Formal Tropos: language and semantics

A. Fuxman[1] R. Kazhamiakin[2] M. Pistore[2,3] M. Roveri[3]

[1] Department of Computer Science, University of Toronto, Canada
[2] Department of Information and Communication Technology, University of Trento, Italy
[3] ITC-irst, Trento, Italy

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

Version 1.0: November 4, 2003

Abstract

In this document we provide a description of the Formal Tropos language and its semantics.

Contents

1 Introduction

Early requirements engineering is the phase of the software development process that models and analyzes the operational environment where a software system will eventually function. In order to analyze such environment, it is necessary to investigate the objectives, business processes, and interdependencies of different stakeholders. At least in principles, the understanding of these "strategic" aspects of the operational environment is necessary to motivate and direct the development of the software system. Although errors and misunderstandings at this stage are both frequent and costly, early requirements engineering is usually done informally (if at all).

Formal methods have been successfully applied to the verification and certification of software systems. In several industrial fields, formal methods are becoming integral components of standards [2]. However, the application of formal methods to early requirements is by no means trivial. Most formal techniques have been designed to work (and have been mainly applied) in later phases of software development, e.g., at the architectural and design level. As a result, there is a mismatch between the concepts used for early requirements specifications (such as actors, goals, needs…) and the constructs of formal specification languages such as Z [15], SCR [12], TRIO [10, 14].

Formal Tropos (hereafter FT) is a formal framework that adapts results from the Requirements Engineering and Formal Methods communities to facilitate the precise modeling and analysis of early requirements.

The FT framework supports the automatic verification of early requirements specified in a formal modeling language. This framework is part of a wider on-going project called *Tropos*, whose aim is to develop an agent-oriented software engineering methodology, starting from early requirements. The methodology is to be supported by a variety of analysis tools based on formal methods.

The FT language offers all the primitive concepts of *i** [17] (such as actors, goals, and dependencies among actors), but supplements them with a rich temporal specification language inspired by KAOS [6, 7]. The *i** notations allow for the description of the "structural" aspects of the early requirements model, for instance in terms of the network of relationships and dependencies among actors. FT permits to represent also the "dynamic" aspects of the model, describing for instance how the network of relationships evolves over time. In FT one can define the circumstances under which a given dependency among two actors arises, as well as the conditions that permit to consider the dependency fulfilled. In our experience, representing and analyzing these dynamic aspects allows for a more precise understanding of the early requirements model, and reveals gaps and inconsistencies that are by no means trivial to discover without the help of formal analysis tools.

A tool, called T-Tool, has been developed to support the analysis of FT specifications. The T-Tool is based on the state-of-the-art symbolic model checker NuSMV [4]. It translates automatically an FT specification into an Intermediate Language (hereafter IL) specification that could potentially link FT with different verification engines. The IL representation is then automatically translated into NuSMV, which can then perform different kinds of formal analysis, such as consistency checking, animation of the specification, and property verification.

In this document we formally define the syntax and the semantics of the FT and of the IL languages. This document is structured as follows. Section 2 will describe the grammar and the concepts of the Formal Tropos language. In Section 3 we will describe the grammar and the semantic of the Intermediate Language. In Section 4 we will define the formal semantics of Formal Tropos in terms of its translation in the Intermediate Language. In Section 5 we describe the architecture and the functionalities of the tool named T-Tool, developed to support the Formal Tropos methodology. Finally in Section 6 we will draw some conclusions and we briefly discuss the extensions we have in mind.

2 Formal Tropos

Although a graphical notation such as *i** is valuable for human communication, its models are not detailed enough to be used as a 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.

2.1 Syntax of the FT Language

In this section, we explain the different elements of a Formal Tropos specification. The complete grammar is given in Figure 1.

A specification in Formal Tropos consists of a sequence of declarations of *entities*, *actors*, *goals*, *tasks*, *resources*, *dependencies*, and *global properties*. Declarations for entities, actors, goals, resources, task, softgoals 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, goals, tasks, resources, softogoals 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 primitive (integer, boolean, etc.) or correspond to other classes (entities, actors, goals, tasks, softgoals, resources or dependencies) of the specification. Facets represent frequently used properties of attributes. The only facet currently supported is **constant**, that represents the fact that the value of the attribute cannot change after its initialization. Other possible facets are: **optional**, which means that the attribute might assume no value; **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.

Like entities, *actors* have attributes, and **creation** and **invariant** properties.

Goals, task, resources, softgoals and dependencies have a **mode** and a set of **fulfillment** properties, which will be explained shortly. If the actor is not able to fulfill a certain goal by itself, then the goal should be refined into sub-goals, operationalized in tasks and delegated to other (dependee) actors via dependencies.

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

/ The outer layer */*

specification := (*entity* | *actor* | *int-element* | *dependency* | *global-properties*)*

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

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

int-element := *type name mode* **Actor** *name* [*attributes*] [*creation-properties*] [*invar-properties*] [*fulfill-properties*]

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

type := (**Goal** | **Softgoal** | **Task** | **Resource**)

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

/ Attributes */*

attributes := **Attribute** *attribute*$^+$

attribute := *facets name* : *sort*

facets := [**constant**] . . .

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

/ The inner layer */*

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

creation-property := *property-category event-category temporal-formula*

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

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

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

fulfill-property := *property-category event-category temporal-formula*

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

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

/ Global properties */*

global-properties := **Global** *global-property*$^+$

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

Figure 1: The Formal Tropos grammar.

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.

Intentional elements are characterized by a *mode*, which declares the attitude of the involved 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 an intentional object is controlled by three kinds of properties. **Creation** properties determine the moment in which a new instance of the object can be created; **fulfillment** properties must hold in order to consider that an intentional object is fulfilled; and **invariants** represent conditions that should be true along the life of the object. We remark again that the meaning of the fulfillment conditions changes according to the modality of the corresponding intentional element. 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.

2.2 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 Section 3.

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

$$\begin{array}{rcl} f & := & \mathbf{X}f \mid \mathbf{F}f \mid \mathbf{G}f \mid f\ \mathbf{U}\ f \mid \mathbf{Y}f \mid \mathbf{H}f \mid \mathbf{P}f \mid f\ \mathbf{S}\ f \mid \\ & & \mathbf{JustFulfilled}(t) \mid \mathbf{JustCreated}(t) \mid \mathbf{Fulfilled}(t) \end{array} \tag{1}$$

The meaning of the operators is the following:

- $\mathbf{X}f$ (next state)
- $\mathbf{F}f$ (eventually)
- $\mathbf{G}f$ (henceforth, always in the future)
- $f_1\ \mathbf{U}\ f_2$ (until)
- $\mathbf{Y}f$ (previous state)
- $\mathbf{P}f$ (sometimes in the past)
- $\mathbf{H}f$ (always in the past)
- $f_1\ \mathbf{S}\ f_2$ (since)

In temporal formulas we consider also some primitive predicates, such as **JustCreated** (t) (an instance of object t is being created at this point in time), and **JustFulfilled** (t) (an object t is being fulfilled at the current point in time). Another predicate, **Fulfilled** (t) 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** (t) is made true at the moment of the fulfillment and stays true forever. For a **maintain** dependency, **Fulfilled** (t) 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).

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.

$$f := \mathbf{Forall}\ x : \textit{sort}\ (f) \mid\ \mathbf{Exists}\ x : \textit{sort}\ (f) \mid \cdots \tag{2}$$

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

$$f := (f \mid f) \mid (f \,\&\, f) \mid !f \mid f \rightarrow f \mid f \leftrightarrow f \mid \cdots \tag{3}$$

$$f := t = t \mid t \,!= t \mid \cdots \tag{4}$$

Terms can be constants (c), variables (x), attributes of an object ($t.a$).

$$t \ := \quad c \mid x \mid t.a \mid \textbf{self} \mid \textbf{actor} \mid \textbf{depender} \mid \textbf{dependee} \tag{5}$$

Valid formulas must satisfy some constraints. First, they must be *well-sorted* (e.g., comparison can be performed among terms of the same sort). Second, each variable x that appears in a formula must respect one of the following rules:

- 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, goal, task, resource or dependency that contains the property;
- x is the identifier **depender** or the identifier **dependee** (only applicable to properties inside a dependency);
- x is the identifier **actor** (only applicable to properties inside a goal, softgoal, task or resource); this identifier refers to the actor of the current goal, task, resource, softgoal.

2.3 Obsolete Syntax

The declaration of **dependency** element has changed in the new version of the language. The old version of the grammar definition for **dependency** is the following:

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

This syntax is used in several FT case studies and examples, and the T-Tool supports it. In the future the old version of the syntax will be deprecated.

The facets **for depender**, **for dependee**, **for domain**, described by the grammar rules below, which denote the origin of a class property, are simply ignored in the translation.

creation-properties := *property-category event-category property-origin temporal-formula*
invar-property := *property-category property-origin temporal-formula*
fulfill-properties := *property-category event-category property-origin temporal-formula*
property-origin := [**for depender** | **for dependee** | **for domain**]

These facets will probably be removed form future version of the language.

3 The Intermediate Language

In this section we present an Intermediate Language, a smaller language, which allows for a simpler formal semantics. Furthermore, the 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.

3.1 The Syntax of IL

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
$a_1 : s_1$
$\dots$
$a_n : s_n$

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**, **boolean**, ...) or correspond to class names.

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 is described by the following rules:

$$f \;:=\; \mathbf{X}f \mid \mathbf{F}f \mid \mathbf{G}f \mid f\ \mathbf{U}\ f \mid \mathbf{Y}f \mid \mathbf{H}f \mid \mathbf{P}f \mid f\ \mathbf{S}\ f \tag{6}$$

$$f \;:=\; \textbf{Forall}\ x : \textit{sort}\ (f) \mid\ \textbf{Exists}\ x : \textit{sort}\ (f) \mid \cdots \tag{7}$$

$$f \;:=\; (f \mid f) \mid (f\ \&\ f) \mid\ !f \mid f \rightarrow f \mid f \leftrightarrow f \mid \cdots \tag{8}$$

$$f \;:=\; t = t \mid t\,! = t \mid \cdots \tag{9}$$

$$t \;:=\; c \mid x \mid t.a \tag{10}$$

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.

3.2 Formal definition of the class signature

3.2.1 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 [16] 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** $x : s\ (\mathbf{G}\phi(x))$, 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** $x : s\ (\mathbf{H}\phi(x))$, 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** $x : s\ (\mathbf{H}\phi(x))$ 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 $\mathbf{H}\phi(x)$ formula to the past snapshots where all the objects referred in ϕ existed.

The interpretation that we give to formula **Forall** $x : s\ (\mathbf{P}\phi(x))$ 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 $\mathbf{P}\phi$ we require ϕ to have held in a past snapshot where all the objects referred in ϕ existed.

The strong previous state operator $\mathbf{Y}\phi$ is true only if all the objects referred in ϕ exist in the previous snapshot (and ϕ holds in that snapshot).

3.2.2 Basic sorts

Basic sorts correspond to the elementary data types used in the Intermediate Language, such as integer, booleans, etc. We assume that a given set S_0 of basic sorts is defined and a fixed interpretation domain D_0^s is associated to each basic sort $s \in S_0$.

3.2.3 Class signature

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

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

- C is a set of *class identifiers*;
- for each class $c \in C$, set A_c describes the attributes of class c.

All attributes must have a value all along the life of their corresponding

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. The function 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.

3.3 Formal definition of the logic specification

3.3.1 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 $c \in C$ 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 $t \in \mathcal{T}_{\Sigma,\mathrm{V}}^s$, 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 $c \in C$ 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
  - $\mathcal{I}_{\mathcal{W},\Phi}[t.a] = \mathcal{I}_{c,a}(\mathcal{I}_{\mathcal{W},\Phi}[t])$, 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.

3.3.2 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 $t \in \mathcal{T}_{\Sigma,\mathrm{V}}^{boolean}$, then $t \in \mathcal{F}_{\Sigma,\mathrm{V}}$.
- If $t_1, t_2 \in \mathcal{T}_{\Sigma,\mathrm{V}}^s$ 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 $!\, f$, $f_1 \,\&\, f_2 \in \mathcal{F}_{\Sigma,\mathrm{V}}$.
- If $f, f_1, f_2 \in \mathcal{F}_{\Sigma,\mathrm{V}}$ the also $\mathbf{X}f, \mathbf{Y}f, f_1 \;\mathbf{U}\; f_2, f_1 \;\mathbf{S}\; f_2 \in \mathcal{F}_{\Sigma,\mathrm{V}}$.
- 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:

- $ff \overset{\mathrm{def}}{=} !\ tt$
- $f_1 \mid f_2 \overset{\mathrm{def}}{=}\ !\ (!\ f_1 \,\&\, !\ f_2)$
- $f_1 \to f_2 \overset{\mathrm{def}}{=}\ !\ f_1 \mid f_2$
- $f_1 \leftrightarrow f_2 \overset{\mathrm{def}}{=} (f_1 \to f_2) \,\&\, (f_2 \to f_1)$
- **Exists** $x : s\ (f) \overset{\mathrm{def}}{=}\ !$ (**Forall** $x : s\ (!\ f)$)
- $\mathbf{F}f \overset{\mathrm{def}}{=} tt \;\mathbf{U}\; f$
- $\mathbf{G}f \overset{\mathrm{def}}{=} !\ \mathbf{F}!\ f$
- $\mathbf{P}f \overset{\mathrm{def}}{=} tt \;\mathbf{S}\; f$
- $\mathbf{H}f \overset{\mathrm{def}}{=} !\ \mathbf{P}!\ f$

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.
- $\mathcal{R}, i, \Phi \models !\, f$ if not $\mathcal{R}, i, \Phi \models f$.
- $\mathcal{R}, i, \Phi \models f_1 \& f_2$ if $\mathcal{R}, i, \Phi \models f_1$ and $\mathcal{R}, i, \Phi \models f_2$.
- $\mathcal{R}, i, \Phi \models \mathbf{X} f$ if $\mathcal{R}, i{+}1, \Phi \models f$.
- $\mathcal{R}, i, \Phi \models f_1 \ \mathbf{U}\ f_2$ if there is some j, with $i \leq j$, such that $\mathcal{R}, j, \Phi \models f_2$ and, for each l with $i \leq l < j$, $\mathcal{R}, l, \Phi \models f_1$.
- $\mathcal{R}, i, \Phi \models \mathbf{Y} f$ if $i > 0$, $\Phi(v) \in D_{i-1}^s$ for each $v : s \in \mathrm{FV}(f)$, and $\mathcal{R}, i{-}1, \Phi \models f$.
- $\mathcal{R}, i, \Phi \models f_1 \ \mathbf{S}\ f_2$ 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 $\mathcal{R}, 0, \emptyset \models f$.

3.4 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.

4 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.

4.1 From the Formal Tropos outer layer

4.1.1 Class signature

Each Formal Tropos class (entity, actor, goal, softgoal, resource, task, or dependency) is translated to a corresponding Intermediate Language class as follows:

Rule 1 (class signature) *For each Formal Tropos class of name C, add the following class to the Intermediate Language class signature*

> ***CLASS*** C
> $a_1 : s_1$
> $\ldots$
> $a_n : s_n$

where each variable a_i corresponds to some attribute of the Formal Tropos class, and each sort s_i denotes the type of the attribute.

We also add other attributes which, although not present in the Formal Tropos specification, are necessary for the formalization of its semantics. In particular, for each class corresponding to an intentional element (goal, softgoal, task, resource, or dependency) , we add the boolean attribute fulfilled. The intuitive meaning is that this attribute becomes true when the intentional element is fulfilled. For each class corresponding to a goal, softgoal, task, or resource we include the attribute actor. The sort of this attribute is the actor mentioned in the **Actor** clause of the class. For each dependency we include the attributes depender and dependee. Their sorts are the actors mentioned in the **Depender** and **Dependee** clauses of the dependency class.

Rule 2 (fulfilled attribute) *For each goal, softgoal, task, resource, and dependency of the Formal Tropos specification, add the attribute* fulfilled *of sort* ***boolean*** *to the corresponding Intermediate Language class.*

Rule 3 (actor attribute) *For each goal, softgoal, task, and resource of the Formal Tropos specification containing clause "**Actor** A", add the attribute* actor *of sort A to the corresponding Intermediate Language class.*

Rule 4 (depender and dependee attributes) *For each dependency of the Formal Tropos specification containing clauses "**Depender** Dr" and "**Dependee** De", add the attributes* depender *of sort Dr and* dependee *of sort De to the corresponding Intermediate Language class.*

4.1.2 Logic specification

An Intermediate Language class signature is not able to capture the complete semantics of the Formal Tropos outer layer. The following rules formalize those aspects of the semantics of classes that are implicit in Formal Tropos.

In the following rule, the meaning of the **constant** facet in given by appropriate constraints in the Intermediate Language logic specification.

Rule 5 (constant facet) *For each constant attribute a of type t declared in class C, we add the Intermediate Language constraint*

$$\textbf{\textit{Forall}}\ c : C\ (\textbf{\textit{Forall}}\ v : t\ (c.a = v \rightarrow \textbf{\textit{X}}(c.a = v)))$$

Also the attributes actor, depender, and dependee added by rules 3 and 4 are constant attributes.

Rule 6 (additional attributes are constant) *For each class C that corresponds to a goal, softgoal, task, or resource declaration in the Formal Tropos specification containing clause "**Actor** A" we add the Intermediate Language constraint*

$$\textbf{\textit{Forall}}\ c : C\ (\textbf{\textit{Forall}}\ a : A\ (c.actor = a \rightarrow \textbf{\textit{X}}(c.actor = a)))$$

*For each class C that corresponds to a dependency declaration in the Formal Tropos specification containing clauses "**Depender** Dr" and "**Dependee** De" we add the Intermediate Language constraints*

$$\textbf{\textit{Forall}}\ c : C\ (\textbf{\textit{Forall}}\ d : Dr\ (c.depender = d \rightarrow \textbf{\textit{X}}(c.depender = d)))$$

and

$$\textbf{\textit{Forall}}\ c : C\ (\textbf{\textit{Forall}}\ d : De\ (c.dependee = d \rightarrow \textbf{\textit{X}}(c.dependee = d)))$$

In the next rule we formalize the assumption that, once fulfilled, goals, softgoals, tasks, resources, and dependencies remain in that state forever, regardless of the future evolution of the system.

Rule 7 (fulfillment forever) *For each class C that corresponds to a goal, softgoal, task, resource, or dependency declaration in the Formal Tropos specification, we add the following constraint*

$$\textbf{\textit{Forall}}\ c : C\ (c.fulfilled \rightarrow \textbf{\textit{X}}c.fulfilled)$$

4.2 From the Formal Tropos inner layer

We now explain the translation of the properties of a Formal Tropos specification into Intermediate Language formulas.

In a preliminary step, we replace all Formal Tropos primitive predicates by variables of the Intermediate Language. 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 Intermediate Language translations. Notice that we will continue using **JustFulfilled** and **JustCreated** in the Intermediate Language formulas, but they should be considered as just macros.

Rule 8 (substitutions for primitive predicates) *For every Formal Tropos formula,*

- *every occurrence of predicate* ***Fulfilled***(t)*, is replaced by* $t.fulfilled$*;*
- *every ocurrence of predicate* ***JustFulfilled***(t)*, is replaced by:*

$$t.fulfilled \,\&\, !\,\mathbf{Y}t.fulfilled;$$

- *every occurrence of predicate* ***JustCreated***(t)*, is replaced by* $(t = t \,\&\, !\,\mathbf{Y}(t = t))$.

Global properties only require the substitutions described in rule 8.

Rule 9 (global properties) *For every formula* f *corresponding to a Formal Tropos global property, add formula* $\widehat{f}$ *to the Intermediate Language specification, where* $\widehat{f}$ *is the formula obtained by applying rule 8.*

Unlike Formal Tropos, the formulas of the Intermediate Language are no longer associated to a particular class, and are not anchored to a particular event in the life of the class. This difference has to be taken into account for the translation of class properties (i.e., properties that are not global). The following rules describe the translation of these properties. In particular, the formulae obtained after the translation refer to a new variable c corresponding to the Formal Tropos class they refer to. Class c universally quantified if the property corresponds to a **constraint** or an **assertion**, and existentially quantified if it corresponds to a **possibility**.

Rule 10 (closure w.r.t. the class) *For very formula corresponding to a property of a Formal Tropos class* C*:*

- *every occurrence of* ***self*** *is replaced with variable* c*;*
- *every occurrence of a free attribute* a_i*, i.e., every attribute* a_i *that is not in the scope of some* $t.\_$ *prefix, is replaced with* $c.a_i$*;*

In the rest of this section, we denote with $\widehat{f}$ the formula obtained from f via the substitutions of rules 8 and 10.

In the case of invariants, the translation is defined by the following rule.

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

$$\textbf{\textit{Forall}}\ c : C\ (\widehat{f}) \qquad \textit{if } f \textit{ is a } \textbf{\textit{constraint}} \textit{ or } \textbf{\textit{assertion}}$$
$$\textbf{\textit{Exists}}\ c : C\ (\widehat{f}) \qquad \textit{if } f \textit{ is a } \textbf{\textit{possibility}}$$

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 Formal Tropos 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 f corresponding to a* ***creation condition*** *property, add the formula:*

$$\textbf{\textit{Forall}}\ c : C\ (\textbf{\textit{JustCreated}}(c) \rightarrow \widehat{f}) \qquad \textit{if } f \textit{ is a } \textbf{\textit{constraint}} \textit{ or } \textbf{\textit{assertion}}$$
$$\textbf{\textit{Exists}}\ c : C\ (\textbf{\textit{JustCreated}}(c)\ \&\ \widehat{f}) \qquad \textit{if } f \textit{ is a } \textbf{\textit{possibility}}$$

The translation for **creation trigger** s 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 f 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 f and $A_1, A_2, \ldots, A_n$ are their sorts:*

$$\textbf{\textit{Forall}}\ a_1 : A_1\ (\textbf{\textit{Forall}}\ a_2 : A_2\ (\textbf{\textit{Forall}}\ a_n : A_n(f \rightarrow \textbf{\textit{Exists}}\ c : C(c.a_1 = a_1\ \&\ c.a_2 = a_2 \ldots c.a_n = a_n))))$$

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 Intermediate Language 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 f corresponding to a* ***fulfillment*** *property of a class C with* ***achieve*** *modality, we add the formula*

$$\textbf{\textit{Forall}}\ c : C\ (\textbf{\textit{JustFulfilled}}(c) \rightarrow \widehat{f})) \qquad \textit{if } f \textit{ is a } \textbf{\textit{constraint condition}} \textit{ or } \textbf{\textit{assertion condition}}$$
$$\textbf{\textit{Forall}}\ c : C\ (\widehat{f} \rightarrow c.fulfilled)) \qquad \textit{if } f \textit{ is a } \textbf{\textit{constraint trigger}} \textit{ or } \textbf{\textit{assertion trigger}}$$
$$\textbf{\textit{Exists}}\ c : C\ (\textbf{\textit{JustFulfilled}}(c)\ \&\ \widehat{f})) \qquad \textit{if } f \textit{ is a } \textbf{\textit{possibility}}$$

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

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

Forall $c : C\ (c.fulfilled \rightarrow (\boldsymbol{G} \wedge \boldsymbol{H})\widehat{f})$	*if f is a **constraint condition** or **assertion condition***
Forall $c : C\ ((\boldsymbol{G} \wedge \boldsymbol{H})\widehat{f} \rightarrow c.fulfilled)$	*if f is a **constraint trigger** or **assertion trigger***
Exists $c : C\ (c.fulfilled\,\&(\boldsymbol{G} \wedge \boldsymbol{H})\widehat{f})$	*if f is a **possibility***

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

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

Forall $c : C\ (c.fulfilled \rightarrow (\boldsymbol{G} \wedge \boldsymbol{H})!\,\widehat{f})$	*if f is a **constraint condition** or **assertion condition***
Forall $c : C\ ((\boldsymbol{G} \wedge \boldsymbol{H})!\,\widehat{f} \rightarrow c.fulfilled)$	*if f is a **constraint trigger** or **assertion trigger***
Exists $c : C\ (c.fulfilled\,\&(\boldsymbol{G} \wedge \boldsymbol{H})!\,\widehat{f})$	*if f 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 f of a **fulfillment** property of a class C with **achieve & maintain** modality, we add the formula*

Forall $c : C\ (c.fulfilled \rightarrow \boldsymbol{G}\widehat{f})$	*if f is a **constraint condition** or **assertion condition***
Forall $c : C\ (\boldsymbol{G}\widehat{f} \rightarrow c.fulfilled)$	*if f is a **constraint trigger** or **assertion trigger***
Exists $c : C\ (c.fulfilled\,\&\,\boldsymbol{G}\widehat{f})$	*if f 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 f be a formula of a Formal Tropos specification and let f' be the corresponding formula added to the Intermediate Language specification according to Rules 9 or 11–18.*

- *If f belongs to a property which has the facet **constraint** (or no* property-category *facet), then $f' \in \mathcal{C}$, where $\mathcal{C}$ is the set of all constraints of the specification.*
- *If f belongs to a property which has facet **assertion**, then $f' \in \mathcal{A}$, where $\mathcal{A}$ is the set of all assertions about the specification.*
- *Finally, if f belongs to a property with **possibility** facet, then $f' \in \mathcal{P}$, where $\mathcal{P}$ is the set of all possibilities for the specification.*

4.3 Discussion on the translation

There are some aspects of the translation that deserve further comments.

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 3.4, 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.

Another point is that **creation trigger** properties are *not* translated by adding the formula **Forall** $c : C\ (\widehat{f} \rightarrow$ **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 adopted 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 $a_1, a_2, \ldots, a_n$ are the free variables of f and $A_1, A_2, \ldots, A_n$ are their sorts:

$$\textbf{Forall}\ a_1 : A_1\ (\textbf{Forall}\ a_2 : A_2\ (\ldots\ \textbf{Forall}\ a_n : A_n$$
$$(f \rightarrow \textbf{Exists}\ c : C(c.a_1 = a_1\ \&\ c.a_2 = a_2 \ldots c.a_n = a_n))))$$

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.

As a final note, we remark that he 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.).

5 The T-Tool

In this section we describe the T-Tool, a tool that supports the analysis of FT specifications. The T-Tool is available at the URL `http://dit.unitn.it/~ft/`.

The T-Tool is based on finite-state model checking [5]. The advantages of model checking with respect to other formal techniques (e.g., theorem proving; see [11] for a comparison) are that it allows for an automatic verification of a specification and that (counter-)example traces are produced as witnesses of the validity (or invalidity) of the specification. A limit of finite-state model checking is that it requires a model with a finite number of states. This forces to define an upper bound to the number of class instances that can be created during model checking.

The T-Tool input is an FT specification along with parameters that specify the upper bounds for the class instances. On the basis of this input, the T-Tool builds a finite model that represents all possible behaviors of the domain that satisfy the constraints of the specification. The T-Tool then verifies whether this model exhibits the desired behaviors. The T-Tool provides different verification functionalities, including interactive animation of the specification, automated consistency checks, and validation of the specification against possibility and assertion properties. The verification phase usually generates feedback on errors in the FT specification and hints on how to fix them. The verification phase iterates on each fixed version of the model, possibly with different upper bounds of the number of class instances, until a reasonable confidence on the quality of the specification has been achieved.

5.1 T-Tool functionalities

5.1.1 Animation

An advantage of formal specifications is the possibility to animate them. Through animation, the user can obtain immediate feedback on the effects of constraints. An animation session consists of an interactive generation of a valid scenario for the specification. Stepwise, the T-Tool proposes to the user next possible valid evolutions of the animation and, once the user has selected one, the system evolves the state of the animation. Animation allows for a better understanding of the specified domain, as well as for the early identification of trivial bugs and missing requirements that are often taken for granted, and are therefore difficult to detect in an informal setting. Animation also facilitates communication with stakeholders by generating concrete scenarios for discussing specific behaviors.

5.1.2 Consistency checks

Consistency checks are standard checks to guarantee that the FT specification is not self-contradictory. Inconsistent specifications occur quite often due to complex interactions among constraints in the specification, and they are very difficult to detect without the support of automated analysis tools. Consistency checks are performed automatically by the T-Tool and are independent of the application domain. The simplest consistency check verifies whether there is any valid scenario that respects all the constraints of the FT specification. Another consistency check verifies whether there exists a valid scenario where all the class instances specified by input parameters will be eventually created. This check aims at verifying whether these parameters violate any cardinality constraint in the specification. The T-Tool also checks whether there exists a valid scenario where all the instances of a particular goal or dependency will be eventually created and fulfilled, i.e., the fulfillment conditions for that goal or dependency are “compatible” with other constraints in the specification. Not all the consistency checks may be relevant for a given model. For instance, in a model it may be perfectly reasonable that there is no single scenario where instances are generated for all classes. In this case, this consistency check is excluded for the model under investigation.

5.1.3 Possibility checks

Possibility checks verify whether the specification is over-constrained, that is, whether we have ruled out scenarios expected by the stakeholders. When a **Possibility** property of the FT specification is checked, the T-Tool verifies that there are valid traces of the specification that satisfy the condition expressed in the possibility. The expected outcome of a possibility check is an example trace that confirms that the possibility is valid. In a sense, possibility checks are similar to consistency checks, since they both verify that the FT specification allows for certain desired scenarios. Their difference is that consistency is a generic formal property independent of the application domain, while possibility properties are domain-specific.

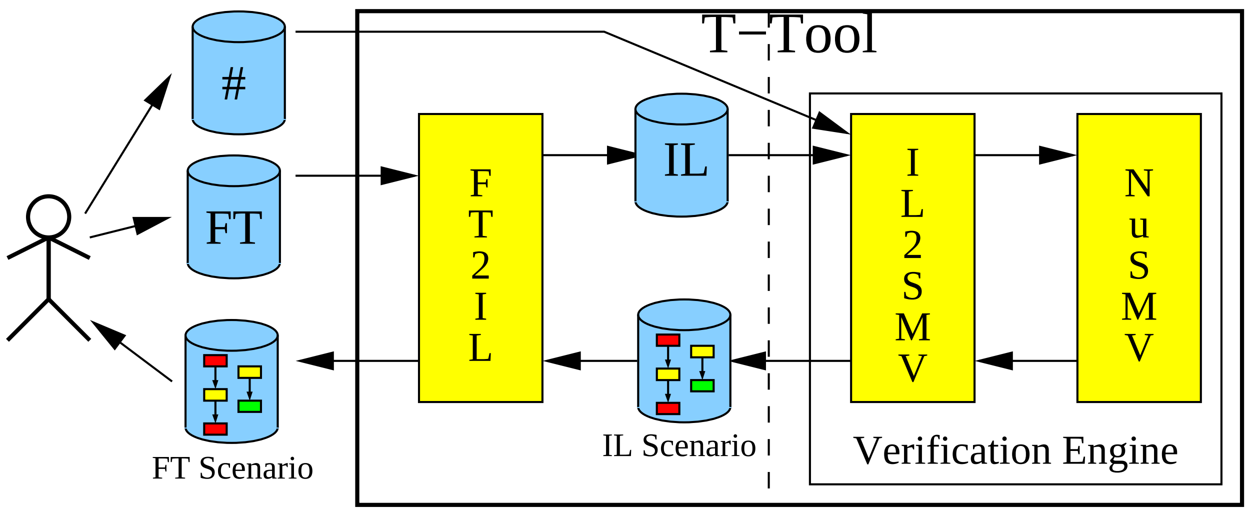

Figure 2: The T-Tool framework.

5.1.4 Assertion checks

The goal of **Assertion** properties is dual to that of possibilities. The aim is to verify whether the requirements are under-specified and allow for scenarios violating desired properties. Unsurprisingly, the behavior of the T-Tool in the case of assertion checks is dual to the behavior for possibility checks, namely, the tool explores all the valid traces and checks whether they satisfy the assertion property. If this is not the case, an error message is reported and a counter-example trace is generated. Such counter-examples facilitate the detection of problems in the FT specification that caused the assertion violation.

5.2 The T-Tool architecture

The T-Tool performs the verification of an FT specification in two steps (see Figure 2). In the first step, the FT specification is translated into an Intermediate Language (IL) specification. In the second step, the IL specification is given as input to the verification engine, which is built on top of the NuSMV model checker [4].

5.2.1 The role of IL

The IL plays a fundamental role in bridging the gap between FT and formal methods. First, IL is much more compact than FT, and therefore allows for a much simpler formal semantics. In fact, in the previous sections we showed how the formal semantics of FT is defined on the top of the semantics of IL, via the translation rules that map an FT specification into an IL specification. Second, IL, while more suitable to formal analysis, is still independent of the particular analysis techniques that we employ. For the moment, we have applied only model checking techniques; however, we plan to also apply techniques based on satisfiability or theorem proving. Finally, IL is rather independent of the particular constructs of FT. By moving to different domains, it will probably become necessary to "tune" FT, for instance by adding new modalities for the dependencies.

The formal approach described in this paper can be also applied to these dialects of FT, at the cost of defining a new translation. Furthermore, the IL can be applied to requirements languages that are based on a different set of concepts than those of FT, such as KAOS [6, 7].

5.2.2 The model checking verification engine

The actual verification is performed by NuSMV [4]. NuSMV implements several state-of-the-art model checking algorithms. It also provides an open architecture that facilitates the implementation of new algorithms and the customization of the verification process to the specific application domain.

NuSMV is based on symbolic model checking techniques. Symbolic techniques have been developed to reduce the effects of the state-explosion problem, thereby enabling the verification of large designs [5, 13]. NuSMV adopts symbolic model checking algorithms based on Binary Decision Diagrams (BDD) [3] and on propositional satisfiability (SAT) [1]. BDD-based model checking performs an exhaustive traversal of the model by considering all possible behaviors in a compact way. Such exhaustive exploration allows BDD-based model checking algorithms to conclude whether a given property is satisfied (or falsified) by the model. On the other hand, this exhaustive exploration makes BDD-based model checking very expensive for large models. SAT-based model checking algorithms look for a trace of a given length that satisfies (or falsifies) a property. SAT-based algorithms are usually more efficient than BDD-based algorithms for traces of reasonable length, but, if no trace is found for a given length, then it may still be the case that the property is satisfied by a longer trace. That is, SAT-based model checking verifies the satisfiability of a property only up to a given length, and is hence called Bounded Model Checking (BMC) [1]. The T-Tool exploits both BDD-based and SAT-based model checking.

6 Conclusions and Future Work

The FT language is continuously evolving to allow for capturing new requirements. Among the several extensions we are considering we list, the introduction of the **optional** facet for attributes, the support for sets, and extensions of the language to better characterize goal decomposition and means-end analysis.

7 Acknowledge

We would like to thank all the people that worked and is working on Formal Tropos. Part of the material presented in this document has been extracted from several documents, e.g. from Ariel's thesis [8], from the [9] journal paper.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the* 5^{th} *International Conference on Tools and Algorithms for the Construction*

and Analysis of Systems, number 1579 in Lecture Notes in Computer Science, pages 193–207, Amsterdam, The Netherlands, March 1999. Springer.

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

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

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

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

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

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

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

[9] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in Tropos. *Requirements Engineering*, 2003. To appear.

[10] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO, a logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 2(12):107–123, May 1990.

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

[12] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specification. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, 1996.

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

[14] A. Morzenti and P. San Pietro. Object-oriented logic specifications of time critical systems. *Transactions on Software Engineering and Methodologies*, 3(1):56–98, January 1994.

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

[16] 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.

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