Supporting Automated Verification of Reconfigurable Systems with Product Lines and Model Checking

Faiz Ul Muram[1][a], Samina Kanwal[2][b] and Muhammad Atif Javed[3][c]

[1]*School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden*

[2]*National University of Sciences and Technology, Islamabad, Pakistan*

[3]*RISE, Research Institutes of Sweden, Västerås, Sweden*

Keywords: Reconfigurable Systems, Product Lines, Model Transformations, Model Checking, Formal Methods, LTL.

Abstract: The capability to dynamically reconfigure in response to change of mode or function, failures, or unanticipated hazardous conditions is fundamental for many critical systems. The modelling and verification of such systems are frequently carried out with product lines and model checking, respectively. At first, the objectives and related requirements of reconfigurable systems are mapped to a feature model, whereas the units related to operational modes are selected in individual configurations. After that, the proposed approach performs automated transformation of particular models into formal constraints and descriptions for leveraging the analytical powers of model checking techniques; the formal verification of completeness, consistency and conflict is carried out with NuSMV model checker. Finally, in circumstances when the counterexample is produced, its analysis is performed for the identification of corresponding problems and their resolutions. The applicability of the proposed approach is demonstrated through case study of attitude and orbit control system.

1 INTRODUCTION

The principal intention of reconfigurable systems is to enhance the responsiveness in consequence to mode or function changes, environmental conditions and unexpected failures (Muram et al., 2020). The reconfigurable systems possess the advantages of both dedicated product lines and of flexible systems. In particular, they focus on customized flexibility that can be supported through product lines. The product lines have been used for the identification and systematization of commonalities and variabilities to concurrently engineer a set of scenarios; the achievement of a single configuration is based on the selection and composition of commonalities and variabilities (Javed and Gallina, 2018; Javed et al., 2019). The integration between product line and model checking is essential for the formal verification of reconfigurable system behaviour against its formal requirements that are often expressed in temporal logic (Gan et al., 2014).

There exist several efforts to apply the model checking technique for product line verification, for example, (Gruler et al., 2008; Lauenroth et al., 2009; Lochau et al., 2016). However, previous approaches have not considered the automated transformations of product lines. These approaches often assume that formal descriptions and constraints of the product line can be easily created. Unfortunately, this makes the approaches hard to apply in practice because creating formal descriptions and constraints requires the underlying knowledge of formal techniques (Muram et al., 2016; Muram et al., 2017; Muram et al., 2019). Moreover, this task is often accomplished in a laborious and manual manner, which is also error-prone (Gruler et al., 2008; Lauenroth et al., 2009). In case of violations the results produced by model checkers (e.g., counterexamples) are rather cryptic and verbose (Muram et al., 2015; Muram et al., 2019). As a consequence, they are difficult to interpret and understand for those who have limited knowledge of the underlying formal techniques.

This paper focuses on the automated transformations and formal verification of reconfigurable systems. In particular, the objectives and related requirements of reconfigurable systems are mapped to a feature model, which in turns automatically transformed into the formal constraints i.e., Linear Temporal Logic (LTL) (Pnueli, 1977); whereas the units related to operational modes selected in individual configurations are transformed into state based Symbolic Model Verifier (SMV) (Cimatti et al., 1999) language. These transformations are achieved by us-

[a] https://orcid.org/0000-0001-6613-4149
[b] https://orcid.org/0000-0002-1079-9712
[c] https://orcid.org/0000-0003-3781-4756

Muram, F., Kanwal, S. and Javed, M.
Supporting Automated Verification of Reconfigurable Systems with Product Lines and Model Checking.
DOI: 10.5220/0010455702970305
In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2021)*, pages 297-305
ISBN: 978-989-758-508-1

ing Eclipse Xtend. Specifically, the mapping rules are defined to transform the feature model and configurations into formal constraints and descriptions. This way, our approach helps to alleviate the burden of manually encoding formal constraints and descriptions, and therefore, increase productivity and avoid potential translation errors. The formal verification of completeness, consistency and conflict is carried out with the NuSMV[1] 2.6.0, a new version of symbolic model checker. Completeness considers whether a set of configurations satisfy the system objectives and related requirements. Consistency checking examines the correctness of configurations against the structure and sequence of the system. Conflict verifies whether the configurations are compatible with the system constraints or not. In circumstances when the counterexample is produced by the model checker, its analysis is performed for the identification of corresponding problems and their resolutions. The applicability and technical feasibility of the proposed approach is demonstrated through case study of Attitude and Orbit Control System (AOCS).

The rest of this paper is organized as follows: Section 2 describes the tool-supported method for the formal verification of reconfigurable systems. Section 3 demonstrates the effectiveness of proposed approach through AOCS product line scenario. Section 4 discusses related work. Section 5 concludes the paper and presents future research directions.

2 APPROACH

The modern variable systems are often composed of several distinct sensors, controllers and actuators that can operate in both isolation and conjunction. For such systems, the capability to adjust, in particular, the reconfiguration in consequences to mode or function changes, environmental conditions and system failures is perceived as fundamental. Product lines provide the means to handle (re)configurations of systems. To be able to build a product line, the objectives and related requirements are mapped to a feature model; however, for (re)configurations, the scenario information or operational modes could be used (Muram et al., 2020). The principal focus of our approach is to formally verify the completeness, consistency and conflict in reconfigurable systems. The violations can be produced when the configurations are incomplete with respect to the system objective and related requirements, inconsistent with the organization and sequence of the system (i.e., decomposition hierarchy and groups), and incompatible with the system (cross-tree) constraints. Counterexample analysis detects the violation cause(s) and provide measures to make the configurations valid. An overview of our approach is shown in Figure 1. In the following sections, we describe each step of our approach.

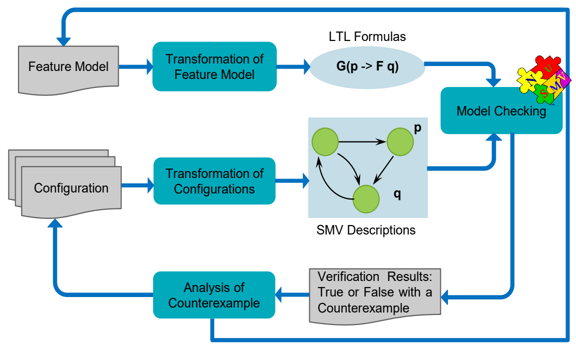

Feature Model
Transformation of Feature Model
LTL Formulas
G(p -> F q)
Model Checking
Configuration
Transformation of Configurations
p
q
SMV Descriptions
Analysis of Counterexample
Verification Results: True or False with a Counterexample

Figure 1: An Overview of the Proposed Approach.

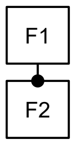
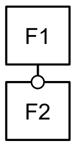
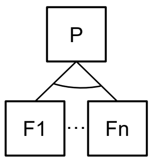
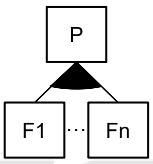
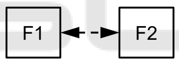
2.1 Generating LTL Constraints

In this subsection, we present our algorithmic solution for the generation of formal constraints from a feature model. The relationships between features (e.g., decomposition hierarchy, groups and cross-tree constraints) need to be represented in an appropriate formalism so that the selection or deselection of features will become the formal constraints for the configuration(s). In this context, LTL is used for describing the temporal relationships or properties among the features (Pnueli, 1977). LTL extends the classical propositional logic, for instance, negation ($\neg\varphi$), implication ($\rightarrow$), conjunction ($\wedge$), and disjunction ($\wedge$) with several future operators, such as **F**φ (“**F**inally/Sometime”), **G**φ (“**G**lobally/Always”) and φ_1 **U** φ_2 (“**U**ntil”). To make the formulation of some formulas significantly more concise and intuitive, we have also used the LTL past operators, to reason about previous states and transitions, such as **H**φ (“**H**istorically”) and **O**φ (“**O**nce”).

The construction of LTL formulas (constraints) for formal verification of reconfigurable systems is a highly knowledge intensive endeavour. Therefore, the mapping rules are defined to formally represent the semantics of a feature model elements. By using these mapping rules, an input feature model is automatically transformed to the corresponding LTL formulas. Table 1 summarises the modelling elements of a feature model along with their definitions and mapping rules that constitutes the relationships between features, such as optional and mandatory features, feature groups, and cross-tree constraints. The mapping is achieved by using Xtend

[1]http://nusmv.fbk.eu/

Table 1: Mapping Rules for a Feature Model.

Graphical Symbols	Relationships Definition	Mapping Rules
	**Mandatory**: If parent feature F1 is selected in a product, then child feature F2 must be selected. This indicates the commonality.	`G (F1 -> F F2)& H (F2 -> O F1)`
	**Optional**: If parent feature F1 is selected in a product, then child feature F2 may be selected or not. This expresses the variability. In particular, F2 will be selected if F1 and isSelected condition are true.	`G ((F1 & isSelected -> F F2)| (F1 -> F ! F2))`
	**Alternative (Xor)**: The semantic represents the case in which only one feature of the children needs to be selected when its parent feature P is part of the product.	`(P -> F ((F1 & ! ... & ! Fn)| (! F1 & !... & Fn)))`
	**Or**: If parent feature P is selected in a product, then at least one of the children must be selected. The traversal of features depend on the evaluation of isSelected conditions.	`G ((P & isSelected_1)-> F F1) |...| G ((P & isSelected_n)-> F Fn)`
	**Excludes**: Both features cannot be part of the same product. If F1 is selected then F2 cannot not selected, and vice versa.	`! (F1 & F2)`, or more complex, `(i)F (F1)-> (! F1 U isSelected)` `(ii)F (F2)-> (! F2 U !isSelected)`

language, which is tightly integrated with Java but has more concise syntax than Java. All the aspects of a domain specific language implemented in Xtext can be implemented in Xtend, since it is easier to use and allows to write better readable code. Algorithm 1 shows the skeleton of transformation models. Similar to our previous research (Muram et al., 2016; Muram et al., 2017; Muram et al., 2019), the breadth-first search algorithm is extended with three helper functions: `get_parent_features()`, `get_child_features()`, and `generate_ltl_formulas()`. The function `get_parent_features()` returns a set of parent features. Given a certain feature f, its child-features (or siblings) can be obtained by using the `get_child_features()` function. A feature r is called a child feature (or sibling) of f if there is an edge (relation) from f to r, so that a set of child-features (or siblings) of f can be obtained by following all of its edges.

The `generate_ltl_formulas(`f`)` function is not realised as a single function but rather a polymorphism of multiple functions. That is, depending on the type of relationships between features i.e., decomposition hierarchy and cross-tree constraints, a particular function for generating LTL formulas for that relationship will be invoked. The functions can have the same name but require different input types. This can be obtained in traditional programming languages by using the "`if/then/else`" or "`switch/case`" construct. The generation of Mandatory feature is presented in Algorithm 1 (Line 21–25). For code generation the string templates are surrounded by the pair of a triple single quotes (`'''`). However, the terminals for interpolated expression, called guillemets (« and ») are used to represent the parametrised placeholders that will be bound to and substituted with the actual values extracted from the input model elements by the Xtend engine. In NuSMV, LTL formulas are introduced by the LTLSPEC keyword.

2.2 Generating SMV Descriptions

This subsection presents the automated transformation of a set of configurations into formal SMV model (descriptions). The basic purpose of the SMV language is to describe the transition relation of a fi-

Algorithm 1: Translate Models.

```
 1: Input: Feature Model/Configurations
 2: Output: LTL Formulas/SMV Descriptions
 3: procedure TRANSFORM
 4:     $S_n \leftarrow \varnothing$          ▷ queue of non-visited features
 5:     $S_v \leftarrow \varnothing$              ▷ queue of visited features
 6:     $S_n \leftarrow S_n \cup$ get_parent_features($P$)
 7:     for all $f \in S_n$ do
 8:         $S_v \leftarrow S_v \cup \{f\}$
 9:         $S_n \leftarrow S_n \setminus \{f\}$
10:         $F_{features} \leftarrow$ get_child_features($f$)
11:                         ▷ use generate_smv_model
12:         generate_ltl_formulas($f$)
13:         for all $r \in F_{features}$ do
14:             if ($r \notin S_v$) then
15:                 $S_n \leftarrow S_n \cup \{r\}$
16:             end if
17:         end for
18:     end for
19:     extract <feature> id, name, relati-
20:     ons and generate ltl formulas:
21:     for all $relations.isTypeOf(Mandatory)$ do
22:    '''
23:   LTLSPEC G(«$f$» -> F «$r$») & H(«$r$» -> O «$f$»)
24:    '''
25:     end for
26: end procedure
```

nite Kripke structure. The encoding of configurations in terms of SMV descriptions provide the infrastructure to facilitate the verification of completeness, consistency and conflicts with a feature model, and especially, analysing the verification results to provide useful feedback for aiding the manager/developers in resolving violations. Similar to the generation of LTL formulas, the mapping of configurations into SMV descriptions is obtained by using Xtend framework. As described in Section 2.1, three helper functions are created. First two functions are same, however, the generate_smv_model(f) function is responsible for generating SMV descriptions for a set of configurations. We illustrate the skeleton of the function generate_smv_model(f) in Algorithm 2.

The SMV model consists of several modules, which are used to encapsulate variables deceleration and assignments. The identifier after the keyword MODULE is the name associated with the module. An instance of a module is created using the VAR declaration. In particular, the features are specified by boolean state variables in the section VAR and their corresponding transition relations are described in the section ASSIGN by a combination of two functions given in NuSMV (Cimatti et al., 1999). These variables are initialised with initial values as shown in the init(identifier) clause, whereas the function next(identifier) defines the next state of the variable based on current states. It is often combined with the branching structure "case/esac" for selecting one of many possible choices. The state is initially set to False. However, if the branch condition(s) are satisfied, it is changed to a True state. The branch condition(s) can be a guard expression and/or finishing of the preceding state. The feature's state shall be switched back to False after finishing the execution. Similar to generate_ltl_formulas(f), the function generate_smv_model(f) is not realised as a single function but rather a polymorphism of multiple functions.

Algorithm 2: Generating SMV descriptions.

```
 1: procedure GENERATE(smv_model)
 2:     extract <feature> id, name, relati-
 3:     ons and generate SMV descriptions:
 4:     '''
 5:     MODULE «main»
 6:     VAR
 7:         «f» : boolean; ▷ Variable declaration
 8:     ASSIGN
 9:         init(«f») := «feature-initial-state»
10:                 ▷ Definitions of state transitions
11:         next(«f») := case
12:             «condition(s)» : TRUE;
13:             «f» : FALSE;
14:         esac;
15:     '''
16: end procedure
```

A consistent and complete set of configurations is obtained by selecting a complete group of features in a manner that follows a correct feature hierarchy. Moreover, the semantics of the configurations should not be contradicted with constraints (requires, excludes) of the system (i.e., feature model). It is assumed that two features cannot have the same name. This assumption is rather realistic because the same name can imply ambiguity problems during (re)configuration of systems. While selecting the configurations, if a parent feature is selected (or deselected) then all mandatory child-features linked with a particular parent feature will be selected (or deselected). The ASSIGN section defines the transition relation of features. The child-feature is initially set to false. However, if the condition(s) are satisfied, it is changed to a true state (see Line 15), as shown in Figure 2. The feature's state shall be switched back to false after the execution.

The Xor-groups (Alternatives) are transformed as a set of configurations which show that exactly one

```
MODULE main
VAR
   «ParentFeature» : boolean;
   «ChildFeature» : boolean;
ASSIGN
   init(«ParentFeature») := TRUE;
   next(«ParentFeature») := case
         «ParentFeature» : FALSE;
         TRUE : «ParentFeature»;
      esac;
   init(«ChildFeature») := FALSE;
   next(«ChildFeature») := case
         «ParentFeature» : TRUE;
         «ChildFeature» : FALSE;
         TRUE : «ChildFeature»;
      esac;
```

Figure 2: Generic rules for SMV description.

child-feature will be selected in each configuration. In reality, the developers/corresponding manager(s) are responsible for the exclusiveness of conditions, i.e., selection of the child-feature based on a specific configuration. Figure 3 shows the rules for mapping an `Alternative` relationship into SMV descriptions. For the purpose of verification, it is possible to initialise a constraint, namely isSelected with the boolean values that is evaluated to true. However, in cases when the different types of variables may have dependencies among constraints, temporary variables of enumerated types can be introduced to handle them. Accordingly, a temporary variable named `tem_alternative_k` is defined, where k is an incrementally generated number. This variable is used as a program counter to determine which procedure, function or feature state is being executed at present. The variable `tem_alternative_k` has an enumerated type that includes a normal state "`undetermined`" and the values corresponding to the configurations or child-features (Line 7). It might be noted that the non-deterministic assignment is a powerful means provided by the NuSMV model checker for exhaustively exploring multiple possible execution paths yielded by the values of an enumerated state. The evaluation of the conditions is made using a "`case/esac`" construct (Line 16–19). The next state of `tem_alternative_k` will be either *selected*_1, *selected*_2 or *selected*_3 (Line 22, 27).

An `Or` relationship represents that any number of features can be selected in a configuration instead of one and only one. We initialised a constraint, namely isSelected with the boolean values that is evaluated to true when the corresponding element is selected. We use the logical *AND* operator ("&") to represent all incoming conditions, as shown in Figure 4. The cardinality <min..max> might be applied, where min denotes a lower bound limit and max denotes upper

```
1  VAR
2     «ParentFeature»: boolean;
3     «ChildFeature1» : boolean;
4     ...
5     «ChildFeature3» : boolean;
6     -- temporary variable
7     «tem_alternative_k» : {undetermined, «
          selected_1»,..., «selected_3»};
8  ASSIGN
9     -- if the parent feature is not a root
          feature, then FALSE must be used.
10    init(«ParentFeature») := TRUE;
11    next(«ParentFeature») := case
12          «ParentFeature» : FALSE;
13       esac;
14    ... -- the initialisations of
          conditions are omitted
15    init(«tem_alternative_k») :=
          undetermined;
16    next(«tem_alternative_k») := case
17          «ParentFeature» & («
              tem_alternative_k =
              undetermined») : {«selected_1
              »,..., «selected_3»};
18          TRUE : undetermined;
19       esac;
20    init(«ChildFeature1») := FALSE;
21    next(«ChildFeature1») := case
22          «tem_alternative_k» = «selected_1
              » : TRUE;
23          «ChildFeature1» : FALSE;
24       esac;
25    init(«ChildFeature3») := FALSE;
26    next(«ChildFeature3») := case
27          «tem_alternative_k» = «selected_3
              » : TRUE;
28          «ChildFeature3» : FALSE;
29       esac;
```

Figure 3: Mapping rules for Alternatives.

bound limit within a group of features the number of features that can be part of a configuration.

2.3 Formal Verification and Dealing with Violations

This subsection discusses the identification and resolution of violations (i.e., incompleteness, inconsistencies and conflicts) in the reconfigurable systems. The formal verification is achieved by using the NuSMV model checker that takes the generated SMV model (descriptions) and LTL formulas, and exhaustively explore the deviations of an LTL formula by traversing complete state space. In case an LTL formula does not satisfy the SMV model, NuSMV will generate a counterexample that consists of the execution traces of the SMV descriptions leading to the deviation. The incompleteness may occur due to the absence of units

```
VAR
   «ParentFeature»: boolean;
   «ChildFeature1» : boolean;
   «ChildFeature2» : boolean;
           -- temporary variables
   «isSelected_1» : boolean;
   «isSelected_2» : boolean;
   ASSIGN
      ... -- the initialisation of
          condition is omitted
      init(«ChildFeature1») := FALSE;
      next(«ChildFeature1») := case
         «ParentFeature» & «isSelected_1»
             : TRUE;
         «ChildFeature1» : FALSE;
      esac;
      ...
      init(«ChildFeature2») := FALSE;
      next(«ChildFeature2») := case
         «ParentFeature» & isSelected_2» :
               TRUE;
         «ChildFeature2» : FALSE;
      esac;
```

Figure 4: Mapping rules for Or.

or features in configurations with respect to system objectives and requirements. The inconsistency of configurations may happen when a hierarchical or sequential order is not followed, such as inaccurate relation or groups between features/units, or misplacement of features. Conflict occurs when a cross-tree constraint is violated, for instance, an optional feature is excluded, however, due to the requires constraint, it has to be included in some configurations. The analysis of counterexample not only detects the typical possible causes of incompleteness, inconsistency, and conflict but also provides the recommendations to resolve the specific deviation.

To locate the causes of deviations, the verification result (i.e., output trace file) is scrutinised and parsed to identify the false LTL formulas. The extracted formulas and SMV descriptions together with mapping rules for a feature model are traversed to find out why the particular configuration(s) deviate from the specification defined in a feature model. This is performed in three steps. First, the absence of units or missing obligatory feature cause (either one or multiple features/units could be missing) is detected and the corrective measures, i.e., add the missing obligatory features/units is suggested. Second, the hierarchical dependency and sequence between features are scrutinised from the SMV descriptions, for example, multiplicity causing the deviations of the LTL formulas are located and appropriate recommendations, i.e., remove or replace the features or units are presented. Finally, the constraints between features are scrutinised, for instance, requires or excludes causing the deviation of the LTL formulas are located and appropriate recommendations, i.e., insert or delete features are provided. Based on the deviation causes and given recommendations, the systems can be reconfigured and transformed again into to their formal descriptions, and then will be re-verified until no more deviations are detected.

3 CASE STUDY

In this section, we briefly describe a variable system that belongs to a space domain, in particular, the AOCS (Javed et al., 2019). It consists of sensors, actuators, software, and ground support equipment to control the satellite's attitude and orbit. In the ECSS-E-ST-60-30C standard, the AOCS requirements, operational objectives, tailoring rules and customer specifications are presented. The standard objectives and requirements for AOCS are mapped into a feature model, whereas tailoring rules, customer specifications, units related to AOCS operational modes are selected in individual configurations (Javed et al., 2019). In this paper, we focus on four different configurations of AOCS, which are: (i) Sun Sensors to THrusters (SSTH); (ii) Sun Sensors to Reaction Wheels (SSRW); (iii) Star Tracker to THrusters (STTH); and (iv) Star Tracker to Reaction Wheels (STRW). Figure 5 shows the feature model of AOCS that presents the different allowed combinations of sensors, actuators, as well as the associated functional software modules.

- **Sensing Principles.** In SSTH and SSRW configurations, Sun Sensors are used as the primary sensor in combination with a Magnetometer to provide full attitude determination capability. In STTH and STRW configurations, a Star Tracker is used as a primary attitude sensor. In all configurations, the attitude is estimated using a kinematic (Kalman) filter-based estimation for which Gyro measurements are an exogenous input.
- **Actuation Principles.** In SSTH and STTH configurations, the spacecraft attitude is modified using a Thruster actuator, whereas in SSRW and STRW configurations, the spacecraft attitude is primarily controlled using Reaction Wheels.
- **Software Functional Modules.** They are classified into different functional groups, such as Sensor Processing, Guidance, Estimation, Control and Command Distribution.
- **Operational Modes.** The AOCS consists of several modes, for instance, in case of major anomaly

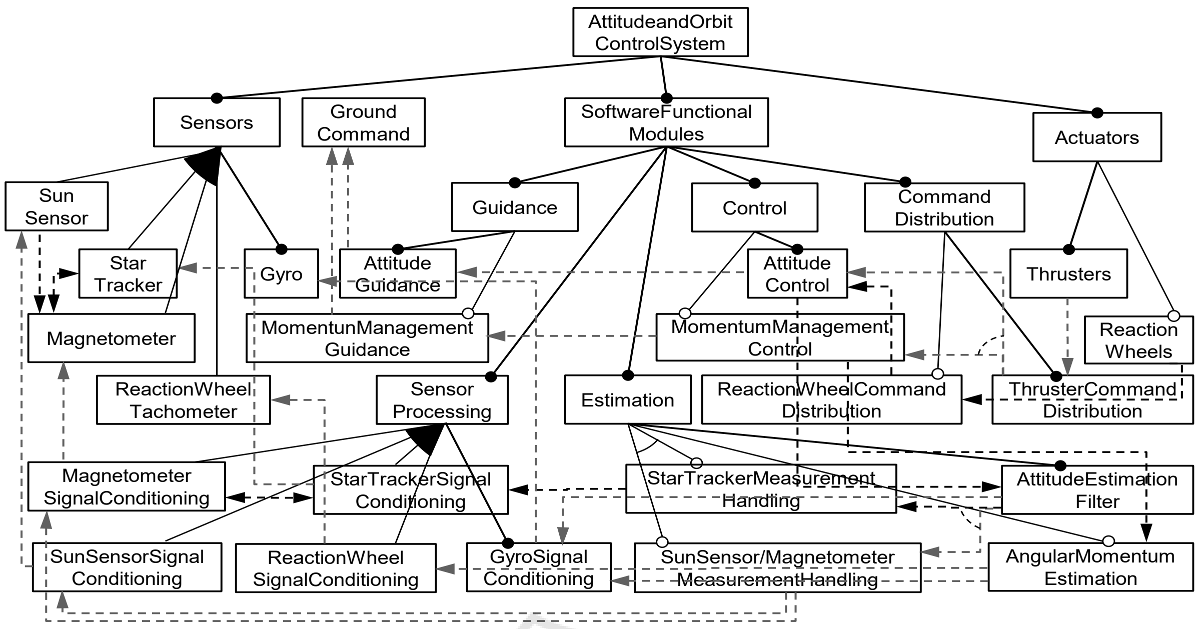

AttitudeandOrbit ControlSystem
Sensors
Ground Command
SoftwareFunctional Modules
Actuators
Sun Sensor
Star Tracker
Gyro
Attitude Guidance
Guidance
Control
Command Distribution
Attitude Control
Thrusters
Magnetometer
MomentumManagement Guidance
MomentumManagement Control
Reaction Wheels
ReactionWheel Tachometer
Sensor Processing
Estimation
ReactionWheelCommand Distribution
ThrusterCommand Distribution
Magnetometer SignalConditioning
StarTrackerSignal Conditioning
StarTrackerMeasurement Handling
AttitudeEstimation Filter
SunSensorSignal Conditioning
ReactionWheel SignalConditioning
GyroSignal Conditioning
SunSensor/Magnetometer MeasurementHandling
AngularMomentum Estimation

Figure 5: Feature Model of AOCS.

a Safe Mode is automatically initiated to reach and control safe pointing attitude and angular rates. A Normal Mode concerns the mission level performances of spacecraft during the operational phase. Typically, SSTH variant would be used for a Safe Mode. It might be used in combination with SSRW for long term safety and variant STRW for a Normal Mode.

To demonstrate the qualification of AOCS configurations, the formal verification is performed by using the NuSMV model checker. In particular, the AOCS feature model and configurations are automatically transformed into LTL formulas and SMV descriptions by applying mapping rules described in Section 2.1 and Section 2.2, respectively. The generated SMV descriptions and LTL formulas are combined into one input file and executed by the NuSMV model checker version 2.6.0. The evaluation is conducted on a regular computer equipped with a 2.90 GHz i7 processor and sixteen gigabytes of memory running Windows 10. The approach is implemented using Java and automated transformations have been realised using Eclipse Xtend[2]. Figure 6 shows the verification results including the list of satisfied and unsatisfied LTL formulas. The NuSMV model checker generates a counterexample demonstrating a sequence of permissible state executions leading to a state in which the deviation occurs in LTL formula. By looking at the results, we found that four LTL formulas are false. In order to identify the deviations

```
$ NuSMV AOCS.smv
-- specification (G (Sensors -> F Gyro) & H (
     Gyro -> O Sensors)) is false
-- as demonstrated by the following execution
     sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    AttitudeandOrbitControlSystem = TRUE
    Sensors = FALSE
  ...
-- specification (G (
     AttitudeandOrbitControlSystem -> F (
     Sensors & Actuators & SoftwareFunctional
      Modules)) & H ((Sensors & Actuators &
     SoftwareFunctionalModules) -> O
     AttitudeandOrbitControlSystem)) is false
  ...
-- specification (G ((GroundCommand &
     Guidance) -> F AttitudeGuidance) & H (
     AttitudeGuidance -> O (GroundCommand &
     Guidance))) is false
  ...
-- specification ((G ((Sensors & isSelected_2
     ) -> F Magnetometer) | G (((Magnetometer
      & isSelected_9) & Sensors) -> F
     SunSensor)) | G ((Sensors & isSelected_4
     ) -> F ReactionWheelTachometer)) is true
-- specification (F StarTracker -> (!
     StarTracker U (!isSelected_2 & Sensors))
     ) is false
    ...
```

Figure 6: NuSMV Verification Results.

[2]See http://www.eclipse.org/xtend

causes, the counterexample analysis is performed (as depicted in Section 2.3). In this case, the specification of AOCS system is deviated due to (i) incomplete configurations, i.e., missing `GroundCommand` feature, which is required by `AttitudeGuidance`; (ii) inconsistency in hierarchical dependency and sequence: (a) `Gyro` feature can only be selected after its parent feature `Sensors` is selected, and (b) when parent feature `AttitudeandOrbitControlSystem` is selected, then mandatory child feature `Sensors` must be selected, subsequently `Gyro` would be selected; and (iii) conflict: an optional feature `StarTracker` is added, but due to the excludes constraint, it has to be excluded in SSRW configuration. These deviations can be resolved by adding `GroundCommand` feature, removing `StarTracker` feature in the SSRW configuration and correcting the sequence of `AttitudeandOrbitControlSystem`, `Sensors` and `Gyro`, respectively. NuSMV took three seconds to verify AOCS system, which is quite reasonable. Without the counterexample analysis, users would have to study and investigate the syntax and semantics of the trace file in order to determine the relationship between the execution traces and models, and then locate the corresponding violations manually.

4 RELATED WORK

The work presented in (Gruler et al., 2008) extended the process algebra Calculus of Communicating Systems (CCS) by introducing a variant operator that allows to model an alternative choice between two processes. A multivalued modal μ-calculus is used for specifying the properties of individual configurations of a PL-CCS program. However, the verification using a model checker is not performed that is prerequisite for practical evaluation. For incremental verification of software product lines, a delta-oriented extension of CSS (DeltaCCS) is described in (Lochau et al., 2016). A core product (process) is first verified against a given property by using Maude model checker, then different variants are derived from the core process by applying CCS Delta sets. Lauenroth et al. (Lauenroth et al., 2009) presented a model-checking approach to verify every permissible product specified with I/O-automata against provided Computational Tree Logic (CTL) properties. The authors just used three operators (EX, EU and EG) to define one property of each type.

Classen et al. (Classen et al., 2011) focused on the feature-oriented extension of SMV language called fSMV that is equivalent to high-level representation of finite transition systems. An algorithm on top of the symbolic model-checking is presented to check whether a set of valid products in the feature diagram (fSMV) satisfies one or more feature CTL properties. Shi et al. (Shi et al., 2014) presented the Bilattice-based Feature Transition Systems (BFTSs) for modelling partial (i.e., incomplete) software product line designs, while the Action CTL formulas are used to express system behavioural properties. To carry out the model checking with exiting χChek engine, the translations from BFTS to multi-valued Kripke structure and from ACTL to CTL formulas are presented. Beek et al. (ter Beek et al., 2016) presented QFLan specifications to model the quantitative properties such as price and weight of a product line, and a prototypical Maude interpreter integrated with Z3 and MultiVeStA model checker. Previous studies on the model checking of product lines have not considered the automated transformations of feature model and configurations into LTL constraints and formal descriptions. In addition, the counterexample analysis for identifying the causes of deviations and their resolutions has not been considered.

Gan et al. (Gan et al., 2014) used the NuSMV model checker to verify the design of an AOCS. They manually translated the AOCS implementation i.e., Ada source code abstractions to SMV language. The required system behaviour that is extended state machine diagrams with prioritized transitions are translated into LTL properties. However, they have not considered the product lines. The automatic transformations of sequence diagrams (Muram et al., 2016), activity diagrams (Muram et al., 2019) and service choreographies (Muram et al., 2015; Muram et al., 2017) are investigated for the containment checking problem. In this paper, we performed the automated transformations and formal verification of reconfigurable systems modelled in a product line.

5 CONCLUSIONS

This paper targets the integration of product line and model checking for the formal verification of reconfigurable system behaviour against its formal requirements. To achieve this, the objectives and related requirements of reconfigurable system are mapped to a feature model, whereas the units related to operational modes are selected in individual configurations. To leverage the NuSMV model checker for the formal verification, the automated transformations of these models into LTL constraints and formal descriptions are performed. Subsequently, the analysis of model checker results is performed for the identification and resolution of deviations (i.e., incompleteness, inconsistency and conflict), in order to satisfy the configuration(s) with a feature model. The applicability of the proposed approach is demonstrated through AOCS family/product line. It is noteworthy that the proposed approach is generally applicable to the broad range of scenarios and domains.

In the future, we plan to perform automatic extraction of feature and configuration models from document files. We also plan to consider additional scenarios and application, such as electric quarry site (Javed et al., 2020). Another direction for future work is to attach the evidence obtained from runtime model checking with the assurance cases that are constructed to provide comprehensive, logical and defensible justification of the safety and security of a reconfigurable production system (Muram et al., 2020).

ACKNOWLEDGEMENTS

This work is partially supported by FiC (Future factories in the Cloud) project funded by SSF (Swedish Foundation for Strategic Research). The third author has also participated during the tenure of an ERCIM “Alain Bensoussan” Fellowship Programme.

REFERENCES

Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. (1999). NuSMV: A new symbolic model verifier. In *11th Int'l Conf. on Computer Aided Verification (CAV)*, pages 495–499.

Classen, A., Heymans, P., Schobbens, P., and Legay, A. (2011). Symbolic model checking of software product lines. In *33rd International Conference on Software Engineering, ICSE '11, Waikiki, Honolulu, HI, USA*, pages 321–330.

Gan, X., Dubrovin, J., and Heljanko, K. (2014). A symbolic model checking approach to verifying satellite onboard software. *Sci. Comput. Program.*, 82:44–55.

Gruler, A., Leucker, M., and Scheidemann, K. D. (2008). Modeling and model checking software product lines. In *10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS '08, Oslo, Norway*, volume 5051, pages 113–131.

Javed, M. A. and Gallina, B. (2018). Safety-oriented process line engineering via seamless integration between EPF composer and BVR tool. In *22nd International Systems and Software Product Line Conference - Volume 2, SPLC '18, Gothenburg, Sweden*, pages 23–28.

Javed, M. A., Gallina, B., and Carlsson, A. (2019). Towards variant management and change impact analysis in safety-oriented process-product lines. In *34th ACM/SIGAPP Symposium on Applied Computing, SAC '19, Limassol, Cyprus*, pages 2372–2375.

Javed, M. A., Muram, F. U., Fattouh, A., and Punnekkat, S. (2020). Enforcing geofences for managing automated transportation risks in production sites. In *Dependable Computing - EDCC Workshops, DREAMS '20, Munich, Germany*, volume 1279, pages 113–126.

Lauenroth, K., Pohl, K., and Toehning, S. (2009). Model checking of domain artifacts in product line engineering. In *24th IEEE/ACM International Conference on Automated Software Engineering, ASE '09, Auckland, New Zealand*, pages 269–280.

Lochau, M., Mennicke, S., Baller, H., and Ribbeck, L. (2016). Incremental model checking of delta-oriented software product lines. *J. Log. Algebraic Methods Program.*, 85(1):245–267.

Muram, F. U., Javed, M. A., Hansson, H., and Punnekkat, S. (2020). Dynamic reconfiguration of safety-critical production systems. In *25th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC '20, Perth, Australia*, pages 120–129.

Muram, F. U., Javed, M. A., Tran, H., and Zdun, U. (2017). Towards a framework for detecting containment violations in service choreography. In *2017 IEEE International Conference on Services Computing, SCC' 17, Honolulu, HI, USA*, pages 172–179.

Muram, F. U., Tran, H., and Zdun, U. (2015). Counterexample analysis for supporting containment checking of business process models. In *13th International Business Process Management Workshops, BPM '15, Innsbruck, Austria*, volume 256, pages 515–528.

Muram, F. U., Tran, H., and Zdun, U. (2016). A model checking based approach for containment checking of UML sequence diagrams. In *23rd Asia-Pacific Software Engineering Conference, APSEC '16, Hamilton, New Zealand*, pages 73–80.

Muram, F. U., Tran, H., and Zdun, U. (2019). Supporting automated containment checking of software behavioural models using model transformations and model checking. *Sci. Comput. Program.*, 174:38–71.

Pnueli, A. (1977). The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA*, SFCS '77, pages 46–57.

Shi, Y., Wei, O., and Zhou, Y. (2014). Model checking partial software product line designs. In *International Workshop on Innovative Software Development Methodologies and Practices, InnoSWDev '14, Hong Kong, China*, pages 21–29.

ter Beek, M. H., Legay, A., Lluch-Lafuente, A., and Vandin, A. (2016). Statistical model checking for product lines. In *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA '16, Imperial, Corfu, Greece*, volume 9952, pages 114–133.