Specifying and Analyzing Early Requirements: Some Experimental Results

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

[1]*Department of Computer Science, University of Toronto, 40 St. George St. M5S 2E4, Toronto, Canada*

[2]*Department of Information and Comm. Technology, University of Trento, Via Sommarive 14, I-38050, Trento, Italy*

[3]*ITC-IRST, Via Sommarive 18, I-38050, Trento, Italy*

{*afuxman,liu,jm*}*@cs.toronto.edu pistore@dit.unitn.it roveri@irst.itc.it*

Abstract

Formal Tropos is a specification language for early requirements. It is based on concepts from an agent-oriented early requirement model framework (i) and extends them with a rich temporal specification language. In earlier work, we demonstrated through a small case study how model checking could be used to verify early requirements written in Formal Tropos. In this paper we address issues of methodology and scalability for our earlier proposal. In particular, we propose guidelines for producing a Formal Tropos specification from an i* diagram and for deciding what model checking technique to use when a particular formal property is to be validated. We also evaluate the scope and scalability of our proposal using a tool, the T-Tool, that maps Formal Tropos specifications to a language that can be handled by* NuSMV*, a state-of-the-art model checker. Our experiments are based on a course management case study.*

1. Introduction

Early requirements engineering [18] is concerned with the analysis of the operational environment where a software system will eventually function. For organizational software, this environment consists of stakeholders and their objectives, business processes, and interdependencies. Early requirements engineering is usually done informally (if at all), and errors or misunderstandings at this stage are both frequent and costly. We are working on the development of a formal framework for modeling and analyzing early requirements. Our framework adapts results from the formal methods community to offer means for the precise modeling and analysis of an organizational environment.

Formal methods have been successfully applied to the verification and certification of software systems. In several industrial fields, formal methods are becoming integral components of standards [5]. Generally, formal methods have been applied to later phases of the development process, e.g., at the architectural or detailed design phase. This is basically due to mismatches between the constructs supported by formal specification languages and the concepts used in (early) requirements.

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

The Formal Tropos (hereafter FT) specification language [10, 11] has integrated concepts from *i** [18] with a temporal specification language. In [10, 11] the formal verification of an FT specification was performed using standard model checking techniques. A simple case study was used to illustrate the benefits of the formal analysis during early requirement analysis, e.g., by revealing incompleteness or inconsistencies that were not trivial to discover in an informal setting.

In this paper we tackle the problem of applying the techniques of [10, 11] to a case study of more substantial size than the example used in [10, 11], and discuss extensions, refinements and improvements we have developed to this purpose. We also introduce a prototype tool, called the T-Tool, which is based on the state-of-the-art symbolic model checker NuSMV [8]. The T-Tool automatically translates an FT specification into an Intermediate Language (IL) specification that could potentially link FT with different verification engines. The IL representation is then automatically translated into NuSMV, which performs different kinds of formal analysis, such as consistency checking, animation of the specification, and property verification.

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

The paper is structured as follows. Section 2 shows how to build an FT specification from an *i** model. In Section 3

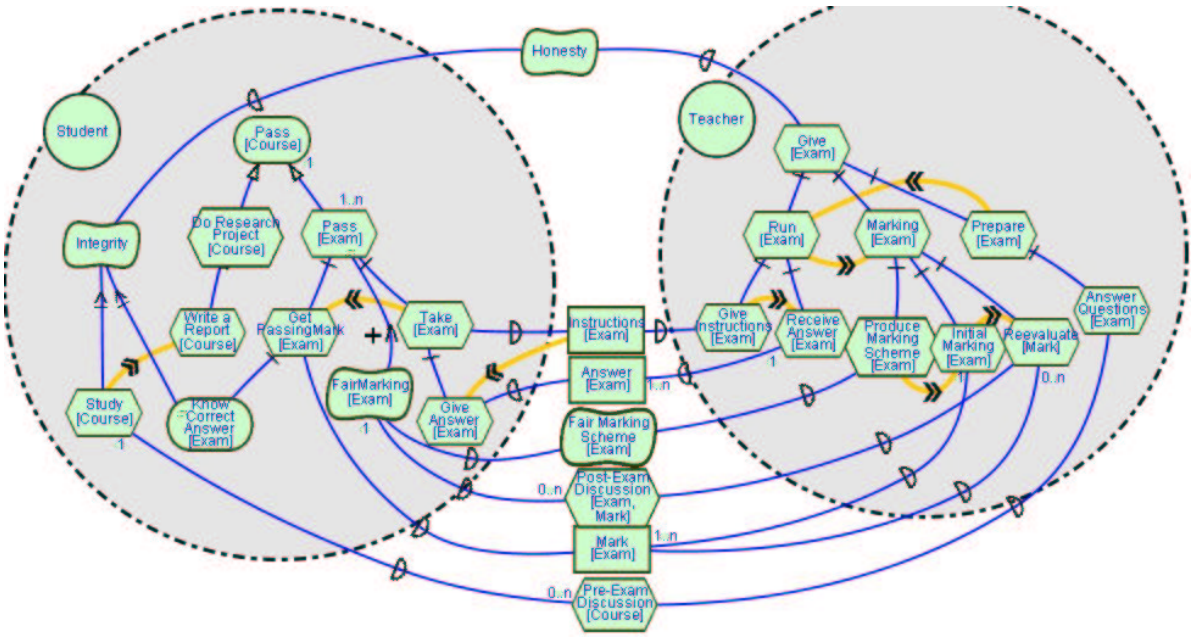

Figure 1. Annotated *i model of the course exam management case study**

we present the T-Tool, focusing on its functionalities, architecture, and usage guidelines. Section 4 presents the experiments we carried out. Section 5 discusses related work, draws conclusions, and outlines future work.

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

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

2.1. Strategic modeling with *i**

The *i** framework [18] supports goal- and agent-oriented modeling of early requirements of complex systems. It offers three main categories of concepts: actors, intentional elements, and intentional links. An *actor* is an active entity that carries out actions to achieve its goals. Figure 1 depicts the *i** model of a course exam management case study with its two main actors — a **Student** and a **Teacher**. From the *i** diagram, we can see that each actor has her own high-level goals/tasks and alternative ways to refine and operationalize them. Moreover, the goal hierarchies of the actors are interconnected through dependency/delegation relationships, in terms of which individual actors form social and operational networks.

Intentional elements in *i** include *goals*, *softgoals*, *tasks*, and *resources*, and can either be internal to an actor, or define dependency relationships between actors. A *goal* (rounded rectangle) is a condition or state of affairs in the world that the stakeholders would like to achieve. For example, a student's objective to pass a course is modeled as goal **Pass[Course]**. A *softgoal* (irregular curvilinear shape) is typically a non-functional attribute, with no clear-cut criteria as to when it is achieved. For instance, the fact that a student may want that the marking of her exam is fair is modeled as softgoal **FairMarking[Exam]**. A *task* (hexagon) specifies a particular course of action that produces a desired effect. In our example, all intentional elements of **Teacher** are modeled as tasks. A *resource* (rectangle) is a physical or information entity. Thus, **Instructions[Exam]**, **Answer[Exam]**, and **Mark[Exam]** are modeled as resources. In Figure 1 a boundary delimits the intentional elements internal to an actor. Intentional elements outside the boundaries correspond to goals, softgoals, tasks and resources whose responsibility is delegated from one actor to another.

Intentional links include *means-ends*, *decomposition*, *contribution* and *dependency* 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 the student wants to pass an exam (**Pass[Exam]**), she needs to attend the exams (**Take[Exam]**), and get a passing mark (**GetPassingMark[Exam]**). A *contribution* link (—→—) describes the impact that an element has on another. This can be negative or positive (**FairMarking[Exam]**), and its extent can be partial or sufficient. *Dependency* links (—▷—) describe inter-agent dependencies. For example, the student depends on her teacher for **Instructions[Exam]** and **Mark[Exam]**, while the teacher depends on the student for **Answer[Exam]** and **Honesty**.

In Figure 1 we have annotated the *i** model with ad-

ditional constraints on the valid dynamics of the domain. *Prior-to* links (⋙) can be used to represent the temporal order of tasks. 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. The numbers labeling the links define *cardinality constraints*. For instance, for each **Pass[Course]** goal there must be at least one **Pass[Exam]** subgoal, while to satisfy the student's expectation on the fairness of marking (**FairMarking[Exam]**) there may be zero or more petitions (**PostExamDiscussion[Exam,Mark]**). Links without number suggest one-to-one connections.

2.2. Formal Tropos specifications

An FT specification extends an *i** model with annotations in an expressive temporal specification language that permits to restrict the valid behaviors of the model. With an FT specification, one can ask questions such as: Can we construct valid operational scenarios based on the model? Is it possible to fulfill the primary goals of actors in the current model? Do the decomposition links and the prior-to constraints induce a meaningful temporal order for goal fulfillment? Do the dependencies represent a valid synergy or synchronization between actors?

An FT specification consists of a sequence of class declarations such as entities, actors, intentional elements, and dependencies. Each declaration associates a set of attributes to the class and characterizes its instances. Moreover, class declarations contain temporal constraints expressed in a typed first-order linear time temporal logic (LTL). These constraints describe the valid lifetime evolutions of the class instances and define synchronization between different classes instances. A full definition of the Formal Tropos language can be found in [10, 11].

Figure 2 is an excerpt of the FT specification of the course exam example. The initial FT specification skeleton can be obtained from the *i** model by mapping actors and intentional elements into corresponding FT classes and by adding non-intentional entities of the domain if any (e.g., **Course** and **Exam**).

Most of the attributes in FT are references to other classes. For example, goal **PassExam** refers to the specific exam to be passed (attribute **exam**), and to the **PassCourse** goal that motivates the student to pass the exam (attribute **pass_course**). Similarly, dependency **Mark** refers to the exam that has to be marked (attribute **exam**) and to **GetPassingMark** goal of the student that motivates the expectation of having a mark (attribute **gpm**). There are also attributes of elementary type, which define relevant states of a class. For instance, Boolean attribute **passed** of dependency **Mark** determines whether a mark is passing or not. 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 such as **passed** usually change during the lifetime of class instances.

```
Entity Course
Entity Exam
   Attribute constant course : Course
Actor Student
Actor Teacher
Goal PassCourse
   Actor Student
   Mode achieve
   Attribute constant course : Course
Goal PassExam
   Actor Student
   Mode achieve
   Attribute constant exam : Exam
            constant pass_course : PassCourse
Goal GetPassingMark
   Actor Student
   Mode achieve
   Attribute constant exam : Exam
            constant pass_exam : PassExam
Softgoal Integrity
   Actor Student
   Mode maintain
Task GiveExam
   Actor Teacher
   Mode achieve
   Attribute constant exam : Exam
Resource Dependency Mark
   Depender Student
   Dependee Teacher
   Mode achieve
   Attribute constant exam : Exam
            constant gpm : GetPassingMark
            passed : boolean
```

Figure 2. Excerpt of FT class declaration

Since actor **Student** is the owner of goal **PassExam** and of softgoal **Integrity**, the FT specification has **Student** as the Actor attribute of the two goals. Similarly, Depender and Dependee attributes of dependencies represent the two parties involved in a delegation relationship. Intentional elements also have a Mode attribute, which defines the modality of the fulfillment of the goal. For instance, the mode of goal **PassExam** is achieve, which means that the student wants to reach a state where the exam has been passed, and therefore the goal is fulfilled. Softgoal **Integrity**, instead, has a maintain mode, since the condition of no cheating is to be continuously maintained.

Figure 3 contains some examples of constraints on the lifetime of class instances. Invariant constraints define conditions that should be true throughout the lifetime of 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 3 binds a **Mark** object with its associated **GetPassingMark** object, while the second invariant imposes a cardinality constraint for **Mark** objects.

Two critical moments in the lifecycle of intentional elements and dependencies are the instants of their creation and fulfillment. The creation of a goal is interpreted as the moment in which the owner or depender expects or desires to achieve the goal, while its fulfillment is the mo-

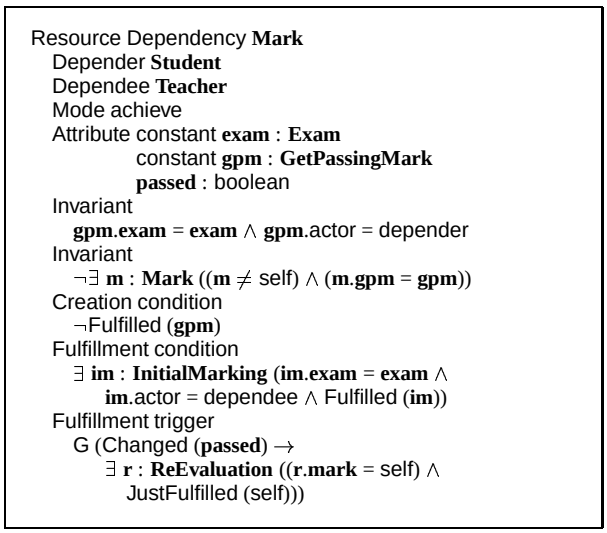

Figure 3. Example of FT constraints

ment in which the goal condition is actually achieved. In FT, creation and fulfillment constraints can be used to define conditions for these two moments in the life of intentional elements. Creation constraints should be satisfied whenever a new object is created, while Fulfillment constraints should hold whenever a goal or softgoal is satisfied, a task is performed, a resource is made available, or a dependum is delivered. Typically, primary intentional elements, (e.g., **Integrity**, **PassCourse**) have fulfillment constraints, but no creation constraints: we are not interested in modeling the reasons why a student wants to pass a course and to maintain her integrity. Subordinate intentional elements (e.g., **PassExam**, **GetPassingMark**) typically have constraints that relate their creation with the state of their parent intentional elements. For instance, Figure 3 shows that a creation condition for an instance of dependency **Mark** is that the parent goal **GetPassingMark** is not yet fulfilled: if the student has received a passing mark, there is no need to ask for another mark. Creation and fulfillment constraints are further distinguished as sufficient conditions (keyword trigger), necessary conditions (keyword condition), and necessary and sufficient conditions (keyword definition) (see [10, 11] for details). We note that the creation condition of dependency **Mark** together with the fulfillment condition of task **GetPassingMark** elaborate the delegation relationship between **Student** and **Teacher** in the corresponding *i** diagram. Goal decomposition relationships can be specified in a similar fashion.

In an FT specification we also need to specify properties desired to hold in the domain. These properties are then verified against the model we built. Figure 4 presents such properties for the course exam case study. We distinguish between assertion properties (A1-4) that should hold for all valid evolutions of the FT specification, and possibility properties (P1-4) that should hold for at least one. 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 **PassExam** is created, no other instance of **PassExam** exists corresponding to the same exam and the same student. Properties A4, P1, P2, and P4 are "global", i.e., they express conditions on the entire model, and are not attached to any particular event.

2.2.1. From *i** to FT: Translation guidelines

It is usually hard to develop a satisfactory formal specification of a system, even when one starts from a good informal specification. In our experience, the difficulties of writing 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. For instance, in dependency **Mark** (see Figure 3), the two invariant conditions, the creation condition and the fulfillment conditions express constraints that are related to goal delegation and to cardinality constraints in the *i** model. These constraints can all be derived from the *i** model automatically by applying specific translation rules. The additional non-standard constraints that should be manually added to the FT specification are necessary to capture the nature of the applicative domain. For instance, the last constraint in dependency **Mark** expresses that a sufficient condition for the fulfillment of the dependency is that whenever a mark change status there must be a corresponding reevaluation just fulfilled.

The following are some of the rules that we have defined for generating FT specification based on *i** model:

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

Goal **PassExam**
Creation assertion condition */* A1: A student can only pass an exam once. */*
$\forall$ **p** : **PassExam** (**p**.actor = actor $\wedge$ **p**.**exam** = **exam** $\wedge$ **p**.**pass_course** = **pass_course** $\rightarrow$ **p** = self)
Resource dependency **Mark**
Fulfillment assertion condition */* A2: For each mark there was an answer corresponding to it. */*
$\exists$ **a** : **Answer** (**a**.dependee = depender $\wedge$ **a**.depender = dependee $\wedge$ **a**.**exam** = **exam** $\wedge$ Fulfilled (**a**))
Resource dependency Mark
Fulfillment assertion condition */* A3: A mark can only be changed if there is a petition. */*
$\neg$**passed** $\wedge$ F **passed** $\rightarrow$ F $\exists$ **ped** : **PostExamDiscussion** (**ped**.**mark** = self)
Global Assertion */* A4: If the student wants to maintain her integrity, she cannot pass an exam without studying. */*
$\forall$ **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 Possibility */* P1: It is possible for a student to pass a course. */*
$\exists$ **pc** : **PassCourse** (Fulfilled (**pc**))
Global Possibility */* P2: It is possible that a student passed a course without passing the exam. */*
$\exists$ **pc** : **PassCourse** (Fulfilled (**pc**) $\wedge$ $\neg$ $\exists$ **pe** : **PassExam** ((**pe**.**pass_course** = **p**) $\wedge$ Fulfilled (**pe**))
Goal **PassExam**
Fulfillment possibility condition */* P3: It is possible that a student passed an exam, but still thinks that the marking is not fair. */*
$\exists$ **f** : **FairnessBeDecided** (**f**.**pass_exam** = self $\wedge$ Fulfilled (**f**) $\wedge$ $\neg$ **f**.**satisfied**)
Global Possibility */* P4: It is possible that a teacher expects an exam answer from a student who is never committed to the exam. */*
$\exists$ **a** : **Answer** (G $\neg$ $\exists$ **p** : **PassExam** (**p**.**exam** = **a**.**exam** $\wedge$ **p**.actor = **a**.dependee))

Figure 4. Example of Formal Tropos properties

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

3. The T-Tool

The T-Tool is based on finite-state model checking [9]. It takes as input an FT specification along with parameters that specify which parts of the specification to consider and, for the selected classes, an upper bound to the number of class instances that can be instantiated. The T-Tool builds a finite model that represents all possible behaviors of the domain that satisfy the constraints of the specification, and checks the model to ensure that it exhibits desired behaviors. The 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 comes out with feedback on errors in the FT specification and with 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.

3.1. T-Tool functionalities

3.1.1. 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.

3.1.2. 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. The 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 will be eventually created. This check aims at verifying whether the current upper bounds of the number of class instances are reasonable, and whether they 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.

3.1.3. Possibility checks

Possibility checks verify whether we are over-constraining the specification, 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 witnesses the fact that the possibility is valid. If no such trace is found, an error message is reported. 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.

3.1.4. 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 allowing for invalid scenarios. Also 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 case study, an assertion that we wish to hold is "a student can never pass a course without taking all the exams of the course and without doing a research project". If this (quite reasonable) assertion is false, the T-Tool will produce a trace that shows under what circumstances the student can pass the course without passing exams and doing a research project. Discussions with the stakeholder may then clarify whether the trace produced corresponds to a valid scenario (and hence the assertion has to be changed) or whether the FT specification has to be strengthened in order to prohibit the counter-example.

3.2. T-Tool architecture

The T-Tool performs the verification of an FT specification in two steps (see Figure 5). In the first step, the FT specification is translated into an Intermediate Language (IL) specification. In the second step, the IL specification is given in input to the verification engine, which is built on top of the NuSMV model checker [8].

3.2.1. From FT to IL

IL can be seen as a simplified version of FT, where the syntactic sugar of the FT specification is removed. The focus of IL is on the dynamic aspects of the application domain. An IL specification consists of four parts: class declarations,

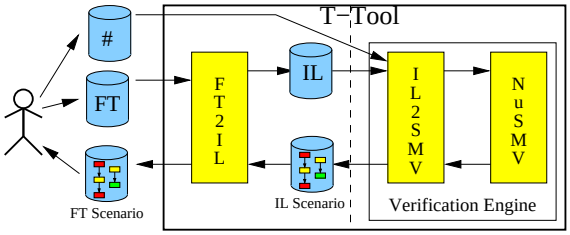

Figure 5. The T-Tool framework

temporal specifications, possibilities, and assertions. The *class declarations* in IL correspond to the entity, actor, goal, and dependency declarations of an FT specification. The distinction among these different types of classes is ignored and all the special attributes are transformed into standard attributes. *Temporal specifications* in IL restrict the valid temporal behaviors of the objects in the specification. As in FT, *assertions* (resp. *possibilities*) in IL describe expected properties that should be exhibited by all (resp. by some of) the valid scenarios of the specification. Unlike FT, temporal specifications, assertions and possibilities are all global in IL, that is, they are not anchored to the relevant "strategic" components anymore.[1]

The FT2IL architectural block takes care of the translation of an FT specification to the corresponding IL. 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.

In this architecture, IL plays an important role in providing the T-Tool with an open architecture and a flexible approach for linking requirements specification languages and the languages used by a variety of formal verification tools. On one hand, it allows for the adoption of existing verification techniques for different specification languages, at the cost of writing new translators to IL. On the other hand, it allows the tool to be linked with other analyzers, by writing translators from IL to the native language of these analyzers.

3.2.2. The model checking verification engine

The actual verification is performed by NuSMV [8]. NuSMV is a state-of-the art model checker based on symbolic model checking techniques. Symbolic techniques have been developed to reduce the effects of the state-explosion problem, thus enabling the verification of large designs [9, 16]. NuSMV adopts symbolic model checking algorithms based on Binary Decision Diagrams (BDD) [6] and on propositional satisfiability (SAT) [4]. BDD-based model checking performs an exhaustive traversal of the model by considering all possible behaviors in a compact way. Such exhaustive exploration allows BDD-based model checking algorithms to conclude whether a given property is satisfied (or falsified) by the model. On the other hand, this

[1] For lack of space, we refer to [10, 11] for a more elaborate description of IL and its semantics.

exhaustive exploration makes BDD-based model checking very expensive for large models. SAT-based model checking algorithms look for a trace of a given length that satisfies (or falsifies) a property. SAT-based algorithms are usually more efficient than BDD-based algorithms for traces of reasonable length, but, if no trace is found for a given length, then it may still be the case that the property is satisfied by a longer trace. That is, SAT-based model checking verifies the satisfiability of a property only up to a given length, and is hence called Bounded Model Checking (BMC) [4]. As we will discuss in Section 3.3, the T-Tool exploits both BDD-based and SAT-based model checking.

Several extensions have been applied to the NuSMV model checker to allow for the verification of IL specifications. An IL2SMV module has been added. It takes an IL specification and builds a finite state machine in the NuSMV format. Given the IL specification and the upper bounds of the number of class instances, 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 constraints. Since the NuSMV formalism does not allow for the creation of new objects at run-time, to deal with instance creation, a special flag is added to each class during the translation. Quantifiers in IL are interpreted over the number of class instances that exist in the current state. To construct the model, IL2SMV adopts the synthesis algorithm for LTL specification provided by NuSMV.

To handle the different facets of an FT specification, NuSMV has also been extended with new functionalities: for instance, the BMC engine has been extended with past operators [2], and a new more flexible interactive simulator has been added.

3.3. Heuristics for model construction and property verification

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

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

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

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

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

Table 1. Results for possibility checks

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

Table 2. Results for assertion checks

4. Experimental results

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

4.1. 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 4. More results can be found at the URL `http://sra.itc.it/tools/t-tool/experiments/cm`.

To stress the scalability of the proposed verification techniques, we have performed the tests considering models of different size. More precisely, we have considered different upper bounds to the number of instances for each class. We report here the case of 1 and 2 instances for each class, and one intermediate 1..2 case where we allow 2 instances for some classes (in particular, the student and its goals and tasks), but only 1 instance for other classes (the teacher and its tasks, and the course). Moreover, we experimented with the different model checking techniques, namely SAT-based bounded model checking ("BMC" in the tables), BDD-based model checking ("BDD"), and, in the case of assertions, BDD-based model checking on reduced models, as described in Section 3.3 ("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 2^{477} to 2^{1077} states while moving from the 1 instance to the 2 instance per class.

4.2. Results

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

1. Possibilities P1-3 are valid, and witness scenarios of length 3, 3 and 4 respectively are produced by the T-Tool.
2. Possibility P4 is invalid. No witness scenario is found up to length 10 for the 1 and 1..2 instances and up to

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

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. No counter-example is found in 1 instance case, but 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 since, according to the FT specification, the teacher only starts marking if at least one student takes the exam.

4.3. Discussion

4.3.1. Effectiveness

For our case study, the proposed approach was effective in producing an FT specification of good quality. It also led to an improved understanding of the domain by revealing several tricky aspects of the case study. The validation techniques provided by the T-Tool have been useful in detecting bugs, while animation was useful during early validation steps by identifying trivial bugs. For instance, due to a missing creation condition for the student goal **TakeExam**, a student was allowed to try to take an exam even if no teacher was giving it. Likewise, the consistency checks have been able to detect a trivial error in the creation condition of student's goal **Study**, which does not allow two students to study the same course. The validation of assertions and possibilities has revealed subtle bugs due to the interaction of different goals, dependencies and constraints. For instance, due to an error in the fulfillment condition of **ReceiveAnswers**, a student could prevent the teacher from fulfilling the task **GiveExam** by declaring her intention to take the exam and by never taking it. In another case, a student could not decide on the fairness of marking (softgoal **FairMarking**) even after she received a **Mark**, since she was expecting a marking scheme from the wrong teacher. This was due to a missing creation condition in the dependency **FairMarkingScheme**. In both cases, the T-Tool's ability to generate counter-examples helped in pinpointing the problems.

The experiments show that the usage of the abstraction techniques described in Section 3.3 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.

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

4.3.2. Performance

Performance results on the T-Tool are 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 heuristics proposed for reducing the model point out a promising direction for the verification of assertion properties, even if they do not (yet) completely solve the performance problem. 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.

5. Related work and conclusions

In earlier work [10, 11] we have proposed a framework for the specification and verification of early requirements. This paper presents the T-Tool, a prototype tool that supports the process of verification, and demonstrates through experiments that the framework can scale up and serve as a useful basis for the verification of early requirements. The T-Tool permits the generation of finite models from an FT specification and supports model checking on such models. The T-Tool is based on NuSMV, an open architecture for model checking. In our experience, the possibility of extending NuSMV with new functionalities (e.g., a new input language, past operators, enhanced simulator) has been crucial for its effective application to the analysis of FT specification.

Formal analysis is often used to verify correctness of specifications, but, it is usually applied in later phases. For instance, in [1, 12] formal verification techniques were used for the analysis of specifications expressed in the SCR formalism, and in [7] NuSMV is used for the verification

of RSML specifications. The works that are most relevant to ours are Alcoa/Alloy [14, 13], KAOS [15], and the work on "Topoi Diagrams" [17]. *Alcoa* [14] is a tool for analyzing object models that describe the architectural or structural properties of a system design. It has been used to verify various architectural frameworks, protocols, and schemes. The input language – *Alloy* [13] – is a notation based on Z, but has been tailored to fit object models and is amenable to automatic analysis. Similarly to the T-Tool, Alcoa uses SAT-based bounded model checking for assertion analysis (under-specify checking) and possibility analysis (over-specify checking). The main differences between Alcoa and the T-Tool is their focus on different applications (object vs requirements models). Moreover, the T-Tool supports a broader set of verification techniques, including BDD-based model checking and heuristics for reducing the model size for proving assertion properties. *KAOS* [15] is a framework that supports (early) requirements analysis. It shares with FT the goal-oriented flavor. Also the design of the temporal logic component in FT has been inspired by KAOS. The main difference between the two frameworks is in the analysis techniques used. The T-Tool supports model checking verification techniques, while KAOS is based on theorem proving techniques. *Topoi diagrams* [17] represent statements of gradual influence between variables (e.g., the more X, the more Y) and can be used in system requirements to describe how designers believe influence should propagate through a system. Topoi diagrams are related to *i** diagrams, where intentional links describe influences between the intentional elements of a domain. On top of the topoi diagrams, temporal logic formulas describing a property of the model can be checked. The focus of this approach is limited to formulas of a specific form that check whether a given input results in an expected output. Moreover, the framework in [17] is based on explicit state model checking techniques [9], rather than on symbolic techniques.

There are several directions for further research. First, we are investigating the use of techniques that guarantee that an FT specification is correct regardless of the upper bounds of the number of class instances. We are also working to the refinement and the automation of the verification approach proposed in this paper, by defining heuristics to choose and refine the set of constraints considered while proving a property, and by alternating automatically phases in which the tool tries to prove the validity of a model and phases where the tool tries to find bugs. Optimizations of the model generator and advanced abstraction techniques that exploit, for instance, possible symmetries in the specification are also under investigation. 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.

References

[1] J. M. Atlee and J. Gannon. State-based model checking of event-driven systems requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, Jan. 1993.

[2] M. Benedetti and A. Cimatti. Bounded Model Checking for Past LTL. In *the* 9^{th} *Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[3] S. Berezin, S. Campos, and E. M. Clarke. Compositional reasoning in model checking. In *COMPOS*, volume 1536 of *LNCS*, pages 81–102. Springer, Sept. 1998.

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

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

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

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

[8] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, number 2404 in LNCS, Copenhagen (DK), July 2002. Springer.

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

[10] A. Fuxman. Formal analysis of early requirements specifications. Master's thesis, University of Toronto, 2001.

[11] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in Tropos. In *IEEE Int. Symposium on Requirements Engineering*, pages 174–181, Toronto (CA), Aug. 2001. IEEE Computer Society.

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

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

[14] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *the* 22^{nd} *Int. Conference on on Software Engineering*, Limerik, June 2000. ACM Press.

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

[16] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[17] T. Menzies, J. Powell, and M. E. Houle. Fast formal analysis of requirements via "topoi diagrams". In *the* 23^{rd} *Int. Conference on Software Engineering*, pages 391–400, Toronto, CA, May 2001. ACM Press.

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