COMPASS 3.0

Marco Bozzano, Harold Bruintjes, Alessandro Cimatti, Joost-Pieter Katoen, Thomas Noll, Stefano Tonetta

TACAS 2019

Prague, Czech Republic; April 9, 2019

1Funded by the European Space Agency (ESA-ESTEC) under contract 4000115870/15/NL/FE/as
2Embedded Systems Unit, Fondazione Bruno Kessler, Trento, Italy
3Software Modeling and Verification Group, RWTH Aachen University, Germany
The COMPASS mission

Develop a model-based approach to system-software co-engineering while focusing on a coherent set of modelling and analysis techniques for evaluating system-level correctness, safety, dependability, and performance of on-board computer-based aerospace systems.
The COMPASS mission

Develop a model-based approach to system-software co-engineering while focusing on a coherent set of modelling and analysis techniques for evaluating system-level correctness, safety, dependability, and performance of on-board computer-based aerospace systems.

Derived objectives

1. Modelling formalism: System-Level Integrated Modelling Language SLIM (variant of AADL)
2. Verification methodology based on state-of-the-art formal methods
3. Toolset supporting the analysis of SLIM models
4. Evaluation on industrial-size case studies from aerospace domain
History

2008

- COMPASS (rel. 2.2)
  - AUTOGEF
  - COMPASS (rel. 2.3)
  - FAME
    - HASDEL
      - DMILS
      - CATSY
      - CITADEL
    - COMPASS3 (rel. 3.0)
  - ESA-funded
  - EU-funded

2016
The COMPASS 3.0 Toolset

Intermediate Artefact

Analysis/Output

Input

System/HW/SW

Safety/Dependability

Nominal Model

Error Model

Fault Injection

Properties

Extended Model

Formal Models

Symbolic Trans. System

Markov Chain

Temp. Logic Formulas

Fault Management Effectiveness

- Fault detection
- Fault isolation
- Fault recovery
- Diagnosability
- TTPG analysis

Safety & Dependability

- Dynamic fault tree analysis (FTA)
- Dynamic FMCA
- Fault tolerance

Properties Validation

- Consistency check
- Possibility check
- Assertion check
- Contract-based refinement (and tightening)

Functional Correctness

- Model checking (discrete/hybrid)
- Simulation
- Deadlock checking
- Contract-based verification

Performability Analyses

- Performance evaluation
- Probabilistic risk
- Probabilistic assessment FTA

Signs indicate the flow of the toolset process.
A Nominal Model

Simple counter component

device Counter
features
  cnt: out data port int
  {Default => "0";};
end Counter;
device implementation Counter.Imp
states
  idle: activation state;
  wait: state;
  active: state;
transitions
  idle -[then cnt:=1]-> wait;
  wait -[then cnt:=2]-> active;
  active -[then cnt:=cnt+1]-> active;
  active -[then cnt:=0]-> idle;
end Counter.Imp;

Nominal behaviour

idle
  cnt := 1
  -> wait
  cnt := 0
  -> active
cnt := cnt+1
  active
  wait
  cnt := 2
  -> idle
A Nominal Model

Simple counter component

device Counter
features
  cnt: out data port int
  {Default => "0";};
end Counter;

device implementation Counter.Imp
states
  idle: activation state;
  wait: state;
  active: state;
transitions
  idle -[then cnt:=1] -> wait;
  wait -[then cnt:=2] -> active;
  active -[then cnt:=cnt+1] -> active;
  active -[then cnt:=0] -> idle;
end Counter.Imp;

Nominal behaviour

idle          cnt := 1          wait
               
cnt := 0

active


cnt := cnt+1

Many more features

- data types + time units
- timed + hybrid behaviour
  (guards, invariants, linear DEs)
- component hierarchies
- event signals + synchronisation
- port connections + data flows
- system reconfiguration
- ...

5 of 13 COMPASS 3.0
Thomas Noll
TACAS 2019
An Error Model

### Binary error model

```plaintext
error model CounterFailure
features
  ok: activation state;
  failed: error state;
end CounterFailure;

error model implementation
  CounterFailure.Imp
  events
    fail: error event
      occurrence poisson 0.01;
    recover: error event
      occurrence poisson 0.5;
  transitions
    ok -[fail]-> failed;
    failed -[recover]--> ok;
end CounterFailure.Imp;
```

### Error behaviour

- **ok** → **failed**
- **failed** → **recover**
- **recover** → **ok**
Binary error model

```java
error model CounterFailure
features
  ok: activation state;
  failed: error state;
end CounterFailure;

error model implementation CounterFailure.Imp
events
  fail: error event
    occurrence poisson 0.01;
  recover: error event
    occurrence poisson 0.5;
transitions
  ok -[fail]--> failed;
  failed -[recover]--> ok;
end CounterFailure.Imp;
```

Error behaviour

- ok
- fail
- recover
- failed

Many more features

- error propagations
- reset events
- ...

An Error Model

Many more features
Model Extension by Example

Nominal behaviour

idle\[\text{cnt} := 1\] \rightarrow \text{wait}
idle\[\text{cnt} := 0\] \rightarrow \text{active}
active \rightarrow \text{cnt} := \text{cnt} + 1
active \rightarrow \text{cnt} := \text{cnt} + 1

Error behaviour

ok \rightarrow \text{fail} \rightarrow \text{failed}
ok \rightarrow \text{recover} \rightarrow \text{failed}
ok \rightarrow \text{recover} \rightarrow \text{failed}
ok \rightarrow \text{recover} \rightarrow \text{failed}
Model Extension by Example

Nominal behaviour

- idle
  - cnt := 1
  - cnt := 0
  - active
  - cnt := cnt+1
- wait
  - cnt := 2

Fault injection

- failed: cnt := -1

Error behaviour

- ok
- fail
- recover
- failed

Automatically extended model

- idle#ok
- idle#failed
- wait#ok
- wait#failed
- active#ok
- active#failed
- cnt := 1
- cnt := 2
- cnt := 0
- cnt := cnt+1
- fail: cnt := -1
- fail: cnt := -1
- fail: cnt := -1
- recover
- recover
- recover
- cnt := -1
- cnt := -1
- cnt := -1
- cnt := -1
Model Extension by Example

Nominal behaviour

idle → wait

wait

cnt := 1

cnt := 0

cnt := 2

cnt := cnt+1

active

Fault injection

failed: cnt := -1

Error behaviour

ok ↔ fail ↔ failed

recover

Automatically extended model

idle#ok → idle#failed

idle#failed

idle#ok

cnt := 1

cnt := -1

wait#ok → wait#failed

wait#failed

wait#ok

cnt := 0

cnt := -1

active#ok → active#failed

active#failed

active#ok

cnt := cnt+1

cnt := -1

cnt := -1

recover

recover

recover

recover
Contract-Based Compositional Reasoning

Contract refinement

system PSU
...

  properties
  SLIMpropset::Contracts => ([Name => "power";
                   Assumption => "true";
                   Guarantee => "always (has_power)";])
end PSU;

system implementation Power.Imp
  subcomponents
    battery: system Battery;
    generator: system Generator;
  ...

  properties
  SLIMpropset::ContractRefinements => ([Contract => "power";
                   SubContracts => ("battery.power", "generator.power");])
end Power.Imp;
Contract-Based Compositional Reasoning

![COMPASS Toolset](image)

- **Contracts**
  - **Validation**
  - **Contract Refinement**
  - **Tightening**

- **Algorithms**
  - BMC
  - klive
  - kzeno

- **Bound**
  - 0

- **Summary**: Everything is OK

- **Example**
  - maxdelay
    - implementation.maxdelay (OK)
    - environment.sub1.maxdelay (OK)
    - environment.sub2.maxdelay (OK)
Hierarchical Fault Trees
Timed Failure Propagation Graphs
Case Studies

- Preliminary Design Review (PDR) of satellite platform (FDIR, 90 components) [ICSE 2012]
- Critical Design Review (CDR) of satellite platform (diagnosability, 250 components) [RESS 2014]
- Launcher system (probabilistic reachability, 40 components) [DSN 2015]
- Solar orbiter (failure propagation, 40 components) [PhD Bittner 2016]
Welcome to COMPASS!

The COMPASS Project is an international research project for developing a theoretical and technological basis for the system-software co-engineering approach focusing on a coherent set of specification and analysis techniques for evaluation of system-level correctness, safety, dependability and performability of on-board computer-based aerospace systems. These techniques shall significantly improve the reliability of modern and future space missions.

Modeling

COMPASS uses the SLIM language, based on AADL, for its input models. Using it, it is possible to describe both the hardware and software components of the system, and their connections. Separate error models can be defined to describe faults, which can automatically be injected into the model.

Example SLIM model

Properties can be defined for the model to describe desired or undesired behavior, and are specified by means of patterns, design attributes or directly as formulas.