

# Safety, Dependability and Performance Analysis of Aerospace Systems

Thomas Noll<sup>(✉)</sup>

Software Modeling and Verification Group,  
RWTH Aachen University, Aachen, Germany  
`noll@cs.rwth-aachen.de`  
<http://moves.rwth-aachen.de/>

**Abstract.** The size and complexity of software in spacecraft is increasing exponentially, and this trend complicates its validation within the context of the overall spacecraft system. Current validation methods are labour-intensive as they rely on manual analysis, review and inspection. In this paper we give an overview of an integrated 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. It features both a tailored modelling language and toolset for supporting (semi-)automated validation activities. Our modelling language is a dialect of the Architecture Analysis and Design Language, AADL, and enables engineers to specify the system, the software, and their reliability aspects. The COMPASS toolset employs state-of-the-art model checking techniques, both qualitative and probabilistic, for the analysis of requirements related to functional correctness, safety, dependability and performance.

## 1 Introduction

Building modern aerospace systems is highly demanding. They should be extremely dependable, offering service without interruption (i.e., without failure) for a very long time – typically years or decades. Whereas “five nines” dependability, i.e., a 99.999 % availability, is satisfactory for most safety-critical systems, for aerospace on-board systems it is not. Faults are costly and may severely damage reputations. Dramatic examples are known. Fatal defects in the control software of the Ariane-5 rocket and the Mars Pathfinder have led to headlines in newspapers all over the world. Rigorous design support and analysis techniques are called for. Bugs must be found as early as possible in the design process while performance and reliability guarantees need to be checked

---

We thank all co-workers in the COMPASS project for their contributions, including the groups of Alessandro Cimatti (FBK, Trento, IT), Xavier Olive (Thales Alenia Space, FR), David Lesens (Airbus Defence and Space, FR) and Yuri Yushtein (ESA/ESTEC, NL). This research has been funded by the European Space Agency via several grants.

whenever possible. The effect of fault diagnosis, isolation and recovery must be quantifiable.

Tailored effective techniques exist for specific system-level aspects. Peer reviewing and extensive testing detect most of the software bugs, performance is checked using queueing networks or simulation, and hardware safety levels are analysed using a profiled Failure Modes and Effects Analysis (FMEA) approach. Fine. But how is the consistency between the analysis results ensured? What is the relevance of a zero-bug confirmation if its analysis is based on a system view that ignores critical performance bottlenecks? There is a clear need for an integrated, coherent approach! This is easier said than done: the inherent heterogeneous character of on-board systems involving software, sensors, actuators, hydraulics, electrical components, etc., each with its own specific development approach, severely complicates this.

The COMPASS project [15] advances the system-software perspective by providing means for its validation in the early design phases, such that system architecture, software architecture, and their interfacing requirements are aligned with the overall functional intents and risk tolerances. Validation in the current practice is labour-intensive and consists mostly of manual analysis, review and inspection. We improve upon this by adopting a *model-based approach* using formal methods. In COMPASS, the system, the software and its reliability models are expressed in a single modelling language. This language originated from the need for a language with a rigorous formal semantics, and it is a dialect of the Architecture Analysis & Design Language (AADL). Models expressed in our AADL dialect are processed by the COMPASS toolset that automates analyses which are currently done manually. The automated analyses allow studying functional correctness of discrete, real-time and hybrid aspects under degraded modes of operation, generating safety & dependability validation artefacts, performing probabilistic risk assessments, and evaluating effectiveness of fault management. The analyses are mapped onto discrete, symbolic and probabilistic model checkers, but all of them are completely hidden away from the user by appropriate model-transformations. The COMPASS toolset is thus providing an easy-to-use push-button analysis technology.

The first ideas and concepts for the development of the COMPASS toolset emerged in 2007, due to a series of significant advances in model checking [2], and especially in its probabilistic counterpart [1]. These advances opened prospects for an integrated model-based approach towards system-software correctness validation, safety & dependability assessment and performance evaluation during the design phase. Its technology readiness level was estimated at level 1, i.e. basic principles were observed and reported. The European Space Agency (ESA) issued a statement of work to improve system-software co-engineering and this was commissioned to the COMPASS consortium consisting of RWTH Aachen University, Fondazione Bruno Kessler and Thales Alenia Space. Development started soon after, and in 2009 a COMPASS toolset prototype was delivered to the European space industry. Maturation was followed by subsystem-level

case studies performed by Thales Alenia Space [10]. As of 2012, two large pilot projects took place in ESA for a spacecraft in development. This marked the maturation of the COMPASS toolset to early level 4, namely laboratory-tested. This paper summarises the background work. Altogether, it describes the current state of the art in system-software spacecraft co-engineering, ranging from the used techniques, to the tools and the conducted industrial projects.

The remainder is organised as follows. An introduction to the developed modelling language is given in Sect. 2, followed by an overview of the toolset and its supported analyses in Sect. 3. Section 4 draws a conclusion about the evaluation activities.

## 2 Modelling Using an AADL Dialect

The Architecture Analysis and Design Language (AADL) [24, 35] is an industry standard for modelling safety-critical system architectures and it is developed and governed by the Society of Automotive Engineers (SAE). Although standardized by the SAE, it is backed by the aerospace community as well. AADL provides a cohesive and uniform approach to model heterogeneous systems, consisting of software (e.g., processes and threads) and hardware (e.g., processors and buses) components, and their interactions. Our variant of AADL was designed to meet the needs of the European space industry. It extends a core fragment of AADL 1.0 [32] by supporting the following essential features:

- Modelling both the system’s *nominal and faulty behaviour*. To this aim, AADL provides primitives to describe software and hardware faults, error propagation (i.e., turning fault occurrences into failure events), sporadic (transient) and permanent faults, and degraded operation modes (by mapping failures from architectural to service level).
- Modelling (partial) *observability* and the associated observability requirements. These notions are essential to deal with diagnosability and Fault Detection, Isolation and Recovery (FDIR) analyses.
- Specifying *timed and hybrid behaviour*. In particular, to analyze continuous physical systems such as mechanics and hydraulics, our modelling language supports continuous real-valued variables with (linear) time-dependent dynamics.
- Modelling *probabilistic* aspects. These are important to specify random faults and systems repairs with stochastic timing.

In the following, we present the capabilities of our AADL dialect using a running example. A complete AADL specification consists of three parts, namely a description of the nominal behaviour, a description of the error behaviour and a fault injection specification that describes how the error behaviour influences the nominal behaviour. These three parts are discussed below. Due to space constraints, we refer the interested reader to [12] for a description of the formal semantics.

## 2.1 Nominal Behaviour

An AADL model is hierarchically organized into *components*, distinguished into software (processes, threads, data), hardware (processors, memories, devices, buses), and composite components (called *systems*). Components are defined by their *type* (specifying the functional interfaces as seen by the environment) and their *implementation* (representing the internal structure). An example of a component’s type and implementation for a simple battery device [8] is shown in Fig. 1.

The component type describes the ports through which the component communicates. For example, the type interface of Fig. 1 features three ports, namely an outgoing event port **empty** which indicates that the battery is about to become discharged, an incoming data port **tryReset** which indicates that the battery device should (attempt to) reset, and an outgoing data port **voltage** which makes its current voltage level accessible to the environment.

A component implementation defines its subcomponents, their interaction through (event and data) port connections, the (physical) bindings at runtime, the operational behaviour via modes, the transitions between them, which are spontaneous or triggered by events arriving at the ports, and the timing and hybrid behaviour of the component. For example, the implementation of Fig. 1 specifies the battery to be in the **charged** mode whenever activated, with an **energy** level of 100 % as indicated by the **default** value of 1.0. This level is continuously decreased by 2 % (of the initial amount) per time unit (**energy’** denotes the first derivative of **energy**) until a threshold value of 20 % is reached, upon which the battery changes to the **depleted** mode. This mode transition triggers the **empty** output event, and the loss rate of energy is increased to 3 %. Moreover, the **voltage** value is regularly computed from the **energy** level (ranging between 6.0 and 4.0 [volts]) and made accessible to the environment via the corresponding outgoing data port. In addition, the battery reacts to the **tryReset** port to decide when a **reset** operation should be performed in reaction to faulty behaviour (see the description of error models below).

In general, the mode transition system—basically a finite-state automaton—describes how the component evolves from mode to mode while performing events. Invariants on the values of data components (such as “**energy** >= 0.2” in mode **charged**) restrict the residence time in a mode. Trajectory equations (such as those associated with **energy’**) specify how continuous variables evolve while residing in a mode. This is akin to timed and hybrid automata [28]. Here we assume that all invariants are linear. Moreover we constrain the derivatives occurring in trajectory equations to real constants, i.e., the evolution of continuous variables is described by simple linear functions.

A mode transition is given by  $m - [e \text{ when } g \text{ then } f] -> m'$ . It asserts that the component can evolve from mode  $m$  to mode  $m'$  upon occurrence of event  $e$  (the trigger event) provided that guard  $g$ , a Boolean expression that may depend on the component’s (discrete and continuous) data elements, holds. Here “data elements” refers to (both incoming and outgoing) data ports and

```

device Battery
  features
    empty: out event port;
    tryReset: in data port bool default false;
    voltage: out data port real default 6.0;
end Battery;
device implementation Battery.Imp
  subcomponents
    energy: data continuous default 1.0;
  modes
    charged: initial mode while energy' = -0.02 and energy >= 0.2;
    depleted: mode while energy' = -0.03 and energy >= 0.0;
  transitions
    charged -[then voltage := 2.0*energy+4.0]-> charged;
    charged -[reset when tryReset]-> charged;
    charged -[empty when energy = 0.2]-> depleted;
    depleted -[then voltage := 2.0*energy+4.0]-> depleted;
    depleted -[reset when tryReset]-> depleted;
end Battery.Imp;

```

Fig. 1. Specification of a battery component.

```

system Power
  features
    alert: out data port bool observable;
end Power;
system implementation Power.Imp
  subcomponents
    batt1: device Battery in modes (primary);
    batt2: device Battery in modes (backup);
    mon: device Monitor;
  connections
    data port batt1.voltage -> mon.voltage in modes (primary);
    data port batt2.voltage -> mon.voltage in modes (backup);
    data port mon.alert -> alert;
    data port mon.alert -> batt1.tryReset in modes (primary);
    data port mon.alert -> batt2.tryReset in modes (backup);
  modes
    primary: initial mode;
    backup: mode;
  transitions
    primary -[batt1.empty]-> backup;
    backup -[batt2.empty]-> primary;
end Power.Imp;

```

Fig. 2. The complete power system.

```

device Monitor
  features
    voltage: in data port real;
    alert: out data port bool;
  end Monitor;
device implementation Monitor.Imp
  flows
    alert := (voltage < 4.5);
  end Monitor.Imp;

```

**Fig. 3.** Specification of the monitor.

data subcomponents of the respective component. On transiting, the effect  $f$  which may update data subcomponents or outgoing data ports (like **voltage**) is applied. The presence of event  $e$ , guard **when**  $g$  and effect **then**  $f$  is optional. If absent,  $e$  defaults to an internal event,  $g$  to **true**, and  $f$  to the empty effect.

Mode transitions may give rise to modifications of a component's configuration: subcomponents can become (de-)activated and port connections can be (de-)established. This depends on the **in modes** clause, which can be declared along with port connections and subcomponents. This is demonstrated by the specification in Fig. 2, which shows the usage of the battery component in the context of a redundant power system. It contains two instances of the battery device, namely **batt1** and **batt2**, being respectively active in the **primary** and the **backup** mode. The mode switch that initiates reconfiguration is triggered by an **empty** event arriving from the battery that is currently active. The data ports are reconfigured too in this example. The **voltage** port of **batt2** is connected to the overall power system once switched to the **backup** mode.

A similar reconfiguration is also performed for the alerts from the monitor component, which checks the current voltage level and raises an alarm if it falls below a critical threshold of 4.5 [volts]. Its specification is shown in Fig. 3; it employs another modelling concept, a so-called *flow*. A flow establishes a direct dependency between an outgoing data port of a component and (some of) its incoming data ports, meaning that a value update of one of the given incoming data ports immediately causes a corresponding update of the outgoing data port.

## 2.2 Error Behaviour

*Error models* are an extension to the specification of nominal models [34] and are used to conduct safety and dependability analyses. For modularity, they are defined separately from nominal specifications. Akin to nominal models, an error model is defined by its type and its associated implementation.

An error model *type* defines an interface in terms of error states and (incoming and outgoing) error propagations. Error *states* are employed to represent the current configuration of the component with respect to the occurrence of errors. Error *propagations* are used to exchange error information between components. They are similar to input and output event ports, but differ in that error events

```

error model BatteryFailure
  features
    ok: initial state;
    dead: error state;
    resetting: error state;
    batteryDied: out error propagation;
  end BatteryFailure;
error model implementation BatteryFailure.Imp
  events
    fault: error event occurrence poisson 0.001;
    works: error event occurrence poisson 0.2;
    fails: error event occurrence poisson 0.8;
  transitions
    ok -[fault]-> dead;
    dead -[batteryDied]-> dead;
    dead -[reset]-> resetting;
    resetting -[works]-> ok;
    resetting -[fails]-> dead;
end BatteryFailure.Imp;

```

**Fig. 4.** Specification of the battery error model.

are matched by identifier rather than by an explicit declaration of an event port connection.

An error model *implementation* provides the structural details of the error model. It is defined by a (probabilistic) machine over the error states declared in the error model type. Transitions between states can be triggered by error events, reset events, and error propagations.

Figure 4 presents a basic error model for the battery device. It defines a probabilistic error event, **fault**, which occurs once every 1000 time units on average. Whenever this happens, the error model changes into the **dead** state. In the latter, the battery failure is signalled to the environment by means of the outgoing error propagation **batteryDied**. Moreover, the battery is enabled to receive a **reset** event from the nominal model to which the error behaviour is attached. It causes a transition to the **resetting** state, from which the battery recovers with a probability of 20 %, and returns to the **dead** state otherwise.

### 2.3 Fault Injection

As error models bear no relation with nominal models, an error model does not influence the nominal model unless they are linked through *fault injection*.

A fault injection describes the effect of the occurrence of an error on the nominal behaviour of the system. More concretely, it specifies the value update that a data element of a component implementation undergoes when its associated error model enters a specific error state. To this aim, each fault injection has to be given by the user by specifying three parts: a state *s* in the error model

(such as **dead** in Fig. 4), an outgoing data port or subcomponent  $d$  in the nominal model (such as **voltage** in Fig. 1), and the fault effect given by the expression  $a$  (such as the value 0, indicating the collapse of power). Multiple fault injections between error models and nominal models are possible.

The automatic procedure that integrates both models and the given fault injections, the so-called *model extension*, works as follows. The principal idea is that the nominal and error models are running concurrently. That is, the state space of the extended model consists of pairs of nominal modes and error states, and each transition in the extended model is due to a nominal mode transition, an error state transition, or a combination of both (in case of a reset operation). The aforementioned fault injection becomes enabled whenever the error model enters state  $s$ . In this case the assignment  $d := a$  is carried out, i.e., the data subcomponent  $d$  is assigned with the fault effect  $a$ . This error effect is maintained as long as the error model stays in state  $s$ , overriding possible assignments to  $d$  in the nominal model. When  $s$  is left, the fault injection is disabled (though another one may be enabled). An example of an extended model can be found in [12].

### 3 The COMPASS Toolset

The COMPASS toolset is the result of a significant implementation effort carried out by the COMPASS Consortium. The GUI and most subcomponents are implemented in Python, using the PyGTK library. Pre-existing components, such as the NuSMV and MRMC model checker, are instead written in C. Overall, the core of the toolset consists of about 100,000 lines of Python code. Figure 5 shows the functionality of the toolset.

COMPASS takes as input one or more AADL models, and a set of properties. The latter are provided in the form of instantiated property *patterns* [17, 25], which are templates containing placeholders that have to be filled in by the user. The COMPASS toolset provides templates for the most frequently used patterns, that ease property specifications by non-experts through hiding the details of the underlying temporal logic. The tool generates several outputs, such as traces, fault trees and FMEA tables, diagnosability and performability measures.

The toolset builds upon the following main components. NuSMV [14, 23] (New Symbolic Model Verifier) is a symbolic model checker that supports state-of-the-art verification techniques such as BDD-based and SAT-based verification for CTL and LTL [2]. MRMC [30, 31] (Markov Reward Model Checker) is a probabilistic model checker that supports the analysis of discrete-time and continuous-time Markov reward models. Specifications are written in PCTL (Probabilistic Computation Tree Logic) and CSL (Continuous Stochastic Logic [1], a probabilistic real-time version of CTL). SigRef [37] is used to minimize, amongst others, Interactive Markov Chains (IMC) [29] based on various notions of bisimulation. It is a symbolic tool using multi-terminal BDD representations of IMCs and applies signature-based minimization algorithms. A walkthrough of the toolset in terms of its screenshots is shown in Fig. 6.



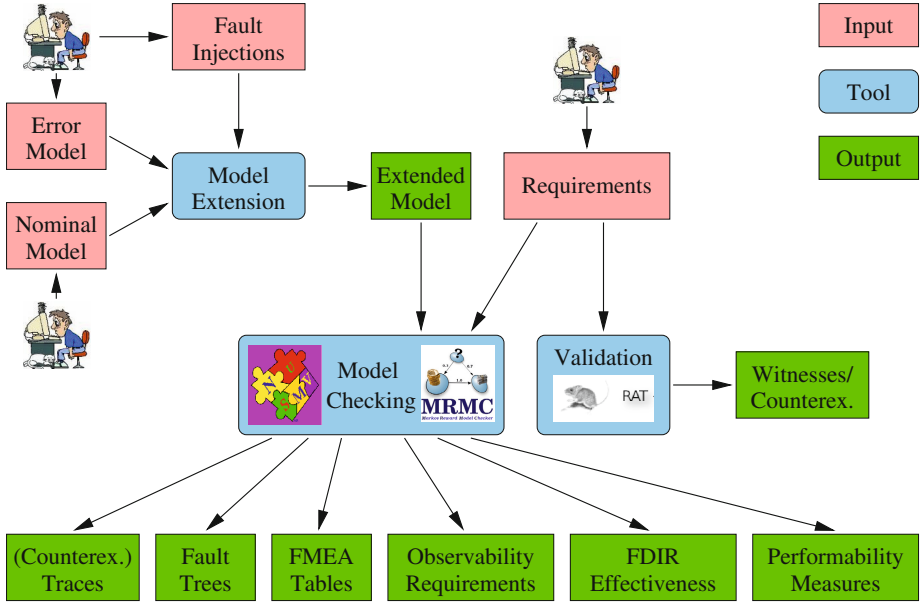


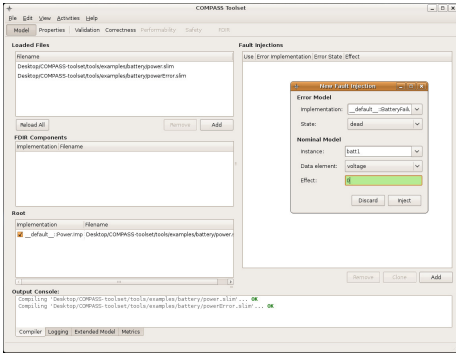
Fig. 5. Functional view of the COMPASS platform.

The tool also supports a graphical notation of our AADL dialect, that is a derivation of the AADL graphical notation [33]. We developed a graphical drawing editor enabling engineers to construct models visually using the adopted graphical notation. The editor is called the COMPASS Graphical Modeller and is part of the COMPASS toolset.

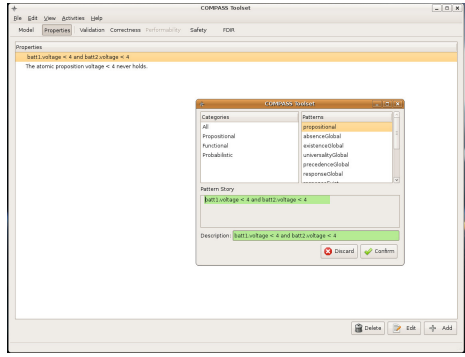
### 3.1 Functional Correctness

COMPASS supports random and guided *model-based simulation* of AADL models. Guided simulation can be performed by choosing either the next transition to be taken, or a target value for one or more variables. The generated traces can be inspected using a trace manager that displays the values of the model variables of interest (filtering is possible) for each step.

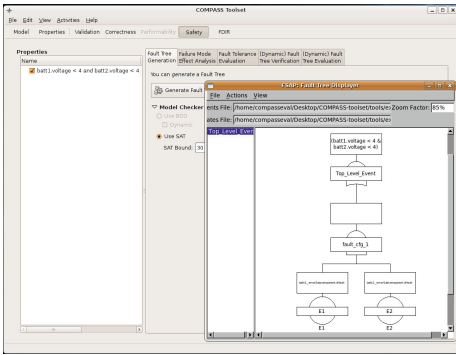
*Property verification* is based on model checking [2], an automated technique that verifies whether a property expressed in temporal logic, holds for a given model. Symbolic techniques [3, 4, 27] are used to tackle the problem of state space explosion. COMPASS relies on the NuSMV model checker, which supports both BDD-based and SAT-based verification for finite-state systems, and SMT-based verification techniques for timed and hybrid systems, based on the MathSAT solver [7, 22]. On refutation of a property, a counterexample is generated, showing an execution trace of the model violating the property. An example of this is shown in Fig. 6(d). Finally, it is possible to run *deadlock checking*, in order to pinpoint deadlocks (i.e., states with no outgoing transitions) in the model.



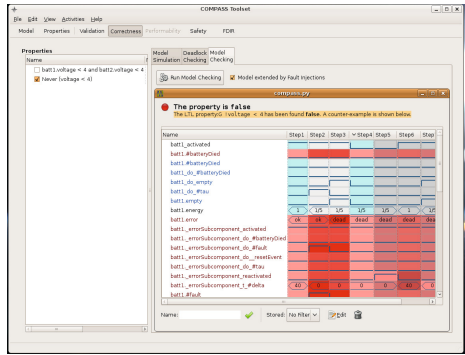
(a) Adding a fault injection.



(b) Adding a property.



(c) A generated fault tree.



(d) A model-checking counterexample.

**Fig. 6.** Walkthrough of the COMPASS toolset.

### 3.2 Safety Assessment

COMPASS implements model-based safety assessment techniques, based on symbolic model checking [9, 21], and supports traditional techniques such as *Failure Mode and Effects Analysis* (FMEA) [19] and *Fault Tree Analysis* (FTA) [18]. FMEA is an inductive technique that starts by identifying a set of (combinations of) failure modes and, using forward reasoning, assesses their impact on a set of system properties. The results are summarised in an *FMEA table*. It is also possible to generate *dynamic* FMEA tables, namely to enforce an order of occurrence between failure modes. FTA is a deductive technique, which, given a *top-level event* (TLE), i.e., the specification of an undesired condition, constructs all possible chains of basic faults that contribute to its occurrence. Pictorially, these chains are organized in a *fault tree* with a two-layer logical structure, corresponding to the disjunction of its minimal cut sets [9] (MCSs), where each MCS is a conjunction of basic faults. COMPASS also supports the generation of dynamic fault trees [6], where ordering constraints between basic faults are represented using priority AND (PAND) gates. Figure 6c depicts a simple fault tree for the

power system model of Sect. 2, where the top level event is “`batt1.voltage < 4.0 and batt2.voltage < 4.0`”. The tree shows that the only cause that can lead to the occurrence of TLE is when both batteries die.

### 3.3 Diagnosability and FDIR Analysis

The COMPASS toolset supports diagnosability and FDIR (Fault Detection, Isolation and Recovery) effectiveness analysis. These analyses work under the hypothesis of *partial observability*. Variables and ports in our AADL dialect can be declared to be observable (see, e.g., the data port `alert` in Fig. 2).

*Diagnosability analysis* investigates the possibility for an ideal diagnosis system to infer accurate and sufficient run-time information on the behaviour of the observed system. The COMPASS toolset follows the approach described in [13], where the violation of a diagnosability condition is reduced to the search of *critical pairs* in the so-called *twin plant* model, i.e., a pair of executions that are observationally indistinguishable but hide conditions that should be distinguished. As an example, property “`batt1.voltage < 4.0 and batt2.voltage < 4.0`” is not diagnosable, as the `alert` observable does not allow to distinguish the case where the batteries’ voltages are low from the case where they are depleted through use. If we add the observable “`alert2 := (voltage < 4.0)`”, then the property becomes diagnosable. Using techniques similar to those used for computing MCSs, it is also possible to automatically synthesize a set of observables that ensure diagnosability of a given model [5].

*FDIR effectiveness analysis* is a set of analyses carried out on an existing fault management subsystem. Fault detection is concerned with detecting whether a given system is malfunctioning, namely searching for observable signals such that every occurrence of the fault will eventually make them true. As an example, observable `alert` is a detection means for property “`batt1.voltage < 4.0 and batt2.voltage < 4.0`”. Fault isolation analysis aims at identifying the specific cause of malfunctioning. It generates a fault tree that contains the minimal explanations that are compatible with the observable being true. As an example, observable `alert` has two possible failure explanations: either `batt1` has died, or `batt2` has died. The latter failure, that `batt2` has died, is not dependent on the death of `batt1`, since the switch-over to the second battery can also occur by natural depletion of the first battery. Finally, fault recovery analysis is used to check whether a user-specified recoverability property holds. For instance, property “`always (batt1.voltage < 4.4 implies eventually batt1.voltage > 5.5)`” is true in the nominal model, but it is false when error behaviour is taken into account, as a battery may die.

### 3.4 Performability Analysis

We use probabilistic model checking techniques [2, Ch. 10] for analyzing a model on its performance. The COMPASS toolset in particular supports performance properties expressed in the probabilistic pattern system by [25]. It allows for

the formal specification of steady-state, transient probabilities, timed reachability probabilities and more intricate performance measures such as combinations thereof. Examples of typical performance parameters are “the probability that the first battery dies within 100 h” or “the probability that both batteries die within the mission duration”. These properties have a direct mapping to Continuous Stochastic Logic (CSL) [1] and are input to the underlying probabilistic model checker.

The probabilistic model checker furthermore requires a Markov model as input. This is obtained from the extended model through several steps. First, the extended model’s reachable state space is generated through an exhaustive symbolic exploration. Second, the probabilistic rates as specified in the error models (cf. Sect. 2.2) are interwoven through the state space by replacing the transition label with the associated probabilistic rate. The resulting state space is a symbolic representation of an Interactive Markov Chain, i.e., a Continuous-Time Markov Chain (CTMC) that may exhibit non-determinism [29]. This IMC is passed through the third phase, in which its size is reduced using weak bisimulation minimization [16, 36]. In this last step, the IMC may turn into a CTMC. In the final phase the CSL formulae are extracted from the performance requirements and then together with the CTMC are fed to the MRMC probabilistic model checker, to compute the desired probabilities. The result is a graph showing the cumulative distribution function over the time horizon specified in the performance requirement. In case the resulting IMC from the model does not yield a CTMC after bisimulation minimization, new analysis techniques using real-time stochastic games can be used [26]. These techniques are planned to be integrated into the toolset. Similar techniques are also used for fault tree evaluation, i.e., computing the probability of the top-level event in dynamic fault trees [6].

## 4 Industrial Evaluation

The COMPASS approach and toolset was intensively tested on serious industrial cases by Thales Alenia Space in Cannes (France). These cases include thermal regulation in satellites and satellite mode management with its associated FDIR strategy. It was concluded that the modelling approach based on AADL provides sufficient expressiveness to model all hardware and software subsystems in satellite avionics. The hierarchical structure of specifications and the component-based paradigm enables the reuse of models. Also incremental modelling is very well supported. The Reliability, Availability, Maintainability and Safety (RAMS) analyses as provided by the toolset were found to be mature enough to be adopted by industry, indicating that the integrated COMPASS approach significantly reduces the time and cost for safety analysis compared to traditional on-board design processes [38]. Those findings were confirmed by applying our formal modelling and analysis techniques on a regular industrial-size design of a modern satellite platform in parallel with the conventional software development of the platform [11, 20].

## References

1. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. *IEEE Trans. Softw. Eng.* **29**(6), 524–541 (2003)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
3. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
4. Biere, A., Heljanko, K., Junttila, T.A., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. *Log. Methods Comput. Sci.* **2**(5), 1–64 (2006)
5. Bittner, B., Bozzano, M., Cimatti, A., Olive, X.: Symbolic synthesis of observability requirements for diagnosability. In: Proceedings of 11th Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA 2011), ESA/ESTEC (2011) [http://robotics.estec.esa.int/ASTRA/Astra2011/Astra2011\\_Proceedings.zip](http://robotics.estec.esa.int/ASTRA/Astra2011/Astra2011_Proceedings.zip)
6. Boudali, H., Crouzen, P., Stoelinga, M.: A rigorous, compositional and extensible framework for dynamic fault tree analysis. In: Dependable and Secure Computing, pp. 128–143. IEEE (2010)
7. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Schulz, S., Sebastiani, R.: Mathsats: tight integration of SAT and mathematical decision procedures. *J. Autom. Reasoning* **35**, 265–293 (2005)
8. Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Roveri, M.: The COMPASS approach: correctness, modelling and performability of aerospace systems. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) SAFECOMP 2009. LNCS, vol. 5775, pp. 173–186. Springer, Heidelberg (2009)
9. Bozzano, M., Cimatti, A., Tapparo, F.: Symbolic fault tree analysis for reactive systems. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 162–176. Springer, Heidelberg (2007)
10. Bozzano, M., Cavada, R., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Olive, X.: Formal verification and validation of aadl models. In: Embedded Real Time Software and Systems Conference, AAAF & SEE (2010)
11. Bozzano, M., Cimatti, A., Katoen, J.P., Katsaros, P., Mokos, K., Nguyen, V.Y., Noll, T., Postma, B., Roveri, M.: Spacecraft early design validation using formal methods. *Reliab. Eng. Syst. Saf.* **132**, 20–35 (2014)
12. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, dependability, and performance analysis of extended AADL models. *Comput. J.* **54**(5), 754–775 (2011)
13. Cimatti, A., Pecheur, C., Cavada, R.: Formal verification of diagnosability via symbolic model checking. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 363–369. Morgan Kaufmann (2003)
14. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
15. COMPASS Consortium: The COMPASS project web site. <http://compass.informatik.rwth-aachen.de/>
16. Derisavi, S., Hermanns, H., Sanders, W.H.: Optimal state-space lumping in Markov chains. *Inf. Process. Lett.* **87**(6), 309–315 (2003)

17. Dwyer, M., Avrunin, G., Corbett, J.: Patterns in property specifications for finite-state verification. In: International Conference on Software Engineering (ICSE), pp. 411–420. IEEE CS Press (1999)
18. ECSS: Space product assurance: Fault tree analysis - adoption notice ECSS/IEC 61025. ECSS Standard Q-ST-40-12C, European Cooperation for Space Standardization, July 2008
19. ECSS: Space product assurance: Failure modes, effects (and criticality) analysis (FMEA/FMECA). ECSS Standard Q-ST-30-02C, European Cooperation for Space Standardization, March 2009
20. Esteve, M.A., Katoen, J.P., Nguyen, V.Y., Postma, B., Yushtein, Y.: Formal correctness, safety, dependability and performance analysis of a satellite. In: 34th International Conference on Software Engineering (ICSE 2012), pp. 1022–1031. ACM and IEEE CS Press (2012)
21. FBK: FSAP: The formal safety analysis platform. <http://fsap.fbk.eu/>
22. FBK: MathSAT. <http://mathsat.fbk.eu>
23. FBK: NuSMV: A new symbolic model checker. <http://nusmv.fbk.eu>
24. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: an introduction to the sae architecture analysis & design language. Addison-Wesley Professional, Boston (2012)
25. Grunske, L.: Specification patterns for probabilistic quality properties. In: International Conference on Software Engineering (ICSE), pp. 31–40. ACM (2008)
26. Guck, D., Han, T., Katoen, J.-P., Neuhäuser, M.R.: Quantitative timed analysis of interactive Markov chains. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 8–23. Springer, Heidelberg (2012)
27. Heljanko, K., Junttila, T.A., Latvala, T.: Incremental and complete bounded model checking for full PLTL. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 98–111. Springer, Heidelberg (2005)
28. Henzinger, T.: The theory of hybrid automata. In: IEEE Symposium on Logic in Computer Science (LICS), pp. 278–292. IEEE CS Press (1996)
29. Hermanns, H.: Interactive Markov chains in practice. In: Hermanns, H. (ed.) Interactive Markov Chains. LNCS, vol. 2428, p. 129. Springer, Heidelberg (2002)
30. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. *Perform. Eval.* **68**(2), 90–104 (2011)
31. MRMC Consortium: MRMC – The Markov Reward Model Checker. <http://www.mrmc-tool.org/>
32. SAE: Architecture Analysis and Design Language (AADL). SAE Standard AS5506, International Society of Automotive Engineers, May 2004
33. SAE: Architecture Analysis and Design Language (AADL) Annex, Volume 1, Annex A: Graphical AADL Notation. SAE Standard AS5506/1, International Society of Automotive Engineers, June 2006
34. SAE: Architecture Analysis and Design Language Annex (AADL), Volume 1, Annex E: Error Model Annex. SAE Standard AS5506/1, International Society of Automotive Engineers, June 2006
35. SAE: Architecture Analysis and Design Language (AADL) Rev. B. SAE Standard AS5506B, International Society of Automotive Engineers, September 2012
36. Valmari, A., Franceschinis, G.: Simple  $O(m \log n)$  time Markov chain lumping. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 38–52. Springer, Heidelberg (2010)

37. Wimmer, R., Herbstritt, M., Hermanns, H., Strampp, K., Becker, B.: Sigref – a symbolic bisimulation tool box. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 477–492. Springer, Heidelberg (2006)
38. Yuste, Y., Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V., Noll, T., Olive, X., Roveri, M.: System-software co-engineering: dependability and safety perspective. In: Proceedings of the 4th IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT 2011), pp. 18–25. IEEE CS Press (2011)

Formal Techniques for Safety-Critical Systems  
Third International Workshop, FTSCS 2014,  
Luxembourg, November 6-7, 2014. Revised Selected  
Papers  
Artho, C.; Ölveczky, P.C. (Eds.)  
2015, X, 257 p. 86 illus., Softcover  
ISBN: 978-3-319-17580-5