Spacecraft Early Design Validation using Formal Methods

Marco Bozzano[a], Alessandro Cimatti[a], Joost-Pieter Katoen[b], Panagiotis Katsaros[d], Konstantinos Mokos[c,2], Viet Yen Nguyen[c,3,*], Thomas Noll[b], Bart Postma[c,1], Marco Roveri[a]

[a] *Fondazione Bruno Kessler, Via Sommarive 18, Povo, 38123 Trento, Italy*
[b] *RWTH Aachen University, Software Modeling and Verification Group, 52056 Aachen, Germany*
[c] *European Space Agency, Postbus 299, 2200 AG Noordwijk, The Netherlands*
[d] *Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece*

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 labor-intensive as they rely on manual analysis, review and inspection. For future space missions, we developed - with challenging requirements from the European space industry - a novel modeling language and toolset for a (semi-)automated validation approach. Our modeling language is a dialect of AADL and enables engineers to express the system, the software, and their reliability aspects. The COMPASS toolset utilizes state-of-the-art model checking techniques, both qualitative and probabilistic, for the analysis of requirements related to functional correctness, safety, dependability and performance. Several pilot projects have been performed by industry, with two of them having focused on the system-level of a satellite platform in development. Our efforts resulted in a significant advancement of validating spacecraft designs from several perspectives, using a single integrated system model. The associated technology readiness level increased from level 1 (basic concepts and ideas) to early level 4 (laboratory-tested).

Keywords: safety & dependability analysis, performance analysis, model checking, AADL modeling language, spacecraft

1. Introduction

A spacecraft is a machine that fulfills mission objectives outside of Earth. Spacecraft design involves a vast body of natural sciences and many engineering fields, including, but not completely covering, materials, optics, power, propulsion, performance, reliability and security engineering. These disciplines are integrated into an interdisciplinary field known as *systems engineering* that addresses the design of systems and the management of complex engineering projects over their life-cycle. Space systems engineering [1] is an evolving field and its current state of practice is strongly influenced by a relatively new engineering discipline, namely that of software development.

Spacecraft in the early space age included software whose size was at most a few dozens of lines of code. The advent of digital interfaces of parts and equipment, and the flexibility of software-based control over analogue interfaces and electrical/mechanical control led to an exponential growth of the size of the deployed software [2]. Nowadays, the latter is compiled from millions lines of code. Hence, software dictates the overall spacecraft behavior to an ever-increasing degree. This is also reflected within the space systems engineering life-cycle. More emphasis is now given to the system-software perspective that encompasses the interaction between the software and the remainder of the system, typically perceived by the software engineers as hardware.

The COMPASS project [3] 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 labor-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 modeling 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 artifacts, 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 COM-

*Corresponding author
Email address: vietyen.nguyen@iese.fraunhofer.de (Viet Yen Nguyen)
[1] Present affiliation: Sogeti Netherlands
[2] Present affiliation: Terma A/S, Denmark
[3] Present affiliation: Fraunhofer IESE, Germany

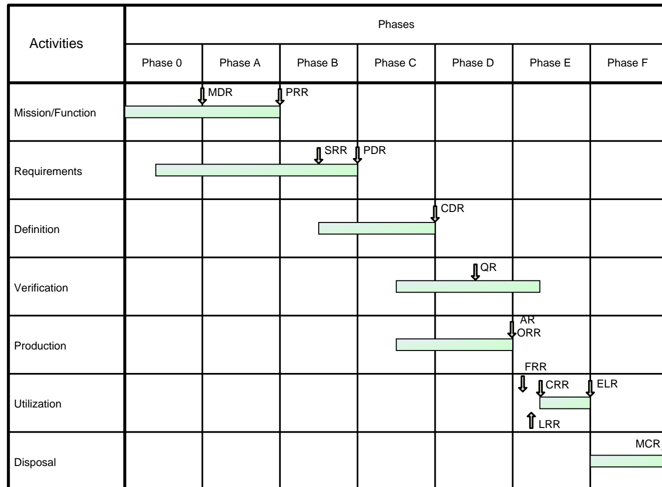


Figure 1: Space systems engineering life-cycle of European missions. Source: ECSS Standard on Project Planning and Implementation [8].

PASS toolset emerged in 2007, due to a series of significant advances in model checking [4], and especially in its probabilistic counterpart [5]. 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 [6]. 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. The novel contribution of this paper is the report on the second pilot project. This paper furthermore summarizes the background work and the first pilot project, whose results were published elsewhere [7]. 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.

This paper is organized as follows. A brief overview of space systems engineering is given in Section 2, which is followed by an introduction to the developed modeling language (Section 3), the toolset (Section 4) and its analyses. The spacecraft platform is described in Section 5, and the pilot projects are presented in Section 6 and 7. The paper wraps up with the related work (Section 8) and the conclusions (Section 9).

2. Space Systems Engineering

The European tradition and practice of spacecraft engineering is codified in the ECSS standards [9] issued by the European Space Agency. The spacecraft system life-cycle is depicted in Figure 1. It starts with mission analysis in phase 0. In phase A, management and engineering plans are set up and functional aspects and feasibility are investigated. During phase B, a preliminary system design is drafted and reviewed. In the phases that follow, the system design is refined to its implementation and the system is verified, launched and operated. In the early phases and most notably in phase B, the system is decomposed into its constituent parts, including the thermal, power, attitude control subsystems and the software. Experts on the respective engineering discipline further refine the subsystems, while system engineers ensure coherency among them.

Increasing Software Functionality. Software plays an increasing and vital role in the overall system that is evident from the exponential growth of the source code sizes in modern spacecraft [2]. Today, software accompanies modern microprocessors in order to provide unprecedented functionality for an ever increasing range of mission demands. For example, navigation systems are equipped with software that delivers strict orbit control for improved precision. Also, on-board software handles enormous mission data sets generated by the high-resolution sensors used in Earth observation satellites. Dedicated software implements functionality that addresses key reliability and autonomy requirements. Especially for deep space missions, where communication windows are short and delays are long, autonomy in terms of survivability is essential to mission success. This is mainly achieved by a *fault management system* [10], the major part of which is realized through software. Supported functions include monitoring, detection, isolation and mitigation of spacecraft faults that may occur due to the harsh space conditions (mechanical stress, wear and radiation). Functional correctness, safety, and dependability of the overall system has to be ensured under the presence of all known space hazards.

Quality Assurance. Verification and validation take place continuously throughout the ECSS life-cycle [11, 12]. Verification examines whether the various requirements (e.g. for a subsystem or an equipment) are met, whereas validation typically focuses on whether higher-level requirements (e.g. mission objectives, system requirements) are met by a constituent system or software. Verification and validation activities are systematically planned based on the drafted requirements and the lessons learned from earlier activities in the engineering life-cycle. The activities' outcomes include justifications, utilities, trade-offs, sizing, feasibility assessments, compliance matrices, source code, mission diagrams and other artifacts. Typical examples are the journals of test suite executions, as well as a radiation analysis from data of similar previous missions. Two analyses are typically conducted for the safety and dependability aspects, namely fault tree analysis (FTA) and failure mode and effects analysis (FMEA). FTA can be visualized as a tree that describes how lower-level failures, or their combination by Boolean AND/OR gates, result in a top-level event, e.g. a system failure. FMEA tables describe the (observable) effect of failure occurrences on the system. Fault trees and FMEA tables are essential input in the development of a fault management system, which may be sub-

ject to failures as well, and thus deserves further analysis to ensure its correctness, safety and dependability. Additionally, probabilistic risk assessments are conducted by computing, from the failure rates of lower-level failures, the rates or probabilities of higher-level failures. This can be done using the fault tree as a behavioral structure, or alternatively by a dedicated reliability model such as a Markov chain. Both are typically crafted manually and this involves a labor-intensive activity that requires a broad and deep understanding of the overall spacecraft behavior under degraded conditions. In any phase, verification and validation enhance the understanding of the system under development and eventually attain informed decisions for the subsequent phases. Validation outcomes may justify design alterations, or the need for additional requirements. Also, they can be used to revise or refine budgets, schedules and resource allocation (e.g. manpower, test laboratories), since critical behaviors are better understood and their correctness can be verified in subsequent phases. Verification outcomes can be used to fine-tune spacecraft operation scenarios and manuals. The overall result of continuous verification and validation is a reduction of the technical and program risk and a process improvement in the engineering life-cycle.

Engineering Challenges. The development, verification and validation activities are supported by a plethora of methods and tools. Models are analysis artifacts representing (parts of) the system behavior, but they cannot replace the system to be deployed. They can be either physical (e.g. structural models, material models), virtual (e.g. software models, thermal models) or a combination of both (e.g hardware-in-the-loop simulation). Models typically abstract from detail and emphasize a particular aspect, typically one that pragmatically supports particular design or verification/validation activities. For example, reliability block diagrams and fault trees focus heavily on failure events, thereby abstracting from the nominal behavior. Matlab/Simulink models are often used to model and simulate continuous processes, like temperature, power and propulsion control and omit architectural aspects. Likewise, for system software and especially fault management, dedicated modeling approaches and formalisms exist, like for example UML or variants thereof. These approaches by themselves do not highlight the software's interoperation with the remainder of the system, especially the software's impact on the overall system safety and dependability aspects and vice versa. Therefore, an integrated and coherent view is required. In this article, we address this need with a methodology that captures both the system, the software and their erroneous behavior in isolation, yet providing a technique to automatically merge them into an overall system-software model covering nominal and degraded operations. On this model, we defined automated analysis methods that support common-practice validation objectives in industry. Our methods have been implemented in a formal methods toolset called the COMPASS toolset [13].

3. The AADL Dialect

The Architecture Analysis and Design Language (AADL) [14, 15] is an industry standard for modeling 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 [14] by supporting the following essential features:

- Modeling both the system's *nominal and faulty behavior*. 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).
- Modeling (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 behavior*. In particular, to analyze continuous physical systems such as mechanics and hydraulics, our modeling language supports continuous real-valued variables with (linear) time-dependent dynamics.
- Modeling *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 behavior, a description of the error behavior and a fault injection specification that describes how the error behavior influences the nominal behavior. These three parts are discussed below. Due to space constraints, we refer the interested reader to [13] for a description of the formal semantics.

3.1. Nominal Behavior

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 [16] is shown in Figure 2.

The component type describes the ports through which the component communicates. For example, the type interface of Figure 2 features three ports, namely an outgoing event port

```
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;
```

Figure 2: 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;
```

Figure 3: The complete power system.

`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 behavior via modes, the transitions between them, which are spontaneous or triggered by events arriving at the ports, and the timing and hybrid behavior of the component. For example, the implementation of Figure 2 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 behavior (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 [17]. 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 **`when`** g **`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 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 Figure 3, 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

```
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;
```

Figure 4: Specification of the monitor.

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 Figure 4; it employs another modeling 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.

3.2. Error Behavior

Error models are an extension to the specification of nominal models [18] 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 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 5 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 signaled 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 behavior is attached. It causes a transition to the resetting state, from which the battery recovers with a probability of $\frac{1}{5}$, and returns to the dead state otherwise.

```
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;
```

Figure 5: Specification of the Battery error model.

3.3. 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 behavior 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 Figure 5), an outgoing data port or subcomponent d in the nominal model (such as voltage in Figure 2), 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 [13].

4. 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 6 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* [19, 20], 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 [21, 22] (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 [4]. MRMC [23, 24] (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 [5], a probabilistic real-time version of CTL). SigRef [25] is used to minimize, amongst others, Interactive Markov Chains (IMC) [26] 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 in shown in Figure 7.

The tool also supports a graphical notation of our AADL dialect, that is a derivation of the AADL graphical notation [18]. We developed a graphical drawing editor enabling engineers to construct models visually using the adopted graphical notation. The editor is called the COMPASS Graphical Modeler and is part of the COMPASS toolset.

4.1. 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 [4], an automated technique that verifies whether a property expressed in temporal logic, holds for a given model. Symbolic techniques [27–29] 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 [30, 31]. 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 Figure 7d. Finally, it is possible to run *deadlock checking*, in order to pinpoint deadlocks (i.e., states with no outgoing transitions) in the model.

4.2. Safety Assessment

COMPASS implements model-based safety assessment techniques, based on symbolic model checking [32–35], and supports traditional techniques such as *Failure Mode and Effects Analysis* (FMEA) [36] and *Fault Tree Analysis* (FTA) [37]. 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 summarized 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 [35] (MCSs), where each MCS is a conjunction of basic faults. COMPASS also supports the generation of dynamic fault trees [38], where ordering constraints between basic faults are represented using priority AND (PAND) gates. Figure 7c depicts a simple fault tree for the power system model of Section 3, 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.

4.3. 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 Figure 3).

Diagnosability analysis investigates the possibility for an ideal diagnosis system to infer accurate and sufficient run-time information on the behavior of the observed system. The COMPASS toolset follows the approach described in [39], 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 [40].

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,

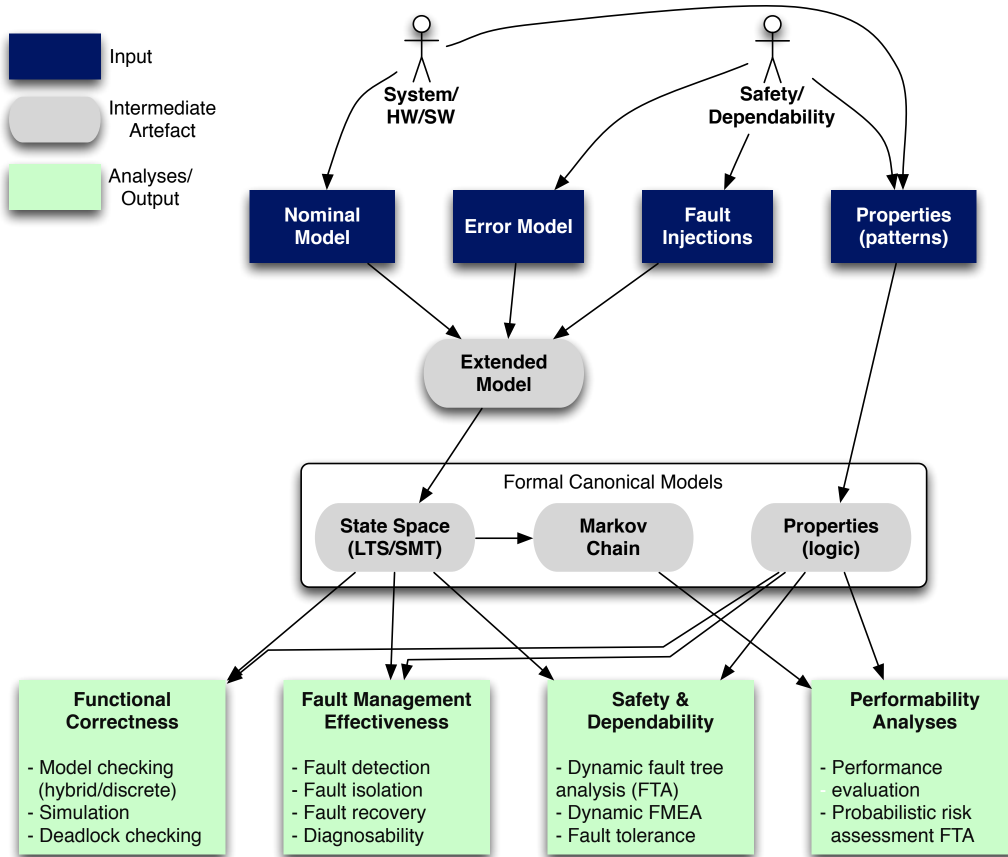


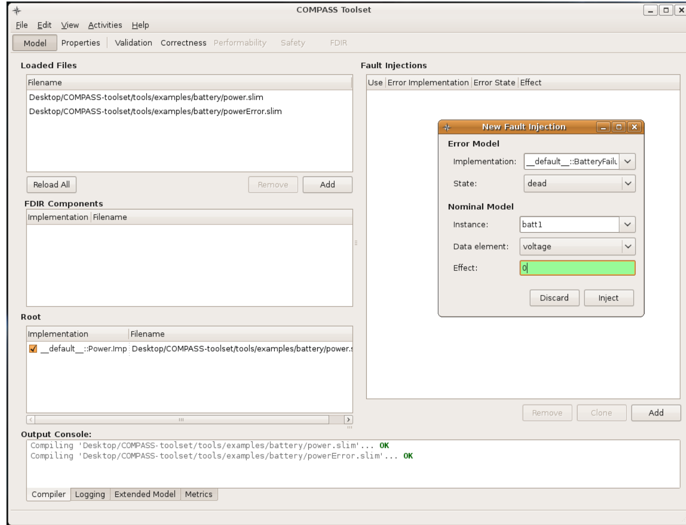
Figure 6: Functional view of the COMPASS Platform.

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 behavior is taken into account, as a battery may die.

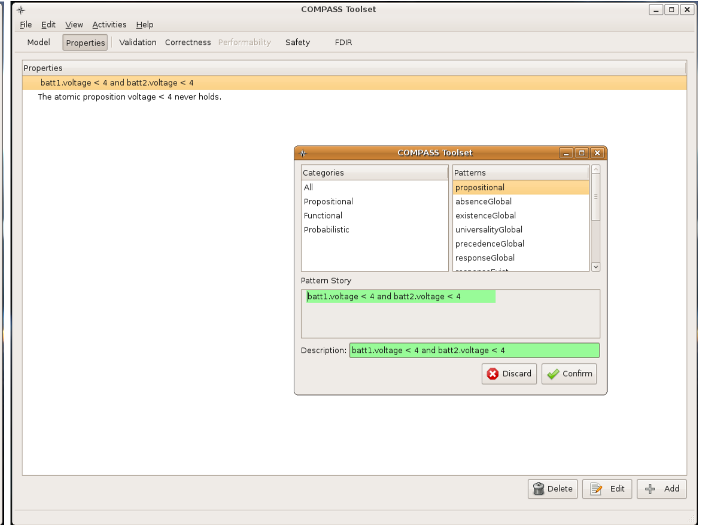
4.4. Performability Analysis

We use probabilistic model checking techniques [4, Ch. 10] for analyzing a model on its performance. The COMPASS toolset in particular supports performance properties expressed in the probabilistic pattern system by [20]. 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 hours" or "the probability that both batteries die within the mission duration". These properties have a direct mapping to Continuous Stochastic Logic (CSL) [5] and are input to the underlying probabilistic model checker.

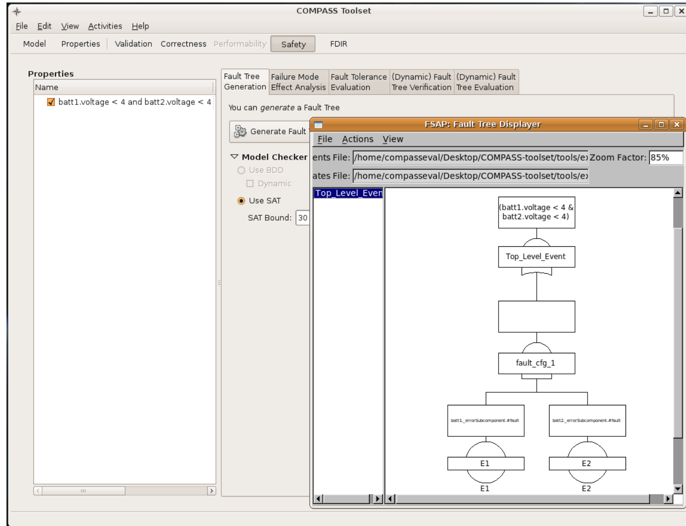
The probabilistic model checker furthermore requires a Markov model as input. This is obtained from the extended model through several steps. First, the extended model's reachable state space is generated through an exhaustive symbolic exploration. Second, the probabilistic rates as specified in the error models (cf. Section 3.2) 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 [26]. This IMC is passed through the third phase, in which its size is reduced using weak bisimulation minimization [41, 42]. 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 [43].



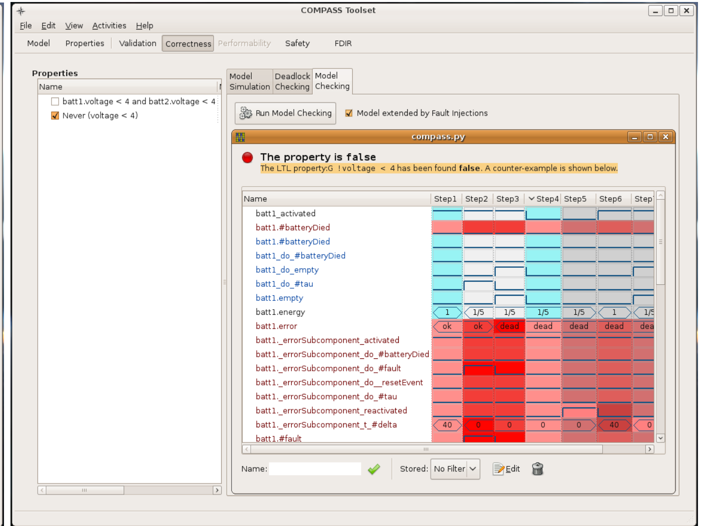
(a) Adding a fault injection.



(b) Adding a property.



(c) A generated fault tree.



(d) A counterexample from model checking.

Figure 7: Walkthrough of the COMPASS toolset using the battery model (cf. Section 3) as an example.

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 [38].

5. The Satellite Platform

Evaluations of the COMPASS toolset have been conducted during its development on subsystem case studies. More about this is reported Section 8. Afterwards, we wanted to scale up and see how well the technology handles a system-level design. At the European Space Agency, we initiated a pilot where we modeled the full satellite platform in development during its early design stage. This pilot was received well, resulting in a successor pilot project where we modeled the same satellite platform in a later design stage. These pilot projects are subsequently referred to as respectively the phase B and phase C pilot projects. This section will elaborate the satellite itself.

5.1. Platform, Payload and Subsystems

At the highest conceptual level, the satellite is composed of the payload and the platform. The *payload* comprises mission-specific subsystems and the *platform* contains all subsystems needed to keep the satellite orbiting in space. The payload is usually designed and tailored from scratch with the mission needs (e.g., weather or telecommunication satellite), whereas for the platform lots of design heritage applies. For this reason, our pilot project focuses on the platform, as this might benefit future projects too.

The platform is comprised of several subsystems. It contains for example the control & data unit (CDU) which can be viewed as the main computer. It furthermore comes with an electric power system (EPS), which is typically a combination of solar arrays and batteries. The attitude & orbit control system (AOCS) is equipped with several kinds of sensors, like Sun sensors, Earth sensors and star trackers, as well as actuators, like magnetic

torque rods, thrusters (OCS) and reactionwheels for orienting and positioning the satellite. There is also a radio system called the telemetry, tracking & control (TT&C). It is the interface to ground control stations located on Earth.

5.2. Fault Management

The majority of these subsystems are designed with certain degrees of fault tolerance, depending on the criticality of the subsystem. Hot and cold redundancies with reconfigurations, voting algorithms, correcting codes and compensation procedures are part of comprehensive strategies for achieving fault tolerance. In the extreme case, the satellite should survive a certain number of days without ground intervention assuming no additional failure occurs. As faults could occur at any level in the system's hierarchy (system, subsystem, equipment), the fault management system obeys a cross-cutting design according to the FDIR paradigm. This paradigm separates fault management into three functions. The function of *fault detection* continuously monitors the system and in case of anomalous values, emits appropriate events to react upon them. Monitoring is decentralized and performed at all levels of the system's hierarchy. After emitting fault detection events, *fault isolation* kicks in. This function is responsible for identifying the affected system's scope by determining the cause of the fault events. The function of *fault recovery* then takes appropriate actions to mitigate the fault events, and if possible, restores the subject to a nominal state. As faults can occur at all levels, and as their effects can propagate throughout the system horizontally (same level) and vertically (across levels), the fault management system is partitioned into five levels depending on the complexity of the respective FDIR functions:

Level 0 Failures are associated to a single unit and recovery can be performed by the unit itself.

Level 1 Failures are associated to a single subsystem, and an external subsystem. The on-board software is responsible for their mitigation.

Level 2 Failures are associated to multiple subsystems, and an external subsystem. The on-board software is responsible for their mitigation.

Level 3 Failures are occurring in the on-board software or in the processor modules. Dedicated reconfiguration modules are responsible for their mitigation.

Level 4 Failures that are not covered by lower levels and that are completely managed by hardware.

Failures are mitigated at their appropriate level. As with most Earth orbiting satellites, this satellite is required to be single-fault tolerant. If a fault is detected, all fault monitoring is disabled and the isolation and recovery of the detected fault is prioritized. As such, it is designed to handle one fault at a time.

6. Phase B Pilot Project

The first pilot project was conducted in parallel to the satellite's early design, which is phase B in Figure 1. The total duration of the pilot was six months including the learning curve of becoming acquainted with AADL, COMPASS and the space systems engineering domain. The modeler was Master-level educated with basic knowledge of model checking but without prior knowledge of the COMPASS toolset nor of space systems engineering. Due to the confidential nature of the case, the model is not publicly available.

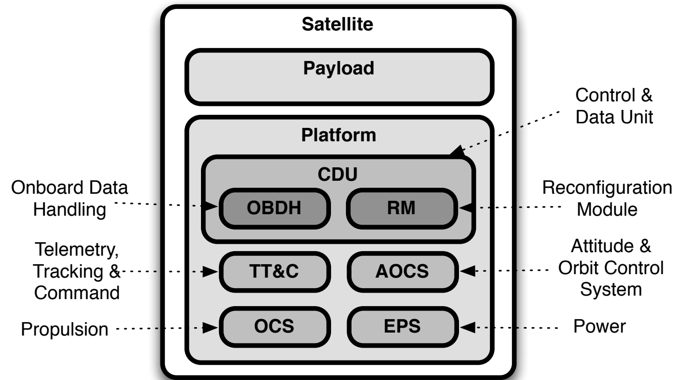


Figure 8: Decomposition of the satellite at phase B.

6.1. Objectives

We started our pilot at the preliminary design review stage (PDR) of the satellite project, where the details of design started to mature. In the spacecraft engineering lifecycle, several objectives have to be met in order to proceed to the critical design stage. A selection of these objectives are of interest to us. Foremost, we wanted to check the compliance of the preliminary design with the functional and operational requirements and their justifications. Second, we wanted to check this too for the reliability, availability, maintainability and safety (RAMS) recommendations. And third, we paid particular attention to the consistency of the hardware/software redundancies and fault management concepts. For the scope of this pilot, we investigated the suitability of our toolset to support these verification and validation objectives. We aimed to develop best modeling practices, discover strengths and weaknesses of the theory and techniques underlying the toolset and understand how the toolset supplements and/or replaces existing practices in the spacecraft design phase.

The satellite's development team was on a strict schedule, and hence it would have been unwise to inject novel development approaches – like our initiative – into the production process. Instead, *we ran our pilot project in parallel with the actual development as an experimental side-track.* This benefited us, because we were not presented with fully crystallized and matured design documentations, but with volatile design information that was undergoing improvement and refinement. Findings in our pilot were therefore directly relevant and could be provided as feedback to the satellite development team. Furthermore, we had to learn to cope with the constant influx of updated design details and how to continually adapt our own model to that. If our pilot had run after the system's design, the effort would have become an afterthought in which design information has fully crystallized and matured.

6.2. Modeling Phase B

Behavior-wise, the overall composite system is described by two important modes of satellite operation: *nominal* and *safe*. While being in the nominal mode, the system is functioning in nominal conditions. Upon faults, recoveries are initiated for resuming nominal operation. If these recoveries fail, the satellite then transits into safe mode for which the system reconfigures itself for survival until ground can perform an intervention. This important transition has system-level effects and hence it is critical.

During modeling, we focused on one subsystem/equipment at a time as the design of each of them corresponds more or less to a specific (section of a) design and a requirements document. We progressively increased coverage by adding more detailed subsystems to the overall model, while keeping high-level abstractions or stubs for the remaining subsystems. The result is a model containing 86 components, 937 ports, 244 modes, 20 error models, 16 recovery procedures, 3831 lines without comments and a state space of the nominal behavior counting 48,421,100 states. In the following paragraphs, we highlight selected lessons from our modeling effort.

Discretization. Various design aspects are often specified in terms of ranges. Thus, for the Sun sensors, ranges are used in degrees of Sun ray impact to determine exposure to the Sun. To avoid a combinatorial explosion of the state space, these ranges have been abstracted with respect to the desired functionality, e.g., a Boolean indicating Sun exposure (or not). Enumerations are used when there are gradations within the ranges.

Errors and Fault Injections. Our primary source for error modeling is the preliminary Failure Mode and Effects Analysis (FMEA). It lists the possible detectable failures as an event and relates it to the effect on the system. This mapping is nearly equal to the fault injections. It also provides the information for constructing the error models. We found that in all cases, the probabilistic behavior was either shaped as a single step from an error-free state to an error state (so-called permanent errors), or that they follow a fault-repair loop-structure on the error-free/error states (so-called transient errors). The FMEA also lists the failure rates. They are expressed in failures in time, indicating the expected number of failures in 10^9 hours.

Assumptions. At first sight, the amount of design information is so overwhelming that it is inconceivable to comprehend the whole system at once, especially if one is not familiar with the system under development. Information might be perceived as incomplete, unclear, or wrong due to this, and this delays the modeling process. We developed a practice of quickly continuing modeling using assumptive modeling decisions. Model elements were marked as *Abstracted*, along with a justification in case the abstraction was not obvious. The annotation *Assumption* was used to express an assumptive understanding of the peculiar design. Model elements were annotated with *Underspecified* when parts of the design documents did not provide sufficient information to have them reflected in the model. Assumptive and underspecified parts could be checked later in review meetings, where we had the opportunity to address them.

6.3. Requirements

Not all requirements at our disposal were amenable to formal analysis. There are several reasons for this fact.

Higher-level requirements, like those from the mission definition, lack the detail needed for formal validation. This can be seen in their form, which is typically prose-like, e.g., "FDIR functions must be active in all AOCS modes". We focus on the more refined system- and subsystem level requirements instead.

A significant part of the requirements do not capture behaviors, but reflect the system's organization. They for example state which components should be present and which functions should be available, but do not detail how the functions should behave. These requirements are typically subject to refinement in detailed design, and hence are cases of intended underspecification.

Requirements could also be out of scope for our objectives. For example, those about the payload were left out intentionally from our pilot because we focused on the satellite platform. Behaviors covering these requirements are typically abstracted in the model.

Consequently, from many thousands of requirements, a selection of 24 relevant requirements was made. They were mapped to the specification patterns from [19, 20], which showed that 92% of them could be phrased into eight patterns. These patterns have an underlying temporal logic form, which can be processed by the COMPASS toolset.

6.4. Analyses

Modeling is highly intertwined with analysis, since the output from analysis provides valuable information for possible refinements of the model. The most widely-used analysis method during modeling is model simulation, as inspection of traces is a fast sanity check before running a resource-consuming analysis.

During all analyses, particular sets of fault injections were disabled/enabled depending on the aim. This was necessary for this pilot, as we observed that fault injections lead to a significant increase of the state space (see Figure 9). The figure shows the state space increase as a multiplication of the nominal state space, which is 48 million states. The explanation is that a fault injection basically yields the cross-product of the subsystem to which the error is injected, and the error model. There is no direct correlation between the amount of fault injections and the increase, although there is a relation between the severity of fault injections and the observed increase. Fault injections that have system-level impact (e.g., processor module failures) add more behavior than fault injections with lower-level impact (e.g., Earth sensor failures), as the former affect a larger fragment of the state space. The figure in general indicates the degree of scalability that can be expected with the computing resources typical in 2010.

All analyses were performed on a set of identical computers running 64-bit Linux, each with a 2.1 GHz AMD Opteron CPU and 192 GB of RAM. A more comprehensive report on this can be found in [7].

Functional Verification. We separated this task into two activities: discrete and real-time/hybrid verification. Sixteen properties were verified on the discrete part, taking between 224 to

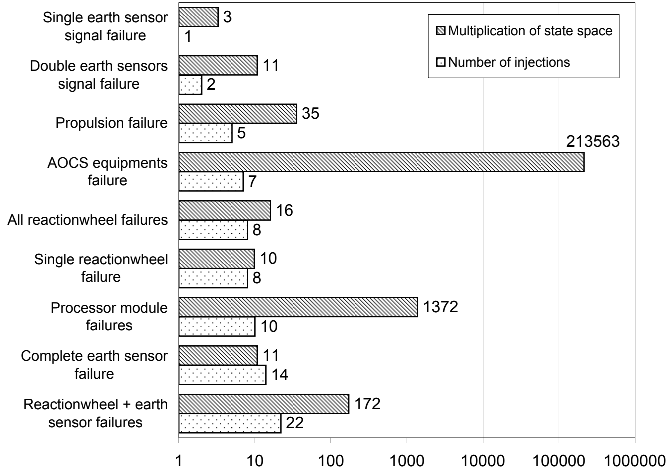


Figure 9: Degrees of state space increase with respect to nominal state space size when injecting failures (logarithmic scale).

677 seconds and on average 126 MB, depending on the injected failures. The real-time/hybrid verification activities are more elaborate and are discussed in more depth in [7].

Safety and Dependability. The platform's most critical event that affects safety and dependability is the transition to safe mode. This transition is triggered upon the occurrence of severe failures. The design documents give a manually developed (static) fault tree of 66 nodes explaining this behavior. Our toolset can produce the same fault tree, but in a *fully automated* manner within two hours and using 239 MB of memory. The algorithm behind this generation is described in [35].

A FMEA table was generated in the same manner as fault trees for associating the sensor failures with three system effects: detection of failures, the setting of the fail-operational flag and the transition to safe mode. The generated table did not provide additional value to the fault tree, as it directly maps failures to the user-provided effects. We learned here that it is more interesting to have the toolset synthesize a mapping from failures to a chain of effects, showing how the first effect directly caused by the failure propagates through the system to subsequent effects and eventually becomes a system-level failure [44].

Fault Management Effectiveness. For fault detection, we checked which observables are triggered when the transition to safe mode is made. This can trigger 129 observables. This takes 19 minutes and 142 MB. Subsequently, fault isolation on these observables takes six hours and 136 MB. Diagnosability analysis was performed to see whether a double Earth sensor failure is diagnosable (for the satellite operator) when the transition to safe mode occurs. Without any result, we had to stop the analysis after seven days and consuming nearly 1400 MB at its peak. The algorithm computing diagnosability has a much higher time-complexity than model checking, thus explaining this result.

Performability. Reliability requirements are usually defined as a cumulative distribution function and state that the foreseen reliability must be at least as good. Their probabilistic nature fits performability analysis. On our model, we wanted to determine the reliability of the satellite in the presence of a sensor failure. Performability analysis however ran out of allocatable memory after nine hours. Here, the weak bisimulation algorithm that transforms the state space to a Markov chain is the bottleneck. Even though the space and time-complexities are polynomial, the state space is simply too large to handle with our current systems.

Another approach to verifying the reliability requirements is by computing the probabilities of the transition-to-safe-mode fault tree which was generated during safety & dependability analysis. This computation occurs in a split second. This approach, and its result, is coarser than the one using performability. Fault trees are essentially abstract state spaces where the relations between the top-level-events and the failures are conservatively over-approximated by AND-, OR- and PAND-gates (Priority AND). With performability on the other hand, these relations are precisely preserved, which however comes with increased complexity when the underlying Markov chain needs to be determined.

6.5. Discussion

With regard to supporting the verification & validation objectives of the preliminary design review, we encountered several inconsistencies in the design documents. Most of them were found during modeling, due to the critical interpretation of the design documents. They were reported to the satellite development manager and were subsequently corrected. This results from the benefit of having a formally coherent modeling language that captures and covers the system, software and safety aspects in a single model.

As with the suitability of the toolset, not being exposed to the underlying logic and model checking tools was found to be pleasant. For safety and dependability analyses, the performance and the features are sufficient. The usefulness of FMEA however can be improved (as stated in Section 6.4). The pilot also showed that theoretical improvements to the algorithms underlying performability and diagnosability are needed to make them tractable for larger models.

The pilot also delivered a deeper insight into best formal modeling practices. We found that assumptive modeling as well as maintaining traceability data are key to keep the pace. We furthermore drafted a set of abstraction guidelines as well as commonly occurring modeling patterns to aid beginning modelers. At the moment, they are tailored to this pilot, but could be further developed to become more generic. These guidelines, and the satellite model itself, can be used to kickstart subsequent formal modeling activities.

7. Phase C Pilot Project

The second pilot was conducted in a time frame of one year, during the Critical Design Review stage (CDR) of the satellite project, and consequently the design details were much more mature than in the previous pilot. The modeler had a PhD-level

education in software engineering and prior experience with using model checkers.

7.1. Objectives

Due to the increase of maturity, the specifications at phase C were roughly twice as detailed as their counterpart descriptions in phase B. Our main objectives are summarized as the following two: firstly, we wanted to investigate strategies for modeling satellites with twice the amount of design detail; and second, we wanted to focus on diagnosability analyses for determining the sufficiency of the sensor configuration. The latter was not possible in the previous pilot, due to the size of the obtained state space.

The main source of information for the satellite system behavior was the Critical Design Review (CDR) documents that refine the documents used for the phase B pilot project. The fault management design in this phase was still undergoing changes. Error modeling was based on CDR's requirements baseline for the FDIR routines, and was subject to changes for the duration of the pilot.

7.2. Modeling Phase C

The developed model focused on the platform of the satellite system, as in the previous pilot and for the same reasons. The organization of components in the model was slightly altered. More specifically, the Control and Data Unit (CDU) was logically modeled as a component embedded in the Attitude and Orbit Control System (AOCS). Physically however, the AOCS runs on a processor module inside the CDU. The latter interfaces with sensors and actuators used by the AOCS through I/O boards. Modeling this organization causes all sensor and actuator interfaces being forwarded by port connections inside the CDU, increasing the model's size. This is avoided by modeling the CDU inside AOCS, such that the sensors and actuators used by the AOCS are its direct subcomponents. Most components were modeled from scratch using the PDR model (from Section 6) as an inspirational reference. Changes and additional details (e.g., Course Earth and Sun Sensors, and Ground commands) of the refined satellite platform design were reflected in the new model.

A bottom-up design approach was followed to model the subsystems of the satellite. Each subsystem was then top-down refined by breaking it down into its equipment. The equipment were verified individually. This approach enables a preliminary analyses of each subsystem model. Also, we could model more detail of the functionality per subsystem, without dealing with the state space explosion of the overall system model. And most important, the satellite fault management design documents at this phase were grouped and organized per subsystem.

The remainder of this section surveys the full satellite model and provides more detail for one of the subsystems, namely the Electric Power System (EPS). Further detail of this model is confined to confidentiality restrictions, as in the previous pilot project.

Scope	Metric	Count
Model	Components	246
	Ports	1,565
	Modes	242
	Error models	173
	Recoveries	162
	Nominal state space	2,341
	LOC (without comments)	6,357
Requirements	Propositional	28
	Absence	7
	Universality	5
	Response	19

Table 1: Metrics of the full satellite phase C platform model and requirements.

7.2.1. Full Platform

The overall system can be in three different modes: a nominal, an intermediate safe and an ultimate safe mode. The purpose of the nominal mode is the same as the one considered in the phase B pilot. The safe modes have a slightly refined meaning. Depending on the mission scenario, a transition to the intermediate safe mode means that ground systems continue supplying commands without platform service warranty. The ultimate safe mode is for handling critical failures while ensuring the satellite's safety. In the remainder of this section the term non-nominal mode refers to the two safe modes.

Table 1 illustrates the size of the developed satellite model in terms of the number of components that comprise it, the lines of code and other aspects. This model resulted in a state space of 2341 states for the nominal behavior, which is many orders of magnitude smaller than the model state space of the previous pilot. The difference is attributed to three main factors, namely the experience gained in the previous pilot, the longer time frame and the given emphasis on model efficiency. However, the model encompassing erroneous behaviors still has a considerable state space size. We designed 173 error models for a series of transient and permanent errors, whose combinations correspond to a pile of fault configurations. From the many requirements, 59 were behavioral and within the scope of our model. The selected requirements were mapped to the specification patterns shown in the lower part of Table 1.

7.2.2. Power Subsystem

For spacecraft systems in general, the Electrical Power Subsystem (EPS) is a major driver towards the overall system reliability [45]. We focus on it in this section. It is responsible for generating, storing, conditioning and the provision of electric power to all on-board satellite units. Its decomposition is shown in Figure 10 and it consists of the following main subunits:

Power Conditioning and Distribution Unit (PCDU) receives the electric power from the solar array and/or battery and distributes it to the satellite subsystems through the Latching Current Limiter (LCL) and the resettable variants of it, called R-LCL.

Battery stores the electric power provided by the solar array during Sun visibility periods and provides power to the satellite

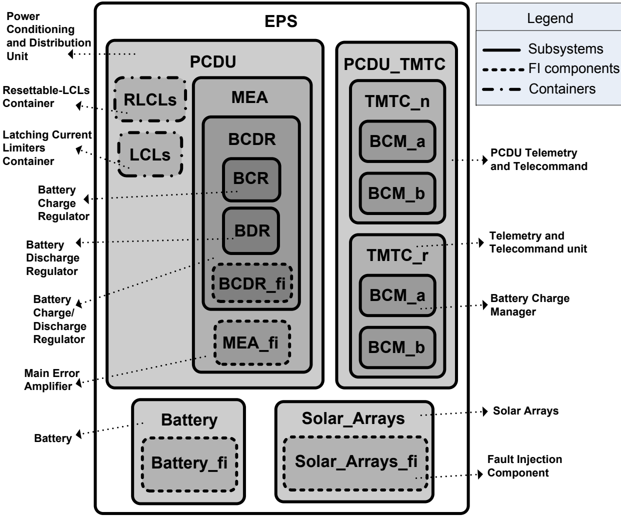


Figure 10: Decomposition of the pilot's satellite Power System at phase C.

subsystems during eclipse phases. It is charged and discharged respectively by the Battery Charge Regulator (BCR) and Battery Discharge Regulator (BDR).

Main Error Amplifier (MEA) stabilizes the power to the default voltage.

Solar Array delivers power to the satellite subsystems through the PCDU. It supplies the satellite during Sun exposure and in parallel charges the battery after the eclipse phases.

PCDU Telemetry and Telecommand (TMTC) controls the PCDU with commands external to the subsystem, e.g., power on and off, charging and discharging from the AOCS and ground control. It is designed to be redundant (i.e., "a" equipment and "b" equipment).

The FI components and the containers in Figure 10 are used respectively for fault injection and packaging of subsystems with common behavior. All these are further discussed in the next section. The size of EPS was close to 15% of the overall size of the full model (i.e., 38 components, 245 ports, 21 modes and 15 errors models).

The electric power subsystem is susceptible to failures at all fault management levels (see also Section 6.2). From the 15 error models, four are handled as level 0 failures, one is handled as a level 1 failure, five are handled as level 2 failures and five are handled as level 3 and 4 failures. Apart from level 0 and level 1 failures, all other failures result in critical system-level effects and hence a transition is made into non-nominal mode.

7.3. Modeling Effectiveness

Key to an effective model is balancing the detail to be taken into account against the resulting complexity. A very detailed model captures more useful behaviors of the satellite, while reduction of the model's complexity is typically achieved by abstracting (parts of) the system's behavior. Apart from the existing modeling strategies developed for the phase B pilot (cf.

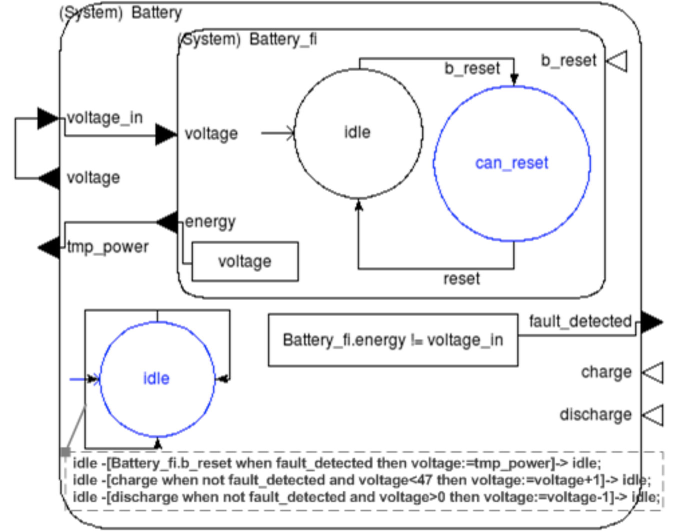


Figure 11: Battery subsystem with fault injections.

Section 6.2), we developed two additional strategies to cope with the increased detail of the phase C design.

The foremost modeling strategy is the use of dedicated fault injection components in the nominal model, in order to separate erroneous data effects from nominal data effects. This strategy came initially from model behavior observations, due to the experienced language limitations (fault injections could only be expressed based on the mode state of a component or based on logical constraints of its supercomponent). More precisely, we observed that upon injecting failures directly on the nominal model, we were actually introducing additional unspecified behavior to the system (e.g., discharging the battery, while the satellite system was powered off).

Figure 11 shows an example with the behavior of the Battery component, using the AADL graphical notation of the COMPASS toolset. It uses rounded boxes to represent system components, circles to represent modes, arrows to represent transitions and (filled/open) triangles to represent (data/event) ports. The battery voltage output is affected by nominal data effects such as charge and discharge if, and only if, a logical constraint is fulfilled (e.g., subsystem is powered on and no fault is detected). These constraints initially limit the state space explosion. Furthermore, the battery voltage output is affected by erroneous data effects if, and only if: (i) it is attached to the `Battery_fi` fault injection subcomponent and (ii) there is a transition due to additional logic constraints (e.g., fault is detected). Extra constraints in model design like the aforementioned one further limit the state space explosion.

The `Battery` component is represented as a container responsible for the fault injection interface, while at the same time detecting faults using the fault detected observable. In case of an injected error on the energy output port of the `Battery_fi` subcomponent (e.g., "`Battery_fi.energy := 25`"), the following operations are executed:

1. `Battery_fi.energy` takes a new value, different from the one specified by `Battery.voltage` and updates

the Battery.tmp_power. At this point, the Battery.voltage is still unaffected, since it is updated through a transition from Battery.tmp_power;
2. the Battery.fault_detected observable becomes true and the system executes only error behavior operations;
3. the Battery_fi.b_reset event is then triggered and simultaneously the Battery_fi changes from the initial idle mode to the can_reset mode, while Battery.voltage is updated with the injected faulty value;
4. if the injected error is a transient one, then a reset event is triggered and Battery_fi changes from the can_reset mode to the idle one. Otherwise, Battery_fi mode stays in can_reset, and no further fault injection is allowed (permanent error);
5. the Battery.fault_detected observable turns to false and the system executes only nominal behavior operations.

In Figure 11, the b_reset event updates the values of the shown component and consequently resets the value of the Battery.fault_detected observable to false, while the **reset** event resets the state of the fault injection component to its initial idle mode.

The second modeling strategy adopted in this pilot is packaging functionality in *containers*. A rule of thumb towards this direction is that the time required for analysis increases along with the number of components. By packaging coupled functionality in a supercomponent, the state space can be reduced when specifying additional erroneous behavior at the supercomponent level. Consider for example what is shown in Figure 10. The Latching Current Limiters (LCLs) and the Resettable LCLs are packaged in a single container respectively, instead of separate containers for each subcomponent.

We also realized that the number of transition events in the model increased the analysis time significantly. It was possible to avoid this cost by refactoring these events into flows and data ports. However, since the discussed performance degradation was a result of the way the COMPASS toolset transforms AADL events into NuSMV language constructs, this model improvement cannot be considered as a general modeling strategy.

The model's efficiency was determined by computing the reachable states. These computations took place on a workstation with two 2.7 GHz Intel Xeon processors and 48 GB RAM running 64-bit Linux. The results are shown in Table 2. The diameter is the depth of the state space given as number of transitions. Noteworthy here is the difference in execution time between the scenarios without fault injections and those with one permanent fault injection in the solar arrays. The latter has a system-level impact and opens up new behaviors compared to the nominal scenario. However, as BDD methods are used to compute the reachable states, the needed time depends heavily on BDD variable reordering strategies, which may turn out better for particular models.

7.4. Diagnosability Efficiency

In the Phase B pilot, diagnosability analysis was deemed intractable as it needed more computing resources than we had available. In this pilot, we investigated methods to overcome this by a more efficient model.

Besides the general improvement of the model's efficiency, we investigated the idea of local diagnosability analysis. Instead of considering the whole satellite platform, we exploited the system hierarchy and the containers to create smaller subsystem models, on which we conducted diagnosability analysis. Then either permanent or transient errors were injected in those models and it was checked whether they could be diagnosed. The results of these analyses are shown in Table 3. The column "Model" is either full, EPS, or AOCS, meaning respectively the full satellite platform, or only the electric power subsystem or only the attitude and orbit control subsystem.

The table shows that, in general, failure scenarios considering permanent errors require less analysis time and less memory than scenarios with transient errors. This is also visible when the model's state spaces of respectively transient and permanent errors are compared (cf. Table 2).

In two cases we still run out of memory (indicated by o.o.m. in the memory column), namely when double permanent errors are injected in the Earth sensors and a single transient error in the main error amplifier (MEA). Both are on the full satellite platform model. In those two cases no conclusion can be drawn regarding the diagnosability of the failure.

One scenario is undiagnosable. The cause for this is the notion of delayed diagnosis. The double permanent errors cause the diagnosis property, the setting of the fail-operational flag, to change. Yet the observables are not directly changed along, but with a delay. The COMPASS toolset detects this as an undiagnosable discrepancy, as after setting the fail-operational flag to true, the observables are the same as when the flag was false. The satellite design however specifies that the fail-operational flag is set within a bounded *delay*, and this is reflected in the model as well. The currently used diagnosability analysis algorithm in COMPASS cannot measure this delay and assumes a diagnosable model must have a zero-delay between the change of diagnosis properties and the change of observables. To this end, the COMPASS toolset will be enhanced in the future, in order to correctly determine diagnosability for non-zero diagnosis delays.

7.5. Discussion

The longer time frame for this pilot compared to its predecessor phase B pilot, as well as the fact of having employed a higher qualified modeler, led to a satellite platform model that captures more detail, yet keeping it manageable for analyses. The result is the largest and most detailed system-level formal model of a satellite platform known to date, that can be used as a reference for future formal modeling initiatives. In addition to existing identified modeling strategies (see Section 6.2), two more strategies were added to further avoid the state space explosion. Additionally, special focus was put on diagnosability analysis, which in the phase B pilot was deemed untractable. Diagnosability analysis becomes increasingly important in the engineering life-cycle as fault management designs become more involved to meet mission demands. We experimented with local diagnosability analyses and with different failure scenarios.

Scenario	Diameter	Reachable States	Time (hours)	Memory (GB)
No fault injections	20	2341	6.0	1.2
Permanent fault in solar array	84	403,220,000	1.5	3.6
Transient fault in solar array	142	135,895,000,000	180.0	14.0

Table 2: Reachable states on full satellite model.

Failure scenario	Model	Diagnosis property	Outcome	Time (sec)	Memory (MB)
Single transient in MEA	Full	Fault detected	n.a.	565383	o.o.m.
Single permanent in battery	Full	Fault detected	Diagnosable	383891	36905
Single permanent in battery	EPS	Fault detected	Diagnosable	259	76
Single transient in battery	EPS	Fault detected	Diagnosable	577	88
Double permanent in Earth sensors	Full	Fail-operational flag is set	n.a.	515836	o.o.m.
Double permanent in Earth sensors	AOCS	Fail-operational flag is set	Undiagnosable	164049	28775
Single transient in solar array	EPS	Fault detected	Diagnosable	775	93

Table 3: Diagnosability analysis report.

The outcome resulted in a clear need for an enhanced diagnosality analysis algorithm that also accounts for delayed diagnostic means.

8. Related Work

Several works have been reported in the literature that focus on modeling languages, formal semantics, analysis algorithms and tools similar to the COMPASS approach. The most closely related results are discussed in the following paragraphs.

High-Level Specification Languages. Several research works focus on high-level modeling languages like the Architecture Analysis and Design Language (AADL). The Unified Modeling Language (UML) [46] is found in many applications. A survey by [47] overviews its use for safety and dependability analysis. It also has been customized to fit the system engineering domain, leadding to a language called SysML [48]. Unfortunately, SysML inherits the weaknesses of UML, with the most serious one for our setting being the loose formalism for nominal and error behavior modeling. In [49], the authors introduce a toolchain in order to model system software in a component-oriented manner, using the Focus language. Focus models can be formally verified based on a translation to Isabelle/HOL and are the basis for the generation of C code implementing their behaviour. The COMPASS toolset does not provide similar functionality, but on the other hand, adds a safety and dependability dimension to the modeling and analysis of nominal system software.

Our AADL dialect has some similarities with existing approaches for the specification of component-based systems, such as interaction systems [50] and constraint automata [51], as well as formalisms for composing automata, such as interface automata [52] and hybrid I/O automata [53]. A recent work by [54] focused on AADL aiming to map the time-constrained event-behavior of AADL's mode transitions to timed Petri nets. The authors also mention the possibility to use colored Petri nets, in order to handle data-dependent constraints. More recently, in [55], the authors provide a denotational semantics for a subset of AADL and its Behavior Annex. However, none of the aforementioned works offers the necessary expressiveness for modeling reconfiguration of components and port connections and for explicitly representing data elements with hybrid aspects, multi-way communication and probabilistic error behavior.

Formal Semantics. Apart from our AADL dialect, an approach towards defining a formal semantics for AADL that is worth mentioning is the one reported in [56]. It introduces an operational semantics of a subset of AADL by giving a translation into the BIP component framework, thus realizing a possibility to perform timed analyses. In [57], the Arcade framework is presented that allows probabilistic analysis of the modeled architectures. Another framework for dependability analysis of AADL models is described in [58]. Based on a translation of AADL into generalized stochastic Petri nets (GSPNs), the authors present transient and steady-state analysis of AADL models with error behaviour. However, none of the aforementined works combines timed or hybrid extensions and probabilistic or safety analyses (FDIR, FMEA, FTA) into a single framework.

A related area of active research interest is the incorporation of probabilistic information into various formalisms. In [59], a probabilistic extension of the Statecharts semantics is proposed. The approach supports different semantic models, such as stochastic Petri nets and probabilistic automata. In [60], the author introduces a probabilistic extension of the component-based modeling formalism Reo [61], which in fact targets only software components and does not account for hardware/software co-design aspects. Finally, in [62] the authors present an object-oriented approach towards dependable co-design based on UML and the Parallel Object-Oriented Specification Language.

Tools. COMPASS is not the only existing AADL based toolset. The TOPCASED project [63] provides a toolchain that translates AADL and its Behavior Annex into Fiacre, and from Fiacre into timed Petri nets, which can be then model checked using the

TINA toolbox. There is also another backend that translates Fiacre into Lotus NT, the process algebraic language that can be analyzed within the CADP toolbox [64]. Both Fiacre backends cover the AADL Behavior Annex and support the analysis of real-time properties, but as opposed to COMPASS, they do not integrate the specification and analysis of erroneous behavior. In [65], the authors present an ontology-based transformation of AADL models into the Altarica formal language. Their approach utilizes the ontology languages built-in reasoning capabilities to bridge the semantic gap between AADL and Altarica. This allows to detect lack of model elements and semantically inconsistent parts of the system design, but timed or hybrid extensions and probabilistic analyses remain out of scope. Other worth to mention AADL based tools are ADeS [66] for simulation of system architectures, as well as Cheddar [67] and the Furness toolset [68] for schedulability analysis.

Applications. In the past, other case studies have been conducted within the space domain. We highlight a few of them that bear similarity with the projects described in this paper.

In [69], a formal approach is described for the specification of an Attitude and Orbit Control System (AOCS). Key in that approach is that correctness properties are verified with each refinement of the specification. The COMPASS approach allows for incremental specification by deepening the system hierarchy, but verifies correctness properties on a whole system architecture at once. In our pilot projects, we cover the full satellite platform, which includes the Attitude and Orbit Control System.

In [70], the architectures of a spacecraft and a space-based network are analyzed with respect to their reliability under degraded modes of operation, also referred to as their survivability. It employs stochastic Petri-nets as their fundamental model. In contrast to their work, we use a high-level modeling language for expressing the case, which we believe is more engineer-friendly. Also, we validate properties beyond survivability, such as correctness and performance aspects.

In [71], NASA studied the use of formal analysis techniques for finding bugs in the Martian Rover software. They focus on code verification of spacecraft software implementations. In our projects we focus on the spacecraft system software design, and analyze it from a correctness, safety & dependability and performance perspective.

In [72], the validation of a satellite's design is supported by simulation of several linked models comprising parts of the spacecraft and its Failure Detection, Isolation and Recovery concept. The result is an integrated simulatable model of high-fidelity upon which functional correctness properties can be validated. Our pilot project does not reuse and link existing models, and the fidelity of our model is comparatively lower in order to mitigate the state space explosion problem. However, we validate properties beyond functional correctness, covering the safety, dependability and performance aspects.

The COMPASS toolset was also evaluated by Thales Alenia Space, on two case studies of their satellite subsystems [6]. The first case study relates to the definition of *satellite mode management* and its associated *fault management strategy*. It models the AOCS (Attitude and Orbit Control System) equipment and other functional subsystems, and the (re-)configuration sequences, representing sequences of commands which are sent to the AOCS units in case of a detected failure. Injected faults include transmission error, electrical default and data inconsistency. The model was simulated, and several properties of the model were verified using FTA and model checking. The second case study models a *thermal subsystem*, consisting of thermal lines, heaters and sensors, that regulates the satellite's temperature by performing both active and passive regulation. The functional model, covering the whole perimeter of the thermal regulation function, was complemented by a more detailed model, taking into account the behavior of the thermal lines. A thermal line consists of heater lines, safety switches and thermistors. Several forms of redundancies are foreseen, to deal with failures. Injected faults include stuck-at-value for sensors, no heating and always heating for a heater, stuck-at-ON and stuck-at-OFF for a switch. Different alternative models were built, simulated and analyzed for correctness, reliability, safety and diagnosability capabilities using model checking, FTA and FMEA, fault detection and fault recovery analysis. Both subsystem case studies sparked the interest in a pilot on a system-level spacecraft, which we performed and reported the results in Sections 6 and 7.

9. Conclusions

In five years, we implemented the latest advances in (probabilistic) model checking into a full-fledged graphical toolset for formal model-based system-level design and rigorous analysis of correctness, safety, dependability and performability properties, and demonstrated its effectiveness for spacecraft design validation in an industrial setting. The pilot projects, described in this article, were performed on satellite system-level and subsystem-level designs of past and ongoing space missions. These projects led to the definition and analysis of extremely large spacecraft system models and resulted in an advancement of validating spacecraft designs for correctness, safety, dependability and performance based on an integrated system model. The associated technology readiness level increased from level 1 (in 2008) to early level 4 (in 2012). Follow-up projects are going on and planned for, and thus further activities build upon the existing cooperation between academia and the space industry. Even though the COMPASS toolset is developed to meet the needs of the space industry, its general focus on safety-critical systems also applies to similar domains such as the automotive, aviation and railway industries. Distribution of the COMPASS toolset is restricted to ESA member states. The literal license, along with manuals, tutorials and presentations are available on our website [3].

Acknowledgements. This work was partially supported by ESA/ESTEC (contract no. 4000100798) and Thales Alenia Space (contract no. 1520014509/01).

References

[1] System Engineering General Requirements, Tech. Rep. ECSS-E-ST-10C, ESA Requirements and Standards Division, Noordwijk, Netherlands (March 2009).

[2] D. L. Dvorak, NASA Study on Flight Software Complexity, Tech. rep., California Institute of Technology (2009).
[3] COMPASS Website, `compass.informatik.rwth-aachen.de`.
[4] C. Baier, J.-P. Katoen, Principles of Model Checking, MIT Press, 2008.
[5] C. Baier, B. Haverkort, H. Hermanns, J.-P. Katoen, Model-Checking Algorithms for Continuous-Time Markov Chains, Transactions on Software Engineering 29 (6) (2003) 524–541.
[6] M. Bozzano, R. Cavada, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, X. Olive, Formal Verification and Validation of AADL Models, in: Embedded Real Time Software and Systems Conference, AAAF & SEE, 2010.
[7] M.-A. Esteve, J.-P. Katoen, V. Y. Nguyen, B. Postma, Y. Yushtein, Formal Correctness, Safety, Dependability and Performance Analysis of a Satellite, in: International Conference on Software Engineering, IEEE, 2012, pp. 1022–1031.
[8] Project Planning and Implementation, Tech. Rep. ECSS-M-ST-10C Rev. 1, ESA Requirements and Standards Division, Noordwijk, Netherlands (March 2009).
[9] European Cooperation for Space Standardization, `ecss.nl`.
[10] NASA Fault Management Handbook, version 1 (draft) Edition, no. NASA-HDBK-1002, NASA, 2011.
[11] Verification, Tech. Rep. ECSS-E-ST-10-02C, ESA Requirements and Standards Division, Noordwijk, Netherlands (March 2009).
[12] Verification Guidelines, Tech. Rep. ECSS-E-HB-10-02A, ESA Requirements and Standards Division, Noordwijk, Netherlands (December 2010).
[13] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, M. Roveri, Safety, Dependability and Performance Analysis of Extended AADL Models, Computer Journal 54 (5) (2011) 754–775.
[14] Architecture, Analysis and Design Language, no. AS5506, Society of Automotive Engineers, 2004.
[15] P. H. Feiler, D. P. Gluch, Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, Addison-Wesley Professional, 2012.
[16] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, M. Roveri, The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems, in: Computer Safety, Reliability, and Security (SAFECOMP), Vol. 5775 of LNCS, Springer, 2009, pp. 173–186.
[17] T. A. Henzinger, The Theory of Hybrid Automata, in: Logic in Computer Science, IEEE, 1996, pp. 278–292.
[18] SAE Architecture Analysis and Design Language Annex Volume 1, no. AS5506/1, Society of Automotive Engineers, 2006.
[19] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, Patterns in Property Specifications for Finite-State Verification, in: International Conference on Software Engineering, ACM, 1999, pp. 411–420.
[20] L. Grunske, Specification Patterns for Probabilistic Quality Properties, in: International Conference on Software Engineering, ACM, 2008, pp. 31–40.
[21] NuSMV Model Checker. `nusmv.fbk.eu`.
[22] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV 2: An OpenSource Tool for Symbolic Model Checking, in: Computer Aided Verification, Vol. 2404 of LNCS, Springer, 2002, pp. 359–364.
[23] MRMC – The Markov Reward Model Checker, `mrmc-tool.org`.
[24] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, D. N. Jansen, The Ins and Outs of the Probabilistic Model Checker MRMC, Performance Evaluation 68 (2) (2011) 90–104.
[25] R. Wimmer, M. Herbstritt, H. Hermanns, K. Strampp, B. Becker, Sigref - A Symbolic Bisimulation Tool Box, in: Automated Technology for Verification and Analysis, Vol. 4218 of LNCS, Springer, 2006, pp. 477–492.
[26] H. Hermanns, Interactive Markov Chains: The Quest for Quantified Quality, Vol. 2428 of LNCS, Springer, 2002.
[27] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu, Symbolic Model Checking without BDDs, in: Tools and Algorithms for Construction and Analysis of Systems, Vol. 1384 of LNCS, Springer, 1999, pp. 193–207.
[28] A. Biere, K. Heljanko, T. A. Junttila, T. Latvala, V. Schuppan, Linear Encodings of Bounded LTL Model Checking, Logical Methods in Computer Science 2 (5).
[29] K. Heljanko, T. Junttila, T. Latvala, Incremental and Complete Bounded Model Checking for Full PLTL, in: Computer Aided Verification, Vol. 3576, Springer, 2005, pp. 98–111.
[30] MathSAT. `mathsat.fbk.eu`.
[31] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Rossum, S. Schulz, R. Sebastiani, MathSAT: Tight Integration of SAT and Mathematical Decision Procedures, Automated Reasoning 35 (1-3) (2005) 265–293.
[32] ESACS Project. `www.esacs.org`.
[33] ISAAC Project. `isaac-fp6.org`.
[34] FSAP/NuSMV-SA Platform. `sra.fbk.eu/tools/FSAP`.
[35] M. Bozzano, A. Cimatti, F. Tapparo, Symbolic fault tree analysis for reactive systems, in: Automated Technology for Verification and Analysis, Vol. 4762 of LNCS, Springer, 2007, pp. 162–176.
[36] Failure Modes, Effects (and Criticality) Analysis, Tech. Rep. ECSS-Q-ST-30-02C, ESA Requirements and Standards Division, Noordwijk, Netherlands (March 2009).
[37] Fault Tree Analysis, Tech. Rep. ECSS-Q-ST-40-12C, ESA Requirements and Standards Division, Noordwijk, Netherlands (July 2008).
[38] H. Boudali, P. Crouzen, M. Stoelinga, A Rigorous, Compositional and Extensible Framework for Dynamic Fault Tree Analysis, in: Dependable and Secure Computing, IEEE, 2010, pp. 128 – 143.
[39] A. Cimatti, C. Pecheur, R. Cavada, Formal Verification of Diagnosability via Symbolic Model Checking, in: International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 2003, pp. 363–369.
[40] B. Bittner, M. Bozzano, A. Cimatti, X. Olive, Symbolic Synthesis of Observability Requirements for Diagnosability, in: Advanced Space Technologies in Robotics and Automation, European Space Agency, 2011.
[41] S. Derisavi, H. Hermanns, W. H. Sanders, Optimal State-Space Lumping in Markov chains, Information Processing Letters 87 (6) (2003) 309–315.
[42] A. Valmari, G. Franceschinis, Simple O(m log n) Time Markov Chain Lumping, in: Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2010, pp. 38–52.
[43] D. Guck, T. Han, J.-P. Katoen, M. R. Neuhäußer, Quantitative Timed Analysis of Interactive Markov Chains, in: NASA Formal Methods Symposium, Vol. 7226 of LNCS, Springer, 2012, pp. 8–23.
[44] B. Ern, V. Y. Nguyen, T. Noll, Characterization of Failure Effects on AADL Models, in: Computer Safety, Reliability and Security, Vol. 8153 of LNCS, Springer, 2013, pp. 241–252.
[45] S. Y. Kim, J.-F. Castet, J. H. Saleh, Spacecraft Electrical Power Subsystem: Failure Behavior, Reliability, and Multi-state Failure Analyses, Reliability Engineering and System Safety 98 (1) (2012) 55 – 65. doi:http://dx.doi.org/10.1016/j.ress.2011.10.005.
[46] Unified Modeling Language, `uml.org`.
[47] S. Bernardi, J. Merseguer, D. C. Petriu, Dependability Modeling and Analysis of Software Systems Specified with UML, ACM Comput. Surv. 45 (1) (2012) 2:1–2:48. doi:10.1145/2379776.2379778.
URL `http://doi.acm.org/10.1145/2379776.2379778`
[48] OMG Systems Modelling Language, `omgsysml.org`.
[49] F. Hölzl, M. Feilkas, AutoFocus 3: A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems, in: Model-based Engineering of Embedded Real-Time Systems, Vol. 6100, Springer, 2010, pp. 317–322.
[50] G. Gössler, S. Graf, M. Majster-Cederbaum, M. Martens, J. Sifakis, An Approach to Modelling and Verification of Component Based Systems, in: Current Trends in Theory and Practice of Computer Science, Vol. 4362, Springer, 2007, pp. 295–308.
[51] C. Baier, M. Sirjani, F. Arbab, J. Rutten, Modeling Component Connectors in Reo by Constraint Automata, Science of Computer Programming 61 (2) (2006) 75–113.
[52] L. de Alfaro, T. A. Henzinger, Interface Automata, in: European Software Engineering Conference /Foundations of Software Engineering, ACM, 2001, pp. 109–120.
[53] N. Lynch, R. Segala, F. Vaandrager, Hybrid I/O Automata, Information and Computation 185 (1) (2003) 105–157.
[54] X. Renault, F. Kordon, J. Hugues, Adapting Models to Model Checkers, A Case Study: Analysing AADL Using Time or Colored Petri Nets, in: Rapid System Prototyping, IEEE, 2009, pp. 26–33.
[55] S. Björnander, C. Seceleanu, K. Lundqvist, P. Pettersson, ABV - A Verifier for the Architecture Analysis and Design Language, in: International Conference on Engineering of Complex Computer Systems, IEEE, 2011, pp. 355–360.
[56] M. Y. Chkouri, A. Robert, M. Bozga, J. Sifakis, Translating AADL into BIP - Application to the Verification of Real-Time Systems, in: Model-Based Architecting and Construction of Embedded Systems (ACES-MB),

Vol. 503, CEUR, 2008, pp. 5–19.

[57] H. Boudali, P. Crouzen, B. Haverkort, M. Kuntz, M. Stoelinga, Architectural Dependability Evaluation wth Arcade, in: Dependable Systems and Networks, IEEE, 2008, pp. 512–521.

[58] A.-E. Rugina, K. Kanoun, M. Kaâniche, A System Dependability Modeling Framework Using AADL and GSPNs, in: Workshop on Software Architectures for Dependable Systems, Vol. 4615 of LNCS, Springer, 2007, pp. 14–38.

[59] E. Bode, M. Herbstritt, H. Hermanns, S. Johr, T. Peikenkamp, R. Pulungan, R. Wimmer, B. Becker, Compositional Performability Evaluation for STATEMATE, in: Quantitative Evaluation of Systems, IEEE, 2006, pp. 167–178.

[60] C. Baier, Probabilistic Models for Reo Connector Circuits, Journal of Universal Computer Science 11 (10) (2005) 1718–1748.

[61] F. Arbab, Reo: A Channel-Based Coordination Model for Component Composition, Mathematical Structures in Computer Science 14 (3) (2004) 329–366.

[62] B. D. Theelen, O. Florescu, M. C. W. Geilen, J. Huang, P. H. A. van der Putten, J. P. M. Voeten, Software/Hardware Engineering with the Parallel Object-Oriented Specification Language, in: Formal Methods and Models for Codesign, IEEE, 2007, pp. 139–148.

[63] B. Berthomieu, J. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, F. Vernadat, et al., Fiacre: An Intermediate Language for Model Verification in the TOPCASED Environment, in: Embedded Real-Time Software and Systems, 2008.

[64] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes, in: Tools and Algorithms for the Construction and Analysis of Systems, Vol. 6605, Springer, 2011, pp. 372–387.

[65] K. Mokos, G. Meditskos, P. Katsaros, N. Bassiliades, V. Vasiliades, Ontology-Based Model Driven Engineering for Safety Verification, in: Software Engineering and Advanced Applications, IEEE, 2010, pp. 47 –54.

[66] ADeS, a simulator for AADL, `www.axlog.fr/aadl/ades_en.html`.

[67] Cheddar: a free real time scheduling tool, `wiki.sei.cmu.edu/aadl/index.php/Cheddar`.

[68] Furness^{TM} Toolset, `furnesstoolset.com`.

[69] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, T. Latvala, Developing Mode-Rich Satellite Software by Refinement in Event-B, Science of Computer Programming 78 (7).

[70] J.-F. Castet, J. H. Saleh, On the Concept of Survivability, with Application to Spacecraft and Space-based Networks, Reliability Engineering and System Safety 99 (0) (2012) 123 – 138.

[71] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, W. Visser, R. Washington, Experimental Evaluation of Verification and Validation Tools on Martian Rover Software, Formal Methods in System Design 25 (2-3) (2004) 167–198.

[72] A. Rugina, C. Leorato, E. Tremolizzo, Advanced Validation of Overall Spacecraft Behaviour Concept Using a Collaborative Modelling and Simulation Approach, in: Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2012, pp. 262–267.