Model Checking Early Requirements Specifications in Tropos

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

Marco Pistore
IRST-ITC
pistore@itc.it

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

Paolo Traverso
IRST-ITC
traverso@itc.it

Abstract

The paper describes an approach that bridges the gap between early requirements specifications and formal methods. In particular, we propose a new specification language, called Formal Tropos, *that offers the primitive concepts of early requirements frameworks (actor, goal, strategic dependency) [13], but supplements them with a rich temporal specification language. We also extend existing formal analysis techniques, and in particular* model checking, *to allow for an automatic verification of relevant properties of the early requirements specification. Our preliminary experiments show 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 trying to understand the organizational context for an information system, and the goals and social dependencies of its stakeholders. This phase demands critical interactions with the users; a misunderstanding at this point may lead to expensive errors during later development stages. Not surprisingly, several approaches have been researched in recent years on suitable concepts, languages and analysis techniques specifically tailored for this phase (e.g., [8, 13, 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, in certain fields, they are even becoming integral components of standards [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, e.g. the design phase (see for instance [6]). As a result, there is a mismatch between the concepts used for early requirements specifications (such as goal, actor ...) and the constructs of formal specification languages such as Z [11], SCR [10], etc.

Our aim is to provide a framework for the effective use of formal methods in the early requirements phase. The framework allows 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. We achieve these results by extending and formalizing an existing early requirements modeling language, and by building on state-of-the-art formal verification techniques.

In order to allow for formal analysis, we extend the i* modeling language [13] into a formal specification language called *Formal Tropos*. 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].

In order to support formal analysis within the Formal Tropos framework, we extend an existing formal verification technique, model checking [9]. To accomplish this, we define an intermediate language that serves as a link between Formal Tropos and model checking. Tropos specifications can be mapped into the intermediate language, which is amenable to model checking analysis. As a result, early requirements specifications can be checked for contradictions (consistency checking), and properties that are supposed to hold can be verified on the specification (property validation). Moreover, it is possible to use model checking to actually animate Tropos specifications.

Our prototype implementation of the above ideas comes in the form of a tool which translates automatically early requirements written in Tropos into the intermediate language. In addition, the tool extends NuSMV [5], a state-of-the-art symbolic model checker, to do analysis on the intermediate language, including consistency checking, property validation, and animation. We have experimented with the proposed framework and the supporting tool, using a simple case study. In spite of its simplicity, the case study demonstrates the benefits of formal analysis in revealing incompleteness/inconsistencies errors that are by no means trivial to discover in an informal setting.

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.

Structure of the paper. In Section 2 we present 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 process and the tool. Finally, Section 6 presents some concluding remarks and discusses future research directions.

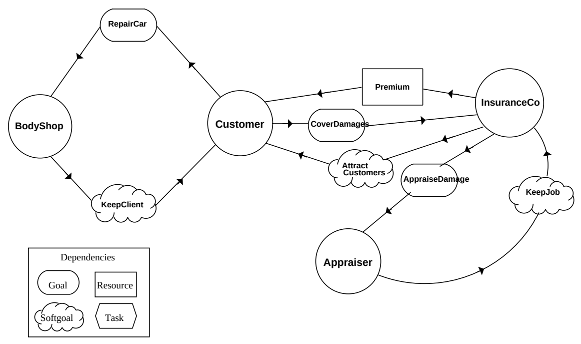

Figure 1. SD diagram of the case study.

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 necessary to 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 graphical notations to describe the requirements of the system. The SD diagram, for instance, is used to represent a central concept of i*: the *strategic dependencies* of the actors. 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 (for instance, it is a matter of personal feeling, or the fulfillment 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 to the depender.

The details on i* are presented in [13]; we will briefly review it by using the Insurance Company case study, initially introduced in [14]. 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 `BodyShop` and `Appraiser`, and additional dependencies. For instance, the `Customer` depends on the `BodyShop` 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 presents an i* SD diagram for the case study. We will use a subset of this case study as the running example for the rest of the paper. In particular, we will focus on the `CoverDamages` and `RepairCar` dependencies. The formal specification of this subset is given in Figure 2.

3. The Formal Tropos Language

A Formal Tropos specification describes the relevant objects of the modeled domains and their relationships. The description of each object is structured in two layers. The outer layer is similar to a class declaration, since it defines the structure of the instances together with their attributes. The inner layer expresses constraints on the lifetime of the object, given in a first-order linear-time temporal logic.

A class can be of type *actor*, *dependency*, or *entity*. The notion of entity is not included in i*. Entities are used to represent non-intentional elements that exist in the environment or organizational setting, but are not directly relevant to an actor’s strategic goals. In our example (see Figure 2) we have the entities `Claim` and `Car`.

The attributes of a Formal Tropos class denote relationships among the different objects being modeled. For example, each claim generated 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 be kept forever. Formal Tropos defines other attribute facets, like **multivalued** and **optional**, which do not appear in our example.

As in i*, actors in Formal Tropos can have *goals* that describe their strategic interests. This is the case of goal `BeInsured` of actor `Customer`. Also, the intentional relationships between 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

```
Entity Claim
  Attribute constant car: Car
Entity Car
  Attribute runsOK: boolean
Actor InsuranceCo
Actor BodyShop
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 BodyShop
  Attribute constant cl : Claim
  Creation
    condition $\neg cl.car.runsOK$
  Fulfillment
    condition for depender $cl.car.runsOK$
```

Figure 2. The Formal Tropos specification.

dependee are included as special attributes in the dependency declaration.

Goals and dependencies can be fulfilled in different *mode*s. For example, the modality of `RepairCar` is **achieve**, which means that an instance of the dependency is satisfied if the car is eventually repaired at least *once*. Dependency `CoverDamage` is also an **achieve** dependency, since it is satisfied as soon as the insurance company reimburses the damages. Goal `BeInsured` of agent `Customer`, instead, is of **maintain** modality: the customer will stay insured forever, not just once. There are other modalities, such as **achieve&maintain**, which is a combination of the previous two modes (it requires the fulfillment conditions to be achieved at some point and then satisfied in a continuing way); and **avoid**, which means that the fulfillment condition should be prevented.

The inner layer of a Formal Tropos class declaration describes constraints on the possible evolution of the instances of that class. In the case of a dependency declaration we have three types of constraints. **Creation** assertions impose constraints at the time of creation of a new instance of the dependency; **fulfillment** assertions define conditions for the satisfaction of the dependency; and **invariants** represent conditions that should be true during the whole life of the dependency. We also distinguish formulas that express sufficient conditions (facet **trigger**), necessary conditions (facet **condition**), and necessary-and-sufficient conditions (facet **definition**) for the event.

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 is created; its creation trigger represents the fact that, whenever a customer fills a claim cl (formula $JustCreated(cl)$), then a dependency for covering the repairs arises from the customer to its insurance company. 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 body-shop would be happy to declare a car repaired even if it does not run); in Formal Tropos, we represent this fact with the facet **for depender**.

Not only dependencies, but also entities and actors of a Tropos specification may have creation conditions and invariants; however, they cannot have fulfillment conditions. Actor goals do have fulfillment conditions, but do not have creation conditions (they are assumed to be there with their owning actor) or invariants (the actor goals do not have attributes).

Constraints on the lifetime of the class instances (i.e., the inner layer of a Formal Tropos specification) are given in a first-order linear-time temporal logic. In the formulas we have the usual first-order logic quantifies $\forall$ and $\exists$, that can range over all the instances of a given class. As Formal Tropos allows for the dynamic creation of new instances of a given entity, the range of a quantifier depends on the state in which the quantifier is evaluated. So, in the fulfillment definition for goal `BeInsured` of Figure 2, quantifier $\forall cov : CoverDamages \ldots$ binds variable cov to range over all the instances of class `CoverDamages` that exist in the system. The logic formulas that appear inside a given class declaration of the Tropos specification may refer to the attributes of the class via their names. Also, each instance of the class may express properties about itself using 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 $\Box f$ (always in the future f) expresses the fact that formula f should hold in the current state and in all the future states of the evolution of the system. Formula $\blacklozenge f$ (sometimes in the past), holds if f 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 enough 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, then it will eventually be fulfilled (formula $\Diamond Fulfilled(cov)$); since the goal is a **maintain** goal, this property has to hold in a continuous way for the goal to be fulfilled.

The conditions in our example are *required* to hold for all models of the specification. In general, we distinguish two sets of formulas in the specification: those that are enforced on the system and those that express *desired* behaviors. The formulas of the former set express facts on the behavior of the system that we assume to hold. The formulas in the latter set express expectations (denoted by the facet **assertion**) on the behaviors of the system. In the next section, we will show how we check whether these assertions are satisfied by the specification.

Besides the class declarations, a Formal Tropos specification is completed by *system-level formulas* that describe properties of the system as a whole. We distinguish among three kinds of system formulas. First, they may be invariants that are *required* to hold in all states of the system (keyword **System invariant**). Second, they may be *desired* properties of *all* the executions of the system (keyword **System assertion**). Finally, they may express desired properties that are expected to hold on *some* possible behaviors of the system (keyword **System possibility**); they describe scenarios that are expected to be compatible with the requirements. In the next section we will see an example of the latter kind of system level formulas.

4. Formal Analysis

Formal analysis techniques are usually applied during late phases of development, and are used to validate the implementation of a system against its requirements. In the (early) requirements phase, however, the aim must be different, since, in fact, we are still gathering the requirements! Nevertheless, formal analysis techniques can assist the analyst in the requirements elicitation, by allowing her to identify errors and limitations of the specification that are not evident in an informal setting.

We developed a tool that, starting from the Tropos specification, builds an automaton that represents all the possible executions of the system that satisfy the requirements enforced in the specification (see Section 5). Once the automaton is built, our tool verifies the expected behaviors of the system, expressed via **assertion** and **possibility** formulas. Whenever a verification fails, a counterexample is reported, that is, a scenario where the expected property failure is shown to the users.

Property validation. The designer can represent expected behaviors of the system via **assertion** properties in the Formal Tropos specification. The assertions come from different sources. First, they may represent expectations of the stakeholder ("if all the requirements are met by the system, then I expect that this situation never happens"). Second, they may be formulas included 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 by the requirements engineer. Finally, it is often useful to add simple properties to catch errors due to the inherent difficulty of writing formal specifications.

The first kind of assertions, the expected outcomes from the stakeholder, is the most important. 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 elicit their requirements with greater accuracy, and drive the refinement of the formal specification.

In 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 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 done for *that* claim.

	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.

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

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 `CoverDamages` is also created and associated to `cl1`. Then, at time t_3, an instance `rep1` of `RepairCar` is created and 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 done for another claim but for the same car. 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 body-shop and attempt to get damage costs from both insurance companies, clearly 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 done before the claim is fulfilled. This can be achieved by adding the following fulfillment condition to the specification of Figure 2.

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

Consistency check. The aim of the *consistency check* is to verify that the specification allows for a set of behaviors that are possible in the system.

As a simple form of consistency check, we can ask whether there is any execution of the system that respects all the constraints enforced by the requirements; if this is not the case, the specification is *inconsistent*. Inconsistencies among the requirements occur quite often, in particular if the requirements are obtained from different stakeholders, so it is important to identify and eventually solve them. As inconsistencies can be caused by the the interplay of different objects in the specification, it is often hard to detect them without the support of automatic analysis techniques.

Besides checking that the specifications are consistent, it is important to check that they allow for some execution histories (or scenarios) that the stakeholder expects to happen. It is often the case that a specification is consistent, but some reasonable scenarios are ruled out as valid executions because they are in conflict with some requirements.

For instance, suppose we would like to check that after a car breaks, it may never run OK again, but the damages are nevertheless covered by the insurance company. This is sensible, since a car might be so damaged that it is impossible to repair. This check is specified as the following system-level formula:

System possibility $\exists cov : CoverDamages$
$$JustCreated(cov) \wedge \Diamond Fulfilled(cov)$$
$$\wedge \Box \neg cov.cl.car.runsOK$$

This formula asserts that it is possible that some `CoverDamages` cov is eventually fulfilled even if the associated car never runs OK again after dependency cov has been raised. The consistency check for the resulting specification fails. Indeed, there is a global inconsistency between the system formula and the fulfillment condition for `CoverDamages` introduced previously in this Section. The problem is that the fulfillment condition does not allow the insurance company to cover the damages if a repair has not been performed. The consistency check succeeds if we fix that fulfillment condition and allow the damages to be covered by the insurance company in the case the car never runs OK again.

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

Animation of the specification. Our tool allows the user to interactively explore the automaton generated from the early requirements specification. Since the automaton exhibits only those sequences of states that respect all the requirements, the user gets immediate feedback on their effects.

While very simple, the animation of requirements is extremely useful to identify missing trivial requirements, which are often assumed for granted in an informal setting. For instance, if we had forgotten to add the creation condition $\neg cl.car.runsOk$ in the specification of `RepairCar`, by exploring the system we would have obtained histories where the goal of repairing a car arises when the car is running OK. Moreover, the possibility of showing possible evolutions of the system is often a very effective way of communicating with the stakeholders.

5. From Formal Tropos to Model Checking

In this section we describe the technicalities of the formal analysis performed on a Tropos specification.

The first step carried out by the tool consists of transforming a given Formal Tropos specification into an equivalent specification in a suitable Intermediate Language. During this translation, the strategic flavor of the Formal Tropos specification is lost and the focus shifts to the dynamic aspects of the system. [1] This Intermediate Language specification is then passed to the NuSMV model verifier, which synthesizes an automaton for the specification and performs the actual analysis.

5.1. The Intermediate Language

We start by giving in Figure 4 an excerpt of the Intermediate Language specification for our running example. It consists of four parts: “class” declarations, temporal “specifications”, “assertion” formulas and “possibility” formulas.

The *class declarations* (keyword **CLASS**) define the data types of the system; they correspond to the entities, actors, and dependencies (the outer layer) of the Formal Tropos specification. We remark that some new attributes, not present in the Formal Tropos specification, are added to the classes during the translation. This is the case, for instance, of attribute `fulfilledBeInsured` of `Customer`, or of 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 from Formal Tropos specification into the Intermediate Language.

The *temporal specifications* (keyword **SPECIFICATION**) 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 **SPECIFICATION** formulas in Figure 4 express the fact that attribute car of a $Claim$ and attribute $claim$ 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 **SPECIFICATION** formulas in Figure 4 correspond, respectively, to the **creation** and **fulfillment condition** of goal dependency `RepairCar`; and the last **SPECIFICATION** formula corresponds to the **fulfillment condition** of goal `BeInsured`. As these formulas are no longer syntactically anchored to a particular event of the specification, (e.g., the fulfillment of the dependency), they need a “context” to define their meaning. This context is defined in the translation rules that map a Tropos specification into an Intermediate Language specification. For instance, the fulfillment condition ϕ of a dependency Dep with an **achieve** modality is mapped into a **SPECIFICATION** of the form

$$\forall d : Dep\ (d.fulfilled \rightarrow \blacklozenge\phi)$$

meaning that “if an achieve dependency is fulfilled, then its fulfillment condition was true at least once in the past”. 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 IL specification. These operators depend not only on the kind of formula but also on the mode of the dependency. For instance, in the case of a **maintain** depen-

CLASS Claim
  car: Car
CLASS Car
  runsOK: boolean
CLASS Customer
  fulfilledBeInsured: boolean
CLASS InsuranceCo
CLASS BodyShop
CLASS CoverDamages
  depender: Customer
  dependee: InsuranceCo
  cl: Claim
  fulfilled: boolean
CLASS RepairCar
  depender: Customer
  dependee: BodyShop
  cl: Claim
  fulfilled: boolean
SPECIFICATION $\forall cl : Claim \quad \forall car : Car$
  $(cl.car = car \rightarrow \circ(cl.car = car))$
SPECIFICATION $\forall rc : RepairCar \quad \forall cl : Claim$
  $(rc.cl = cl \rightarrow \circ(rc.cl = cl))$
SPECIFICATION $\forall rc : RepairCar$
  $(JustCreated(rc) \rightarrow \neg rc.cl.car.runsOK)$
SPECIFICATION $\forall rc : RepairCar$
  $(rc.fulfilled \rightarrow \blacklozenge rc.cl.car.runsOK)$
SPECIFICATION $\forall cust : Customer$
  $(cust.fulfilledBeInsured \leftrightarrow$
    $(\square\wedge\blacksquare)(\forall cov : CoverDamages$
      $cov.depender = cust \rightarrow \lozenge cov.fulfilled))$
ASSERTION $\forall cov : CoverDamages$
  $(cov.fulfilled \rightarrow$
    $\blacklozenge(cov.cl.car.runsOK \rightarrow \exists rep : RepairCar$
      $(rep.cl = cov.cl \wedge rep.fulfilled)))$
POSSIBILITY $\exists cov : CoverDamages$
  $(JustCreated(cov) \wedge \lozenge cov.fulfilled$
    $\wedge\ \square\neg cov.cl.car.runsOK)$

Figure 4. Example of Intermediate Language.

[1] We remark that the translation from the Tropos specification to the Intermediate Language specification is performed in a completely automatic way by the tool: we do not require the user to “operationalize” the specification in order to verify it.

dency, the translation of the fulfillment condition ϕ is given by rule

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

meaning that "if a maintain dependency is fulfilled, then its conditions should hold during the whole life of the dependency". In our specification, a similar rule is applied to goal `BeInsured` of the `Customer`.

The *assertion* and *possibility* formulas (keywords **ASSERTION** and **POSSIBILITY**) specify expected properties of the behavior of the system. The former correspond to the **assertion** formulas of Formal Tropos, and they are translated in a similar way as temporal specification formulas. The latter correspond to the **System possibility** formulas of the Formal Tropos specification.

We remark that some of the details of the Formal Tropos specification are lost in the corresponding Intermediate Language specification; this is the case, for instance, of 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, and so they are discarded when moving to the Intermediate Language.

The Intermediate Language plays a fundamental role in covering the gap between early requirements and formal methods. First of all, it is much smaller than Formal Tropos, and therefore allows for a much simpler formal semantics.[2] Second, it is rather independent from the particularities 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 Tropos, at the cost of defining a new translation. Furthermore, the Intermediate Language can be applied to requirements languages that are based on different concepts from the ones of Tropos, such as KAOS [8].

Finally, the Intermediate Language, while more suitable to formal analysis, is still independent from the particular analysis techniques. For the moment, we have applied only model checking techniques; however, we plan to apply techniques based on LTL-satisfiability or theorem proving.

5.2. Model Checking

Starting from an Intermediate Language representation of the Tropos specification, the actual verification is performed on top of the NuSMV verification framework. NuSMV [5] is a state-of-the-art model checker based on a symbolic representation of the domain to be verified. Symbolic techniques [3] have been developed to face the well-known state explosion problem. When performing model checking, it is necessary to explore the states of the system; if the system is huge, as it is usually the case in real applications, it is impossible to explore it explicitly. Symbolic techniques allow for representing sets of states via boolean propositions and for casting the basic operations of model checking algorithms as logical operations on these formulas. In this way, it is not necessary to enumerate the states explicitly.

Although they make it possible to analyze large systems, the techniques provided by NuSMV still require the system to be finite. In our case, the consequence is that we have to put an upper bound in the number of instances of each class of entities, actors or dependencies that can be created in the system. We do this by declaring these upper bounds in the Intermediate Language specification.

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 the classes is allowed in the system. Consider for instance the scenario, discussed in Section 4, of the customer that presents claims to two different insurance companies for the same repair; 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 also show 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 just 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 the (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 **SPECIFICATION** formulas.

NuSMV provides a synthesis algorithm for LTL specifications, that is based on a tableau construction technique [7]. In order to deal with the particularities of the Intermediate Language, we had to extend the algorithm in some directions. For instance, the tableau construction described in [7], and the LTL logics usually exploited in model checking, only consider future temporal operators. In the early requirements specification, instead, 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 that can be instantiated, it 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. To deal with instance creation, we have adapted the tableau construction and other routines of NuSMV. Internally, the fact that an instance has

[2]For lack of space, we do not present the formal semantics of the Intermediate Language in the details. In brief, the semantics is defined using standard techniques for interpreting LTL specifications on domains with an algebraic structure of states.

been created is modeled by a special bit of its status; the 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 the inconsistency check. In fact, if the specifications are inconsistent, the synthesis process fails and no automaton is built. If the specifications are consistent, instead, the formal analysis can proceed. The animation of the specifications 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 in the system. Consistency check and property validation 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. A counterexample is provided also in the case of an invalid **POSSIBILITY**. In this case, by definition, all the executions of the model do not satisfy the possibility formula; nevertheless, the returned counterexample should help the user to understand and identify the problem.

6. Conclusions

We have described a formal modeling language for early requirements and a prototype tool which supports the analysis of specifications. The novelty of our approach lies in extending model checking techniques — which rely mostly on design-inspired specification languages — so that they can be used with an expressive modeling language suitable for early requirements modeling and analysis. The contribution of our preliminary results is to show that formal analysis techniques are useful in development phases that were once considered to be informal by nature, as is the case of *early* requirements engineering.

Our proposal complements analysis techniques proposed in the KAOS project, which rely mostly on theorem proving to support requirements analysis [8, 12]. In [12], for instance, the emphasis is put on obtaining a formal specification of the goal conflicts that occur in the requirements specification; our techniques, instead, provide concrete scenarios of these conflicts. While model checking techniques allow for an automatic generation of the scenarios, the formal analysis techniques of [12] 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 exact evaluation of the scalability of our methodology to real applications. 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 for requirements engineering. For instance, we should enhance the animator of the specifications. At the moment, NuSMV represents the evolution of the system in a tabular format similar to the one of Figure 3; we are investigating different ways to make the traces produced by the animator more readable to the engineer. Third, we will investigate on the possibility of applying some of the techniques of the KAOS framework to Formal Tropos. Finally, a promising direction of future research is the support for other forms of analysis, such as checking the validity of goal decompositions or incorporating requirements traceability techniques.

References

[1] A. Anton. Goal based requirements analysis. In *Proc. 2nd Int. Conf. on Requirements Engineering 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, July 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, June 1992.

[4] J. Castro, M. Kolp, and J. Mylopoulos. A Requirements-Driven Development Methodology. *13th Int. Conf. on Advanced Information Systems Engineering (CAiSE'01)*, June 2001. To appear.

[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)*. To appear.

[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, February 1997.

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

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

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

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

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

[13] E. Yu. Towards modelling and reasoning support for early-phase requirements engineering. *Proc. 3rd IEEE Int. Symp. on Requirements Engineering RE'97*, pages 226–235, January 1997.

[14] 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.