T-Tool Tutorial

R. Kazhamiakin[1] M. Pistore[1,2] M. Roveri[2]

[1] Department of Information and Communication Technology,
University of Trento, Via Sommarive 14, I-38050, Trento, Italy
[2] ITC-irst, via Sommarive 18, I-38050, Trento, Italy
{pistore,raman}@dit.unitn.it roveri@irst.itc.it

ft@dit.unitn.it

Version 1.0: November 4, 2003

Abstract

In this document we provide a very short tutorial of the use of the T-Tool. For the semantics of the Formal Tropos language the reader can refer to the documents available from `http://www.dit.unitn.it/~ft`.

1 Introduction

T-Tool, the Formal Tropos tool, provides a framework for the effective formal analysis of FT specifications. It is based on finite-state model checking verification techniques and is built on top of NuSMV, an open-source model checker that implements state-of-the-art symbolic verification techniques. T-Tool allows for different kinds of analysis on a Formal Tropos specification. For instance, it allows for checking whether the specification is consistent, or whether it respects a number of desired properties. Moreover, a specification can be animated in order to give the user immediate feedback on its implications.

2 Preliminaries

In order to use the T-Tool you need to create a perl script or start an interactive session of perl. In the following we consider the case of the use of a script. We assume that the `TTool.pm` perl library has been installed in a location known to perl (e.g. /usr/lib/perl). Moreover, we assume that `TNuSMV`, and the tools `ft2smv`, `ft2il`, `il2smv` and `ltl2smv` have been installed in the search path of the shell (e.g. `/usr/local/bin`).

3 T-Tool at work

We now illustrate the usage of Formal Tropos in refining an early requirements specification, and the use of the T-Tool to support and guide the refinement. For explanatory purposes we focus on

the model, shown in Figure 1, and we assume this specification being saved in a file named `cm.ft`.

This specification is strongly under-specified. In particular, it does not cover at all the dynamic aspects of the domain. We will interactively improve and refine it, guided by results provided by the analysis.

```
Entity Course

Entity Exam
  Attribute constant course :  Course

Actor Student

Goal PassCourse
  Mode achieve
  Actor Student
  Attribute constant course :  Course

Actor Teacher

Task GiveExam
  Mode achieve
  Actor Teacher
  Attribute constant exam :  Exam

Dependency Answer
  Type Resource
  Mode achieve
  Depender Teacher
  Dependee Student
  Attribute constant exam :  Exam

Dependency Mark
  Type Resource
  Mode achieve
  Depender Student
  Dependee Teacher
  Attribute constant exam :  Exam
            passed :  boolean
```

Figure 1: The initial FT specification for the analysis of the case study.

3.1 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.

Once the initial model is available we can start performing the analysis. The initial perl script is illustrated below. (Let us assume the file is stored in file `cm0.pl`.)

```
#! /usr/bin/perl
```

```
use TTool;

my @classes = ();
my %bounds1;
my %bounds2;
my $tstfile = "cm.ft";

# extracts the classes of a FT specification
@classes = ExtractClasses($tstfile);

# creates bounds for the analysis
foreach $cl (@classes) {
   $bounds1{$cl} = 1;
   $bounds2{$cl} = 2;
}

# exists the possibility to create all the instances
DoExistsTest("CONSISTENCY CHECK 1 instance","$tstfile", %bounds1);

# exists the possibility to create all the instances in t
DoExistsTest("CONSISTENCY CHECK 2 instances","$tstfile", %bounds2);
```

The routine `ExtractClasses` analyzes the FT specification in a file and extracts the set of FT "classes". Once this set has been extracted, we proceed to create the upper "bounds" on the number of class instances, saving this information in the perl variables `bounds1` and `bounds2`.

The call to `DoExistsTest` performs a consistency check that aims to verify whether if it is the case that all the instances of the classes will be eventually generated. This amounts to check whether the bounds specified are compatible with the possible cardinality constraints of the specification.

If we run the perl script we obtain the following results (notice that the output has been slightly modified to make it more readable).

```
schubert > ./cm0.pl
This is T-Tool library 1.0
******************** Test CONSISTENCY_CHECK_1_insance ********************
#####################################################
DATA CONSISTENCY_CHECK_1_instance: Time = 1.33 sec
DATA CONSISTENCY_CHECK_1_instance: Property status = CTX/Witn
DATA CONSISTENCY_CHECK_1_instance: Counterexample length = 2
DATA CONSISTENCY_CHECK_1_instance: No Bug at length = -1
DATA CONSISTENCY_CHECK_1_instance: Time (NuSMV) = 1.13 sec
DATA CONSISTENCY_CHECK_1_instance: SBRK mem = 1292 Kb
DATA CONSISTENCY_CHECK_1_instance: Initialized mem = 133 Kb
DATA CONSISTENCY_CHECK_1_instance: Uniinitialized mem = 1422 Kb
DATA CONSISTENCY_CHECK_1_instance: Size SMV file = 25124 b
#####################################################
### The scenario extracted is the following
#####################################################
-- loop starts here --
-> State 1.1 <-
```

```
    Teacher_1.exists = 1
    Exam_1.exists = 1
    Answer_1.fulfilled = 1
    Answer_1.exists = 1
    PassCourse_1.fulfilled = 1
    PassCourse_1.exists = 1
    Mark_1.fulfilled = 1
    Mark_1.exists = 1
    Mark_1.passed = 1
    Course_1.exists = 1
    Student_1.exists = 1
    GiveExam_1.fulfilled = 1
    GiveExam_1.exists = 1
-> State 1.2 <-
    Teacher_1.exists = 1
    Exam_1.exists = 1
    Answer_1.fulfilled = 1
    Answer_1.exists = 1
    PassCourse_1.fulfilled = 1
    PassCourse_1.exists = 1
    Mark_1.fulfilled = 1
    Mark_1.exists = 1
    Mark_1.passed = 1
    Course_1.exists = 1
    Student_1.exists = 1
    GiveExam_1.fulfilled = 1
    GiveExam_1.exists = 1
################################################
```

The results shows two witness scenarios where all the classes will be eventually created. For the sake of readability we report only the one instance case output.

As you can see, additional informations are printed out, that's is some statistics on the time and memory required by the tool to provide an answer to the query.

3.2 Assertion checks

Assertions check whether the requirements are under-specified and allow for scenarios violating desired properties. In this phase of model analysis 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.

As an example we consider the achievement of goal PassCourse. In our case study, a student passes a course if she takes all the exams for the course and if there is a passing mark for each exam. To capture this requirement, we modify the FT model by adding task PassExam as a means for the achievement of goal PassCourse. To fulfill goal PassCourse we require that for each exam of the course there exists at least one instance of PassExam that is fulfilled. We also require that an instance of PassExam can be created only if the corresponding PassCourse has not been fulfilled yet: if the goal PassCourse has already been fulfilled, there is no need to pass any further exam for that course. We allow for several instances of the class Mark for each PassExam. A sufficient condition for passing the exam is that at least one corresponding mark is passing.

```
Goal PassCourse
  Fulfillment definition
    ∀ e : Exam (e.course = course →
      ∃ p : PassExam (p.exam = e ∧
        p.pc = self ∧ Fulfilled (p)))
Task PassExam
  Mode achieve
  Actor Student
  Attribute constant pc : PassCourse
            constant exam : Exam
  Creation condition
    ¬ Fulfilled (pc)
  Invariant
    pc.actor = actor ∧ pc.course = exam.course
  Fulfillment condition
    ∃ m : Mark (m.depender = actor ∧ m.exam = exam ∧
        Fulfilled (m) ∧ m.passed)
```

We want to guarantee that a passing mark to be present if a PassExam goal is fulfilled.

```
#ifdef ASSE1
Global assertion
  ∀ pe : PassExam (Fulfilled (pe) →
    ∃ m : Mark (m.exam = pe.exam ∧ Fulfilled (m) ∧ m.passed))
#endif
```

We add in the perl script above the following call, where we also select to use bounded model checking to answer to the queries.

```
# switch to BMC for further verifications
SetNuSMVUseBMC();

DoGenericTest("ASSERTION 1 on 1 instance", "$tstfile", "ASSE1", %bounds1);
```

The execution of the script generates a counterexample as shown below.

```
******************** Test ASSERTION_1_on_1_instance ********************
##################################################
DATA ASSERTION_1_on_1_instance: Time = 0.57 sec
DATA ASSERTION_1_on_1_instance: Property status = CTX/Witn
DATA ASSERTION_1_on_1_instance: Counterexample length = 4
DATA ASSERTION_1_on_1_instance: No Bug at length = 2
DATA ASSERTION_1_on_1_instance: Time (NuSMV) = 0.38 sec
DATA ASSERTION_1_on_1_instance: SBRK mem = 3480 Kb
DATA ASSERTION_1_on_1_instance: Initialized mem = 133 Kb
DATA ASSERTION_1_on_1_instance: Uniinitialized mem = 1422 Kb
DATA ASSERTION_1_on_1_instance: Size SMV file = 38037 b
##################################################
### The scenario extracted is the following
##################################################
-> State 1.1 <-
    Exam_1.exists = 1
```

```
    PassCourse_1.fulfilled = 0
    PassCourse_1.exists = 1
    Course_1.exists = 1
    PassExam_1.fulfilled = 0
    PassExam_1.exists = 1
    Student_1.exists = 1
-> State 1.2 <-
    Teacher_1.exists = 1
    Exam_1.exists = 1
    PassCourse_1.fulfilled = 1
    PassCourse_1.exists = 1
    Mark_1.fulfilled = 1
    Mark_1.exists = 1
    Mark_1.passed = 1
    Course_1.exists = 1
    PassExam_1.fulfilled = 1
    PassExam_1.exists = 1
    Student_1.exists = 1
-- loop starts here --
-> State 1.3 <-
    Teacher_1.exists = 1
    Exam_1.exists = 1
    PassCourse_1.fulfilled = 1
    PassCourse_1.exists = 1
    Mark_1.fulfilled = 1
    Mark_1.exists = 1
    Mark_1.passed = 0
    Course_1.exists = 1
    PassExam_1.fulfilled = 1
    PassExam_1.exists = 1
    Student_1.exists = 1
-> State 1.4 <-
    Teacher_1.exists = 1
    Exam_1.exists = 1
    PassCourse_1.fulfilled = 1
    PassCourse_1.exists = 1
    Mark_1.fulfilled = 1
    Mark_1.exists = 1
    Mark_1.passed = 0
    Course_1.exists = 1
    PassExam_1.fulfilled = 1
    PassExam_1.exists = 1
    Student_1.exists = 1
####################################################
```

The counterexample shows that the expected property does not hold at time t_3 and at time t_4 since the instance of Mark exists and is fulfilled, but the corresponding passed attribute is false.

To enforce the requirement that a mark – once produced – does not change its value, we add the following invariant constraint to the dependency Mark.

```
Resource Dependency Mark
  Invariant
    Fulfilled (self) → (passed ↔ X passed)
```

Notice that, the value of attribute passed is only relevant once the dependency has been fulfilled, therefore we do not care if it changes before its fulfillment.

3.3 Possibility checks

As the specification grows, it is important to detect over-constrained situations that rule out desired behaviors. This kind of analysis is called possibility checking. 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.

For instance, we want to make it sure that the specification allows a student to pass a course. This requirement is formulated with the following possibility.

#ifdef POSS1
Global possibility
∃ p : PassCourse (**Fulfilled** (p))
#endif

An existence proof for this possibility can be generated by adding to the perl script the following call.

```
DoGenericTest("POSSIBILITY 1 on 1 instance", "$tstfile", "POSS1", %bounds1);
```

The results of the execution of the script is thus the following.

```
******************** Test POSSIBILITY_1_on_1_instance ********************
DATA POSSIBILITY_1_on_1_instance: Time = 0.41 sec
DATA POSSIBILITY_1_on_1_instance: Property status = CTX/Witn
DATA POSSIBILITY_1_on_1_instance: Counterexample length = 3
DATA POSSIBILITY_1_on_1_instance: No Bug at length = 1
DATA POSSIBILITY_1_on_1_instance: Time (NuSMV) = 0.24 sec
DATA POSSIBILITY_1_on_1_instance: SBRK mem = 2640 Kb
DATA POSSIBILITY_1_on_1_instance: Initialized mem = 133 Kb
DATA POSSIBILITY_1_on_1_instance: Uniinitialized mem = 1422 Kb
DATA POSSIBILITY_1_on_1_instance: Size SMV file = 37843 b
##################################################
### The scenario extracted is the following
##################################################
-> State 1.1 <-
    PassCourse_1.fulfilled = 1
    PassCourse_1.exists = 1
    Course_1.exists = 1
    Student_1.exists = 1
-- loop starts here --
-> State 1.2 <-
    PassCourse_1.fulfilled = 1
    PassCourse_1.exists = 1
    Course_1.exists = 1
```

```
    Student_1.exists = 1
-> State 1.3 <-
    PassCourse_1.fulfilled = 1
    PassCourse_1.exists = 1
    Course_1.exists = 1
    Student_1.exists = 1
##################################################
```

4 Further Information

Further informations among Formal Tropos and the T-Tool can be retrieved at the following places:

- The Formal Tropos web site (`http://www.dit.unitn.it/~ft`), where several papers and example case-studies can be downloaded.
- The Tropos project web site (`http://www.troposproject.org`).

More informations or questions can be addressed to `ft@dit.unitn.it`.