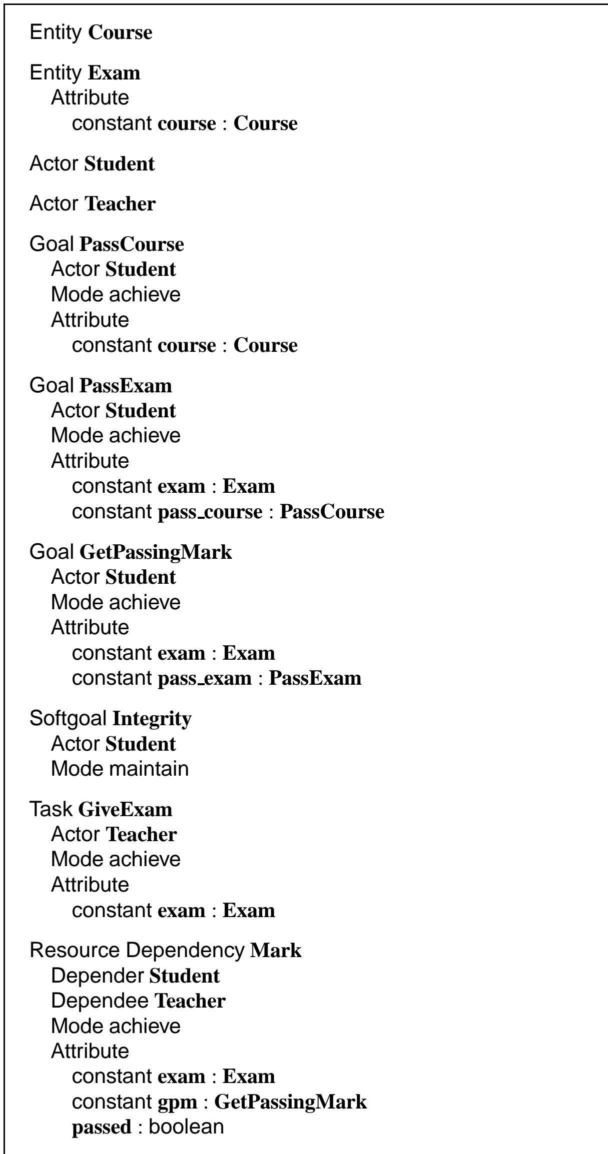
event-category := trigger | condition | definition

/ Global properties */*

global-properties := Global *global-property*$^+$

global-property := *property-category temporal-formula*

Fig. 4. The Formal Tropos grammar.

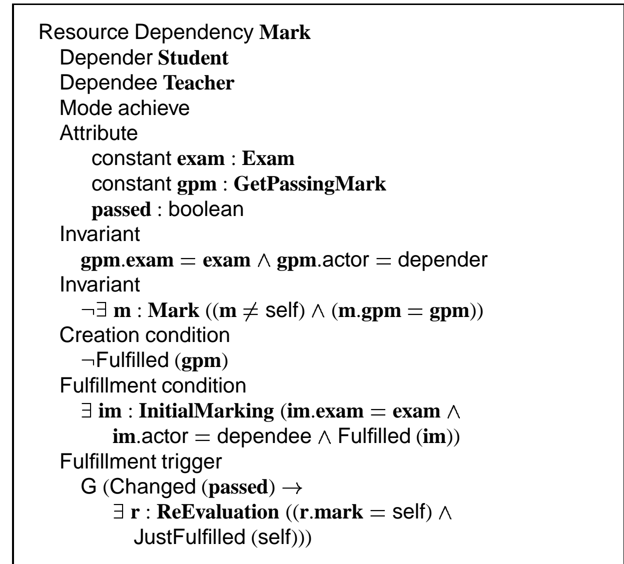


```
Entity Course

Entity Exam
  Attribute
    constant course : Course

Actor Student

Actor Teacher

Goal PassCourse
  Actor Student
  Mode achieve
  Attribute
    constant course : Course

Goal PassExam
  Actor Student
  Mode achieve
  Attribute
    constant exam : Exam
    constant pass_course : PassCourse

Goal GetPassingMark
  Actor Student
  Mode achieve
  Attribute
    constant exam : Exam
    constant pass_exam : PassExam

Softgoal Integrity
  Actor Student
  Mode maintain

Task GiveExam
  Actor Teacher
  Mode achieve
  Attribute
    constant exam : Exam

Resource Dependency Mark
  Depender Student
  Dependee Teacher
  Mode achieve
  Attribute
    constant exam : Exam
    constant gpm : GetPassingMark
    passed : boolean
```

Fig. 5. Excerpt of outer layer of an FT class declaration.

Facets represent basic properties of attributes. The facet optional means that the attribute may be undefined. The facet constant means that the value of the attribute does not change after its initialization. In most cases attributes that refer to other classes are constant, i.e., their values do not change over time, while the values of user-defined attributes usually change during the lifetime of class instances. In the case of attribute **passed** of dependency **Mark**, for instance, a change of value is used to model a change of mark due to a re-evaluation of the exam.

Special attributes are present in the declarations of the intentional elements. Internal intentional elements are associated to the corresponding actor with the special attribute Actor (for instance, **Pass[Exam]** has a **Student** instance as Actor attribute). Similarly, Depender and Dependee attributes of dependencies represent the two parties involved in a delegation relationship.



```
Resource Dependency Mark
  Depender Student
  Dependee Teacher
  Mode achieve
  Attribute
    constant exam : Exam
    constant gpm : GetPassingMark
    passed : boolean
  Invariant
    gpm.exam = exam ∧ gpm.actor = depender
  Invariant
    ¬∃ m : Mark ((m ≠ self) ∧ (m.gpm = gpm))
  Creation condition
    ¬Fulfilled (gpm)
  Fulfillment condition
    ∃ im : InitialMarking (im.exam = exam ∧
        im.actor = dependee ∧ Fulfilled (im))
  Fulfillment trigger
    G (Changed (passed) →
        ∃ r : ReEvaluation ((r.mark = self) ∧
          JustFulfilled (self)))
```

Fig. 6. Example of FT constraints.

An important aspect of FT is its focus on the conditions for the *fulfillment* of goals and dependencies. An intentional element is characterized by a mode, which declares the modality of its fulfillment. For example, the modality of goal **Pass[Exam]** is achieve, which means that the student expects to reach a state where the exam has been passed. Softgoal **Integrity** has a maintain mode, since the condition of no cheating is to be continuously maintained. There are other modalities, such as achieve&maintain, which is a combination of the previous two modes and requires the fulfillment condition to be achieved and then to be continuously maintained; and avoid, which means that the fulfillment conditions should be prevented.

2.2.2 The inner layer

The inner layer of an FT class declaration consists of constraints that describe the dynamic aspects of entities, actors, goals, and dependencies. Figure 6 presents an excerpt of constraints on the lifetime of dependency **Mark** of the course-exam management case study.

Invariant constraints of a class define conditions that should hold throughout the lifetime of all class instances. Typically, invariants define relations on the possible values of attributes, or cardinality constraints on the instances of a given class. For instance, the first invariant of Figure 6 states a relationship between attributes of the instances of the **Mark** class. The second invariant imposes a cardinality constraint for **Mark** objects, namely, there can be at most one **Mark** for a given **GetPassingMark** goal.

Creation and Fulfillment constraints define conditions on the two critical moments in the lifetime of intentional elements and dependencies, namely their *creation* and *fulfillment*. The creation of a goal is interpreted as the moment when an actor (the one associated to an intentional element, or the depender of a dependency) begins to desire a goal. The fulfillment

of a goal occurs when the goal is actually achieved. Creation and fulfillment constraints can be used to define conditions on those two time instants. Creation constraints can be associated with any class. Such constraints should be satisfied whenever an instance of the class is created. Fulfillment constraints can be associated only with intentional elements and with dependencies. These constraints should hold whenever a goal or softgoal is achieved, a task is completed, or a resource is made available. Creation and fulfillment constraints are further distinguished as sufficient conditions (keyword trigger), necessary conditions (keyword condition), and necessary and sufficient conditions (keyword definition). The actual constraints are described with formulas given in a typed first-order linear-time temporal logic (see Section 2.2.3).

In an FT specification, primary intentional elements (e.g., **Pass[Course]** and **Integrity**) typically have fulfillment constraints, but no creation constraints. We are not interested in modeling the reasons why a student wants to pass a course, or maintain her integrity, since these are taken for granted. Subordinate intentional elements (e.g., **Pass[Exam]**, **GetPassingMark[Exam]**) typically have constraints that relate their creation with the state of their parent elements. For instance, according to Figure 6, a creation condition for an instance of dependency **Mark** is that the parent goal **GetPassingMark** has not been fulfilled so far. In other words, if the student has received a passing mark, there is no need to ask for another mark. We note that the creation condition of dependency **Mark** together with the fulfillment condition of task **GetPassingMark[Exam]** elaborate the delegation relationship between **Student** and **Teacher** in the corresponding *i** diagram. Goal decomposition relationships can be specified in a similar fashion.

2.2.3 The FT temporal logic

In FT, constraints are described with formulas in the typed first-order linear-time temporal logic described by the following syntax:

$$
\begin{array}{llr}
f ::= & f \wedge f \mid f \vee f \mid \neg f \mid t & \text{(boolean op.)} \\
& \mid\ t = t \mid t < t \mid t \leq t & \text{(relational op.)} \\
& \mid\ \forall x : \mathit{sort}.f \mid \exists x : \mathit{sort}.f & \text{(quantifier)} \\
& \mid\ \mathsf{X}f \mid \mathsf{F}f \mid \mathsf{G}f \mid f\mathsf{U}f & \text{(future op.)} \\
& \mid\ \mathsf{Y}f \mid \mathsf{O}f \mid \mathsf{H}f \mid f\mathsf{S}f & \text{(past op.)}
\end{array}
$$

$$
\begin{array}{llr}
t ::= & c \mid x \mid t.a & \text{(const. and var.)} \\
& \mid\ \mathsf{self} \mid \mathsf{actor} \mid \mathsf{depender} \mid \mathsf{dependee} & \text{(special term)} \\
& \mid\ \mathsf{JustCreated}(t) \mid \mathsf{Changed}(t) & \\
& \mid\ \mathsf{Fulfilled}(t) \mid \mathsf{JustFulfilled}(t) & \text{(special pred.)}
\end{array}
$$

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 set of *future* and *past temporal operators*. These allow for expressing properties that are not limited to the current state, but may refer also to its past and future history. For instance, formula $\mathsf{X}f$ (next f) expresses the fact that formula f should hold in the next state reached by the model, while formula $\mathsf{Y}f$ (previous f) requires condition f to hold in the previous state. Formula $\mathsf{F}f$ (eventually f) requires that formula f is either true now or that it becomes eventually true in some future state; formula $\mathsf{O}f$ (once f) expresses the same requirement, but on the past states. Formula $\mathsf{G}f$ (always in the future f) expresses the fact that formula f should hold in the current state and in all future states of the evolution of the model, while formula $\mathsf{H}f$ (always in the past g) holds if f is true in the current state and in all past state of the model. Formula $f_1\mathsf{U}f_2$ (f_1 until f_2) holds if there is some future state where f_2 holds and formula f_1 holds until that state; finally, formula $f_1\mathsf{S}f_2$ (f_1 since f_2) holds if there is some past state where f_2 holds and formula f_1 holds since that state.

The terms t on which the formulas are defined may be integer and Boolean constants (c), variables (x), or may refer to the attribute's values of the class instances ($t.a$, where a can either be a standard attribute, or a special attribute like actor or depender). Also, instances may express properties about themselves using the keyword self (see the second invariant of dependency **Mark** in Figure 6). Special predicates can appear in temporal logic formulas. Predicate $\mathsf{JustCreated}(\mathbf{t})$ holds if element $\mathbf{t}$ exists in this state but not in the previous one. Predicate $\mathsf{Changed}(\mathbf{t})$ holds in a state if the value of term t has changed with respect to the previous state. Predicate $\mathsf{Fulfilled}(\mathbf{t})$ holds if $\mathbf{t}$ has been fulfilled, while predicate $\mathsf{JustFulfilled}(\mathbf{t})$ holds if $\mathsf{Fulfilled}(\mathbf{t})$ holds in this state, but not in the previous one. The two latter predicates are defined only for goals and dependencies.

The temporal logic incorporated in FT is quite expressive. However, only simple formulas are typically used in a specification. The possibility of anchoring temporal formulas to critical events in the lifetime of an object, together with the possibility of expressing modalities for goals and dependencies, provide implicitly an easy-to-understand subset of the language of temporal logics. Only when the lifetime events and the modalities are not sufficient for capturing all the temporal aspects of a condition, temporal operators need to appear explicitly in the formulas. This is the case, for instance, with dependency **Mark** (see Figure 6). The only temporal operator that appears explicitly in the specification is the one in the Fulfillment **trigger**. This operator is needed since we want to bind the fulfillment of the dependency with a condition that should hold from that moment on (namely, the fact that attribute **passed** should change only because of a re-evaluation).

2.2.4 Assertions and possibilities

The constraints represented in Figure 6 express conditions that are required to hold for all possible scenarios. In an FT specification, we can also specify properties that are desired to hold in the domain, so that they can be verified with respect to the model. Figure 7 presents such properties for the course-exam management case study. We distinguish between assertion properties (A1-4) which are desired to hold for all valid evolutions of the FT specification, and possibility properties (P1-4) which should hold for at least one valid scenario. Prop-

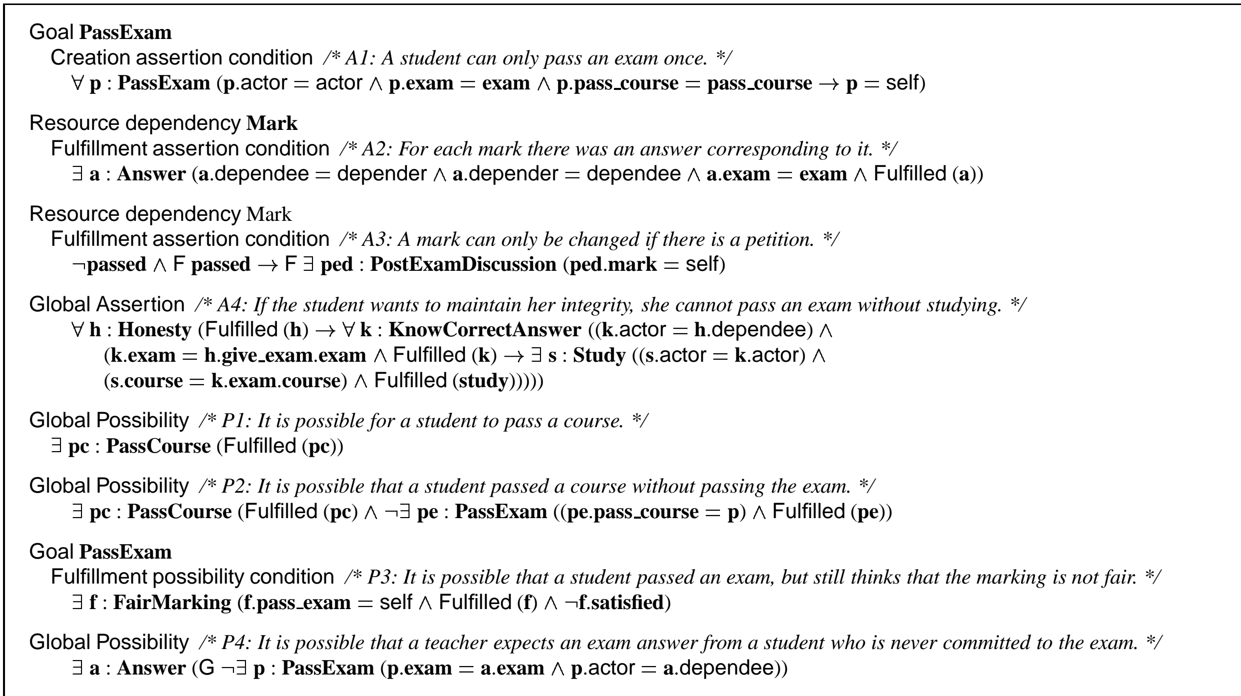


Fig. 7. Example of Formal Tropos properties.

erties A1, A2, A3, and P3 are "anchored" to some important event in the lifetime of a class instance. For example, assertion A2 requires that, whenever an instance of **Pass[Exam]** is created, no other instance of **Pass[Exam]** exists corresponding to the same exam and the same student. Properties A4, P1, P2, and P4 are examples of "global properties", i.e., they express conditions on the entire model, and are not attached to any particular event.

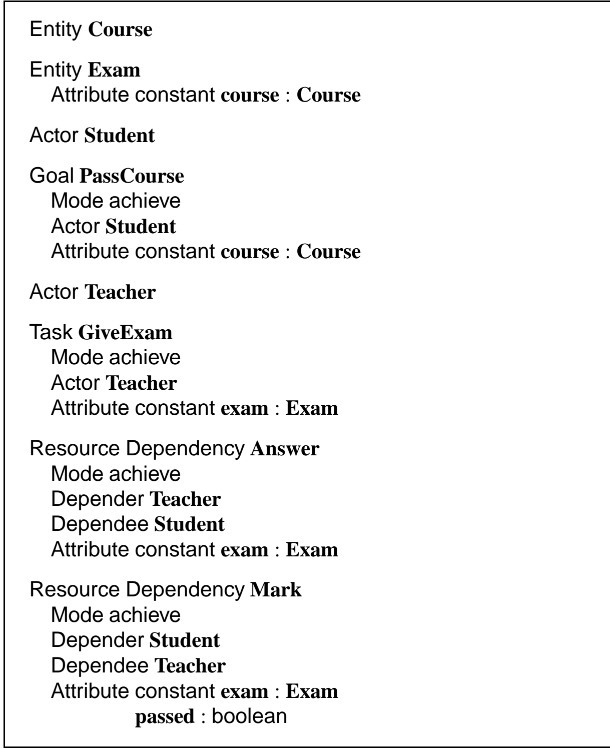
2.2.5 From *i** to FT: Translation guidelines

Developing a satisfactory formal software system specification can be hard, even when one starts from a good informal model. According to our experience, the difficulties in developing an FT specification can be substantially reduced if one extracts as much information as possible from the *i** model in order to produce a "reasonable" initial FT model. In fact, most of the constraints of an FT specification already appear implicitly in the *i** model. For instance, for the dependency **Mark** (see Figure 6), the invariants, the creation and the fulfillment conditions express constraints that are related to goal delegation and cardinality constraints in the *i** model. These constraints can be systematically derived from the *i** model by applying specific translation rules. In order to capture the nature of the application domain, additional non-standard constraints need to be manually added to the FT specification. For instance, the last constraint for dependency **Mark** in Figure 6 expresses that a sufficient condition for considering the dependency fulfilled is that we are committed to change the passing status of a mark only if a re-evaluation has occurred.

The following are some of the heuristic rules we have identified for generating an FT specification from the *i** model:

- The default creation condition of a sub-goal is that the parent goal exists, but has not been fulfilled yet. The default creation condition for a dependency is that the depender goal exists but has not been fulfilled yet.
- The fulfillment condition of a parent goal (or task) usually depends on the fulfillment of the sub-goals (tasks). If the sub-goals are connected to the parent goal with *means-ends* links, then the fulfillment of *at least one* of the sub-goals is necessary for the fulfillment of the parent goal. If they are connected with *decomposition* links then the fulfillment of *all* the subgoals is necessary.
- Parent goals and sub-goals typically share the same entity and owner. So, an invariant condition should be added to the sub-goal in order to force the binding between the attributes of the two goals. For instance, **Pass[Exam]** and **Take[Exam]** refer to the same **exam**.
- When there is a prior-to constraint between two sub-goals with a common parent, an extra creation condition needs to be added to the goal that comes later. Such constraints state that a necessary condition for the creation of the later goal is that the previous goal has already been fulfilled.

These rules are not meant to be definitive and exhaustive, but their systematic application leads to a quick generation of a reasonable FT model, that can then be corrected and improved using the techniques described in the next sections. We are currently developing a tool to support the designer in the semi-automatic extraction of an initial FT specification starting from an *i** diagram.



Entity **Course**

Entity **Exam**
  Attribute constant **course** : **Course**

Actor **Student**

Goal **PassCourse**
  Mode achieve
  Actor **Student**
  Attribute constant **course** : **Course**

Actor **Teacher**

Task **GiveExam**
  Mode achieve
  Actor **Teacher**
  Attribute constant **exam** : **Exam**

Resource Dependency **Answer**
  Mode achieve
  Depender **Teacher**
  Dependee **Student**
  Attribute constant **exam** : **Exam**

Resource Dependency **Mark**
  Mode achieve
  Depender **Student**
  Dependee **Teacher**
  Attribute constant **exam** : **Exam**
      passed : boolean

Fig. 8. The initial FT specification for the analysis of the case study.

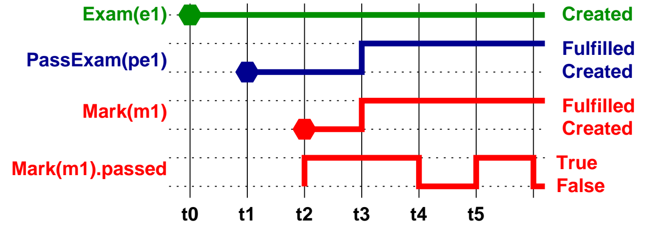
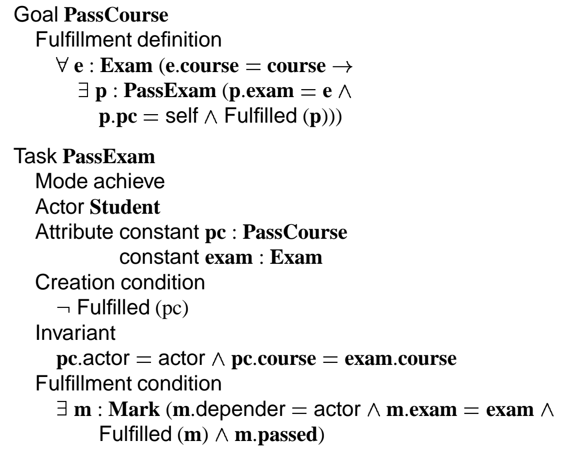


Fig. 9. A scenario where attribute **passed** oscillates.



Goal **PassCourse**
  Fulfillment definition
    ∀ **e** : **Exam** (**e**.**course** = **course** →
      ∃ **p** : **PassExam** (**p**.**exam** = **e** ∧
        p.**pc** = self ∧ Fulfilled (**p**)))

Task **PassExam**
  Mode achieve
  Actor **Student**
  Attribute constant **pc** : **PassCourse**
      constant **exam** : **Exam**
  Creation condition
    ¬ Fulfilled (pc)
  Invariant
    pc.actor = actor ∧ **pc**.**course** = **exam**.**course**
  Fulfillment condition
    ∃ **m** : **Mark** (**m**.depender = actor ∧ **m**.**exam** = **exam** ∧
      Fulfilled (**m**) ∧ **m**.**passed**)

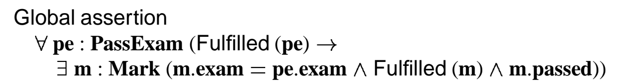
3 Formal Analysis of the Case Study

We now illustrate the usage of FT in refining an early requirements specification. In the next section we describe the T-Tool, a tool that supports the analysis performed in this section. For explanatory purposes we focus on a subset of the FT specification. This initial model, shown in Figure 8, is strongly under-specified. We will interactively improve and refine it, guided by results provided by the analysis.

In this section, scenarios are represented by diagrams like the one in Figure 9, and can be automatically generated by the T-Tool. With $t_0, t_1, \ldots$ we denote different time instants of the scenario. Symbol ● indicates the instant of creation of an object. For simplicity, the diagrams report only the relevant objects of the scenario.

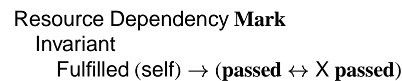
As a first refinement step we consider the achievement of goal **PassCourse**. In our case study, a student passes a course if she takes all the exams for the course and if there is a passing mark for each exam. To capture this requirement, we modify the FT model by adding task **PassExam** as a means for the achievement of goal **PassCourse**. To fulfill goal **PassCourse** we require that for each exam of the course there exists at least one instance of **PassExam** that is fulfilled. We also require that an instance of object **PassExam** can be created only if the corresponding **PassCourse** has not been fulfilled yet: if the goal **PassCourse** has already been fulfilled, there is no need to pass any further exam for that course. We allow for several instances of the class **Mark** for each **PassExam**. A sufficient condition for passing the exam is that at least one corresponding mark is passing.

The analysis of the extended specification reveals some problems. The specification allows for unrealistic scenarios such as the one depicted in Figure 9. This scenario shows, correctly, that an instance of **PassExam** is created at time t_1, and that it is fulfilled at time t_3, when a passing **Mark** for that exam has been fulfilled. However, the scenario shows also that the value of attribute **passed** of the dependency **Mark** may oscillate once the dependency has been fulfilled. Consider the following assertion, that requires a passing mark to be present if a **PassExam** goal is fulfilled.



Global assertion
  ∀ **pe** : **PassExam** (Fulfilled (**pe**) →
    ∃ **m** : **Mark** (**m**.**exam** = **pe**.**exam** ∧ Fulfilled (**m**) ∧ **m**.**passed**))

It does not hold between time t_4 and time t_5 of the scenario depicted in Figure 9. To enforce the requirement that a mark – once produced [1] – does not change its value, we add the following invariant constraint to the dependency **Mark**.



Resource Dependency **Mark**
  Invariant
    Fulfilled (self) → (**passed** ↔ X **passed**)

In the FT specification, one would like to specify that the teacher is not going to give an exam if there is no student interested in passing it. Once the task of giving the exam has been created, it can be fulfilled only once all the answers given by the students have been marked. These requirements can be modeled by the following additional constraints for task **GiveExam**.

[1] Notice that the value of attribute **passed** is only relevant once the dependency has been fulfilled, therefore we do not care if it changes before its fulfillment.

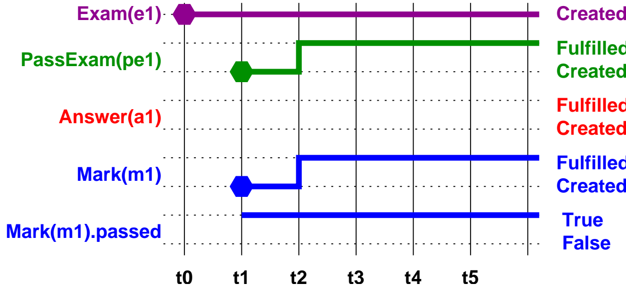


Fig. 10. A student receives a mark for an exam without providing an answer.

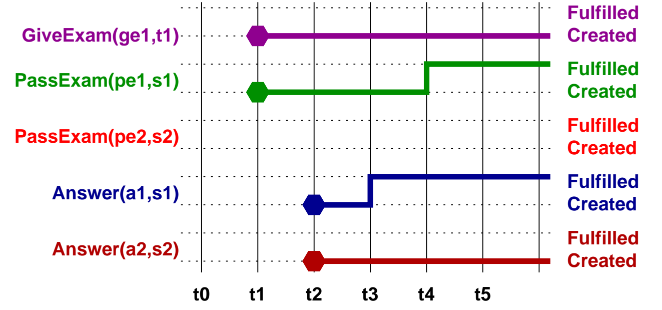


Fig. 11. The teacher waits for an answer that will never arrive.

Task **GiveExam**
  Creation condition
    ∃ **pe** : **PassExam** (**pe.exam** = **exam**)
  Fulfillment condition
    ∀ **a** : **Answer** ((**a.exam** = **exam** ∧ **a**.depender = actor) →
      ∃ **m** : **Mark** (**m.exam** = **exam** ∧ **m**.dependee = actor ∧
        m.depender = **a**.dependee ∧ Fulfilled (**m**)))

A first, trivial, problem of this specification is that it allows for scenarios where a mark is given to a student even if there is no answer from that student. For example, according to the scenario of Figure 10, instances of objects **PassExam** and **Mark** are created at time t_1 and fulfilled at time t_2, while no instance of object **Answer** is ever created. These behaviors can be easily disallowed by adding the following creation constraint to dependency **Mark**.

Resource Dependency **Mark**
  Creation
    condition for domain
      ∃ **a** : **Answer** (**a**.depender = dependee ∧
        a.dependee = depender ∧
          a.exam = **exam** ∧ Fulfilled (**a**))

The specification also suffers of a more subtle problem. We expect that, thanks to the creation condition of **GiveExam**, the teacher never waits for answers from students that are not committed to pass the exam. This expectation is captured by the following assertion.

Global assertion
  ∀ **a** : **Answer** (F ∃ **pe** : **PassExam** (**pe.exam** = **a.exam** ∧
    pe.actor = **a**.dependee))

Unfortunately, the scenario depicted in Figure 11 shows a case where this assertion is not valid. In this scenario the teacher gives the exam at time t_1 for student s_1. This student is committed to pass the exam, as proven by the instance of the class **PassExam** that is created at time t_1. However, the teacher is waiting for an answer also from student s_2 even if this student is not interested in giving the exam. The subtlety of this problem relies in the fact that more than one instance of class **Student** is necessary in order to reveal it. This behavior suggests that we need to refine the specification by introducing a registration mechanism to the exams. We also constrain the creation of resource **Answer** for an exam to the existence of a student aiming to pass that exam.

Resource Dependency **Answer**
  Creation condition
    ∃ **p** : **Passexam** (**p**.actor = dependee ∧ **p**.exam = exam)

As the specification grows, it is important to detect overconstrained situations that rule out desired behaviors. For instance, we want to make it sure that the specification allows a student to pass a course. This requirement is formulated with the following possibility.

Global possibility
  ∃ **p** : **PassCourse** (Fulfilled (**p**))

An existence proof for this possibility is shown in Figure 12. The class instance **PassCourse(pc1)** is created at time t_1 jointly with the class instances **PassExam(pc1)**, **Mark(m1)** and **Exam(e1)**. The class instance **Mark(m1)** is fulfilled at time t_2, and since **Mark(m1).passed** is true then also **PassExam(pe1)** is fulfilled. At time t_3 object **PassCourse(pc1)** is fulfilled.

A more interesting scenario that we do not want to rule out is that, if a student fails to pass an exam, she should still be able to pass the course. This requirement can be formulated by the following possibility.

Global possibility
  ∃ **s** : **Student** (∃ **m** : **Mark** (∃ **p** : **PassCourse** : (
    m.depender = **s** ∧ Fulfilled (**m**) ∧ ¬ **m.passed** ∧
      p.course = **m.exam.course** ∧ **p**.actor = **s** ∧ F Fulfilled (**p**))))

This possibility is false when there is one instance per class. This is the case because there is no possibility to obtain a second mark for the exam. We have to consider multiple instances for various classes to satisfy this possibility.

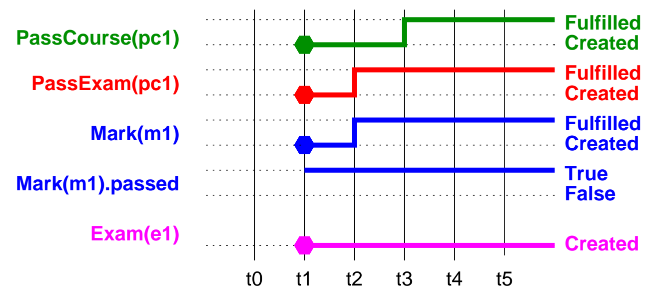


Fig. 12. A student passes a course.

4 The T-Tool

The T-Tool is based on finite-state model checking [9]. Its input is an FT specification along with parameters that specify which parts of the specification to consider. These parameters define an upper bound to the number of class instances to be created during model checking. On the basis of this input, the T-Tool builds a finite model that represents all possible behaviors of the domain that satisfy the constraints of the specification. The T-Tool then verifies whether this model exhibits the desired behaviors. The T-Tool provides different verification functionalities, including interactive animation of the specification, automated consistency checks, and validation of the specification against possibility and assertion properties. The verification phase usually generates feedback on errors in the FT specification and hints on how to fix them. The verification phase iterates on each fixed version of the model, possibly with different upper bounds of the number of class instances, until a reasonable confidence on the quality of the specification has been achieved.

4.1 T-Tool functionalities

4.1.1 Animation

An advantage of formal specifications is the possibility to animate them. Through animation, the user can obtain immediate feedback on the effects of constraints. An animation session consists of an interactive generation of a valid scenario for the specification. Stepwise, the T-Tool proposes to the user next possible valid evolutions of the animation and, once the user has selected one, the system evolves the state of the animation. Animation allows for a better understanding of the specified domain, as well as for the early identification of trivial bugs and missing requirements that are often taken for granted, and are therefore difficult to detect in an informal setting. Animation also facilitates communication with stakeholders by generating concrete scenarios for discussing specific behaviors.

4.1.2 Consistency checks

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. The simplest consistency check checks whether there is any valid scenario that respects all the constraints of the FT specification. Another consistency check verifies whether there exists a valid scenario where all the class instances specified by input parameters will be eventually created. This check aims at verifying whether these parameters violate any cardinality constraint in the specification. The T-Tool also checks whether there exists a valid scenario where all the instances of a particular goal or dependency will be eventually created and fulfilled, i.e., the fulfillment conditions for that goal or dependency are "compatible" with other constraints in the specification.

4.1.3 Possibility checks

Possibility checks verify whether the specification is overconstrained, that is, whether we have ruled out scenarios expected by the stakeholders. When a possibility property of the FT specification is checked, the T-Tool verifies that there are valid traces of the specification that satisfy the condition expressed in the possibility. The expected outcome of a possibility check is an example trace that confirms that the possibility is valid. In a sense, possibility checks are similar to consistency checks, since they both verify that the FT specification allows for certain desired scenarios. Their difference is that consistency is a generic formal property independent of the application domain, while possibility properties are domain-specific.

4.1.4 Assertion checks

The goal of assertion properties is dual to that of possibilities. The aim is to verify whether the requirements are underspecified and allow for scenarios violating desired properties. Unsurprisingly, the behavior of the T-Tool in the case of assertion checks is dual to the behavior for possibility checks, namely, the tool explores all the valid traces and checks whether they satisfy the assertion property. If this is not the case, an error message is reported and a counter-example trace is generated. Such counter-examples facilitate the detection of problems in the FT specification that caused the assertion violation. For instance, in the course-exam management case study, a sample assertion is "a student can never pass a course without taking all the exams of the course and without doing a research project". If this (quite reasonable) assertion is false, the T-Tool will produce a trace that shows under what circumstances the student can pass the course without passing exams and doing a research project. Discussions with the stakeholder may then clarify whether the trace produced corresponds to a valid scenario (and hence the assertion has to be changed) or whether the FT specification has to be strengthened in order to prohibit the counter-example.

4.2 The T-Tool architecture

The T-Tool performs the verification of an FT specification in two steps (see Figure 13). 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, which is built on top of the NuSMV model checker [8].

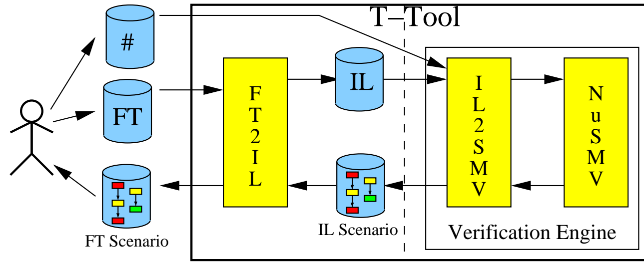
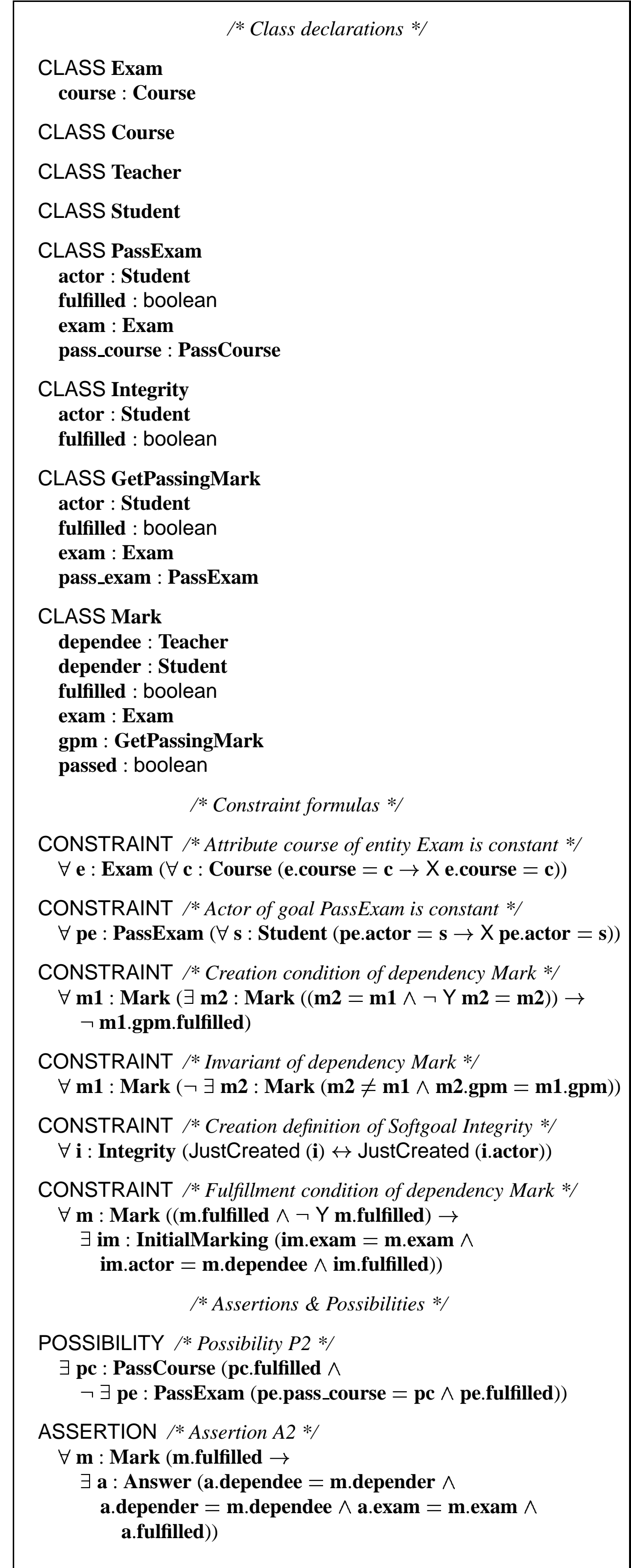


Fig. 13. The T-Tool framework.



```
/* Class declarations */
CLASS Exam
  course : Course
CLASS Course
CLASS Teacher
CLASS Student
CLASS PassExam
  actor : Student
  fulfilled : boolean
  exam : Exam
  pass_course : PassCourse
CLASS Integrity
  actor : Student
  fulfilled : boolean
CLASS GetPassingMark
  actor : Student
  fulfilled : boolean
  exam : Exam
  pass_exam : PassExam
CLASS Mark
  dependee : Teacher
  depender : Student
  fulfilled : boolean
  exam : Exam
  gpm : GetPassingMark
  passed : boolean
/* Constraint formulas */
CONSTRAINT /* Attribute course of entity Exam is constant */
  ∀ e : Exam (∀ c : Course (e.course = c → X e.course = c))
CONSTRAINT /* Actor of goal PassExam is constant */
  ∀ pe : PassExam (∀ s : Student (pe.actor = s → X pe.actor = s))
CONSTRAINT /* Creation condition of dependency Mark */
  ∀ m1 : Mark (∃ m2 : Mark ((m2 = m1 ∧ ¬ Y m2 = m2)) →
    ¬ m1.gpm.fulfilled)
CONSTRAINT /* Invariant of dependency Mark */
  ∀ m1 : Mark (¬ ∃ m2 : Mark (m2 ≠ m1 ∧ m2.gpm = m1.gpm))
CONSTRAINT /* Creation definition of Softgoal Integrity */
  ∀ i : Integrity (JustCreated (i) ↔ JustCreated (i.actor))
CONSTRAINT /* Fulfillment condition of dependency Mark */
  ∀ m : Mark ((m.fulfilled ∧ ¬ Y m.fulfilled) →
    ∃ im : InitialMarking (im.exam = m.exam ∧
      im.actor = m.dependee ∧ im.fulfilled))
/* Assertions & Possibilities */
POSSIBILITY /* Possibility P2 */
  ∃ pc : PassCourse (pc.fulfilled ∧
    ¬ ∃ pe : PassExam (pe.pass_course = pc ∧ pe.fulfilled))
ASSERTION /* Assertion A2 */
  ∀ m : Mark (m.fulfilled →
    ∃ a : Answer (a.dependee = m.depender ∧
      a.depender = m.dependee ∧ a.exam = m.exam ∧
        a.fulfilled))
```

Fig. 14. Excerpt of the IL translation for our running example.

4.2.1 From FT to IL

The FT2IL module takes care of the translation of an FT specification to a corresponding IL specification. Moreover, it translates back in FT the counter-examples scenarios produced by the verification engine. Thus, the internals of the verification engine are hidden to the user.

IL can be seen as a simplified version of FT, where the syntactic sugar of the FT language is removed. The focus of IL is on the dynamic aspects of the application domain. In Figure 14, we give an excerpt of the IL translation for our running example. An IL model consists of three parts: class declarations, constraints, and assertion & possibility properties.

The *class declarations* (keyword CLASS) define the data types of the specification. Their instances represent entities, actors, and dependencies (i.e., the outer layer) of the FT specification. We note that some attributes, not explicitly declared as such in the FT specification, are added to class definitions during the translation. This is the case, for instance, for the attribute **actor** of type **Student** for classes **PassExam** and **Integrity**, as well as for the attributes **depender** and **dependee**, respectively of type **Student** and **Teacher**, for dependency **Mark**. A Boolean attribute **fulfilled** is added to classes corresponding to goals, task, resources, and softgoals. Notice that, fulfillment is a primitive concept in FT (Fulfilled predicate), while in IL it is encoded as a state variable (attribute **fulfilled**). This is an example of the change of focus that occurs when translating an FT specification into IL. However, the IL still allows for the dynamic creation of class instances. For instance, in Figure 14, predicate JustCreated is used in the fifth constraint to check whether a given instance of a class has been created in the current time instant of a scenario.

Constraint formulas (keyword CONSTRAINT) restrict the valid temporal behaviors of the system. Some of these formulas model the semantics of an FT specification. For instance, the first two constraint formulas in Figure 14 express respectively the fact that attribute **course** of all instances of **Exam** and attribute **actor** of all instances of **Pass[Exam]** are constant. Other formulas correspond to the temporal constraints that constitute the inner layer of the FT specification. For instance, the third and fifth constraint formulas in Figure 14 correspond to the creation condition of classes **Mark** and **Integrity** respectively. The fourth constraint corresponds to the cardinality constraint for class **Mark**.

In IL constraints on the fulfillment and creation of classes are no longer syntactically anchored to the corresponding class. There is thus the need to give them a "context" to precisely define their meaning. This context is provided by the translation rules that map an FT specification into a corresponding IL one. For instance, the fulfillment condition f of a dependency **D** with an achieve modality is mapped into a constraint of the form

$$\forall\, \mathbf{d} : \mathbf{D}\ ((\mathbf{d.fulfilled} \wedge \neg\, \mathsf{Y}\ \mathbf{d.fulfilled}) \rightarrow f)$$

stating that "when an achieve dependency becomes fulfilled, its fulfillment condition should hold". This is the rule that has been applied to the fulfillment condition of dependency **Mark** in Figure 6 which results in the sixth constraint of Figure 14. In the translation from FT to IL, auxiliary temporal operators are added to the IL specification. Not only these operators depend on the kind of formula being translated, but also on the mode of the dependency. For instance, in the case of a maintain dependency, the translation of the fulfillment condition f is given by the rule

$$\forall\, \mathbf{d}\!: \mathbf{D}\ (\mathbf{d}.\mathsf{fulfilled} \rightarrow (\mathsf{G}\ f \wedge \mathsf{H}\ f))$$

stating that "if a maintain dependency is fulfilled, then its conditions should hold during the full lifetime of the dependency". Similar rules apply also to goals, softgoals, task and resources.

The *possibility* and *assertion* formulas (keywords POSSIBILITY and ASSERTION respectively) state expected properties of the behavior of the system. They correspond to the assertion and possibility properties of the FT specification. Notice that, also in the translation of assertions and possibilities, there is the need to add a context (see the assertion in Figure 14).

The semantics of an IL specification is given in terms of sets of scenarios, where each scenario is an infinite sequence of states. Each state consists of a set of instances of the classes of the IL specification and of a definition of the values of the attributes of these instances. Valid states must conform to the attribute sorts declared in the specification. A valid scenario is a sequence of valid states that satisfy all the temporal conditions expressed in the CONSTRAINT declarations of the specification. We refer to [10] for a precise definition of the semantics of the IL.

The IL plays a fundamental role in bridging the gap between FT and formal methods. First of all, it is much more compact than FT, and therefore allows for a much simpler formal semantics, as discussed above. Second, it is rather independent of the particular constructs of FT. By moving to different domains, it will probably become necessary to "tune" FT, for instance by adding new modalities for the dependencies. The formal approach described in this paper can be also applied to these dialects of FT, at the cost of defining a new translation. Furthermore, the IL can be applied to requirements languages that are based on a different set of concepts than those of FT, such as KAOS [17]. Finally, the IL, while more suitable to formal analysis, is still independent from the particular analysis techniques that we employ. For the moment, we have applied only model checking techniques; however, we plan to also apply techniques based on satisfiability or theorem proving.

4.2.2 The model checking verification engine

The actual verification is performed by NuSMV [8]. NuSMV is a state-of-the art model checker based on symbolic model checking techniques. Symbolic techniques have been developed to reduce the effects of the state-explosion problem, thereby enabling the verification of large designs [9, 18]. NuSMV adopts symbolic model checking algorithms based on Binary Decision Diagrams (BDD) [6] and on propositional satisfiability (SAT) [4]. BDD-based model checking performs an exhaustive traversal of the model by considering all possible behaviors in a compact way. Such exhaustive exploration allows BDD-based model checking algorithms to conclude whether a given property is satisfied (or falsified) by the model. On the other hand, this exhaustive exploration makes BDD-based model checking very expensive for large models. SAT-based model checking algorithms look for a trace of a given length that satisfies (or falsifies) a property. SAT-based algorithms are usually more efficient than BDD-based algorithms for traces of reasonable length, but, if no trace is found for a given length, then it may still be the case that the property is satisfied by a longer trace. That is, SAT-based model checking verifies the satisfiability of a property only up to a given length, and is hence called Bounded Model Checking (BMC) [4]. The T-Tool exploits both BDD-based and SAT-based model checking.

Several extensions have been applied to the NuSMV model checker to allow for the verification of IL specifications. An IL2SMV module has been added. It takes an IL specification and builds a finite state machine in the NuSMV format. Given the IL specification and the upper bounds of the number of class instances (# in Figure 13), IL2SMV synthesizes a model for the specification. The states of the model respect the CLASS part of the IL specification, while its transitions are those that respect the temporal specification defined by the CONSTRAINT formulas. Since the NuSMV formalism does not allow for the creation of new objects at run-time, during the translation a special flag is added to each class to deal with instance creation. Quantifiers in IL are interpreted over the number of class instances that exist in the current state. To construct the model, IL2SMV adopts the synthesis algorithm for LTL specification provided by NuSMV. An immediate outcome of the synthesis process is consistency checking. In fact, if a specification is inconsistent with respect to the declared number of instances, the synthesis process fails and no automaton is built.

A new more flexible interactive animator has also been added to NuSMV. It allows both for an interactive exploration of the automaton, and for a random execution of a certain number of steps. Finally, the BMC engine has been extended with past operators [2] to allow for the verification of ASSERTION and POSSIBILITY formulas against the executions of the automaton.

4.3 Heuristics for model construction and property verification

The T-Tool builds a finite state model from an infinite state specification. Thus, an upper bound of the number of class instances has to be specified in the FT specification. The choice of the upper bound plays a critical role in the verification step. There can be bugs that only appear when a certain number of class instances are allowed, as well as valid scenarios that require a given number of class instances. Therefore, the checks performed by the T-Tool only guarantee the correctness of the specification with the considered number of class instances. In practice, it is convenient to generate and check various models with different number of class instances, so that a larger set of possible cases is covered in the verification. As we set the upper bound of class instances, three basic approaches are used. First, a uniform upper bound can be set for all classes, e.g., a 1-instance or a 2-instance case. Second, according to the cardinality constraints in the $i*$ model, different upper bounds can be set for different groups of classes, e.g., there is 1 teacher vs. 2 students, 1 course vs. 2 exams, etc. Third, a subset of the classes can be selected for instantiation, based on the property to be verified. No instance is allowed for the classes that are not selected. This approach is referred to as the reduced case.

For complex FT specifications, verification of properties against a given model can take a very long time and can require considerable effort. For this situation, we provide some guidelines for an effective application of the verification methods supported by the T-Tool.

For possibility (and consistency) checks, SAT-based bounded model checking techniques are preferable, as they are very effective in finding scenarios of bounded length that satisfy a given property. Since most scenarios are actually short, if no scenario is found within reasonable length (typically 5 to 10 steps), then it is likely the case that the possibility cannot be satisfied. In this case, direct inspections of the specification and interactive animations have shown to be effective means for finding the problem in the FT specification.

For assertion checks, SAT-based bounded model checking techniques can only be used to give preliminary results. In fact, these techniques are able to find counter-examples if the given assertion is false, but are able to prove the truth of the assertion only up to a given length of the possible counter-examples. To guarantee that an FT specification satisfies a given assertion, BDD-based techniques are a must, since they allow for an exhaustive analysis of the model. A strategy that can help when checking assertions using BDD-based techniques is to consider only a subset of the constraints in the FT specification. The rationale behind this is that whenever we check an assertion φ on a specification composed of a finite set I of constraints C_i with $i \in I$, we are looking for solutions to the following problem: $\bigwedge_{i \in I} C_i \Rightarrow \varphi$. If we can derive a positive answer using a subset $J \subseteq I$ of constraints, the job is done. Indeed, the more constraints we add, the more restricted is the behavior of the system. Since we are interested in verifying that all possible scenarios compatible with the specification satisfy φ, if we prove that φ holds in an under-constrained system, φ must hold in the more constrained system. If we fail in checking the property we need to consider a new set of constraints L, such that $J \subset L \subseteq I$, and iterate. The counter-example produced for subset J can guide the selection of new constraints to be added to L, since it exhibits a possible behavior that violates relevant constraints not yet considered. This iterative process will eventually terminate since the set of constraints I is finite. While in theory the initial set of constraints can be chosen arbitrarily (e.g., it can be the empty set), in practice starting with a good guess for J is very important to reduce the number of iterations. In most practical cases, the user has in mind the reason why a given assertion needs to hold and how to exploit such knowledge to choose a suitable set J. We remark that the "abstraction" techniques described here are common practice in the model checking community [3].

5 Experimental results

Following the guidelines described in the previous sections, we have conducted several iterations of experiments. During each iteration, an FT specification was validated by human inspection, animation, consistency checking, and possibility/assertion verification. Whenever a bug was detected, the FT specification (and, in some cases, the $i*$ model) was revised, and a new iteration was performed. This iterative refinement of the specification ended when all checks on the FT specification were successful.

5.1 Setup of the experiments

In order to illustrate the performance of the tool, and the verification process, we present the experiments results of an intermediate version of the FT specification that still contains some bugs. Moreover, we report the results only for some of the assertions and possibilities that are present in the model, namely for assertions A1-4 and for possibilities P1-4 in Figure 7. More results can be found at the URL `http://sra.itc.it/tools/t-tool/experiments/cm`.

To stress the scalability of the proposed verification techniques, we have performed the tests considering models of different size. More precisely, we have considered different upper bounds to the number of instances for each class. We report here the case of 1 and 2 instances for each class, and one intermediate 1..2 case where we allow 2 instances for some classes (in particular, the student and its goals and tasks), but only 1 instance for other classes (the teacher and its tasks, and the course). Moreover, we experimented with the different model checking techniques, namely SAT-based bounded model checking ("BMC" in the tables), BDD-based model checking ("BDD"), and, in the case of assertions, BDD-based model checking on reduced models, as described in Section 4.3 ("BDD-reduced"). The case study is composed

Possibility Checks						
	1 instance		**1..2 instances**		**2 instances**	
	BMC	**BDD**	**BMC**	**BDD**	**BMC**	**BDD**
P1	Valid[3] 9.4sec / 29Mb	Valid[3] 1786sec / 64Mb	Valid[3] 55.7sec / 77Mb	Undecided T.O.	Valid[3] 860sec / 295Mb	Undecided M.O.
P2	Valid[3] 9.3sec / 29Mb	Valid[3] 1719sec / 63Mb	Valid[3] 55.6sec / 77Mb	Undecided T.O.	Valid[3] 842sec / 295Mb	Undecided M.O.
P3	Valid[4] 14.2sec / 38Mb	Valid[5] 1979sec / 64Mb	Valid[4] 94.9sec / 96Mb	Undecided T.O.	Valid[4] 1629sec / 375Mb	Undecided M.O.
P4	Undecided[10] 105sec / 84Mb	Invalid 1626sec / 64Mb	Undecided[10] 2143sec / 237Mb	Undecided T.O.	Undecided[4] T.O	Undecided M.O.

Table 1. Results for possibility checks.

Assertion Checks						
	1 instance			**1..2 instances**		
	BMC	**BDD**	**BDD-reduced**	**BMC**	**BDD**	**BDD-reduced**
A1	NoBug[10] 100sec / 83Mb	Valid 1298sec / 64Mb	Valid 0.3sec / 2Mb	NoBug[10] 1086sec / 237Mb	Undecided T.O.	Valid 30.8sec / 4.2Mb
A2	NoBug[10] 111sec / 84Mb	Valid 1295sec / 64Mb	Valid 44sec / 17Mb	Invalid[3] 57.6sec / 77Mb	Undecided T.O.	Invalid[7] 757sec / 100Mb
A3	NoBug[10] 107sec / 83Mb	Valid 2110sec / 64Mb	Valid 2.5sec / 4Mb	NoBug[10] 2837sec / 234Mb	Undecided T.O.	Undecided T.O.
A4	NoBug[10] 114sec / 83Mb	Valid 1297sec / 63Mb	Valid 0.1sec / 2Mb	NoBug[9] T.O.	Undecided T.O.	Undecided T.O.

Table 2. Results for assertion checks.

of 33 classes and 229 constraints. The model with 1 instance per class requires 477 Boolean state variables, while the 2 instance requires 1077 Boolean state variables. Thus, the state space grows from 2^{477} to 2^{1077} states while moving from the 1 instance to the 2 instance per class.

5.2 Results

The results of the experiments carried out are reported in Table 1 and Table 2. The experiments were executed on a PC Pentium III, 700 MHz, 6GB of RAM, running Linux. All the verification tests have been executed with a time limit of 3600 seconds (1 hour) and memory limit of 1GB. For each problem we report the CPU time in seconds and the amount of memory in MB. With "T.O." we mark the experiments that did not complete within the time limit, while with "M.O." we mark those experiments that exced memory limits. The maximum length considered for bounded model checking experiments is 10.[2] The experiments show that:

1. Possibilities P1-3 are valid, and witness scenarios of length 3, 3 and 4 are produced by the T-Tool.
2. Possibility P4 is invalid. No witness scenario is found up to length 10 for the 1 and 1..2 instances and up to length 4 for 2 instances. An analysis of the specification shows that possibility P4 ("A teacher expects an exam answer from a student that does not intend to pass the exam") cannot occur, because we have assumed that the teacher knows which students want to pass the exam (e.g., by requiring them to register). This possibility has been removed in the final version of the FT specification.
3. Assertions A1, A3, and A4 are correct. No counter-example scenarios are found in the performed checks.
4. Assertion A2 is false. A counter-example of length 3 is found in the 1..2 instances case. This is due to a missing creation condition for dependency **Mark** that allows the teacher to assign marks to students that have not provided exam answers. This bug has been fixed in the final version of the FT specification. We remark that in the case of 1 instance no counter-example is found. This is right since, according to the FT specification, the teacher only starts marking if at least one student takes the exam.

[2] The experiments confirm that this is a reasonable bound: all generated witness scenarios and counter-examples are of length 5 or shorter.

5.3 Discussion

5.3.1 Effectiveness

For our case study, the proposed approach was effective in producing an FT specification of good quality. It also led to an improved understanding of the domain by revealing several tricky aspects of the case study. The validation techniques provided by the T-Tool have been useful in detecting bugs, while animation was useful during early validation steps by identifying trivial bugs. For instance, due to a missing creation condition for the student goal **TakeExam**, a student was allowed to try to take an exam even if no teacher was giving it. Likewise, the consistency checks have been able to detect a trivial error in the creation condition of student's goal **Study**, which did not allow two students to study the same course. The validation of assertions and possibilities has revealed subtle bugs due to the interaction of different goals, dependencies and constraints. For instance, due to an error in the fulfillment condition of **ReceiveAnswers**, a student could

prevent the teacher from fulfilling the task **GiveExam** by declaring her intention to take the exam and by never taking it. In another case, a student could not decide on the fairness of marking (softgoal **FairMarking**) even after she received a **Mark**, since she was expecting a marking scheme from the wrong teacher. This was due to a missing creation condition in the dependency **FairMarkingScheme**. In both cases, the T-Tool's ability to generate counter-examples helped in pin-pointing the problem.

A limiting factor of the current framework consists in the fact that correctness of the specification can be asserted up to the considered upper bounds of the number of class instances. We are currently investigating heuristics and techniques for choosing upper bounds that guarantee the correctness of the FT specifications regardless of the upper bounds.

5.3.2 Performance

The performance results on the T-Tool are very encouraging, even though further work is needed in order to allow for a black box usage of these techniques. The fact that the T-Tool allows for the usage of different verification techniques is a very important factor for its effectiveness. In particular, BDD-based and BMC-based model checking complement each other. BMC-based verification is efficient in checking possibility properties. On average, a valid scenario for a possibility property can be produced in a few seconds. BMC-based verification is also good for a preliminary verification of assertion properties. On the other hand, BDD-based model checking does not work in practice for large models with big state spaces.

The experiments show that the usage of the abstraction techniques described in Section 4.3 for checking assertions on a reduced model is very promising. For most properties, the use of these techniques has resulted in speed-ups of one to two orders of magnitude with respect to the case of the whole model. This allows us to check the correctness of assertions for the 1..2 instances case, but is not enough for the 2-instances case.

The animation of the specification was useful, but it should be improved by reducing the setup time and by improving its usability, e.g., allowing the automated generation of a scenario given a set of target states.

6 Related work and conclusions

In this work, we have proposed a framework for the specification and verification of early requirements. This framework includes FT, a formal specification language for early requirements, and the T-Tool, a tool that supports the verification of FT specifications. The T-Tool is based on NUSMV, an open architecture for model checking. In our experience, the possibility of extending NUSMV with new functionality (e.g., a new input language, past operators, enhanced animator) has been crucial for its effective application to the analysis of FT specification. Finally, we tested the scalability of the approach through a number of experiments.

An important contribution of this work is to demonstrate that formal analysis techniques are useful during early development phases. The novelty of the approach lies in extending model checking techniques — which rely mostly on design-inspired specification languages — so that they can be used for early requirements modeling and analysis. Our results suggest that the approach is successful in identifying subtle bugs that are difficult to detect in an informal setting. Moreover, such bugs can be detected even when we consider examples with a small number of instances.

Formal analysis is often used to verify correctness of specifications, but it is usually applied in later phases. For instance, in [1, 14] formal verification techniques were used for the analysis of specifications expressed in the SCR formalism. In [7] NUSMV is used for the verification of RSML specifications. The works that are the most relevant to our work are Alcoa/Alloy [16, 15], KAOS [17], and the work on "Topoi Diagrams" [19]. *Alcoa* [16] is a tool for analyzing object models that describe the architectural or structural properties of system design. It has been used to verify various architectural frameworks, protocols, and schemes. The input language – *Alloy* [15] – is a notation based on Z, but has been tailored to fit object models and is amenable to automatic analysis. Similarly to the T-Tool, Alcoa uses SAT-based bounded model checking for assertion analysis (under-specify checking) and possibility analysis (over-specify checking). The main differences between Alcoa and the T-Tool is their focus on different applications (object vs requirements models). Moreover, the T-Tool supports a broader set of verification techniques, including BDD-based model checking and heuristics for reducing the model size for proving assertion properties. *KAOS* [17] is a framework that supports (early) requirements analysis. It shares with FT the goal- and agent-oriented flavor. Also the design of the temporal logic component in FT has been inspired by KAOS. The main difference between the two frameworks is in the analysis techniques used. The T-Tool supports automatic model checking verification techniques, while KAOS is based on interactive theorem proving techniques. *Topoi diagrams* [19] represent statements of gradual influence between variables (e.g., the more X, the more Y) and can be used in system requirements to describe how designers believe influence should propagate through a system. Topoi diagrams are related to *i** diagrams, where intentional links describe influences between the intentional elements of a domain. On top of the topoi diagrams, temporal logic formulas describing a property of the model can be checked. The focus of this approach is limited to formulas of a specific form that check whether a given input results in an expected output. Moreover, the framework in [19] is based on explicit state model checking techniques [9], rather than on symbolic techniques.

There are several directions for further research. First, we are investigating the use of techniques that guarantee that an FT specification is correct with no qualifications. We are also working on the refinement and the automation of the veri-

fication process. For example, we are developing heuristics for choosing the set of constraints considered while proving a property, and also heuristics for automatically alternating phases where the tool tries to prove the validity of a model, and phases where it tries to find bugs. We are also investigating optimizations of the model generator and advanced abstraction techniques that exploit, for instance, possible symmetries in the specification. Finally, we are planning to develop a graphical front end to the T-Tool, that will allow the user to write the FT specifications as annotations of an $i*$ model, and to see the scenarios produced by the T-Tool as animations of the $i*$ diagrams [21].

References

1. J. M. Atlee and J. Gannon. State-based model checking of event-driven systems requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.
2. M. Benedetti and A. Cimatti. Bounded Model Checking for Past LTL. In *Proceedings of the* 9th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2619 in Lecture Notes in Computer Science, pages 18–33, Warsaw, Poland, April 2003. Springer.
3. S. Berezin, S. Campos, and E. M. Clarke. Compositional reasoning in model checking. In *Proceedings of International Symposium on Compositionality (COMPOS'97)*, number 1536 in Lecture Notes in Computer Science, pages 81–102, Bad Malente, Germany, September 1998. Springer.
4. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the* 5th *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, March 1999. Springer.
5. J. Bowen and V. Stavridou. Safety critical systems, formal methods and standards. *IEEE/BCS Software Engineering Journal*, 8(4), July 1993.
6. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Survey*, 24(3):293–318, 1992.
7. Y. Choi and M. P. E. Heimdahl. Model checking RSML^{-e} requirements. In *Proceedings of the* 7th *IEEE International Symposium on High Assurance Systems Engineering*, pages 109–119, Tokyo, Japan, October 2002. IEEE Computer Society.
8. 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 *Proceedings of Computer Aided Verification Conference*, number 2404 in Lecture Notes in Computer Science, Copenhagen (DK), July 2002. Springer.
9. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
10. A. Fuxman. Formal Analysis of Early Requirements Specifications. Master's thesis, University of Toronto, 2001.
11. A. Fuxman, L. Liu, , M. Pistore, M. Roveri, and J. Mylopoulos. Specifying and analyzing early requirements in Tropos. In *Proceedings of the* 11th *IEEE International Requirements Engineering Conference*, Monterey Bay, California USA, September 2003. ACM-Press.
12. C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO, a logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 2(12):107–123, May 1990.
13. C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specification. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, 1996.
14. C. Heitmeyer, J. Kirby, and B. Labaw. The SCR method for formally specifying, verifying, and validating requirements: tool support. In *Proceedings of the* 19th *International Conference on Software Engineering*, pages 610–611. ACM Press, 1997.
15. D. Jackson. Alloy: a lightweight object modeling notation. *ACM Transaction on Software Engineering Methodology*, 11(2):256–290, 2002.
16. D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy Constraint Analyzer. In *Proceedings of the* 22nd *International Conference on on Software Engineering*, Limerik, June 2000. ACM Press.
17. E. Leiter. *Reasoning about Agents in Goal-oriented Requirements Engineering*. PhD thesis, Universite Catholique de Louvain, 2001.
18. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.
19. T. Menzies, J. Powell, and M. E. Houle. Fast formal analysis of requirements via "Topoi Diagrams". In *Proceedings of the* 23rd *International Conference on Software Engineering*, pages 391–400, Toronto, CA, May 2001. ACM Press.
20. A. Morzenti and P. San Pietro. Object-oriented logic specifications of time critical systems. *Transactions on Software Engineering and Methodologies*, 3(1):56–98, January 1994.
21. A. Perini, M. Pistore, M. Roveri, and A. Susi. Agent-oriented modeling by interleaving formal and informal specification. In *Proceedings of the* 4th *International Workshop on Agent-Oriented Software Engineering*, Lecture Notes in Computer Science, Melbourne, Australia, July 2003. Springer.
22. J. Spivey. *The Z Notation*. Prentice Hall, 1989.
23. E. Yu. Towards modeling and reasoning support for early requirements engineering. In *Proceedings of the IEEE International Symposium on Requirement Engineering*, pages 226–235. IEEE Computer Society, 1997.