Model Checking for Formal Verification of Space Systems

Michał Kurowski[1], Rafał Babski [1], Steve Duncan [2], Maxime Perrotin[3], Matt Webster [4]

1) N7 Space sp. z o.o. 2) Thales Alenia Space UK, 3) European Space Agency, 4) Liverpool John Moores University

Abstract

The goal of the presented activity is to integrate an existing model checking engine – SPIN[1] – with the TASTE[2] MBSE environment. For this purpose, the modelling languages used in TASTE – ASN.1, AADL and SDL need to be translated into PROMELA, a language for modelling and verification of concurrent systems. This paper describes the current achievements of the activity – proposal of a TASTE model checking workflow, formalization of requirement specification and established PROMELA translation patterns. Finally, the development of SDL models for validation of the tools and exploration of their utility in the design of space systems is discussed.

Introduction

TASTE is ESA's MBSE-based development environment for heterogenous, embedded, real-time systems. At its core it uses standard modelling languages for architecture (AADL), data (ASN.1/ACN) and behaviour (SDL) definition. It provides the capabilities to design, implement, build, deploy and test embedded applications. It can be connected with higher-level MBSE solutions (such as Capella)[3] and integrated into traditional/hybrid workflows[4].

The formalized, unambiguous high-level abstraction provided by the used modelling languages enables various analyses, simulation and model checking. Model checking is a method of verifying that a finite-state model of a system satisfies a given specification. In simple terms, it can be thought of as generalized testing, in which all possible system behaviours are automatically explored and verified against the defined requirements, or system properties. These properties are usually categorized into two groups: safety ("nothing bad ever happens") and liveness ("something good eventually happens").

The automatic exploration of all the possible behaviours may discover unforeseen issues and reduce the effort of manual or automated testing, e.g., by supporting the generation of test scenarios. However, it is not intended to replace the testing altogether – not all requirements can be expressed in a way compatible with model checking or in an efficient way. Due to the exhaustive nature, model checking may require significant computational power, memory or execution time. Additionally, as it operates on models, it cannot exercise the final system in its operational (or other concretized) environment. Model checking should be perceived as a complementary measure for ensuring high reliability and high degree of correctness of a system.

Despite the potential benefits, TASTE currently lacks a full-fledged model checker. The goal of the presented activity is to integrate a state-of-the-art model checker engine with the TASTE environment. As space systems have to operate in harsh environments and are (despite dedicated hardware hardening) prone to various kinds of hardware faults (e.g., memory corruption, interference), the model checking engine is to be extended with the capabilities to inject errors into the models, so that Fault Detection, Isolation and Recovery (FDIR) procedures can be properly verified.

Tools and Workflow

Two approaches to model checking have been considered and analysed: development of a native model checker or integration of an existing one. ESA expressed their explicit preference for the latter. Several state-of-the art model checkers were evaluated (e.g., PRISM, nuXmv and UPPAAL) and SPIN was selected.

The main reasons were broad and active community, abundance of training material, good performance and compatible licensing, important for integration with TASTE.

SPIN (Simple PROMELA Interpreter) is an open-source explicit state model checker. It achieves state of the art performance by transforming a PROMELA model into an executable C program that performs the actual state exploration. PROMELA is a formal language tailored to high-level modelling of concurrent systems. Its semantics introduces the concept of "non-deterministic" execution – scheduling (the order of process activation for a given scenario is non-deterministic) and decisions (some constructs allow several simultaneous code paths to be taken, only one of which is evaluated in a given scenario). However, as multiple scenarios are exercised to exhaustively explore the state space, in the end all possible orderings and decisions are evaluated. SPIN accepts a dialect of PROMELA that allows for embedded C code, enabling workarounds for certain PROMELA deficiencies.

Model is expressed in TASTE using AADL/XML Interface View, ASN.1 data model and SDL behaviour. Environment is defined implicitly by the Interface View and data model, which can be additionally refined to reduce the possible state space. The presented activity introduces 3 possible ways of formally (in an unambiguous, machine readable and processable way) defining the requirements: Stop Conditions, Verification Message Sequence Charts (MSC) and Observers. Stop conditions are simple Boolean expressions supporting a small subset of Linear Temporal Logic (LTL): never, always and eventually clauses. Verification MSC express desired or undesired communication patterns. Observers are extended SDL state machines capable of both observing and manipulating the entire system state. Observers can be triggered both by signal exchanges (both before and after signal processing by the recipient) and system state changes (detected via continuous signals).

A simplified version of the proposed workflow is as follows:

- Define an iteration of the model structure, so that a framework (e.g., signals, states, data structures) is available for the formalization of requirements.
- Formalize the requirements by translating the informal ones into Stop Conditions, Observers and Verification Message Sequence Charts.
- Define an iteration of model behaviour using SDL.
- Define a model checking scenario by selecting the relevant subset of the model under verification and applicable requirements, as well as optionally refining input vector generation and preparing additional Observers transitioning the model into the desired initial state.

The presented activity aims to translate the TASTE Interface View, ASN.1 and SDL directly into PROMELA. As the major involved components of TASTE (SpaceCreator and OpenGEODE) are open source, it is possible to integrate the solution with the existing codebase, re-using the facilities that are already implemented.

The assembly of the PROMELA model, combining the translation of the original TASTE system and the applicable formalized requirements, is to be performed by SpaceCreator, as it is the node integrating most of the relevant data – base architecture, data model, MSCs, as well as the relevant subset selections. The only data not available directly to SpaceCreator is SDL behaviour specification, which is to be translated into PROMELA directly within OpenGEODE. The assembled PROMELA model is then to be explored via SPIN model checker, according to the settings configured for the given scenario.

Translation into PROMELA

The project is currently in the prototyping phase. The formal requirement specification languages have been designed, both in terms of syntax and semantics. PROMELA generation is not yet developed, however, some patterns are already established (possibly subject to change along the project execution).

An example trivial model to be verified (see Figure 1) contains a Counter process (see Figure 2), which has two states (*ENABLED* and *DISABLED*), as well as an internal *value*, resetting after exceeding 10. It can receive two signals: *enable*, indicating the desired counter state, and *count*, carrying a delta to be applied to the internal value in case the counter is *ENABLED*. If it is *DISABLED*, its internal value shall always be 0. *DISABLED* is the starting state. The proposed PROMELA model, in line with TASTE runtime semantics, is as follows:

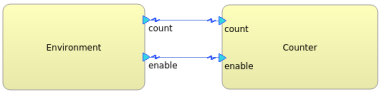

Figure 1 Model Interface View

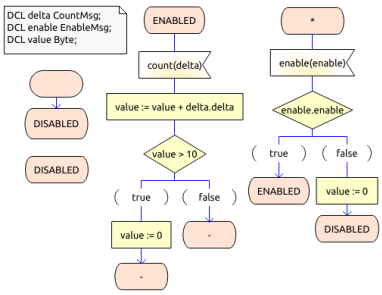

Figure 2 SDL specification of the Counter

- A globally accessible aggregated state, including the Counter's state and internal variables, such as *value*.
- Processes, one for each of SDL input signals (provided sporadic interfaces) – see Figure 3 for an example.
- Inlines defining Counter reactions (transitions) to the given signal in the given state.

Adherence to ASN.1 type constraints is evaluated after each assignment, using assertions. The environment is defined via processes, sending one of the allowed signal values per model exploration path (see Figure 4).

```
active proctype Counter_Count() priority 2 {
atomic{
        do
        ::count?counterCountMsg ->
            CountMsgAssign(model.counter.delta, counterCountMsg);
            if
                ::(model.counter.state == STATE_COUNTER_DISABLED) ->
                    Counter_DISABLED_Count(model.counter.delta);
                ::(model.counter.state == STATE_COUNTER_ENABLED) ->
                    Counter_ENABLED_Count(model.counter.delta);
            fi
            Observer_WAIT_Count_postdelivery(counterCountMsg);
        od
    }
}
```

Figure 3 Counter's Count signal reception process

```
active proctype Environment_Enable() priority 1 {
atomic{
    do
        ::if
            ::stop -> goto end;
            ::else -> skip;
        fi
        if
            ::envEnableMsg.enable = true;
            ::envEnableMsg.enable = false;
        fi
        Observer_WAIT_Enable_predelivery(envEnableMsg, observerResult);
        if
            ::observerResult.silenceSignal -> skip;
            ::!observerResult.silenceSignal -> enable!envEnableMsg;
        fi
    od
    }
    end: skip;
}
```

Figure 4 Environment's Enable signal sending process

The requirements for the example are specified via a single Stop Condition (Counter's *value* must be 0 in the *DISABLED* state), expressed directly via an LTL clause, which is evaluated alongside model exploration.

```
#define STOP_CONDITION_1 (model.counter.value == 0 || model.counter.state == STATE_COUNTER_ENABLED)
ltl sc1 {always STOP_CONDITION_1}
```

Additionally, an Observer is used for narrowing the state exploration by silencing certain signals. As Observers are not native to PROMELA, their logic needs to be injected into appropriate locations. Observers can react to a signal just after it has been sent (predelivery - thus having the capability to modify its value or silence it), or after it has been delivered and processed by the recipient (postdelivery). The above examples illustrate both predelivery (Environment_Enable in Figure 4, between signal value generation and submission to a channel) and postdelivery (Counter_Count in Figure 3, at the end of the loop) injections, the former including the logic for handling the possible silencing of a signal. As Observers are just extended SDL, their bodies are to be generated the same way as the inlines for regular SDL state machines.

The presented example has been exercised using SPIN. Two artificially injected bugs (lack of Counter's value resetting when exceeding 10 or on Counter disable) were successfully detected using the Stop Condition and type checking assertion. On the other hand, the existence of these bugs was successfully (and deliberately) masked by the Observer, which (post-delivery) stops model exploration when Counter's value is larger than 5 and (pre-delivery) silences the disable signals.

Demonstration Use Cases

A set of models is presented that serve as demonstration use cases for the model checker. These models are to be used to validate the model checker and explore its capabilities and limits, whilst providing representative practical examples that demonstrate the applicability of the model checking technique. Three model categories are foreseen, covering a range of sizes and complexity:

- Toy Model
- Subsystem Model
- Application Model

The Toy Model is a small model based on a common computational problem for which the behaviour, whether correct or otherwise, is already well understood and documented in computer science literature. It allows to perform end-to-end validation of the model transformation and model checking, with the output of the tools being compared against known results. A clear pass/fail criterion is therefore possible. The chosen Toy Model is an SDL implementation of Dekker's algorithm[5], including a correct version and defective variants that have known faults.

The Subsystem Model is a larger model with a narrowly scoped but possibly complex range of behaviour. It is chosen to be representative of standards-based subsystems that are commonly used in missions. It may contain known flaws or limitations that the model checker should be able to detect. Additionally, the Subsystem Model may contain flaws that are currently unknown, which the model checker may uncover. The choice of a common, standards-based subject for the Subsystem Model is key, as it allows consensus to be established, before the model checking process begins, that the design of the model is essentially correct. The chosen Subsystem Model is an SDL model of the ECSS-E-ST-50-15C CANopen protocol stack.

The Application Model is larger than the Subsystem Model but potentially less complex in terms of state space size. It is system-focused, being composed of static architectural blocks that represent major subsystems or, at the highest level, separate interacting systems. The Application Model is intended to be created early in the system engineering life cycle. Desirable features are that it should be accessible (easy to construct from scratch and for untrained users to gain a basic comprehension), informative (bringing value to the requirements analysis and design process through animation and scenario evaluation) and mutable (amenable to change and supporting the coexistence of alternative versions). The chosen Application Model is an SDL model of a control system for in-orbit fuel transfer between two spacecraft.

Conclusions and Future Work

Model checking workflow and languages for formal requirement definition have been specified in terms of syntax and semantics, thus establishing the base model checker capabilities and use-cases. SPIN has been evaluated and chosen as the backend. Translation into PROMELA of a trivial, yet complete TASTE system has been manually prototyped and exercised within the target environment. Scope of the demonstration use cases has been established. Further activities within the project will focus on the translation patterns, as well as the design and implementation of the automatic translator which will be integrated with Space Creator.

References

1) *The SPIN model checker: primer and reference manual*. Gerard Holzmann. 2003. Addison-Wesley.
2) *TASTE portal*. https://taste.tools/
3) *Capella to TASTE MBSE bridge*, M. Kurowski, A. Wójcik, M. Kocon, D. Kaczmara, B. Juszczyk, M. Mosdorf, H. P. de Koning, M. Perrotin. MBSE2020.
4) *TASTE in action*. M. Perrotin, K. Grochowski, M. Verhoef, D. Galano, M. Mosdorf, M. Kurowski, F. Denis, E. Graas. 8th ERTS 2016, Jan 2016, TOULOUSE, France.
5) *Over de sequentialiteit van procesbeschrijvingen (EWD-35).* Dijkstra, Edsger W. E.W. Dijkstra Archive. Center for American History, University of Texas at Austin.