Electronic Communications of the EASST
Volume 46 (2011)


# Proceedings of the 11th International Workshop on Automated Verification of Critical Systems (AVoCS 2011)

## A Symbolic Model Checking Approach to Verifying Satellite Onboard Software

Xiang Gan, Jori Dubrovin and Keijo Heljanko

15 pages

Guest Editors: Jens Bendisposto, Cliff Jones, Michael Leuschel, Alexander Romanovsky
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

# A Symbolic Model Checking Approach to Verifying Satellite Onboard Software

**Xiang Gan, Jori Dubrovin and Keijo Heljanko**

Aalto University
School of Science
Department of Information and Computer Science
PO Box 15400, FI-00076 Aalto, Finland
xiang.gan, jori.dubrovin, keijo.heljanko@aalto.fi

**Abstract:** This paper discusses the use of symbolic model checking technology to verify the design of an embedded satellite software control system called attitude and orbit control system (AOCS). This system is mission-critical because it is responsible for maintaining the attitude of the satellite and for performing fault detection, isolation, and recovery decisions of the satellite. An executable AOCS implementation by Space Systems Finland has been provided to us in Ada source code form. In order to use symbolic model checking methods, the Ada implementation of the system was modeled at a quite detailed implementation level using the input language of the symbolic model checker NuSMV 2. We describe the modeling techniques and abstractions used to alleviate the state explosion problem due to handling of timers and the large number of system components controlled by AOCS. The specification of the required system behavior was also provided to us in a form of extended state machine diagrams with prioritized transitions. These diagrams have been translated to a set of temporal logic properties, allowing the piecewise checking of the system behavior one extended state machine transition at a time. We also report on the scalability of symbolic model checking tools for the case study at hand as well as discuss potential topics for future work.

**Keywords:** symbolic model checking, AOCS, NuSMV 2, verification, satellite software

## 1 Introduction

Model checking [CGP99, BK08] is a technology where a formal model of a system's behavior is checked against its formal requirements often expressed in temporal logic. One of the main approaches in model checking is symbolic model checking using binary decision diagrams (BDDs) [BCM+92] that is especially suitable for hardware designs. Symbolic model checking is also suitable for analyzing other systems with a high branching degree due to environment non-determinism. Bounded model checking (BMC) [BCCZ99] was invented to scale symbolic model checking to analyzing even larger systems, especially for finding bugs in hardware designs with a large number of state bits. The basic idea in bounded model checking is to look for counterexamples to the specified property that are shorter than a user specified maximum

length called the bound. With this length restriction the search for counterexamples of bounded length can be reduced into propositional satisfiability (SAT), and the search can be performed by efficient SAT solvers, see e.g., [ES03].

The linear temporal logic (LTL) (see e.g., [BK08]) is a widely used temporal logic in model checking. Bounded model checking for LTL was shown to be linearly encodable to propositional satisfiability (SAT) in [HJL05, BHJ+06]. The same papers also describe a complete BMC algorithm that is guaranteed to terminate either with a counterexample or by proving the property holds but quite often requiring very high bounds in case the property holds. Both the incomplete and complete variants of this approach have been implemented in the NuSMV 2 model checker [CCG+02, HJL05, BHJ+06], and are used in the experiments of this paper.

This paper discusses the use of symbolic model checking technology to verify the design of an embedded satellite software control system called attitude and orbit control system (AOCS). Our approach is based on modeling an implementation given in Ada source code in the input language of the NuSMV 2 model checker [CCG+02].

The AOCS system is mission-critical because it is responsible for maintaining the attitude of the satellite, and for performing fault detection, isolation, and recovery decisions of the satellite. An executable AOCS implementation by Space Systems Finland has been provided to us in Ada source code form. In order to use symbolic model checking methods, the Ada implementation of the system was modeled at a quite detailed implementation level using the input language of the symbolic model checker NuSMV 2. We describe the modeling techniques and abstractions used to alleviate the state space explosion problem due to handling of timers and the large number of system components controlled by AOCS.

The specification of the required system behavior was also provided to us in a form of mode transition diagrams, which basically are extended state machines with prioritized transitions. These diagrams have been translated to a set of linear temporal logic (LTL) properties, allowing the piecewise checking of the system behavior one extended state machine transition at a time. We also report on the scalability of symbolic model checking tools for the case study at hand as well as discuss potential topics for future work.

We have also done earlier work on using model checking methods for verifying safety-critical systems in the nuclear safety area [BFV+09a, BFV+09b, VPB+08]. Also in that context the modeled systems are quite similar to the embedded mission-critical software considered here: the systems have a relatively large number of timers, as well as having to cope with a highly non-deterministic environment that includes a number of faulty components that have to be recovered from during the runtime of the system. In that domain, we have been using both NuSMV 2 as well as the Uppaal model checker [BDL+06]. Our experience is that NuSMV 2 is usually performing better for systems with a high branching degree (such as the AOCS system considered here), while Uppaal is performing much better for systems having a complex timing behavior. The main difference between the safety-critical and the mission-critical environments is that in the mission-critical systems, bug hunting methods (e.g., incomplete BMC based methods) can typically be seen as sufficient, while for safety-critical systems, the main focus is on complete system verification (complete model checking, e.g., BDD-based LTL model checking).

The same AOCS system has been used as a case study in the DEPLOY project, and modeling the AOCS system using refinement methodology in Event-B can be found in [ITL+10a, ITL+10b]. The concrete Event-B models described in these works can be found in [ILT10]. Our

model is very detailed and directly based on the Ada code implementation. Our motivation has been analyzing the correctness of the Ada implementation against its specifications, not to use refinement methodology to derive correct implementations. Thus our approach does not require significant changes in the software engineering methodology.

## 2 Attitude and Orbit Control System

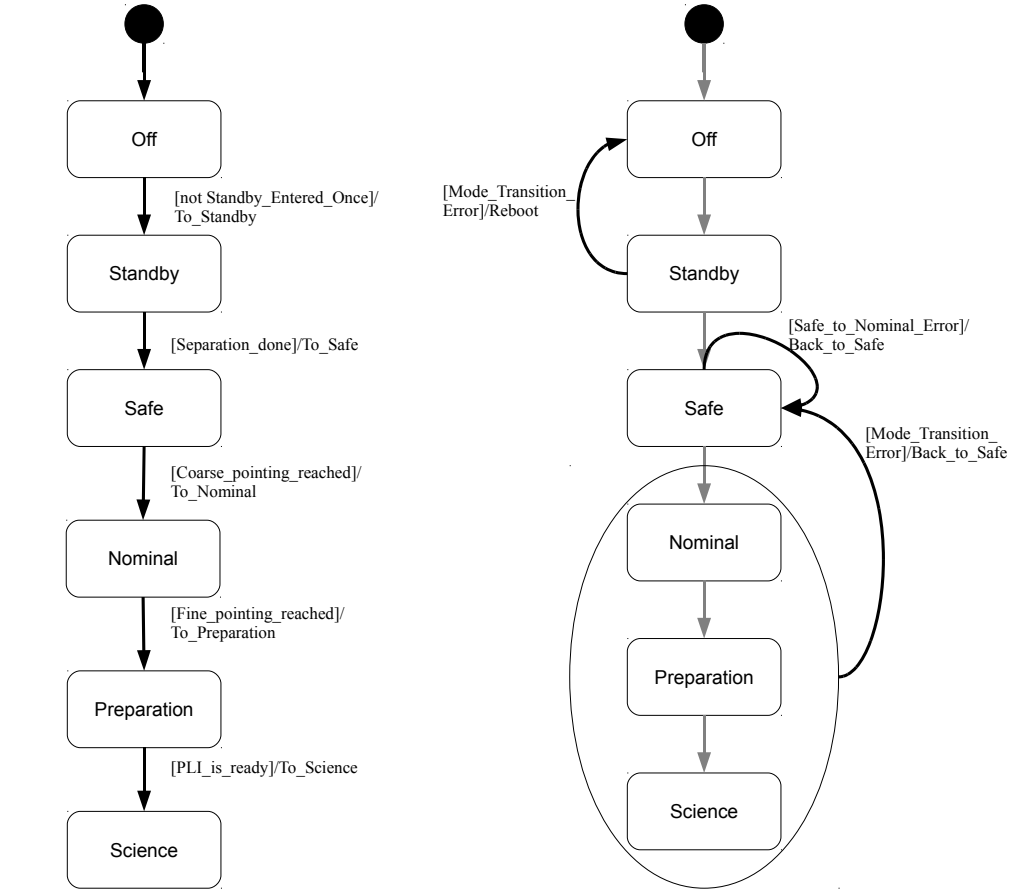
The Attitude and Orbit Control System (AOCS) [Var10] is a generic component of satellite onboard software. It is mainly used to determine and control the attitude of the spacecraft while it is in orbit. Since there is disturbance from environment, if left uncontrolled, the spacecraft will change its orientation. Because of this, its attitude needs to be monitored and adjusted continuously. The information from various sensors provides the necessary input for the AOCS computation. Based on this, the actuators are used to preserve or change the attitude or orbit of the spacecraft.

In AOCS, different software functionalities are realized by corresponding managers. There are four managers, AOCS manager, FDIR manager, Mode manager and Unit manager, which are executed in sequence to fulfill various functionalities. The AOCS manager has as its main responsibility to compute the attitude control algorithm. The FDIR manager (Fault Detection, Isolation and Recovery) is executed every time when new monitor data becomes available. There are three types of possible errors that are handled by the system: mode transition errors, attitude errors and unit errors. The Mode manager is in charge of mode transitions. In this AOCS implementation, there are six operational modes [Var10] listed below.

- **Off**. Normally, the spacecraft is in this mode after the system is booted.
- **Standby**. The system stays in this mode until separation from the launcher is completed.
- **Safe**. When the system is in this mode, it indicates that a stable attitude is obtained and the system endeavors to keep the coarse pointing control.
- **Nominal**. In this mode, the system is further trying to reach the fine pointing control.
- **Preparation**. The fine pointing control is reached and the unit Payload Instrument (PLI) is getting ready for the science mission.
- **Science**. The PLI is ready to carry out tasks. The overall goal is to stay in this mode as long as possible.

The normal mode transitions are shown in Figure 1(a) as a state diagram. In Figure 1(b), the bold arrows demonstrate the handling of mode transition errors in AOCS by the FDIR manager.

The Unit manager is used to manage unit resources, which encompasses avoiding conflicts in the usage of units and handling the units during their reconfiguration. There are in all seven different units: Earth Sensor (ES), Sun Sensor (SS), GPS, Star Tracker (STR), Reaction Wheel (RW), Thruster (THR) and Payload Instrument (PLI). The first four in this list are sensors, while RW and THR are actuators. The last unit, Payload Instrument, is a scientific measurement unit. In addition, there are two configurations, nominal and redundant, for each unit. Nominal unit




(a) Normal mode transitions. (b) Mode transition error handling.

Figure 1: Possible mode transitions in AOCS.

configuration is the start setup where the unit begins its operation. All of the units also have a redundant backup copy. The redundant unit is used when the nominal unit fails. Both the nominal and redundant units have two possible configurations, on or off, in the available operational modes.

## 3 Modeling and Verifying of AOCS

We modeled the previously described AOCS system in the input language of the NuSMV 2 model checker [CCG+02], starting from an Ada implementation. The system model is checked against a set of Linear Temporal Logic (LTL) properties that are generated from a specification given as extended state machine diagrams with prioritized transitions. Below, Section 3.1 describes the general modeling of the system, and Section 3.2 shows how the LTL properties are generated from the specification.

```
MODULE units
 VAR
   PLI : unit_t;

MODULE unit_t
 VAR
   br : branches;
   cubrid : {A_branch, B_branch};
   reconf_ongoing : boolean;
   orig_status : {free, locked};
   target_state : {none, pli_standby, pli_science};
   step : 0 .. 4;

MODULE branches
 VAR
   A_branch : branch_t;
   B_branch : branch_t;

MODULE branch_t
 VAR
   state : {none, pli_standby, pli_science};
   status : {free, locked};
   target_state : {none, pli_standby, pli_science};
   step : 0 .. 4;
   timer : {0, 3, 11, 16, 101};
   error : {none, timeout, loss_of_accuracy, invalid_data, commanding_failure};
   error_counter : boolean;
```

Figure 2: Modeling of the unit PLI.

## 3.1 Modeling the AOCS System

The current implementation of AOCS system is written in Ada, and the system is modeled in NuSMV 2 at a level of detail that closely follows the Ada source code. The main aspects of modeling are described below.

### 3.1.1 Modeling the Units

There are in all seven units in the system as described in Section 2. Incorporating so many units in a single model is likely to cause state explosion [Val96]. Our current solution to cope with this problem is to construct a concrete model of only one unit, the PLI, and to introduce an abstraction of other units. Specifically, as only the *error* property of other units is mentioned in the LTL formula to be checked, the abstraction omits the other aspects of the units. Thus, the basic strategy is to introduce one Boolean variable representing the *error* property of each unit that non-deterministically obtains its truth value at each time point modeling the fact that any subset of other units can generate an error for the software to handle at any time. By these means, the state space of the model is controlled to a reasonable size.

Figure 2 shows the data structures for modeling the PLI unit in the NuSMV 2 language. The module **units** accommodates the concretely modeled units. As discussed above, the PLI is currently the only unit fully modeled, but new units could be added to the variable list in this module as needed. The module **unit_t** defines the basic properties of a unit. Here, the variable *orig_status*

```
Procedure Execute_Step is
begin
  case Trans.Target is
    when Standby =>
      Transitions.To_standbyMode;
    when ...
  end case;
end Execute_Step;


Procedure To_StandbyMode is
  begin
    case Trans.Step is
      when 0 =>
        Trans.Step := Trans.Step + 1;
      when ...
    end case;
end To_StandbyMode;
```

(a) Source code sample snippet.

```
Module main
VAR
  label : {execute_step, to_standbymode};
  trans.target : {off, standby};
  trans.step : 0 .. 1;

ASSIGN
  init(label) := execute_step;
  init(trans.target) := standby;
  init(trans.step) := 0;

  next(label) := case
    label = execute_step &
          trans.target = standby: to_standbymode;
    TRUE : label;
    esac;

  next(trans.step) := case
    label = to_standbymode & trans.step = 0 : 1;
    TRUE : trans.step;
    esac;
```

(b) NuSMV 2 modeling code sample.

Figure 3: Modeling of a code snippet in NuSMV 2.

records the status of the use of the unit. The unit can be either free or locked. The variable *step* is used to record the number of steps it has used to make transition to a new operational state since the transition is usually a multi-step process. One of the interesting properties is *br*, which specifies the two configurations, nominal and redundant, represented as A and B in the model of the unit. This property is further illustrated in module **branches**, which uses the module **branch_t** to fully define the properties of each configuration of the unit. The functions manipulating the properties of units are modeled in detail as described in Section 3.1.2 below.

### 3.1.2 Modeling Ada Code

The overall structure of the Ada code is an infinite loop in which the four managers, AOCS manager, FDIR manager, Mode manager and Unit manager, are executed in sequence. All the actual functions and procedures defined in the Ada source code implementation are modeled as state labels in the NuSMV 2 model. The modeling of source code generally mimics the implementation. At this point, the translation is not yet done automatically. The core of this manual modeling is to treat each function entry point in the implementation as a potential program counter value and encode each function invocation as a single time step of the model. A similar idea of constructing models from program code is presented in [BCG+09] for C programs.

Figure 3 presents an example on how the source code is manually mapped to a NuSMV 2 model. Figure 3(a) is a part of the source code excerpted from the implementation with some irrelevant code removed. The working procedure of this sample code is quite straightforward. In procedure **Execute_Step**, it checks whether the value of variable *Trans.Target* is Standby. If this condition holds, then it calls the procedure **To_StandbyMode**. The latter procedure first checks the value of the variable *Trans.Step*. If the value of this variable is 0, then it will be increased to 1. Figure 3(b) is the corresponding model in NuSMV 2. It first defines the variables used in

the procedures. Note that it has defined an additional variable *label* that does not appear in the source code. Generally, this variable is used as a program counter to determine which procedure or function is being executed at present. Thus, *label* has the two procedure names, **execute_step** and **to_standbymode**, as its possible values. Next, these variables are initialized with initial values as shown by the **init** clause. Lastly, the **next** state transitions are defined for variables *label* and *trans.step*. These transitions in the model are exactly as those implemented in the source code. Variable *label* is used as an example to show how variable values are updated. In the **case** clause, it is first checked whether the current values of *label* and *trans.target* are **execute_step** and **standby**, respectively. If this holds, the value of *label* is updated to **to_standbymode**. Otherwise, the value is unchanged. Note that the possible transitions of variable *trans.target* are not presented here for the ease of explanation. Note also that the Ada code has no recursion which makes modeling simpler, as there is no need to explicitly model the stack of the program.

### 3.1.3 Timer Abstraction

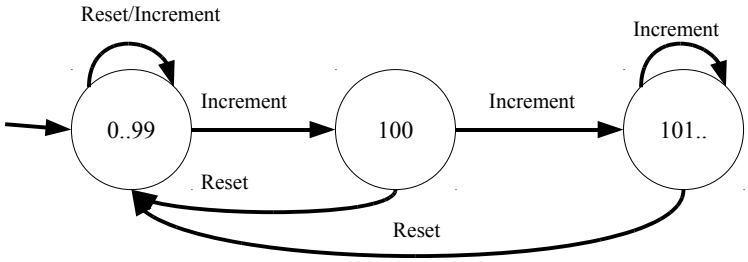
Besides abstracting entire units as described in Sect. 3.1.1, we apply a form of data abstraction. Most of the data structures in the implementation are modeled as they are, however, some of them are abstracted in the model. For instance, there are several timers in AOCS. The timers are mainly used to trigger certain events to occur and to record the timeouts of events in the Ada code. In the implementation, timers are defined as an integer type. If these were directly modeled as they are, then the modeling would become very expensive to analyze using NuSMV 2 due to state explosion. Therefore, we conduct an abstraction [CBKK94] of these timers that removes unnecessary details with non-deterministic choice in a sound way: the abstract model has more behaviors than the concrete one. If we are able to prove an LTL property for the abstract model, it will also hold in the concrete one.

As a simple example, suppose there is a timer initialized with value 0. Under normal conditions, this timer is incremented by 1 every time a periodic timer interrupt occurs. The timer is reset to 0 when a reset command is issued. Assume that increasing the timer to 100 triggers a special event. We create an abstraction of this timer as a state machine diagram shown in Figure 4(a). The value 100 of the timer is modeled as a separate state, while the values 0..99 and 101.. are collapsed into single abstract states, respectively. Especially note how non-determinism is used in state ”0..99” when the increment operation occurs to either stay in the same state or to go to the state ”100”. Such a timer abstraction can, of course, result in spurious counterexamples. However, in our case study, no spurious counterexamples are observed.

The NuSMV 2 model code for this timer is shown in Figure 4(b).

## 3.2 LTL Property Generation

The LTL properties that are used to check the constructed model are generated from the mode transition diagrams in the requirements document. In general, four state diagrams, normal mode transitions (Figure 1(a)), mode transition error handling (Figure 1(b)), attitude error handling and unit error handling, are used for the generation of LTL properties. The generation procedure of these LTL properties is automated from a tabular notation. The main steps are summarized below. First, the extended state machine diagrams with prioritized transitions are transformed




(a) Abstract states and transitions modeling the timer.

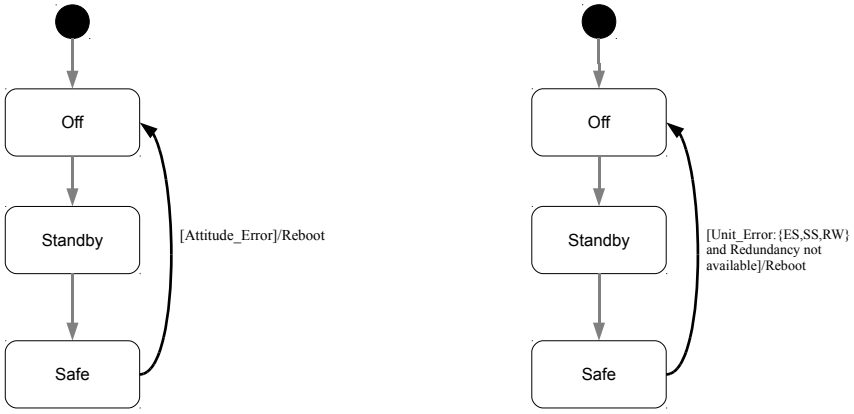
```
Module timer_xyz
VAR
  timer : {0, 100, 101};

ASSIGN
  init(timer) := 0;

  next(timer) := case
    reset_condition : 0;
    timer = 0 : {0, 100};
    timer = 100 : 101;
    TRUE : timer;
    esac;
```

(b) NuSMV 2 model code snippet for the abstract timer.

Figure 4: Abstraction of a timer.

manually to a tabular notation consisting of a priority list in which the possible mode transitions of each state are prioritized according to the specific mode. Next, a Python script is written which can read the priority list and generate the corresponding LTL properties from it.

Mode ”safe” is used below to demonstrate how its related LTL properties are automatically generated from the mode transition diagrams. First, there are three types of possible errors that can occur at a specific mode according to the previous description of the AOCS system. It might be possible that two or more errors are occurring at the same time. Thus, it is necessary to prioritize the possible errors as well as the normal mode transition so that the property can reflect the fact that the system is handling one type of mode transition at a time. According to the nature of the AOCS system, the priorities of possible state transitions are inferred as follows. The mode transition error will always have the highest priority. The priority of attitude error handling follows it. Unit error handling in turn follows the attitude error handling. Finally, normal mode transition has the lowest priority. In case of mode ”safe”, its state transition under the condition mode transition error can be found from Figure 1(b) while its related transition in the context of normal mode transition is depicted in Figure 1(a). Its mode transitions in case of attitude error and unit error are extracted from the related state diagrams and shown in Figure 5(a) and (b), respectively. Note that in Figure 5, the depicted mode transition diagrams are truncated in order




(a) Attitude error in Safe. (b) Unit error in Safe.

Figure 5: Attitude and unit errors handling in Safe mode.

Table 1: Priority list of mode transitions for Safe mode.

| Mode | Priority | Guard | Target mode | Atomic |
|---|---|---|---|---|
| safe | 1 | safe_to_nominal_error | safe | yes |
| | 2 | attitude_error | off | yes |
| | 3 | unit_error_ES_SS_RW and redundancy_not_available | off | yes |
| | 4 | coarse_pointing_reached | nominal | no |

to only highlight the modes involved in the transitions. The corresponding priority list of mode transitions can be constructed as shown in Table 1. In the column ”Priority” of Table 1, smaller integer indicates a higher priority of the specific mode transition.

The LTL properties related to a specific state can be generated automatically from the priority list composed in the former step. The core idea is to extract the information about current mode, guard and target mode from the list. All guards belonging to transitions of higher priorities than those of the current mode transition should be disallowed for a lower priority transition to be enabled. According to these, the following LTL formula is constructed as the template for the generation of all the properties.

$$\begin{aligned}&\mathbf{G}\left(\left((mode = CURRENT\_MODE) \wedge \neg HIGHER\_PRIORITY\_GUARDS \wedge GUARD\right) \rightarrow\right.\\&\quad \mathbf{X}\left((mode = CURRENT\_MODE)\,\mathbf{U}\,((mode = TARGET\_MODE) \vee\right.\\&\quad \left.\left.POSSIBLE\_HIGHER\_PRIORITY\_TRANSITION)\right)\right) \qquad (1)\end{aligned}$$

In the above formula, variables *CURRENT_MODE*, *HIGHER_PRIORITY_GUARDS*, *GUARD* and *TARGET_MODE* will be substituted by the concrete values extracted from the priority list. The variable *POSSIBLE_HIGHER_PRIORITY_TRANSITION* at the end of the template formula is needed for correct translation of mode transitions that are multi-step processes instead of atomic steps, as indicated in the column “Atomic” in Table 1. When we ran an earlier version of the experiments, we noticed an anomaly with some property that had an unexpected counterexample. We managed to trace this counterexample back to a mismatch of the levels of atomicity

between the mode transition diagrams and the Ada code implementation of the system. Namely, in the Ada implementation some of the state transitions are not atomic, and a lower priority mode transition needs to be aborted by the enabling of a higher priority transition "half-way through" a multi-step mode transition from one mode to the next. Such a mismatch between the specification and the implementation level of atomicity can possibly lead to subtle interpretations of the required behavior of the system, and our formal model checking was able to point such a case to us. The addition of *POSSIBLE_HIGHER_PRIORITY_TRANSITION* enables correct handling of aborted transitions.

As an example, the concrete LTL formulas generated from Table 1 are listed as follows.

$$\mathbf{G}\big(\big((mode = safe) \wedge (mode\_trans\_error \neq none)\big) \rightarrow \mathbf{X}\big((mode = safe)\,\mathbf{U}\,(mode = safe)\big)\big) \tag{2}$$

$$\mathbf{G}\big(\big((mode = safe) \wedge (mode\_trans\_error = none) \wedge (attitude\_error \neq none)\big) \rightarrow \mathbf{X}\big((mode = safe)\,\mathbf{U}\,(mode = off)\big)\big) \tag{3}$$

$$\mathbf{G}\big(\big((mode = safe) \wedge (mode\_trans\_error = none) \wedge (attitude\_error = none) \wedge unit\_error\big) \rightarrow \mathbf{X}\big((mode = safe)\,\mathbf{U}\,(mode = off)\big)\big) \tag{4}$$

$$\mathbf{G}\big(\big((mode = safe) \wedge (mode\_trans\_error = none) \wedge (attitude\_error = none) \wedge \neg unit\_error \wedge coarse\_pointing\_reached\big) \rightarrow \mathbf{X}\big((mode = safe)\,\mathbf{U}\,\big((mode = nominal) \vee (mode\_trans\_error \neq none) \vee (attitude\_error \neq none) \vee unit\_error\big)\big)\big) \tag{5}$$

Note that in formulas (3) and (4), the mode transition from **Safe** to **Off** is atomic according to the implementation. Thus, no additional higher priority transition needs to be added. By contrast, in formula (5), the mode transition from **Safe** to **Nominal** is a multi-step process in the implementation. The possible higher priority transitions, therefore, must be considered for the purpose of aborting handling.

It can be easily shown that these LTL formulas can be extended to related CTL formulas by simply adding the universal path quantifier **A**. Since CTL formulas are also checked in the following experiments, formula (2) is used as an example to show how CTL formula (6) can be obtained from the corresponding LTL formula.

$$\mathbf{AG}\big(\big((mode = safe) \wedge (mode\_trans\_error \neq none)\big) \rightarrow \mathbf{AXA}\big((mode = safe)\,\mathbf{U}\,(mode = safe)\big)\big) \tag{6}$$

## 4 Experimental Results

The experiment is carried out in a computing cluster environment with some background load. Each compute node in the cluster has two 6-core AMD Opteron 2345 CPUs and the memory

is 32 GB (about 2.5 GB per core). The cluster has in all 112 compute nodes. We use NuSMV 2.5.2 to model and verify the AOCS system. The code of the AOCS system is about 2200 lines of Ada while the NuSMV 2 model code is about 800 lines with only the unit PLI fully modeled. The model has roughly 80 variables and 25 state labels. The model currently has in total $2^{124.483}$ states, out of which 84.5 million states are reachable.

BDD-based LTL model checking is used in NuSMV 2 to check whether the generated LTL formulas hold or not. In addition, bounded model checking (BMC) is also used to find possible counterexamples. BMC is based on the reduction of the bounded model checking problem to a propositional satisfiability problem. NuSMV 2 internally invokes a propositional SAT solver to search for an assignment that satisfies the generated problem. In this experiment setup, NuSMV 2 is compiled to link to MiniSat [ES03], a high performance and open-source SAT solver. Specifically, the incremental BMC algorithm (`check_ltlspec_sbmc_inc`) [BHJ+06, ES03] is used in NuSMV 2 to check the generated LTL specifications. Since NuSMV's default BMC is incomplete, we also tried to supply the command line option *-c* that performs the completeness check [BHJ+06]. To obtain better performance results for BDD-based model checking, the value of the environment variable *partition_method* is configured as *Iwls95CP* instead of the default one. In general, this method is conjunctive partitioning with clusters generated and ordered according to the heuristic introduced in [RAB+95].

In this experiment, there are 28 LTL properties generated from the mode transition diagrams in the system according to the generation method previously described. In all the cases, the time bound set to check each LTL property is configured to be 30 minutes. The purpose is to make the checking time long enough so that it can go deeper into the state space and find possible counterexamples. CTL model checking is also carried out with the generated CTL properties as demonstrated in Subsection 3.2. In general, the CTL model checking can deliver almost the same amount of conclusive results as LTL model checking does. The used time, however, is about an order of magnitude slower than that of LTL model checking. For the BMC part, a very large integer is supplied as the bound, so that the check will only terminate when a timeout or memory out is reached, or when a counterexample is found. The experimental results are summarized in Table 2. The 28 properties are listed as P1 to P28 in the table.

In Table 2, the column "BDD_LTL" indicates the results using the NuSMV 2 BDD-based LTL model checking algorithm. **T** indicates that the property holds while **F** means that the property does not hold. The number after the forward slash is the time used by the checking and it is measured in seconds. T.O. means that the check exceeds the configured time bound. The column "BMC incomplete" represents the results using the incremental BMC algorithm while its neighbor "BMC complete" indicates the results using the same command but with command line option *-c* enabled so that it will perform the completeness check which also tries to prove the property holds. M.O. indicates that the check exceeds the memory limit. For P4', the numbers after the forward slash in these two columns indicate the time to find counterexamples. The number in the parenthesis is the step at which timeout, memory out is reached or a counterexample is found. For instance, the "BMC incomplete" column of property P1 states that the check is timed out at step 158. This means that there is no counterexample against P1 in 158 time steps. The table, as a whole, shows that the BMC without completeness check can reach larger bounds than its complete variant.

Table 2: Results of model checking LTL properties with partition method as Iwls95CP.

| Property | BDD_LTL | BMC incomplete | BMC complete | Property | BDD_LTL | BMC incomplete | BMC complete |
|---|---|---|---|---|---|---|---|
| P1 | **T**/101.59 | T.O.(158) | M.O.(141) | P15 | **T**/121.10 | T.O.(434) | M.O.(141) |
| P2 | **T**/100.99 | T.O.(434) | M.O.(141) | P16 | T.O. | T.O.(158) | T.O.(141) |
| P3 | **T**/102.78 | T.O.(158) | T.O.(113) | P17 | T.O. | T.O.(119) | T.O.(113) |
| P4 | **T**/79.82 | T.O.(119) | T.O.(113) | P18 | T.O | T.O.(119) | T.O.(113) |
| P4' | **F**/125.70 | **F**(61)/5.23 | **F**(61)/22.28 | P19 | T.O. | T.O.(119) | T.O.(113) |
| P5 | **T**/99.76 | T.O.(434) | M.O.(141) | P20 | T.O. | T.O.(119) | T.O.(113) |
| P6 | **T**/1133.09 | T.O.(119) | T.O.(113) | P21 | T.O. | T.O.(119) | T.O.(113) |
| P7 | **T**/1434.05 | T.O.(119) | T.O.(85) | P22 | **T**/120.53 | T.O.(434) | M.O.(141) |
| P8 | T.O. | T.O.(119) | T.O.(85) | P23 | T.O. | T.O.(158) | T.O.(141) |
| P9 | **T**/105.49 | T.O.(434) | T.O.(141) | P24 | T.O. | T.O.(119) | T.O.(113) |
| P10 | T.O. | T.O.(119) | T.O.(113) | P25 | T.O. | T.O.(119) | T.O.(113) |
| P11 | T.O. | T.O.(119) | T.O.(113) | P26 | T.O. | T.O.(119) | T.O.(113) |
| P12 | **T**/1205.76 | T.O.(119) | T.O.(113) | P27 | T.O. | T.O.(119) | T.O.(113) |
| P13 | T.O. | T.O.(119) | T.O.(113) | P28 | **T**/78.47 | T.O.(198) | T.O.(141) |
| P14 | T.O. | T.O.(119) | T.O.(113) | | | | |

Let us study in detail the property P4, which has the LTL representation

$$\begin{aligned}&\mathbf{G}\big(\big((mode = standby) \wedge (mode\_trans\_error = none) \wedge (attitude\_error = none) \wedge \\ &\quad separation\_done\big) \rightarrow \\ &\quad \mathbf{X}\big((mode = standby)\,\mathbf{U}\,((mode = safe) \vee (mode\_trans\_error \neq none) \vee \\ &\qquad\qquad (attitude\_error \neq none))\big)\big). \end{aligned} \tag{7}$$

The formula contains the condition *POSSIBLE_HIGHER_PRIORITY_TRANSITION*, as discussed in Section 3.2 above, to enable aborting the multi-step transition by a higher-priority transition. A previous version of the property, denoted by P4' in Table 2, has the LTL representation

$$\begin{aligned}&\mathbf{G}\big(\big((mode = standby) \wedge (mode\_trans\_error = none) \wedge (attitude\_error = none) \wedge \\ &\quad separation\_done\big) \rightarrow \\ &\quad \mathbf{X}\big((mode = standby)\,\mathbf{U}\,(mode = safe)\big)\big) \end{aligned} \tag{8}$$

and omits the possibility of aborting the transition. As seen from the table, using P4' results in a false negative model checking result.

## 5 Conclusions

The AOCS system is a typical instance of a mission-critical system with various mode transitions triggered by inputs from a highly non-deterministic environment, including recovering from components faults. We have described how a symbolic model checker input language can be used to model an implementation of the AOCS system given its implementation in Ada source code. One of the key methods employed has been abstraction which can control the state space of

the model to a reasonable size. We have described techniques and abstractions used to alleviate the state explosion problem due to handling of timers and the large number of system components controlled by AOCS. The LTL properties used to check the model are generated based on mode transition diagrams of the system and the generation procedure is automated from a tabular notation.

Even after the abstractions, the AOCS system is currently too large to be fully automatically verified using symbolic model checking methods. Instead of getting rid of the close one-to-one correspondence of the Ada source code and the corresponding NuSMV 2 model, and thus making the model state space more manageable, we have instead resorted to incomplete model checking methods, bounded model checking in particular. This leaves us with challenging problems for the symbolic model checker development work that we are also concurrently doing in other projects.

In the experiments, the BDD-based LTL model checking can deliver results to about half of all the checked properties while for the other properties the approach times out. The incremental BMC algorithm is mainly used for bug hunting but it does also show the non-existence of short counterexamples to the remaining properties. A timed out BDD-based LTL model checking run, however, does not provide any additional information of the property. Out of curiosity, we also ran the complete BMC algorithm, which tries to also prove properties correct, not only look for counterexamples. Similar to our previous experiences, the complete BMC algorithm is not able to prove properties correct for models of significant size.

We were also able to collect some comments about the model checking results from Space Systems Finland. In general, they think that using the BDD-based LTL model checking to prove 12 out of 28 properties in the experiment is acceptable from the industrial perspective. On the other hand, since BMC always goes beyond 100 steps in the experiment, it would increase their confidence if there would be a better explanation on what this means in the context of the system. That is, the BMC part might be understood better if the BMC experimental results could be quantified in some systematic way. For example, one could be looking at structural coverage of the model in 100 time steps or looking at scenarios that can or cannot happen in this bound. This opens up interesting topics for further study.

There are many additional topics for further work. Firstly, the modeling from Ada source code to NuSMV 2 models is currently manual work. If this part was automated, a lot of optimizations that are easy to do automatically would become available. The Ada code is sequential, and as only the system mode changes are observed from the outside, many of the internal states could potentially be automatically removed by an optimizing “model compiler”. However, doing such optimizations manually is currently too time consuming and risky (it is easy to make modeling errors when “hand optimizing the model”).

On the BMC side, the parallel and distributed BMC engine of [WNH09] could be used to go deeper into the system state space by exploiting multiple multi-core computers running on a single BMC instance in parallel. While the properties we check are not strictly safety properties (some of the counterexamples could be infinite loops where e.g., the mode does not change at all), they still have some counterexamples that can be represented by finite paths (e.g., the mode entered next is a wrong one). For these latter “safety” counterexamples, the approach of [LHJ10] can be used, which is tailored to more efficiently finding the safety counterexamples of PSL (superset of LTL) properties using BDD-based engines.

**Acknowledgements:** We would like to thank Space Systems Finland for providing us with all the materials needed to complete this case study. This project has been financially supported by the RECOMP project funded by ARTEMIS-JU, Tekes - Finnish Funding Agency for Technology and Innovation, Conformiq Software, Space Systems Finland, and Academy of Finland (projects 128050 and 139402).

## Bibliography

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In Cleaveland (ed.), *TACAS*. LNCS 1579, pp. 193–207. Springer, 1999.

[BCG+09] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, R. Sebastiani. Software model checking via large-block encoding. In *FMCAD'2009*. Pp. 25–32. 2009.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang. Symbolic Model Checking: $10^{20}$ States and Beyond. *Inf. Comput.* 98(2):142–170, 1992.

[BDL+06] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, M. Hendriks. UPPAAL 4.0. In *QEST*. Pp. 125–126. IEEE Computer Society, 2006.

[BFV+09a] K. Björkman, J. Frits, J. Valkonen, J. Lahtinen, K. Heljanko, I. Niemelä, J. J. Hämäläinen. Verification of Safety Logic Designs by Model Checking. In *Proceedings of the Sixth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies NPIC&HMIT 2009*. Knoxville, Tennessee, April 2009.

[BFV+09b] K. Björkmann, J. Frits, J. Valkonen, K. Heljanko, I. Niemelä. Model-Based Analysis of a Stepwise Shutdown Logic. VTT Working Papers 115, VTT Technical Research Centre of Finland, Espoo, 2009.
http://www.vtt.fi/inf/pdf/workingpapers/2009/W115.pdf

[BHJ+06] A. Biere, K. Heljanko, T. Junttila, T. Latvala, V. Schuppan. Linear Encodings of Bounded LTL Model Checking. *Logical Methods in Computer Science* 2(5:5), 2006.

[BK08] C. Baier, J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[CBKK94] P. J. Clarke, D. Babich, T. M. King, B. M. G. Kibria. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems* 16:1512–1542, 1994.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *CAV'2002*. LNCS 2404, pp. 359–364. Springer, 2002.

[CGP99] E. M. Clarke, O. Grumberg, D. A. Peled. *Model Checking*. The MIT Press, 1999.

[ES03] N. Eén, N. Sörensson. An Extensible SAT-solver. In Giunchiglia and Tacchella (eds.), *SAT'2003*. LNCS 2919, pp. 502–518. Springer, 2003.

[HJL05] K. Heljanko, T. Junttila, T. Latvala. Incremental and Complete Bounded Model Checking for Full PLTL. In Etessami and Rajamani (eds.), *CAV'2005*. Lecture Notes in Computer Science 3576, pp. 98–111. Springer, 2005.

[ILT10] A. Iliasov, L. Laibinis, E. Troubitsyna. An Event-B Model of the Attitude and Orbit Control System. http://deploy-eprints.ecs.soton.ac.uk/, 2010.

[ITL+10a] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, T. Latvala. Developing Mode-Rich Satellite Software by Refinement in Event-B. In Kowalewski and Roveri (eds.), *FMICS*. LNCS 6371, pp. 50–66. Springer, 2010.

[ITL+10b] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, P. Väisänen, D. Ilic, T. Latvala. Verifying Mode Consistency for On-Board Satellite Software. In Schoitsch (ed.), *SAFECOMP*. LNCS 6351, pp. 126–141. Springer, 2010.

[LHJ10] T. Launiainen, K. Heljanko, T. Junttila. Efficient Model Checking of PSL Safety Properties. In *Proceedings of the 10th International Conference on Application of Concurrency to System Design (ACSD'2010 )*. Pp. 95–104. Braga, Portugal, June 2010.

[RAB+95] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, C. Pixley. Efficient BDD Algorithms for FSM Synthesis and Verification. In *IEEE/ACM Proceedings International Workshop on Logic Synthesis, Lake Tahoe (NV)*. 1995.

[Val96] A. Valmari. The State Explosion Problem. In Reisig and Rozenberg (eds.), *Petri Nets*. Lecture Notes in Computer Science 1491, pp. 429–528. Springer, 1996.

[Var10] K. Varpaaniemi. DEPLOY Work Package 3 Attitude and Orbit Control System Software Requirements Document (DEP-RP-SSF-R-005, issue 1.0). http://deploy-eprints.ecs.soton.ac.uk/, 2010. Documentation. DEPLOY project. Space Systems Finland Ltd.

[VPB+08] J. Valkonen, V. Petterson, K. Björkman, J.-E. Holmberg, M. Koskimies, K. Heljanko, I. Niemelä. Model-Based Analysis of an Arc Protection and an Emergency Cooling System – MODSAFE 2007 Working Report. VTT Working Papers 93, VTT Technical Research Centre of Finland, Espoo, Finland, Feb. 2008.
http://www.vtt.fi/inf/pdf/workingpapers/2008/W93.pdf

[WNH09] S. Wieringa, M. Niemenmaa, K. Heljanko. Tarmo: A framework for Parallelized Bounded Model Checking. In Brim and Pol (eds.), *Proceedings of the 8th International Workshop on Parallel and Distributed Methods in Verification (PDMC'09 )*. Electronic Proceedings in Theoretical Computer Science (EPTCS) 14, pp. 62–76. 2009.
http://dx.doi.org/10.4204/EPTCS.14.5