Model Checking Early Requirements Specifications in Tropos

Ariel Fuxman
University of Toronto
afuxman@cs.toronto.edu

Marco Pistore
ITC-IRST
pistore@itc.it

John Mylopoulos
University of Toronto
jm@cs.toronto.edu

Paolo Traverso
ITC-IRST
traverso@itc.it

Abstract

This paper describes an attempt to bridge the gap between early requirements specification and formal methods. In particular, we propose a new specification language, called Formal Tropos*, that is founded on the primitive concepts of early requirements frameworks (actor, goal, strategic dependency) [15], but supplements them with a rich temporal specification language. We also extend existing formal analysis techniques, in particular* model checking*, to allow for an automatic verification of relevant properties for an early requirements specification. Our preliminary experiments demonstrate that formal analysis reveals gaps and inconsistencies in early requirements that are by no means trivial to discover without the help of formal analysis tools.*

1. Introduction

Early requirements analysis is one of the most important and difficult phases of the software development process. It is the phase where the requirements engineer is concerned with understanding the organizational context for an information system, and the goals and social dependencies of its stakeholders. This phase demands critical interactions with users and other stakeholders. A misunderstanding at this point may lead to expensive errors during later development stages. Not surprisingly, several approaches have been proposed in recent years on suitable concepts, languages and analysis techniques specifically tailored for this phase (e.g., [8, 15, 1]).

Formal methods have a great potential as powerful means for the specification, early debugging and certification of software. They have been successfully applied in several industrial applications, and they are even becoming integral components of standards in certain fields [2]. 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, such as the design phase (see for instance [6]). As a result, there is a mismatch between the concepts used for early requirements specifications (e.g., goal and actor) and the constructs of formal specification languages such as Z [13], SCR [11], etc.

Our long-term aim is to provide a framework for the effective use of formal methods in the early requirements phase. The framework should allow for the formal and mechanized analysis of early requirements specifications expressed in a formal modeling language. In this paper, we present some results that constitute a first step towards this goal. This is accomplished by extending and formalizing an existing early requirements modeling language, also by building on state-of-the-art formal verification techniques.

In order to allow for formal analysis, we extend the i* modeling language [15] into a formal specification language called *Formal Tropos*[1]. The language offers all the primitive concepts of i* (such as actors, goals, and dependencies among actors), but supplements them with a rich temporal specification language inspired by KAOS [8].

We also extend an existing formal verification technique, model checking [10], in order to support the mechanized analysis of Formal Tropos specifications. Using this machinery, we provide for different kinds of analysis on a Formal Tropos specification. For instance, checking whether the specification is consistent, or whether it respects a number of desired properties. Moreover, a specification can be animated in order to give the user immediate feedback on its implications.

The proposed approach has been implemented as a prototype tool, called *T-Tool*[2]. T-Tool is built on top of NuSMV [5], a state-of-the-art symbolic model checker originally designed for hardware verification. In order to adapt the verification techniques of NuSMV to the new application domain, we have defined an intermediate language, and we have extended NuSMV to support this new input language. T-Tool translates a Formal Tropos specification into the in-

[1] Formal Tropos is part of a wider-scope framework, called *Tropos* [4], which proposes the application of concepts from the early requirements phase to the whole software development process, including late requirements, architectural and detailed design, and implementation.

[2] Up-to-date information on Formal Tropos and T-Tool can be found at `http://sra.itc.it/tools/t-tool/`.

termediate language and then calls NuSMV for the actual verification. We have experimented with Formal Tropos and T-Tool, using a simple case study. In spite of its simplicity, the case study demonstrates the benefits of formal analysis in revealing incompleteness and inconsistency errors that are by no means trivial to discover in an informal setting.

Structure of the paper. In Section 2 we describe the i* modeling language and introduce the case study we will work on in the rest of the paper. Section 3 presents the Formal Tropos language and explains its original aspects with respect to an i* specification. Section 4 elaborates the different kinds of formal analysis that the engineer can perform within the proposed framework, while Section 5 describes the technical aspects of the verification performed by T-Tool. Finally, Section 6 presents some concluding remarks and discusses future research directions.

2. The i* Modeling Language

The i* modeling language has been specifically designed for the description of early requirements. It assumes that during this phase it is important to understand and model social settings which involve actors who depend on each other for goals to be achieved, tasks to be performed, and resources to be furnished. The language provides a graphical notation to describe such settings. An SD diagram, for instance, is used to represent the *strategic dependencies* of the actors, a central concept of i*. Dependencies express intentional relationships that exist among actors in order to fulfill some strategic objectives. A dependency describes an "agreement" between two actors, the *depender* and the *dependee*. The *type* of the dependency describes the nature of the agreement. *Goal* dependencies are used to represent delegation of responsibility for fulfilling a goal. *Softgoal* dependencies are similar to goal dependencies, but their fulfillment cannot be defined precisely (because it depends on subjective criteria, or can occur only to a given extent). *Task* dependencies represent situations where the dependee is required to perform a given activity, while *resource* dependencies require the dependee to provide a resource.

The details on i* are presented in [15]. Here we briefly review it by using the Insurance Company case study, initially introduced in [16]. The actors of the case study are the customers and the insurance company, Customer and InsuranceCo. The main goal of the customer is to be reimbursed for damages in case of an accident (goal BeInsured in what follows). As the customer is not able to fulfill this goal by herself, the goal is refined into a goal dependency CoverDamages, from the customer to the insurance company. Conversely, the insurance company depends on its customers to have a continued business, by fulfilling softgoal dependencies such as AttractCustomers. In order to achieve the previous goals, it is necessary to include additional actors, such as Garage and Appraiser, and additional dependencies. For instance, the Customer depends on the Garage to have her car repaired (dependency RepairCar) and the insurance company depends on the Appraiser to estimate the reasonability and amount of the damages (AppraiseDamage).

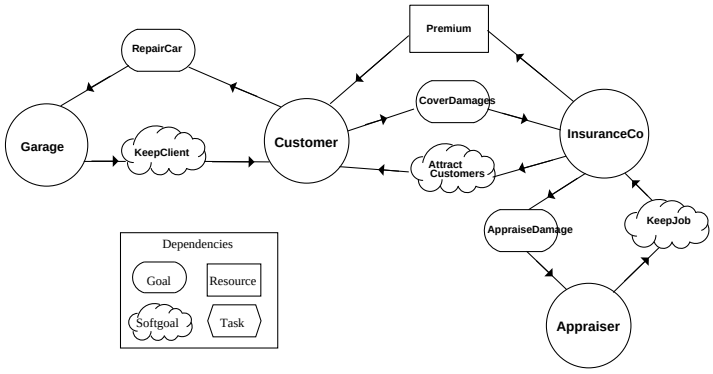

Figure 1. SD diagram of the case study.

Figure 1 presents an i* SD diagram for the case study. We use a subset of this case study as the running example for the rest of the paper. In particular, we focus on the CoverDamages and RepairCar dependencies.

3. The Formal Tropos Language

This section presents the most relevant aspects of Formal Tropos using the Insurance Company example (see Figure 2). A thorough definition of the Formal Tropos language appears in [9].

Formal Tropos has been designed to supplement the i* specification language with the description of the dynamic aspects of dependencies among actors. While using it, we focus not only on the dependencies 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. In order to be able to represent these dynamic aspects, dependencies as well as actors become "classes" in the Formal Tropos specification. We can have many instances of any one class during the evolution of the system (e.g., different RepairCar dependencies may exist for different Customers, or for different accidents of the same car), and attributes are associated to the instances of actors and dependencies, in order to represent their relationships (e.g., a customer wants *her* car to be repaired).

A Formal Tropos specification describes the relevant objects of a domain and the relationships among them. The description of each object is structured in two layers. The

Entity Claim
  Attribute constant car: Car
Entity Car
  Attribute runsOK: **boolean**
Actor InsuranceCo
Actor Garage
Actor Customer
  Goal BeInsured
    Mode maintain
    Fulfillment definition $\forall$cov : CoverDamages
      (cov.depender = **self** $\rightarrow$ $\Diamond$**Fulfilled**(cov))
Dependency CoverDamages
  Type goal
  Mode achieve
  Depender Customer
  Dependee InsuranceCo
  Attribute constant cl: Claim
  Creation
    condition $\neg$cl.car.runsOK
    trigger JustCreated(cl)
Dependency RepairCar
  Type goal
  Mode achieve
  Depender Customer
  Dependee Garage
  Attribute constant cl : Claim
  Creation
    condition $\neg$cl.car.runsOK
  Fulfillment
    condition for depender cl.car.runsOK

Figure 2. The Formal Tropos specification.

outer layer is similar to a class declaration, since it defines the structure of the instances together with their attributes. The inner layer expresses properties of the lifetime of the object, given in a typed first-order linear-time temporal logic. A class can be an *actor*, a *dependency*, or an *entity*. Entities are used to represent non-intentional elements that exist in the organizational setting being modeled. In our example (see Figure 2) we have the entities Claim and Car.

The attributes of a Formal Tropos class denote relationships among different objects. For example, each claim filed by a customer refers to a specific car, represented by attribute car of entity Claim (see Figure 2). The facet **constant** of this attribute states that, once a car is associated to a claim, the relationship must remain unchanged forever. Formal Tropos defines other attribute facets, such as **multivalued** and **optional**, which do not appear in our example.

As in i*, actors in Formal Tropos have *goals* that describe their strategic interests. For instance, in our example, the actor Customer has a goal named BeInsured. Furthermore, intentional relationships among actors are represented as dependencies (see, e.g., dependencies CoverDamages and RepairCar in the example). The **type** of a dependency (**goal**, **softgoal**, **task**, **resource**), as well as its **depender** and **dependee** actors, are included as special attributes in the dependency class declaration.

An important aspect of Formal Tropos is that it focuses on the conditions for the *fulfillment* of goals and dependencies. More precisely, goals and dependencies can be fulfilled with different *modalities*. For example, the modality of RepairCar is **achieve**, which means that an instance of the dependency becomes fulfilled once the car is eventually repaired. Dependency CoverDamages also has **achieve** modality, since it becomes fulfilled as soon as the insurance company reimburses the damages. On the other hand, goal BeInsured of actor Customer is of **maintain** modality, as the customer expects to be insured in a continuing way. There are other modalities, such as **achieve&maintain**, which is a combination of the previous two modes. Also **avoid** modality, which means that the fulfillment conditions should be prevented.

The inner layer of a Formal Tropos class declaration consists of properties that describe the dynamic aspects of actors, goals, and dependencies. Important moments in the lifetime of an instance of a dependency are its creation and its fulfillment. Consequently, we distinguish three types of properties for dependencies. **Creation** properties should hold at the time of creation of a new instance of the dependency, while **fulfillment** properties should hold when a dependency is satisfied. Finally, **invariant** properties should be true throughout the lifetime of the dependency. Actor goals have fulfillment properties, but not creation properties or invariants, since they are assumed to arise and exist together with their corresponding actor. On the other hand, entities and actor classes may have creation and invariant properties, but clearly not fulfillment ones. In addition to the type of the property, we also distinguish those that express sufficient conditions (facet **trigger**), necessary conditions (facet **condition**), and necessary-and-sufficient conditions (facet **definition**).

In our example, dependency CoverDamages has a creation condition (formula $\neg$cl.car.runsOK in Figure 2) that states that the car should not be working at the time the goal arises. Its creation trigger represents the fact that, whenever a customer files a claim cl (formula **JustCreated**(cl)), a dependency for covering the repairs arises. According to the specification in Figure 2, the goal of repairing a car can only arise if the car is not working. Similarly, a necessary condition for the fulfillment of the dependency is that the car should be running OK. This is a condition that the customer imposes (...the garage would be happy to declare a car repaired even if it does not run). Formal Tropos represents this fact with the facet **for depender**.

In addition to facets, Formal Tropos properties are described with formulas given in a typed first-order linear-time temporal logic. In the logic, quantifiers $\forall$ and $\exists$ range over all the instances of a given class. The formulas may refer to the attributes of the class that the corresponding property belongs to. Also, instances may express properties about themselves using the keyword **self** (see the fulfillment definition of goal BeInsured in Figure 2). Three special predicates can appear in the temporal logic formulas: predicate **JustCreated**(el) holds in a state if element el exists in this state but not in the previous one. Predicate **Fulfilled**(el) holds if el has been fulfilled. Finally, predicate **JustFulfilled**(el) holds if **Fulfilled**(el) holds in this state, but not in the previous one. Predicates **Fulfilled** and **JustFulfilled** are defined only for goals and dependencies.

Using suitable temporal operators, the logic makes it possible to express properties that are not limited to the current state of the system, but also to its past and future history. For instance, formula $\Box\phi$ (always in the future ϕ) expresses the fact that formula ϕ should hold in the current state and in all the future states of the evolution of the system. Formula $\blacklozenge\phi$ (sometimes in the past) holds if ϕ is true in the current state or if it was true in some past state of the system. The classical temporal operators used in the Formal Tropos formulas are $\circ$ (next state), $\bullet$ (previous state), $\Diamond$ (eventually in the future), $\blacklozenge$ (sometimes in the past), $\Box$ (always in the future) and $\blacksquare$ (always in the past). Other useful operators are $(\Diamond \vee \blacklozenge)$ (once, in the past or in the future) and $(\Box \wedge \blacksquare)$ (always in the past and in the future). Goal and dependency modalities often hide the usage of temporal operators. This is done on purpose. Reducing the number of temporal operators in the formulas results in more intuitive and readable specifications. Modalities provide an easy-to-understand subset of the language of temporal logics. When the modalities are not sufficient to capture all the temporal aspects of a condition, the temporal operators may appear explicitly in the formulas. This is the case for goal BeInsured of actor Customer in the example in Figure 2. Its fulfillment condition requires that whenever a CoverDamages dependency exists for the customer, it will eventually be fulfilled (formula $\Diamond$**Fulfilled**(cov)). Since the goal has a **maintain** modality, this property has to hold in a continuous way for the goal to be fulfilled.

The properties used so far in the example are *required* to hold for all scenarios. In general, we distinguish two sets of properties: those that are enforced, and those that express *desired* behaviors. The properties of the former set define constraints on the behavior of the system that hold in all valid scenarios. The properties of the latter set denote expectations on the behavior of the system and are used to validate the model. These can be either **assertion** properties, which are desired to hold for all the valid scenarios of a system; or **possibility** properties, which should hold for at least one valid scenario. In the next section, we illustrate how to check whether assertions and possibilities satisfy the constraints of a specification.

4. Formal Analysis

Formal analysis techniques are typically applied during late phases of development in order to validate the design or implementation of a system against its requirements. Our aim is to use formal analysis techniques in order to assist the analyst during requirements elicitation, by allowing her to identify errors and limitations of the specification that are not evident in an informal setting.

Our formal analysis tool, T-Tool, takes a Formal Tropos specification as input, and builds an automaton that represents all the possible executions of the system that satisfy the constraints of the specification (see Section 5). Once the automaton is built, T-Tool verifies expected behaviors of the system, expressed with **assertion** and **possibility** properties. Whenever an assertion fails, T-Tool reports a counterexample scenario in which the assertion is violated.

Consistency check. This is the simplest form of validation of a specification. It aims to verify that there is *at least* one scenario of the system that respects all the constraints enforced by the requirements specification. If this is not the case, the specification is *inconsistent*. Inconsistencies occur quite often, especially so when the requirements are acquired from different stakeholders.

Assertion validation. The designer can represent expected behaviors of the system through **assertion** properties. Assertions come from different sources. The most important ones are those that represent expectations from the stakeholders ("if all the requirements are met by the system, then I expect this property to be valid"). In general, we are interested in gathering assertions which, although not likely to be verified immediately, are expected to yield interesting counterexamples. Such counterexamples should enable the stakeholders to express their goals with greater accuracy, and drive the elicitation of other goals. Assertion properties can be also specified directly by the engineer in order to check whether she is correctly modeling the intended behavior of the system. For instance, if there are two ways to specify a requirement that seem equivalent, one might be enforced and the other checked. If the two requirements are not equivalent, the behavior that distinguishes them exhibits situations that were not taken into account.

Going back to our example, an important goal of the stakeholders is to avoid "unreasonable" claims, though it is difficult for them to precisely define the concept. Nevertheless, they are able to present particular scenarios that involve such claims. For instance, they do not want to cover claims for which there is no proof (e.g., an invoice) that the

	t_1	t_2	t_3	t_4	t_5
car1.runsOK	$\top$	$\bot$	$\bot$	$\top$	$\top$
cl1.car		car1	car1	car1	car1
cl2.car		car1	car1	car1	car1
cov1.cl		cl1	cl1	cl1	cl1
Fulfilled(cov1)		$\bot$	$\bot$	$\bot$	$\top$
rep1.cl			cl2	cl2	cl2
Fulfilled(rep1)			$\bot$	$\top$	$\top$

Figure 3. An example of a counterexample.

car was repaired. We state this as an assertion: if an instance of CoverDamages for a given claim is fulfilled and the car runs OK, then the stakeholders expect that a repair has been performed for *that* claim.

Dependency CoverDamages
Fulfillment
assertion condition for dependee
$\mathsf{cl.car.runsOK} \rightarrow \exists \mathsf{rep} : \mathsf{RepairCar}$
$(\mathsf{rep.cl} = \mathsf{cl} \wedge \textbf{Fulfilled}(\mathsf{rep}))$

When checking the assertion, the tool exhibits the counterexample shown in Figure 3. The key point is that, when the car stops running OK at time t_2, the customer makes two claims cl1 and cl2, for the *same car* car1, but with different insurance companies. At time t_2, an instance cov1 of goal dependency CoverDamages for cl1 arises. Then, at time t_3, an instance rep1 of RepairCar arises, but is associated to the other claim (cl2). Later, at time t_4, rep1 is fulfilled and the car starts running OK; eventually, at time t_5, cov1 is fulfilled. The problem here is that damages are covered for a certain claim, while the repair is performed for another claim regarding the same car. Indeed, this situation might occur in real life if a customer has policies at two insurance companies. When her car breaks, she can repair it at one garage and attempt to get damage costs from both insurance companies. This is an inadmissible situation, at least in the domain we consider. A preliminary fix for this problem is to require the repairs for a certain claim to be performed before the claim is covered. This can be achieved by adding the following fulfillment condition constraint to the specification of Figure 2.

Dependency CoverDamages
Fulfillment condition for dependee
$\exists \mathsf{rep} : \mathsf{RepairCar}(\mathsf{rep.cl} = \mathsf{cl} \wedge \textbf{Fulfilled}(\mathsf{rep}))$

Possibility check. It is often the case that, even though a specification is consistent, reasonable scenarios are ruled out as valid executions because they are in conflict with some constraints of the system. Therefore, it is important to check that the specification allows for all the scenarios that the stakeholders regard as possible. This kind of scenario is specified in Formal Tropos in terms of **possibility** properties. When performing a *possibility check*, T-Tool verifies that there is *at least one* scenario in which such properties hold. In case of an affirmative answer, the tool presents an *example*, which is a scenario that accounts for the possibility. For instance, the stakeholders may mention the possibility that a car is so damaged that it is impossible to repair. In this case, the insurance company is still responsible for covering damages. This can be specified with a property for dependency CoverDamages that states the **possibility** that after a car breaks, it may never run OK again, as follows:

Dependency CoverDamages
Creation
possibility condition for depender
$\Diamond \textbf{Fulfilled}(\textbf{self}) \wedge \Box \neg \mathsf{cl.car.runsOK}$

The property states that it is possible that some CoverDamages instance be eventually fulfilled even if the associated car never runs OK again after the instance is created. When we perform the possibility check, T-Tool informs the user that the **possibility** does not exist. The check reveals a conflict between the **possibility** property and the **fulfillment** constraint for CoverDamages previously introduced in this section. The problem is that the constraint does not allow the insurance company to cover the damages if a repair has not been performed. If we modify the constraint in order to allow the insurance company to cover damages in case the car never runs OK again, T-Tool returns success for the possibility check.

Dependency CoverDamages
Fulfillment condition for dependee
$\exists \mathsf{rep} : \mathsf{RepairCar}(\mathsf{rep.cl} = \mathsf{cl} \wedge \textbf{Fulfilled}(\mathsf{rep})))$
$\vee \Box \neg \mathsf{cl.car.runsOK}$

Animation of the specification. T-Tool also allows the user to interactively explore the automaton generated from the Formal Tropos specification. Since the automaton exhibits only the sequences of states that respect all the requirements, the user gets immediate feedback on their effects. Though very simple, the animation of requirements is extremely useful to identify missing trivial requirements, which are often taken for granted in an informal setting. For instance, had we forgotten to add the creation condition $\neg$cl.car.runsOk in the specification of RepairCar, we would have obtained histories where the goal of repairing a car arises when the car is running OK. Moreover, the possibility of showing valid evolutions of the system is often an effective way of communicating with the stakeholders.

5. From Formal Tropos to Model Checking

In this section we give technical details on the analysis performed by T-Tool on a Formal Tropos specification (a

more thorough explanation is given in [9]). The first step carried out by T-Tool consists of transforming a given Formal Tropos specification into an equivalent specification in a suitable Intermediate Language. This translation is performed in a completely automatic way, and does not require the user to "operationalize" the specification in order to verify it. The Intermediate Language specification is then passed to the NuSMV model verifier, which performs the actual analysis.

The Intermediate Language. In the Intermediate Language, the strategic flavor of Formal Tropos is lost, and the focus shifts to the dynamic aspects of the system. Some details of the Formal Tropos specification are removed during the translation. This is the case, for instance, for the distinction among the different dependency types. While these aspects are important in the overall description and specification of the system, they do not play any role in the formal analysis described here.

In Figure 4, we give an excerpt of the Intermediate Language translation for our running example. It consists of four parts: *class* declarations, *constraints*, *assertions*, and *possibility* properties.

The *class declarations* (keyword **CLASS**) define the data types of the system; they correspond to the entities, actors, and dependencies (i.e., the outer layer) of the Formal Tropos specification. We note that some new attributes, not present in the Formal Tropos specification, are added to class definitions during the translation. This is the case, for instance, of attribute fulfilledBeInsured of Customer, or attribute fulfilled of dependency CoverDamages. The fact that goals and dependencies have been fulfilled is primitive in Formal Tropos (**Fulfilled** predicate), but is encoded as a state variable in the Intermediate Language; this is an example of the change of focus that occurs when translating a Formal Tropos specification into the Intermediate Language. The Intermediate Language still allows for the dynamic creation of class instances. In Figure 4, predicate **JustCreated** is used to check whether a given instance of a class has been created in the current time instance of a scenario.

Constraint formulas (keyword **CONSTRAINT**) restrict the valid temporal behaviors of the system. Some of these formulas model the semantics of a Formal Tropos specification. For instance, the first two **CONSTRAINT** formulas in Figure 4 express the fact that attribute car of a Claim and attribute cl of a RepairCar are **constant**. Other formulas correspond to the temporal constraints that constitute the inner layer of the Formal Tropos specification. For instance, the third and fourth **CONSTRAINT** formulas in Figure 4 correspond, respectively, to the **creation** and **fulfillment condition** of goal dependency RepairCar, while the last **CONSTRAINT** formula corresponds to the **fulfillment condition** of goal BeInsured. As these formulas are

```
CLASS Claim
  car: Car
CLASS Car
  runsOK: boolean
CLASS Customer
  fulfilledBeInsured: boolean
CLASS InsuranceCo
CLASS Garage
CLASS CoverDamages
  depender: Customer
  dependee: InsuranceCo
  cl: Claim
  fulfilled: boolean
CLASS RepairCar
  depender: Customer
  dependee: Garage
  cl: Claim
  fulfilled: boolean
CONSTRAINT ∀cl : Claim  ∀car : Car
  (cl.car = car → ∘(cl.car = car))
CONSTRAINT ∀rc : RepairCar  ∀cl : Claim
  (rc.cl = cl → ∘(rc.cl = cl))
CONSTRAINT ∀rc : RepairCar
  (JustCreated(rc) → ¬rc.cl.car.runsOK)
CONSTRAINT ∀rc : RepairCar
  ((rc.fulfilled ∧ ¬ • rc.fulfilled) → rc.cl.car.runsOK)
CONSTRAINT ∀cust : Customer
  (cust.fulfilledBeInsured ↔
    (□∧■)(∀cov : CoverDamages
      cov.depender = cust → ◊cov.fulfilled))
ASSERTION ∀cov : CoverDamages
  ((cov.fulfilled ∧ ¬ • cov.fulfilled) →
    (cov.cl.car.runsOK → ∃rep : RepairCar
      (rep.cl = cov.cl ∧ rep.fulfilled)))
POSSIBILITY ∃cov : CoverDamages
  (JustCreated(cov) ∧ ◊cov.fulfilled
    ∧ □¬cov.cl.car.runsOK)
```

Figure 4. Example of Intermediate Language.

no longer syntactically anchored to the creation or fulfillment of a dependency, they need a "context" to define their meaning. This context is provided by the translation rules that map a Formal Tropos specification into an Intermediate Language one. For instance, the fulfillment condition ϕ of a dependency Dep with an **achieve** modality is mapped into a **CONSTRAINT** of the form

$$\forall \mathsf{d} : \mathsf{Dep}\ ((\mathsf{d.fulfilled} \wedge \neg \bullet \mathsf{d.fulfilled}) \rightarrow \phi)$$

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 RepairCar (compare Figures 2 and 4).

As we can see in this translation, we add auxiliary temporal operators to the Intermediate Language specification. These operators depend not only on the kind of formula be-

ing translated, but also on the mode of the dependency. For instance, in the case of a **maintain** dependency, the translation of the fulfillment condition ϕ is given by the rule

$$\forall \mathsf{d} : \mathsf{Dep}\ (\mathsf{d.fulfilled} \rightarrow (\Box \wedge \blacksquare)\phi)$$

stating that "if a **maintain** dependency is fulfilled, then its conditions should hold during the full lifetime of the dependency". In our specification, a similar rule applies for goal BeInsured of the Customer.

The *assertion* and *possibility* formulas (keywords **ASSERTION** and **POSSIBILITY**) state expected properties of the behavior of the system. They correspond to the **assertion** and **possibility** properties of Formal Tropos.

The Intermediate Language plays a fundamental role in bridging the gap between Formal Tropos and formal methods. First of all, it is much more compact than Formal Tropos, and therefore allows for a much simpler formal semantics (see [9] for details). Second, it is rather independent from the particular constructs of Formal Tropos. By moving to different domains, it will probably become necessary to "tune" Formal Tropos, for instance by adding new modalities for the dependencies. The formal approach described in this paper can be also applied to these dialects of Formal Tropos, at the cost of defining a new translation. Furthermore, the Intermediate Language can be applied to requirements languages that are based on a different set of concepts than those of Formal Tropos, such as KAOS [8]. Finally, the Intermediate Language, while more suitable to formal analysis, is still independent from the particular analysis techniques that we employ. For the moment, we have applied only model checking techniques; however, we plan to also apply techniques based on satisfiability or theorem proving.

Model Checking. Starting from an Intermediate Language representation of a Formal Tropos specification, the actual verification is performed within the NuSMV framework. NuSMV [5] is a state-of-the-art model checker based on symbolic representation techniques. Symbolic techniques [3] have been developed to face the well-known state space explosion problem. Model checking is founded on the idea of exploring the whole state space of a finite state machine which describes the possible evolutions of a specification. If that state space is huge, as it is usually the case in real applications, it is impossible to explore it explicitly. Symbolic techniques represent sets of states in terms of boolean propositions and cast the basic operations of model checking algorithms as logical operations on these propositions.

Although symbolic techniques make it possible to analyze large systems, they still assume that the system to be analyzed is finite. In our case, when we pass an Intermediate Language specification to NuSMV, we put an upper bound in the number of instances of each class of entities, actors or dependencies that can be created. The choice of the number of instances is a critical point. In our experiments we have seen that many subtle bugs only appear when more that one instance of a particular class is introduced. Consider for instance the scenario discussed in Section 4, of the customer who presents claims to two different insurance companies for the same accident. Clearly, this scenario requires us to allow for more than one instance of Claim and InsuranceCo in the system. On the other hand, our experiments suggest that bugs usually become evident with just a small number of instances. In particular, in the Insurance Company case study all the mistakes became evident with at most two instances of each class.

Given the Intermediate Language specification and the bounds in the number of instances, the first step performed by the tool is to synthesize a (symbolic) automaton for the specification. The states of this automaton respect the **CLASS** structure of the Intermediate Language specification, and its executions are all and only the executions that respect the **CONSTRAINT** formulas.

NuSMV provides a synthesis algorithm for LTL specifications that is based on a tableau construction technique [7]. In order to deal with the features of the Intermediate Language, we had to extend this algorithm in several directions. For instance, the tableau construction described in [7] and the LTL logics usually exploited in model checking only consider future temporal operators. For early requirements, however, it is also convenient to reason about the past. Therefore, we have extended the tableau construction to deal with the past fragment of LTL. Also, although it is possible to define classes in NuSMV and instantiate them, NuSMV does not allow the creation of new instances at run-time. This is because NuSMV was initially designed to verify hardware systems, where there are no dynamic creations of components. Internally, we model the fact that an instance has been created with special status bit. Moreover, quantifiers are interpreted so that their range is restricted to the instances of a class that exist in the current state.

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. If the specification is consistent, the formal analysis can proceed. Animation of the specification is performed using the simulator provided by NuSMV, which allows both for an interactive exploration of the automaton, and for a random execution of a certain number of steps. Assertion validation and possibility checking are performed using the standard approach of model checking, by verifying the **ASSERTION** and **POSSIBILITY** formulas against the executions of the automaton. Whenever one of these checks fails, the tool reports the failure to the user. In the case of an invalid **ASSERTION**, NuSMV provides a counterexample, which corresponds to a scenario that violates the assertion. For **POSSIBILITY**

formulas, if they hold, T-Tool presents an example which corresponds to a scenario that respects the possibility.

6. Conclusions

We have described a formal modeling language for early requirements and a prototype tool which supports its analysis. An important contribution of this work is to demonstrate that formal analysis techniques are useful during early development phases, such as *early* requirements engineering. 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. Preliminary 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 two bodies of related work that are worth mentioning in this context. Alloy [12] is a language for the modeling and the validation of the structural aspects of a software system. While completely outside the scope of early requirements, Alloy proposes a methodology for applying formal analysis to the early discovery of bugs in a specification that is very similar to ours. For instance, under- and over-specification of a system are identified using expected properties, which are very much like our assertion and possibility properties. KAOS [8, 14] is a framework that supports (early) requirements analysis, but relies on a different methodology and analysis techniques. KAOS relies mostly on theorem proving to support requirements analysis, rather than on model checking. As a consequence, in KAOS [14] the emphasis is on obtaining a formal specification of the goal conflicts that occur in the requirements specification. Our techniques, on the other hand, provide concrete scenarios of these conflicts. While model checking techniques allow for an automatic generation of the scenarios, the formal analysis techniques of [14] may be very expensive.

There are several directions for further research on this project. First, we are working on the application of the methodology to more complex case studies, which should give an extensive evaluation of the scalability of our methodology to real applications, and to domains where a large number of instances is necessary for the verification. Second, we are working in extending the formal verification tool. So far, we have mostly adapted verification techniques of NuSMV to the new domain; however, there is much work to be done on formal methods techniques specifically tailored to requirements engineering. For instance, we should enhance the animator of the specifications. At the moment, NuSMV allows for an exploration of the evolution of the system in terms of the Intermediate Language specification, and represents traces in a tabular format similar to the one of Figure 3. We are investigating different ways to extend T-Tool, so that traces are presented at the Formal Tropos abstraction level and the executions of the system are proposed in a form convenient for the user. Finally, we will investigate the possibility of applying some of the techniques of the KAOS framework to Formal Tropos, such as goal decomposition and operationalization.

References

[1] A. Anton. Goal based requirements analysis. In *Proc. ICRE'96*, 1996.

[2] J. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, 1993.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[4] J. Castro, M. Kolp, and J. Mylopoulos. A requirements-driven development methodology. In *Proc CAiSE'01*, 2001.

[5] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Checker. *Int. Journal on Software Tools for Technology Transfer (STTT)*. 2(4):410–425, 2000.

[6] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso. Formal Verification of a Railway Interlocking System using Model Checking. *Journal on Formal Aspects in Computing*, 10:361–380, 1998.

[7] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):57–71, 1997.

[8] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.

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

[10] J. Halpern and M. Vardi. Model checking vs. theorem proving: A manifesto. In *Proc. KR'91*, 1991.

[11] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. on Software Eng. and Methodology*, 5(3):231–261, 1996.

[12] D. Jackson. Alloy: A lightweight object modelling notation. Technical report, MIT, July 2000.

[13] J. Spivey. *The Z Notation.* Prentice Hall, 1989.

[14] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Transaction on Software Engineering*, 1998.

[15] E. Yu. Towards modelling and reasoning support for early-phase requirements engineering. *Proc. RE'97*, 1997.

[16] E. Yu and J. Mylopoulos. Towards modelling strategic actor relationships for information systems development – with examples from business process reengineering. In *Proc. 4th Workshop on Information Technologies and Systems*, 1994.