University of Toronto
Department of Computer Science

Formal Analysis of Early Requirements Specifications in Tropos

Ariel Fuxman

Joint work with Marco Pistore and Paolo Traverso (IRST, Trento, Italy)

April 5th, 2001

Motivations

The *Tropos Project* proposes to

- start the development process from the early requirements phase.
- apply the concepts of early requirements (actors, goals, social interactions) to the later phases of the development process.

Another goal of the project is to integrate *Formal Methods* techniques inside the Tropos framework.

In this talk: Analysis in the Tropos framework

- using Formal Methods
- starting from the (early) requirements phase.

Early Requirements Engineering

This phase is specially concerned with understanding the organizational context of a system.

Modelling languages:

- KAOS
  - Goal-oriented Requirements Framework
  - Formal, but no support for automated analysis
- i*
  - Rich ontology for early requirements (actors, strategic dependencies)
  - Starting point for the Tropos project.
  - Analysis is performed mostly at an informal level

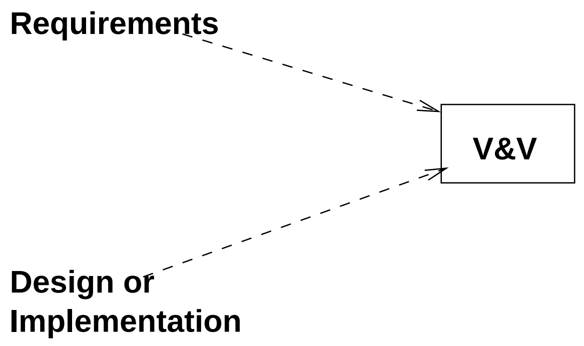
Formal Methods

Formal specification languages have

- very formal semantics.
- strong support for formal analysis.

Designed to work in the later phases of software development.

- The ontology of formal specification languages is not appropriate for early requirements.
- The typical approach for the application of Formal Methods is not suitable for early requirements.



Formal Analysis during Early Requirements

Our approach for formal analysis in the early requirements phase aims to *assist the analyst in the requirements elicitation* (e.g, by showing errors and limitations of the specification that are not evident in the informal setting).

Our work includes

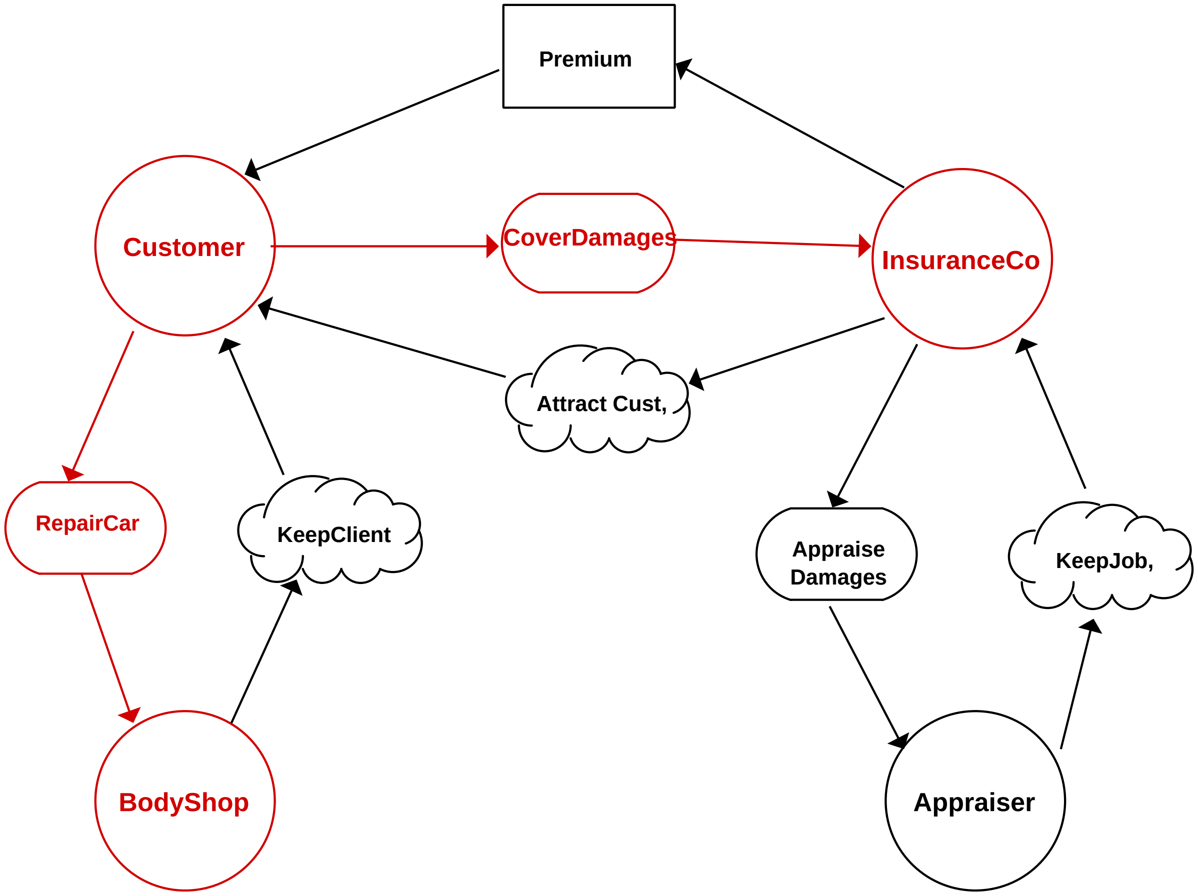
- *Formal Tropos* language, as an extension of i*.
- Prototype tool, based on model checking, a Formal Methods technique, for the analysis of Formal Tropos specifications.
- Application of the framework to a simple case study

Overview

- Introduction
- Case Study
- Formal Tropos Language
- Formal Analysis in Tropos
- Technical Details
- Conclusions
- Future Work

Case Study

Insurance Company Case Study



Formal Tropos Language

A textual notation for i*...

Actor InsuranceCo

Actor BodyShop

Actor Customer
  Goal BeInsured

Dependency CoverDamages
  Type goal
  Depender Customer
  Dependee InsuranceCo

Dependency RepairCar
  Type goal
  Depender Customer
  Dependee BodyShop

with an extended “class” layer.

Entity Car
  Attribute runsOK: **boolean**

Entity Claim
  Attribute constant car: Car

Actor InsuranceCo

Actor BodyShop

Actor Customer
  Goal BeInsured

Dependency CoverDamages
  (Type/Depender/Dependee)
  Attribute constant cl: Claim

Dependency RepairCar
  (Type/Depender/Dependee)
  Attribute constant cl: Claim

Modeling dynamic aspects: goal modalities...

Actor goals and dependencies have a **mode** that expresses its fulfillment modality: possible modes are **achieve**, **maintain**, **achieve&maintain**, **avoid** ...

Actor Customer
  Goal BeInsured
    Mode maintain

Dependency CoverDamages
  Type goal
  Mode achieve
  Depender Customer
  Dependee InsuranceCo

Dependency RepairCar
  Type goal
  Mode achieve
  Depender Customer
  Dependee BodyShop

… and behavioral constraints

Dependency CoverDamages
  Type goal
  Mode achieve
  Depender Customer
  Dependee InsuranceCo
  Attribute constant cl: Claim
  Creation condition ¬cl.car.runsOK
Dependency RepairCar
  Type goal
  Mode achieve
  Depender Customer
  Dependee BodyShop
  Creation condition ¬cl.car.runsOK
  Fulfillment condition for depender cl.car.runsOK

Constraints

Constraints are formulas that determine the possible evolutions of the objects in the specification.

Three kinds of constraints:

- **creation** constraints
- **invariant**s
- **fulfillment** constraints

Formulas may express:

- necessary **condition**s (for creation, fulfillment . . .)
- sufficient conditions, or **trigger**s
- necessary and sufficient conditions, or **definition**s

Temporal formulas

Behavioral constraints are given in a *first-order linear-time temporal* logic.

- Quantifiers $\forall$, $\exists$ range over all the existing instances of a given class:

  $$\forall \mathrm{cov} : \mathrm{CoverDamages}(\ldots)$$

- Special predicate "JustCreated(obj)", "Fulfilled(dep)" identify particular moments in the life of the objects

- Past and future temporal operators can be used in the formulas:
  - $\Box\phi$ (always in the future), $\Diamond\phi$ (eventually), $\circ\phi$ (in the next state)
  - $\blacksquare\phi$ (always in the past), $\blacklozenge\phi$ (sometimes in the past), $\bullet\phi$ (in the previous state)

We aim to minimize the use of temporal operators. For instance,

- **maintain** hides a $\Box$.

- **achieve** hides an $\Diamond$.

Formal Analysis

Formal analysis

Given

- a Formal Tropos specification
- a bound on the number of instances of each class

our tool performs the following kinds of formal analysis:

- *consistency check*: "there is *at least one* scenario for the system"
- *possibility check*: "there is *at least one* scenario for the system that respects certain **possibility** formulas"
- *assertion validation*: "*all* scenarios for the system respect certain **assertion** formulas"
- *animation*: allows the user to interactively explore valid scenarios for the system
  - gives immediate feedback on the effects of the constraints
  - allows to catch trivial errors
  - is an effective way of communicating with the stakeholder

Assertion validation

Constraints express *required* conditions; they *define* the valid scenarios.

On the other hand, **assertion**s to express *desired* conditions; that is *expected* behaviors of the valid scenarios.

Assertions can be specified in Formal Tropos with the same syntax as constraints, but they have a different semantics.

Example: "the insurance company should not cover claims for which there is no proof (e.g., an invoice) that the car was repaired"

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

A first counterexample...

The tool returns the following *counterexample* scenario

	t_0	t_1	t_2	t_3
cov2		$\top$	$\top$	$\top$
Fulfilled(cov2)		$\bot$	$\top$	$\top$
cl1		$\top$	$\top$	$\top$
car1		$\top$	$\top$	$\top$
car1.RunsOK		$\bot$	$\top$	$\bot$

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

Possible fix:

Entity Car

Attribute RunsOK: boolean

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

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

Another counterexample...

	t_0	t_1	t_2	t_3
cov1		$\top$	$\top$	
cov2		$\top$	$\top$	
Fulfilled(cov2)		$\bot$	$\top$	
rep1		$\top$	$\top$	
Fulfilled(rep1)		$\bot$	$\top$	
cl1		$\top$	$\top$	
cl2		$\top$	$\top$	
car1		$\top$	$\top$	
car1.RunsOK		$\bot$	$\top$	

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

Possible fix: add to **Dependency** CoverDamages the following constraint:

Fulfillment condition $\exists \text{rep} : \text{RepairCar}(\text{rep.cl} = \text{cl} \wedge \text{Fulfilled}(\text{rep}))$

Possibility check

A specification might be so constrained that it rules out reasonable scenarios as valid executions of the system.

Example: "when cars are so damaged that they cannot be repaired, the insurance company is still responsible for covering damages"

Dependency CoverDamages

Creation possibility condition for dependee

$\Diamond \mathrm{Fulfilled(cov)} \wedge \Box \neg \mathrm{cov.cl.car.runsOK}$

The possibility check fails. There is *no* suitable *example* scenario.

Possible fix: modify the constraint for **Dependency** CoverDamages

Fulfillment condition

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

Example Scenario

	t_0	t_1	t_2	t_3
cov1		$\top$	$\top$	$\top$
Fulfilled(cov1)		$\bot$	$\top$	$\top$
cl1		$\top$	$\top$	$\top$
car1		$\top$	$\top$	$\top$
car1.RunsOK		$\bot$	$\bot$	$\bot$

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

Continuing with the example....

New scenario from the stakeholders: "the insurance company might cover damages for the car even if it is not running, but as long as it is confident that the car will be *eventually* repaired."

After another iteration, we come up with the following **possibility condition** for `CoverDamages`.

Fulfilment possibility condition for depender

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

The **possibility** does not exist. Possible fix: modify the constraint of `CoverDamages` once again.

Fulfilment condition for dependee

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

Continuing with the example....

The `possibility` now holds and the tool gives an example scenario.

Side effect: the tool gives a counterexample to the original assertion (that one that states that there must be a proof of the repairs)

We modify the constraint for `CoverDamages` yet once again...

Fulfilment condition for dependee

$$\exists rep : RepairCar(rep.cl = cl \wedge (\neg cl.car.RunsOK\, \mathcal{U}\, Fulfilled(rep)))$$
$$\vee \Box \neg cl.car.RunsOK$$

... and now all **assertions** and **possibilities** hold.

Technical Details

A 3-step Approach

For performing formal analysis, the following steps are followed

- The analyst writes a *Formal Tropos* specification.
- The tool *automatically* translates the specification into an *Intermediate Language*.
- Formal analysis is performed on top of NuSMV, an state-of-the-art model checker. An appropriate translation from the Intermediate Language to the input language of NuSMV is defined.

The Intermediate Language plays an important role:

- small core language with a clean semantics
- independent from the particularities of Formal Tropos (the Intermediate Language may be applied to other requirements languages)
- independent from any particular analysis technique (model checking, LTL satisfiability, theorem proving...)

The Intermediate Language: **CLASS** declarations

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

- the classes correspond to entities, actors and dependencies of the Formal Tropos specification
- some attributes are added during the translation (eg the "fulfilled" attributes)
- facets (e.g, **constant**) are removed

The Intermediate Language: Logic Specification

LTL temporal formulas that restrict the define the behavior of the system. The resulting formulas after an automatic translation from Formal Tropos:

- model the semantics of Formal Tropos: eg, **constant** facets:

  $$\mathbf{CONSTRAINT}\ \forall \mathrm{cl} : \mathrm{Claim} \quad \forall \mathrm{car} : \mathrm{Car}\ (\mathrm{cl.car} = \mathrm{car} \rightarrow \circ(\mathrm{cl.car} = \mathrm{car}))$$
  $$\mathbf{CONSTRAINT}\ \forall \mathrm{rc} : \mathrm{RepairCar} \quad \forall \mathrm{cl} : \mathrm{Claim}\ (\mathrm{rc.cl} = \mathrm{cl} \rightarrow \circ(\mathrm{rc.cl} = \mathrm{cl}))$$

- give semantics to behavioral constraints:

  creation condition:

  $$\mathbf{CONSTRAINT}\ \forall \mathrm{rc} : \mathrm{RepairCar}\ (\mathrm{JustCreated(rc)} \rightarrow \neg \mathrm{rc.cl.car.runsOK})$$

  "achieve" fulfillment condition:

  $$\mathbf{CONSTRAINT}\ \forall \mathrm{rc} : \mathrm{RepairCar}(\mathrm{JustFulfilled(rc)} \rightarrow \mathrm{rc.cl.car.runsOK})$$

The Intermediate Language (cont.)

- Example of rules for the translation

  an **achieve fulfillment condition** ϕ of a dependency Dep is converted as

  CONSTRAINT $\forall \mathrm{d} : \mathrm{Dep}(\mathrm{JustFulfilled}(\mathrm{d}) \rightarrow \phi)$

- **assertion** and **possibility** formulas are managed similarly to constraints:

  ASSERTION $\forall \mathrm{cov} : \mathrm{CoverDamages}(\mathrm{cov.fulfilled} \rightarrow$
  $\blacklozenge(\mathrm{cov.cl.car.runsOK} \rightarrow \exists \mathrm{rep} : \mathrm{RepairCar}(\mathrm{rep.cl} = \mathrm{cov.cl} \wedge \mathrm{rep.fulfilled})))$
  POSSIBILITY $\exists \mathrm{cov} : \mathrm{CoverDamages}$
  $(\mathrm{JustCreated}(\mathrm{cov}) \wedge \lozenge \mathrm{cov.fulfilled} \wedge \Box \neg \mathrm{cov.cl.car.runsOK})$

Automated Verification: Model Checking

Problem: Check that a formula is satisfied on a *finite* representation of a system

Advantages:

- Potential for fully automatic, "push-button" verification
- Explicit negative information as counterexamples

Disadvantages:

- systems must be finite
- state explosion problem

NuSMV

State-of-the art *symbolic* model checker.

Developed at IRST and CMU.

Initially intended to verify synchronous systems (e.g, hardware circuits).

To reuse the model checking algorithms provided by NuSMV we:

- adapted an existing algorithm for translating the Intermediate Language LTL constraint formulas to the NuSMV input language.
- extended the algorithm to handle past operators.
- dealt with quantifier elimination and the “pointer” problem

Conclusions

Conclusions

It is feasible to apply model checking to early requirements.

We obtained feedback from a formal analysis tool even when dealing with just a few instances.

Our methodology relies on two kinds of scenarios

- from the stakeholders (**assertion** and **possibilities**).
- automatically returned by the tool (examples and counterexamples).

Future work

- Test the approach on "real" case studies.
- Give an "agent-oriented" semantics to the language.
- Deal with *goal decomposition* and *operationalization*, give semantics to task dependencies.
- Extend the approach to later development phases (late requirements, architectural design, implementation . . .).
- Enhance the tool:
  - better interaction with the user
  - develop specifically tailored verification algorithms (e.g. embedded quantifier elimination)
  - improve the animation techniques

Relevant links

- The Tropos development methodology:
  `http://www.cs.toronto.edu/~mkolp/tropos/`

- The "Agent-Oriented Software Engineering" project at IRST:
  `http://sra.itc.it/research_topic.epl?name=AOSE`

- The "Agent-Oriented Software Engineering" project at UniTN:
  `http://www.science.unitn.it/~pgiorgio/aose/`

- The paper on "Model Checking Early Requirements in Tropos:
  `http://sra.itc.it/people/pistore/papers/FT.ps.gz`

- The NuSMV model verifier:
  `http://sra.itc.it/tools/nusmv/`