Applicability of Model Checking for Verifying Spacecraft Operational Designs

Philipp Chrszon*§, Paulina Maurer*§, George Saleip*§, Sascha Müller*§, Philipp M. Fischer*§, Andreas Gerndt*†§, and Michael Felderer*‡§

*Institute for Software Technology
German Aerospace Center (DLR)
Braunschweig, Germany

†University of Bremen
Bremen, Germany

‡University of Cologne
Cologne, Germany

§Email: philipp.chrszon@dlr.de, paulina.maurer@dlr.de, george.nasralla@dlr.de, sa.mueller@dlr.de, philipp.fischer@dlr.de, andreas.gerndt@dlr.de, michael.felderer@dlr.de

***Abstract*—Guaranteeing safety and correctness is one of the main objectives during the development of space systems. This is a challenging task, since many different engineering disciplines are involved in the development and the constituent parts of a spacecraft are highly interconnected and interdependent. Increasingly, formal methods such as model checking are applied to verify safety-critical parts of spacecraft designs and also implementation, since they may prove the absence of design errors. Generally, a major challenge for adopting model checking into the design process is its scalability. Usually, the whole state space of a system, which grows exponentially with, e.g., the number of parallel processes, must be explored.**

In this paper, we consider operational designs of spacecraft as they may occur during early development phases and systematically evaluate the scalability of model checking for verifying such models. For this, we created an arbitrarily scalable operational design describing the mode management of a satellite. Transformations of the models into the modeling languages of different model-checking tools enables a comparative scalability study of various model-checking algorithms. The evaluation shows promising results for symbolic model-checking approaches. A comparatively low analysis time and memory usage suggest that model checking for early operational designs can be incorporated into existing design processes.

***Index Terms*—Aerospace, Formal Models, Formal Methods, Model Checking**

I. Introduction

Space systems are widely considered as safety-critical. Not only are failures expensive, they often jeopardize the mission objectives, and in the worst case may even endanger human lives. However, ensuring that such systems work correctly becomes increasingly difficult. Today's spacecraft are complex autonomous systems controlled by sophisticated on-board computer systems. Furthermore, spacecraft consist of many highly interconnected parts, most of which are developed in collaboration by scientists and engineers from different disciplines. This interdependence makes the correction of design errors in later development phases costly. After launch, correcting hardware faults is seldom possible and fixing software issues often comes with a considerable risk. Therefore, it is highly desirable to find design errors as soon as possible.

In order to detect errors in the design and to ensure the correctness of the system, verification and validation efforts traditionally employ testing and simulation. Here, the system, a model of the system, or a combination of both, is executed and observed for unintended behaviors. As full coverage of all possible executions is never achieved in practice, this can only show the presence of errors. Especially for critical systems or subsystems, simulation can be complemented by formal verification, which can provide proof that the system or system model satisfies certain properties over all possible executions. Thus, it can also show the absence of errors.

Model checking is a formal verification approach where a systematic exploration of a system's state space is performed fully automatically. It is particularly well-suited for analyzing highly concurrent systems and can uncover errors that are caused by very specific orderings of task execution. Such concurrency-related errors are usually hard to detect using simulation alone, as they might occur only rarely and often nondeterministically.

A major challenge for the practical application of model checking is its scalability, since potentially the whole state space of the system must be represented and explored. Many approaches and techniques have been developed to mitigate this issue, which can make the verification of large-scale systems tractable. However, the application of these techniques often requires expert knowledge in formal modeling and verification. This severely hinders the widespread application of model checking in industry, and the aerospace domain is no exception [1], [2].

In this paper, we investigate the applicability of model checking within early spacecraft design phases. In particular, we examine up to which model size and complexity model checking is still tractable and fast enough to be incorporated into the design process. For that, we

The published version of the article is available at https://doi.org/10.1109/MODELS58315.2023.00011.

1) created a representative model for mode management of a spacecraft which can be scaled arbitrarily,
2) implemented transformations of the model into the input languages of various model-checking tools, and
3) systematically examined the analysis durations for detecting deadlocks and livelocks using different model checkers.

For modeling the mode management, we utilize a simple state-machine formalism, where transitions between modes may be triggered using commands and additional mode constraints can be formulated. The goal of our verification approach is that the model-checking process can be integrated transparently into the design process and may be used by non-experts in formal verification. Thus, we only consider automated optimizations and state-space explosion mitigation techniques that are provided by the model-checking tools and exclude hand-crafted optimizations or abstractions. In order to achieve a wide coverage of different state-space explosion mitigation techniques, we selected one or more representative model checkers for each technique, which allows us to compare their effectiveness for our specific application.

II. Related Work

This section gives an overview of spacecraft systems engineering, model checking, and previous works on the formal verification of space systems using model checking.

A. Spacecraft Systems Engineering

The European Cooperation for Space Standardization (ECSS) divides the life-cycle of a space system into seven phases, starting with phase 0, followed by phases A to F. Within phases 0 to D, the design, development, and manufacturing of spacecraft takes place [3]. The early design phase 0/A can be (partially) carried out in Concurrent Engineering Centers (CEC). Concurrent Engineering (CE) is an approach for design and development that emphasizes teamwork, discussion, and rapid iteration. Here, the development tasks from the different engineering disciplines are conducted in parallel and collaboratively. The goal is to quickly establish a consistent system design incorporating the subsystems, equipment, and satellite configuration. The involved engineers exchange and store information using a common system model [4]. This model is created and manipulated using a CE software, such as Virtual Satellite [5].

The Concurrent Engineering Facility (CEF) is the CEC of the German Aerospace Center (DLR). A CE study in the CEF usually takes between one to three weeks and consists of several sessions. A session, generally lasting for ca. 4 hours, is divided into moderated and unmoderated work. The sessions are interrupted by breaks of up to one hour [4], [6]. A typical CE schedule can be found in [7]. During a CE study, the engineers contribute to the system model. First, a decomposition of the system is established, e.g., by modeling the components that constitute the different subsystems. Then, various parameters, including masses and average power usage, are assigned to the components in order to enable system-level budget calculations. These are then analyzed for various system modes [6].

Verification activities throughout the life cycle of a space system are defined within the technical memorandum ECSS-E-TM-10-21A [8]. It covers requirement and design verification from phase 0 to B. Different types of simulators are introduced to verify mission, system, and performance requirements. Several approaches for verification, analysis, and simulation that can be run during or between CE sessions have been presented. Fischer et al. show a formal approach for checking whether an early design is feasible for reaching the mission goals [9]. Here, a verification time of several minutes is considered practical and allows for checking mission feasibility under certain restrictions. The approach has been optimized in [10] and extended to allow for the inclusion simulation models. In [11], it is shown how this approach can be automated and used for continuous verification during early design. A CE process tailored to launcher design is presented in [12]. Here, simulations are integrated into the design evaluation and are executed in parallel to unmoderated sessions. Then, the simulation results are ready for the next moderated session. For verification in phase B, a process for generating simulator configurations for a functional engineering simulator as well as a software validation facility is presented in [13].

B. Model Checking

Model checking [14], [15] is a fully automatic verification technique. The system under consideration is classically represented using an automata-based formalism, e.g., labeled transition systems [16] or Kripke structures. The operational behavior of the system is expressed by transitions between states. Transition labels may represent, e.g, actions, commands, messages, or function calls. The system requirements are given as a formal specification expressed in a temporal logic, such as Linear Temporal Logic (LTL) [17] or Computation Tree Logic (CTL) [14]. Given both a model and a specification, a model-checking tool automatically checks whether the model satisfies the specification. In case there is a violation, a counterexample is produced. Commonly, the counterexample is a trace, i.e., a sequence of states and transitions, that shows how a state violating the specification can be reached. Using this information, either the model or the specification can be adjusted. For an in-depth introduction to model checking, we refer to [18], [19].

In order to reason about quantitative system properties, more expressive modeling formalisms are applied. Systems that exhibit both nondeterministic as well as probabilistic behaviors can be described by Markov Decision Processes and analyzed using probabilistic model checking [20]. If not only discrete system dynamics but also continuous dynamics need to be considered, hybrid model checking [21]–[23] is commonly utilized.

When model checking is applied to analyze complex systems, the corresponding models may become prohibitively large. The model size generally grows exponentially in, e.g., the number of concurrent processes. This issue is known as the state-space

explosion problem. Several approaches have been developed to mitigate this problem, including, for instance, partial-order reduction [24], [25], assume-guarantee reasoning [26], [27], symmetry reduction [28], and SAT-based model checking [29], [30]. Furthermore, symbolic approaches may be utilized [31], [32]. Here, whole sets of states are represented symbolically using Binary Decision Diagrams (BDDs) [33], rather than representing each state explicitly. With this technique, even the verification of large-scale systems becomes tractable [31].

C. Applications of Model Checking in Spacecraft Engineering

Several successful applications of model checking in the aerospace domain have been reported in the literature. The SPIN model checker [34] has been utilized to verify a dually redundant spacecraft controller [35], the downlink module, sequencing module [36] as well as the multi-threaded plan execution module of the Deep Space One's flight software [37], critical parts of the Mars Science Laboratory's software [38], and parts of a launch vehicle's on-board computer software [39]. Brat et al. analyzed an autonomous docking system [40] using the LTSA model checker [41]. In [42], an Attitude and Orbit Control System (AOCS) is verified using the symbolic model checker NuSMV [43] to show the applicability of model checking for this use case. Esteve et al. demonstrate a formal modeling and analysis approach accompanying the development of a modern satellite platform [44]. They apply the COMPASS tool set [45] for various analyses. In [46], the control software of the CubETH nanosatellite is verified using nuXmv [47], an extended version of NuSMV featuring SAT-based and SMT-based model checking. Nardone et al. propose an approach for verifying the autonomous reconfiguration functionality of a satellite system [48] using the probabilistic model checker Prism [49]. Prism is also utilized in [50] to analyze the reliability, availability, and maintainability of a satellite system. The approach presented in [51] focuses on the concurrency and interactions of components in a satellite's mode management. Here, the CAAL [52] tool is utilized for the verification. An application of hybrid model checking is presented by Chan and Mitra [53]. Here, the safety of an autonomous spacecraft rendezvous is analyzed using SpaceEx [23] and their own hybrid model-checking approach.

While model checking can help to detect errors which are hard to find using testing and simulation, its limited scalability is still perceived as a major hurdle for widespread adoption [2], [54]. Several of the above mentioned works ([35], [39], [46], [48], [51]) state that they either faced or expected scalability issues of their chosen approach. However, none actually quantified or systematically evaluated the scalability of the utilized model-checking tools.

The mentioned case studies and approaches can be divided by the project phase they target. While [36]–[39], [42], [44] focus on the verification of flight software which is typically developed during phases C and D, the approaches in [35], [46], [48], [51], [53] are applicable during earlier project phases. Only few of the cited works state that the verification activities were actually performed alongside the development [38], [44] rather than after the fact. Furthermore, the formal modeling and analysis were carried out by experts in formal verification and thus were not transparently integrated into the design process.

III. Modeling of Operational Behavior

For describing an operational design, we employ a simplified variant of UML state machines, since they are sufficient to describe the mode management of satellites in early design [55]. A design may consist of multiple state machines, where interactions and dependencies are expressed using constraints between individual states. Typically, such constraints are derived from mode tables [55] which specify the corresponding subsystem and equipment modes for a given system mode or mission phase. Further constraints may be derived from dependencies between the subsystems or physical constraints. In the following, the syntax, graphical notation, and semantics of the modeling formalism is presented. We begin with the definition of a state machine.

Definition 1 (state machine)**.** *A* state machine *is a tuple* $\mathcal{M} = (\mathcal{L}, \mathcal{C}, \longrightarrow, s^{init})$ *where* $\mathcal{L}$ *is a finite set of states,* $\mathcal{C}$ *is a finite set of commands,* $\longrightarrow \subseteq \mathcal{L} \times \mathcal{C} \times \mathcal{L}$ *is the transition relation, and* $s^{init} \in \mathcal{L}$ *is the initial state. For a given command* $c \in \mathcal{C}$ *and state* $s \in \mathcal{L}$*, the successor state of* s *must be unique, i.e., if* $(s, c, s_1) \in \longrightarrow$ *and* $(s, c, s_2) \in \longrightarrow$*, then* $s_1 = s_2$*.*

Intuitively, the operational behavior of a state machine is represented by the transitions between its states, i.e., if $(s, c, s') \in \longrightarrow$ then the state machine moves from state s to state s' upon receiving the command c. We write $s \xrightarrow{c} s'$ for $(s, c, s') \in \longrightarrow$. The finiteness of $\mathcal{L}$ and $\mathcal{C}$ ensures that the state machine can be explored exhaustively and that model checking is decidable.

The overall system may consist of multiple state machines $\mathcal{M}_i = (\mathcal{L}_i, \mathcal{C}_i, \longrightarrow_i, s_i^{init})$ for $i \in \{1, 2, \ldots, n\}$ which are executed concurrently. A system state $g \in \mathcal{L}_1 \times \mathcal{L}_2 \times \ldots \times \mathcal{L}_n$ comprises the local states for each state machine. We write $g[s_i/s_i']$ to denote the system state that arises from g by replacing the local state of the i^{th} state machine with s_i', i.e., $\langle s_1, s_2, \ldots, s_i, \ldots, s_n\rangle[s_i/s_i'] = \langle s_1, s_2, \ldots, s_i', \ldots, s_n\rangle$. Furthermore, we define the set $L = \mathcal{L}_1 \cup \mathcal{L}_2 \cup \ldots \cup \mathcal{L}_n$ of all local states. To streamline the notations, we additionally define the function $local\colon \mathcal{L}_1 \times \ldots \times \mathcal{L}_n \to \wp(L)$ (where $\wp(X)$ denotes the powerset of X) that returns the set of local states constituting a system state, i.e., $local(\langle s_1, s_2 \ldots s_n\rangle) = \{s_1, s_2, \ldots, s_n\}$.

In order to describe the allowed states of the system and the interactions between state machines, constraints can be added between states of different state machines, as shown in the example in Fig. 1. A *forbid constraint* (s, t), which is denoted as $\rightarrowtail\!\!\!\!\!\!\leftarrowtail$, expresses that the states s and t must not be active at the same time. For instance, the state where the Spacecraft is in the Operate state but the Payload is turned Off is not a valid system state. A *required constraint* $(s, \{t_1, t_2, \ldots, t_k\})$, which is denoted as $\longrightarrow\!\!\!\!\!\!\bullet$, indicates that state s can only be active if at least one of the states $t_1, t_2, \ldots, t_k$ is active as well. In the example, the required constraint $(\mathsf{Idle}, \{\mathsf{Off}\})$ expresses that the Spacecraft state machine can only be in the Idle

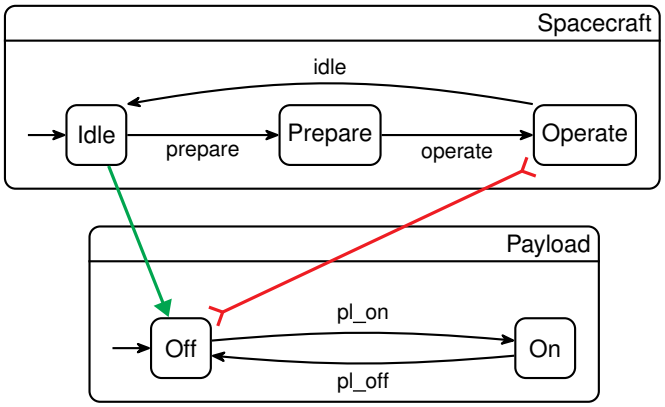

Fig. 1. Two state machines with constraints between them. States are denoted rounded boxes and are connected by transitions.

state if the Payload state machine is in the Off state. Note that a single required constraint $(s, \{t_1, \ldots, t_k\})$ expresses a disjunction over the states $t_1, \ldots, t_k$. This is useful if these states belong to the same state machine (since only ever one can be active at the same time). To express a conjunction, multiple required constraints can be used. For instance, the constraints $(s, \{t_1\})$ and $(s, \{t_2\})$ together require that both t_1 and t_2 must be active if s is active. Based on these notions, we formally define the *valid* states of a system.

Definition 2 (valid system state)**.** *A system state $g \in \mathcal{L}_1 \times \mathcal{L}_2 \times \ldots \times \mathcal{L}_n$ is called* valid *w.r.t. a set of forbid constraints $\mathcal{F} \subseteq L \times L$, where $\mathcal{F}$ is a symmetric relation, and a set of required constraints $\mathcal{R} \subseteq L \times \wp(L)$, if the following conditions hold:*

1) $\forall (s,t) \in \mathcal{F}. \neg\big(s \in \mathit{local}(g) \wedge t \in \mathit{local}(g)\big)$
2) $\forall (s,T) \in \mathcal{R}. s \in \mathit{local}(g) \implies \exists t \in T. t \in \mathit{local}(g)$

The semantics of a set of state machines running concurrently is defined in terms of a transition system, a standard formalism for describing operational behavior. First, we recall the definition of a transition system.

Definition 3 (transition system)**.** *A* transition system *is a tuple $\mathcal{T} = (\mathcal{S}, \mathit{Act}, \mathit{trans}, \mathcal{S}^{\mathit{init}})$ where $\mathcal{S}$ is a finite set of states, Act is a set of actions, $\mathit{trans} \subseteq \mathcal{S} \times \mathit{Act} \times \mathcal{S}$ is the transition relation, and $\mathcal{S}^{\mathit{init}} \subseteq \mathcal{S}$ is the set of initial states.*

We rely on a standard interleaving semantics to represent the concurrent execution of the state machines. In each system state $g = \langle s_1, s_2, \ldots, s_n \rangle$, the outgoing transitions of g consist of the outgoing transitions of s_1, s_2, …, and s_n. That means there is a nondeterministic choice between the transitions of the local states. Intuitively, this nondeterminism may correspond, for instance, to external control inputs, environment changes, or different task scheduling. The constraints are resolved by only allowing transitions to valid system states.

Definition 4 (transition system semantics)**.** *The behavior of a set of concurrently running state machines $\mathcal{M}_i = (\mathcal{L}_i, \mathcal{C}_i, \longrightarrow_i, s_i^{\mathit{init}})$ with $i \in \{1, \ldots, n\}$ under a set of forbid constraints $\mathcal{F}$ and a set of required constraints $\mathcal{R}$, where the initial system state $(s_1^{\mathit{init}}, \ldots, s_n^{\mathit{init}})$ is a valid system state, is defined as a transition system*

$$\mathcal{T} = \big(\mathcal{L}_1 \times \ldots \times \mathcal{L}_n,\ \mathcal{C}_1 \cup \ldots \cup \mathcal{C}_n,\ \mathit{trans},\ \big\{\langle s_1^{\mathit{init}}, \ldots, s_n^{\mathit{init}} \rangle\big\}\big)$$

where
$\mathit{trans} = \big\{(g, c, g') \colon g' = g[s_i / s_i'],\ s_i \xrightarrow{c}_i s_i',\ g' \text{ is valid}\big\}$.

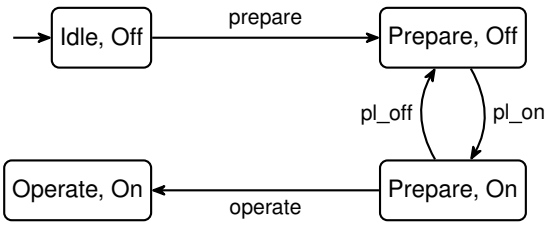

Fig. 2. Transition system semantics of the example in Fig. 1

Consider again the example in Fig. 1. Its transition-system semantics is presented in Fig. 2. In the Prepare, On state, both the Spacecraft state machine and the Payload state machine may execute a transition, thus this state has two outgoing transitions. Furthermore, resolving the constraints reveals that the example system actually contains a deadlock state, namely Operate, On, that has no outgoing transition. The Spacecraft cannot move to the Idle state, since the Payload is not in its Off state and thus the required constraint would be violated. Similarly, the Payload cannot be turned Off, as this is prevented by the forbid constraint.

Using a standard operational formalism as semantics enables us to employ a wide range of existing model-checking tools for the analysis of operational designs.

IV. Transformation-based Analysis Approach

To enable the formal verification of the state-machine models described in Section III, we have implemented a transformation from state machines into the modeling languages of selected modeling-checking tools. In particular, the tools SPIN, NuSMV, Prism, and Storm were chosen for this work in order to cover different model-checking approaches and state-space explosion mitigation techniques. An overview of these tools is provided in Table I. Both Prism and Storm support multiple so-called *engines*, e.g., mtbdd and sparse, which differ in the underlying internal model representation as well as the analysis approach. The tools SPIN, Prism, and Storm support explicit model checking, where each state and transition of the system is represented individually. SPIN utilizes partial-order reduction to reduce redundancies caused by equivalent interleavings due to concurrent execution of state machines. Furthermore, NuSMV, Prism, and Storm implement symbolic model checking where the model is represented using BDDs. Although the systems we consider in this paper are purely nondeterministic, we included the probabilistic model checkers Prism and Storm, since they are still actively developed and include state-of-the-art optimizations for state-space explosion mitigation. Additionally, their quantitative analysis capabilities

```
1  module sc
2    // Idle: 0, Prepare: 1, Operate: 2
3    state_sc : [0 .. 2] init 0;
4
5    [prepare] state_sc = 0
6     -> (state_sc' = 1);
7    [operate] state_sc = 1 & state_pl != 0
8     -> (state_sc' = 2);
9    [idle] state_sc = 2 & state_pl = 0
10    -> (state_sc' = 0);
11 endmodule
12
13 module pl
14   // Off: 0, On: 1
15   state_pl : [0 .. 1] init 0;
16
17   [pl_on] state_pl = 0 & state_sc != 0
18    -> (state_pl' = 1);
19   [pl_off] state_pl = 1 & state_sc != 2
20    -> (state_pl' = 0);
21 endmodule
```

Listing 1. Corresponding PRISM model for the system shown in Fig. 1

```
1  label "SC_Idle" = (state_sc = 0);
2  label "SC_Prepare" = (state_sc = 1);
3  label "SC_Operate" = (state_sc = 2);
4
5  // state Spacecraft.Prepare is reachable
6  E [ F "SC_Prepare" ];
7
8  // state Spacecraft.Operate is reachable
9  E [ F "SC_Operate" ];
10
11 // ...
12
13 // from Idle, Prepare is reachable
14 A [ G "SC_Idle" => E [ F "SC_Prepare" ] ];
15
16 // from Operate, Idle is reachable
17 A [ G "SC_Operate" => E [ F "SC_Idle" ] ];
18
19 // ...
```

Listing 2. Snippet of CTL properties corresponding to the system shown in Fig. 1

are required in case the state machine formalism is extended with quantitative properties. Since Markov Decision Processes (MDPs) subsume transition systems (by setting all transition probabilities to 1.0), probabilistic model checkers can also be applied to verify purely nondeterministic models. Note that neither partial-order reduction nor symbolic model checking require any additional information or annotations and can be used as-is on the transformed models.

In the following, the approach for transforming a state-machine model into the modeling language of a model checker is illustrated. For that, we consider again the example in Fig. 1. Its transformation into the PRISM modeling language is shown in Listing 1. Each state machine corresponds to a `module` in the PRISM model (lines 1 and 13). Within a `module`, a bounded integer variable ranging over the local states of the state machine is created (lines 3 and 15). Finally, each state machine transition is transformed into a guarded command. A command has the form `[command] guard -> update`, where the guard specifies the source state and the update assigns the successor state to the local variable. For instance, the transformation of the System state machine from Idle to Prepare is described by the command in lines 5–6. The `modules` in a PRISM model are executed concurrently, thus the only thing left to be transformed are the constraints. These are encoded into the transition guards such that moving to an invalid system state is prevented. For example, moving from Prepare to Operate is forbidden in case the Payload is Off, which is captured by the condition `state_pl != 0` in line 7 (the symmetric guard is in line 19). The required constraint is encoded analogously (lines 9 and 17). The state space of this PRISM model corresponds exactly to the one shown in Fig. 2.

The STORM model checker supports PRISM models as input and the transformation into PROMELA (the input language of SPIN) is, besides syntactical differences, very similar to the transformation into the PRISM language. Thus, we only consider the transformation into SMV (the input language of NUSMV) next, which is shown in Listing 3 for the example system. In contrast to the other model checkers, NUSMV does not provide a built-in mechanism for concurrently executing modules that is compatible with our semantics. Therefore, all state machines are combined into a single `MODULE`. The transition relation is described by a single Boolean expression over the variables of the model and the updated variables of the successor (`next`) state. The interleaving of the state machines' execution is achieved by only ever updating the state of exactly one state machine and fixing all others. For instance, lines 8–9 describe a transition of the System state machine and thus the state of the Payload state machine remains the same (`next(pl) = pl`). Since the state-machine formalism does not allow for the synchronization of state machines, the case where multiple local transitions are executed within the same system transition does not need to be handled. Thus, there is no combinatorial blow-up when combining the state machines into a single module. The transformation of the constraints proceeds in the same fashion as described for the PRISM language.

In addition to the behavioral model, a specification in the form of temporal properties is automatically generated from the state-machine model. Thus, the user does not need to be familiar with temporal logics to check basic properties. Intuitively, the first part of the generated specification requires that every local state of each state machine is actually reachable. More specifically, for each local state s in the model there should be a reachable system state g that contains the local state s, i.e., $s \in \mathit{local}(g)$. Furthermore, the second part of the specification requires that a local state can eventually be left by transitioning to any of its successor states. This ensures that every state machine can eventually make progress and does not get stuck within a state indefinitely. In the context of commanding a spacecraft, this ensures that all components of

TABLE I
OVERVIEW OF THE SELECTED MODEL-CHECKING TOOLS

Model Checker	Model Types	Model Representation and Analysis	Modeling Language
SPIN [34]	transition systems	explicit, partial-order reduction	PROMELA
NUSMV [43]	transition systems	symbolic using BDDs	SMV
PRISM [49]	Markov chains, Markov Decision Processes		PRISM language
explicit		explicit	
mtbdd		symbolic using multi-terminal BDDs	
sparse		MTBDDs, sparse matrices	
hybrid		MTBDDs, sparse matrices	
STORM [56]	MCs, MDPs, Markov Automata		PRISM language, JANI, other
sparse		explicit, sparse matrices	
dd		MTBDDs	
hybrid		MTBDDs, sparse matrices	

```
1  MODULE main
2  VAR
3    sc : {Idle, Prepare, Operate};
4    pl : {Off, On};
5  INIT
6    sc = Idle & pl = Off;
7  TRANS
8    (sc = Idle &
9     next(sc) = Prepare & next(pl) = pl) |
10   (sc = Prepare & pl != Off &
11    next(sc) = Operate & next(pl) = pl) |
12   (sc = Operate & pl = Off &
13    next(sc) = Idle & next(pl) = pl) |
14   (pl = Off & sc != Idle &
15    next(pl) = On & next(sc) = sc) |
16   (pl = On & sc != Operate &
17    next(pl) = Off & next(sc) = sc);
18 CTLSPEC EF sc = Prepare;
19 CTLSPEC EF sc = Operate;
20 CTLSPEC EF pl = On;
21 CTLSPEC AG (sc = Idle -> EF sc = Prepare);
22 CTLSPEC AG (sc = Prepare -> EF sc = Operate);
23 CTLSPEC AG (sc = Operate -> EF sc = Idle);
24 CTLSPEC AG (pl = Off -> EF pl = On);
25 CTLSPEC AG (pl = On -> EF pl = Off);
```

Listing 3. Corresponding SMV model for the system shown in Fig. 1

the system remain controllable. Additionally, the satisfaction of these properties guarantees that the system does not contain local deadlocks, i.e., situations where only a subset of the state machines are in deadlock but the remaining state machines can still make progress.

The temporal properties derived from the state-machine model are formulated in CTL. Consider again the example in Listing 3. In the SMV language, the properties are defined as part of the module description. The first three properties (lines 18–20) correspond to the first part of the specification. The EF operator intuitively means that there **E**xists a path (from the initial system state), where **F**inally (i.e., eventually, after some finite amount of steps) the condition given after the operator holds. Thus, line 18 specifies that there is a path, such that the Spacecraft state machine is in the Prepare state. This property is satisfied, since there is such a path (see the first transition in Fig. 2). The remaining properties formalize the second part of the specification (lines 21–25). An AG formula is satisfied if for **A**ll paths it holds that the condition after the operator is true **G**lobally (in every state of the path). Thus, the property in line 23 expresses that from every state, where the Spacecraft is in its Operate state, a state where the Spacecraft is Idle can be reached eventually. Since the system state that contains the local Operate state has no outgoing transitions (see Fig. 2), this property is violated. For PRISM and STORM, the properties are put in a separate file. An excerpt of the PRISM properties is shown in Listing 2. Apart from syntactical differences, the properties are equivalent to the ones described for the SMV model.

The modeling of state machines as well as their transformation into the model checker input languages have been implemented into the tool Virtual Satellite [5].

V. EVALUATION

In this section, we present the experimental evaluation of model-checking performance for spacecraft operational designs. In particular, the following research questions are addressed to determine whether the analysis of large-scale early operational designs using model checking is feasible.

(RQ1) How much time is required for the analysis, depending on the system size (in the number of states and state machines)?

(RQ2) How much memory does the analysis of an operational design require?

In order to answer these research questions, we have created a representative operational design that may arise within an early design phase of a satellite. The corresponding model can be scaled by increasing the number of state machines which, in turn, also increases the number of states. Note that the number of states grows roughly exponentially with the number of state machines. The largest operational designs we consider consist of up to 252 state machines, as, from our experience, a design with more components is highly unlikely to arise during early design phases. The generated operational designs have been transformed into the input languages of selected model-checking tools and were finally verified, measuring both time and memory consumption.

The evaluation follows the guidelines for performing empirical studies on formal methods [57] where appropriate. The

generated models, scripts for running the experiments, and the measurement data is available online [58].

A. Analyzed Operational Design

To provide more context to the experiments, this section gives a brief overview of the analyzed operational design. The model is inspired by the state-machine models discussed in [42], [51], an internal reference model, and our own experience in satellite system design. The different system, subsystem, and equipment modes may be switched via commands from the ground segment or due to autonomous reconfigurations. The exact mechanism for mode switching is abstracted away and not part of the model. Furthermore, the model is time-abstract. The design comprises a system state machine with 5 local states capturing the top-level mode, an attitude and orbit control system (AOCS) state machine with 6 states (see Fig. 3) as well as several ancillary state machines for thrusters, reaction wheels, batteries, sensors, payloads, and other equipment with two or three local states each. The states of the state machines are connected with various constraints stating which modes must and must not be active at the same time. For instance, the *On* state of the payload requires that the *Fine Pointing* mode of the AOCS is active and in the *Station Keeping* AOCS mode the reaction wheels must not be turned *Off*, expressed using a forbid constraint. Generally, the states of the ancillary state machines are constrained by the top-level mode as well as the AOCS state. However, there are no constraints between the ancillary state machines. The smallest instance of the design comprises 8 state machines and its corresponding transition system has 468 states.

During the creation of the operational design, we detected and fixed multiple unintended and unexpected deadlocks using the implementation described in Section IV. The found defects were not immediately apparent in a visual inspection of the state-machine diagrams and finding the deadlocks using simulation alone would have been difficult, since they were only triggered by a very specific scheduling of the state machine execution. While the evaluation of the approach's practicability was not the focus of the evaluation, this experience indicates that the formal verification of early operational designs is indeed useful in practice.

Scaling of the operational design to larger instances with more state machines is accomplished by cloning the ancillary state machines. For generating a design with $n + 1$ state machines from a design with n state machines, one of the ancillary state machines with the least number of copies is selected. Then, a copy of this state machine, including all connected forbid and required constraints, is created and added to the design. The selection of the next state machine proceeds in a round-robin fashion, i.e., the least recently cloned state machine is selected next. The round-robin strategy guarantees a deterministic scaling of the model and also ensures an even distribution of the different types of ancillary state machines. We have also experimented with a randomized selection of the next state machine to clone. However, this had no significant influence on the measurements. Note that the general structure of the model, i.e., several equipment state machines constrained by a central system state machine and an AOCS state machine, is preserved by the described scaling approach. Thus, even the scaled-up models provide a reasonable approximation of real-world models. Alternatively, the model size could also be increased by adding new states to the existing state machines. We did not pursue this option since equipment with tens or hundreds of modes are highly unlikely to occur in early design phases. All instances of the operational design, that have been generated as described, are free of deadlocks to ensure that the whole state space is explored during model checking.

B. Data Collection Procedure

The model has been instantiated for a number of state machines ranging from 8 to 252, where the corresponding transition system of the largest instance has 4×10^{70} states. Note that in the model-checking community, even models with more than 10^8 states are considered as large-scale [31]. Each generated instance has then been automatically transformed into the modeling languages Promela (for SPIN), SMV (for NuSMV), and the PRISM language (for PRISM and STORM). All mentioned model checkers with the exception of SPIN support symbolic model checking which uses binary decision diagrams (BDDs) for the model representation. The size of these BDDs, and with it the verification time, crucially depends on the ordering of the variables that span the model's state space. To avoid scalability issues due to a "bad" variable ordering, reordering was applied to the generated models. For the SMV models, the built-in reordering support of NuSMV has been utilized. At the time of writing, the standard version of PRISM did not support automated variable reordering. Therefore, we used an extended version of PRISM [59] to perform the BDD optimization.

We conducted two sets of experiments for measuring the time and memory requirements for verification using the selected model checkers. In the first set, the generated model instances were checked for global deadlocks, i.e., system states that have no outgoing transition, meaning that none of the state machines can progress. Since none of the models actually contains a deadlock, the whole reachable state space of each instance is explored completely. This is the worst-case for deadlock checking and thus the measurements provide an upper bound for the analysis time and memory consumption. In the second set of experiments, the satisfaction of the generated temporal properties (see Section IV) is checked. For the time measurements, we rely on the built-in diagnostics of the model-checking tools. In what follows, we consider the analysis time as the combined time needed for model parsing, construction, and analysis. The peak memory consumption was measured using the `time` command as supplied by most Linux distributions. We additionally used the elapsed-time measurement of the `time` command to cross-check the analysis time reported by the model checkers. The experiments were executed sequentially, from the smallest to the largest instance. A series of runs for a single model checker was aborted once a timeout of 30 minutes or a memory consumption of 16 GB

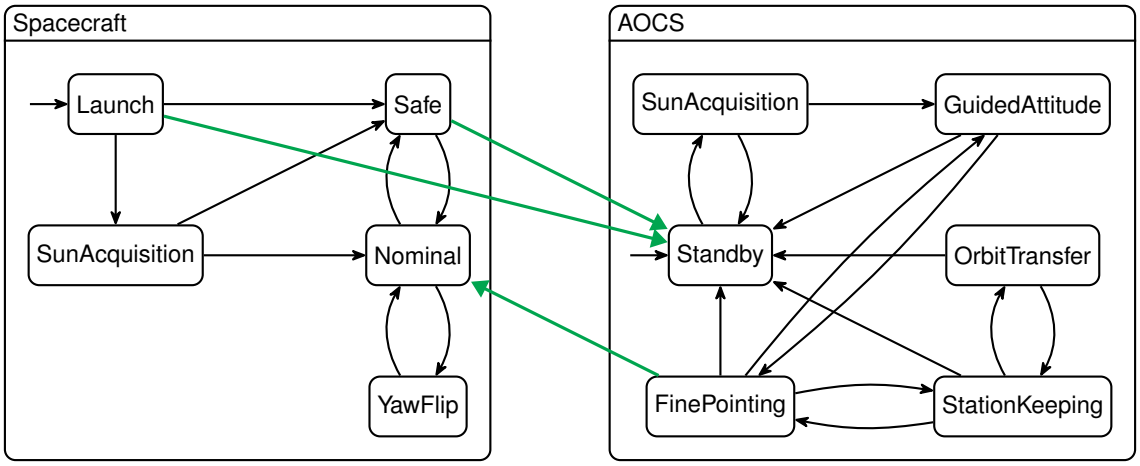

Fig. 3. The spacecraft and AOCS state machines of the analyzed design. The command labels have been omitted for better readability.

was reached. Both PRISM and STORM implement multiple *engines*, which use different internal model representations and model-checking algorithms. We have utilized all engines as far as they were supported for the given model size and set of properties. All experiments have been conducted on a workstation with an Intel Core i7-4790 CPU (3.6 GHz) and 32 GB RAM running Ubuntu 20.04 LTS.

C. Results

In the following, the experiments are evaluated with respect to the research questions.

1) Analysis Time: We first consider the required time for deadlock checking of increasingly large models using different model checkers. The measurement results are shown in Fig. 4. The model-checking engines relying on an explicit representation of the state space, where every individual state is stored separately, are most susceptible to the state-space explosion problem and scale only to 19 (PRISM explicit) or 22 (STORM sparse) state machines before reaching a timeout. The SPIN model checker is also based on explicit model checking, but additionally uses partial-order reduction and thus reached a size of 27 state machines. All other used model checkers and engines which are based on a symbolic state space representation with BDDs allowed us to check for deadlocks even for the largest system instance without hitting a timeout, with NUSMV being the fastest. Even though the analysis time grows exponentially for all approaches, the growth for symbolic approaches is much slower than for explicit. The measurements for verifying temporal properties show similar results (see Fig. 5). These results indicate that the model's structure, i.e., concurrent state machines constrained by a small set of central state machines, admits a compact symbolic representation. Note that the symbolic engines of STORM currently do not support the analysis of all considered temporal properties and thus their run times were omitted from the plot.

Given these results, we can answer **RQ1**, concluding that an analysis of the operational design using *symbolic model checking* requires only minutes or tens of minutes even for the largest instances.

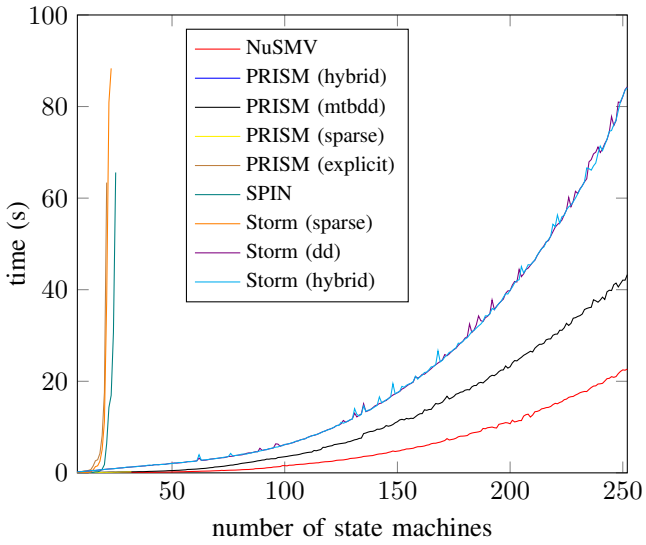

Fig. 4. Time needed for deadlock checking

2) Memory Usage: Figure 6 shows the peak memory usage for verifying the temporal properties depending on the state space size. The symbolic engines of PRISM (mtbdd, sparse, hybrid) and STORM (dd, hybrid) use roughly equal amounts of memory, respectively, thus only one representative is shown in the plot. For explicit model checking (SPIN, PRISM explicit, STORM sparse) the memory usage grows exponentially. The same is true for NUSMV, although the growth is much slower. For STORM and PRISM, the memory usage is capped at just below 4 GB and between 1 GB and 2 GB, respectively. In case of PRISM, the available memory for the BDD representation is limited to 1 GB by default. However, increasing this limit did neither incur a higher memory usage nor did it reduce the analysis time.

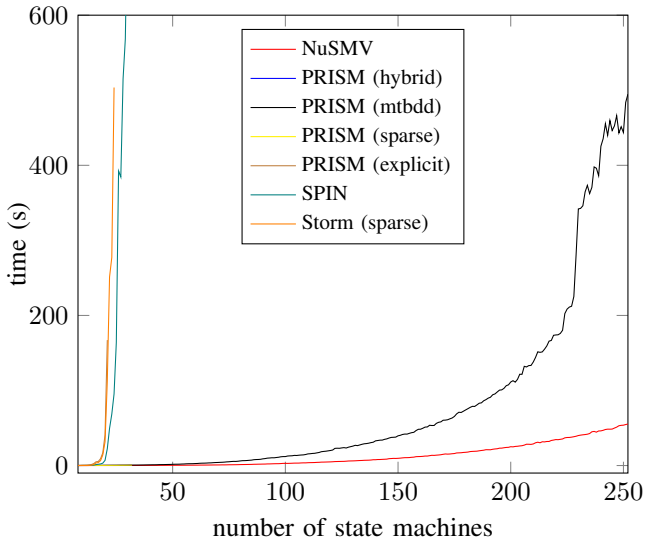

Fig. 5. Time needed for checking temporal properties

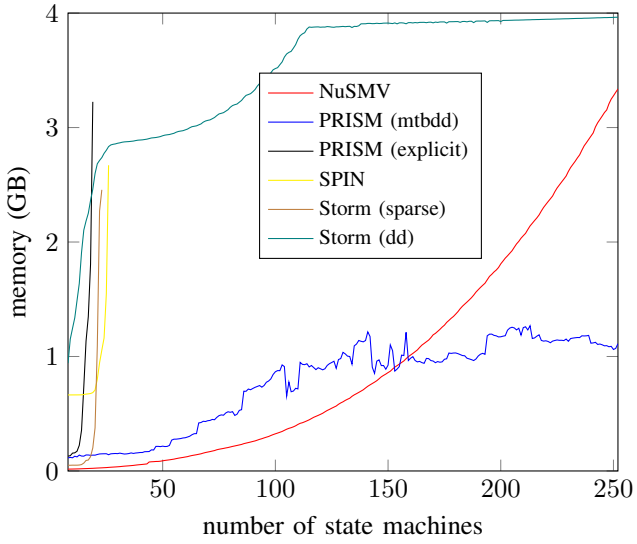

Fig. 6. Peak memory usage for checking temporal properties

These results provide an answer for **RQ2**. For the considered operational design, 4 GB of memory are sufficient for all model sizes and all considered model-checking tools.

D. Threats to Validity

The operating system and the hardware may cause variations in the time measurements, which threatens internal validity. This has been accounted for by using three runs for each analysis with a warm-up run beforehand. The maximum relative standard deviation between runs was 13%, but for almost all analysis runs the deviation was below 5%, showing minimal, or at least consistent, outside influences. Internal validity is further threatened by the fact that the considered model checkers use different input languages, which may introduce subtle semantic differences in the generated models. In order to make sure all model checkers operate on the same state spaces, the number of states and transitions as reported by the model checkers have been compared for all generated instances. For any given model instance, the number of states and transitions of the SMV, PRISM, and PROMELA models were identical. Additionally, the state-space structure of the smallest generated models has been investigated using the simulation facilities of the respective model-checking tools. No differences between the generated models have been found.

Using a single operational design for the experiments threatens external validity. However, the considered operational model was specifically designed to be representative and structurally close to the designs that may arise within real projects. Therefore, the results should be applicable to similar models. External validity is further threatened by the chosen modeling formalism, where interactions and dependencies between state machines may only be expressed using constraints. It is to be expected that using a more sophisticated formalism, which also allows for communication via message passing, synchronization, or shared variables, will increase the time and memory requirements.

E. Discussion

The experiment results indicate that even for large-scale early operational designs, the verification using model checking is feasible. This has an important consequence: Formal modeling and verification can already be performed in the earliest space system engineering phases, even before the implementation of the on-board software. Thus, design errors can potentially be detected much earlier in the design process. Since fixing design errors generally becomes more expensive the longer they remain undetected [60], this can significantly lower the project costs and duration. In our experience, another advantage of early modeling activities is that they can uncover ambiguities and inconsistencies in the design. Furthermore, having a model with a known and fixed semantics can facilitate a common understanding of all involved engineers.

Given the comparatively low time requirements of the model-checking process, the verification can be potentially conducted even alongside concurrent engineering sessions (cf. Section II-A). Since the presented transformation approach is fully automated and requires no hand-crafted abstractions or optimizations of the generated models, the verification can be transparently integrated into existing modeling and engineering software. Also, while having an expert in formal verification in the engineering team certainly is advantageous, it is not strictly required for applying the presented approach. The memory requirements of the verification can easily be met by today's commodity hardware. Thus, the analysis can potentially be executed on the engineers' local workstations.

VI. Conclusion

We have investigated the scalability of model checking for verifying spacecraft operational designs within early space system design phases. For this, a model of a satellite's mode management has been investigated. Its behavior is expressed

in a state-machine formalism that allows for the concurrent execution of state machines and the formulation of constraints between states. The goal of the verification was to show that the model contains no deadlocks and that all system modes, subsystem modes, and equipment modes can eventually be activated or deactivated. The model was built to be easily scalable by increasing the number of concurrent state machines. We have implemented transformations of the state-machine model as well as the specification into the input modeling languages of various model-checking tools. This enabled us to compare different state-space explosion mitigation techniques w.r.t. analysis time and memory consumption. The results show that model-checking tools that utilize symbolic model representations using binary decision diagrams scale very well for the type of model we have considered. Even for large-scale models with over 250 state machines, the verification could be performed in under one minute while requiring only up to 4 GB of memory. These results could be achieved without requiring hand-crafted model optimizations or abstraction, only relying on the automated built-in optimizations of the model checkers. This means that the verification approach using model checking can be transparently integrated into early design processes without requiring the support of experts in formal verification.

The work presented here can be extended in several directions. The utilized modeling formalism mainly targets early design phases where many implementation details are not yet available and thus only allows for the modeling of abstract behavior. To enable modeling and verification also for later project phases, the modeling formalism needs to be extended for increased expressiveness. For instance, facilities for modeling message passing and communication between state machines could be added. Furthermore, extensions for modeling and reasoning about quantitative properties, e.g., energy consumption and reliability, could be added. While these extensions would allow for a more sophisticated verification, it is to be expected that this will decrease the scalability of the approach. Further investigation for automated optimizations are required in this case. The current implementation of the approach utilizes the tool Virtual Satellite [5] for creating and transforming the state-machine models. This implementation could be fully integrated into the tool, which would enable a more streamlined integration into the design process.

REFERENCES

[1] J. A. Davis, M. A. Clark, D. D. Cofer, A. Fifarek, J. Hinchman, J. A. Hoffman, B. W. Hulbert, S. P. Miller, and L. G. Wagner, "Study on the barriers to the industrial adoption of formal methods," in *Formal Methods for Industrial Critical Systems - 18th International Workshop, FMICS 2013, Madrid, Spain, September 23-24, 2013. Proceedings*, ser. Lecture Notes in Computer Science, C. Pecheur and M. Dierkes, Eds., vol. 8187. Springer, 2013, pp. 63–77.

[2] S. A. Chien, "Formal methods for trusted space autonomy: Boon or bane?" in *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, ser. Lecture Notes in Computer Science, J. V. Deshmukh, K. Havelund, and I. Perez, Eds., vol. 13260. Springer, 2022, pp. 3–13.

[3] ECSS Secretariat, "ECSS-M-ST-10C: Space project management – project planning and implementation," ESA-ESTEC Requirements & Standards Division, Noordwijk, Netherlands, Standard, 2009.

[4] P. M. Fischer, M. Deshmukh, V. Maiwald, D. Quantius, A. M. Gomez, and A. Gerndt, "Conceptual data model: A foundation for successful concurrent engineering," *Concurr. Eng. Res. Appl.*, vol. 26, no. 1, pp. 55–76, 2018.

[5] DLR. (2023) Virtual Satellite. [Online]. Available: https://github.com/virtualsatellite/VirtualSatellite4-Core

[6] S. S. Jahnke, A. M. Gomez, P. M. Fischer, and C. Lange, "Concurrent engineering in later project phases: current methods and future demands," in *69th International Astronautical Congress (IAC)*, 10 2018. [Online]. Available: https://elib.dlr.de/121731/

[7] D. Quantius, H. Wessel, P. M. Fischer, and D. Peters, "Progression visualisation of mass parameters during a concurrent engineering study," in *Cooperative Design, Visualization, and Engineering - 19th International Conference, CDVE 2022, Virtual Event, September 25-28, 2022, Proceedings*, ser. Lecture Notes in Computer Science, Y. Luo, Ed., vol. 13492. Springer, 2022, pp. 13–20.

[8] ECSS Secretariat, "ECSS-E-TM-10-21A: Space engineering – system modeling and simulation," ESA-ESTEC Requirements & Standards Division, Noordwijk, Netherlands, Tech. Rep., 2010.

[9] P. M. Fischer, D. Lüdtke, V. Schaus, O. Maibaum, and A. Gerndt, "Formal verification in early mission planning," in *Simulation and EGSE facilities for Space Programmes*, 9 2012. [Online]. Available: https://elib.dlr.de/119907/

[10] P. M. Fischer, D. Lüdtke, V. Schaus, and A. Gerndt, "A formal method for early spacecraft design verification," in *2013 IEEE Aerospace Conference*, 2013, pp. 1–8.

[11] V. Schaus, P. M. Fischer, D. Lüdtke, M. Tiede, and A. Gerndt, *A Continuous Verification Process in Concurrent Engineering*, 2013.

[12] P. M. Fischer, M. Deshmukh, A. Koch, R. Mischke, A. M. Gomez, A. Schreiber, and A. Gerndt, "Enabling a conceptual data model and workflow integration environment for concurrent launch vehicle analysis," in *69th International Astronautical Congress (IAC)*, 10 2018. [Online]. Available: https://elib.dlr.de/122158/

[13] P. Fischer, H. Eisenmann, and J. Fuchs, "Functional verification by simulation based on preliminary system design data," in *6th International Workshop on Systems and Concurrent Engineering for Space Applications (SECESA)*, 2014, pp. 8–10.

[14] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, ser. Lecture Notes in Computer Science, D. Kozen, Ed., vol. 131. Springer, 1981, pp. 52–71.

[15] J. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, ser. Lecture Notes in Computer Science, M. Dezani-Ciancaglini and U. Montanari, Eds., vol. 137. Springer, 1982, pp. 337–351.

[16] R. M. Keller, "Formal verification of parallel programs," *Communications of the ACM*, vol. 19, no. 7, pp. 371–384, 1976.

[17] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57.

[18] C. Baier and J. Katoen, *Principles of model checking*. MIT Press, 2008.

[19] E. M. Clarke, O. Grumberg, D. Kroening, D. A. Peled, and H. Veith, *Model checking, 2nd Edition*. MIT Press, 2018. [Online]. Available: https://mitpress.mit.edu/books/model-checking-second-edition

[20] M. Y. Vardi, "Automatic verification of probabilistic concurrent finite-state programs," in *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*. IEEE Computer Society, 1985, pp. 327–338.

[21] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theor. Comput. Sci.*, vol. 138, no. 1, pp. 3–34, 1995.

[22] T. A. Henzinger, P. Ho, and H. Wong-Toi, "HYTECH: A model checker for hybrid systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 1, no. 1-2, pp. 110–122, 1997. [Online]. Available: https://doi.org/10.1007/s100090050008

[23] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "Spaceex: Scalable verification of hybrid systems," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science,

G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 379–395.

[24] A. Valmari, "A stubborn attack on state explosion," *Formal Methods Syst. Des.*, vol. 1, no. 4, pp. 297–322, 1992.

[25] D. A. Peled, "All from one, one for all: on model checking using representatives," in *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, ser. Lecture Notes in Computer Science, C. Courcoubetis, Ed., vol. 697. Springer, 1993, pp. 409–423.

[26] A. Pnueli, "In transition from global to modular temporal reasoning about programs," in *Logics and Models of Concurrent Systems - Conference proceedings, Colle-sur-Loup (near Nice), France, 8-19 October 1984*, ser. NATO ASI Series, K. R. Apt, Ed., vol. 13. Springer, 1984, pp. 123–144.

[27] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "You assume, we guarantee: Methodology and case studies," in *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, ser. Lecture Notes in Computer Science, A. J. Hu and M. Y. Vardi, Eds., vol. 1427. Springer, 1998, pp. 440–451.

[28] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn, "Exploiting symmetry in temporal logic model checking," *Formal Methods Syst. Des.*, vol. 9, no. 1/2, pp. 77–104, 1996.

[29] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, ser. Lecture Notes in Computer Science, R. Cleaveland, Ed., vol. 1579. Springer, 1999, pp. 193–207.

[30] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta, "Combining decision diagrams and SAT procedures for efficient symbolic model checking," in *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, ser. Lecture Notes in Computer Science, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Springer, 2000, pp. 124–138.

[31] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Inf. Comput.*, vol. 98, no. 2, pp. 142–170, 1992.

[32] K. L. McMillan, *Symbolic model checking*. Kluwer, 1993.

[33] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[34] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[35] F. Schneider, S. M. Easterbrook, J. R. Callahan, and G. J. Holzmann, "Validating requirements for fault tolerant systems using model checking," in *3rd International Conference on Requirements Engineering (ICRE '98), Putting Requirements Engineering to Practice, April 6-10, 1998, Colorado Springs, CO, USA, Proceedings*. IEEE Computer Society, 1998, pp. 4–13.

[36] P. Gluck and G. Holzmann, "Using spin model checking for flight software verification," in *Proceedings, IEEE Aerospace Conference*, vol. 1, 2002, pp. 1–1.

[37] K. Havelund, M. R. Lowry, and J. Penix, "Formal analysis of a spacecraft controller using SPIN," *IEEE Trans. Software Eng.*, vol. 27, no. 8, pp. 749–765, 2001.

[38] G. J. Holzmann, "Mars code," *Commun. ACM*, vol. 57, no. 2, pp. 64–73, 2014.

[39] R. Krishnan and V. R. Lalithambika, "Modeling and validating launch vehicle onboard software using the spin model checker," *Journal of Aerospace Information Systems*, vol. 17, no. 12, pp. 695–699, 2020. [Online]. Available: https://doi.org/10.2514/1.I010876

[40] G. Brat, E. Denney, D. Giannakopoulou, J. Frank, and A. Jonsson, "Verification of autonomous systems for space applications," in *2006 IEEE Aerospace Conference*, 2006, pp. 11 pp.–.

[41] J. Magee and J. Kramer, *State models and java programs*. wiley Hoboken, 1999.

[42] X. Gan, J. Dubrovin, and K. Heljanko, "A symbolic model checking approach to verifying satellite onboard software," *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, vol. 46, 2011.

[43] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," in *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, ser. Lecture Notes in Computer Science, E. Brinksma and K. G. Larsen, Eds., vol. 2404. Springer, 2002, pp. 359–364.

[44] M. Esteve, J. Katoen, V. Y. Nguyen, B. Postma, and Y. Yushtein, "Formal correctness, safety, dependability, and performance analysis of a satellite," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE Computer Society, 2012, pp. 1022–1031.

[45] M. Bozzano, A. Cimatti, J. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri, "The COMPASS approach: Correctness, modelling and performability of aerospace systems," in *Computer Safety, Reliability, and Security, 28th International Conference, SAFECOMP 2009, Hamburg, Germany, September 15-18, 2009. Proceedings*, ser. Lecture Notes in Computer Science, B. Buth, G. Rabe, and T. Seyfarth, Eds., vol. 5775. Springer, 2009, pp. 173–186.

[46] E. Stachtiari, A. Mavridou, P. Katsaros, S. Bliudze, and J. Sifakis, "Early validation of system requirements and design through correctness-by-construction," *J. Syst. Softw.*, vol. 145, pp. 52–78, 2018.

[47] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 334–342.

[48] V. Nardone, A. Santone, M. Tipaldi, and L. Glielmo, "Probabilistic model checking applied to autonomous spacecraft reconfiguration," in *2016 IEEE Metrology for Aerospace (MetroAeroSpace)*, 2016, pp. 556–560.

[49] M. Z. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_47

[50] Z. Peng, Y. Lu, A. Miller, C. W. Johnson, and T. Zhao, "A probabilistic model checking approach to analysing reliability, availability, and maintainability of a single satellite system," in *Seventh UKSim/AMSS European Modelling Symposium, EMS 2013, 20-22 November, 2013, Manchester UK*, D. Al-Dabass, A. Orsoni, and Z. Xie, Eds. IEEE, 2013, pp. 611–616.

[51] V. Nardone, A. Santone, M. Tipaldi, D. Liuzza, and L. Glielmo, "Model checking techniques applied to satellite operational mode management," *IEEE Syst. J.*, vol. 13, no. 1, pp. 1018–1029, 2019.

[52] J. R. Andersen, N. Andersen, S. Enevoldsen, M. M. Hansen, K. G. Larsen, S. R. Olesen, J. Srba, and J. K. Wortmann, "CAAL: concurrency workbench, aalborg edition," in *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, ser. Lecture Notes in Computer Science, M. Leucker, C. Rueda, and F. D. Valencia, Eds., vol. 9399. Springer, 2015, pp. 573–582.

[53] N. Chan and S. Mitra, "Verifying safety of an autonomous spacecraft rendezvous mission," in *ARCH17. 4th International Workshop on Applied Verification of Continuous and Hybrid Systems, collocated with Cyber-Physical Systems Week (CPSWeek) on April 17, 2017 in Pittsburgh, PA, USA*, ser. EPiC Series in Computing, G. Frehse and M. Althoff, Eds., vol. 48. EasyChair, 2017, pp. 20–32.

[54] S. A. Jacklin, "Survey of verification and validation techniques for small satellite software development," Tech. Rep., 2015.

[55] J. Eickhoff, *Onboard computers, onboard software and satellite operations: an introduction*. Springer Science & Business Media, 2011.

[56] C. Hensel, S. Junges, J. Katoen, T. Quatmann, and M. Volk, "The probabilistic model checker storm," *Int. J. Softw. Tools Technol. Transf.*, vol. 24, no. 4, pp. 589–610, 2022.

[57] M. H. ter Beek and A. Ferrari, "Empirical formal methods: Guidelines for performing empirical studies on formal methods," *Software*, vol. 1, no. 4, pp. 381–416, 2022.

[58] P. Chrszon, "Applicability of Model Checking for Verifying Spacecraft Operational Designs - Artifact," Apr. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8186567

[59] J. Klein, C. Baier, P. Chrszon, M. Daum, C. Dubslaff, S. Klüppelholz, S. Märcker, and D. Müller, "Advances in probabilistic model checking with PRISM: variable reordering, quantiles and weak deterministic büchi automata," *Int. J. Softw. Tools Technol. Transf.*, vol. 20, no. 2, pp. 179–194, 2018.

[60] P. Liggesmeyer, M. Rothfelder, M. Rettelbach, and T. Ackermann, "Qualitätssicherung Software-basierter technischer Systeme – Problembereiche und Lösungsansätze," *Inform. Spektrum*, vol. 21, no. 5, pp. 249–258, 1998.