klase

Formal Tropos

ITC-irst, Università di Trento, Università di Genova

Abstract. This deliverable described Formal Tropos, a framework that supports the formal verification of requirements specifications. Formal Tropos is based on the Tropos methodology, and extends it with formal notations and semantics, and with automated reasoning tools, in order to support the verification.

Document Identifier	Deliverable D3.1
Project	MIUR-FIRB project RBNE0195K5 "Knowledge Level Automated Software Engineering"
Version	v1.0
Date	October 31, 2006
State	Final
Distribution	Public

Acknowledgements.

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

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

Executive Summary

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 state-of-the-art NUSMV model checker.

We also investigate the possibility to apply on Formal Tropos advanced verification techniques, based on QBF-based Bounded Model Checking, which are better suited to exploit the specificities of the Formal Tropos language than the NUSMV model checker.

Finally, we report the results of the evaluate of the Formal Tropos 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.

Contents

Chapter 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 [Yu97]. In order to analyze such environment, it is necessary to investigate the objectives, business processes, and interdependencies of different stakeholders. At least in principles, the understanding of these "strategic" aspects of the operational environment is necessary to motivate and direct the development of the software system. 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 [BS93]. 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 actors, goals, needs...) and the constructs of formal specification languages such as Z [Spi89], SCR [HJL96], TRIO [GMM90, MP94].

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 document, we focus on the application of model checking techniques to early requirements specifications.

To allow for formal analysis, we have introduced a formal specification language called *Formal Tropos* (hereafter FT). The language offers all the primitive concepts of *i** [Yu97] (such as actors, goals, and dependencies among actors), but supplements them

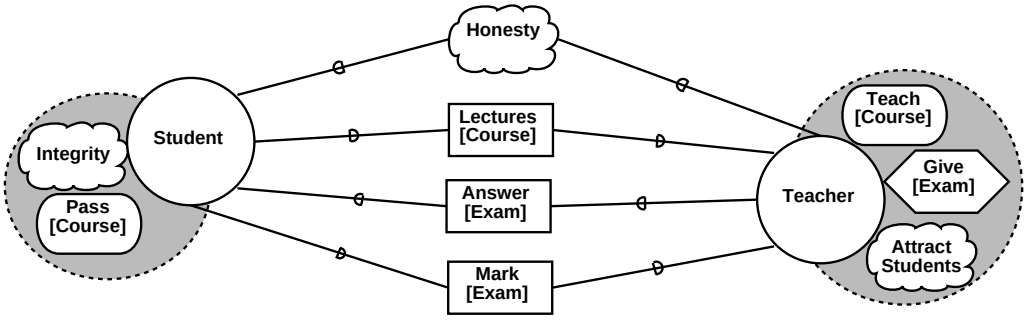

Figure 1.1: High-level *i** model of the course-exam management case study.

with a rich temporal specification language inspired by KAOS [DvLF93, DDMvL98]. The *i** notations allow for the description of the "structural" aspects of the early requirements model, for instance in terms of the network of relationships and dependencies among actors. FT permits to represent also the "dynamic" aspects of the model, describing for instance how the network of relationships evolves over time. In FT one can define the circumstances under which a given dependency among two actors arises, as well as the conditions that permit to consider the dependency fulfilled. In our experience, representing and analyzing these dynamic aspects allows for a more precise understanding of the early requirements model, and reveals gaps and inconsistencies that are by no means trivial to discover without the help of formal analysis tools.

In order to support the automated analysis of FT specifications, we have extended an existing formal verification technique, model checking [CGP99]. 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 [CCG+02] and on ad-hoc model checking verification techniques for Quantified Boolean Formulas (QBF) [GNP+06a]. 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 the specific language of the verification engines (NuSMV or QBF-bases), 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 have defined some heuristic techniques for rewriting an *i** diagram into a corresponding FT specification. The methodology also offers 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.

The document is structured as follows. Chapter 2 introduces a case study and shows how to build an FT specification from an *i** model. Chapter 3 uses the case study to illustrate how FT can be used for the incremental refinement of a specification. Chapter 4 describes the T-Tool, focusing on its functionalities, architecture, and usage guidelines. In Chapter 5 we report the results of a series of experiments that we conducted in order to evaluate the scope and scalability of the approach. Chapter 6 discusses related work,

and Chapter 7 draws conclusions and outlines future work. Finally, Chapter 8 described the evolution of the work described in this document along the duration of the KLASE project.

Chapter 2

From *i** to Formal Tropos

In this chapter we use a course-exam management case study to describe how an FT specification can be obtained. We use *i** models as starting point for our methodology, since they provide an informal graphical description of the organizational setting. This graphical description is then translated into a formal language that is more suitable for the analysis of the dynamic aspects of the operational setting.

2.1 Strategic modeling with *i**

The *i** modeling language [Yu97] 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 strategic aspects underlying the organizational setting within which the software system will eventually function. By understanding these strategic aspects one can better identify the motivations for the software system and the role that it will play inside the organizational setting. For instance, in order to develop a software system that supports the teacher in running and marking an exam, we need to understand those aspects of the interdependencies among teacher and students that define the process of giving exams.

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 1.1 depicts a high-level *i** diagram for the course-exam management case study, with its two main actors: the Student and the Teacher.[1]

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*

[1] A more complete early requirements model should include also other actors, like the teaching assistant and the secretariat. For presentation purposes, in this paper we concentrate only on the two main actors Student and Teacher.

(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 1.1. 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. Also the goal AttractStudents of the teacher is a softgoal, since there is not a precise number of students to be "attracted" in order to consider the goal fulfilled. A *task* (hexagon) specifies a particular course of action that produces a desired effect. In our example, the element Give[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 1.1 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 dependency from the student to the teacher. In the diagrams, *dependency* links (—▷—) are used to represent these inter-actor relationships.

Figure 2.1 zooms into one of the actors of this domain, 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 exam (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 (+). For instance, FairMarking[Exam] has a positive impact on Pass[Exam], since a fair marking makes it easier for the student to judge when she is ready to take the exam.

In Figure 2.1 we use some elements that are not present in the original *i** definition, but turn out to be useful in subsequent phases of our methodology. *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 2.2 completes Figure 2.1 with the inner description of the teacher. One can observe that new dependencies between the teacher and the student have been added with

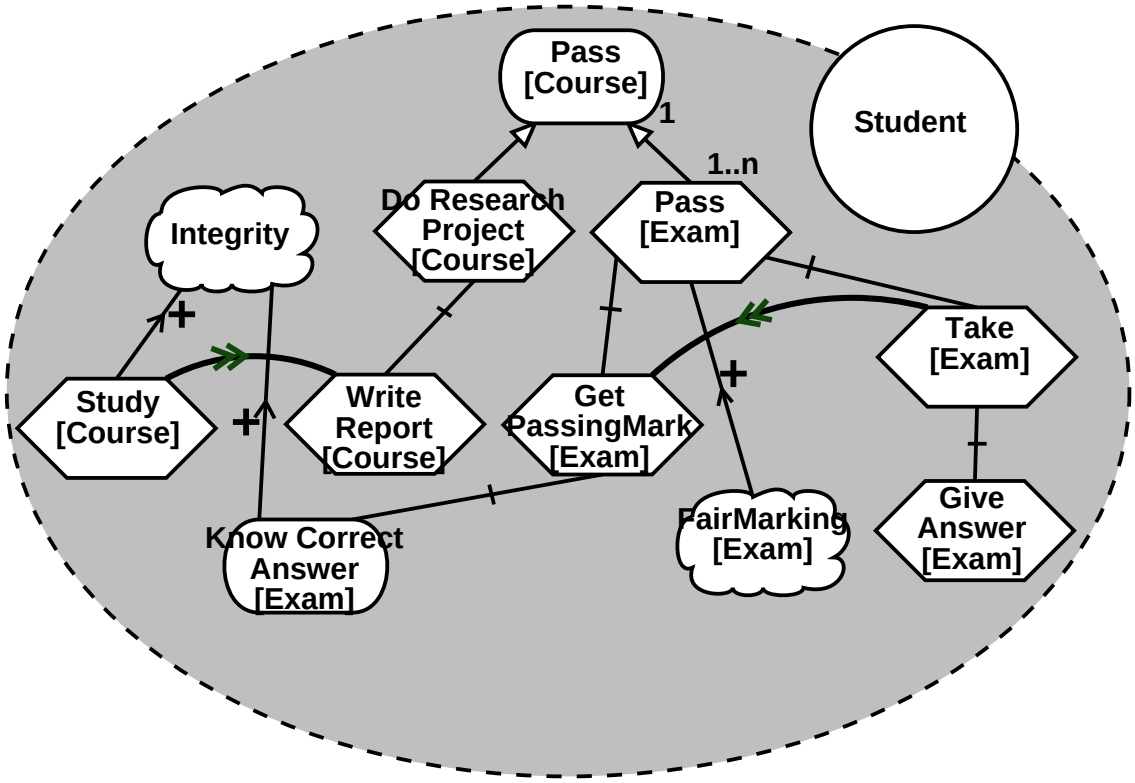

Figure 2.1: High-level *i** model focusing on the Student.

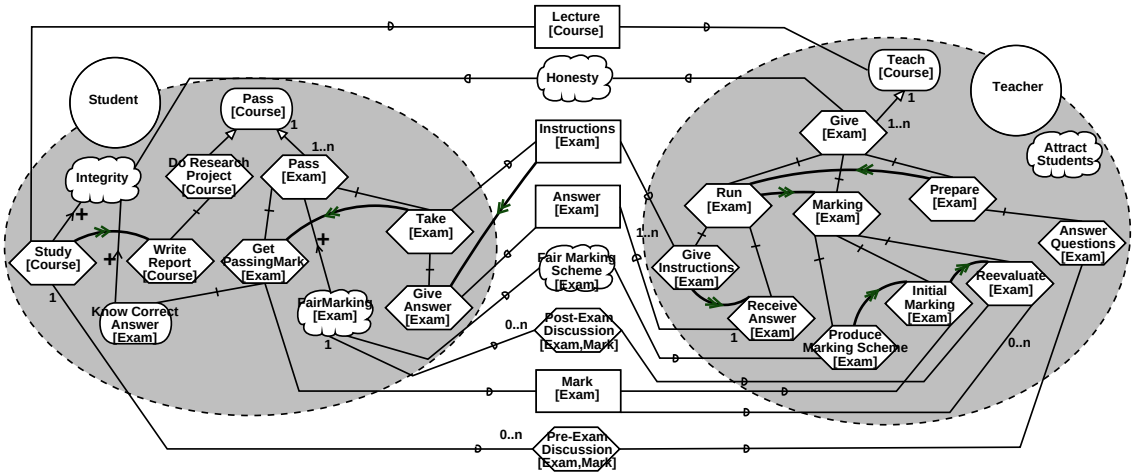

Figure 2.2: Annotated *i** model of the course-exam management case study.

respect to the high-level diagram of Figure 1.1.

2.2 Dynamic modeling with Formal Tropos

The *i** model of Figure 2.2 provides a static description of the course-exam management domain. In order to fully understand the domain, one needs to describe and analyze also the strategic aspects of its dynamics. For instance, the expectations and dependencies of a student that wants to pass an exam change over time, depending on whether she has still to take the exam, is waiting for the marking, or wants to discuss the marking with the teacher. Moreover, if a student has still to take the exam, she may possibly decide to write a report instead, but this is very unlikely to happen if the student has already got a passing mark.

The prior-to links and cardinality constraints shown in Figure 2.2 permit to describe some aspects of the dynamics of the course-exam management case study, but their expressive power is very limited. In general, informal notations like the ones provided by *i** are inadequate to carry out an accurate dynamic analysis. Formal specification languages are more suited for this purpose.

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 dynamic 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?

A precise definition of FT and of its semantics can he found in [Fux01, FT003]. Here we present the most relevant aspects of the language. The grammar of FT is given in Figure 2.3. An FT specification describes the relevant objects of a domain and the relationships among them. The description of each object is structured in two layers. The outer layer is similar to a class declaration and defines the structure of the instances together with their attributes. The inner layer expresses constraints on the lifetime of the objects, 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 2.4 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 declarations 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 each element may exist during the evolution of the system. For example, different Pass[Course] goals may exist for different Student instances, or for different courses taken by the same student. For this reason, we refer to the different elements that compose an FT declarations as "classes".

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 the GetPassingMark[Exam] goal of the student that motivates the expectation of having a mark (attribute gpm).

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. This initialization happens when an instance of the class the attribute belongs to is created, or, in the case of attributes that are both **optional** and **constant**, at any time after the creation of the instance. 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.

/ 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*

Figure 2.3: The Formal Tropos grammar.

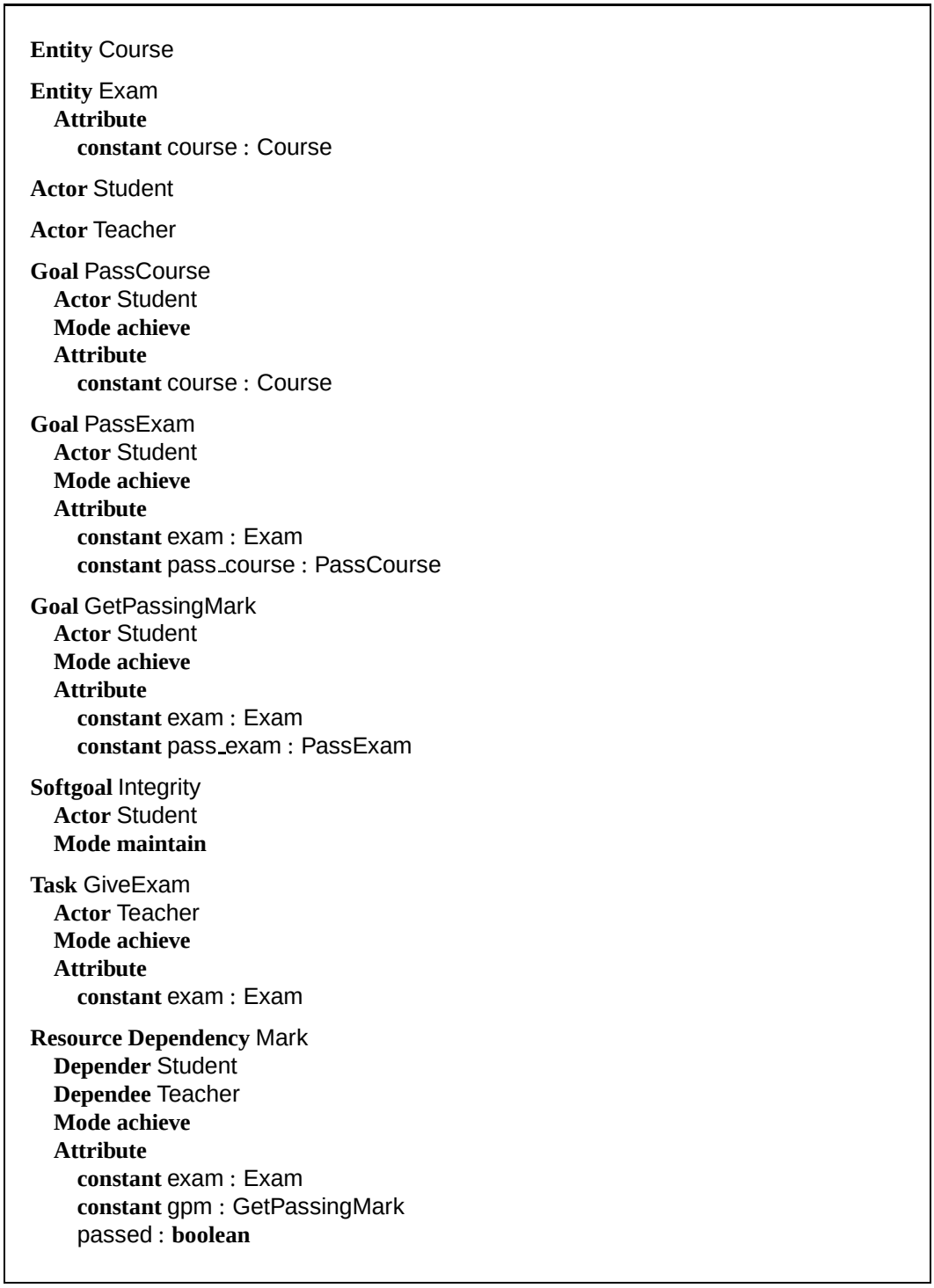

Figure 2.4: Excerpt of outer layer of an FT class declaration.

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**)))

Figure 2.5: Example of FT constraints.

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.

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 the student's goal Pass[Exam] is **achieve**, which means that the student expects to reach a state where the exam has been passed. The student's 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 2.5 presents an excerpt of con-

straints on the lifetime of dependency Mark of the course-exam management case study. The actual constraints are described with formulas given in a typed first-order linear-time temporal logic (see Section 2.2.3). In FT we distinguish different kinds of constraints (namely, **Invariant**, **Creation**, and **Fulfillment** constraints), depending on their scope.

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 2.5 states a relationship between attributes of the instances of the Mark class. The second invariant imposes a cardinality constraint for Mark instances, 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, including actors and entities. 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**).

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 2.5, 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 a typed first-order linear-time temporal logic. Linear-time temporal logic is quite common in formal specification and verification frameworks. The temporal operators provided by this logic have shown to capture

the relevant temporal properties of dynamic systems. In FT the temporal operators are complemented with quantifiers that take into account the other aspect of the dynamics of FT models, namely, the presence of multiple instances of classes. It is hence possible to represent in FT most, if not all, dynamic aspects of a model that one may want to express.

The FT temporal logic is described by the following syntax:

$$
\begin{array}{rll}
f ::= & f \wedge f \mid f \vee f \mid \neg f \mid t & \text{(boolean op.)} \\
\mid & t = t \mid t < t \mid t \leq t & \text{(relational op.)} \\
\mid & \forall x : \textit{sort}.f \mid \exists x : \textit{sort}.f & \text{(quantifier)} \\
\mid & \mathbf{X}f \mid \mathbf{F}f \mid \mathbf{G}f \mid f\mathbf{U}f & \text{(future op.)} \\
\mid & \mathbf{Y}f \mid \mathbf{O}f \mid \mathbf{H}f \mid f\mathbf{S}f & \text{(past op.)} \\
\mid & \textbf{JustCreated}(t) \mid \textbf{Changed}(t) & \\
\mid & \textbf{Fulfilled}(t) \mid \textbf{JustFulfilled}(t) & \text{(special pred.)} \\
 & & \\
t ::= & c \mid x \mid t.a & \text{(const. and var.)} \\
\mid & \textbf{self} \mid \textbf{actor} \mid \textbf{depender} \mid \textbf{dependee} & \text{(special term)}
\end{array}
$$

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 $\mathbf{X}f$ (next f) expresses the fact that formula f should hold in the next state reached by the model, while formula $\mathbf{Y}f$ (previous f) requires condition f to hold in the previous state. Formula $\mathbf{F}f$ (eventually f) requires that formula f is either true now or that it becomes eventually true in some future state; formula $\mathbf{O}f$ (once f) expresses the same requirement, but on the past states. Formula $\mathbf{G}f$ (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 $\mathbf{H}f$ (always in the past g) holds if f is true in the current state and in all past state of the model. Formula $f_1\mathbf{U}f_2$ (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 $f_1\mathbf{S}f_2$ (f_1 since f_2) holds if there is some past state where f_2 holds and formula f_1 holds since that state.

Special predicates can appear in temporal logic formulas. Predicate **JustCreated**(t) holds if element t exists in this state but not in the previous one. Predicate **Changed**(t) holds in a state if the value of term t has changed with respect to the previous state. Predicate **Fulfilled**(t) holds if t has been fulfilled, while predicate **JustFulfilled**(t) holds if **Fulfilled**(t) holds in this state, but not in the previous one. The two latter predicates are defined only for intentional elements and dependencies.

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 2.5).

While the temporal logic incorporated in FT is quite expressive, 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. Consider, e.g., the creation condition and the fulfillment condition of dependency Mark in Figure 2.5. No temporal operators are needed is these constraints, since the kinds of the constraints already define the lifetime instants the conditions refer to. Only when the lifetime events and the modalities are not sufficient for capturing all the temporal aspects of a conditions, temporal operators need to appear explicitly in the formulas. For instance, the temporal operator that appears explicitly in the **Fulfillment** trigger of dependency Mark 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 2.5 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 2.6 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. Properties 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.

2.3 From *i** to FT: Translation guidelines

Developing a satisfactory formal specification for a software system 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 to produce a "reasonable" initial FT model. In fact, most of the constraints of an FT specification already appear implicitly in the *i** model.

We have identified a set of translation rules that permit to systematically derive these constraints. These rules capture the intuitive semantics that we use when designing an *i** model. For instance, decomposition and means-ends links describe possible ways to

Goal PassExam
/ A1: A student can only pass an exam once. */*
Creation assertion condition
$\forall$ p : PassExam (p.**actor** = **actor** $\wedge$ p.exam = exam $\wedge$ p.pass_course = pass_course $\rightarrow$ p = **self**)

Resource dependency Mark
/ A2: For each mark there was an answer corresponding to it. */*
Fulfillment assertion condition
$\exists$ a : Answer (a.**dependee** = **depender** $\wedge$ a.**depender** = **dependee** $\wedge$ a.exam = exam $\wedge$ **Fulfilled** (a))

Resource dependency Mark
/ A3: A mark can only be changed if there is a petition. */*
Fulfillment assertion condition
$\neg$passed $\wedge$ **F** passed $\rightarrow$ **F** $\exists$ ped : PostExamDiscussion (ped.mark = **self**)

Global
/ A4: If the student wants to maintain her integrity, she cannot pass an exam without studying. */*
Assertion
$\forall$ h : Honesty (**Fulfilled** (h) $\rightarrow$ $\forall$ k : KnowCorrectAnswer ((k.**actor** = h.**dependee**) $\wedge$
(k.exam = h.give_exam.exam $\wedge$ **Fulfilled** (k) $\rightarrow$ $\exists$ s : Study ((s.**actor** = k.**actor**) $\wedge$
(s.course = k.exam.course) $\wedge$ **Fulfilled** (study)))))

Global
/ P1: It is possible for a student to pass a course. */*
Possibility
$\exists$ pc : PassCourse (**Fulfilled** (pc))

Global
/ P2: It is possible that a student passed a course without passing the exam. */*
Possibility
$\exists$ pc : PassCourse (**Fulfilled** (pc) $\wedge$ $\neg\exists$ pe : PassExam ((pe.pass_course = p) $\wedge$ **Fulfilled** (pe))

Goal PassExam
/ P3: It is possible that a student passed an exam, but still thinks that the marking is not fair. */*
Fulfillment possibility condition
$\exists$ f : FairMarking (f.pass_exam = **self** $\wedge$ **Fulfilled** (f) $\wedge$ $\neg$f.satisfied)

Global
/ P4: It is possible that a teacher expects an exam answer from a student not committed to the exam. */*
Possibility
$\exists$ a : Answer (**G** $\neg\exists$ p : PassExam (p.exam = a.exam $\wedge$ p.**actor** = a.**dependee**))

Figure 2.6: Example of Formal Tropos properties.

achieving a parent goal in terms of lower-level sub-goals. Therefore, sub-goals are created only with the purpose of fulfilling the parent goal. This leads to the following two translation rules for the creation and fulfillment conditions of the sub-goals:

- The default creation condition of a sub-goal of a decomposition or means-ends link is that the parent goal exists, but has not been fulfilled yet.
- The fulfillment condition of a parent goal depends on the fulfillment of the sub-goals. If the sub-goals are connected to the parent goal with *means-ends* links, then the fulfillment of *at least one* of the subgoals 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. For instance, Pass[Exam] and Take[Exam] refer to the same exam, and the Student that wants to pass the exam is the same that has to take it. This leads to the following translation rule:

- An invariant condition is added to the sub-goal in order to force the binding between the owners of the two goals and between those attributes that are present in both goals.

Other rules capture the semantics of cardinality constraints. For instance, the case of a one-to-one connection along a refinement or means-ends link is handled by the following translation rule:

- When a sub-goal is connected to the parent goal with a one-to-one refinement or means-ends link, then an invariant is added to the sub-goal, requiring that at most one instance of the sub-goal is associated to each instance of the parent goal.

As a last example, we interpret a prior-to link as a temporal constraint that requires to fully complete the former goal before moving to the latter one. This leads to the following translation rule:

- When there is a prior-to constraint between two sub-goals with a common parent, a creation condition is added to the goal that comes later, requiring that the former 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 initial model. In the case of dependency Mark (see Figure 2.5), the invariants, the creation condition, and the fulfillment condition formalize the goal delegation and the cardinality constraints that appear in the *i** mode. Therefore they are automatically generated using the rules. In order to capture the nature of the application domain, the generated constraints need to be corrected, and additional

non-standard constraints need to be manually added to the FT specification. For instance, the last constraint for dependency Mark in Figure 2.5 is non-standard. It 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.

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.

Chapter 3

Formal Tropos at work

We now illustrate the usage of FT in refining an early requirements specification. In the next chapter we describe the T-Tool, a tool that supports the analysis performed in this chapter. For explanatory purposes we focus on a subset of the FT specification. This initial model, shown in Figure 3.1, is strongly under-specified. In particular, it does not cover at all the dynamic aspects of the domain. We will interactively improve and refine it, guided by results provided by the analysis.

In this chapter, scenarios are represented by diagrams like the one in Figure 3.2, 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 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.

Goal PassCourse
  Fulfillment definition
    $\forall$ e : Exam (e.course $=$ course $\rightarrow$
      $\exists$ p : PassExam (p.exam $=$ e $\wedge$
        p.pc $=$ **self** $\wedge$ **Fulfilled** (p)))

Task PassExam
  Mode achieve
  Actor Student
  Attribute constant pc : PassCourse

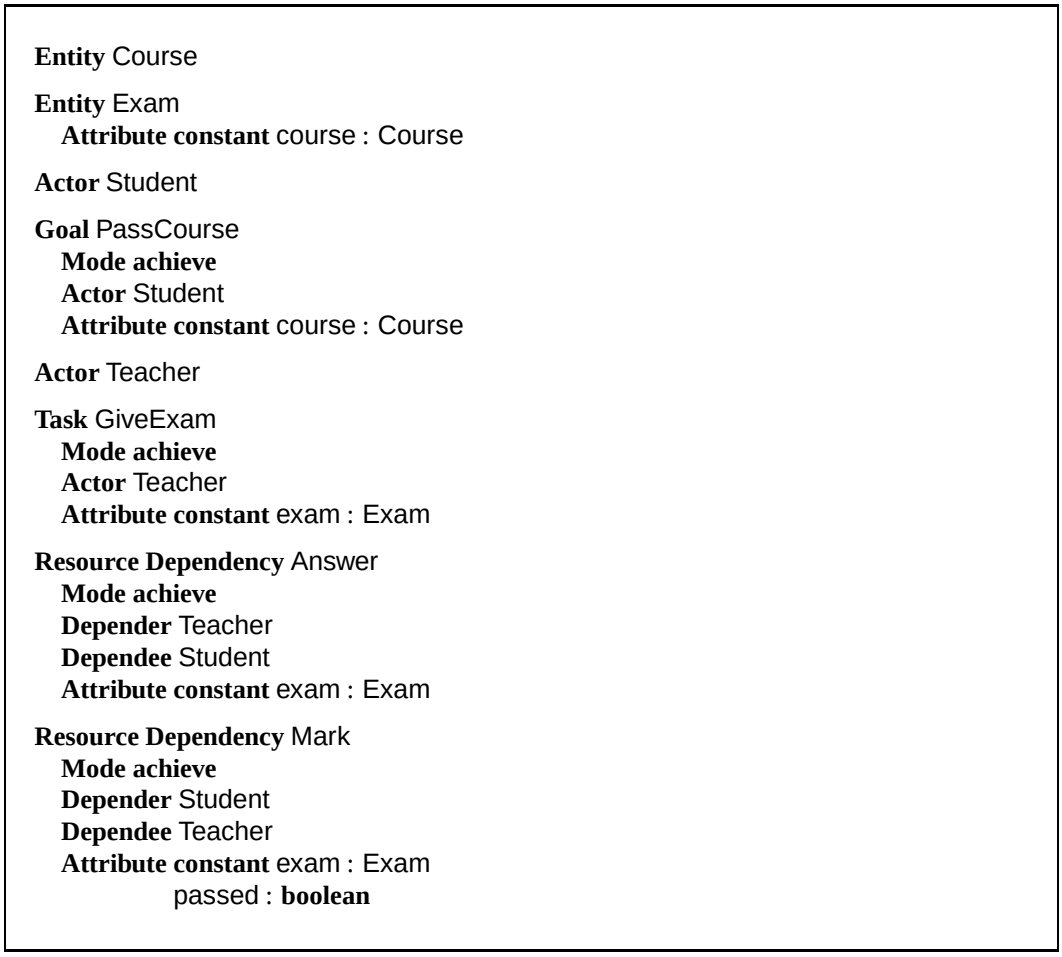

Figure 3.1: The initial FT specification for the analysis of the case study.

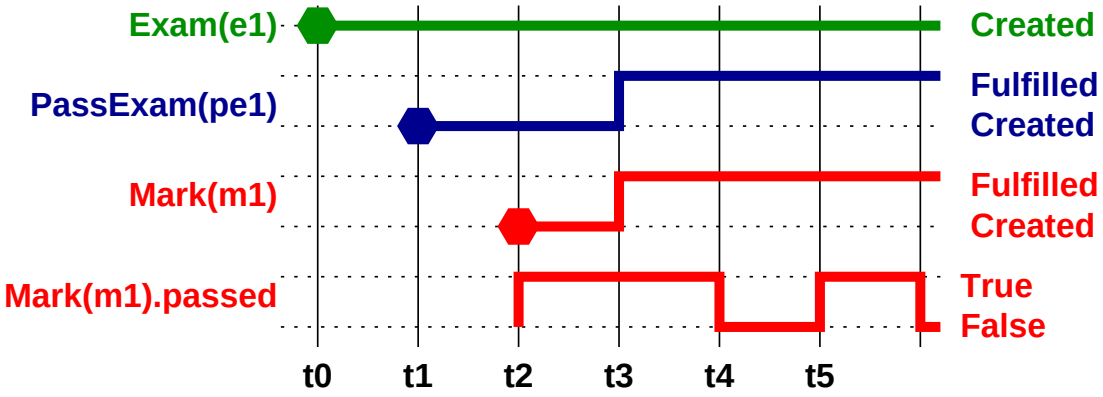

Figure 3.2: A scenario where attribute passed oscillates.

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)

The analysis of the extended specification reveals some problems. The specification allows for unrealistic scenarios such as the one depicted in Figure 3.2. 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 3.2. 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)

[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.

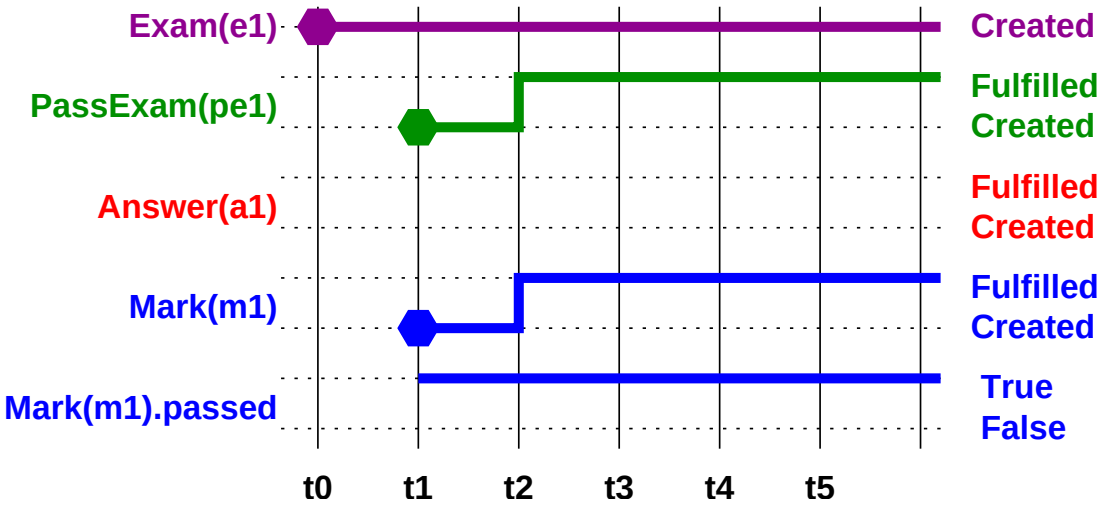

Figure 3.3: A student receives a mark for an exam without providing an answer.

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.

Task GiveExam
  Creation condition
    $\exists$ pe : PassExam (pe.exam = exam)
  Fulfillment condition
    $\forall$ a : Answer ((a.exam = exam $\wedge$ a.**depender** = **actor**) $\rightarrow$
      $\exists$ m : Mark (m.exam = exam $\wedge$ m.**dependee** = **actor** $\wedge$
        m.**depender** = a.**dependee** $\wedge$ **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 3.3, instances of PassExam and Mark are created at time t_1 and fulfilled at time t_2, while no instance of Answer is ever created. These behaviors can be easily ruled out by adding the following creation constraint to dependency Mark.

Resource Dependency Mark
  Creation
    condition for domain
      $\exists$ a : Answer (a.**depender** = **dependee** $\wedge$
        a.**dependee** = **depender** $\wedge$
          a.exam = exam $\wedge$ **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.

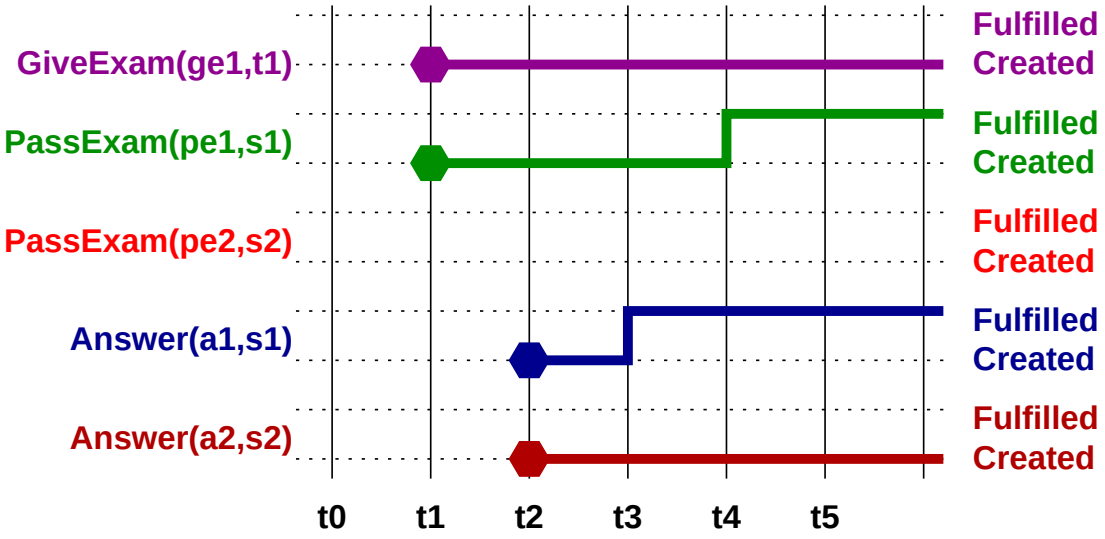

Figure 3.4: The teacher waits for an answer that will never arrive.

Global assertion
$\forall$ a : Answer (**F** $\exists$ pe : PassExam (pe.exam $=$ a.exam $\wedge$
pe.**actor** $=$ a.**dependee**))

Unfortunately, the scenario depicted in Figure 3.4 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 on 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
$\exists$ p : Passexam (p.**actor** $=$ **dependee** $\wedge$ p.**exam** $=$ **exam**)

As the specification grows, it is important to detect over-constrained 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
$\exists$ p : PassCourse (**Fulfilled** (p))

An existence proof for this possibility is shown in Figure 3.5. The class instance Pass-

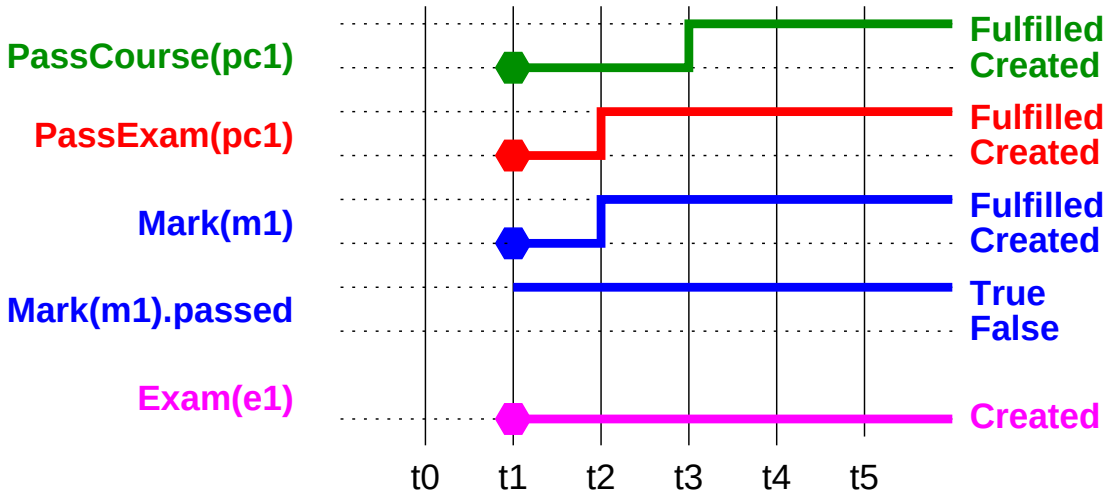

Figure 3.5: A student passes a course.

Course(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
$\exists$ s : Student ($\exists$ m : Mark ($\exists$ p : PassCourse : (
m.**depender** $=$ s $\wedge$ **Fulfilled** (m) $\wedge$ $\neg$ m.passed $\wedge$
p.course $=$ m.exam.course $\wedge$ p.**actor** $=$ s $\wedge$ **F Fulfilled** (p))))

We remark that this possibility is false if we allow only one instance per class. In this case, there is no possibility to obtain a second mark for the exam. We have to consider multiple class instances to satisfy this possibility.

Chapter 4

The T-Tool

In this chapter we describe the T-Tool, a tool that supports the analysis described in the previous chapter. The T-Tool is available at the URL `http://dit.unitn.it/~ft/`.

The T-Tool is based on finite-state model checking [CGP99]. The advantages of model checking with respect to other formal techniques (e.g., theorem proving; see [HV91] for a comparison) are that it allows for an automatic verification of a specification and that (counter-)example traces are produced as witnesses of the validity (or invalidity) of the specification. A limit of finite-state model checking is that it requires a model with a finite number of states. This forces to define an upper bound to the number of class instances that can be created during model checking.

The T-Tool input is an FT specification along with parameters that specify the upper bounds for the class instances. 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.

All the verification functionalities provided by the T-Tool are based on standard model checking concepts and algorithms. However, their application to FT has required a substantial effort of customization and the development of some extensions. We will report on these extensions in Section 4.2, after we have described the T-Tool functionalities in more detail.

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 verifies 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. Not all the consistency checks may be relevant for a given model. For instance, in a model it may be perfectly reasonable that there is no single scenario where instances are generated for all classes. In this case, this consistency check is excluded for the model under investigation.

4.1.3 Possibility checks

Possibility checks verify whether the specification is over-constrained, 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 under-specified 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 which 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 4.1). In the first step, the FT specification is translated into an Intermediate Language (IL) specification. In the second step, the IL specification is given as input to the verification engines, which are built on top of the NuSMV model checker [CCG+02] and on top of model checking approaches based on "Quantified Boolean Formulas" [GNP+06a].

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. In particular, the strategic flavor of FT is lost, and the focus shifts to the dynamic aspects of the system. Some details of the FT specification are removed during the translation. This is the case, for instance, for the distinction among the different types of intentional elements and dependencies. While these

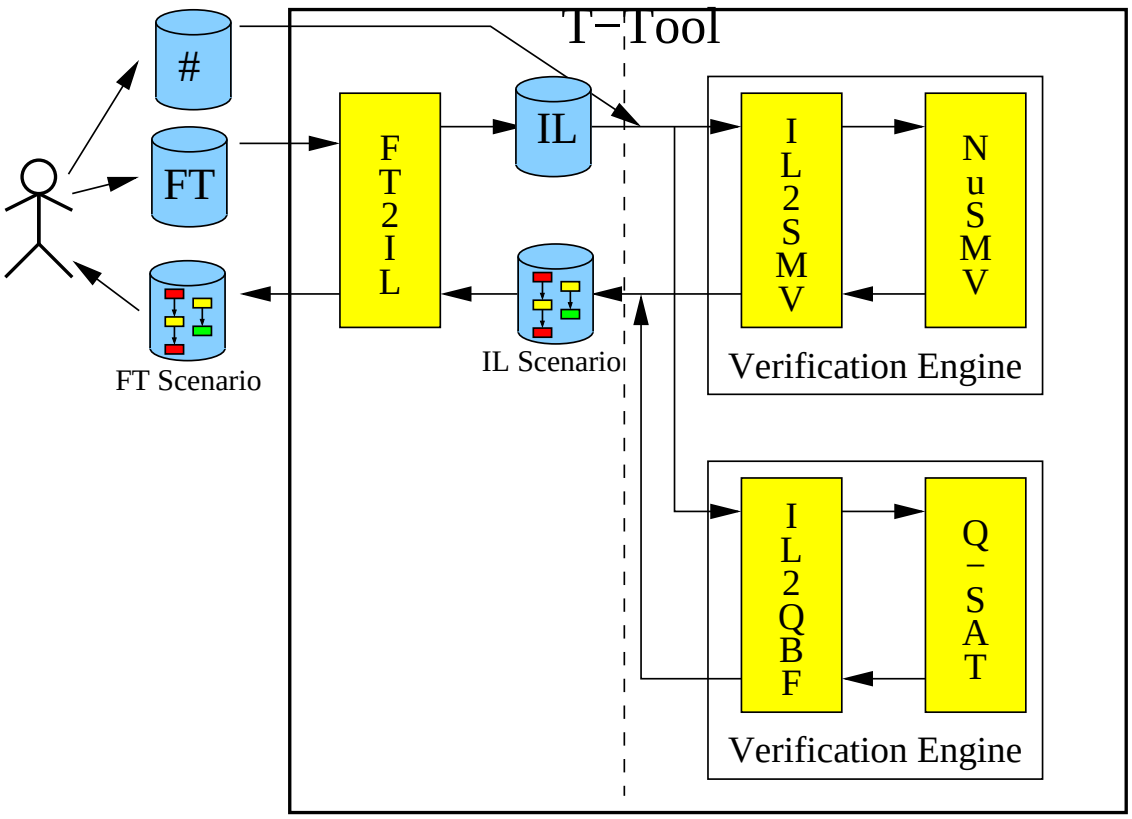

Figure 4.1: The T-Tool framework.

```
/* 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
```

Figure 4.2: Excerpt of the IL translation for our running example.

aspects are important in the overall description and specification of the organizational setting, they do not play any role in the formal analysis.

In Figures 4.2 and 4.3, 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,

```
                /* 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))
```

Figure 4.3: Excerpt of the IL translation for our running example (cont.)

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 4.3, 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 4.3 express respectively the fact that attribute course of all instances of Exam and attribute actor of all instances of PassExam 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 4.3 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 creation and fulfillment of class instances 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

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

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 2.5 which results in the sixth constraint of Figure 4.3. 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

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

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 4.3).

4.2.2 The role of IL

The IL plays a fundamental role in bridging the gap between FT and formal methods.

A first advantage is that IL is much more compact than FT, and therefore allows for a much simpler formal semantics. In fact, in [Fux01, FT003] the formal semantics of FT is defined on the top of the semantics of IL, via the translation rules that map an FT specification into an IL specification. 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.

The IL also allows for a clear definition of the verification of assertions and possibilities on a given specification. Let C_i with $i \in I$ be the set of constraints of an IL specification. Checking if assertion A is valid corresponds to checking whether the implication $\bigwedge_{i \in I} C_i \Rightarrow A$ is valid in the semantic model, i.e., if all scenarios that satisfy the constraints also satisfy the assertion. Checking if possibility P is valid corresponds to checking whether the formula $\bigwedge_{i \in I} C_i \wedge P$ is satisfiable, i.e., if there are some scenarios that satisfy the constraints and the possibility.

Another advantage is that the IL, while more suitable to formal analysis, is still independent of 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. Moreover, even if we restrict to model checking, it is easy to exploit different tools by defining suitable translations of IL into the languages of these tools (see Figure 4.1).

Finally, IL 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 [DvLF93, DDMvL98].

4.2.3 The model checking verification engines: NuSMV

The T-Tool allows for two different engines for performing the actual verification: the NuSMV model checker [CCG+02], described in this section, and of satisfiability procedures for Quantified Boolean Formulas [GNP+06a], described in Section 4.2.4.

NuSMV implements several state-of-the-art model checking algorithms. It also provides an open architecture that facilitates the implementation of new algorithms and the

customization of the verification process to the specific application domain.

NUSMV is 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 [CGP99, McM93]. NUSMV adopts symbolic model checking algorithms based on Binary Decision Diagrams (BDD) [Bry92] and on propositional satisfiability (SAT) [BCCZ99]. 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) [BCCZ99]. The T-Tool exploits both BDD-based and SAT-based model checking.

We have implemented several extensions to the NUSMV model checker in order to allow for the verification of IL specifications. In particular, 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 4.1), 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 linear-time temporal logic 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.

Another extension to the NUSMV model checker is a new more flexible interactive animator, that allows both for an interactive exploration of the automaton, and for a random execution of a certain number of steps. Finally, to allow for the verification of **ASSERTION** and **POSSIBILITY** formulas against the executions of the automaton, the BMC engine has been extended with past operators [BC03].

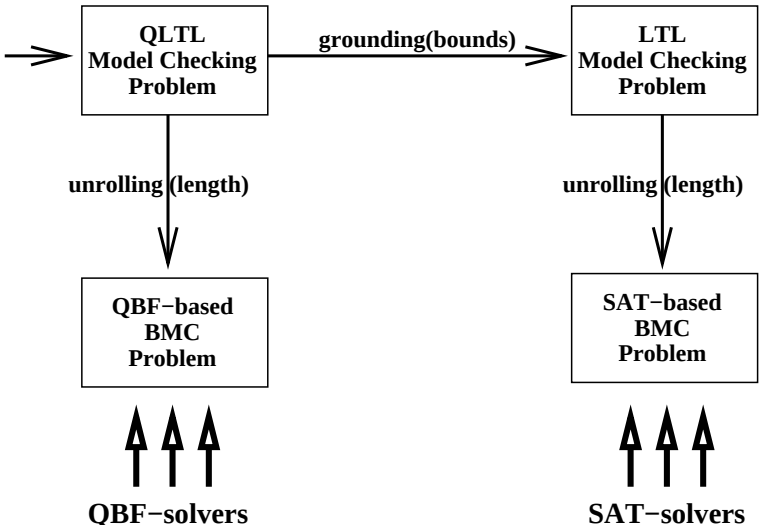

Figure 4.4: SAT-based versus QBF-based bounded model checking.

4.2.4 The model checking verification engines: QBF-based model checking

If applied to the verification of FT specifications, the SAT-based approach described above has a bottleneck, namely it requires to transform the first-order Linear Time Formulas of IL into a non-quantified formula suitable to be verified by NUSMV. A *grounding* procedure is performed in IL2SMV which removes all the quantifiers by replacing them with a suitable conjunction or disjunction on all the possible instances of the quantified variables. In many cases, however, the formula that results from the grounding is very large. Moreover, the "structure" of the original formula, which is given by the quantifiers is lost in the grounded version, and which could be used for more efficient satisfiability check algorithms, is lost in the grounding.

The approach just described, which consists in applying a grounding step and then a unrolling step to the *Quantified* LTL formula, and then to exploit SAT-solvers on the resulting problem, is illustrated in the top/right-side of Figure 4.4

The left-side part of the figure illustrates an alternative approach, which is based on QBF-based Bounded Model Checking [GNP+06a]. Similarly to SAT-based BMC, also in this case we look for a counter-example trace of fixed length, and hence we perform an unrolling on the temporal formula to be verified. In this case, however, quantifiers are allowed in the specification of the unrolled property, which is hence modeled as a Quantified Boolean Formula (QBF). This way, the grounding step is not necessary anymore, more compact formulas are obtained and, more important, QBF-solvers such as the one described in [GNT04] can be exploited to the verification. The reader interested in the details of QBF-based Bounded Model Checking can refer to [GNP+06a].

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, and there can be 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 respect to 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 and QBF-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 and QBF-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 needed, 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.[1] We have seen that, when we check an assertion A on a specification consisting of constraints C_i with $i \in I$, we are

[1] This is an example of "abstraction" technique, since the verification is done on a more general specification that is obtained by removing irrelevant details. Abstraction techniques are common practice in the model checking community, see for instance [BCC98].

looking for solutions to the following problem: $\bigwedge_{i \in I} C_i \Rightarrow A$. 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 A, if we prove that A holds in an under-constrained system, we can infer that A also holds 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. For the moment, the intervention of the user is needed for choosing the initial set of constraints and for adding new constraints. In most practical cases, this approach is successful, since 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 are currently implementing and testing different heuristics for supporting the user in the selection of the relevant constraints.

Chapter 5

Experimental results

Following the guidelines described in the previous chapters, 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 2.6. More results, on an extended set of properties and assertions, can be found at the following URL: `http://dit.unitn.it/~ft/cs/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 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

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 5.1: 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 5.2: Results for assertion checks.

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 5.1 and Table 5.2. 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.[1] The experiments show that:

1. Possibilities P1-3 are valid, and witness scenarios of length 3, 3 and 4 are produced

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

Test	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_10
SAT	48.8	53.8	42.6	62.3	FAIL	62.3	54.4	48.8	FAIL	61.5
QBF	12.5	21.7	10.3	12.5	30.9	17.3	14.2	13.4	51.5	12.9

Table 5.3: Comparison of SAT-based and QBF-based bounded model checking.

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.

The results discussed above exploit NUSMV as verification engine. We conclude the section with some results on the comparison of the two different approaches to Bounded Model Checking, namely, the SAT-based approach provided by NUSMV, and the QBF-based approach described in Section 4.2.4. The comparison, reported in Table 5.3, considers 10 different properties, extracted from FT specifications, and reports the time required to verify the satisfiability of the formula both exploiting the SAT-based approach and the QBF-based approach. The results clearly show the advantage of QBF in terms of time required for the verification.

5.3 Discussion

5.3.1 Effectiveness

For our case study, the proposed approach was effective in improving the early requirements specification. The discussion of the dynamic aspects of the specification led to a

better understanding of the domain and revealed several tricky aspects of the case study that were not evident in the *i** model.

All validation techniques provided by the T-Tool have been useful in detecting and correcting bugs. For instance, animation revealed that, 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.

The T-Tool's ability to generate (counter-)examples helped in pinpointing the problems and in refining the FT specification. Sometimes the (counter-)example generated by T-Tool is a trivial or irrelevant scenario that is not informative for the user. An example is given by the assertion claiming that it is never the case that a teacher is waiting for an answer from a student that is not committed to take the exam (see Chapter 3). The T-Tool generates a trivial counter-example that violates the assertion since an insufficient number of answers are created. Also in these cases, it is easy to refine the property so that the trivial counter-examples are ruled out. For instance, if we impose that an instance of the class Answer exists for each student, then the T-Tool produces a much more informative counter-example, namely the one depicted in Figure 3.4.

A limiting factor of the current framework is that correctness of the specification can be asserted only up to the considered upper bounds of the number of class instances. We are currently looking to infinite state model checking techniques, which guarantee that if the model is correct for sufficiently high upper bounds, then it is correct regardless of the upper bounds. We are also looking to verification techniques (e.g., theorem proving) that do not require a finite-state model and that can hence verify directly the correctness of the unbounded model.

To conclude, we comment on the scalability of the approach. The operational framework of the course-exam management case study is tiny if compared to real domains. However, when our approach is used for larger domains, we expect to apply the formal analysis techniques only on selected "critical" parts of the *i** model. We claim that the size of the course-exam management model that we considered in our experiments is comparable to the sizes of the formal models for these critical subsets.

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.

Chapter 6

Related work

There are several proposals in the literature that are related to our work. However, to the best of our knowledge, all of them differ from our approach either in the target domain (early requirements in our case) or in the support for automated analysis (model checking techniques in our case).

Among the huge number of tools and techniques that place emphasis on automated analysis, the most relevant to our work is Alcoa [JSS00]. Several aspects and design choices of the T-Tool and Alcoa are similar. Also Alcoa offers a user-friendly front-end language, Alloy, which is translated to a smaller, formally defined, intermediate language. As FT, Alloy has been "built with analysis in mind" [Jac02], i.e., it was designed in such a way that it becames amenable to automated verification. Similarly to the T-Tool, Alcoa relies on the idea of bounding the number of instances of the specification in order to verify a finite model of the system. It also promotes a methodology based on automatically-generated scenarios. It supports the same forms of analysis as FT, namely consistency, possibility and assertion checks, and animation, even if the temporal logic of Alcoa is less expressive than the one of FT. Finally, Alcoa is based on SAT-based bounded model checking, which is one of the two formal analysis techniques supported by FT. The fundamental difference between FT and Alcoa is their target domain. Alcoa is a tool for describing structural properties, and it is not suitable for early requirements specification. The main reason is that it "is not for describing dynamic interactions between objects" [Jac02], and it is thus unable to model and analyze the dynamic aspects of actors and dependencies in an organizational setting. Furthermore, while FT is inspired by *i**, the starting point of Alcoa's front-end language is Z [Spi89], which does not offer notations adequate to reason on the strategic aspects of an organizational setting typical of early requirements. Similar considerations hold for other methodologies that apply automated verification to requirements specifications (e.g., SCR [HKL97], RSML [CH02]). They are able to describe dynamic systems, but their input languages are rather low-level and too operational. Not suprisingly, such tools have been applied to the specification of embedded systems and process control, but never to the analysis of organizational settings.

Among the several proposals that concentrate on the verification of early requirements, the closest to our approach is KAOS [DvLF93, DDMvL98]. As FT, it offers goal- and agent-oriented constructs, and it has a formal semantics. However, it relies on fundamentally different analysis techniques. In particular, it advocates theorem proving, as opposed to model checking. Thus, it is unable to provide animation of the models and concrete scenarios that can direct the user to iteratively refine the specification.

Chapter 7

Concluding remarks

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, and on ad-hoc QBF-based verification algorithms. 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. On the other hand, the ad-hoc QBF-based verification approach has shown to be very convenient for some of the verification tasks (in particular, verification of possibility properties) in terms of performance.

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.

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 looking both to theorem proving techniques, which do not require a finite-state model, and to infinite-state model checking techniques, which guarantee that if the model is correct for sufficiently high bounds then it is correct also with no bounds. We are also working on the refinement and the automation of the verification 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 [PPRS03].

Chapter 8

History of the Deliverable

In this chapter a short description of the history of the deliverable along the four years of the KLASE project.

8.1 1st year

During the first year of the KLASE project, the main effort has been devoted to the refinement of the definition of the FT language, to the formalization of its semantics, to the completion of the implementation of the T-Tool, and to the experimental evaluation of its applicability. The most important outcome of the 1st year activities has been the distribution, in Open Source of the T-Tool (see `http://dit.unitn.it/~ft`). The activity of the first year has also produced papers [FL+03, PPRS03, FLM+04] and technical reports documenting FT and the T-Tool (these technical reports are available from `http://dit.unitn.it/~ft`).

8.2 2nd and 3rd year

In the second and third years, the research work has focused on the possibility to adapt and extend the FT language in order to allow a more efficient description of requirements in Web Service composition scenarios. The outcomes of this research work are reported in [D13] and [D32].

8.3 4th year

In the last year of the project, the verification techniques implemented by the T-Tool have been extended in order to support the QBF-based model checking approach defined by

some of the partners of the project [GNP+06a]. This extension, which is generic and can be applied to any domain specified in FT, is particularly relevant in the case of verification of requirements for Web Service compositions [GNP+06b].

Bibliography

[BC03] M. Benedetti and A. Cimatti. Bounded Model Checking for Past LTL. In *Proceedings of the 9^{th} 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.

[BCC98] 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.

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

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Survey*, 24(3):293–318, 1992.

[BS93] J. Bowen and V. Stavridou. Safety critical systems, formal methods and standards. *IEEE/BCS Software Engineering Journal*, 8(4), July 1993.

[CCG+02] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of Computer Aided Verification Conference*, number 2404 in Lecture Notes in Computer Science, Copenhagen (DK), July 2002. Springer.

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

[CH02] Y. Choi and M. P. E. Heimdahl. Model checking RSML^{-e} requirements. In *Proceedings of the 7^{th} IEEE International Symposium on High Assurance Systems Engineering*, pages 109–119, Tokyo, Japan, October 2002. IEEE Computer Society.

[D13] *A Knowledge Level Methocology for Web Service Composition*. KLASE Deliverable D1.3.

[D32] *Formal Tropos for Web Service Compositions*. KLASE Deliverable D3.2.

[DDMvL98] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: An Environment for Goal–Driven Requirements Engineering. In *Proceedings of the 20^{th} International Conference on Software Engineering*, volume 2, pages 58–62, Kyoto (Japan), April 1998.

[DvLF93] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal–Directed Requirements Acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[FL+03] A. Fuxman, L. Liu, , M. Pistore, M. Roveri, and J. Mylopoulos. Specifying and analyzing early requirements in Tropos: Some experimental results. In *Proceedings of the 11^{th} IEEE International Requirements Engineering Conference*, Monterey Bay, California USA, September 2003. ACM-Press.

[FLM+04] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in Tropos. *Requirements Engineering Journal*, 2004.

[FT003] The Formal Tropos language, 2003. Available from `http://dit.unitn.it/~ft/doc/`.

[Fux01] A. Fuxman. Formal Analysis of Early Requirements Specifications. Master's thesis, University of Toronto, 2001.

[GMM90] 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.

[GNP+06a] E. Giunchiglia, M. Narizzano, M. Pistore, M. Roveri, and P. Traverso. Qbf-based bounded model checking. Technical report, ITC-irst, 2006.

[GNP+06b] E. Giunchiglia, M. Narizzano, M. Pistore, M. Roveri, and P. Traverso. Using quantified boolean logics to verify web service composition requirements. In *Proc. ECAI'06 Workshop on "AI for Service Composition"*, 2006.

[GNT04] E. Giunchiglia, M. Narizzano, and A. Tacchella. Qube++: an efficient qbf solver. In *Proc. FMCAD'04*, 2004.

[HJL96] 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.

[HKL97] C. Heitmeyer, J. Kirby, and B. Labaw. The SCR method for formally specifying, verifying, and validating requirements: tool support. In *Proceedings of the* 19^{th} *International Conference on Software Engineering*, pages 610–611. ACM Press, 1997.

[HV91] J. Halpern and M. Vardi. Model checking vs. theorem proving: A manifesto. In *Proceedings of the* 2^{nd} *International Conference on Principles of Knowledge Representation and Reasoning*, pages 325–334, 1991.

[Jac02] D. Jackson. Alloy: a lightweight object modeling notation. *ACM Transaction on Software Engineering Methodology*, 11(2):256–290, 2002.

[JSS00] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy Constraint Analyzer. In *Proceedings of the* 22^{nd} *International Conference on on Software Engineering*, Limerik, June 2000. ACM Press.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.

[MP94] 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.

[PPRS03] A. Perini, M. Pistore, M. Roveri, and A. Susi. Agent-oriented modeling by interleaving formal and informal specification. In *Proceedings of the* 4^{th} *International Workshop on Agent-Oriented Software Engineering*, Lecture Notes in Computer Science, Melbourne, Australia, July 2003. Springer.

[Spi89] J. Spivey. *The Z Notation*. Prentice Hall, 2nd edition, 1989.

[Yu97] 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.