Formal Analysis of Early Requirements Specifications

by

Ariel Damián Fuxman

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Formal Analysis of Early Requirements Specifications

Ariel Damián Fuxman

Master of Science

Graduate Department of Computer Science

University of Toronto

2001

Early Requirements Engineering is the phase of the software development process that is concerned with understanding the organizational context of a system, and the goals and social dependencies of its stakeholders. *Formal Methods* techniques have a great potential as a powerful means for specification, early debugging and certification of software. So far, however, their application to early requirements modeling and analysis has been limited by a mismatch between the concepts used for describing early requirements and the constructs offered by formal specification languages.

This thesis describes an approach that bridges the gap between Early Requirements Engineering 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), and supplements them with a rich temporal specification language. We also developed a tool, called *T-Tool*, that is able to perform formal analysis on the *Formal Tropos* specifications. The tool is based on Formal Methods techniques, in particular *model checking*.

Our preliminary experiments on a case study show that formal analysis techniques reveal gaps and inconsistencies in early requirements that are by no means trivial to discover without the help of formal analysis tools.

Acknowledgements

First of all, I would like to thank my supervisor, Prof. John Mylopoulos, for his support and guidance during all this process. His patience and advice were fundamental for me in the moments I was unsure of what direction to take for the thesis. I am also very grateful to Marco Pistore, with whom I had the opportunity to work during his stay in Toronto. He read several drafts of this thesis, and gave excellent suggestions on how to improve it. I also thank the members of the *Tropos Project* and *Formal Methods* groups at University of Toronto for the interesting discussions and feedback. Finally, I thank Prof. Steve Easterbrook for accepting to be the second reviewer of this thesis.

From a personal side, I would like to thank my friends from Argentina, and the new ones I made in Canada, for their continuous encouragement. In particular, I thank Sebastián Sardiña for helping me to settle in Toronto when I arrived two years ago.

Last, but definitely not least, I would like to thank my parents (Silvia and Miguel) and my brothers (Adrián and Diego) for their unconditional love and support.

Contents

Chapter 1

Introduction

Requirements engineering is the process of identifying the purpose of a software system, and documenting it in a form that is amenable to analysis, communication and subsequent implementation [NE00]. The importance of software requirements has long been recognized in the Software Engineering community. For instance, it has been estimated that an error that is not identified and corrected during the requirements phase can cost as much as two hundred times more to correct in subsequent phases [Boe81].

Much research in Requirements Engineering has been based on the underlying assumption that an initial statement of the requirements is always available. Considerably less attention has been given to the *early phase* of Requirements Engineering [Yu95], which encompasses the activities that precede the formulation of the initial requirements. This phase is primarily concerned with understanding the organizational context of a system, and the *goals* and *social dependencies* of its stakeholders. The emphasis is put on *why* the requirements arise rather than on *what* requirements should be met.

Formal Methods have a great potential as a powerful means for specification, early debugging and certification of software. They have been successfully applied in several industrial applications, and, in certain domains, they are even becoming integral components of standards [BS93]. However, the application of formal methods to Requirements

Engineering is by no means trivial. Most formal techniques have been designed to work (and have been mainly applied) to the design (not requirements) phase of software engineering (see for instance [CGM+98]). As a result, there is a mismatch between the concepts used for early requirements specification (goal, actor, etc.) and the constructs of formal specification languages such as Z [Spi92], SCR [HJL96], etc.

This thesis aims to provide a framework for the effective use of formal methods in the early phase of Requirements Engineering. The framework allows for the formal and mechanized analysis of early requirements specifications expressed in a formal modeling language. 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. Our contribution is twofold: we bring formality to Early Requirements Engineering, and we introduce a new application domain for Formal Methods. In particular, the contributions of this thesis include:

- Extension of an existing early requirements modeling language, i* [Yu95], 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 the KAOS framework[DvLF93].

- Adaptation of an existing formal verification technique, model checking [CGP99], so that it can be applied to early requirements specifications.

- Definition of an intermediate language based on linear time temporal logic that serves as a link between Formal Tropos and model checking.

- Development of a tool, called *T-Tool*, based on the state-of-the-art symbolic model checker NuSMV [CCGR00]. This tool is able to perform formal analysis on the intermediate language representation of a Tropos specification, including *non-emptiness checking*, *assertion validation*, and *animation*.

- Experiment 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/inconsistency errors that are by no means trivial to discover in an informal setting.

The rest of the thesis is organized as follows. Chapter 2 reviews research from the Requirements Engineering and Formal Methods communities that is relevant to this thesis. It also gives an introduction to the i* framework, which will be used as a starting point for Formal Tropos. Chapter 3 presents the Formal Tropos language and illustrates its use in the context of our case study. In Chapter 4 we introduce some of the formal analysis techniques that the engineer can perform within the proposed framework. The next chapters concentrate on the technical details of our approach. Chapter 5 describes the Intermediate Language and the mapping from Formal Tropos specifications; Chapter 6 presents T-Tool, with special emphasis on the adaptation of model checking techniques for our purposes. Finally, Chapter 7 offers some concluding remarks and discusses directions for future research.

Chapter 2

Literature Review

2.1 Requirements Engineering

In the late seventies, several techniques were developed for modelling functional requirements for software systems. Their ontologies were *function-oriented*, based on concepts such as *data* and *operations*. Among the most widely-used techniques were the Entity-Relationship Diagram [Che75] for data modeling, and Structured Analysis [DeM78] for describing operations. SADT [Ros77], introduced by Ross and Schoman in 1977, was probably the first language for modeling requirements. It was based on a data/operation duality principle: data was defined by producing/consuming operations; and operations were defined by their input/output data.

A disadvantage of this first family of languages is that the resulting specifications were vague and imprecise. The languages provided a semi-formal syntax and a completely informal semantics [Dub89]. For instance, in SADT, it was possible to give a formal declaration of the data and operations, but their constraints could only be asserted in natural language.

RML [GBM86] was the first requirements modeling language to have a formal semantics. It was based on three conceptual units: entities, operations, and constraints. The latter were expressed in a formal assertion language with built-in support for temporal referencing. The formal semantics was given in terms of mappings to first-order predicate calculus. This approach of giving formal semantics by mapping to a logic language is also taken in this thesis for Formal Tropos, though we map to a different logic (a first order lineal time temporal logic).

RML can also be considered as a precursor of object-oriented analysis techniques, as it introduced new structuring mechanisms such as generalization, aggregation and classification. Object-oriented analysis techniques, with UML [BRJ99] emerging as an standard, are the current state-of-the-practice in Requirements Engineering. However, it is important to realize that their ontology is not significantly different from that of the first requirements modeling languages, and that they lack a formal semantics.

The next generation of requirements languages is based on richer ontologies. A line of research has addressed the fact that systems and their environment are made of active components, which interact with each other. An appropriate model should identify appropriate *assumptions* on the environmental components, and *requirements* on the system components. The languages based on this ontology are called *agent-oriented* [Fea87]. Notice, however, that the term “agent” is used in a different sense than in Artificial Intelligence, where agents are characterized by their desires and intentionality [CL90].

Albert II [Boi95] is a requirements language based on these principles. It provides a graphical notation for declaring the agents of a system and the actions they are capable of; and a textual notation for declaring constraints on their behavior. Its semantics is defined in terms of an Intermediate Language based on temporal logic; and it is amenable to formal analysis, in particular animation [EDD94]. However, the resulting descriptions are not abstract enough to be useful in the early phases of requirement engineering. Furthermore, the resulting approach does not provide support for automated verification, at least in the way it is addressed in our work.

Although the need for focusing on *why* requirements arise has always been recognized, it was not until recently that research was conducted on *goal-oriented* ontologies [MCY99]. The NFR framework [MCN92] deals with goals related to non-functional requirements, and provides heuristics for their identification, and algorithms for their qualitative evaluation. Anton proposes a goal-based method called Goal-Based Requirements Analysis Method (GBRAM) and provides heuristics and procedural guidance for the initial elicitation of goals, but they do not provide a modelling language. The KAOS [DvLF93] framework of Laamswerde et. al. does provide a model for goal-oriented requirements. It consists of an informal layer for declaring concepts (such as goal, entity, etc.) and their relationships; and a formal layer for expressing constraints on those concepts in a typed first-order temporal logic with real-time constraints. Goal refinements are captured through AND/OR graph structures. It is possible to prove whether a refinement is correct and complete [DvL96], and formally detect conflicts among goals [vLDL98, vLL00]. It also allows the specification of agents, the *operationalization* of goals into actions, and the assignment of actions to agents.

Goals are sometimes hard to elicit. Before they gain a clear understanding of the system, stakeholders are sometimes more comfortable reasoning on operational scenarios. *Scenarios* are basically temporal sequences of interaction events between the system and the agents of the environment. It is important to remark that most efforts for automated analysis within the Requirements Engineering community have been done in conjunction with scenario validation, using techniques such as animation [EDD94], planning [FH92] and test case generation [HSG+94].

Scenarios can be useful to discover goals and validate goal specifications. Anton [ADS00] has worked on goal-based elicitation in conjunction with scenarios, but mainly at an informal level; and KAOS offers procedural guidance for inferring high level goals from operational scenarios [vLW98]. However, none of these techniques can be automated. In contrast, the approach that we present in this thesis for Formal Tropos offers a methodology that links scenarios to goal specifications and provides fully automated support based on model checking techniques.

2.2 Formal Methods and Model Checking

Unlike the Requirements Engineering community, the Formal Methods research area is specifically concerned with developing models with formal semantics, and useful for further analysis. In this thesis, we rely on *model checking* [CGP99] as our analysis technique. Model checking has a number of advantages over traditional approaches to automated verification, such as theorem proving. In particular, it has a potential for fully automatic, "push-button" verification. Furthermore, it is able to produce counterexamples of violated properties, which are extremely useful for debugging a specification.

Most formal techniques have been designed to work in the later phases of software development, such as architectural [AG94] and detailed design [CGM+98]. As a result, they offer an ontology that is not appropriate for modeling requirements. However, there has been work on the application of automated verification to requirements modeling. The most notable is by Heitmeyer et. al. [HJL96] on tool support for the SCR tabular notation for requirements specification [Hen80]. The toolset includes consistency/completeness checking, model checking, theorem proving and animation. Another relevant work is the RSML [LHH94] language, which provides more structuring mechanisms than SCR, and a tool for checking the completeness and consistency of specifications.

The previous approaches are restricted to the particular domain of embedded systems and process control. Furthermore, they do not support the goal- or agent-oriented ontologies suitable for describing early requirements. In contrast, it is worth mentioning a work that applies concepts from the area of Knowledge Representation to the formal analysis of early requirements specifications [Wan01]. In particular, they explore the combined use of the i* early requirements language and the ConGolog [GLL99] logic framework. They propose an annotation for the i* Strategic Rationale diagrams, based on ConGolog's synchronization primitives (sequence, alternative, concurrency, etc.) and control structures (while-loop, for-loop, iteration, etc.), and a mapping from these annotated models into ConGolog. Since a ConGolog specification is based on a set of completely specified processes (though non-determinism is allowed), they require the initial requirements models to be operationalized at an important extent.

The semantics of ConGolog is based on a powerful logical theory, the *situation calculus* [MH79], and provides a framework for formally proving assertions on the specification. However, it relies on theorem proving, rather than model checking, which makes the prospects of automated verification more challenging. In fact, [Wan01] only attempts to simulate the models using an interpreter, and does not provide support for automated reasoning.

2.3 Early Requirements Engineering

In [Yu95], a distinction is made between *early* and *late* requirements. Since early requirements arise at the very beginning of the RE process, they are more concerned with the goals of the organization in which the system is embedded than with setting the basis for the future implementation of a system. Even more, the techniques of the early requirements phase can even be used when no automated system is envisioned, and the only objective is to understand an organization *as it is*.

There are a number of techniques for eliciting organizational requirements. Traditional techniques include questionnaires, surveys, interviews, and brainstorming. There are also *cognitive* techniques, originally developed for knowledge acquisition in knowledge-based systems [SG96], and scenario-based techniques [PTA94].

However, for a long time there was no modelling language available for this kind of requirements, and the result of the elicitation process was typically expressed in natural language. It was not until recently that early requirements modeling has started receiving the attention it deserves in the research community [Yu95]. There is a clear need for early requirements modelling languages, and also for elicitation techniques capable of mapping requirements into such languages.

Modeling early requirements brings about a number of issues that are not present in later phases. Agents at this level of specification are the actual stakeholders of the organization, and therefore their intentionality and high level goals should be taken into account. Furthermore, it should also be possible to allow for some kind of reasoning even in the absence of detailed information.

In general, agent- and goal-oriented ontologies are appropriate for this phase. However, some existing goal- or agent-oriented languages do not provide suitable abstractions for describing early requirements. This is the case, for instance, of the Albert II language, which forces the stakeholder to think in terms of fine-grained actions which are usually not available at this early stage of analysis. On the other hand, the KAOS framework is indeed suitable for early requirements, since it provides both an elicitation and modeling methodology for capturing the high-level goals of an organization.

Formal Tropos aims to address the problems of the early requirements phase, but it is part of a wider-scope research effort, called *Tropos* [CKM01], 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.

2.3.1 The i* framework

i* [Yu95] is a requirements elicitation and modeling framework that was specifically designed for early requirements specifications. It has both a goal- and agent-oriented ontology. It assumes that during the early requirements 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 two kinds of graphical diagrams to describe the requirements of a system. The Strategic Dependency (SD) diagram is used to represent *strategic dependencies*, which 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, the appreciation is subjective, 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 Strategic Rationale (SR) diagram provides a description of the intentions of the actors in terms of process elements and the rationale behind them. This rationale is given via *means-end* links that relate the elements of the SD diagram to sub-elements, much in the same way as AND/OR decompositions.

The elements of i* are formalized following ideas of the agents research community in Artificial Intelligence [CL90]. The logical operators defined allow the representation of concepts such as *workability*, *commitment* and *transfer of workability*. Unfortunately, this formalization is not expressive enough to allow formal analysis on i* specifications, and the analysis must be done mostly at an informal level. Formal Tropos is largely inspired in i*, but it provides a language that has formal semantics, and is amenable to formal analysis.

2.3.2 An example in i*

We illustrate the use of i* in the context of the Insurance Company case study originally introduced in [YM94]. The case study will be used as a running example in the rest of this thesis. It is based on a setting in which customers insure their cars at an insurance

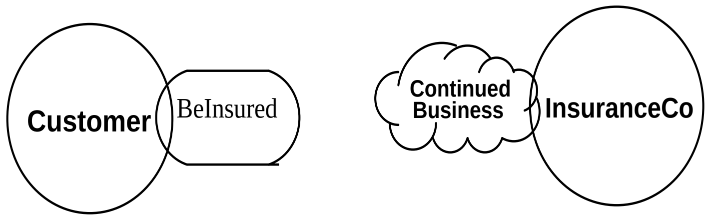

Figure 2.1: Fundamental actors of the case study, and their goals

company. In case of an accident, the customers expect to be reimbursed for the amount of the damages.

In Figure 2.1, we identify the fundamental actors of this domain, the customers `Customer` and the insurance company (`InsuranceCo`). The goal of the customer is to be insured against accidents of his car, while the insurance company expects to have a continued business.

The initial goals can be refined into goal dependencies and expressed in an SD diagram. In our case, in order to be insured, the `Customer` depends on the `InsuranceCo` (`CoverDamages`). Conversely, the insurance company depends on their customers to have a continued business. In particular, it must be able to attract customers (`Attract-Customers`), either by getting new customers or keeping the existing ones happy.

Once the fundamental actors and their dependencies have been identified, it is possible to discover additional actors that are necessary for the fulfillment of their goals. In our domain, it is clear that customers will depend on a `BodyShop` in order to repair their cars. The insurance company depends on `Appraiser`s to estimate the reasonability and amount of damages. The newly introduced actors also have dependencies on the previous ones. For instance, the appraisers depend on the insurance company to keep their job and the bodyshops depend on their customers to keep clients and have their own continued business.

We conclude by remarking that it is possible to appreciate in this example some of the limitations of the i* graphical notation. For instance, it is not possible to state

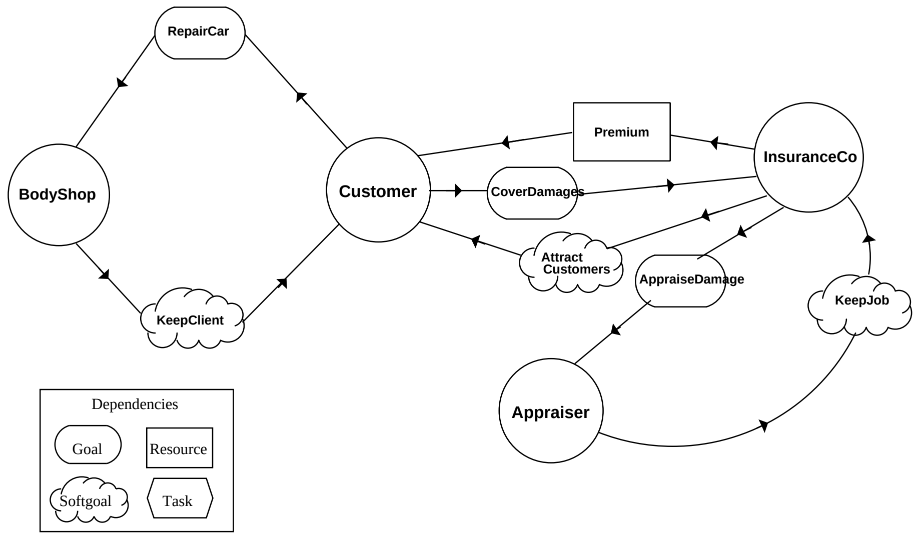

Figure 2.2: SD diagram for Insurance Company

relationships among dependencies, such as the fact that the insurance company will only cover damages if they have been previously appraised. As we will see in the next chapter, Formal Tropos extends i* in order to make this possible.

Chapter 3

The Formal Tropos Language

Although a graphical notation such as i* is valuable for human communication, its models are not detailed enough to be used as an starting point for performing formal analysis. In this section, we describe a textual language called *Formal Tropos*, which has more expressive power than i*, and is amenable to formal analysis. A precise presentation of its semantics is given in Chapter 5.

3.1 The Language

In this section, we explain the different elements of a Formal Tropos specification. The complete grammar is given in Figure 3.1.

A specification in Formal Tropos consists of a sequence of declarations of *entities*, *actors*, *dependencies*, and *global properties*. Declarations for entities, actors, and dependencies are structured in two layers. An outer layer declares their attributes, and is in a sense similar to a class declaration. The inner layer expresses constraints on the instances, and thus implicitly determines their evolution.

In the outer layer, entities, actors and dependencies are declared as classes which have an associated list of attributes that characterize their instances. Each attribute has a corresponding *sort* (i.e., its type) and one or more *facets*. Sorts can be either *declaration* := (*entity* | *actor* | *dependency* | *global-properties*)*

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

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

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

global-properties := **Global** [*name*] *global-property*+

actor-goals := (**Goal** | **Softgoal**) *name mode* [*fulfill-properties*]

attributes := **Attribute** *attribute*+

attribute := *attribute-facets name* : *sort*

attribute-facets := [**constant**] [**optional**] ...

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

type := **type** (**goal** | **softgoal** | **task** | **resource**)

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

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

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

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

creation-property := *property-category event-category property-origin* [1] *temporal-formula*

fulfill-property := *property-category event-category property-origin* [1] *temporal-formula*

invar-property := *property-category property-origin* [1] *temporal-formula*

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

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

event-category := **trigger** | **condition** | **definition**

property-origin := [**for depender** | **for dependee** | **domain**]

temporal-formula := ...

Figure 3.1: Formal Tropos Grammar

primitive (integer, date, boolean, etc.) or correspond to other classes (entities, actors, or dependencies) of the specification. Facets represent often-used properties of attributes. For example, the **constant** facet represents the fact that the value of the attribute cannot change after its initialization, while the **optional** facet means that the attribute might assume no value. We have other facets in mind, such as **multivalued**, which means that the attribute can assume more than one value (i.e., it represents a set of possible values).

Entities represent *non-intentional* elements of the environment or organizational setting. Each entity is identified by a *name*, and consists of a set of attributes, a set of **creation** properties, and a set of **invariant** properties. These properties define conditions that should hold, respectively, at the creation and during the life of each instance of an entity.

As an example, the following is a declaration for the entity `Claim` of the Insurance Company case study. The **creation** property specifies that a claim only makes sense if there is a suitable `CoverDamages` goal dependency instance associated to it.

Entity Claim
  Attribute constant insP : InsPolicy
  Creation condition $\exists cover : CoverDamages(cover.cl = self)$

Unlike entities, *actors* have strategic goals and intentionality. They also have attributes, and **creation** and **invariant** properties.

The goals of the actors describe their strategic interests. Goals have a **mode** and a set of **fulfillment** properties, which will be explained shortly. Notice that they do not have neither **invariant** nor **creation** properties; this is because goals are assumed to be created together with their actor, and associated to it along its life. If the actor is not able to fulfill a certain goal by itself, then the goal should be refined into dependencies to other (dependee) actors.

The following is the declaration for the `Customer` actor of the case study. Its goal

[1] The *property-origin* is only used for dependencies.

`BeInsured` represents the fact that the `Customer` expects to be reimbursed whenever there is an accident. The **trigger** constraint indicates that the goal will be fulfilled if, whenever a `Customer` has the goal of covering damages, it will be eventually fulfilled.

Actor Customer
  Goal BeInsured
    Mode maintain
    trigger
      $\forall cover : CoverDamages(cover.depender = self \rightarrow \Diamond Fulfilled(cover)))$

Strategic dependencies represent the relationships that exist among actors in order to fulfill their objectives. In a sense, the concept of strategic dependency is *reified* in Formal Tropos, since dependencies are declared as classes, and can be instantiated just as any other object of the system.

A dependency describes an "agreement" between two actors, the *depender* and the *dependee*. The *type* of the dependency describes the nature of this agreement:

- **goal** dependencies are used to represent delegation of responsibility for fulfilling a goal;
- **softgoal** dependencies are similar to goal dependencies; the difference being that, while it is possible to precisely determine when a goal is fulfilled, the fulfillment of a softgoal cannot be defined exactly (for instance, it can be a matter of personal taste, or the fulfillment can occur only to a given extent);
- **task** dependencies represent delegation of responsibility for performing a given activity;
- **resource** dependencies describe situations in which the dependee should deliver or provide some resource to the depender.

Besides the type, a dependency is also characterized by a *mode*, which declares the attitude of the actors with respect to its fulfillment. The modalities currently supported by Formal Tropos are the following

- **achieve**: fulfillment properties should be satisfied at least once;
- **maintain**: fulfillment properties should be satisfied in a continuing way;
- **achieve&maintain**: as a combination of the previous two modes, it requires the fulfillment properties to be achieved and then satisfied in a continuing way;
- **avoid**: the fulfillment properties should be prevented.

The evolution of a dependency is controlled by three kinds of properties. **Creation** properties determine the moment in which a new instance of the dependency can be created; **fulfillment** properties must hold in order to consider that a dependency is fulfilled; and **invariants** represent conditions that should be true along the life of the dependency. We remark again that the meaning of the fulfillment conditions changes according to the modality of the dependency. For instance, in an **achieve** dependency, it defines the condition that should hold once; and in a **maintain** dependency it defines a condition that should hold continuously after the creation of the dependency.

In addition to temporal formulas, properties have facets that determine their meaning. One kind of facets give what we call *property categories*; they determine how a property influences the valid scenarios for a specification. **Constraint** properties are *enforced*; they are implicitly defining the valid scenarios for the requirements specification. On the other hand, **assertions** and **possibilities** are *desired* properties of the specification; they are not enforced but *checked*, as we will show in the next chapter. While **assertions** are expected to hold in *all* valid scenarios for the specification, **possibility** properties are only expected to hold in *at least one* valid scenario. If none of these facets is present, we assume that the property is a constraint.

In the case of **creation** and **fulfillment** properties, facet **trigger** defines a sufficient condition for the creation or fulfillment; facet **condition** defines a necessary condition; and facet **definition**, a necessary and sufficient condition.

Finally, for the properties that are local to a dependency declaration, we can specify whether they are required by the depender (facet **for depender**); by the dependee (facet **for dependee**); or whether they are assumptions about the environment (facet **domain**). If none of the facets are present, it means that the analyst is not yet sure about the origin of the formula.

As an example, the following is the **goal** dependency `RepairCar`. The `Customer` depends on the `BodyShop` to have his car repaired. The goal can only arise when the car is broken, as represented by the **creation condition**. An obvious **fulfillment condition** from the customer is that the car should start running OK. Notice that this is just a necessary condition, since the car can start running for other reasons (e.g, it is repaired by other bodyshop).

Dependency RepairCar
  Type goal
  Mode achieve
  Depender Customer
  Dependee BodyShop
  Attribute constant cl : Claim, **constant** amount : Currency
  Creation
    domain condition $\neg cl.insP.car.runsOK$
  Fulfillment
    condition for depender $cl.insP.car.runsOK$

Finally, there is one element of the specification that is not declared as a class: *global* properties. They are similar in spirit to the invariants for entities, actors and dependencies; however, they are defined for the system as a whole. They can be useful to define properties that are impossible to accommodate into the semantics of the **creation**, **fulfillment**, or **invariant** properties of a class.

Temporal formulas

The temporal formulas used for specifying properties are given in a linear-time typed first-order temporal logic. We will briefly explain them in this section; a complete explanation of their underlying semantics is given in Chapter 5.

First of all, the formulas can contain past and future temporal operators:

$$f \ := \ \circ f \mid \Diamond f \mid \Box f \mid f\,\mathcal{U}\,f \mid \bullet\, f \mid \blacksquare f \mid \blacklozenge f \mid f\,\mathcal{S}\,f \mid \cdots$$

The meaning of the operators is the following:

- $\circ f$ (next state)
- $\Diamond f$ (eventually)
- $\Box f$ (henceforth, always in the future)
- $f_1\,\mathcal{U}\,f_2$ (until)
- $\bullet f$ (previous state)
- $\blacklozenge f$ (sometimes in the past)
- $\blacksquare f$ (always in the past)
- $f_1\,\mathcal{S}\,f_2$ (since)

Weak operators $f_1\,\mathcal{B}\,f_2$ (weak since), $f_1\,\mathcal{W}\,f_2$ (unless, weak until), $\tilde{\bullet} f$ (weak previous state) are also available. The difference between weak and strong operators is subtle, so we delay it until Chapter 5.

As first-order formulas, they may contain existential and universal quantifiers. Bound variables are typed by a sort, which can be either primitive, or the name of a class. The interpretation of the quantifiers is over all *existing* instances of the sort. Therefore, if an object has not been created at a certain point in time, it will not be taken into account when evaluating the quantifiers.

$$f \ := \ \forall x : sort.f \mid \exists x : sort.f \mid \cdots$$

The classical boolean, equality and relational operators are also available:

$$f \ := \ f \vee f \mid f \wedge f \mid \neg f \mid f \rightarrow f \mid \cdots$$

$$f \ := \ t = t \mid t \neq t \mid t > t \mid t < t \mid \cdots$$

Terms can be constants (c), variables (x), attributes of an object ($t.a$), and functions. Notice that there are two alternative syntax for functions. The first one is for functions that accept a fixed number of arguments. The other one is for functions that take an arbitrary number of arguments, defined by all terms t that satisfy a formula f (both t and f must contain a common free variable x of type $sort$). The set of functions is predefined (it is not possible to add new ones), and the domain and range must be basic sorts.

$$t \ := \ c \mid x \mid t.a \mid function$$

$$function \ := name(t, \ldots, t) \mid name\{t, x : sort, f\}$$

To illustrate functions with arbitrary elements, assume that we have the term following term

$$\mathbf{sum}\{repair.amount, repair : RepairCar, repair.cl = cl \wedge Fulfilled(repair)\}$$

The result of this function is the sum of the amounts of all `RepairCar` instances that are associated to a certain claim and have been fulfilled.

There are also some primitive predicates, such as `JustCreated(x)` (an instance is being created at this point in time), and `JustFulfilled(d)` (a dependency is being fulfilled at the current point in time). Another predicate, `Fulfilled(d)` has different interpretations depending on the modality of the dependency to which it belongs. For an **achieve** dependency, it is true if the dependency *has been* fulfilled. This means that `Fulfilled` is made true at the moment of the fulfillment and stays true forever. For a **maintain** dependency, `Fulfilled` remains true while the fulfillment properties hold (i.e, it is true since the creation of the dependency). For an **avoid** dependency, it its true if the fulfillment properties *never* hold (neither in the past nor in the future).

There are some constraints on valid formulas. First, they must be *well-sorted* (for instance, arithmetical operators can be applied only to terms of integer sort). Second, each variable x that appears in a formula must respect one of the following

- x is bounded by a $\forall x : sort$ or a $\exists x : sort$ quantifier;
- x is the name of one of the attributes of the entity, actor, or dependency that contains the property (not applicable to global properties);
- x is the identifier *self* (not applicable to global properties); *self* is used to denote the entity, actor, or dependency that contains the property;
- x is the identifier *depender* or the identifier *dependee* (only applicable to properties inside a dependency); these identifiers refer, respectively, to the depender and the dependee of the dependency.

3.2 The Case Study in Formal Tropos

In this section, we specify in Formal Tropos some aspects of the Insurance Company case study introduced in the previous chapter. For the moment, we are only concerned with *modelling* issues. Therefore, the specification we present should be considered as a first step in the requirements process, *before* the application of analysis techniques. In the rest of the thesis, we will focus on formal analysis, and illustrate its application for removing gaps and errors from this initial specification.

First of all, we specify the relevant entities of the domain, and their attributes.

Entity Claim
  Attribute constant insP: InsPolicy

Entity InsPolicy
  Attribute constant car: Car, **constant** expirDate: Date

Entity Car
  Attribute runsOK: boolean

Then, we identify the relevant actors and their strategic goals. The goal of the `Customer` is to be insured (i.e, reimbursed whenever there is an accident); the goal of the `InsuranceCo` and `BodyShop` is to run a continued business; and the `Appraiser` intends to have permanent employment.

Actor Customer
  Goal BeInsured
    [all her claims for accidents will be eventually covered]

Actor InsuranceCo
  Softgoal ContinuedBusiness
    [its clients will remain with the company and pay their premiums, and more clients will be attracted]

Actor BodyShop
  Softgoal ContinuedBusiness
    [customers will come back in the future, giving the bodyshop more business]

Actor Appraiser
  Softgoal PermanentEmployment
    [the Insurance Company will regularly use the appraiser's services]

So far, we have only identified the actors who are *interested* in the goals. Identifying the actors *responsible* for the fulfillment of such goals leads to the introduction of strategic dependencies. For instance, the goal `BeInsured` represents the fact that the `Customer` expects to be reimbursed whenever there is an accident. Once we identify `InsuranceCo` as the responsible actor, we can define a goal dependency called `CoverDamages`, where `Customer` is the depender and `InsuranceCo` the dependee.

We specify a number of constraint properties for the `CoverDamages` dependency. First, the insurance company wants to prevent the customer from claiming damages for a car that is not broken. This is expressed as a **creation condition** for the dependency. The actors are also concerned about the amount paid for the damages. The insurance company does not want to pay damages that exceed the amount suggested by some appraiser, while the customer expects to receive at least enough money to pay the cost of the repairs (but she will perhaps not complain if she receives more!). We represent such interests as **fulfillment condition**s for the dependency. The last **condition** uses a predefined operator for integers, **sum**. This is because there might be several bodyshops involved in repairing a car (and therefore several instances of `RepairCar`, for the same claim, that are fulfilled). The customer expects to receive enough money to pay all bodyshops, hence the **sum** operator.

Dependency CoverDamages
  Type goal
  Mode achieve
  Depender Customer
  Dependee InsuranceCo
  Attribute constant cl: Claim, **constant** amount : Currency
  Creation
    condition for dependee $\neg cl.insP.car.runsOK$
  Fulfillment
    condition for dependee
      $\exists app : AppraiseDamage(app.cl = cl \wedge Fulfilled(app) \wedge app.amount \geq amount)$

[the amount paid to the customer should not exceed the amount appraised]

condition for depender

$self.amount \geq$

$\mathbf{sum}\{repair.amount, repair : RepairCar, repair.cl = cl \wedge Fulfilled(repair)\}$

[the amount paid to the customer should exceed the cost of the repairs]

With the definition of `CoverDamages`, we are now able to formalize a property for the goal `BeInsured` of actor `Customer`. It indicates that whenever the customer has a goal of covering damages, the goal will be eventually fulfilled. The property is idealistic, since there might be cases in which the customer considers that she is insured even if some claim is not covered (but as long as the reason for the rejection is reasonable). Therefore, we specify it as a **trigger** property.

Actor Customer

Goal BeInsured

Mode maintain

trigger

$\forall cover : CoverDamages(cover.depender = self \rightarrow \Diamond Fulfilled(cover)))$

Furthermore, we might detect additional constraints in the specification. For instance, we specify that a claim makes sense only if it is associated to a suitable `CoverDamages` instance.

Entity Claim

Creation condition $\exists cover : CoverDamages(cover.cl = self)$

We specify the `ContinuedBusiness` softgoal of `BodyShop` with two idealized properties. The first represents the fact that the bodyshop is always expecting new customers to use its services. It is stated with the following property: the softgoal is fulfilled if all customers have repaired their cars at the bodyshop at some time. The second condition states the fact that customers are expected to come back to the bodyshop in case they have another accident.

Actor BodyShop

Softgoal ContinuedBusiness

Mode maintain

trigger

$\forall cust : Customer(\exists rep : RepairCar(rep.depender = cust \wedge$
$rep.dependee = self \wedge Fulfilled(rep)))$

trigger

$\forall keep : KeepClient(keep.depender = self \rightarrow Fulfilled(keep))$

The goal of repairing a car is represented with the dependency `RepairCar`, which was already introduced as an example in the previous section.

Dependency RepairCar

Type goal

Mode achieve

Depender Customer

Dependee BodyShop

Attribute constant cl : Claim, **constant** amount : Currency

Creation

domain condition $\neg cl.insP.car.runsOK$

Fulfillment

condition for depender $cl.insP.car.runsOK$

The fact that the bodyshop expects to keep its customers is represented with the **softgoal** dependency `KeepClient`.

Dependency KeepClient

Type softgoal

Mode maintain

Depender BodyShop

Dependee Customer

Creation

trigger for depender

$\exists rep : RepairCar(rep.depender = dependee \wedge rep.dependee = depender)$

Fulfillment

condition for depender

$\Diamond \exists rep : RepairCar(JustCreated(rep) \wedge$

$rep.dependee = depender \wedge rep.depender = dependee)$

[the bodyshop expects to be requested for future repairs]

trigger for depender

$\forall rep : RepairCar((rep.depender = dependee \wedge Fulfilled(rep))$

$\rightarrow rep.dependee = depender)$

[all future repairs of the customer will be done with this bodyshop]

This softgoal is triggered when the customer repairs her car with the bodyshop for the first time. This is not a necessary condition, since the bodyshop may also get in touch with customers in other ways, and expect that they will return for more business. Therefore, there is a **Creation trigger** that forces an instance of `KeepClient` to exist whenever a `RepairCar` instance with the same bodyshop and customer (but as dependee and depender) exists. Notice that the same constraint might have been represented as a fulfillment condition for `RepairCar`. However, the trigger representation matches the intuition of the users more closely, since it answers the question of when does the dependency `KeepClient` actually arise.

Although it is not possible to give complete formal definitions for softgoals, we can usually identify some relevant conditions. In the case of `KeepClient`, a fulfillment condition is that the appraiser should keep receiving jobs from the customer in the future. In particular, if the customer continues demanding repairs (`RepairCar` instances are created in which she is the depender), some of these repairs will have the bodyshop as dependee. Notice that this is a weak condition, which does not fully characterize the softgoal. It basically states that the customer will contact the bodyshop *infinitely often*, but the lapse of time between contacts is left open. It is also possible to give a strong (possibly too strong) constraint as a **fulfillment trigger**. In particular, we might state that all future repairs from the customer will be done with this bodyshop. We might say that this strong constraint represents an upper bound for the definition of the dependency. It will probably be violated, but the scenarios that violate it might help us understand the softgoal better. In conclusion, it is important to notice that, even if we are not able to capture the exact meaning of a softgoal dependency, we can approximate it from two opposite directions.

The `PermanentEmployment` goal of `Appraiser` has a similar specification to the goal `ContinuedBusiness` of the `BodyShop`. The first (idealized) property states that all insurance companies are expected to give jobs (appraisals to be done) to the appraiser; the second states that the appraiser expects to receive jobs regularly. It might be specified as follows

Actor Appraiser
  Softgoal PermanentEmployment
    Mode maintain
    trigger

$$\forall ins : InsCo(\exists app : AppraiseDamage.(app.dependee = self \wedge app.depender = ins \wedge Fulfilled(app)))$$

    trigger

$$\forall keep : KeepJob.(keep.depender = self \rightarrow Fulfilled(keep))$$

First of all, let us look at the goal dependency `AppraiseDamage`. Since the insurance company wants to pay as little as possible for the damages, we give a **fulfillment condition** which states that the amount of the appraisal should be minimized (all `AppraiseDamage` instances for the same claim should be for a greater or equal amount). However, such amount should not be arbitrarily low, since otherwise no bodyshop would be interested in repairing the car. This is specified with another fulfillment condition, which states that the appraisal should lead to a covered claim. We expect conflicts between the two conditions, whose resolution should lead to a better specification of the goal.

Dependency AppraiseDamage
  Type goal
  Mode achieve
  Depender InsuranceCo
  Dependee Appraiser
  Attribute constant cl : Claim, **constant** amount : Currency
  Creation
    condition domain
      $\exists cov : CoverDamages(cov.cl = cl)$
  Fulfillment
    condition for depender
      $\forall app' : AppraiseDamage(app'.cl = cl \rightarrow app'.amount \geq amount)$
        [minimize appraisal]
    condition for depender
      $amount > 0 \rightarrow \Diamond(\exists cover : CoverDamages(cover.cl = cl \land Fulfilled(cover)))$
        [the appraisal is fair, and therefore leads to a covered claim]

A strategic interest of the appraiser is to keep her job with the insurance company. We model this as a **softgoal** dependency `KeepJob`, analogous to `KeepClient`. We state the fact that the appraiser expects the insurance company to constantly ask for appraisals in the future with a **fulfillment condition**: at all times, there must eventually be an `AppraiseDamage` instance from the insurance company for this appraiser.

Dependency KeepJob
  Type softgoal
  Mode maintain
  Depender Appraiser
  Dependee InsuranceCo
  Creation
    trigger for depender
      $\exists app : AppraiseDamage(app.depender = dependee \land app.dependee = depender)$

Fulfillment

condition for depender

$\Diamond \exists app : AppraiseDamage(JustCreated(app) \wedge$
$app.dependee = depender \wedge app.depender = dependee)$
[the appraiser expects to be asked for appraisals sometime in the future]

The goal `ContinuedBusiness` of the insurance company consists of attracting new customers and ensuring that the existing ones renew their premiums.

Actor InsuranceCo
Softgoal ContinuedBusiness
Mode maintain
trigger
$\forall cust : Customers \exists attr : AttractCustomers$
$((attr.depender = self \wedge attr.dependee = cust) \rightarrow (Fulfilled(attr)))$

`AttractCustomers` is an **achieve and maintain** dependency. It is achieved when the customer starts business with the company and pays the first premium. After that, it is maintained as the customer renews her insurance by paying a premium whenever it is necessary. We model this by stating that there should always be a premium from the customer that is not expired.

Dependency AttractCustomer
Type goal
Mode achieve and maintain
Depender InsuranceCo
Dependee Customer
Fulfillment
condition for depender
$\exists pr : Premium.(pr.depender = depender \wedge pr.dependee = dependee \wedge$
$pr.insP.expirDate \geq \texttt{now} \wedge Fulfilled(pr))$

`Premium` is a **resource dependency**. Its **fulfillment conditions** must hold when the resource is delivered, in this case when the customer pays the premium. A condition from the customer is that the amount should not be excessively increased at the time of renewal. We state it by imposing that the amount of the increase be below a certain percentage (that is supposed to be an attribute of the `Customer`). Obviously, this condition could be made more realistic, for instance by relating it to the accident rate of the customer (as measured by its `CoverDamages` goals) or to environment conditions, such as the inflation rate.

Another constraint for the fulfillment is that the customer should be satisfied with the insurance company. A specification of this constraint is given by a **trigger** that states that the customer always fulfills her goals of covering damages. This is again an upper bound, since the customer might tolerate uncovered claims as long as the insurance company gives reasonable arguments for the rejection.

Dependency Premium
  Type resource
  Mode maintain
  Depender InsuranceCo
  Dependee Customer
  Attribute constant insP : InsPolicy, dueDate : Date, **constant** amount : Currency
  Fulfillment
    condition for dependee
      $amount \leq (dependee.maxIncreasePremium *$
        $\mathbf{max}\{pr.amount, pr : Premium, pr.insP.car = self.insP.car \wedge Fulfilled(pr)\})$
        [the customer is only willing to accept a limited increase in the premium]
    trigger for dependee
      $\forall cover : CoverDamages((cover.cl.insP = self.insP) \rightarrow \Diamond Fulfilled(cover))$
        [will only pay premium if all claims have been covered]

Chapter 4

Formal Analysis in Tropos

In this chapter, we introduce the different forms of analysis that can be carried out within the Formal Tropos framework, and we illustrate them in the context of our case study.

4.1 Forms of Analysis

Formal analysis techniques have been typically applied to the validation of the design (or implementation) of a system against its requirements. However, this approach is not possible during the early phases of software development, in which we are actually building the requirements specification.

In this thesis, we present a number of formal analysis techniques specifically designed to drive the elicitation and modelling of early requirements. Our techniques are implemented in a tool, called T-Tool, that starts from a Formal Tropos specification, and builds an automaton that models its behavior. The only limitation of the tool is that, due to the nature of the underlying verification techniques, it is necessary to restrict the number of instances of each Formal Tropos class that is being analyzed.

Once the automaton is built, T-Tool verifies expected behaviors of the system, expressed with **assertion** and **possibility** properties in Formal Tropos. The tool presents scenarios to the user that are useful for evaluating the specification. It shows *counterex-*

ample scenarios that depict the violation of an **assertion**, and *example* scenarios that show a situation in which a **possibility** holds. T-Tool also makes it possible to run an animation of the specification, in order to allow the user to gain a better understanding of its behavior.

In this chapter, we introduce the analytical support of Formal Tropos from a *user* perspective. The technical details of the approach and the implementation of the tool are explained in Chapter 6.

The following are the main forms of analysis supported by Formal Tropos:

- *Assertion validation.* T-Tool checks whether the **assertion** properties explained in the previous chapter hold on all possible scenarios of the Formal Tropos specification. If they do not, the tool gives an appropriate *counterexample*, which is a scenario that violates the assertion.

  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"). For instance, in the next section, we check an assertion which states that there must be a proof (e.g., an invoice) of the repairs whenever the Insurance Company covers damages for a claim. 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 and their modelling into a Formal Tropos specification.

  Another kind of properties is 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 probably not taken into account.

- *Non-emptiness check.* 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 obtained from different stakeholders.

- *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. For instance, in the next section we will have to fix an initial version of the specification for our case study because it does not allow a scenario in which a car is so damaged that cannot be repaired.

  This kind of scenario is specified in Formal Tropos with **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 an scenario that accounts for the possibility.

- *Animation.* T-Tool also allows the user to interactively explore scenarios for the Formal Tropos specification. Since such scenarios respect all the constraints of the requirements specification, 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. Moreover, the possibility of showing possible evolutions of the system is often a very effective way of communicating with the stakeholders.

4.2 Formal Analysis of the Case Study

In this section, we illustrate the use of T-Tool for the analysis of Formal Tropos specifications in the context of our Insurance Company case study. In particular, we analyze a subset of the Formal Tropos specification presented in the previous chapter, which includes the goals `CoverDamages` and `RepairCar`.

The initial specification, given in Figure 4.1, is simple and might contain some gaps or imprecisions. We will iteratively improve and refine it, guided by the results provided by our analytical tools.

4.2.1 Dealing with instances

In this subsection, we show a simple example that stresses the fact that T-Tool works with *instances* of the Formal Tropos classes.

In our example, it is clear from the scenarios provided by the stakeholders that claims are made *because* the customer has the goal of covering damages. Consequently, a claim must be created only *while* or *after* its corresponding goal arises. We state this as an assertion for the goal `CoverDamages`.

ASSERTION 1
Dependency *CoverDamages*
  Creation
    assertion domain condition $\Diamond JustCreated(cl)$

If we consider only scenarios that contain one instance of each class, the assertion holds. However, this is not the case with two instances of `CoverDamages`. T-Tool not only informs that the assertion does not hold, but it also provides the counterexample scenario shown in Figure 4.2.

In this scenario, at t_1, a first instance of `CoverDamages` (`cov1`) arises and a claim (`cl1`) is made. Then, at t_2, another instance of the goal `CoverDamages` (`cov2`) arises. When

Entity Car
  Attribute runsOK: boolean

Entity Claim
  Attribute car: Car
    Creation condition $\exists cover : CoverDamages(cover.cl = self)$

Actor Customer

Actor InsuranceCo

Dependency CoverDamages
  Type goal
  Mode achieve
  Depender Customer
  Dependee InsuranceCo
  Attribute constant cl: Claim
  Creation
    condition for dependee $\neg cl.car.runsOK$

Goal RepairCar
  Type goal
  Mode achieve
  Depender Customer
  Dependee BodyShop
  Attribute constant cl : Claim
  Creation
      domain condition $\neg cl.car.runsOK$
  Fulfilment
      condition for depender $cl.car.runsOK$

Figure 4.1: Initial Formal Tropos specification for analysis of the case study

	t_1	t_2
cov1	arises	
cov2		arises
cl1	created	

- cov1.cl = cov2.cl = cl1

Figure 4.2: Counterexample to Assertion 1

the scenario is presented to the stakeholders, they show concern about the existence of two `CoverDamages` instances for the same claim, since the goal of covering damages arises only once per claim. This fact was not correctly captured in the specification, so we fix it by adding the following invariant to the entity `Claim`.

CONSTRAINT 1
Entity *Claim*
Invar $\exists! cover : CoverRepairs(cover.cl = self)$ [1]

With this modification, the assertion now holds for the model.

At this point, it is natural to ask the stakeholders whether the constraint of having just one instance per claim holds for all goals. We do so, and they point out that it does make sense to have several instances of `RepairCar` per claim. For instance, the customer may contact several bodyshops in order to ask for prices (and consequently create several goals) but only one of them ends up repairing the car (fulfilling the goal). It may also be the case that several bodyshops are involved in the repair of a car, and then several `RepairCar` instances are fulfilled for the same claim.

In order to make sure that the previous cases are possible in our specification, we state them as **possibilities** for the `RepairCar` goal. T-Tool informs that they are indeed possible and gives the example scenarios shown in Figure 4.3.

[1] The notation $\exists!$ denotes that there exists *exactly* one instance.

	t_1
rep1	arises
rep2	arises
cl1	created

	t_1	t_2
rep1	arises	fulfilled
rep2	arises	fulfilled
cl1	created	

- rep1.cl = rep2.cl = cl1

Figure 4.3: Example scenarios for Possibility 1

POSSIBILITY 1

Goal *RepairCar*

Creation

possibility condition

$\exists rep : RepairCar(rep.cl = cl \wedge rep \neq self)$

Fulfilment

possibility condition $\exists rep : RepairCar(rep.cl = cl \wedge rep \neq self \wedge Fulfilled(rep))$

4.2.2 Assertion validation

In our example, an important goal of the Insurance Company is to avoid "unreasonable" claims, though it is difficult for the stakeholders 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
cov2	arises	fulfilled	
cl1	created		
car1	created		
car1.runsOK	$\perp$	$\top$	$\perp$

- cov2.cl = cl1
- cl1.car = car1

Figure 4.4: First counterexample scenario to Assertion 2

ASSERTION 2

Dependency *CoverDamages*

Fulfilment

assertion condition for dependee

$cl.car.runsOK \rightarrow \exists rep : RepairCar(rep.cl = cl \wedge Fulfilled(rep))$

We try the assertion in T-Tool; it informs that the assertion does not hold and produces a counterexample. We will see how the counterexamples of T-Tool allow us to modify the initial specification, so that the assertion holds.

The first counterexample produced by the tool is given in Figure 4.4. It shows a car that breaks at t_1 and starts running at t_2, but no instance of `RepairCar` has ever arisen. The stakeholders point out that this situation is not possible in the domain, since the only way of having a broken car running again is by repairing it. We add this fact to our Formal Tropos specification with an appropriate **invariant** constraint to the entity `Car`.

Entity Car

Attribute runsOK: boolean

Invariant $(runsOK \wedge \blacklozenge \neg runsOK) \rightarrow$

$\exists rep : RepairCar(rep.cl.car = self \wedge Fulfilled(rep))$

We check the new requirements specification with T-Tool, and get yet another counterexample scenario to Assertion 2, as shown in Figure 4.5. The problem of this counterexample is that the customer makes two claims, `cl1` and `cl2`, for the *same* car `car1`, but with different insurance companies. At time t_2, the goal `cov2` is fulfilled but there is no `RepairCar` instance for its associated claim `cl2`. The domain invariant for `Car` that we just introduced is not violated because an instance `rep1` (associated to the other claim, `cl1`) is also fulfilled in t_2.

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 ready at the moment the claim is covered. This can be ensured by adding the following constraint to the specification.

CONSTRAINT 2
Dependency *CoverDamages*
  Fulfilment
    condition for dependee
      $\exists rep : RepairCar(rep.cl = cl \wedge Fulfilled(rep))$

With the new constraint, Assertion 2 holds.

4.2.3 Possibility check

The fact that an assertion holds for a certain specification does not necessarily mean that the specification is correct. It might be the case, for instance, that the specification is too

	t_1	t_2
cov1	arises	
cov2	arises	**fulfilled**
rep1	arises	**fulfilled**
cl1	created	
cl2	created	
car1	created	
car1.runsOK	$\perp$	$\top$

- cov1.dependee = insCo1
- cov2.dependee = insCo2
- cov1.cl = cl1
- **cov2.cl = cl2**
- **rep1.cl = cl1**
- **cl1.car = cl2.car = car1**

Figure 4.5: Second counterexample scenario to Assertion 2

constrained and therefore leaves out scenarios that would otherwise be counterexamples to the assertion. In Formal Tropos we can test whether reasonable scenarios are possible by performing a *possibility check*.

For instance, in our case study it is possible that a car is so damaged after an accident that it is impossible to repair. The insurance company is still responsible for covering the damages, and the customer might use that money to buy another car. Therefore, we would like to check that an scenario in which damages are eventually covered but the car never runs OK again (is broken forever) is admissible in our specification.

POSSIBILITY 2

Dependency *CoverDamages*

Creation

possibility condition for depender

$\Diamond Fulfilled(self) \wedge \Box \neg cl.car.runsOK$

	t_1	t_2	t_3
cov1	**arises**	**fulfilled**	
cl1	created		
car1	created		
car1.runsOK	$\perp$	$\perp$	$\perp$

- cov1.cl = cl1
- cl1.car = car1

Figure 4.6: Example scenario for Possibility 2

Unfortunately, T-Tool informs that there is no scenario of the specification that accounts for this possibility. The problem is found in Constraint 2, which is too strong. We replace it with a constraint that allows for the possibility of having cars that break forever.

CONSTRAINT 3
Dependency *CoverDamages*
Fulfilment
condition for dependee
$\exists rep : RepairCar(rep.cl = cl \wedge Fulfilled(rep)) \vee \Box\neg cl.car.runsOK$

With this addition, Possibility 2 is now admissible in the specification. An example scenario given by the tool is shown in Figure 4.6

4.2.4 Detecting an incorrect possibility specification

In the next step, the stakeholders present an scenario that had been completely ignored so far. There are cases in which the insurance company covers damages for a car, even if the car does not run OK, as long as it makes sure that the car will be *eventually* repaired. The following is a preliminary specification of this possibility.

	t_1	t_2	t_3	t_4
cov2	arises		**fulfilled**	
rep1	arises			**fulfilled**
rep2	arises	**fulfilled**		
cl1	created			
car1	created			
car1.runsOK	$\bot$	$\top$	$\bot$	$\top$

- cov2.cl = rep1.cl = rep2.cl = cl1
- cl1.car = car1

Figure 4.7: Example scenario for Possibility 3

POSSIBILITY 3

Dependency *CoverDamages*

Fulfilment

possibility condition for depender

$$\neg cl.car.runsOK \land \exists rep : RepairCar(rep.cl = cl \land \Diamond JustFulfilled(rep))$$

When we add this **possibility**, we expect that the tool complains. The reason is that we believe that the possibility should not exist for our specification. This is because the fulfillment constraint for `CoverDamages` forces either a `RepairCar` instance to exist or the car to break forever. Before we start fixing the specification to allow for this **possibility**, we check that it is indeed incompatible with the current specification. As expected, T-Tool informs that the possibility does not exist for scenarios with one instance of `RepairCar`. Surprisingly, however, it presents an example scenario with two instances of `RepairCar`, as shown in Figure 4.7.

The scenario satisfies the specification of Possibility 3. However, it does not depict the problem that the stakeholders had in mind. This is because although `rep1` is fulfilled after `cov2`, this is deemed irrelevant by the fact that another `RepairCar` instance (`rep2`) had been previously fulfilled. Therefore, our specification of Possibility 3 was not correct, and we fix it in order to consider only those scenarios in which *no* `RepairCar` instance is fulfilled before its corresponding `CoverDamages` goal.

POSSIBILITY 4

Dependency *CoverDamages*

Fulfilment

possibility condition for depender

$$\neg cl.car.runsOK \wedge \exists rep : RepairCar(rep.cl = cl \wedge$$
$$\Diamond JustFulfilled(rep)) \wedge \forall rep : RepairCar(rep.cl = cl \rightarrow \neg Fulfilled(rep))$$

As we were expecting, the newly formulated possibility is not admissible for the specification. Notice that, this time, an scenario proposed by T-Tool allowed us to fix a specification error, rather than an incorrect assumption or requirement from the stakeholders.

4.2.5 Detecting an incorrect constraint specification

In order to allow Possibility 4 to occur, we weaken Constraint 3 so that it just requires a `RepairCar` instance to be *eventually* fulfilled.

CONSTRAINT 4

Dependency *CoverDamages*

Fulfilment

condition for dependee

$$\exists rep : RepairCar(rep.cl = cl \wedge \Diamond Fulfilled(rep)) \vee \Box \neg cl.car.runsOK$$

	t_1	t_2
cov2	arises	fulfilled
rep1	arises	fulfilled
cl1	created	
car1	created	
car1.runsOK	$\perp$	$\top$

- cov2.cl = rep1.cl = cl1
- cl1.car = car1

Figure 4.8: Example scenario for Possibility 4

With the new specification, T-Tool gives the example for Possibility 4 shown in Figure 4.8. However, we get an undesired side effect. Assertion 2, which was made true at the beginning of this example, gets false again after the introduction of Constraint 4. This assertion stated that there must be a proof (e.g, an invoice) of a repair for a claim to be covered.

The counterexample scenario is shown in Figure 4.9. In this scenario, Assertion 2 is violated at t_2 because the car is running OK, but `rep1`, the `RepairCar` instance for `cl1`, is fulfilled only at t_4. But, how could the car be running OK at t_2? The reason is that, once again, it is possible to have a `RepairCar` instance for *another* claim, but the same car. In particular, `rep2` is fulfilled at t_2, but corresponds to claim `cl2` instead of `cl1`.

This problem can be fixed by replacing Constraint 4 with another one that ensures that a car does not run OK until some `RepairCar` instance for the same claim has been fulfilled.

CONSTRAINT 5

Dependency *CoverDamages*

Fulfilment

condition for dependee

$\exists rep : RepairCar(rep.cl = cl \wedge (\neg runsOK \,\mathcal{U}\, Fulfilled(rep)))$

$\vee \Box \neg cl.car.runsOK$

	t_1	t_2	t_3	t_4
cov1	arises	**fulfilled**		
cov2	arises			
rep1	arises			**fulfilled**
rep2	arises	**fulfilled**		
cl1	created			
cl2	created			
car1	created			
car1.runsOK	$\perp$	$\top$	$\perp$	$\top$

- **cov1.cl = rep1.cl = cl1**
- **cov2.cl = rep2.cl = cl2**
- **cl1.car = cl2.car = car1**

Figure 4.9: A new counterexample to Assertion 2

Finally, all assertions and possibilities hold for the resulting Formal Tropos specification.

4.2.6 Conclusions

In this case study, we started with a simple specification and validated a number of assertions and possibilities, such as:

- there is always a proof that the car was repaired when the damages are covered;
- claims may be covered for cars that are so damaged that cannot be repaired;
- claims may be covered even if the car is still broken, as long as it is eventually repaired.

The process allowed the stakeholders to define constraints to their goals with more precision. For instance, in the initial specification there was no **fulfillment** constraint for `CoverDamages`. The examples and counterexamples provided by T-Tool allowed the stakeholders to discover this constraint, and to refine its specification twice. It was also possible to detect missing domain conditions, such as the **invariant** constraint on `Car`.

It is interesting to notice the interplay of scenarios along the case study. The stakeholders propose scenarios (either **possibilities** or **assertions**) and, if they do not hold, the tool responds with a new scenario (an example or counterexample). These scenarios might be useful to debug the specification (in case of an error or omission of the analyst) or to trigger a revision of the requirements from the stakeholders. This revision leads to the proposal of new **assertions** or **possibilities**, and hence to further iterations of this process.

Chapter 5

An Intermediate Language for Formal Tropos

In this chapter we present a translation from Formal Tropos into an Intermediate Language. This translation is intended to give a precise meaning to Formal Tropos specifications in a smaller language, which allows for a simpler formal semantics. Furthermore, the resulting Intermediate Language specification is more amenable to formal analysis, since it removes the strategic flavor of Formal Tropos and shifts the focus to the dynamic aspects of the system.

5.1 The Intermediate Language

An intermediate language specification consists of a *class signature*, which defines the classes (or data types) of the system; and a *logic specification*, which specifies constraints and desired properties on the temporal behavior of the class instances.

A class signature consists of a sequence of class declarations, where each of them is as follows:

CLASS c

$a_1 : s_1$

$\ldots$

$a_n : s_n$

c is the name of the class, and the a_i's are its attributes; s_i specifies the sort of attribute a_i. Sorts s_i can be either primitive (**integer**, **string**, **date** $\ldots$) or correspond to class names. It is also possible to specify whether an attribute is **optional** or not.

The logic specification consists of a set of formulas organized as follows

- **CONSTRAINT** formulas, which restrict the valid executions of the system;
- **ASSERTION** formulas, which are expected to hold in *all* valid executions of the system;
- **POSSIBILITY** formulas, which are expected to hold in *at least one* valid execution of the system.

The formulas are given in a first order linear-time temporal logic with future and past time operators. Objects can be created during execution, and therefore quantifiers $\forall x : s$ and $\exists x : s$ range over the objects of sort s that "exist" at a point in time. As a consequence, free variables do not necessarily "exist" at all moments. The fact that a variable exists might be stated in our logic with a formula such as $Exists(x) = \exists x'(x = x')$.

The logic includes the following temporal operators:

- $\circ f$ (next state)
- $\bullet f$ (previous state)
- $\tilde{\bullet} f$ (weak previous state)
- $f_1 \,\mathcal{U}\, f_2$ (until)

- $f_1 \mathcal{S} f_2$ (since)
- $\Diamond f$ (eventually)
- $\blacklozenge f$ (sometime in the past)
- $\Box f$ (henceforth, always in the future)
- $\blacksquare f$ (always in the past)
- $f_1 \mathcal{W} f_2$ (unless, weak until)
- $f_1 \mathcal{B} f_2$ (weak since)
- $(\Box \wedge \blacksquare) f$ (always, in the past and the future)
- $(\Diamond \vee \blacklozenge) f$ (sometime, in the past or in the future)

The difference between the weak and strong *until* operator is that, while $f_1 \mathcal{U} f_2$ requires f_2 to be eventually true, $f_1 \mathcal{W} f_2$ allows f_2 to be false forever, provided f_1 stays true forever. Similarly $f_1 \mathcal{S} f_2$ requires f_2 to be true sometime in the past, while $f_1 \mathcal{B} f_2$ allows f_2 to have been always false in the past, provided f_1 has been holding from the beginning of the execution. The difference between the strong *previous state* and the weak *previous state* operators will become clear in Section 5.1.1.

A model for a specification consists of a sequence of worlds, that correspond to snapshots of the system at different times (we use the natural numbers as the time domain). Each world provides domains for the basic sorts and the classes defined in the specification. Also, each world has to respect the signature, that is, if $a : s$ is an attribute of class c, then each instance of c has an attribute a in the domain of s.

A valid model must satisfy all **CONSTRAINT** formulas, since they are *enforced* on all valid executions of the system. We say that a specification is *non-empty* if it admits at least one valid model. We say that a specification is *correct* if the **ASSERTION** formulas hold in all valid models of the specification, and the **POSSIBILITY** formulas hold in at least one valid model.

In the next sections, we will introduce a formal definition of the syntax and semantics of the Intermediate Language.

5.1.1 Formal definition of the class signature

Monotonic domains

We have to deal with domains that change over time. This is a hard problem for first-order temporal logic, since it gives the possibility of mixing temporal operators and quantifiers (see [vB95] for a technical explanation of the reasons).

The solution that we adopt is to force domains to be monotonic. That is, we allow new objects to be created during the evolution of the system, but we do not allow already created objects to be destroyed.

This solves the problem of combining quantifiers and future temporal operators. If we say $\forall x : s.\ \Box\phi(x)$, where $\phi(x)$ is a formula that defines some property of x, then we know that all the objects x in the domain associated to sort s that exist at the present time, will also exist in all future snapshots. Therefore, $\phi(x)$ makes sense in all such snapshots.

However, we do have problems when we mix quantifiers and past temporal operators. For instance, for the formula $\forall x : s.\ \blacksquare\phi(x)$, if x is an object in the current domain of s, we are not guaranteed that x also existed in all past snapshots. The interpretation that we give to formula $\forall x : s.\ \blacksquare\phi(x)$ is hence the following: "for all the objects x of sort s in the current snapshot, formula $\phi(x)$ holds only in the past snapshots where object x existed". In this way, we restrict the scope of a $\blacksquare\phi(x)$ formula to the past snapshots where all the objects referred in ϕ existed.

The interpretation that we give to formula $\forall x : s.\ \blacklozenge\phi(x)$ is as follows: "for all the objects x of sort s in the current snapshot, there is some previous snapshot where object

x existed and formula $\phi(x)$ held". That is, in $\blacklozenge\phi$ we require ϕ to have held in a past snapshot where all the objects referred in ϕ existed.

The possibility that some of the objects in a formula do not exist in a previous state leads to two different previous-state operators. The strong previous state operator $\bullet\phi$ is true only if all the objects referred in ϕ exist in the previous snapshot (and ϕ holds in that snapshot). Instead, the weak previous state operator $\tilde{\bullet}f$ is true whenever some objects referred in ϕ do not exist in the previous snapshot (or if ϕ holds in that snapshot). As a consequence, in the initial state of the system, any formula $\bullet\phi$ is false while any formula $\tilde{\bullet}\phi$ is true.

Basic sorts and operations

Basic sorts correspond to the elementary data types used in the Intermediate Language, such as integer, time, date, string, etc. We assume that a given set S_0 of basic sorts is defined.

We also assume that a set of operations OP on basic sorts is defined. Operation $op \in OP$ can have one of the following signatures:

- $op : s_1 \times \cdots \times s_n \rightarrow s$
- $op : \mathcal{P}(s_1) \rightarrow s$

where the s_i's (the argument sorts) and the s (the result sort) are basic sorts. Also, the number n of the arguments may be 0 in which case the operation is a constant; in this case we write $op : s$. Operators can be defined only for basic sorts; the definition of a more general class signature, that allows for generic algebraic structures, is left as future work.

We assume that a fixed interpretation domain D_0^s is associated to each basic sort $s \in S_0$. Interpretations are also defined for the operations on basic sorts, i.e., if $op \in OP$ and $op : s_0 \times \cdots s_n \rightarrow s$, then $\mathcal{I}_{OP}[op] : D_{s_1} \times \cdots \times D_{s_n} \rightarrow D_s$ is a function from the domains of the arguments to the domain of the result. According to this definition, in the case of a constant $op : s$, the interpretation $\mathcal{I}_{OP}[op]$ is a value in the interpretation domain of sort s, namely $\mathcal{I}_{OP}[op] \in D_s$. Finally, the interpretation function is strict: if any of the arguments is undefined, the interpretation $\mathcal{I}_{OP}[op]$ is also undefined.

Class signature

A class signature specifies the classes that are present in a specification, together with their attributes.

A class signature is a pair $\Sigma = \langle C, \{\mathrm{A}_c^{opt}\}_{c \in C}, \{\mathrm{A}_c^{mand}\}_{c \in C}\rangle$, where:

- C is a set of *class identifiers*;
- for each class $c \in C$, set A_c^{opt} describes the *optional* attributes of class c, and set A_c^{mand} describes the *mandatory* attributes of class c.

While *optional* attributes might be undefined, *mandatory* attributes must have a value all along the life of their corresponding classes. In what follows, we will refer to set A_c, where $\mathrm{A}_c = \mathrm{A}_c^{opt} \cup \mathrm{A}_c^{mand}$, whenever the distinction between optional and mandatory attributes is not relevant.

The sorts S_Σ for a signature Σ are the basic sorts S_0 and the class identifiers C; namely, $S_\Sigma = S_0 \cup C$ (we assume that S_0 and C are disjoint).

In the following, we will often refer to sets of objects X that are S_Σ-sorted. This means that a sort $s \in S_\Sigma$ is associated to each object $x \in X$. We write $x : s$ whenever s is the sort of object x; moreover, we denote with X_s the subset of X whose objects are of sort s. We also assume that sorts are associated to the attributes of a class, that is, the sets A_c of the attributes of class c are S_Σ-sorted. In particular, we denote with A_c^s the attributes of class c of sort s.

An *interpretation* (or *world*) for a class signature Σ is a tuple $\mathcal{W} = \langle \{D^s\}_{s \in S_\Sigma}, \{\mathcal{I}_{c,a}\}_{c \in C, a \in \mathrm{A}_c}\rangle$, where:

- D^s is the interpretation domain for each sort s in the set S_Σ of sorts of the signature. We require that $D^s = D_0^s$ for each basic sort s. If $c \in C$, then D^c defines the existing instances of class c in the world.

- If $a : s$ is an attribute of class c, then $\mathcal{I}_{c,a} : D^c \to D^s$ is a function that, given an instance of a class c, returns a value of the attribute a. This defines the interpretation for attribute a of all the instances of class c in the world. If the attribute is optional ($a \in \mathrm{A}_c^{opt}$) the function is partial. Otherwise, it must be total.

We denote as $\mathcal{U}_\Sigma$ the set of all the possible worlds for class signature Σ. Given a S_Σ sorted set of variables V and a world $\mathcal{W} = \langle \{D_c\}_{c \in C}, \{\mathcal{I}_{c,a}\}_{c,a} \rangle$ in $\mathcal{U}_\Sigma$, a valuation of V in $\mathcal{W}$ is a function Φ that associates to each variable $v \in \mathrm{V}^s$ of sort s a value in D^s.

5.1.2 Formal definition of the logic specification

Terms

We now define the set of terms $\mathcal{T}_{\Sigma,\mathrm{V}}$ on the class signature Σ and on the S_Σ-sorted set of variables V. As terms are sorted, we rather define the classes $\mathcal{T}_{\Sigma,\mathrm{V}}^s$ of terms of sort $s \in S_\Sigma$.

- If $x \in \mathrm{V}^s$ then $x \in \mathcal{T}_{\Sigma,\mathrm{V}}^s$.

- If $op \in OP$, $op : s_1 \times \cdots \times s_n \to s$ and $t_i \in \mathcal{T}_{\Sigma,\mathrm{V}}^{s_i}$ for $i = 1, \ldots, n$, then $op(t_1, \ldots, t_n) \in \mathcal{T}_{\Sigma,\mathrm{V}}^s$.

- If $op_y \in OP$, $op_y : \mathcal{P}(s_1) \to s$, $f \in \mathcal{F}_{\Sigma,\mathrm{V}}$ and f introduces a free variable y, then $op_y(\{f\}) \in \mathcal{T}_{\Sigma,\mathrm{V}}^s$.

- If $c \in C$ is a class sort, $a \in \mathrm{A}_c^s$ is an attribute of c of sort s, and $t \in \mathcal{T}_{\Sigma,\mathrm{V}}^c$, then $t.a \in \mathcal{T}_{\Sigma,\mathrm{V}}^s$.

Given a term $t \in \mathcal{T}^s_{\Sigma,\mathrm{V}}$, an interpretation $\mathcal{W} \in \mathcal{U}_\Sigma$, and a valuation Φ of V in $\mathcal{W}$, we can define an interpretation of $\mathcal{I}_{\mathcal{W},\Phi}[t]$ of term t as an element of D^s.

- If $x \in \mathrm{V}^s$ then $\mathcal{I}_{\mathcal{W},\Phi}[x] = \Phi(x)$.
- If $op : s_1 \times \cdots \times s_n \to s$ and $t_i \in \mathcal{T}^{s_i}_{\Sigma,\mathrm{V}}$ for $i = 1, \ldots, n$, then
  - $\mathcal{I}_{\mathcal{W},\Phi}[op(t_1, \ldots, t_n)] = \mathcal{I}_{OP}[op](\mathcal{I}_{\mathcal{W},\Phi}[t_1], \ldots, \mathcal{I}_{\mathcal{W},\Phi}[t_n])$. if $t_1, t_2 \ldots t_n$ are defined.
  - Otherwise, $\mathcal{I}_{\mathcal{W},\Phi}[op(t_1, \ldots, t_n)]$ is undefined.
- If $op_y : \mathcal{P}(s_1) \to s$, $f \in \mathcal{F}_{\Sigma,\mathrm{V}}$ and f introduces a free variable y, then

$$\mathcal{I}_{\mathcal{W},\Phi}[op_y(\{f\})] = \mathcal{I}_{OP}[op_y](\{y' \in s_1 : \mathcal{R}, \Phi[y := y'] \models f(y)\})$$

- If $c \in C$ is a class sort, $a \in \mathrm{A}^s_c$ is an attribute of c of sort s, and $t \in \mathcal{T}^c_{\Sigma,\mathrm{V}}$, then
  - $\mathcal{I}_{\mathcal{W},\Phi}[t.a] = \mathcal{I}_{c,a}(\mathcal{I}_{\mathcal{W},\Phi}[t])$, if $\mathcal{I}_{\mathcal{W},\Phi}[t]$ is defined.
  - $\mathcal{I}_{\mathcal{W},\Phi}[t.a]$ is undefined if $\mathcal{I}_{\mathcal{W},\Phi}[t]$ is undefined.

The third item deserves further explanation. The operator op_y will be applied to a *set* of values. This set consists of all possible instantiations for free variable y such that the formula $f(y)$ holds. For example, in the following expression,

$$\forall cl' : Claim\,(\Sigma_{\mathbf{y}}(\exists rep : RepairCar(rep.cl = cl' \wedge y = rep.amount)) \geq 1000)$$

the variable y represents the amount of all `RepairCar` instances for a claim cl'. The operator produces the sum of all such instances.

Formulae

The formulae $\mathcal{F}_{\Sigma,\mathrm{V}}$ on the class signature Σ and on the S_Σ-sorted set of variables V are defined as follows.

- $tt \in \mathcal{F}_{\Sigma,\mathrm{V}}$.
- If $t \in \mathcal{T}^{boolean}_{\Sigma,\mathrm{V}}$, then $t \in \mathcal{F}_{\Sigma,\mathrm{V}}$.
- If $t_1, t_2 \in \mathcal{T}^{s}_{\Sigma,\mathrm{V}}$ for some sort s, then $t_1 = t_2 \in \mathcal{F}_{\Sigma,\mathrm{V}}$.
- If $f, f_1, f_2 \in \mathcal{F}_{\Sigma,\mathrm{V}}$, then also $\neg f, f_1 \wedge f_2 \in \mathcal{F}_{\Sigma,\mathrm{V}}$.
- If $f, f_1, f_2 \in \mathcal{F}_{\Sigma,\mathrm{V}}$ the also $\circ f, \bullet f, f_1 \mathcal{U} f_2, f_1 \mathcal{S} f_2 \in \mathcal{F}_{\Sigma,\mathrm{V}}$.
- If $f \in \mathcal{F}_{\Sigma,\mathrm{V}'}$ with $\mathrm{V}' = \mathrm{V}[x : s]$ then $\forall x : s.\ f \in \mathcal{F}_{\Sigma,\mathrm{V}}$.

In the definition above, we have represented with $\mathrm{V}[x : s]$ the S_Σ-sorted set obtained from V by setting the sort of variable x to s.

We define as $\mathrm{FV}f$ the free variables of a formula f, i.e., the set of the variables x that appear in f outside the scope of a $\forall x : s$ quantifier. The closed formulae $\mathcal{F}_\Sigma$ for signature Σ are the formulae in $\mathcal{F}_{\Sigma,\emptyset}$.

The following syntactic abbreviations are also defined:

- $f\!f \stackrel{\text{def}}{=} \neg tt$
- $f_1 \vee f_2 \stackrel{\text{def}}{=} \neg(\neg f_1 \wedge \neg f_2)$
- $f_1 \rightarrow f_2 \stackrel{\text{def}}{=} \neg f_1 \vee f_2$
- $f_1 \leftrightarrow f_2 \stackrel{\text{def}}{=} (f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1)$
- $\exists x : s.\ f \stackrel{\text{def}}{=} \neg(\forall x : s.\ \neg f)$
- $\Diamond f \stackrel{\text{def}}{=} tt\, \mathcal{U} f$
- $\Box f \stackrel{\text{def}}{=} \neg \Diamond \neg f$
- $f_1 \mathcal{W} f_2 \stackrel{\text{def}}{=} \Box f_1 \vee (f_1 \mathcal{U} f_2)$
- $\tilde{\circ} f \stackrel{\text{def}}{=} \neg \circ \neg f$

- $\blacklozenge f \stackrel{\text{def}}{=} tt\, \mathcal{S}\, f$
- $\blacksquare f \stackrel{\text{def}}{=} \neg \blacklozenge \neg f$
- $f_1\, \mathcal{B}\, f_2 \stackrel{\text{def}}{=} \blacksquare f_1 \vee (f_1\, \mathcal{S}\, f_2)$
- $\tilde{\bullet} f \stackrel{\text{def}}{=} \neg \bullet \neg f$
- $(\Box \wedge \blacksquare) f \stackrel{\text{def}}{=} \blacksquare f \wedge \Box f$

Formulae are interpreted on *runs*. A run $\mathcal{R}$ is an infinite sequence of worlds $\mathcal{W}_0, \mathcal{W}_1, \ldots$. Since domains are monotonic, if $\mathcal{W}_i = \langle \{D_i^c\}_c, \{\mathcal{I}_{i,c,a}\}_{c,a} \rangle$, then $D_i^c \subseteq D_{i+1}^c$ for all i.

Let $\mathcal{R} = \mathcal{W}_0, \mathcal{W}_1, \ldots$ be a run. Now we define when a formula f holds at position i of $\mathcal{R}$, according to valuation Φ, written $\mathcal{R}, i, \Phi \models f$.

- $\mathcal{R}, i, \Phi \models tt$.
- $\mathcal{R}, i, \Phi \models t$ for $t \in \mathcal{T}_{\Sigma,\mathrm{V}}^{boolean}$ if and only if $\mathcal{I}_{\mathcal{W}_i,\Phi}[t] = \mathbf{true}$.
- $\mathcal{R}, i, \Phi \models t_1 {=} t_2$ for $t_1, t_2 \in \mathcal{T}_{\Sigma,\mathrm{V}}^{s}$ if and only if $\mathcal{I}_{\mathcal{W}_i,\Phi}[t_1] = \mathcal{I}_{\mathcal{W}_i,\Phi}[t_2]$, and $\mathcal{I}_{\mathcal{W}_i,\Phi}[t_1]$ and $\mathcal{I}_{\mathcal{W}_i,\Phi}[t_2]$ are defined.
- $\mathcal{R}, i, \Phi \models \neg f$ if not $\mathcal{R}, i, \Phi \models f$.
- $\mathcal{R}, i, \Phi \models f_1 \wedge f_2$ if $\mathcal{R}, i, \Phi \models f_1$ and $\mathcal{R}, i, \Phi \models f_2$.
- $\mathcal{R}, i, \Phi \models \circ f$ if $\mathcal{R}, i{+}1, \Phi \models f$.
- $\mathcal{R}, i, \Phi \models f_1\, \mathcal{U}\, f_2$ if there is some j, with $i \leq j$, such that $\mathcal{R}, j, \Phi \models f_2$ and, for each l with $i \leq l < j$, $\mathcal{R}, l, \Phi \models f_1$.
- $\mathcal{R}, i, \Phi \models \bullet f$ if $i > 0$, $\Phi(v) \in D_{i-1}^s$ for each $v : s \in \mathrm{FV}(f)$, and $\mathcal{R}, i{-}1, \Phi \models f$.
- $\mathcal{R}, i, \Phi \models f_1\, \mathcal{S}\, f_2$ if there is some j, with $j \leq i$, such that $\mathcal{R}, j, \Phi \models f_2$ and, for each l with $j < l \leq i$, it holds that $\mathcal{R}, l, \Phi \models f_1$. Furthermore, $\Phi(v) \in D_j^s$, and $\Phi(v) \in D_l^s$ for each $v : s \in \mathrm{FV}(f)$.

- $\mathcal{R}, i, \Phi \models \forall x : s.\ f$ if, for each $d \in D_i^s$, $\mathcal{R}, i, \Phi[x := d] \models f$.

In the definition above we have represented with $\Phi[x := d]$ the valuation obtained from Φ by assigning value d to variable x.

We say that formula f holds for $\mathcal{R}$, according to valuation Φ, written $\mathcal{R}, \Phi \models f$, if and only if $\mathcal{R}, 0, \Phi \models f$.

If f is a closed formula, then we say that f holds at position i of $\mathcal{R}$, written $\mathcal{R}, i \models f$, if and only if $\mathcal{R}, i, \emptyset \models f$.

Finally, if f is a closed formula, then we say that f holds for $\mathcal{R}$, written $\mathcal{R} \models f$, if and only if $\mathcal{R}, 0, \emptyset \models f$.

5.1.3 Formal definition of an Intermediate Language specification

Having defined all the appropriate elements, we can now give the formal definition for an Intermediate Language Specification.

A specification in the Intermediate Language is a tuple $\mathcal{S} = \langle \Sigma, \mathcal{C}, \mathcal{A}, \mathcal{P} \rangle$, where:

- Σ is a class signature;
- $\mathcal{C}$ is a set of closed formulae on signature Σ, that specify the runs that are allowed in the valid models;
- $\mathcal{A}$ is a set of closed formulae on signature Σ, that specify properties that are expected to hold on *all* runs of the valid models.
- $\mathcal{P}$ is a set of closed formulae on signature Σ, that specify properties that are expected to hold on *at least one* run of a valid model.

A model for specification $\mathcal{S} = \langle \Sigma, \mathcal{C}, \mathcal{A}, \mathcal{P} \rangle$ is a run $\mathcal{R}$ such that, for each $f \in \mathcal{C}$, $\mathcal{R}, i \models f$ at all times i.

A specification $\mathcal{S}$ is *empty* (or unsatisfiable) if it admits no model.

An assertion $f \in \mathcal{A}$ is *correct* if $\mathcal{R}, i \models f$ holds for *each* model $\mathcal{R}$ of $\mathcal{S}$ at all times i. If assertion f is not correct, then a counterexample for f is a model $\mathcal{R}$ of $\mathcal{S}$ such that $\mathcal{R}, i \not\models f$.

A possibility $f \in \mathcal{P}$ is correct if $\mathcal{R}, i \models f$ holds for *a* model $\mathcal{R}$ of $\mathcal{S}$ at *some* time i. If possibility f is correct, then an example for f is a model such that $\mathcal{R}, i \models f$.

A specification $\mathcal{S}$ is correct if all its assertions $f \in \mathcal{A}$, and all its possibilities $f \in \mathcal{P}$ are correct. Clearly, we are interested in specifications that are both non-empty and correct.

5.2 From Formal Tropos to the Intermediate Language

In this section, we present the translation from Formal Tropos into the Intermediate Language. The resulting specification is a formalization of the semantics of a Formal Tropos specification.

5.2.1 From the Formal Tropos outer layer

Class signature

Each Formal Tropos class (entity, actor, dependency) is translated to a corresponding Intermediate language class as follows

Rule 1 (class signature) *For each FT class C, add the following class to the IL class signature*

CLASS C

$a_1 : s_1$

$\dots$

$a_n : s_n$

where each variable a_i corresponds to an attribute of the FT class, and each sort s_i denotes the type of the attribute.

We also add other attributes which, although not present in the FT specification, will be necessary for the formalization of its semantics. In particular, for each dependency class, we add the boolean attribute `fulfilled` to make it possible to reason about the fulfillment of the dependency, in the way we show in the next section. Similarly, for each actor goal `g`, we add an attribute `g-fulfilled` to the class of its corresponding actor. The introduced attribute is treated basically in the same way as the `fulfilled` attribute of dependencies. Furthermore, we include the attributes `depender` and `dependee`, which should correspond to the actors mentioned in the **Depender** and **Dependee** clauses of a dependency class. The only restriction for these attributes is that their sort must correspond to an actor class.

Rule 2 (additional class attributes for dependencies) *For each dependency of the FT specification, add the attributes `fulfilled`, `depender` and `dependee` to its corresponding IL class. The two latter must be treated as if they were **constant** attributes (and thus translated as indicated in Rule 6). For each actor goal g, add an attribute `g-fulfilled` to the class of its corresponding actor. This attribute will be treated in the translation in the same way as the `fulfilled` attribute of dependency classes.*

Finally, we can use the definition of the class signature to formalize the **optional** facet for attributes.

Rule 3 (optional facet) *If the **optional** facet is present for an attribute a of class C in the FT specification, then $a \in \mathrm{A}_c^{opt}$. Otherwise, $a \in \mathrm{A}_c^{mand}$.*

Logic specification

Since it has less expressive power, an IL class signature is not able to capture the complete semantics of the FT outer layer. For instance, the meaning of the **constant** facet has to be given by appropriate constraints in the IL logic specification.

Some rules formalize aspects of the semantics of class instances that are implicit in Formal Tropos. For instance, as explained in Chapter 3, we assume that, once fulfilled, dependencies remain in that state forever, regardless of the future evolution of the system. This is formalized with the following rule

Rule 4 (fulfillment forever for dependencies) *For each class D that corresponds to a dependency declaration in the FT specification, we add the following constraint*

$$\forall d : D\,(d.fulfilled \rightarrow \circ d.fulfilled)$$

A similar rule is used for formalizing the fulfillment of actor goals.

Rule 5 (fulfillment forever for actor goals) *For each goal g of a class A that corresponds to an actor declaration in the FT specification, we add the following constraint*

$$\forall a : A\,(a.g\text{-}\mathit{fulfilled} \rightarrow \circ a.g\text{-}\mathit{fulfilled})$$

The formalization of the **constant** facet is given by the following rule

Rule 6 (constant facet) *For each constant attribute a of type t declared in class C, we add the IL constraint*

$$\forall c : C\, \forall v : t\,(c.a = v \rightarrow \circ(c.a = v))$$

Notice that the rule also holds for optional attributes (if attribute a is undefined, then $c.a = v$ is false). The interpretation is that the **constant** facet is only applicable after the attribute has been given a value.

5.2.2 From the Formal Tropos inner layer

We will now explain the translation of the properties of a FT specification into IL formulas. First of all, we should replace all FT primitives predicates by variables of the IL. In particular, we replace the primitive predicate $Fulfilled$ by the variable `fulfilled` introduced in the class signature; and the primitive predicates $JustFulfilled$ and $JustCreated$ by appropriate IL translations. Notice that we will continue using $JustFulfilled$ and $JustCreated$ in the IL formulas, but they should be considered as just macros.

Rule 7 (substitutions for primitive predicates) *For every FT formula,*

- *every occurrence of predicate $Fulfilled(d)$, is replaced by $d.fulfilled$;*
- *every ocurrence of predicate $JustFulfilled(d)$, is replaced by $d.fulfilled \wedge \neg \bullet d.fulfilled$;*
- *every occurrence of predicate $JustCreated(d)$, is replaced by $\exists d'(d = d' \wedge \neg \bullet (d = d'))$*

Second, it is necessary to translate the *functions* that appear in the Formal Tropos temporal formulas into Intermediate Language *operators*.

Rule 8 (substitutions for functions) *For every FT formula,*

- *the occurrences of functions with a fixed number of arguments (syntax $\texttt{op}(t, \ldots, t)$) are translated directly to the Intermediate Language.*
- *the occurrences of functions with an arbitrary number of arguments*

  $$op_y(\{t, x : sort, f\})$$

  is replaced by the following Intermediate Language term:

  $$op_y(\{\exists x : sort\,(f \wedge y = t)\})$$

As an example of the second item of the previous rule, the following **fulfillment condition** for `CoverDamages` given in Chapter 3

$$\mathbf{sum}(\{repair.amount, repair : RepairCar, repair.cl = cl \wedge Fulfilled(repair)\})$$

would be translated as

$$\Sigma_y(\{\exists repair : RepairCar\,(repair.cl = cl \wedge Fulfilled(repair) \wedge y = repair.amount)\})$$

Notice that y is introduced as a free variable.

Global properties can be translated directly after the previous substitutions

Rule 9 (global properties) *For every formula ϕ corresponding to a FT global property, add formula ϕ' to the IL specification, where ϕ' is the formula obtained by applying rules 7 and 8 to ϕ.*

Unlike Formal Tropos, the formulas of the Intermediate Language are no longer associated to a particular class. This has to be taken into account for the translation of class properties (i.e., every property except global properties). We do this by introducing a new variable c of the sort of the corresponding class, and relating all the attributes of the class mentioned in the formula to this new variable. In the following, all the rules that we add to the IL specification will be universally quantified on c if they correspond to **constraint** or **assertion** properties, and existentially quantified if they correspond to **possibility** properties.

Rule 10 (substitutions for class properties) *For every formula corresponding to a property of a FT class C*

- *every occurrence of* ***self*** *is replaced with variable c;*
- *every occurrence of a free attribute a_i (i.e., a_i does not appear after a c.) is replaced with $c.a_i$;*

In the rest of this section, we will directly denote with ϕ the formula obtained after the substitutions of rules 7, 8, and 10.

The previous substitutions suffice for the translation of invariants.

Rule 11 (invariant property) *For every formula ϕ corresponding to an **invariant** property of class C, add the following formula*

$$
\begin{array}{ll}
\forall c : C\ \phi & \textit{if } \phi \textit{ is a } \textbf{\textit{constraint}} \textit{ or } \textbf{\textit{assertion}} \\
\exists c : C\ \phi & \textit{if } \phi \textit{ is a } \textbf{\textit{possibility}}
\end{array}
$$

The translation of **creation** and **fulfillment** properties of actors, dependencies, and actor goals is guided by the modality (e.g., **achieve**, **maintain**) and the property category (e.g., **necessary**, **trigger**). For a **Creation condition**, the FT property is simply a precondition for the creation of an instance, and is translated in the following way.

Rule 12 (creation condition property) *For every formula ϕ corresponding to a **creation condition** property ϕ, add the formula*

$$
\begin{array}{ll}
\forall c : C\ (\textbf{\textit{JustCreated}}(c) \rightarrow \phi) & \textit{if } \phi \textit{ is a } \textbf{\textit{constraint}} \textit{ or } \textbf{\textit{assertion}} \\
\exists c : C\ (\textbf{\textit{JustCreated}}(c) \wedge \phi) & \textit{if } \phi \textit{ is a } \textbf{\textit{possibility}}
\end{array}
$$

The translation for creation triggers is more complicated, since we cannot reference the attributes of instances that do not exist yet. In the next section, we explain the rationale for this rule in detail.

Rule 13 (creation trigger property) *For each formula ϕ corresponding to a **creation trigger** of class C , we add the following formula, where $a_1, a_2, \ldots, a_n$ are the free variables of ϕ.*

$$\forall a_1 : A_1, \forall a_2 : A_2, \forall a_n : A_n(\phi(a_1, a_2, \ldots, a_n) \rightarrow \exists c : C(c.a_1 = a_1 \wedge c.a_2 = a_2, \ldots c.a_n = a_n))$$

The translation for **creation definition** properties is a combination of the previous two rules.

Rule 14 (creation definition property) *For each* ***creation definition*** *property, add the formulas of Rules 12 and 13 to the IL specification.*

Since objects already exist when their **fulfillment** properties are evaluated, we do not run into the problems just explained for **creation trigger**s. In fact, **fulfillment trigger**s are just sufficient conditions, in the same way as the **conditions** are necessary conditions. The rules for **definition** properties is a combination of the rules for **condition** and **trigger**. On the other hand, modalities do play a substantial role in the translation of **fulfillment** properties, and we will give particular rules for each of them.

A **fulfillment** property belonging to an **achieve** dependency is translated as follows

Rule 15 (achieve fulfillment property) *For each formula ϕ corresponding to a* ***fulfillment*** *property of a class C with* ***achieve*** *modality, we add the formula*

$\forall c : C\,(JustFulfilled(c) \rightarrow \phi))$	*if ϕ is a* ***constraint condition*** *or* ***assertion condition***
$\forall c : C\,(\phi \rightarrow c.fulfilled))$	*if ϕ is a* ***constraint trigger*** *or* ***assertion trigger***
$\exists c : C\,(JustFulfilled(c) \wedge \phi))$	*if ϕ is a* ***possibility***

Formulas of a **maintain** dependency are translated in the following way

Rule 16 (maintain fulfillment property) *For each formula ϕ of a* ***fulfillment*** *property of a class C with* ***maintain*** *modality, we add the formula*

$\forall c : C\,((c.fulfilled \rightarrow (\Box \wedge \blacksquare)\phi))$	*if ϕ is a* ***constraint condition*** *or* ***assertion condition***
$\forall c : C\,(((\Box \wedge \blacksquare)\phi \rightarrow c.fulfilled))$	*if ϕ is a* ***constraint trigger*** *or* ***assertion trigger***
$\exists c : C\,((c.fulfilled \wedge (\Box \wedge \blacksquare)\phi))$	*if ϕ is a* ***possibility***

Formulas for **avoid** dependencies are obtained from the previous rule by simply negating ϕ

Rule 17 (avoid fulfillment property) *For each fulfillment property ϕ of a class C with* ***avoid*** *modality, we add the formula*

$\forall c : C\,((c.fulfilled \rightarrow (\Box \wedge \blacksquare)\neg\phi))$	*if ϕ is a* ***constraint condition*** *or* ***assertion condition***
$\forall c : C\,(((\Box \wedge \blacksquare)\neg\phi \rightarrow c.fulfilled))$	*if ϕ is a* ***constraint trigger*** *or* ***assertion trigger***
$\exists c : C\,((c.fulfilled \wedge (\Box \wedge \blacksquare)\neg\phi))$	*if ϕ is a* ***possibility***

The **achieve & maintain** modality is a combination of the rules for **achieve** and for **maintain**. The formula should hold at the present state ("achieve" part), and forever in the future ("maintain" part). Notice that, unlike the **maintain** modality, the formula does not necessarily hold from the beginning.

Rule 18 (achieve & maintain fulfillment property) *For each formula ϕ of a* ***fulfillment*** *property of a class C with* ***achieve & maintain*** *modality, we add the formula*

$\forall c : C\,(c.fulfilled \rightarrow \Box\phi)$	*if ϕ is a* ***constraint condition*** *or* ***assertion condition***
$\forall c : C\,(\Box\phi \rightarrow c.fulfilled)$	*if ϕ is a* ***constraint trigger*** *or* ***assertion trigger***
$\exists c : C\,(c.fulfilled \wedge \Box\phi)$	*if ϕ is a* ***possibility***

The formulas obtained according to the previous rules should be treated differently according to whether they correspond to a **constraint**, **assertion** or **possibility** of the Formal Tropos specification.

Rule 19 (constraints, assertions, and possibilities) *Let $\mathcal{S}$ be an Intermediate Language specification defined as in Section 5.1.3.*

If a formula ϕ belongs to a property which has the facet ***constraint*** *(or no* property-category *facet), then $\phi \in \mathcal{C}$, where $\mathcal{C}$ is the set of all constraints of the specification.*

If a formula ϕ belongs to a property which has facet ***assertion****, then $\phi \in \mathcal{A}$, where $\mathcal{A}$ is the set of all assertions about the specification.*

Finally, if a formula belongs to a property with ***possibility*** *facet, then $\phi \in \mathcal{P}$, where $\mathcal{P}$ is the set of all possibilities for the specification.*

An example of the translation

We will now illustrate the translation to the Intermediate Language using our case study. Figure 5.1 shows the class signature that corresponds to the initial Formal Tropos specification used in Chapter 4 (Fig. 4.1). Figure 5.2 shows the logic specification for the `CoverDamages` dependency, which includes all the elements that were incorporated during the analysis carried out in the previous chapter.

The first formula of Figure 5.2 is an instance of Rule 6, and formalizes the fact that `cl` is a **constant** attribute. The `depender` and `dependee` as a **constant** attributes (Rules 2 and 6) are omitted for lack of space. The second formula corresponds to Rule 4. The third is the translation for the **creation** constraint (Rule 12). The macro $JustCreated$ is extended in this formula for illustration purposes. The fourth formula is the translation for the **fulfillment** constraint for `CoverDamages` obtained at the end of the example in the previous chapter. The last four formulas are the translation for the **possiblity** and **assertion** properties used along the example, and they all correspond to Rule 15.

CLASS Claim
  car : Car
CLASS Car
  runsOK : boolean
CLASS Customer
CLASS InsuranceCo
CLASS BodyShop
CLASS CoverDamages
  cl : Claim
  fulfilled : boolean
  depender : Customer
  dependee : InsuranceCo
CLASS RepairCar
  cl : Claim
  fulfilled: boolean
  depender: Customer
  dependee: BodyShop

Figure 5.1: Class signature for case study

[**constant** facet]

CONSTRAINT $\forall e : CoverDamages\, \forall v : Claim(e.cl = v \rightarrow \circ(e.cl = v))$

[fulfillment forever]

CONSTRAINT $\forall e : CoverDamages(e.fulfilled \rightarrow \circ(e.fulfilled))$

[**creation condition** constraint]

CONSTRAINT
$\forall e : CoverDamages((\exists e' : CoverDamages(e' = e \wedge \neg \bullet (e' = e)) \rightarrow \neg e.cl.car.runsOK)$

[**fullfillment condition** (Constraint 5)]

CONSTRAINT $\forall e : CoverDamages(JustFulfilled(e) \rightarrow$
$(\exists r : RepairCar(r.cl = e.cl \wedge (\neg e.cl.car.runsOK\, \mathcal{U}\, r.fulfilled)) \vee \Box(\neg e.cl.car.runsOK)))$

[Assertion 1]

ASSERTION $\forall e : CoverDamages(JustCreated(e) \rightarrow \Diamond JustCreated(e.cl))$

[Assertion 2]

ASSERTION $\forall e : CoverDamages(JustFulfilled(e) \rightarrow$
$(e.cl.car.runsOK \rightarrow \exists r : RepairCar(r.cl = e.cl \wedge r.fulfilled)))$

[Possibility 2]

POSSIBILITY
$\exists e : CoverDamages(JustCreated(e) \wedge \Diamond e.fulfilled \wedge \Box(\neg e.cl.car.runsOK))$

[Possibility 4]

POSSIBILITY $\exists e : CoverDamages(JustFulfilled(e) \wedge \neg e.cl.car.runsOK \wedge$
$\exists r : RepairCar(r.cl = e.cl \wedge \Diamond JustFulfilled(r) \wedge$
$\forall rep : RepairCar(rep.cl = e.cl \rightarrow \neg rep.fulfilled)))$

Figure 5.2: Logic specification for `CoverDamages` dependency

Discussion on the translation

There are some aspects to notice about the translation. To start with, **possibility condition** properties that are local to a class have a different translation than **constraints** and **assertions**. In particular, while the former use existential quantification, the latter employ universal quantification. As we explained in Section 5.1.3, the interpretation of **possibility** properties is by definition existential. This is because we define a *correct* possibility f as one such that $\exists \mathcal{R}.\mathcal{R} \models f$, where $\mathcal{R}$ is a model of the specification. Therefore, it is natural to impose an existential semantics to the **possibility** properties that are attached to a class. For instance, in Chapter 4 the stakeholders stated the **possibility** that a car be so damaged after an accident that it is impossible to repair. It was specified in Formal Tropos in the following way

Dependency CoverDamages
  Creation
    possibility condition for depender
      $\Diamond Fulfilled(cov) \wedge \Box \neg cov.cl.car.RunsOK$

In this example, it becomes evident that the analyst had in mind that there should exist *at least one* car that breaks forever. The property is certainly not expected to be true for all cars.

Another point is that **creation trigger** properties are *not* translated by adding the formula $\forall c : C\,(\phi \rightarrow JustCreated(c))$, which would be analogous to the the one for **creation condition**s. The reason is that it is not possible to reference the attributes of an instance c which does not exist yet. The solution that we adopt is to universally quantify on the attributes of the instance rather than on the instance itself. According to Rule 13, for each **creation trigger** of class C , we add the following formula, where $a_1, a_2, \dots a_n$ are the free variables of ϕ.

$$\forall a_1 : A_1, \forall a_2 : A_2, \forall a_n : A_n(\phi(a_1, a_2, \dots, a_n) \rightarrow \exists c : C(c.a_1 = a_1 \wedge c.a_2 = a_2 \dots c.a_n = a_n))$$

We do not use $JustCreated(c)$ in the formula because we want to allow cases in which only the existence of an instance needs to be enforced when the trigger property holds, but not necessarily the creation of a new instance. For instance, consider the **creation trigger** of softgoal dependency `KeepJob` introduced in Chapter 3.

Dependency KeepJob
  Depender Appraiser
  Dependee InsuranceCo
  Creation
    trigger for depender
    $\exists app : AppraiseDamage(app.depender = dependee \wedge app.dependee = depender)$

The `Appraiser` expects that an instance of `KeepJob` be created whenever the goal of appraising a damage arises from the insurance company. However, it suffices with creating the goal of keeping the job just once: the first time that the insurance company is involved in a `AppraiseDamage` with the insurance company.

Also notice that the $\forall$-quantification is only on the attributes that appear in the formula, not on all the attributes of the entity c. To understand our reasons, consider the following example. Assume that we would like to say that all claims have a corresponding `CoverDamages` instance, whose amount is initially greater than zero.

Entity Claim
Entity CoverDamages
  Attribute cl: Claim, amount: Amount
  Creation condition $amount > 0$
  Creation trigger $JustCreated(cl)$

According to our translation, the formulas are interpreted as follows

$$\forall co : CoverDamages\,(JustCreated(co) \rightarrow co.amount > 0)$$
$$\forall cl : Claim\,(JustCreated(cl) \rightarrow (\exists co : CoverDamages\,(co.cl = cl)))$$

The translation is perfectly reasonable; the **creation trigger** on one attribute does not clash with the **creation condition** that initializes the other. Notice that if, instead, we decided to quantify on all the attributes of the `CoverDamages` class (regardless of whether they appear in the **trigger** property), we would obtain unreasonable results. First of all, the formula

$$\forall cl : Claim\, \forall am : Amount\, (JustCreated(cl) \rightarrow$$
$$(\exists co : CoverDamages.co.cl = cl \wedge co.amount = am))$$

requires that, for each claim there be (infinitely) many different covers, one for each value of amount. Second, this formula clashes with the creation condition, which requires the amount to be initially larger than 0.

The translation to the Intermediate Language makes it clear which aspects of Formal Tropos have a characterized meaning, and which not. For instance, all dependencies are treated in the same way, regardless of their type (**resource**, **softgoal**, etc.). In fact, the current rules were devised having goal dependencies in mind. For softgoal dependencies, the only thing we can say for the moment is that, since they cannot be completely defined, they will have triggers *or* conditions, but not both. We are investigating the possibility of applying logics that make it possible to model the intentionality of agents, such as BDI [RG95], or cope with different degrees of fulfillment, such as multivalued logics [Hun98]; they will probably give a better characterization for softgoals. With respect to **resource** dependencies, it might be possible that they can be simply reduced to **goal** dependencies. For instance, the resource `Premium` might be equivalent to a goal `ObtainPremium`. However, we are not completely sure about this point and the exact characterization of resources remains as future work.

Another point is that the agent-oriented aspects of Formal Tropos have not been formalized yet. First, actor classes are translated exactly in the same way as entities, except for actor goals, which are essentially **fulfillment** properties. Second, the attributes `depender` and `dependee` of the dependencies do not have any special interpretation, besides the the fact that they can be mentioned in the formulas as any other attribute of the class. Finally, the facets **for depender**, **for dependee** and **domain**, which denote the origin of a class property, are simply ignored in the translation. We expect that in the future, if we succeed to give an agent-oriented semantics to Formal Tropos, these elements will play a more substantial role.

Chapter 6

T-Tool: The Formal Tropos Validation Tool

In this chapter we present the technical aspects of the design and implementation of T-Tool, the Formal Tropos Validation Tool. This tool allows the engineer to validate Formal Tropos specifications by performing formal analysis on its Intermediate Language translation.

6.1 Model Checking vs. Theorem Proving

Traditional verification techniques in requirements analysis are based on *theorem proving*. The idea is that, given a *logical theory* (i.e., an axiomatization) of a system, we can determine whether a property holds in the system by proving that it is a *theorem* in the logical theory. For complex systems, this approach has some disadvantages. First, it is often difficult to produce an appropriate logical theory that axiomatizes the system. Second, the proofs to the theorems are usually long, complicated and error prone. Despite the existence of automatic theorem provers (e.g, [ORS92, GH93]), the process always involves some kind of human (and highly-qualified!) intervention. Finally, theorem provers do not provide any hints when we do not succeed in proving a formula. Therefore, it is not possible to tell whether the formula is false, or it is true and might have been proved with additional effort.

On the other hand, *model checking* [CGP99] techniques have a potential for fully automatic, "push-button" verification. They are based on a much simpler problem of formal logic: check that a formula is satisfied on a finite representation of a system; i.e. one *checks* that the finite representation is a valid *model* for the formula. An additional advantage of model checking is that, if a formula is not satisfied, it provides explicit information in the form of a counterexample.

We decided to use the model checking approach for our tool. In this way, it is able to provide fully automatic verification and to show counterexamples scenarios in case of violation of desired properties of the system. Unlike theorem proving, however, model checking techniques require the system to be finite. In our case, the consequence is that we have to put an upper bound on 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 appear only when more that one instance of each class is allowed in the system. Consider for instance the scenario, discussed in Chapter 4, of the customer that 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 also show that bugs usually become evident with just a small number of instances. In particular, all the examples of Chapter 4 have at most two instances of each class.

6.2 NuSMV

Our tool is built on top of the NuSMV verification framework [CCGR00]. NuSMV is a state-of-the-art model checker, based on symbolic representation techniques. Symbolic techniques [BCM+92] are one of the approaches (for others, see [GW93, CVWY92]) that have been developed to cope with the enormous state spaces that arise in real-size applications, also known as *state explosion problem*. Symbolic techniques represent implicitly rather than explicitly the states and transitions that model the system. The usual representation is an efficient encoding of boolean formulas known as Ordered Binary Decision Diagrams (OBDD) [Bry92].

NuSMV has two components. One is the input language, that (implicitly) defines an automaton as a representation of the system; the other is a temporal logic language that defines (desired) properties that will be checked on the automaton.

The input language for the finite-state system can be decomposed into *modules*. Individual modules can be instantiated multiple times, and modules can reference variables declared in other modules. State transitions in a model may be either deterministic or *non-deterministic*. Nondeterminism can reflect events that are controlled by the environment rather than the system; or be used to describe an abstract model where some details are hidden.

The logic language allows the expression of properties that will be checked on the system. The properties can be given in either LTL or CTL propositional logic, and their variables correspond to those declared in the input language. If a property is not true in the automaton, NuSMV shows a counterexample trace, either in textual or graphical form. It also offers an animator for the interactive exploration of the automaton.

To understand the syntax of NuSMV, consider the example of Figure 6.1. Module definitions begin with the keyword `MODULE`. The module `main` is the top-level module. The `VAR` statement is used to declare variables, whose type can be either primitive or correspond to a module. Primitive types must be finite (for instance, we must declare integer ranges rather than just integers); they can be either single valued (`boolean`, range of integers, etc.) or multivalued (arrays).

The `INIT` statement gives the initial value of variables. The `INVAR` statement states conditions on the variables that should be true at all states. The `TRANS` statement gives the transition relation for the automaton. The `next` operator is used to denote the value of a variable in the next state of the transition. The keyword `DEFINE` introduces additional variables that can be used as macros in the formulas. In order to force a given specification to execute infinitely often, we can use a `FAIRNESS` constraint, which restricts the attention of the model checker to only those execution paths along which a given CTL formula is true infinitely often.

The logic part consists of a sequence of `SPEC` statement (for CTL formulas) or `LTLSPEC` statements (for LTL formulas). In the formulas, the operators correspond to those of first-order logic (`&` is used for $\wedge$, and `|` is used for $\vee$), and the future fragment of CTL and LTL. In this work, we use only LTL; the operators are `F` (our $\Diamond$), `G` ($\Box$), `X` ($\circ$), and `U` ($\mathcal{U}$).

In the example, the state space of the system is determined by the declarations of variables `request` and `status`. The initial value of `status`, given by the `INIT` statement, is `ready`. The initial value of `request` is unspecified, so it can be either `0` or `1`.

The translation relation of the automaton is expressed with the the `TRANS` statement, which determines the value of the variables the next state (i.e, after the transition) in terms of their values in the current state. In this case, if `status` is `ready` and there is a `request`, the value of `status` in the next state will be `busy`. Notice that nothing is said about the transition relation in all other cases; its behavior is assumed to be non-deterministic.

The keyword `LTLSPEC` is followed by an LTL formula that will be checked on the automaton. The intuitive reading of the formula is that whenever `request` is true, in all possible future evolutions `state` must eventually become busy.

6.3 T-Tool

The use of T-Tool involves the following three steps

```
MODULE main
VAR
   request : boolean;
   status : {ready, busy};
INIT
   status = ready
TRANS
   (status = ready) & (request = 1) -> next(status = busy);
LTLSPEC
    G (request -> F (status = busy))
```

Figure 6.1: Example of NuSMV specification

- *Formal Tropos*: The analyst inputs a *Formal Tropos* specification, which is translated as explained in Chapter 5. Since there is a complete set of rules, the translation can be completely automatized. [1]

- *Intermediate Language*: The Intermediate Language specification is translated into an automaton specified in the input language of NuSMV. In the next section, we explain the technical considerations underlying this translation. The *automatic* translation of this step has been implemented as part of this thesis.

- *Validation*: The tool is used to perform the forms of analysis explained in Chapter 4: *non-emptiness check*, *possibility check*, *assertion validation*, and *animation*. For this step, we rely on the model checking algorithms provided by NuSMV, as explained in Section 6.5.

6.4 From the Intermediate Language to NuSMV

Given the Intermediate Language specification and a bound on the number of instances, the first step performed by the tool is to synthesize a (symbolic) automaton expressed in the input language of NuSMV. The states of this automaton respect the **CLASS** struc-

[1] The implementation of a translator for this step has not been completed at the time of writing of this thesis.

ture of the Intermediate Language specification, and its executions are all and only the executions that respect the **CONSTRAINT** formulas. An example of this translation is given in Figure 6.2. It corresponds to a excerpt of the NuSMV translation for the Intermediate Language specification of the case study of Chapter 4.

In the following subsections, we explain the different techniques applied by T-Tool in order to translate an Intermediate Language specification into the NuSMV input language.

6.4.1 Translation of the class signature

Each class of the Intermediate Language should be translated into a `MODULE` declaration in NuSMV. Unlike the Intermediate Language, in NuSMV it is necessary to declare all the instances that might exist in the system. We do this with a global array for each class of the Intermediate Language. For example, the following is a declaration for some instances used in the Insurance Company case study, assuming two instances of each class.

```
MODULE main
   VAR coverDamages : array 1..2 of CoverDamages;
   VAR customer : array 1..2 of Customer;
   VAR insuranceCo : array 1..2 of InsuranceCo;
   VAR claim : array 1..2 of Claim;
   VAR car : array 1..2 of Car;
```

NuSMV does not support the concept of *pointer* that exists in most programming languages. We simulate it by interpreting non-primitive sorts as ranges over array indices. So, for instance, the `MODULE` for `CoverDamages` can be translated as

```
MODULE CoverDamages
   VAR cl : 1..2;
   VAR exists : boolean;
   VAR fulfilled : boolean;
   VAR depender : 1..2;
   VAR dependee : 1..2;
```

```
MODULE Claim
  VAR car : 1..1;
  VAR exists : boolean;

MODULE Car
  VAR runsOK : boolean;
  VAR exists : boolean;

MODULE Customer
  VAR exists : boolean;

MODULE CoverDamages
  VAR cl : 1..1;
  VAR exists : boolean;
  VAR fulfilled : boolean;
  VAR depender : 1..1;
  VAR dependee : 1..1;

MODULE main
  VAR coverDamages : array 1..1 of CoverDamages;
  VAR RepairCar : array 1.. of RepairCar;
  VAR insuranceCo : array 1..1 of InsuranceCo;
  VAR customer : array 1..1 of Customer;
  VAR car : array 1..1 of Car;
  VAR claim : array 1..1 of Claim;

INVAR claim[1].exists -> claim[1].car = 1 & car[1].exists
TRANS claim[1].exists -> claim[1].car = next(claim[1].car)
TRANS claim[1].exists -> next(claim[1].exists)
TRANS car[1].exists -> next(car[1].exists)
TRANS customer[1].exists -> next(customer[1].exists)

INVAR coverDamages[1].exists -> coverDamages[1].cl = 1 & claim[1].exists
TRANS (coverDamages[1].exists & coverDamages[1].fulfilled) ->
        next(coverDamages[1].fulfilled))
TRANS (!coverDamages[1].exists & next(coverDamages[1].exists))
   -> (next(coverDamages[1].cl = 1 & claim[1].car = 1 & car[1].runsOK)

LTLSPEC  G ((coverDamages[1].exists -> coverDamages[1].exists & !
             Y coverDamages[1].exists ->  F (coverDamages[1].cl = 1 &
             claim[1].exists & ! Y claim[1].exists))


SPEC ! EG 1
```

Figure 6.2: Excerpt of a translation from the Intermediate Language

Since object references are no longer pointers but integers, we have to split terms that contain more than two object references. For instance, assume we want to evaluate the Intermediate Language term $cover.cl.car.runsOK$ for instances $cover_1$, cl_2 and car_3. We can not translate it as `cover[1].cl[2].car[3].runsOK`. Rather, we have to interpret it as `cover[1].cl = 2 & claim[2].car = 3 & car[3].runsOK`.

It is important to notice that, although all possible instances are declared beforehand, they might not "exist" (in the Intermediate Language sense of the word) at the beginning of the execution. The variable `exists` is introduced for solving this problem, as we will explain later.

6.4.2 Translation of the logic specification

The constraint formulas of the Intermediate Language logic specification define the valid behaviors of the system, and constitute the basis for constructing the NuSMV automaton. Each of these formulas should be translated into a sequence of `INVAR` and `TRANS` sentences of NuSMV.

Modeling the Intermediate Language semantics

The semantics of NuSMV and the Intermediate Language are different. Therefore, it does not suffice with translating the constraints; all aspects of the Intermediate Language semantics must be respected.

In particular, there are two aspects that we have to take into account when modeling the semantics of the Intermediate Language in NuSMV. First, Rule 3 indirectly depends on the *Interpretation Function* for attributes, defined in Chapter 5. This function is clearly not available in NuSMV; however, we can reformulate the rule as follows

Rule 20 (optional facet) *For each* ***mandatory*** *attribute a of class C, whose type is not primitive, add the following constraint*

$$\forall c : C\ (c.a.exists)$$

The rule ensures that a mandatory attribute exists all along the life of its corresponding instance. It can be readily translated into NuSMV by eliminating the quantifier. For attributes of a basic sort, the solution that we adopt consists of adding a new value `undef` to the domain of the sort. So, for instance, a boolean **optional** attribute would have domain {`0,1,undef`}.

Second, we have to ensure the enforcement of monotonic domains. We do this by inhibiting the destruction of objects, with the following rule

Rule 21 (existence forever) *For each FT class C, we add the following constraint to the NuSMV automaton*

$$\forall c : C\,(c.exists \rightarrow \circ c.exists)$$

Quantifier elimination

Since the formulas of the Intermediate Language are given in a first-order logic, they use quantifiers to implicitly reference the instances. In contrast, in NuSMV it is necessary to give explicit references to the instances that might exist in the system.

The rules for *quantifier elimination* must have two points into account. First, array referencing in NuSMV is limited to constant indices. So, for instance we can refer to `coverDamages[1]` but not to `coverDamages[i]`, where `i` is a range on the integers. As a consequence, all instances of a class in a quantified formula must appear explicitly in its translation to the NuSMV input language. Second, the quantifiers in the Intermediate Language are restricted to the worlds in which an instance exists; this has to be made explicit by using the variable `exists` introduced in the `MODULE` for its class.

The rule for $\forall$-elimination is as follows. Given an IL formula $\forall c : C\, \phi$, assume that there can be up to n instances of C in the system. The corresponding translation to NuSMV is `(c[1].exists` $\rightarrow$ ϕ`(c[1])) & ... & (c[n].exists` $\rightarrow$ ϕ`(c[n]))`[2]. For instance, $\forall car : Car(car.runsOK)$ would be translated as `(car[1].exists` $\rightarrow$ `car[1].runsOK) & ... (car[2].exists` $\rightarrow$ `car[2].runsOK)`.

The rule for $\exists$-elimination is similar. Given an IL formula $\exists c : C\, \phi$, assume that there can be up to n instances of C in the system. The corresponding translation to NuSMV is `(c[1].exists &` ϕ`(c[1])) | ... | (c[n].exists &` ϕ`(c[n]))`.

The limitation of this approach is that the number of clauses in the translated formula is exponential in the number of quantified variables. Obviously, this is a problem from a space efficiency point of view: the size of the resulting NuSMV specification might be unreasonably large. However, this problem might be easy to solve. For instance, we are planning to extend NuSMV with embedded capabilities for quantifier elimination. On the other hand, the complexity is unavoidable in general from the computational efficiency point of view, as a consequence of the state explosion problem.

Tableau construction for LTL formulas

The **CONSTRAINTS** of the IL specification are temporal formulas; in NuSMV, however, all the constraints should be expressed as `INVAR` and `TRANS` sentences that define the admissible states of the automaton. We make this translation possible by adapting a synthesis algorithm already provided by NuSMV [CGH94].

The motivation of the authors of the synthesis algorithm is different from ours. In

[2] $\phi(x)$ denotes the appropriate NuSMV translation of IL formula ϕ, where all references to the class instance c are replaced by x.

particular, it is used to extend the original SMV [McM93] – which provided only CTL verification– to allow LTL model checking, by showing that LTL model checking can be reduced to branching time CTL model checking with fairness constraints. This is an important breakthrough. LTL has proven to be more natural than CTL, since it is easier to reason on linear histories rather than on a tree of possible worlds. However, most symbolic model checking algorithms and verification tools are based on CTL, since it is easier to check.

They give a translator to construct a NuSMV automaton that represents a tableau for a given LTL formula f. They show that if the translation of f is added to a NuSMV automaton, together with an appropriate CTL formula and a fairness condition, the models of the resulting automaton are all and only those that satisfy *both* the original NuSMV automaton and the formula. Their application for this procedure, however, is different from ours. They use it for *checking* an LTL formula; we adapt it for *enforcing* the formula as a constraint.

The tableau is constructed as follows. Let AP_f be the set of atomic propositions in f. The tableau associated with f is a structure $T = (S_T, R_T, L_T)$ with AP_f as its set of atomic propositions. Each state in the tableau is a set of *elementary* formulas obtained from f. The set of elementary subformulas of f is denoted by $el(f)$ and is defined recursively in this way:

- $el(p) = p$ if $p \in AP_f$
- $el(\neg g) = el(g)$
- $el(g \vee h) = el(g) \cup el(h)$
- $el(\circ g) = \{\circ g\} \cup el(g)$
- $el(g \,\mathcal{U}\, h) = \{\circ(g \,\mathcal{U}\, h)\} \cup el(g) \cup el(h)$

Thus the set of states S_T of the tableau is $\mathcal{P}(el(f))$. The labeling function L_T is defined so that each state is labeled by the set of atomic propositions contained in the state.

In order to construct the transition relation R_T, we need an additional function sat that associates each elementary subformula g of f with a set of states in S_T. Intuitively, $sat(g)$ will be the set of states that satisfy g.

- $sat(g) = \{\sigma | g \in \sigma\}\ where\, g \in el(f)$
- $sat(\neg g) = \{\sigma | \sigma \notin sat(g)\}$
- $sat(g \vee h) = sat(g) \cup sat(h)$
- $sat(g \mathcal{U} h) = sat(h) \cup (sat(g) \cap sat(\circ(g \mathcal{U} h)))$

They use sat to define the transition relation R_T so that each elementary formula in a state is true in that state as follows

$$R_T(\sigma, \sigma') = \bigwedge_{\circ g \in el(f)} \sigma \in sat(\circ g) \leftrightarrow \sigma' \in sat(g)$$

Figure 6.3 gives the tableau for the formula $g = a \mathcal{U} b$. Each subset of $el(g)$ is a state of T. $sat(\circ g) = \{1, 2, 3, 5\}$ since each of these states contains the formula $\circ g$. $sat(g) = \{1, 2, 3, 4, 6\}$ since each of these states either contains b, or contains a and $\circ g$. There is a transition from each state in $sat(\circ g)$ to each state in $sat(g)$ and from each state in the complement of $sat(\circ g)$ to each state in the complement of $sat(g)$.

Unfortunately, the definition of R_T does not guarantee that *eventuality* properties are fulfilled. We can see this in the example of Figure 6.3. Although state 3 belongs to $sat(g)$, the path that loops forever in state 3 does not satisfy the formula g since b never holds on that path. Consequently, an additional condition is necessary in order to identify those paths along which f holds. The condition is that a path π that starts from

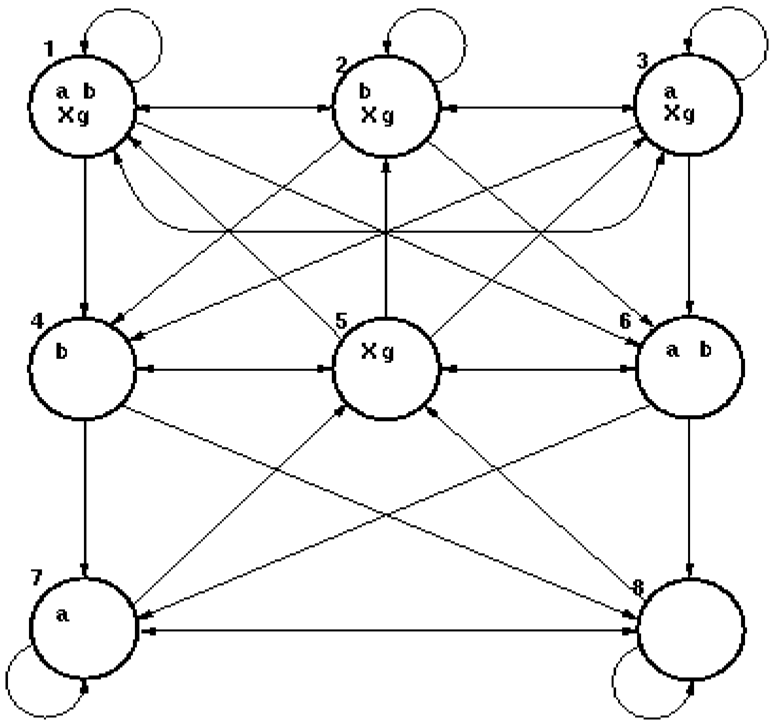

Figure 6.3: Tableau for $a \mathcal{U} b$

a state $\sigma \in sat(h)$ will satisfy f if and only if for every subformula $g\mathcal{U}\,h$ of f and every state σ on π, if $\sigma \in sat(g\mathcal{U}\,h)$, then either $\sigma \in sat(h)$ or there is a later state τ on π such that $\tau \in sat(h)$. In the paper, they prove that this condition is ensured if the CTL formula $\mathbf{EG}\,true$ is satisfied together with the fairness constraint

$$\{sat(\neg(g\mathcal{U}\,h) \vee h) \,|\, g\mathcal{U}\,h \text{ occurs in } f\}$$

Deadend states in the tableau construction procedure

The definition of R_T can cause the tableau to have states with no successors, which are called *deadend* states. For example, a state $\{a, \circ a, \circ\neg a\}$ has no successors, because no state is in both $sat(a)$ and $sat(\neg a)$. Because of the semantics of LTL, all the sequences that satisfy a path formula must be infinite. Thus, if we remove the deadend states from the tableau of f, no path that satisfies f will be eliminated. Therefore, in the tableau of f, we can safely ignore finite sequences that terminate in deadend states.

However, if deadend states are not removed, CTL verification is not possible in general. In particular, $\mathbf{EG}true$, the CTL formula from the previous section, cannot be checked. This is because the formula means that all initial states must have infinite paths; consequently, an initial deadend state makes it false.

Instead of removing deadends, we overcome the problem by checking whether the CTL formula $\neg\mathbf{EG}true$ is false. The formula checks whether there is *no* initial state with an infinite path. If it is true, the automaton is not satisfied by any model. Otherwise, we obtain a counterexample that consists of an infinite execution of the system, which is precisely a model for the automaton.

Notice, however, that this solution is only for checking the CTL formula that we need in the reduction of LTL formulas, not arbitrary CTL formulas. Of course, this is not a problem for us since the Intermediate Language is based on an linear time logic.

Past operators

The tableau construction procedure presented in [CGH94] is presented for the future fragment of LTL only. Our logic also includes the past fragment, since we consider it convenient for reasoning about requirements specifications. Therefore, in this work we adapted the tableau procedure to allow past operators.

It suffices to give rules for the $\bullet$ and $\mathcal{S}$ operators, the others can be obtained from them. The two new rules for obtaining the set of elementary formulas and the *sat* function are

- $el(\bullet g) = \{\bullet g\} \cup el(g)$
- $el(g \,\mathcal{S}\, h) = \{\bullet(g \,\mathcal{S}\, h)\} \cup el(g) \cup el(h)$
- $sat(g \,\mathcal{S}\, h) = sat(h) \cup (sat(g) \cap sat(\bullet(g \,\mathcal{S}\, h)))$

The previous definitions are symmetric to those for the future fragment. However, instead of giving fairness conditions, it is now necessary to give initial values for the elementary formulas. In other words, we have to give the value of *sat* explicitly for the initial state. This is because, at any point in time, we have an infinite future but a finite past (our logic is *anchored* [MP89] to an initial state). Therefore, while the only way of reasoning about the infinite future trace is to give a fairness condition, it suffices to give initial conditions for the finite past trace.

Other tasks were also necessary in order to accommodate past operators. For instance, we modified the grammar of NuSMV to include them: `Y` for $\bullet$, `P` for $\blacklozenge$, `H` for $\blacksquare$, and `S` for $\mathcal{S}$. Furthermore, we had to cope with their semantics in the Intermediate Language. As explained in Section 5.1.1, if any of the objects referred inside a past operator does not "exist", the formula is false. We approach this by adding `x.exists & ...` inside the temporal operator, for each variable that is mentioned. For instance, $\bullet(car.runsOK)$ would be translated as `Y (car.exists & car.runsOK)`.

Formula translation

Based on the described tableau construction, the translation from an Intermediate Language formula to NuSMV is as follows.

First, we associate a state variable with each elementary formula of f. To do this, f is expanded to a formula in which the only operators are $\vee$, $\neg$, $\circ$, $\mathcal{U}$, $\bullet$, and $\mathcal{S}$. The parse tree of f is traversed to find its elementary formulas. If a node associated with formula $\circ g$ (or $\bullet g$) is visited, then the corresponding elementary formula is stored in a list *el_list*. The translator declares a new variable EL_{Xg} for each formula $\circ g$ in the list *el_list*. Since atomic propositions are already declared, they need not be declared again.

In order to generate descriptions for the transition relation and the fairness constraints, we have to construct the characteristic function S_h of $sat(h)$ for each subformula or elementary formula h in f. The translator builds these functions using `DEFINE` statements, by traversing the parse tree of the formula and generating the appropriate statements at each node.

S_h`:=p;`	if p is an atomic proposition.
S_h`:=`EL_h`;`	if h is an elementary formula $\circ g$ or $\bullet g$
S_h`:=!`S_g`;`	if $h = \neg g$
S_h`:=`S_{g_1}`|`S_{g_2}`;`	if $h = g_1 \vee g_2$
S_h`:=`S_{g_2}`|(`S_{g_1}`&`$S_{X(g_1 U g_2)}$`)`	if $h = g_1 \mathcal{U} g_2$
S_h`:=`S_{g_2}`|(`S_{g_1}`&`$S_{Y(g_1 S g_2)}$`)`	if $h = g_1 \mathcal{S} g_2$

The transition relation is represented by using the `next` operator of NuSMV. For each $\circ g$ in *el_list*, the translator adds a `TRANS` statement of the form S_{Xg}`=next(`S_g`)`, and for each $\bullet g$, a `next(`S_{Yg}`) = `S_g. Furthermore, for each $\bullet g$, the translator gives an initial value (we set all variables to be false at the beginning). The translator also traverses the parse tree and generates the `FAIRNESS` constraint for each node associated with a formula of the form $g \mathcal{U} h$:

`FAIRNESS !`S_{gUh} `|` S_h

```
1  MODULE main
2   VAR
3    a: boolean;
4    b: boolean;
5    EL_X_a_U_b : boolean;
6  DEFINE
7    S_a := a;
8    S_b := b;
9    S_X_a_U_B := EL_X_a_U_b;
10   S_a_U_b := S_b | (S_a & S_X_a_U_b);
11 TRANS      S_X_a_U_b = next(S_a_U_b)
12 FAIRNESS    !S_a_U_b | b
13 INVAR       S_a_U_b
14 SPEC        !(EG true)
```

Figure 6.4: Example of the NuSMV translation of a formula

Finally, the translator generates an `INVAR` for the formula f. The invariant simply states that the variable associated to f (i.e, S_f) is true in all states.

The check for non-emptiness is added at the end of the translation of all formulas. It consists of a `SPEC` sentence that checks the CTL formula `!EG true`, which is expected to be false and yield a counterexample of an infinite trace, as described in the previous section.

We illustrate the translation procedure by applying it to the example of Figure 6.3 for formula $a \mathcal{U} b$. The NuSMV translation is shown in Figure 6.4. The translator first determines that a, b and $\circ(a \mathcal{U} b)$ are elementary formulas and generates the declaration for variable `EL_X_a_U_b` in line 5. Next, the `DEFINE` statements of line 7 to 10 are constructed for the characteristic funcions of $sat(a)$, $sat(b)$, $sat(\circ(a \mathcal{U} b))$ and $sat(a \mathcal{U} b)$. The `TRANS` statement in line 11 causes the transition relation of the tableau to be constructed, and line 12 contains the fairness constraint for $a \mathcal{U} b$. Line 13 enforces all states of the synthesized automaton to respect the constraint, in this case $a \mathcal{U} b$. Finally, line 14 contains the check for non-emptiness.

loops	NAME	car[1].RunsOK	coverDamages[2].cl	coverDamages[2].fulfilled	repairCar[1].cl	repairCar[1].fulfilled
	State 1	0	1	0	1	0
	State 2	0	2	0	1	0
	State 3	1	2	1	1	1
	State 4	0	2	1	1	1
	State 5	0	2	1	1	1

Figure 6.5: A trace shown by T-Tool

6.5 Validation in T-Tool

Formal Tropos offers a number of formal analysis techniques: *non-emptiness check*, *possibility check*, *assertion validation*, and *animation*. In Chapter 4, they were presented from the *user* point of view; in this section, we explain them from a technical standpoint.

6.5.1 Assertion validation

We take the standard approach of model checking; that is, we check the **ASSERTION** formulas against the executions of the NuSMV automaton. Unlike constraints, they can be written directly in NuSMV, since it offers the `LTLSPEC` clause. The only point that is worth mentioning is that `LTSPEC` formulas are assumed to be *anchored* to an initial state, whereas **ASSERTION** formulas are supposed to be valid at all points of the execution. Therefore, when translating an **ASSERTION** to NuSMV, we must add a `G` ($\Box$) temporal operator in front of the translated formula.

Whenever an `LTLSPEC` formula results false, the tool shows a trace that corresponds to a counterexample scenario that violates the assertion. As an example, Figure 6.5 gives part of the trace shown by T-Tool for the counterexample of Figure 4.5.

6.5.2 Non-emptiness check

The *non-emptiness check* (or consistency check) checks whether there is *at least* one model that satisfies the specification. In our approach, this is checked with the CTL formula $\mathbf{EG}true$ (or, rather, with the counterexample trace to $\neg\mathbf{EG}true$) that we include during the synthesis process explained in the previous section.

6.5.3 Possibility check

The *possibility check* checks whether the specification allows certain scenarios that the stakeholder expects to take place. We check whether the **POSSIBILITY** formulas hold in *at least* one trace for the automaton.

In order to explain the translation for **POSSIBILITY** properties, it is necessary to understand the semantics of LTL in detail. LTL formulas are considered to be a subset of the more expressive CTL* logic [EL86]. There is a distinction in the literature between *existential* and *universal* LTL [BCCZ99]. An LTL formula is *existentially* valid iff $\mathbf{E}f$ is valid in CTL*; it is *universally* valid iff $\mathbf{A}f$ is valid in CTL*. Clearly, **ASSERTION** properties have a universal interpretation, whereas **POSSIBILITY** formulas have an existential interpretation.

The interpretation for `LTLSPEC` formulas in NuSMV is also *universal*. Therefore, **ASSERTION** formulas can be translated directly. For **POSSIBILITY** formulas, the solution that we adopt consists of negating the formula and checking whether it is false. Specifically, we add `G !` ($\Box\neg$) in front of the formula and include it as an `LTLSPEC`. If the resulting formula is true, it means that the **POSSIBILITY** does not exist. Otherwise, the **POSSIBILITY** exists, and the trace presented by the tool corresponds to an example scenario.

6.5.4 Checking automatically generated properties

In addition to the formulas that correspond to Formal Tropos properties, T-Tool can *automatically* generate and check **ASSERTION** and **POSSIBILITY** formulas for a given Formal Tropos specification. These formulas are useful from different points of view. First, they help the stakeholders and the analyst to detect errors in the specification. Second, they provide hints on how to reduce the number of instances necessary for validating the specification, which is particularly relevant for alleviating the state explosion problem.

Some of these formulas are:

- *triggers must imply (necessary) conditions.* While the violation of this rule actually leads to an empty specification, it facilitates the detection of the cause of the inconsistency.

- *reachability test*: for all **achieve** goals, there is at least one model in which the goal is satisfied. This is important in order to detect goals that are irrelevant in the specification.

- *entity coverage test*: test whether there exists some model in which all the entities are eventually created. This is useful because in some systems there may be constraints that inhibit certain instances from being created at all; and we certainly want to reduce the number of variables of the automaton that we model check. For instance, in the example of Chapter 3, there is a creation trigger which states that a `KeepJob` instance arises whenever there exists an `AppraiseDamage` instance that satisfies certain conditions. If there were no `KeepJob` instance declared in the model, then no `AppraiseDamage` would be ever created, since otherwise the trigger would be violated.

Notice that while the first type of formula can be expressed in the Intermediate

Language as a **CONSTRAINT** formula, the last two correspond to **POSSIBILITY** formulas.

6.5.5 Animation

T-Tool allows the user to interactively explore the evolution of the synthesized automaton. Since the animation exhibits only the sequences of states that respect all the constraints, the user gets immediate feedback on their effects. Our tool uses the NuSMV Simulator for this purpose, which allows two modes of exploration. In *interactive mode*, the user decides on the evolution of the system at each transition; in *random mode*, a certain number of transitions are performed at random by the simulator.

It is important to note that our approach to automaton synthesis (the tableau procedure) has some limitations in its application to the animation of specifications. First, the user might get annoyed if her execution ends up in a deadend state. This means that the execution she has been monitoring so far is not valid on the system, and that she should restart the animation. This problem can be solved by removing all dead-end states of the automaton.

A more serious problem is that the automaton obtained via tableau construction might have some unexpected properties. Consider the LTL formula $a \rightarrow \Diamond b$, which means that if a holds, b will eventually be true. In Figure 6.6, we give a subset of its tableau (the states $\{b\}$, $\{b, \circ\Diamond b\}$ and $\{a, b\}$ are omitted). If a and b are initially false, the tableau forces us to decide whether the initial state is 1 or 2. If the initial state is 1, we keep in it forever; otherwise, the fairness condition leads the execution to eventually reach a state in which b holds.

The problem when using the previous tableau for animation purposes is that it is not reasonable to decide beforehand whether b will be eventually reached. Notice that if we merge states 1 and 2, and we add a fairness condition $a \rightarrow b$, we get a model that is appropriate for animation. We need to investigate whether it is possible to produce this

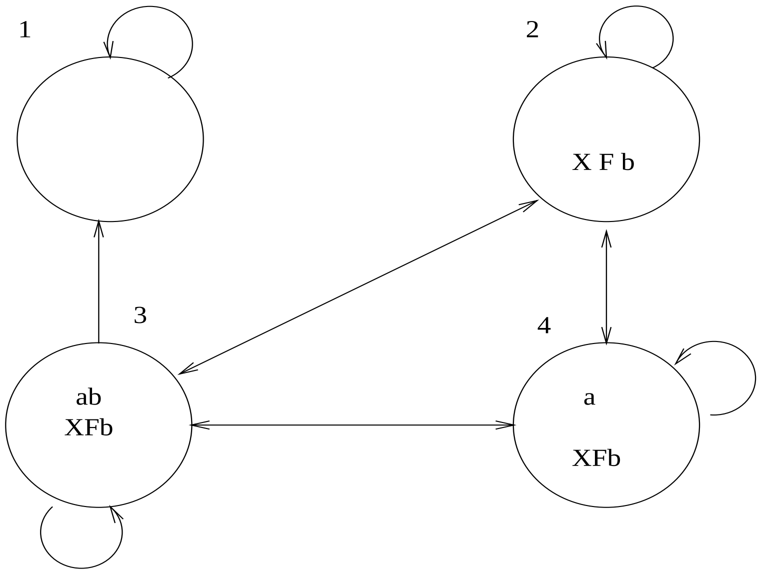

Figure 6.6: Tableau with problem for animation

kind of models for arbitrary formulas.

The "desired" model and the one generated via tableau construction recognize the same language, i.e, they are equivalent from an LTL point of view. However, they have a different branching structure. For instance the CTL formula $AG(\neg a \rightarrow (EX a \wedge EX \neg a))$ discriminates them: it is false in the first model, but true in the second. For this reasons, it is necessary to investigate whether the limitations of the tableau procedure are specific to our construction, or intrinsic of the usage of LTL formulas.

Chapter 7

Conclusions and Future Work

We have defined a formal language specifically suited for the description of early requirements models, and a tool which supports its formal analysis. The novelty of our approach lies in adapting Formal Methods techniques — which rely mostly on design-inspired specification languages — so that they can be used within a framework for early requirements modeling and analysis.

The main contribution of this thesis consists of showing that formal analysis techniques are useful during development phases that were once considered to be informal by nature. In particular, we demonstrate that it is feasible to apply model checking techniques during the early requirements phase of software development. First, we were able to verify interesting properties in a particular example which, although simple, is cast in terms of the concepts used for early requirements. Second, the case study reveals that interesting feedback can be obtained from the tool even for models with a reduced number of instances. This is particularly relevant, since the applicability of model checking to models with a large number of instances is limited by the state explosion problem.

Clearly, the approach should be applied to more examples before a comprehensive *methodology* can be defined. However, from the experience gained in our case study, we can suggest the following steps as a guide for the application of our approach:

1. Produce an initial model of the system as an i* SD diagram.
2. Produce a preliminary Formal Tropos specification.
   (a) Give a textual representation of the i* diagram, and introduce attributes and non-intentional entities.
   (b) Introduce **creation** and **fulfillment** constraint properties informally.
   (c) Give initial specifications for the constraints.
3. Use T-Tool in order to correct and fine-tune the specification.
   (a) Collect scenarios from the stakeholders.
   (b) Write possibilities and assertions from these scenarios.
   (c) Verify whether the possibilities and assertions are true for the valid scenarios of the specification.
   (d) In case of a false assertion or possibility
      i. For an assertion, evaluate the counterexample scenario with the stakeholders. Detect a constraint that conflicts with the counterexample.
      ii. For a possibility, detect a constraint that conflicts with it.
      iii. Reformulate the constraint in order to resolve the conflict.

The approach does not attempt to be complete; rather, it aims to help the stakeholders clarify ideas about their own goals. Furthermore, it is *defensive* and *iterative*. It is defensive, because T-Tool responds with counterexamples to invalid assertions, and the stakeholders must answer back by either justifying the assertion or correcting the specification. It is iterative, because there is a continuous interaction between the user and T-Tool. This interaction is based on two different kinds of scenarios. First, the stakeholders provide scenarios in the form of Formal Tropos **assertion** and **possibility** properties. Then, T-Tool responds with *automatically* generated scenarios that correspond to examples and counterexamples. The answers of T-Tool trigger a revision of the requirements from the stakeholders, and may lead to the modification of the specification, or the proposal of new **assertion** or **possibility** properties.

7.1 Future Work

The results of this thesis can be extended in several directions. The following is an enumeration of some of the points that we consider should be pursued in the near future.

7.1.1 Case studies

- The approach should be applied to more complex case studies, in order to obtain an exact evaluation of its scalability to real applications. It is also important to overcome a limitation of our case study, namely the fact the we were simultaneously playing the role of analyst and stakeholder.

- Experimentation should be conducted in order to characterize the limitations of the approach in terms of the maximum number of instances that an analyzable model can have. For instance, for our case study, we might produce a detailed evaluation of the running time of the specification with different numbers of instances for each class.

7.1.2 Scope of the approach

- The approach should be extended to consider the other phases of software development (late requirements, architectural and detailed design, etc.) in the context of the *Tropos* project. In order to achieve this, it is necessary to incorporate new concepts to our approach, such as *goal decomposition*, *task dependencies*, and *goal* *operationalization*. To this end, it might be possible to adapt some of the techniques already developed in KAOS to our own framework.

- It is necessary to give a more agent-oriented semantics to Formal Tropos. For instance, the underlying logic might be enhanced to cope with intentional operators, as in BDI logics [RG95]. With respect to the implementation of the tool, it might be interesting to investigate the possibility of representing actors as concurrently executing NuSMV processes.

- The approach should provide heuristics that take advantage of the information available in the initial i* diagrams. Some of this information (e.g, softgoals that are impossible to formalize) does not appear in the Formal Tropos specification. However, it can be used to provide guidelines to the analyst and stakeholders on how to correct an specification whenever an **assertion** or **possibility** does not hold.

7.1.3 T-Tool and model checking

- The user interaction aspects of the tool should be enhanced. For instance, at the moment T-Tool shows the evolution of the system in the tabular format shown in Figure 6.5. We are investigating different ways to make the traces produced by the tool more readable to the users. A first step might involve producing traces similar to the ones shown in Chapter 4.

  Furthermore, at the moment, the state exploration offered by the animator and the example and counterexamples shown by T-Tool are given in terms of the Intermediate Language; the user, however, expects to explore the system at the level of the Formal Tropos specification. The tool should take care of performing this inverse translation, and present the scenarios in a form that results convenient to the user.

- So far, we have mostly adapted the verification techniques of NuSMV to the new domain. However, there is much work to be done on adapting other techniques from the Formal Methods community. For instance, we should consider the application of *compositional verification* techniques (e.g, [PDH99]) in order to increase the scalability of the approach to larger systems. These techniques attempt to alleviate the state explosion problem by dividing the problem into modules, checking them separately, and finally verifying that their combination results in a correct system.

Bibliography

[ADS00] A. Anton, J. Dempster, and D. Siege. Deriving goals from a use-case based requirements specification for an electronic commerce system. *Submitted to the Sixth Intl. Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ)*, 2000.

[AG94] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 71–80. IEEE Computer Society Press, 1994.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.

[BCM+92] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[Boe81] B. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[Boi95] P. Du Bois. *The Albert II Language. On the Design and Use of a Formal Specification Language for Requirements Analysis.* PhD thesis, Notre Dame de la Paix, Namur, Belgium, September 1995.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. The Addison-Wesley Object Technology Series. Addison-Wesley, 1999.

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

[BS93] J. Bowen and V. Stavidrou. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, July 1993.

[CCGR00] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[CGH94] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 415–427, Stanford, California, USA, 1994. Springer-Verlag.

[CGM+98] 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 of Computing*, 10:361–380, 1998.

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

[Che75] P. Chen. The Entity-Relationship model: Towards a unified view of data. In D. Kerr, editor, *Proceedings of the International Conference on Very Large Data Bases*, September 1975.

[CKM01] 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.

[CL90] P. Cohen and H. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 32(3), 1990.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[DeM78] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, 1978.

[Dub89] E. Dubois. A logic of action for supporting goal-oriented elaboration of requirements. *Proceedings Fifth International Workshop on Software Specification and Design*, May 1989.

[DvL96] R. Darimont and A. van Laamsweerde. Formal refinement patterns for goal-driven requirements elaboration. *Proc. FSE'4 - Fourth ACM SIGSOFT Symp. on the Foundations of Software Engineering*, October 1996.

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

[EDD94] P. Du Bois E. Dubois and F. Dubru. Animating formal requirements specifications of cooperative information systems. *Proc. 2nd Intl. Conf. on Co-operative Information Systems - CoopIS-94, Toronto (Canada)*, May 1994.

[EL86] E. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1986.

[Fea87] M. Feather. Language support for the specification and development of composite systems. *ACM Transactions on Programming Languages and Systems*, 9(2):198–234, April 1987.

[FH92] S. Fickas and R. Helm. Knowledge representation and reasoning in the design of composite systems. *IEEE Transactions on Software Engineering*, pages 470–482, June 1992.

[GBM86] S. Greespan, A. Borgida, and J. Mylopoulos. A requirements modeling language and its logic. *On Knowledge-Based Management Systems*, pages 471–502, 1986.

[GH93] J. Guttag and J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993. With S. Garland, K. Jones, A. Modet, and J. Wing.

[GLL99] G. De Giacomo, Y. Lesperance, and H. Levesque. ConGolog, a concurrent programming language based on the situation calculus: language and implementation. To be published, 1999.

[GW93] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, April 1993.

[Hen80] K. Heninger. Specifying software requirements for complex system: New techniques and their application. *IEEE Transactions in Software Engineering*, 6(1):2–13, January 1980.

[HJL96] 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.

[HSG+94] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen. Formal approach to scenario analysis. *IEEE Software*, pages 33–41, March 1994.

[Hun98] A. Hunter. Paraconsistent logics. *Handbook of Defeasible Reasoning and Uncertain Information*, 2, 1998.

[LHH94] N. Levenson, M. Heimdahl, and H Hildreth. Requirements specification for process-control systems. *IEEE Transaction in Software Engineering*, 20(9):684–706, September 1994.

[McM93] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[MCN92] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Trans. on Software Engineering*, 18(6):483–497, June 1992.

[MCY99] J. Mylopoulos, L. Chung, and E. Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, pages 31–37, Jnauary 1999.

[MH79] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of Artificial Intelligence. *Machine Intelligence*, 4:463–502, 1979.

[MP89] Z. Manna and A. Pnueli. The anchored version of the temporal framework. *Linear time, branching time, and partial order in logics and models for concurrency, Lecture Notes in Computer Science*, 345:201–284, 1989.

[NE00] B. Nuseibeh and S. Easterbrook. Requirements Engineering: a roadmap. In *ICSE - Future of SE Track*, pages 35–46, 2000.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system, 1992.

[PDH99] C. Pasareanu, M. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *SPIN*, pages 168–183, 1999.

[PTA94] C. Potts, K. Takahashi, and A. Anton. Inquiry-based requirements analysis. *IEEE Software*, 11(2):21–32, March 1994.

[RG95] A. Rao and M. Georgeff. BDI agents: From theory to practice. *Proceedings of the First International Conference on MultiAgent Systems*, pages 312–319, June 1995.

[Ros77] D. Ross. Structured Analysis (SA): A language for communicating ideas. *IEEE Transactions on Software Engineering*, 3(1):16–34, 1977.

[SG96] M. Shaw and B. Gaines. Requirements acquisition. *Software Engineering Journal*, 11(3):149–165, 1996.

[Spi92] J. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 1992.

[vB95] J. van Benthem. *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4, chapter Temporal Logic. D. Gabbay and C. Hogger and J. Robinson, 1995.

[vLDL98] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven Requirements Engineering. *IEEE Transaction on Software Engineering*, November 1998.

[vLL00] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented Requirements Engineering. *IEEE Transactions on Software Engineering, Special Issue on Exception Handling*, 2000.

[vLW98] A. van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*, 1998.

[Wan01] Xiyun Wang. Agent-oriented requirements engineering using ConGolog and i*. Master's thesis, York University, 2001.

[YM94] E. Yu and J. Mylopoulos. Towards modelling strategic actor relationships for information systems development – with examples from business process reengineering. *Proceedings of the 4th Workshop on Information Technologies and Systems*, pages 21 – 28, December 1994.

[Yu95] E. Yu. *Modelling Strategic Relationships for Process Reengineering.* PhD thesis, University of Toronto, Toronto, Canada, 1995.