Model Checking of Spacecraft Operational Designs: A Scalability Analysis

Philipp Chrszon[1*], Paulina Maurer[1], George Saleip[1], Sascha Müller[1], Philipp M. Fischer[1], Andreas Gerndt[1,2], Michael Felderer[1,3]

[1*]Institute of Software Technology, German Aerospace Center (DLR), Braunschweig, Germany.
[2]University of Bremen, Bremen, Germany.
[3]University of Cologne, Cologne, Germany.

*Corresponding author(s). E-mail(s): philipp.chrszon@dlr.de;
Contributing authors: 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

Ensuring the correct and safe behavior of a spacecraft is a main objective in space-system design. Since spacecraft consist of highly complex and tightly integrated components developed by large teams of engineers from various different disciplines, this is a challenging task. Increasingly, formal-verification methods such as model checking are applied to establish the correctness of safety-critical parts or subsystems. Generally, the often limited scalability of model checking due to the state-space explosion problem hinders the wide-spread adoption of this technique. In this paper, we systematically examine the scalability of model checking for verifying behavioral models that arise within early space-system design phases. For this, we created a representative model for the mode management of a satellite that can be scaled in terms of its size and the complexity of interactions between system components. The model can be transformed into the input languages of various model-checking tools, which enables a comparative study of various model checking algorithms and also facilitates analyzing the impact of different communication schemes on the scalability. The evaluation shows promising results regarding the applicability of model checking within the spacecraft design process.

Keywords: Aerospace, Space Systems, State Machines, Model Checking, Scalability

This version of the article has been accepted for publication, after peer review, but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/10.1007/s10270-025-01281-6

1 Introduction

In space-system design, ensuring a high reliability of spacecraft is of utmost importance. Any critical design error that remains undetected until after the launch potentially causes a partial or complete mission failure, which not only leads to significant financial losses, but may also endanger other spacecraft or even human lives. However, the inherent complexity of space systems makes guaranteeing their correct and safe behavior challenging. They consist of various highly interconnected parts, often developed by engineers of different disciplines, and are controlled by sophisticated on-board computer systems. The interdependence and tight integration of components makes the correction of design errors in later development phases very costly. It is therefore highly desirable to find errors in a spacecraft's design as early as possible.

Traditionally, verification and validation schemes for space systems extensively utilize testing and simulation. Here, the system or a model of the system is executed and observed for unintended behaviors. However, this can only show the presence of errors but not their absence, since full coverage of all possible execution paths is never achieved in practice. For systems or subsystems that are safety-critical, simulation can be complemented by formal verification. Such formal techniques aim to provide a proof of correctness, showing that the system's specification is satisfied for all possible executions.

Model checking is a formal verification technique that is particularly well-suited for the analysis of highly concurrent systems and can uncover errors that are caused by very specific task scheduling, e.g., race conditions. Such errors are notoriously hard to reliably detect using simulation, as they rarely manifest themselves. Given a system model or source code, a model-checking tool verifies the system by a systematic exhaustive exploration of the system's state space. The main advantage of this approach is that it works fully automatically and does not require any user input during the verification process.

The main limitation of model checking is its scalability, since potentially the whole state space of a system has to be represented and explored. The state space usually grows exponentially with the number of variables or parallel processes, which is known as the state-space explosion problem. A wide range of approaches and techniques to mitigate this issue have been developed, which can make the verification of large-scale systems tractable. However, the effective use 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 [1–3].

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

1. created a representative model for the mode management of a spacecraft which allows a flexible scaling of its complexity,
2. implemented transformations of the model into the modeling languages of various model-checking tools, and
3. systematically examined the analysis durations and memory usage for detecting deadlocks and livelocks using different model checkers.

To enable a semantically unambiguous modeling of the mode management, we define a formal syntax and semantics for state machines, where transitions between states define the possible mode switches and where additional mode constraints can be formulated. Furthermore, state machines may interact using synchronous and asynchronous communication, such that complex mode-management schemes and (semi-)autonomous behavior can be adequately captured.

The complexity of the representative mode-management model can be adjusted in two dimensions. First, the model can be scaled in the number of state machines. Second, the type of interactions between the state machines can be freely selected, ranging from no communication at all, over synchronous communication, to asynchronous communication with increasing message buffer sizes.

The implemented transformation of state-machine models into the input languages of various model checkers is specifically designed to be transparently integrated into the design process and to be used by non-experts in formal verification. Therefore, it only utilizes fully automated optimizations and state-space explosion mitigation techniques that are provided by the selected model-checking tools and require no hand-crafted model transformations or abstractions. Furthermore, the implementation compensates for the varying feature-sets of the targeted modeling languages by generating additional model code that emulates missing language constructs. 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.

This paper is an extended version of a conference paper [4] and includes the following changes and additions. First, the modeling formalism has been modified to support synchronization between state machines. Second, the implementation of the transformational analysis approach has been completely rewritten to support both synchronous and asynchronous communication within the state-machine models. Third, a new mode-management model has been created that significantly differs from the model used in [4]. Finally, the scalability evaluation now includes an assessment of the impact of including synchronous and asynchronous communication.

2 Background

This section gives an overview of spacecraft systems engineering processes and the model-checking approach for formal verification and analysis.

2.1 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 [5]. 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 of 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 [6]. This model is created and manipulated using a CE software, such as Virtual Satellite [7].

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 approximately four hours, is divided into moderated and unmoderated work. The sessions are interrupted by breaks of up to one hour [6, 8]. A typical CE schedule can be found in [9]. During a CE study, the engineers create, extend, and modify 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 [8].

Verification activities throughout the life cycle of a space system are defined within the technical memorandum ECSS-E-TM-10-21A [10]. 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 [11]. 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 [12] and extended to allow for the inclusion simulation models. In [13], 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 [14]. 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 [15].

2.2 Model Checking

Model checking [16, 17] is a fully automatic verification technique. The system under consideration is classically represented using an automata-based formalism, e.g., labeled transition systems [18] 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) [19] or Computation Tree Logic (CTL) [16]. 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 [20, 21].

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 [22]. If not only discrete system dynamics but also continuous dynamics need to be considered, hybrid model checking [23–25] 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 [26, 27], assume-guarantee reasoning [28, 29], symmetry reduction [30], and SAT-based model checking [31, 32]. Furthermore, symbolic approaches may be utilized [33, 34]. Here, whole sets of states are represented symbolically using Binary Decision Diagrams (BDDs) [35], rather than representing each state explicitly. With this technique, even the verification of large-scale systems becomes tractable [33].

3 Modeling of Operational Behavior

For describing spacecraft operational designs, we define a state-based modeling formalism, which is intended to strike a balance between simplicity and expressiveness. On the one hand, it should be simple enough to quickly and intuitively describe behavior, such as mode management, within early design phases or even during CE sessions. On the other hand, its expressiveness should allow engineers to adequately describe all behavioral aspects that arise during early design, i.e., concurrency, possibly asynchronous communication, and mode constraints. To this end, it incorporates constructs typically found in mode-management diagrams [36] and allows describing message-based communication like in, e.g., Harel statecharts [37].

Within this framework, an operational design may consist of one or more state machines which are executed concurrently. State machines can interact with each other by means of “handshaking”, i.e., they synchronously participate in the interaction by executing their respective transition at the same time. If either one of the state machines is not ready to send or receive, the other’s respective send or receive action is blocked. Choosing synchronization as the basic and sole communication primitive has several advantages. First, its formal definition is rather straightforward and some model-checking tools support it natively. Second, asynchronous communication can be derived from synchronization by modeling the asynchronous communication primitives explicitly as part of the model. This, in turn, enables a flexible choice of the concrete mechanism for message passing, e.g., whether all state machines share a message queue or not and whether messages must be processed in a strict FIFO ordering. Formally,

both concurrency and synchronization are captured by a parallel-composition operator that combines two state machines into one, incorporating the behaviors of both. The operational behavior of the overall system is then obtained by a repeated application of the composition operator until all component state machines have been combined into a single system state machine. Additionally, constraints between individual state can be defined which express forbidden state combinations or dependencies between states. The kinds of constraints supported by the formalism are inspired by the kinds of constraints provided by ESA's Space Systems Design Editor [38]. Typically, such constraints are derived from mode tables [36] which specify the corresponding subsystem and equipment modes for a given system mode or mission phase. Further constraints may be inferred from dependencies between the subsystems or physical constraints. The semantics of a composed state machine under a set of constraints is defined as a transition system, a standard formalism for describing the operational behavior of reactive systems. Using such a standard formalism allows us to apply a wide range of existing tools for the simulation, analysis, and verification of the modeled system.

In the following, the syntax, graphical notation, and semantics of the modeling formalism are presented.

3.1 State Machines

We begin with the definition of a state machine, which is adapted from the definition of labeled transition systems (see, e.g., [20]).

Definition 1 (state machine)**.** *A* state machine *is a tuple* $\mathcal{M} = (\mathcal{S}, Act, \longrightarrow, s^{init}, L, local)$ *where* $\mathcal{S}$ *is a finite set of states,* Act *is a finite set of actions,* $\longrightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$ *is the transition relation,* $s^{init} \in \mathcal{S}$ *is the initial state,* L *is a set of local state labels, and* $local : \mathcal{S} \to \wp(L)$ *labels each state* $s \in \mathcal{S}$ *with the set of local states that are active in* s*. For a given action* $\alpha \in Act$ *and state* $s \in \mathcal{S}$*, the successor state of* s *must be unique, i.e., if* $(s, \alpha, s_1) \in \longrightarrow$ *and* $(s, \alpha, s_2) \in \longrightarrow$*, then* $s_1 = s_2$*.*

Intuitively, the operational behavior of a state machine is represented by the transitions between its states, i.e., if $(s, \alpha, s') \in \longrightarrow$ then the state machine moves from state s to state s' on action α. We write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \longrightarrow$. An action α may be interpreted as a message that is either sent or received, or it may also represent some internal operation. The finiteness of $\mathcal{S}$ and Act ensures that the state machine can be explored exhaustively and that model checking is decidable.

A state machine may either be *atomic*, representing, e.g., a single system component, or it may arise via composition of state machines. In the latter case, each state of the composed state machine in turn comprises a combination of states from the constituent state machines. The purpose of the set L and the function *local* is to preserve the information about which local states are active in a combined system state. For atomic state machines, the set of local states L is equivalent to the set of states, i.e., $L = \mathcal{S}$, and the function *local* is defined as $local(s) = \{s\}$.

We now turn to the concurrent execution of state machines and interactions between them, which are formalized using the standard parallel-composition operator.

Definition 2 (parallel composition)**.** *The* parallel composition *of two state machines* $\mathcal{M}_i = (\mathcal{S}_i, Act_i, \longrightarrow_i, s_i^{init}, L_i, local_i)$ *for* $i \in \{1, 2\}$ *with* $L_1 \cap L_2 = \varnothing$ *is defined as*

$$\mathcal{M}_1 \parallel \mathcal{M}_2 = \big(\mathcal{S}_1 \times \mathcal{S}_2, Act_1 \cup Act_2, \longrightarrow, \langle s_1^{init}, s_2^{init}\rangle, L_1 \cup L_2, local\big)$$

where $local(\langle s_1, s_2\rangle) = local_1(s_1) \cup local_2(s_2)$ *for* $s_1 \in \mathcal{S}_1, s_2 \in \mathcal{S}_2$ *and* $\longrightarrow$ *is the smallest transition relation fulfilling the following rules.*

$$\frac{s_1 \xrightarrow{\alpha_1}_1 s_1' \quad \alpha_1 \notin Act_2}{\langle s_1, s_2\rangle \xrightarrow{\alpha_1} \langle s_1', s_2\rangle} \qquad \frac{s_2 \xrightarrow{\alpha_2}_2 s_2' \quad \alpha_2 \notin Act_1}{\langle s_1, s_2\rangle \xrightarrow{\alpha_2} \langle s_1, s_2'\rangle}$$

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \quad s_2 \xrightarrow{\alpha}_2 s_2' \quad \alpha \in Act_1 \cap Act_2}{\langle s_1, s_2\rangle \xrightarrow{\alpha} \langle s_1', s_2'\rangle}$$

The parallel composition of two state machines is well-defined and yields again a state machine. The composition operator is both associative and commutative. Thus, to obtain the complete system state machine, the component state machines can be composed in any order, which always yields the same result (up to isomorphism).

The first two symmetric rules in Definition 2 realize the concurrent execution of the state machines via interleaving of transitions. That means there is a nondeterministic choice between either transitioning in the first state machine or in the second state machine. Intuitively, this nondeterminism may correspond, for instance, to external control inputs, environment changes, or different task scheduling. In case both state machines are ready to take a transition that is labeled with the same action α, they can take their respective transitions together and thus synchronize their execution, as stated in the third rule. Note that transitions with shared, i.e., synchronized, actions cannot be taken by one state machine alone without involving the other. Therefore, transitions with shared actions are blocked if the respective synchronization partner does not provide a matching transition with the same action.

3.2 Constraints

In an operational design, there are usually combinations of local states or modes that are not allowed under any circumstance. For instance, unfolding the solar panels of a satellite while it is still contained inside the launcher should generally be avoided. In order to describe such invalid state combinations, 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 $\bowtie$, 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 $\longrightarrow$, 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 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

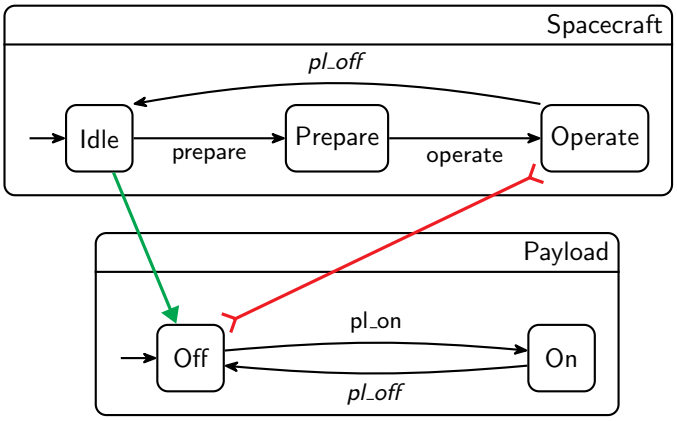

Fig. 1 Two state machines with constraints between them. States are denoted as rounded boxes and are connected by transitions.

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 3 (valid system state)**.** *Given a state machine* $\mathcal{M} = (\mathcal{S}, Act, \longrightarrow, s^{init}, L, local)$*, a system state* $s \in \mathcal{S}$ *is called* valid*, denoted as* $valid_{\mathcal{F},\mathcal{R}}(s)$*, 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. $\forall (p, q) \in \mathcal{F}. \neg\big(p \in local(s) \wedge q \in local(s)\big)$
2. $\forall (p, Q) \in \mathcal{R}. p \in local(s) \implies \exists q \in Q. q \in local(s)$

The semantics of a state machine under a set of constraints is given in terms of a transition system. Since there is no notion of forbid constraints and required constraints in transition systems, these constraints must be resolved. We assume that in the concrete implementation of the modeled system, the violation of the constraints is strictly prevented by construction, the inclusion of some control mechanism, or by operational procedures. Therefore, in the context of the formal semantics, we assume that those state combinations that violate some of the constraints are never entered in any run of the system. Thus, the semantics of the constraints can be embedded into the transition system by simply removing any invalid states from the state space. This is formalized in the following definition.

Definition 4 (transition system semantics)**.** *The behavior of a state machine* $\mathcal{M} = (\mathcal{S}_\mathcal{M}, Act_\mathcal{M}, \longrightarrow_\mathcal{M}, s^{init}_\mathcal{M}, L, local)$ *under a set of forbid constraints* $\mathcal{F}$ *and a set of required constraints* $\mathcal{R}$*, where* $valid_{\mathcal{F},\mathcal{R}}(s^{init})$ *holds, is defined as a transition system* $\mathcal{T} = (\mathcal{S}_\mathcal{T}, Act_\mathcal{T}, \longrightarrow_\mathcal{T}, \mathcal{S}^{init}_\mathcal{T})$ *where*

$$
\begin{aligned}
\mathcal{S}_\mathcal{T} &= \{\, s \in \mathcal{S}_\mathcal{M} : valid_{\mathcal{F},\mathcal{R}}(s) \,\} \\
Act_\mathcal{T} &= Act_\mathcal{M} \\
\longrightarrow_\mathcal{T} &= \{\, (s, \alpha, s') \in \longrightarrow_\mathcal{M} : valid_{\mathcal{F},\mathcal{R}}(s), valid_{\mathcal{F},\mathcal{R}}(s') \,\}
\end{aligned}
$$

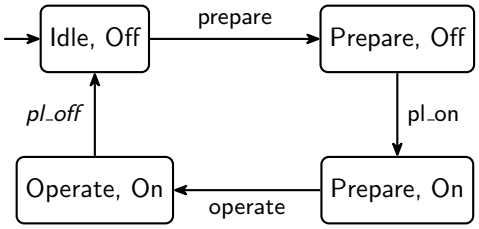

Fig. 2 Transition system semantics of the example in Fig. 1

$$\mathcal{S}_{\mathcal{T}}^{init} = \{\, s_{\mathcal{M}}^{init} \,\}$$

Consider again the example in Fig. 1. Its transition-system semantics after the parallel composition of the two state machines is presented in Fig. 2. Note that the action *pl_off* is shared by the state machines. Thus, in the resulting transition system both state machines change their respective local states on the transition from Operate, On to Idle, Off. Furthermore, in the system state Prepare, On, the Payload state machine cannot execute its transition labeled with *pl_off*, since the Spacecraft state machine does not provide the matching synchronizing transition in the Prepare state. Finally, we can observe that the synchronization of the two state machines is actually crucial under the given set of constraints. Supposing that the transition from Operate to Idle in the Spacecraft state machine were not labeled with the *pl_off* action, the system would contain a deadlock in the Operate, On state. 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.

3.3 Asynchronous Communication

The presented modeling formalism only supports synchronous communication between state machines, but not asynchronous communication where a sent message does not have to be received immediately but can be received at a later point in time. This may seem like a major limitation of the formalism. However, asynchronous message passing can be "implemented" using synchronous message passing by adding dedicated state machines that act as message buffers or queues, which has the advantage of flexibility regarding the choice of the concrete message-passing mechanism. The downside of the outlined approach to asynchronous communication is that the additional state machines have to be specifically defined for each model. However, since our verification approach relies on automated code generation anyway, this is not an issue.

Figure 3 shows an example of a state machine implementing a message queue of capacity 2, which can contain two possible message types a and b. Each state represents one possible configuration of the queue. Whenever a message is sent into the queue (using a "!"-marked transition), it is recorded by moving to the corresponding subsequent configuration. Likewise, receiving a message from the channel (using a "?"-marked transition) removes that message from the queue. Note that in this example, sending a message is blocked in case the queue is full. This is also the approach we took for in our analysis approach and evaluation. Depending on the context, it might also

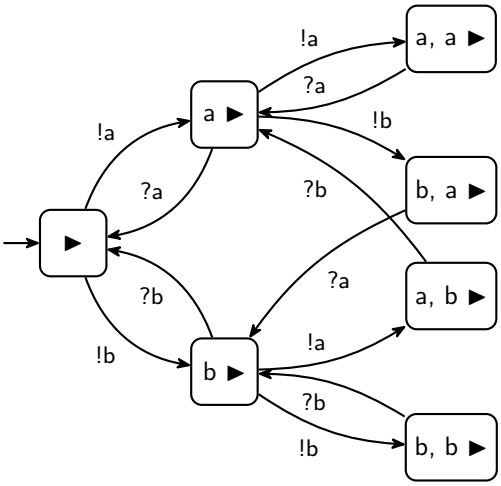

Fig. 3 State machine for a FIFO-channel with capacity 2 and possible messages a and b. The messages enter and leave from left to right (▶), where messages sent into the channel are prefixed with "!" and messages received from the channel are prefixed with "?".

make sense to drop sent messages in case of a full queue. This can be achieved easily by adding self-loops to all states on the right. The example can be straightforwardly extended to allow for more message types or a higher capacity. The former increases the width of the machine's tree-like structure, whereas the latter increases its depth. The size of the queue's state space is both exponential in the number of messages and the capacity, posing a significant challenge to the scalability of verification.

4 Transformation-based Analysis Approach

To enable the formal verification of the state-machine models described in Section 3, we have implemented a transformation from state machines into the modeling languages of selected modeling-checking tools. In particular, the tools SPIN (version 6.4.9), NuSMV (2.6.0), Prism (4.8.1), and Storm (1.9.0) 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 1. 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, as outlined in Table 2. 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 are required in case the state machine formalism is extended with quantitative properties. Since Markov Decision Processes (MDPs)

Table 1 Overview of the selected model-checking tools

Model Checker	Model Types	Modeling Languages
SPIN [39]	transition systems	PROMELA
NUSMV [40]	transition systems	SMV
PRISM [41]	Markov chains, Markov Decision Processes	PRISM language
STORM [42]	MCs, MDPs, Markov Automata	PRISM language, JANI, other

Table 2 Overview of the model-checking engines and internal model representations

Model Checker	Engine	Model Representation and Analysis
SPIN		explicit, partial-order reduction
NUSMV		symbolic using BDDs
PRISM	explicit	explicit
	mtbdd	symbolic using multi-terminal BDDs (MTBDDs)
	sparse	MTBDDs, sparse matrices
	hybrid	MTBDDs, sparse matrices
STORM	sparse	explicit, sparse matrices
	dd	MTBDDs
	hybrid	MTBDDs, sparse matrices

Table 3 Overview of the concepts natively supported by the modeling languages

Language	Concurrent Processes	Synchronization	Asynchronous Messages
PROMELA	✓	✓	✓(bounded channels)
SMV	✖	✖	✖
PRISM language	✓	✓	✖

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.

4.1 Transformation into Formal Modeling Languages

The modeling languages of the selected tools differ in the concepts that are supported natively (see Table 3 for an overview). For this reason, we opted to first transform a given set of state machines into an intermediate representation, namely *guarded-command programs*. Then, further transformation steps, such as the generation of additional model elements or the composition of programs, are added to account for missing language features. Finally, the guarded-command programs are converted into the concrete syntax of the modeling languages. The overall transformation approach is outlined in Fig. 4.

A guarded-command program is represented using an abstract syntax tree and does not have a concrete syntax yet. It comprises a set of bounded integer variables, representing the possible states, and a set of commands that describe the transitions

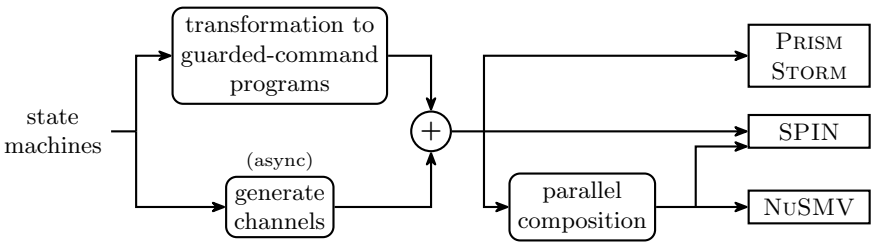

Fig. 4 Workflow for transforming a set of state machines into the input languages of selected model checkers

```
1  module spacecraft
2    // Idle: 0, Prepare: 1, Operate: 2
3    s_spacecraft : [0 .. 2] init 0;
4
5    [prepare] s_spacecraft = 0 -> (s_spacecraft' = 1);
6    [operate] s_spacecraft = 1 & s_payload != 0 -> (s_spacecraft' = 2);
7    [send_pl_off] s_spacecraft = 2 & s_payload = 0 -> (s_spacecraft' = 0);
8  endmodule
9
10 module payload
11   // Off: 0, On: 1
12   s_payload : [0 .. 1] init 0;
13
14   [recv_pl_on]  s_payload = 0 & s_spacecraft != 0 -> (s_payload' = 1);
15   [recv_pl_off] s_payload = 1 & s_spacecraft != 2 -> (s_payload' = 0);
16 endmodule
```

Listing 1 Corresponding PRISM model for the system shown in Fig. 1, transformed for asynchronous communication

between states. A command consists of an *action*, a *guard*, and an *update*, where the guard is a Boolean expression over the state variables. If the guard evaluates to `true` for the current values of the variables, the command may be executed, which updates the variables by assigning new values. A command's action serves the same purpose as the transition actions in Definition 1 and may be used for synchronization. The transformation also resolves the constraints between state machines by encoding them into the appropriate guards.

4.1.1 PRISM Language

For the PRISM language, the transformation of the abstract to the concrete syntax is straightforward. The resulting PRISM model for the example in Fig. 1 is shown in Listing 1. Each state machine and its equivalent guarded-command program corresponds to a `module` in the PRISM model (lines 1 and 10). The integer variable storing the machine's state (lines 3 and 12) is followed by the guarded commands, where each command corresponds to one of the state machine's transitions. For instance, the command in line 5 represents the transition from the Idle to the Prepare state, where

```
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 & next(sc) = Prepare & next(pl) = pl) |
9    (sc = Prepare & pl != Off & next(sc) = Operate & next(pl) = pl) |
10   (pl = Off & sc != Idle & next(pl) = On & next(sc) = sc) |
11   (sc = Operate & pl = On & next(sc) = Idle & next(pl) = Off);
12 CTLSPEC EF sc = Prepare;
13 CTLSPEC EF sc = Operate;
14 CTLSPEC EF pl = On;
15 CTLSPEC AG (sc = Idle -> EF sc = Prepare);
16 CTLSPEC AG (sc = Prepare -> EF sc = Operate);
17 CTLSPEC AG (sc = Operate -> EF sc = Idle);
18 CTLSPEC AG (pl = Off -> EF pl = On);
19 CTLSPEC AG (pl = On -> EF pl = Off);
```

Listing 2 Corresponding SMV model for the system shown in Fig. 1 with synchronous communication

the guard specifies the source state and the update assigns the successor state to the local variable. The encoding of the constraints into the guards prevents the execution of any command that would move the system into an invalid state. For example, moving from Prepare to Operate is forbidden in case the Payload is Off, which is captured by the condition `s_payload != 0` in line 6 (the symmetric guard is in line 15). The required constraint is encoded analogously (lines 7 and 14). The modules in a PRISM model are executed concurrently and synchronize over shared actions, thus no further transformations of the guarded-command programs are required.

4.1.2 SMV

Unlike PRISM and STORM, NUSMV does not provide a built-in mechanism for concurrently executing state machines that is compatible with our semantics. Therefore, it is necessary to perform the composition of the guarded-command programs into a single guarded-command program before the transformation into the SMV language (cf. the parallel-composition step in Fig. 4). The definition of parallel composition (Definition 2) can be lifted to guarded-command programs and proceeds as follows. Let P_1 and P_2 be programs that are composed into the program $P_{1\|2}$. All unsynchronized commands of both P_1 and P_2 are copied without modification to $P_{1\|2}$. The commands that share a common action are pairwise combined into a single command by taking the conjunction of their guards and by joining their updates. Note that the parallel composition on the level of guarded commands may lead to an exponential blow-up in the number of resulting commands, since for the synchronization all possible pairings of commands have to be considered. The result of composing the state machines from the example in

```
1  module chan_spacecraft_payload
2    // empty: 0, pl_off: 1, pl_on: 2
3    c_0 : [0 .. 2] init 0;
4    c_1 : [0 .. 2] init 0;
5
6    // if the channel still has capacity, shift content and add message
7    [send_pl_off] c_1 = 0 -> (c_0' = 1) & (c_1' = c_0);
8    [send_pl_on]  c_1 = 0 -> (c_0' = 2) & (c_1' = c_0);
9
10   // if the message is in the channel, remove it from end
11   [recv_pl_off] c_0 = 1 & c_1 = 0 -> (c_0' = 0);
12   [recv_pl_off] c_1 = 1 -> (c_1' = 0);
13   [recv_pl_on]  c_0 = 2 & c_1 = 0 -> (c_0' = 0);
14   [recv_pl_on]  c_1 = 2 -> (c_1' = 0);
15 endmodule
```

Listing 3 PRISM module for a channel with capacity 2

Fig. 1 and transforming them into SMV is presented in Listing 2. The transition relation (lines 8 to 11) is described by a single Boolean expression over the variables of the model and the updated variables of the successor (`next`) state. Transitions where only one of the state machines moves to its successor state are realized by only updating its variable and fixing all others. For instance, in line 8, only the Spacecraft state machine takes its transition, while the Payload state machine remains in its current local state (`next(pl) = pl`). The synchronization of both state machines over the `pl_off` action is realized in line 11, where both update their respective state simultaneously.

4.1.3 Promela

The transformation of guarded-command programs into Promela is, besides syntactical differences, very similar to the transformation into the Prism language. However, while Promela does provide built-in constructs for both synchronization and even asynchronous message passing, their semantics is slightly different from the one we defined in Section 3. In particular, synchronization that is conditioned on the current state of a state machine is not possible. Rather, the synchronization and evaluation of the guard that determines the current state are done sequentially, which introduces a lot of additional states. The authors are not aware of any workaround that allows us to synchronize and evaluate a guard in a single atomic step. Therefore, we decided to follow the same approach as for NuSMV, i.e., performing the composition beforehand, for all models that contain synchronization.

4.2 Code Generation for Asynchronous Communication

As described in Section 3.3, asynchronous communication between state machines is realized by generating additional guarded-command programs that implement the communication primitives. In this work, we opted for unidirectional point-to-point channels with a fixed capacity. Since each channel usually only needs to handle a few

```
label "SC_Idle" = (s_spacecraft = 0);
label "SC_Prepare" = (s_spacecraft = 1);
label "SC_Operate" = (s_spacecraft = 2);

// state Spacecraft.Prepare is reachable
E [ F "SC_Prepare" ];

// state Spacecraft.Operate is reachable
E [ F "SC_Operate" ];

// ...

// from Idle, Prepare is reachable
A [ G "SC_Idle" => E [ F "SC_Prepare" ] ];

// from Operate, Idle is reachable
A [ G "SC_Operate" => E [ F "SC_Idle" ] ];

// ...
```

Listing 4 Snippet of CTL properties corresponding to the system shown in Fig. 1

different message types, this approach potentially allows for a more compact model representation (in terms of system states) than a single shared message queue. An example for a generated channel between the Spacecraft and Payload state machines with capacity 2 and the possible messages `pl_on` and `pl_off` is shown in Listing 3. This module describes a FIFO-channel, where putting messages into the channel and taking messages from it is achieved via synchronization. Note that, besides the different action names, the corresponding state machine of this module is equivalent to the one shown in Fig. 3. Since channels are also generated as guarded-command programs (see again Fig. 4), they are handled the same as the programs arising from the transformation of the state machines in all subsequent steps, including the parallel composition and transformation into the concrete syntax of the modeling languages.

4.3 Generation of Specifications

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 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. In our implementation, the system must fulfill its specification for any possible environment. Formally this means that in each state where the system may receive a command, one of the enabled commands is chosen nondeterministically and sent to the system. This ensures that the system is free from deadlocks for any possible command sequence.

The temporal properties derived from the state-machine model are formulated in CTL. Consider again the example in Listing 2. In the SMV language, the properties are defined as part of the module description. The first three properties (lines 12-14) 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 12 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 15–19). 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 16 expresses that from every state, where the Spacecraft is in its Prepare state, a state where the Spacecraft is in Operate can be reached eventually. For PRISM and STORM, the properties are put in a separate file. An excerpt of the PRISM properties is shown in Listing 4. Apart from syntactical differences, the properties are equivalent to the ones described for the SMV model.

4.4 Integration into the MBSE Tool Virtual Satellite

The MBSE tool Virtual Satellite [7] provides both a graphical editor and a table-based editor for authoring state machines with constraints as described in Section 3. Furthermore, Virtual Satellite allows the user to extent its functionality using so-called *apps*, which are standard Java programs that are executed within Virtual Satellite and which have access to Virtual Satellite's data model, including the state machines, over an API. We implemented the previously described transformations into the model checker input languages as such an app. For the code generation, we utilized the built-in template expression mechanism of the XTEND language.

5 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) and the kind of message passing between state machines?

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

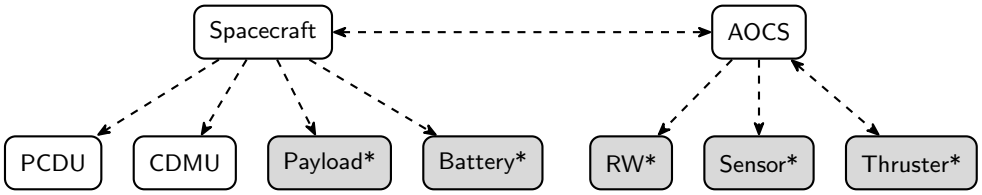

Fig. 5 High-level overview of the operational design. Each box represents a state machine and each dashed line represents a communication channel, where the arrow indicates the direction. State machines that are cloned during model scaling are filled in grey and are marked with an asterisk (*).

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 model can be instantiated for different kinds of message passing between the state machines. In particular, we have created the following model variants:

- **no comm.**: There is no communication, i.e., no message passing, between state machines.
- **sync**: State machines interact via synchronous messages.
- **async** n: Messages between state machines are sent asynchronously, where the capacity n of each channel size ranges from 1 to 3.

All variants of the 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 and, in case of asynchronous communication, with the channel capacity. The largest operational designs we consider consist of up to 254 state machines, as, from our experience, a design with more components is highly unlikely to arise during early design phases. For instantiating the operational designs, we have implemented an app in Virtual Satellite [7] that generates a set of state machines in Virtual Satellite's data model for a given size and communication kind. The generated operational designs have been transformed into the input languages of selected model-checking tools using the implementation outlined in Section 4.4 and were finally verified, measuring both time and memory consumption.

The evaluation follows the guidelines for performing empirical studies on formal methods [43] where appropriate. The generated models, scripts for running the experiments, and the measurement data is available online [44].

5.1 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 [45, 46], 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. In the model variants with message passing, some equipment modes are also switched by other state machines. The model is completely time-abstract and the latency of message passing is not quantified. A high-level architectural overview of the design is shown in Fig. 5. It comprises a Spacecraft state machine which captures the top-level mode, e.g., Launch or Nominal, a state machine describing the attitude and orbit control system (AOCS) as well as power

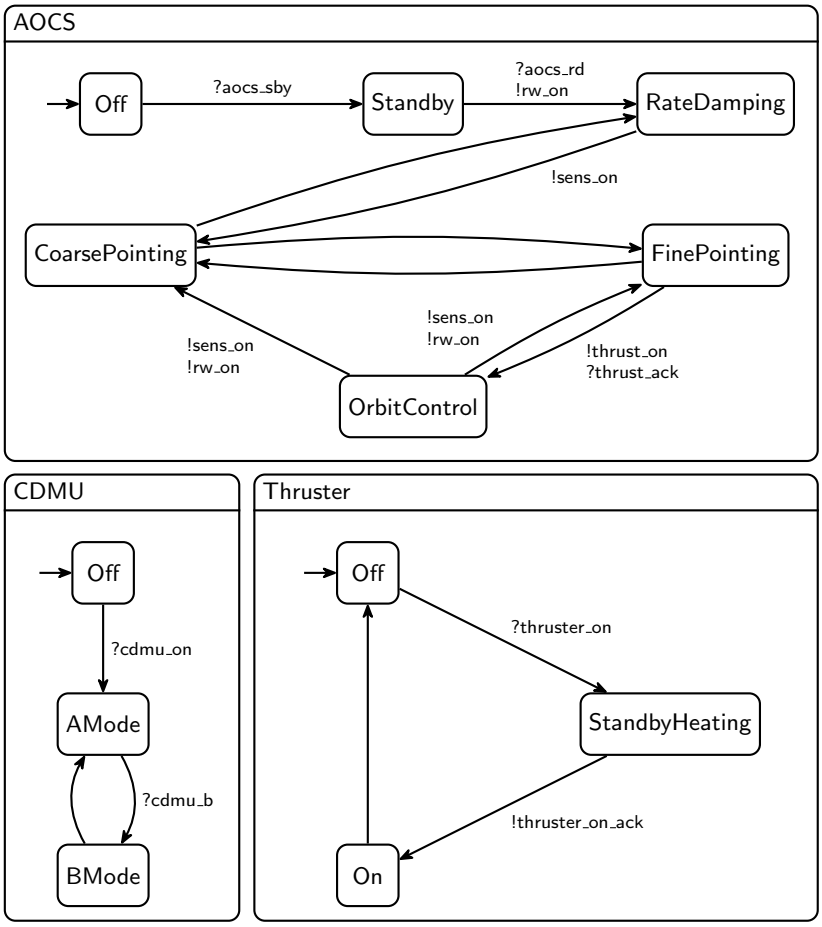

Fig. 6 The AOCS, CDMU, and Thruster state machines of the analyzed design. The constraints have been omitted for better readability.

control and distribution unit (PCDU) and command and data management (CDMU) state machines. Additionally, there are several auxiliary state machines for various equipment (payloads, batteries, reaction wheels (RW), sensors, and thrusters). A representative set of the state machines is presented in Fig. 6.

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 FinePointing mode of the AOCS is active and in the FinePointing AOCS mode the reaction wheels must not be turned Off, expressed using a forbid constraint. Generally, the states of the auxiliary state machines are constrained by the top-level mode as well as the AOCS state. However, there are no constraints between the auxiliary state machines.

In the model variants with message passing, the system state machine as well as the AOCS state machine may exchange messages with the PCDU, CDMU, and auxiliary state machines, as shown in Fig. 5. This is used, for instance, to model

the autonomous transition to a Safe mode. Switching to this mode sends messages to the other system components such that they either switch off or switch to their redundant backup component. Some transitions, like the transition from FinePointing to OrbitControl (cf. Fig. 6), require sending or receiving multiple messages. However, the modeling formalism only allows for sending at most one message on each transition. Therefore, such multi-message transitions are split into a chain of intermediate states and transitions, where each transition is only annotated with a single message.

The smallest instance of the design without communication comprises 9 state machines and its corresponding transition system has 3273 states.

Deadlock Detection and Repair

During the creation of the operational design, we detected and fixed multiple unintended and unexpected deadlocks using the implementation described in Section 4. In particular, the analysis was carried out on the smallest model instance. Since for this model size the results were produced almost instantly, fast iterations of modeling and analysis were possible. One class of defects we found was characterized by the interaction between forbid constraints and requires constraints, similar to the example described in Section 3.2. Another class of deadlocks was caused by a combination of synchronization and constraints. For instance, one state machine could only progress upon receiving a certain message, but the corresponding sender was blocked from actually sending the message due to a forbid constraint. While some of the deadlocks were caused by simple mistakes during modeling, e.g., because two state were erroneously connected by a constraint, other deadlocks revealed actual design errors. In these cases, the system was found to be over-constrained and removing the superfluous constraint fixed the deadlock. Note that 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.

Model Scaling

Scaling of the operational design to larger instances with more state machines is accomplished by cloning the auxiliary state machines, which are marked in grey in Fig. 5. For generating a design with $n+1$ state machines from a design with n state machines, one of the auxiliary 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. If the state machine can send or receive messages, copies of the corresponding message types are created and, in case of the model variants with asynchronous message passing, new instances of the corresponding channels are created as well. Note that adding an additional message type also implies that the other party involved in sending or receiving that message also must be modified, such that the message is actually sent or received. For instance, adding another state machine for a reaction wheel, which receives the rw_on message from the AOCS, also adds one state and one transition to the AOCS state machine. The selection of the next state

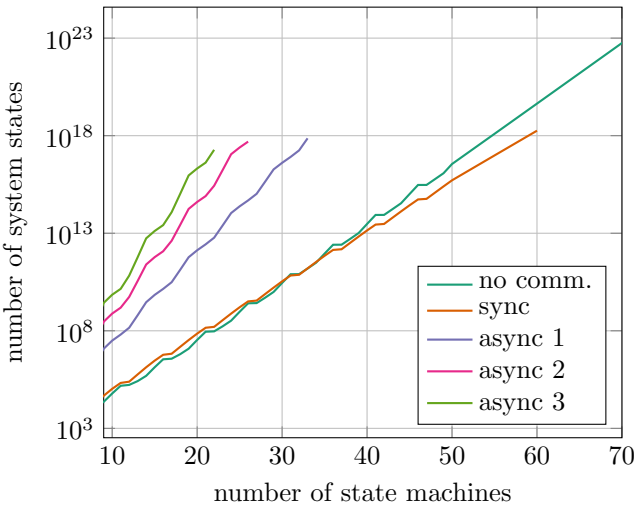

Fig. 7 Number of system states depending on the number of state machines for each model variant (no communication, synchronous communication, asynchronous communication with channel sizes 1 to 3).

machine to clone 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 auxiliary 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 connected to 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. Furthermore, since copying the constraints and communication channels does not introduce new constraint patterns, the scaling does also not introduce new deadlocks. However, we have also verified that this is actually the case by checking all model instances for deadlocks and no deadlocks were found. Since all the generated model instances are free of deadlocks, it is ensured that the whole state space is explored during model checking.

5.2 Data Collection Procedure

The model has been instantiated for a number of state machines ranging from 9 to 254. The resulting state-space sizes of all model variants are shown in Fig. 7. For better readability, the sizes upwards of 70 state machines have been omitted. The largest instance with 254 state machines of the model variant without communication has 5.9×10^{78} states. Note that in the model-checking community, even models with more than 10^8 states are considered as large-scale [33]. As expected, the exponential growth of the variants with asynchronous communication increases with the channel capacity. The variant with synchronous communication has the slowest rate of growth, since the synchronization prohibits some state combinations that are possible in all other variants.

Variable Ordering in Symbolic Model Checking

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 [47] to perform the BDD optimization.

Experiment Setup

We conducted two sets of experiments for measuring the time and memory requirements for the verification of all model variants 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 4) is checked.

Time and Memory Measurements

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, specifically the maximum resident set size, was measured using GNU `time`. 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 10 GB was reached. Both Prism and Storm implement multiple *engines*, which use different internal model representations and model-checking algorithms. For Prism, the hybrid and sparse engine show almost identical timing and memory characteristics to the mtbdd engine, so only the latter is presented in the following. For the same reason, the results for the hybrid engine of Storm are not shown. All experiments have been conducted on a workstation with an Intel Core i9-13900K and 128 GB RAM running Ubuntu 22.04 LTS.

5.3 Results

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

5.3.1 Analysis Time

We first consider the required time for deadlock checking of increasingly large models using different model checkers. The measurement results for the variant without communication are shown in Fig. 8. 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 24 (STORM sparse) state machines before running out of memory or 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 STORM 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. 9). 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. Furthermore, the time for transforming the models from their state-machine representation into the input languages of the model checkers is not included in the presented analysis times. For the largest analyzed instance (254 state machines), the transformation into the SMV model took 1.3 s, whereas the overall analysis took 50 s using NUSMV.

Introducing synchronous communication has a significant negative impact on the scalability, as shown in Fig. 10. Here, only STORM is able to check the largest instance, while PRISM and NUSMV hit the timeout much earlier. Since this model variant's state space is comparable to the variant without any communication, this indicates that synchronization significantly increases the size of the model's symbolic representation. Note that only PRISM and STORM natively support synchronization (as defined in Section 3) and thus allow for better scalability than NUSMV.

For the model variants with asynchronous communication, the scalability is much more limited. Using STORM, checking the instances with 37 (for a channel size of 1) to 26 (for a channel size of 3) state machines is still possible (see Fig. 11). Notably, for these variants the scalability is limited by the available memory rather than the time. Interestingly, for the verification of temporal properties in the variant with channel capacity 3, SPIN is able to outperform both PRISM and NUSMV, as shown in Fig. 12. The reason is that SPIN uses on-the-fly model checking, where the state space is explored during property checking, instead of exploring the whole state space beforehand.

Given these results, we can answer **RQ1**, concluding that an analysis of models with no communication or synchronous communication using *symbolic model checking* requires only minutes or tens of minutes even for the largest instances. For asynchronous communication, only small-to-medium sized models can be analyzed.

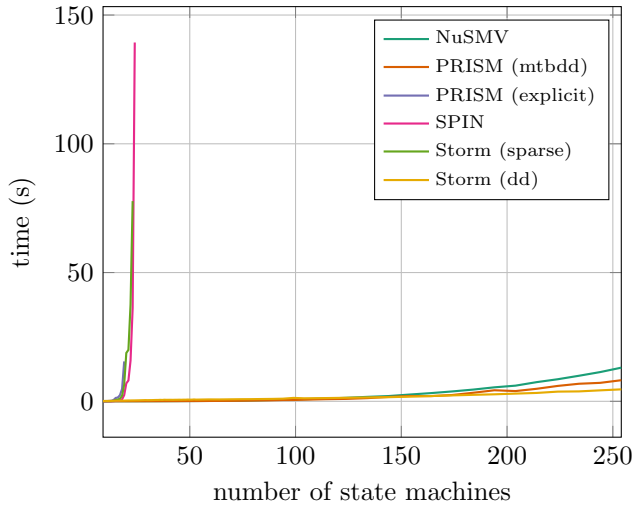

Fig. 8 Time needed for deadlock checking (no communication)

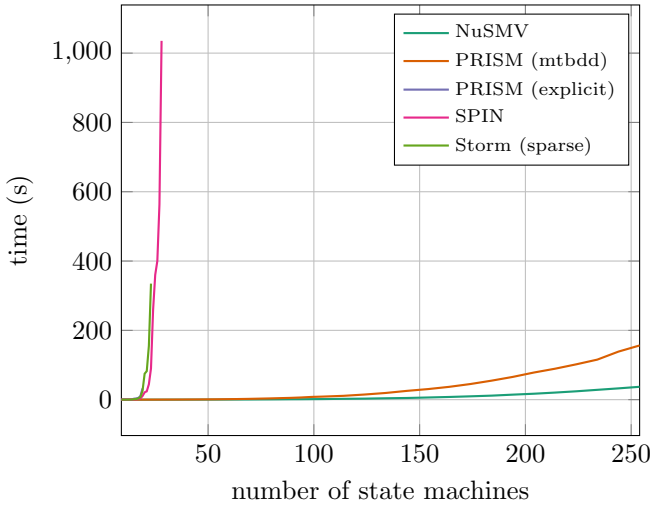

Fig. 9 Time needed for checking temporal properties (no communication)

5.3.2 Memory Usage

Figure 13 shows the peak memory usage for verifying the temporal properties depending on the state space size for the variants without communication. 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

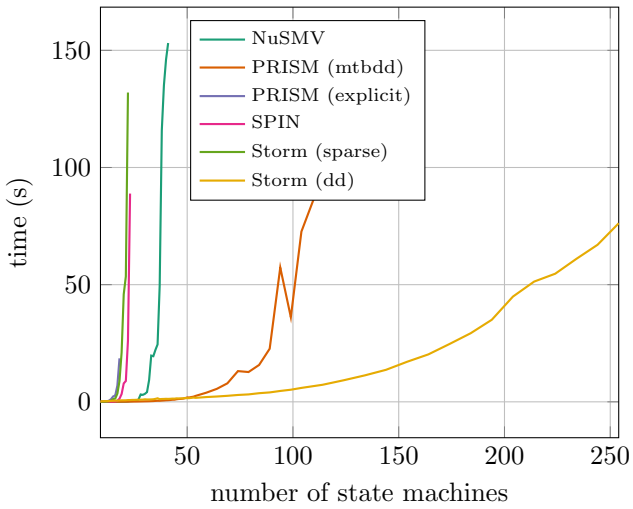

Fig. 10 Time needed for deadlock checking (synchronous communication)

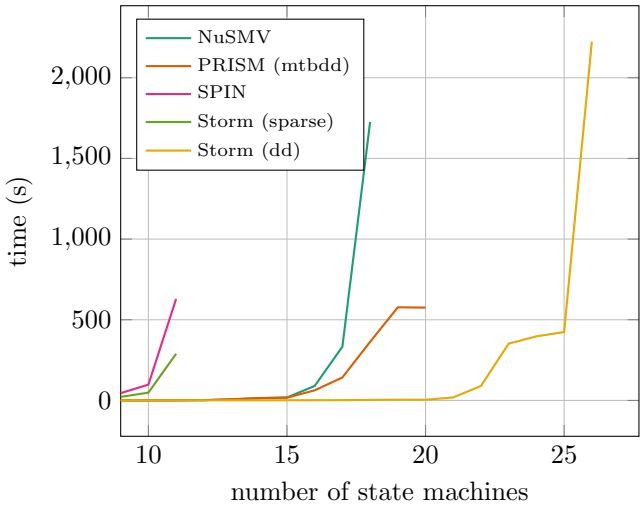

Fig. 11 Time needed for deadlock checking (asynchronous communication, channel size 3)

limited to 1 GB by default. However, increasing this limit did neither incur a higher memory usage nor did it reduce the analysis time.

The memory usage for models with synchronous communication shows similar characteristics (see Fig. 14), with the exception of NuSMV, which exhibits a much faster exponential growth. For asynchronous communication, the memory usage exceeded the limit even for medium-sized models.

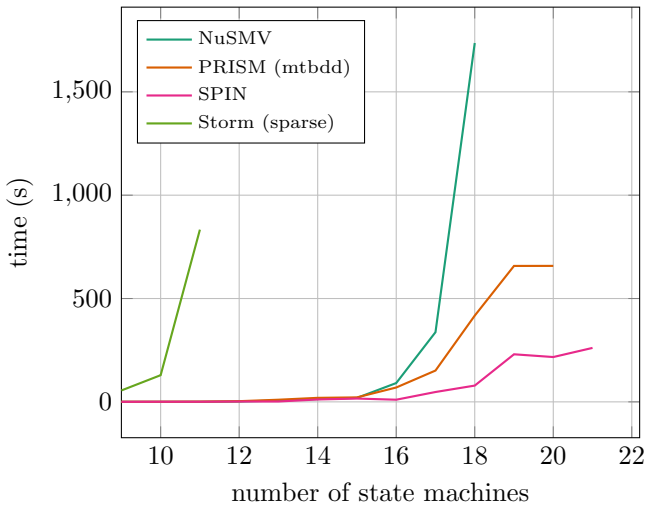

Fig. 12 Time needed for checking temporal properties (asynchronous communication, channel size 3)

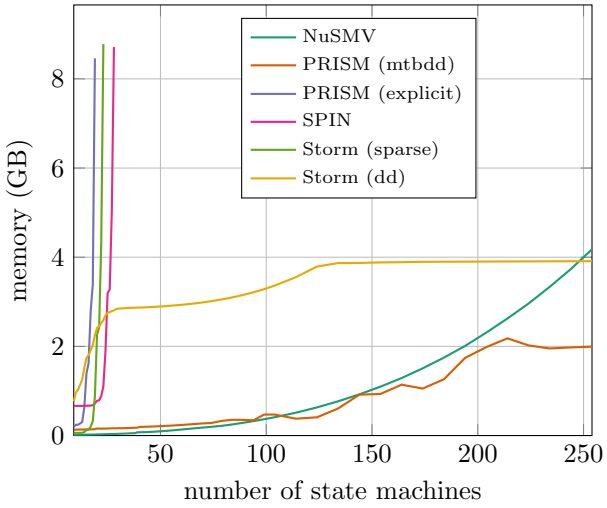

Fig. 13 Peak memory usage for checking temporal properties (no communication)

These results provide an answer for **RQ2**. For the considered operational design, 4 GB of memory are sufficient in case of no communication or synchronous communication between state machines. However, the analysis of models with asynchronous communication exceeded the memory limit of 10 GB even for medium-sized instances.

5.4 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

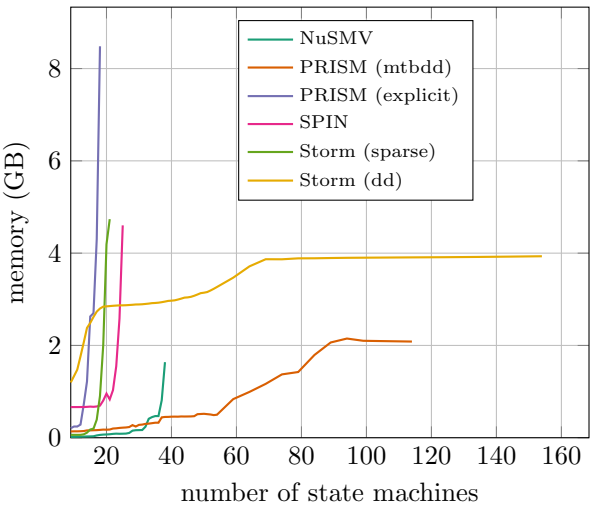

Fig. 14 Peak memory usage for checking temporal properties (synchronous communication)

for each analysis with a warm-up run beforehand. The maximum relative standard deviation between runs was 13.8%, 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. Note that the operational design analyzed in this paper is not the same as the one presented in the conference publication [4], but still exhibits similar time and memory requirements for the analysis. This indicates that the results are indeed generalizable to some degree for similarly structured models.

5.5 Discussion

The experiment results indicate that generally, the verification of operational designs using model checking is feasible. For models without communication between state machines, even large-scale models can be analyzed. The scalability of the approach is more limited in case of synchronous or asynchronous communication within the model, where only small- to medium-sized models were analyzable within the given time and memory limits. However, early operational designs with up to 250 different state

machines are the exception rather than the rule and in practice, such designs are often much smaller. Thus, even for designs that include message passing, the verification using model checking is applicable in most cases.

In contrast to our previous findings [4], where the model checker NuSMV outperformed all other tools for both deadlock checking and checking of temporal properties, the experiments in this paper did not establish a clear best tool for the verification and analysis in all scenarios. For deadlock checking, the symbolic engine (dd) of Storm was performing best in all cases. For the verification of the temporal properties, where Storm's symbolic engine is not fully applicable, NuSMV (for no communication) as well as Prism and SPIN (for models with communication) were favorable. This means that the implementation presented in Section 4 covering the transformation into the input languages of several model checkers is useful beyond the experimental evaluation, as it allows the selection of the most appropriate model-checking tool for a given verification or analysis task.

Given the comparatively low time requirements of the model-checking process, the verification can be potentially conducted even alongside concurrent engineering sessions (cf. Section 2.1). 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.

6 Related Work

In the following, we discuss the related work concerning the formal verification of behavioral models and the application of formal methods in the space domain.

6.1 Formal Verification of State-based Behavioral Models

A wide range of approaches and tools targeting the formalization and formal analysis of high-level state-machine models have been presented, in particular for Harel statecharts [37] as well as UML/SysML state machines. A recent survey by André et al. [48] provides a comprehensive overview of both formal semantics for UML state machines and transformation-based analysis approaches that leverage model checking for the formal verification. Both Fecher et al. [49] and Liu et al. [50] propose an almost complete formal semantics for UML state machines in terms of labeled transition systems, conceptually comparable to our definition in Section 3, but alternative characterizations, e.g., in terms of graph-transformation systems [51], are possible as well. While there are dedicated model-checking tools for UML state machines, e.g., presented in [52], many model-checking approaches are transformation-based, utilizing existing general-purpose model checkers. Examples range from the early work by Latella et al. [53], which employs SPIN to verify a subset of UML statecharts, to the recent approach proposed by Horváth et al. [54], which is based on the Gamma framework [55] and

utilizes UPPAAL [56] as well as THETA [57] to analyze a subset of SysML models, supporting constructs often found in industry-grade models. The tool presented by Kölbl et al. [58] is particularly related to our transformation approach (cf. Section 4), as it also targets PROMELA, SMV, and the PRISM language. However, in contrast to our approach, which (partially) performs the composition of the state machines before the transformation to ensure isomorphic models across all model checkers, their approach encodes the composition into the models, resulting in structurally different state spaces. The latter approach potentially results in much more compact representations of the transformed models, i.e., the size of the resulting PROMELA or SMV source code, but the resulting state space is usually larger due to the additional variables required in the model. Interestingly, only few works [58–61] utilized symbolic model checking, even though it has shown the most promise with regards to scalability in our evaluation. Furthermore, none of the aforementioned transformation-based approaches were evaluated regarding their scalability, but Horváth et al. mention it as a challenge [54].

Note that even though our formalism is not a subset of UML/SysML state machines in the strict sense, it still shares those constructs that have the most significant impact on the scalability of model checking for state machines, in particular message passing/events and concurrent execution. Thus, we expect that the scalability results presented in Section 5 should generalize to the analysis of UML/SysML state machines.

6.2 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 [39] has been utilized to verify a dually redundant spacecraft controller [62], the downlink module, sequencing module [63] as well as the multi-threaded plan execution module of the Deep Space One's flight software [64], critical parts of the Mars Science Laboratory's software [65], and parts of a launch vehicle's on-board computer software [66]. Brat et al. analyzed an autonomous docking system [67] using the LTSA model checker [68]. In [45], an Attitude and Orbit Control System (AOCS) is verified using the symbolic model checker NUSMV [40] 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 [69]. They apply the COMPASS tool set [70] for various analyses. In [71], the control software of the CubETH nanosatellite is verified using NUXMV [72], 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 [73] using the probabilistic model checker PRISM [41]. PRISM is also utilized in [74] to analyze the reliability, availability, and maintainability of a satellite system. The approach presented in [46] focuses on the concurrency and interactions of components in a satellite's mode management. Here, the CAAL [75] tool is utilized for the verification. An application of hybrid model checking is presented by Chan and Mitra [76]. Here, the safety of an autonomous spacecraft rendezvous is analyzed using SPACEEX [25] 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, 3, 77]. Several of the above mentioned works ([46, 62, 66, 71, 73]) 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 [45, 63–66, 69] focus on the verification of flight software which is typically developed during phases C and D, the approaches in [46, 62, 71, 73, 76] are applicable during earlier project phases. Only few of the cited works state that the verification activities were actually performed alongside the development [65, 69] 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.

7 Conclusion

We have examined 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 where interactions between state machines are captured by synchronous or asynchronous communication. 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 and furthermore allows a selection of the complexity of interactions between 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 for models, where dependencies and interactions between state machines are only expressed using constraints, symbolic model checking scales up to large-scale systems with up to 250 state machines. For more complex models that include synchronous or asynchronous communication, the scalability is more limited, but the analysis of medium-sized models with up to 30 state machines is still tractable. 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 modeling formalism could be extended to enable a reasoning about quantitative properties, such as energy consumption or reliability. Since the usage of synchronous or asynchronous communication has a significant impact on the scalability of the present approach, automated techniques for minimizing channel capacities while preserving the relevant system properties could be investigated. The current implementation of the approach utilizes the tool Virtual Satellite [7] for creating and transforming the state-machine models. This implementation could be fully integrated into the tool or also integrated

into other design tools, 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] M. Gleirscher and D. Marmsoler, "Formal methods in dependable systems engineering: a survey of professionals from Europe and North America," *Empir. Softw. Eng.*, vol. 25, no. 6, pp. 4473–4546, 2020.

[3] 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.

[4] P. Chrszon, P. Maurer, G. Saleip, S. Müller, P. M. Fischer, A. Gerndt, and M. Felderer, "Applicability of model checking for verifying spacecraft operational designs," in *26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023, Västerås, Sweden, October 1-6, 2023*. IEEE, 2023, pp. 206–216.

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

[6] 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.

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

[8] 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/

[9] 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.

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

[11] 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/

[12] 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.

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

[14] 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/

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

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

[37] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[38] J. Rey. Modeling with vsee: Definition of guidelines and exploitation of the models. [Online]. Available: https://www.vsd-project.org/download/documents/YGT%20final%20report%20Rey%20V2.pdf

[39] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Software Eng.*, vol. 23,

no. 5, pp. 279–295, 1997.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] P. Chrszon, "Model Checking of Spacecraft Operational Designs: A Scalability Analysis - Artifact," Oct. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.13998669

[45] 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.

[46] 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.

[47] 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.

[48] E. André, S. Liu, Y. Liu, C. Choppy, J. Sun, and J. S. Dong, "Formalizing UML state machines for automated verification – a survey," *ACM Comput. Surv.*, vol. 55, no. 13s, jul 2023.

[49] H. Fecher, M. Kyas, and J. Schönborn, "Semantic issues in UML 2.0 state machines," Christian-Albrechts-Universität Kiel, Tech. Rep., 2005.

[50] S. Liu, Y. Liu, É. André, C. Choppy, J. Sun, B. Wadhwa, and J. S. Dong, "A formal semantics for complete UML state machines with communications," in *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*, ser. Lecture Notes in Computer Science, E. B. Johnsen and L. Petre, Eds., vol. 7940. Springer, 2013, pp. 331–346.

[51] D. Varró, "A formal semantics of UML statecharts by model transition systems," in *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, ser. Lecture Notes in Computer Science, A. Corradini, H. Ehrig, H. Kreowski, and G. Rozenberg, Eds., vol. 2505. Springer,

2002, pp. 378–392.
[52] K. Zurowska and J. Dingel, "Language-specific model checking of UML-RT models," *Softw. Syst. Model.*, vol. 16, no. 2, pp. 393–415, 2017.
[53] D. Latella, I. Majzik, and M. Massink, "Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker," *Formal Aspects Comput.*, vol. 11, no. 6, pp. 637–664, 1999.
[54] B. Horváth, V. Molnár, B. Graics, Á. Hajdu, I. Ráth, Á. Horváth, R. Karban, G. Trancho, and Z. Micskei, "Pragmatic verification and validation of industrial executable SysML models," *Syst. Eng.*, vol. 26, no. 6, pp. 693–714, 2023.
[55] B. Graics, V. Molnár, A. Vörös, I. Majzik, and D. Varró, "Mixed-semantics composition of statecharts for the component-based design of reactive systems," *Softw. Syst. Model.*, vol. 19, no. 6, pp. 1483–1517, 2020.
[56] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, "UPPAAL 4.0," in *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA*. IEEE Computer Society, 2006, pp. 125–126.
[57] T. Tóth, Á. Hajdu, A. Vörös, Z. Micskei, and I. Majzik, "Theta: A framework for abstraction refinement-based model checking," in *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, D. Stewart and G. Weissenbacher, Eds. IEEE, 2017, pp. 176–179.
[58] M. Kölbl, S. Leue, and H. Singh, "From SysML to model checkers via model transformation," in *Model Checking Software - 25th International Symposium, SPIN 2018, Malaga, Spain, June 20-22, 2018, Proceedings*, ser. Lecture Notes in Computer Science, M. Gallardo and P. Merino, Eds., vol. 10869. Springer, 2018, pp. 255–274.
[59] J. Dubrovin and T. A. Junttila, "Symbolic model checking of hierarchical UML state machines," in *8th International Conference on Application of Concurrency to System Design (ACSD 2008), Xi'an, China, June 23-27, 2008*, J. Billington, Z. Duan, and M. Koutny, Eds. IEEE, 2008, pp. 108–117.
[60] M. E. B. Gutiérrez, M. Barrio-Solórzano, C. E. C. Quintero, and P. de la Fuente, "UML automatic verification tool with formal methods," in *Proceedings of the Workshop on Visual Languages and Formal Methods, VLFM 2004, Rome, Italy, September 30, 2004*, ser. Electronic Notes in Theoretical Computer Science, M. Minas, Ed., vol. 127, no. 4. Elsevier, 2004, pp. 3–16.
[61] V. S. W. Lam and J. A. Padget, "Symbolic model checking of UML statechart diagrams with an integrated approach," in *11th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2004), 24-27 May 2004, Brno, Czech Republic*. IEEE Computer Society, 2004, pp. 337–347.
[62] 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.
[63] P. Gluck and G. Holzmann, "Using SPIN model checking for flight software verification," in *Proceedings, IEEE Aerospace Conference*, vol. 1, 2002, pp. 1–1.

[64] K. Havelund, M. R. Lowry, and J. Penix, "Formal analysis of a space-craft controller using SPIN," *IEEE Trans. Software Eng.*, vol. 27, no. 8, pp. 749–765, 2001.
[65] G. J. Holzmann, "Mars code," *Commun. ACM*, vol. 57, no. 2, pp. 64–73, 2014.
[66] 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.
[67] 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.–.
[68] J. Magee and J. Kramer, *State models and Java programs*. wiley Hoboken, 1999.
[69] 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.
[70] 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.
[71] 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.
[72] 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.
[73] 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.
[74] 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.
[75] 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.
[76] 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.

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