MOVEP 2012 Tutorial
Safety, Dependability and Performance Analysis of Extended AADL Models

Part 3: Checking Functional Correctness

European Space Agency
European Space Research and Technology Centre

RWTH Aachen University
Software Modeling and Verification Group
Thomas Noll

Fondazione Bruno Kessler
Centre for Scientific and Technological Research
Alessandro Cimatti

MOVEP 2012 School; December 7, 2012; Marseille, France

Contents of Overview

Outline

1. Introduction
2. Model checking
3. Ongoing Activities
4. Demo Example
5. Tool Support

Verification and Validation

Objectives

- Validate the quality of system requirements
- Simulate the system to ensure behavior is as expected
- Check the absence of unwanted behaviors (e.g., deadlocks)
- Check system behavior against a set of properties

Verification and Validation

Objectives

- Validate the quality of system requirements
- Simulate the system to ensure behavior is as expected
- Check the absence of unwanted behaviors (e.g., deadlocks)
- Check system behavior against a set of properties

Analyses

- Requirements Validation
- Simulation
- Deadlock Checking
- Property Verification

Verification and Validation

Objectives

- Validate the quality of system requirements
- Simulate the system to ensure behavior is as expected
- Check the absence of unwanted behaviors (e.g., deadlocks)
- Check system behavior against a set of properties

Analyses

- Requirements Validation
- Simulation
- Deadlock Checking
- Property Verification

COMPASS Technologies

- Model Checking

Requirements Validation

Motivations

- Ensure that requirements capture the design intent
- Bugs in requirements are very expensive to correct, when discovered late in the development process
- Flaws in the requirements engineering phase are responsible for a significant percentage of product defects and re-engineering efforts

Requirements Validation

Motivations

- Ensure that requirements capture the design intent
- Bugs in requirements are very expensive to correct, when discovered late in the development process
- Flaws in the requirements engineering phase are responsible for a significant percentage of product defects and re-engineering efforts

Goals

- Validate the quality of requirements before the system is implemented
- Ensure that we are "building the right system"
- Detect ambiguities, inconsistencies, and deficiencies in requirements

Outline

Formal Verification

Goals

- Discover as many bugs as possible, as early as possible
- Certify absence of errors
- Shorten time to market, improve quality standards

Formal Verification

Goals

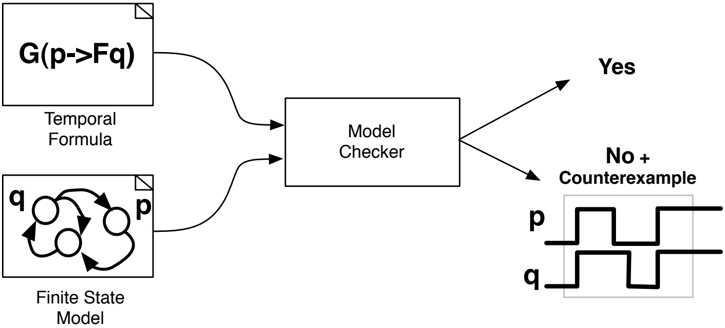
- Discover as many bugs as possible, as early as possible
- Certify absence of errors
- Shorten time to market, improve quality standards

Model Checking

- System formally modeled as a mathematical theory
- Properties expressed using temporal logic
- System correctness as mathematical proving of a theorem
- Model checker: a software tool that can
  - Prove a theorem
  - Find a counterexample that shows that the theorem is wrong
- Fully automated, exhaustive, useful for the designer

Model Checker

A pictorial view of a model checker



Model Checking

Modeling

- State transitions systems are a traditional formalism to model reactive systems and their evolution
- Basis for model checking

Model Checking

Modeling

- State transitions systems are a traditional formalism to model reactive systems and their evolution
- Basis for model checking

State Transition System

Let $\mathcal{P}$ be a set of propositions.
A state transition system (also known as Kripke structure) is a tuple $\langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ where:

- $\mathcal{S}$ is a finite set of states
- $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation
- $\mathcal{L} : \mathcal{S} \longrightarrow 2^{\mathcal{P}}$ is the labeling function

Model Checking

Modeling

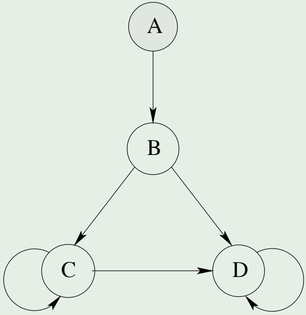
- State transitions systems are a traditional formalism to model reactive systems and their evolution
- Basis for model checking

State Transition System

Let $\mathcal{P}$ be a set of propositions.
A state transition system (also known as Kripke structure) is a tuple $\langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ where:

- $\mathcal{S}$ is a finite set of states
- $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation
- $\mathcal{L} : \mathcal{S} \longrightarrow 2^{\mathcal{P}}$ is the labeling function

An example



Model Checking

Trace

Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ be a Kripke structure. A trace for $\mathcal{M}$ is a sequence $s_0, s_1, \ldots, s_k$ such that $s_i \in \mathcal{S}$, $s_0 \in \mathcal{I}$ and $(s_{i-1}, s_i) \in \mathcal{R}$ for $i = 1 \ldots k$

Model Checking

Trace

Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ be a Kripke structure. A trace for $\mathcal{M}$ is a sequence $s_0, s_1, \ldots, s_k$ such that $s_i \in \mathcal{S}$, $s_0 \in \mathcal{I}$ and $(s_{i-1}, s_i) \in \mathcal{R}$ for $i = 1 \ldots k$

Path

A path is a Kripke structure is an infinite trace, that is, a sequence $\sigma = s_0, s_1, s_2, \ldots$

Model Checking

Trace

Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ be a Kripke structure. A trace for $\mathcal{M}$ is a sequence $s_0, s_1, \ldots, s_k$ such that $s_i \in \mathcal{S}$, $s_0 \in \mathcal{I}$ and $(s_{i-1}, s_i) \in \mathcal{R}$ for $i = 1 \ldots k$

Path

A path is a Kripke structure is an infinite trace, that is, a sequence $\sigma = s_0, s_1, s_2, \ldots$

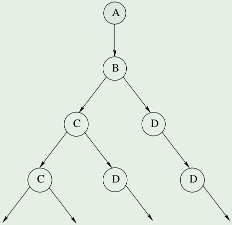
Unwinding of a Kripke Structure

A Kripke structure unwinds into an infinite tree representing all possible paths

Labeling Function

We write $s \models p$ to indicate that $p \in \mathcal{L}(s)$ (proposition p holds in state s)

An Example



Temporal Logic

Temporal Logic

Temporal logic can be used to express properties of reactive systems modeled as Kripke structures

Temporal Logic

Temporal Logic

Temporal logic can be used to express properties of reactive systems modeled as Kripke structures

Safety vs Liveness

System properties can be classified into:

- Safety properties: "nothing bad ever happens"
- Liveness properties: "something desirable will eventually happen"

Temporal Logic

Temporal Logic

Temporal logic can be used to express properties of reactive systems modeled as Kripke structures

Safety vs Liveness

System properties can be classified into:

- Safety properties: "nothing bad ever happens"
- Liveness properties: "something desirable will eventually happen"

Some example properties

- Safety: "Two concurrent processes never execute simultaneously within their critical section"
- Liveness: "A subroutine will eventually terminate execution and return control to the caller"

Temporal Logic

Refuting temporal properties

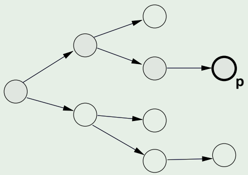
- A safety property can be refuted by a finite counterexample trace such that p holds in the end state of the trace (where p models the *bad state*)
- A liveness property can be refuted by a infinite counterexample trace (with a loop), such that $\neg p$ holds along all states of the trace (where p models the *desirable state*)

Temporal Logic

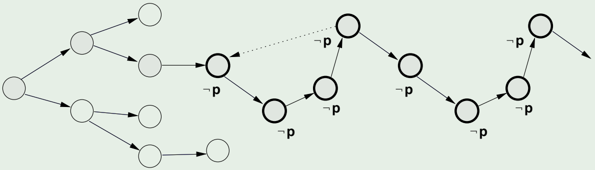
Refuting temporal properties

- A safety property can be refuted by a finite counterexample trace such that p holds in the end state of the trace (where p models the *bad state*)
- A liveness property can be refuted by a infinite counterexample trace (with a loop), such that $\neg p$ holds along all states of the trace (where p models the *desirable state*)

Refuting safety



Refuting liveness (infinite trace)



Temporal Logic

Examples of Temporal Logic

- Computation Tree Logic (CTL)
- Linear Temporal Logic (LTL)

Temporal Logic

Examples of Temporal Logic

- Computation Tree Logic (CTL)
- Linear Temporal Logic (LTL)

Temporal Logic Interpretation

- CTL is interpreted over the unwound tree
- LTL is interpreted over linear paths

Temporal Logic

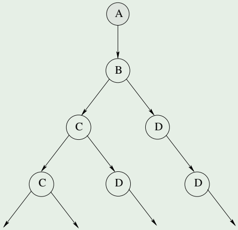
Examples of Temporal Logic

- Computation Tree Logic (CTL)
- Linear Temporal Logic (LTL)

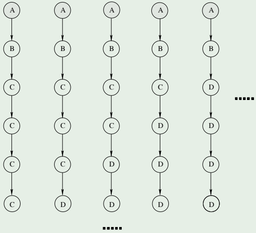
Temporal Logic Interpretation

- CTL is interpreted over the unwound tree
- LTL is interpreted over linear paths

Tree

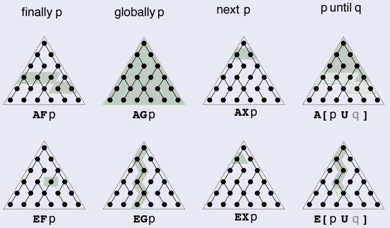


Linear Paths

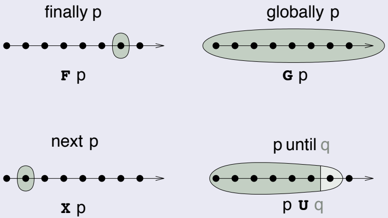


Temporal Logic

Semantics of CTL



Semantics of LTL



Temporal Logic and Property Patterns

Property specification in COMPASS

- Recall from Part I: Patterns, not Formulas!

Temporal Logic and Property Patterns

Property specification in COMPASS

- Recall from Part I: Patterns, not Formulas!

Patterns

- The system shall have a behavior where ϕ globally holds.

Temporal Logic and Property Patterns

Property specification in COMPASS

- Recall from Part I: Patterns, not Formulas!

Patterns

- The system shall have a behavior where ϕ globally holds.

Patterns

- The system shall have a behavior where $80 \leq \mathrm{voltage} \leq 90$ globally holds.

Temporal Logic and Property Patterns

Property specification in COMPASS

- Recall from Part I: Patterns, not Formulas!

Patterns

- The system shall have a behavior where ϕ globally holds.

Patterns

- The system shall have a behavior where $80 \leq \mathrm{voltage} \leq 90$ globally holds.

|

(by automatic transformation)

↓

Logic

- $AG(80 \leq \mathrm{voltage} \leq 90)$ (CTL)
- $G(80 \leq \mathrm{voltage} \leq 90)$ (LTL)

Model Checking

Model Checking Problem

Given a state transition system $\mathcal{M}$ and a temporal formula ϕ, model checking is the problem of deciding whether ϕ holds in $\mathcal{M}$, written $\mathcal{M} \models \phi$

Model Checking

Model Checking Problem

Given a state transition system $\mathcal{M}$ and a temporal formula ϕ, model checking is the problem of deciding whether ϕ holds in $\mathcal{M}$, written $\mathcal{M} \models \phi$

Explicit-State Model Checking

- Based on the expansion and storage of individual states
- Explicit representation of the Kripke structure (e.g., as a labeled, directed graph)
- May suffer from the state explosion problem

Model Checking

Model Checking Problem

Given a state transition system $\mathcal{M}$ and a temporal formula ϕ, model checking is the problem of deciding whether ϕ holds in $\mathcal{M}$, written $\mathcal{M} \models \phi$

Explicit-State Model Checking

- Based on the expansion and storage of individual states
- Explicit representation of the Kripke structure (e.g., as a labeled, directed graph)
- May suffer from the state explosion problem

Symbolic Model Checking

- Manipulates sets of states and transitions as logical formulas
- Logical formulas may admit a large number of models
- Leads to compact representations that can be effectively manipulated

Symbolic Model Checking

Symbolic representation:

- Construct bijection between $\mathcal{S}$ and $2^{\mathcal{P}}$
- States: represented using a vector of Boolean variables $\underline{x}$
- Initial states: $\mathcal{I}(\underline{x})$
- Transition relation: $\mathcal{R}(\underline{x}, \underline{x}')$, where $\underline{x}'$ represent next state variables

Symbolic Model Checking

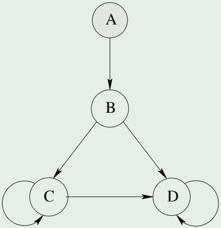
Symbolic Model Checking

Symbolic representation:

- Construct bijection between $\mathcal{S}$ and $2^{\mathcal{P}}$
- States: represented using a vector of Boolean variables $\underline{x}$
- Initial states: $\mathcal{I}(\underline{x})$
- Transition relation: $\mathcal{R}(\underline{x}, \underline{x}')$, where $\underline{x}'$ represent next state variables

An Example

- $\underline{x} = x_1 x_2$
- $A : \neg x_1 \wedge \neg x_2$, $B : \neg x_1 \wedge x_2$, $C : x_1 \wedge \neg x_2$, $D : x_1 \wedge x_2$
- $\mathcal{I}(\underline{x}) = \neg x_1 \wedge \neg x_2$
- $$\begin{aligned} \mathcal{R}(\underline{x}, \underline{x}') = \; & (\neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge x_2') \vee \\ & (\neg x_1 \wedge x_2 \wedge x_1' \wedge \neg x_2') \vee \\ & (\neg x_1 \wedge x_2 \wedge x_1' \wedge x_2') \vee \\ & \ldots \end{aligned}$$



Symbolic Model Checking

BDD-based Model Checking

- Symbolic representation and manipulation of formulas based on BDDs (Binary Decision Diagrams)
- Canonical representation, given a variable ordering
- Operations on sets of states as logical operations on BDDs
- Efficient BDD packages exist for BDD manipulation
- Breakthrough for model checking

Symbolic Model Checking

BDD-based Model Checking

- Symbolic representation and manipulation of formulas based on BDDs (Binary Decision Diagrams)
- Canonical representation, given a variable ordering
- Operations on sets of states as logical operations on BDDs
- Efficient BDD packages exist for BDD manipulation
- Breakthrough for model checking

SAT-based Model Checking

- Also known as Bounded Model Checking (BMC)
- Bounded search for a violation, up to bound k
- Problem is encoded into a propositional formula, by unwinding the symbolic description of the transition relation over time:
  $\mathcal{I}(\underline{x_0}) \wedge \mathcal{R}(\underline{x_0}, \underline{x_1}) \wedge \ldots \wedge \mathcal{R}(\underline{x_{k-1}}, \underline{x_k})$
- Solution leverages the power of modern SAT solvers

BDD-based versus SAT-based Model Checking

Complementary techniques:

- SAT-based may deal with a larger number of variables
- SAT-based useful for bug finding
- BDD-based may be more effective for long counterexamples
- BDD-based may be more effective in proving correctness

References

- Requirements Validation (Pill et. al, DAC 2006)
- Model Checking (Clark, Grumberg, Peled, MIT Press 2000)
- Model Checking (Baier, Katoen, MIT Press 2008)
- Binary Decision Diagrams (Bryant, ACM Comp. Surv. 1992)
- Bounded Model Checking (Biere et. al, TACAS 1999)

Outline

Ongoing Activities

V&V for the Software Reference Architecture

- New ESA study: FOREVER
- Functional requirements and verification techniques for the software reference architecture, including:
  - Formalization of functional and non-functional requirements
  - Contract-based refinement of assumptions and guarantees from system to software level
  - Integration of the software reference architecture in the process of requirements refinement and verification

intecs the Brainware company

FONDAZIONE BRUNO KESSLER

ThalesAlenia Space A Thales / Finmeccanica Company

Ongoing Activities

Model Slicing

- Input: AADL specification and logical property
- Goal: remove parts of specification that are irrelevant for model checking the property
- Reference: *Slicing AADL Specifications for Model Checking* (Odenbrett et al., NASA FM 2010)

Ongoing Activities

Model Slicing

- Input: AADL specification and logical property
- Goal: remove parts of specification that are irrelevant for model checking the property
- Reference: *Slicing AADL Specifications for Model Checking* (Odenbrett et al., NASA FM 2010)

Compositional Model Checking

- Development of compositional analysis techniques to exploit the hierarchical structure of models ("divide & conquer")
- Funded by ESA NPI program

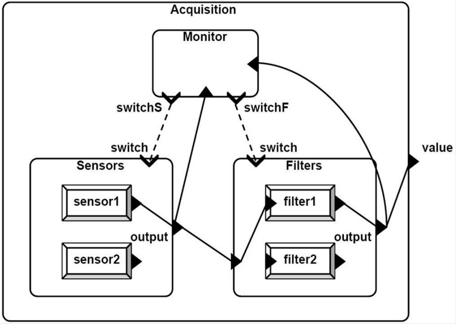
Outline

Demo Example: Sensor-Filter Acquisition System

Redundant Sensor-Filter Example: Nominal Model

- models a value acquisition system
- the value is read by a sensor, filtered by a filter, and returned as output
- two redundant sensors `sensor1` and `sensor2`
- two redundant filters `filter1` and `filter2`
- a central `Monitor` detects anomalies in either the output of the sensor or the filter, and issues a system reconfiguration (`switchS` or `switchF`) whenever needed

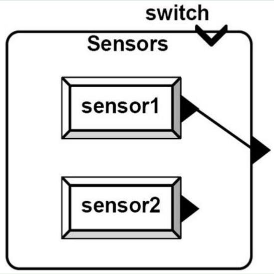
Acquisition System



Modeling Sensors: SLIM Nominal Model (1)

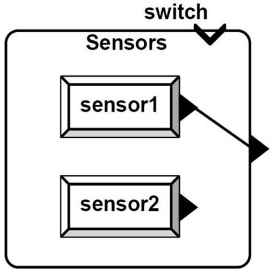
```
system Sensors
  features
    output: out data port int default 1;
    switch: in event port;
  end Sensors


system implementation Sensors.Impl
  subcomponents
    sensor1: device Sensor in modes (Primary);
    sensor2: device Sensor in modes (Backup);
  connections
    data port sensor1.output -> output in modes (Primary);
    data port sensor2.output -> output in modes (Backup);
  modes
    Primary: activation mode;
    Backup: mode;
  transitions
    Primary -[switch]-> Backup;
end Sensors.Impl;
```

Sensors Component



Modeling Sensors

Sensors Component



Modeling Sensors: SLIM Nominal Model (2)

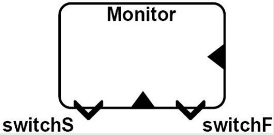
```
device Sensor
  features
    output: out data port int default 1;
end Sensor;
device implementation Sensor.Impl
  modes
    Cycle: activation mode;
  transitions
    Cycle -[when output < 5 then output := output + 1]-> Cycle;
end Sensor.Impl;
```

Modeling the Monitor

Modeling the Monitor: SLIM Nominal Model

```
fdir system Monitor
  features
    valueS: in data port int default 0;
    valueF: in data port int default 0;
    switchS: out event port;
    switchF: out event port;
    alarmS : out data port bool default false;
    alarmF : out data port bool default false;
end Monitor;

fdir system implementation Monitor.Impl
  modes
    OK: activation mode;
    FailS: mode;
    FailF: mode;
    FailSF: mode;
  transitions
    OK -[switchF when valueF = 0]-> FailF;
    OK -[switchS when valueS > 5]-> FailS;
    FailF -[switchS when valueS > 5 then alarmF := valueF = 0]-> FailSF;
    FailF -[when valueF = 0 then alarmF := true]-> FailF;
    FailS -[switchF when valueF = 0 then alarmS := valueS > 5]-> FailSF;
    FailS -[when valueS > 5 then alarmS := true]-> FailS; -- S fails again
    FailSF -[when valueF = 0 then alarmF := true; alarmS := valueS > 5]-> FailSF;
    FailSF -[when valueS > 5 then alarmS := true; alarmF := valueF = 0]-> FailSF;
end Monitor.Impl;
```

Monitor Component

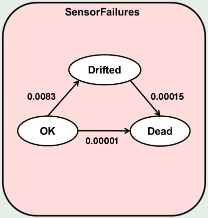


Modeling Errors

Sensor Error model:

- two faulty states: `Drifted` and `Dead`
- poisson distribution

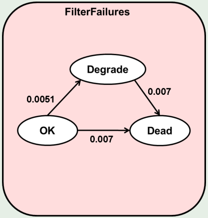
Sensor Error Model



Filter Error model:

- two faulty states: `Degrade` and `Dead`
- poisson distribution

Filter Error Model

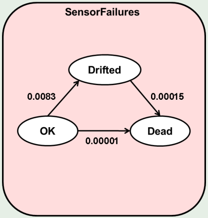


Modeling Errors

Sensor: SLIM Error Model

```
error model SensorFailures
  features
    OK: initial state;
    Drifted: error state;
    Dead: error state;
end SensorFailures;


error model implementation SensorFailures.Impl
  events
    drift: error event occurrence poisson 0.083;
    die: error event occurrence poisson 0.00001;
    dieByDrift: error event occurrence poisson 0.00015
  transitions
    OK -[ die ]-> Dead;
    OK -[ drift ]-> Drifted;
    Drifted -[ dieByDrift ]-> Dead;
end SensorFailures.Impl;
```

Sensor Error Model

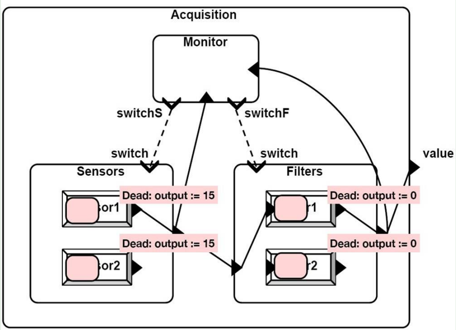


Modeling Errors

Fault Injections:

- in state Dead the output of the sensor is stuck at 15
- in state Dead the output of the filter is stuck at 0

Fault Injections



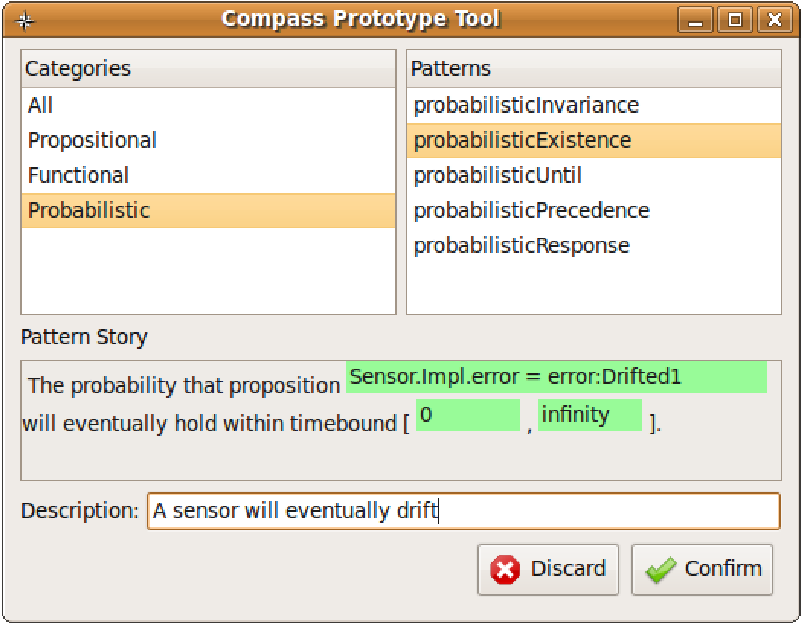
Properties of interest

Some properties of interest

- A filter or a sensor fails
- A sensor fails
  - `sensor1` fails
  - `sensor2` fails
- Filters fail twice
- Monitor reacts to filter failures
- Sensors or filters die within 76 hours
- `sensor2` fails before `filter2` within 512 hours

Outline

Creating Properties



Model Checking